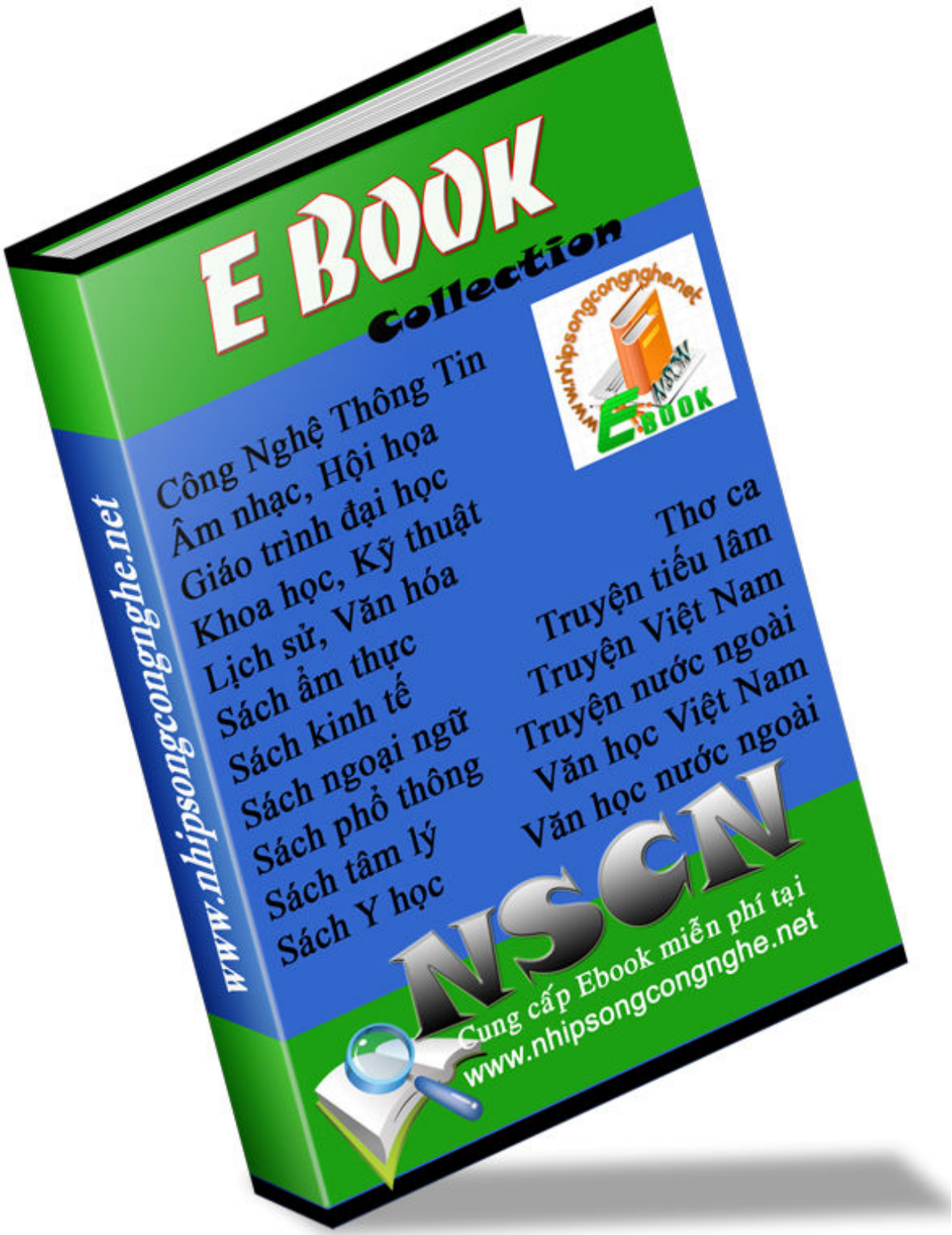

E BOOK
Collection
Công Nghệ Thông Tin
Âm nhạc, Hội họa
Giáo trình đại học
Khoa học, Kỹ thuật
Lịch sử, Văn hóa
Sách ẩm thực
Sách kinh tế
Sách ngoại ngữ
Sách phổ thông
Sách tâm lý
Sách Y học
Thơ ca
Truyện tiếu lâm
Truyện Việt Nam
Truyện nước ngoài
Văn học Việt Nam
Văn học nước ngoài
www.nhipsongcongnghe.net
NSCN
Cung cấp Ebook miễn phí tại
www.nhipsongcongnghe.net

Phần 1: Lí thuyết HĐH Unix/Linux

Mục lục

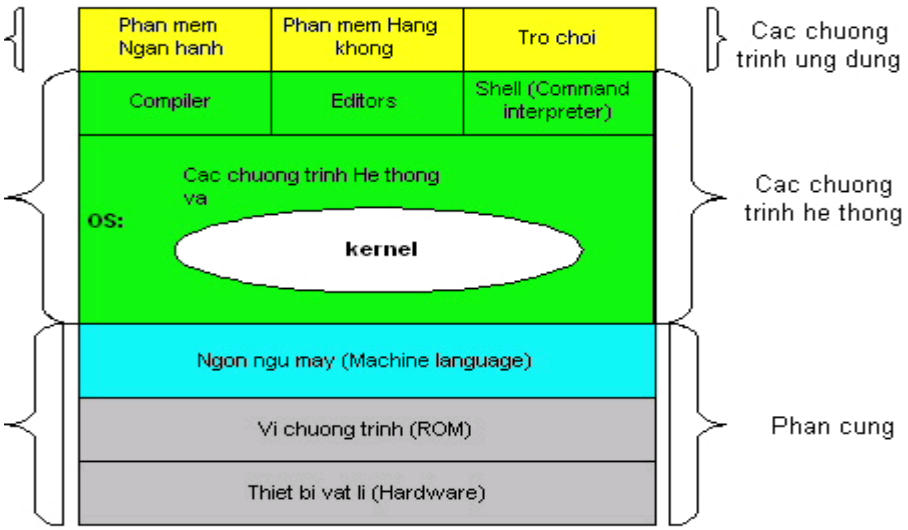
Phần 2: Lập trình trong Unix

Phần 3: Lập trình mạng trong Unix

I. Tổng quan về Hệ Điều Hành

*(An **Operating System** is a group of programs that provide basic functionality on a computer. This functionality is called services. Other word an Operating System can be seen as a set of functionality building blocks upon which other programs depend. It also manages computer resources and resolves resource conflicts, so OS abstracts the real hardware of the system and presents the system's users and its applications with a virtual machine).*

1. Phần mềm máy tính chia ra làm hai loại: các *phần mềm hệ thống*, quản lí hoạt động của bản thân máy tính, và các *chương trình ứng dụng*, giải quyết các yêu cầu của người dùng. Phần căn bản nhất của tất cả các phần mềm hệ thống, gọi là *Hệ điều hành*, mà chức năng cơ bản là kiểm soát tất cả nguồn tài nguyên, cung cấp nền tảng (các hàm chức năng, các dịch vụ hệ thống) để trên đó các chương trình ứng dụng được viết ra sẽ sử dụng. Mô hình một máy tính như sau:



Hình trên cho ta một phần gọi là *kernel*, hay *nhân của HĐH*, kernel hỗ trợ HĐH thực hiện ***chức năng quản lí*** các thành phần sau đây:

1.Thiết bị (devices), cho một giao tiếp để các chương trình người dùng " nói chuyện" với thiết bị;

2.Bộ nhớ (memory), cấp bộ nhớ cho các chương trình (tiến trình) đang chạy;

3.Các Tiến trình (process), tạo, giám sát hoạt động của các tiến trình;

4.Liên lạc (communication) giữa các TT.

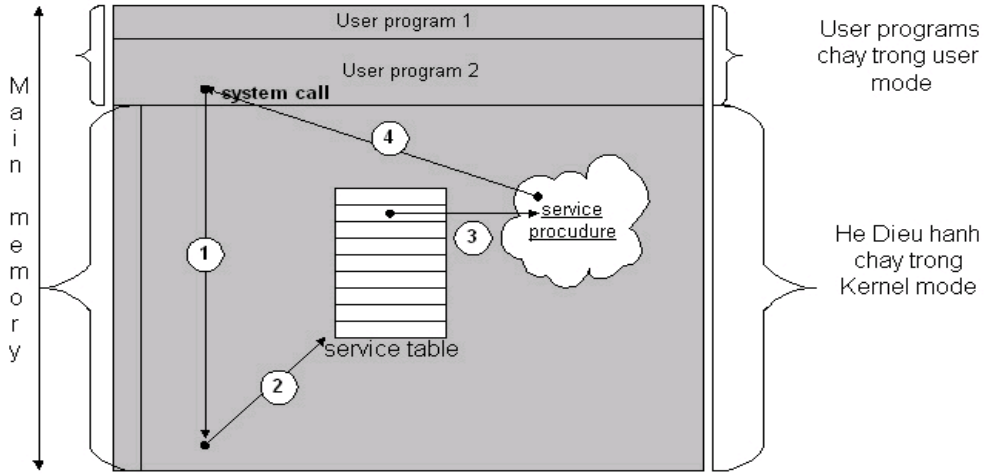
Nguồn tài nguyên máy tính có nhiều, như (CPU(s), bộ nhớ, các thiết bị ngoại vi ghép nối vào máy tính…) tạo thành một hệ thống rất phức tạp. Viết các chương trình để theo dõi tất cả các thành phần, khai thác chúng chính xác và để chúng chạy độc lập một cách tối ưu, là việc rất khó. Và nếu điều này lại để cho từng người dùng quan tâm, thì sẽ có vô số các

chương trình được viết và nếu hệ là loại nhiều người dùng thì, hãy thử tưởng tượng … Như vậy rỏ ràng cần tách người dùng ra khỏi sự phức tạp của phần cứng. Cách có thể đảm bảo là đặt phần mềm (hay lớp phần mềm) lên trên đỉnh của phần cứng và nó quản lí tất cả các phần của máy tính, trong khi trao cho người dùng một giao diện (interface) hay một máy tính ảo (***virtual machine***) dễ hiểu hơn và dễ lập trình ứng dụng hơn. Lớp phần mềm đó gọi là HĐH. Từ đây xuất hiện một quan niệm mới, đó là sự phân biệt chế độ chạy máy, nó bao gồm:

HĐH chạy trong một môi trường đặc biệt, gọi là chế độ nhân (***kernel mode*** hay ***supervisor mode***). Chế độ này được hỗ trợ bởi kiến trúc của CPU (bởi các lệnh máy đặc biệt) và nó ngăn người dùng truy nhập vào phần cứng (quản lí phần cứng chuẩn xác cho nhiều người dùng đồng thời, còn gọi là chế độ được bảo vệ (***protected mode***)).

Thuật ngữ *kernel* đề cập đến phần mã cốt yếu nhất của các chương trình hệ thống, nó kiểm soát các tệp, khởi động và cho chạy các chương trình ứng dụng đồng thời, phân chia thời gian sử dụng CPU cho các chương trình, cấp bộ nhớ cũng như các tài nguyên khác cho các chương trình của người dùng. Bản thân kernel không làm gì nhiều nhưng cung cấp các công cụ nguyên thuỷ (*primitive functions*) mà các tiện ích khác, các dịch vụ khác của HĐH được xây dựng. Do đó các chương trình hệ thống, các trình ứng dụng sử dụng các dịch vụ của HĐH, chạy trong ***user mode***. Tuy nhiên có sự khác biệt là các trình ứng dụng thì tận dụng những tiện ích hệ thống cho, còn các trình hệ thống là cần thiết để máy tính chạy được. Các trình ứng dụng chạy trong chế độ người dùng (***user mode***), các *primitive functions* chạy trong kernel . Việc kết nối giữa hai chế độ chạy trình được thực hiện bởi *gọi hệ thống* (***system call***).

Gọi hệ thống (hay *gọi các dịch vụ của hệ thống*, GHT), là một giao diện lập trình giữa HĐH và ứng dụng. Nó được thực hiện bằng cách đặt các thông số vào những chổ được định nghĩa rỏ ràng (vào các thanh ghi của CPU hay đặt vào stack) và sau đó thực hiện một lệnh bẩy đặt biệt (**trap intruction**) của CPU. Lệnh này chuyển chế độ chạy máy từ *user mode* vào *kernel mode* và từ đó điều khiển chuyển cho HĐH (1). Tiếp theo HĐH kiểm tra *số hiệu* và các *thông số của GHT* để xác định GHT nào sẽ thực hiện (2). Từ trong bảng với chỉ số (số hiệu của GHT), HĐH lấy ra con trỏ trỏ đến qui trình (*procedure*) thực hiện GHT đó (3). Khi thực hiện xong GHT, điều khiển chuyển trả lại cho chương trình của người dùng.

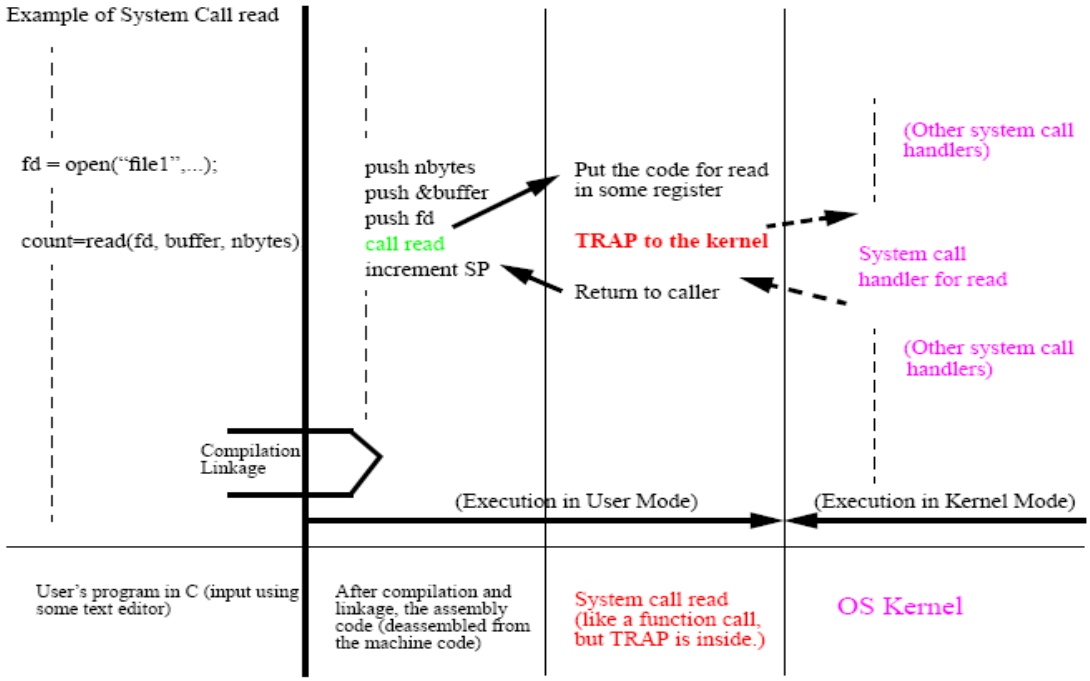


Từ đây có thể thấy cấu trúc cơ bản của GHT như sau:

1. Một chương trình chính kích hoạt dịch vụ hệ thống bằng một GHT.

2. Bẩy (TRAP) chuyển GHT vào nhân HĐH, nhân xác định số hiệu của dịch vụ.
3. Thực hiện dịch vụ.
4. Kết thúc dịch vụ và trở về nơi phát sinh GHT.

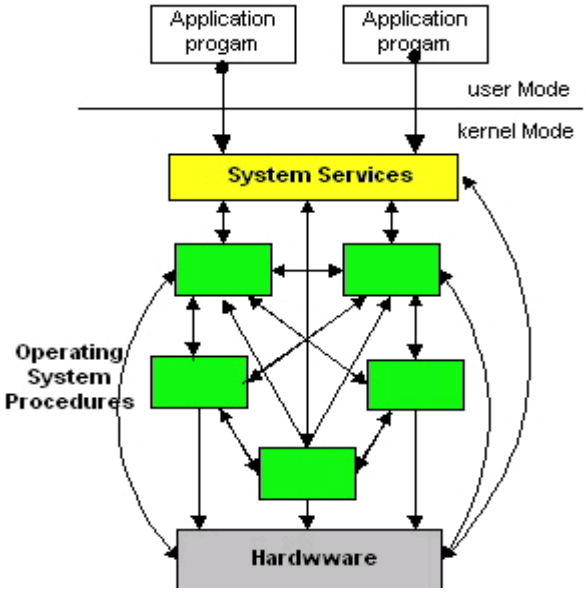
Hình sau cho các bước theo trình tự từ lập trình đến thực thi GHT *read()*:



Khi nhìn cách thực thi một chương trình, phần mã chương trình người dùng được kết hợp với mã của kernel (khi thực hiện các *primitive functions* qua GHT), tạo ra toàn bộ mã chương trình. Nói cách khác vào thời điểm chạy trình, phần mã của kernel thực hiện bởi GHT là mã của chương trình người dùng, chỉ khác ở chế độ thực hiện.

2. Trên cơ sở định nghĩa *kernel mode* và *user mode*, ***kiến trúc của các HĐH*** có thể khác nhau:

***a. Loại đơn thể (monolitic OS)*:**

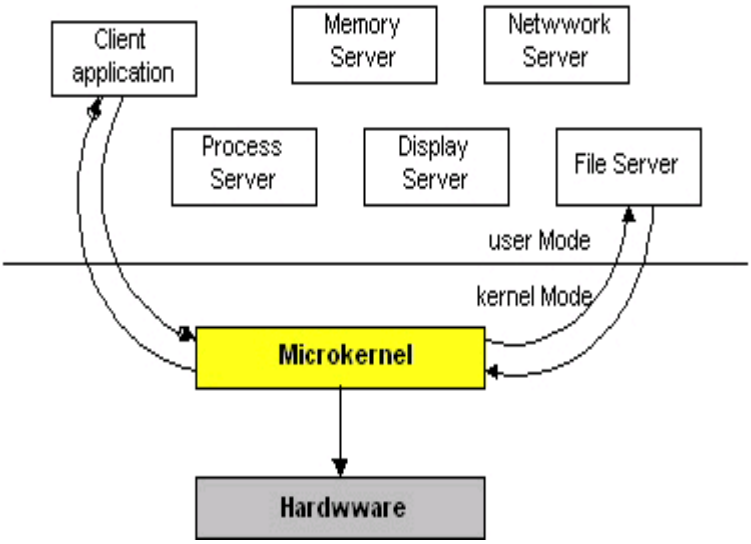


HĐH kiểu đơn thể (monolitic OS)

Các trình ứng dụng chạy ở user mode khi thực hiện gọi một dịch vụ của Hệ thống, HĐH sẽ chuyển việc thực hiện dịch vụ vào kernel mode. Khi dịch vụ hoàn tất HĐH chuyển việc thực hiện chương trình đã phát sinh gọi dịch vụ trở lại user mode, chương trình này tiếp tục chạy. PC DOS là một ví dụ. Đặc điểm chung của loại này là kernel là một thực thể đơn, một chương trình rất lớn, mà các thành phần chức năng truy nhập tới tất cả các cấu trúc dữ liệu và thủ tục của hệ thống.

b. Mô hình Client/Server:

Chia OS ra thành nhiều tiến trình (TT), mỗi TT cung cấp một tập các dịch vụ (ví dụ các dịch vụ bộ nhớ, dịch vụ tạo TT, dịch vụ lập biểu …). Các phần mềm dịch vụ (**server**) chạy trong user mode thực hiện vòng lặp để tiếp nhận yêu cầu các dịch vụ của nó từ các **client.** Client có thể là thành phần khác của HĐH, hay là một ứng dụng, yêu cầu phục vụ bằng cách gởi một thông điệp (message) tới server. Kernel của HĐH, là phần rất nhỏ gọn (**microkernel**) chạy trong kernel mode phát các thông điệp tới server, server thực hiện yêu cầu, kernel trả lại kết quả cho client. Server chạy các TT trong user mode tách biệt, nên nếu có sự cố (fail) thì toàn bộ hệ thống không hề bị ảnh hưởng. Với nhiều CPU, hay nhiều máy kết hợp, các dịch vụ chạy trên các CPU, máy khác nhau, thích hợp cho các tính toán phân tán.



c. Loại cấu trúc theo lớp (layered OS):

HĐH được chia thành các lớp xếp chồng lên nhau. Phân lớp là cấu trúc được sắp xếp theo hai hướng lên-xuống (nhìn tại một lớp bất kì), sao cho mỗi lớp thành *một đơn thể* với chức năng cung cấp các dịch vụ cho *lớp trên liền kề*, và sử dụng tòan bộ dịch vụ của *lớp dưới liền kề*, với nguyên tắc lớp trên yêu cầu và nhận kết quả, lớp dưới thực hiện và trao kết quả cho lớp trên. Với cách xác định tường minh như vậy sẽ tránh được sự trùng lặp chức năng cũng như chồng chéo quan hệ (ví dụ ở mô hình đơn thể nói trên) giữa các đơn thể. Kiểu cấu trúc này mang lại các ưu điểm sau:

- Nếu chuẩn hóa được các dịch vụ ở mỗi lớp, và chuẩn định dạng dữ liệu vào/ra thì các phần mềm thực hiện đơn thể sẽ trở nên phổ quát, dễ dùng chung cho các hệ thống có cấu trúc tương tự. Chương trình nguồn dễ dàng biên dịch lại và chạy ở các phần cứng khác nhau. Đó là tính *portable*.

- Đơn giản hóa quá trình cải tiến hay nâng cấp hệ thống bằng việc thay đổi, nâng cấp các đơn thể các thể, mà không phải chờ đợi cho đến khi hòan tất toàn bộ hệ thống. Chính nhờ vậy mà tăng được hiệu năng hoạt động, tính ổn định của hệ thống.

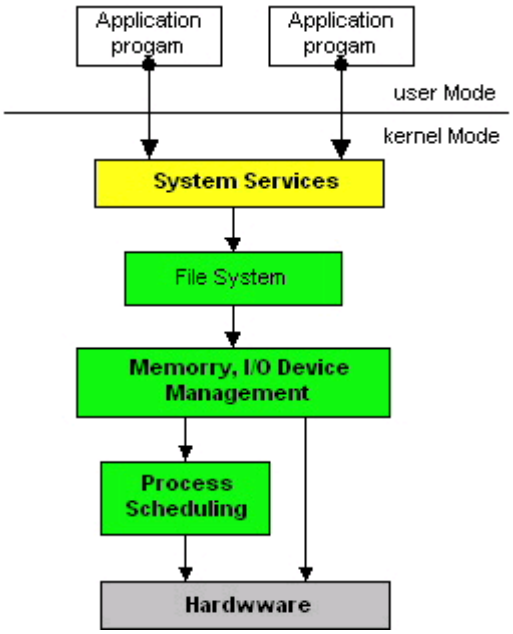
- Hổ trợ tốt cho nhu cầu trao đổi dữ liệu giữa các hệ thống khác nhau khi có sự chuẩn hóa về giao diện (interface), và giao thức (protocol). Đó chính là tính mở của hệ thống.

Các HĐH kiểu UNIX (VAX/VMS hay Multics (tiền thân của UNIX), …) thuộc loại này.

Hãy quan sát mô tả điển hình của cấu trúc phân lớp theo hình sau:

Layer						
7	System call handler					
6	File system 1	...	File system m			
5	Virtual memory					
4	Driver 1	Driver 2	...			Driver n
3	Threads, thread scheduling, thread synchronization					
2	Interrupt handling, context switching, MMU					
1	Hide the low-level hardware					

Trên mô hình này, lớp 1, 2 gần với từng loại kiến trúc máy tính (hardware), trong đó lớp 1 cố gắng dấu đi các kiến trúc phần cứng có thể, tạo ra một lớp phần cứng trừu tượng (Hardware Abstract Layer). Lớp 2 là các thao tác (handling) cơ bản áp dụng trên các phần cứng bên dưới (bao gồm xử lí ngắt, chuyển bối cảnh thực hiện của các tiến trình, quản lí bộ nhớ). Lớp 3 thực thi phân phối thời gian chạy máy (scheduling), đồng bộ tiến trình và luồng. Lớp 4 là các đặc tả thiết bị có trên máy dạng tổng quát, không phụ thuộc vào loại thiết bị cụ thể, ví dụ ở UNIX tại gồm các tệp thiết bị tại thư mục /dev. Lớp 5 và 6 là cách tổ chức tài nguyên mà kernel sẽ thực hiện các biện pháp quản lí (tổ chức chức tệp (File System, Virtual File System), bộ nhớ ảo). Lớp 7 là giao diện của HĐH với trình ứng dụng. Có thể thấy, lớp 3 đến 7 là các lớp tổng quát, không phụ thuộc vào phần cứng. Như vậy mã thực thi có thể triển khai trên bất kì loại kiến trúc máy nào. Mô hình dưới cho thấy một ví dụ của ý tưởng trên:



Unix là một ví dụ điển hình với các đặc điểm như sau:

1. Hệ được viết bằng ngôn ngữ bậc cao, làm cho dễ đọc, dễ hiểu, dễ thay đổi và chạy trên các nền phần cứng khác nhau.
2. Có giao diện người dùng đơn giản, mang lại sức mạnh cung cấp các dịch vụ người dùng yêu cầu.
3. Cung cấp các hàm cơ bản (*primitive*) để phát triển các chương trình phức tạp từ các chương trình đơn giản.
4. Sử dụng hệ thống tệp có cấu trúc, dễ dùng, dễ bảo trì và hiệu quả.
5. Tệp được tổ chức theo kiểu dòng các byte, nhất quán, dễ tạo các ứng dụng.
6. Giao tiếp các thiết bị ngoại vi đơn giản, nhất quán và ổn định.
7. Là hệ nhiều người dùng, nhiều tiến trình, mỗi người dùng có thể chạy nhiều tiến trình đồng thời. Hệ cón là hệ đa xử lí.
8. Người phát triển ứng dụng không cần biết tới cấu trúc máy tính, do đó ứng dụng viết ra có thể chạy trên nhiều phần cứng khác nhau.

Đơn giản, nhất quán, đó là tư tưởng chủ đạo của Unix.

II. Unix/Linux

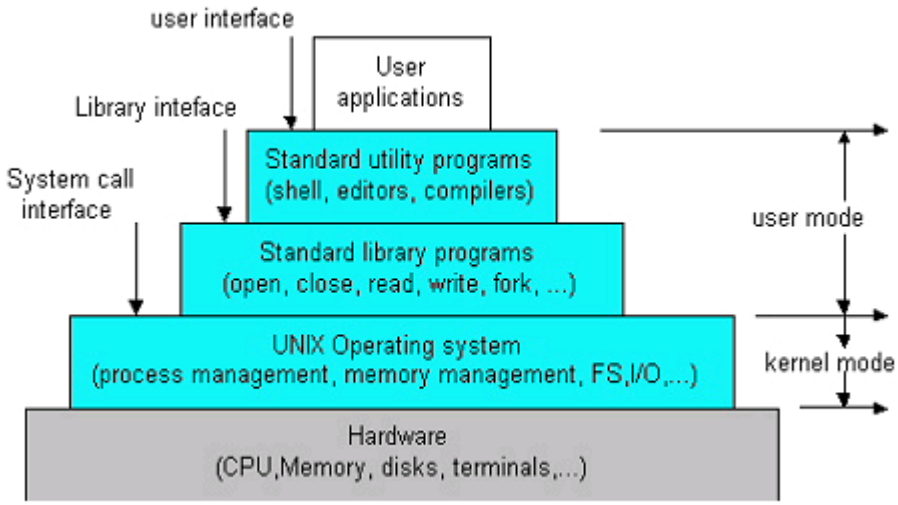
Chương I. Tổng quan hệ thống Unix

1. Cấu trúc hệ thống

Cấu trúc của Unix

Unix có thể xem như một loại kim tự tháp với các lớp chức năng xếp chồng lên nhau và tạo ra các giao diện. Phần cứng (hardware) sẽ đề cập sau. Hệ Điều Hành (HĐH, hay Operating System-OS) tương tác trực tiếp với phần cứng, cung cấp các dịch vụ cơ bản cho các chương trình và ngăn cách các chương trình với phần cứng cụ thể. Nếu nhìn hệ thống như từ các lớp, thì OS thông thường được gọi là ***nhân hệ thống (System Kernel),*** nó được cách li với chương trình của người dùng. Bởi vì các chương trình ứng dụng nói chung, kể cả OS, độc lập với phần cứng, nên dễ dàng chạy trên các phần cứng khác nhau vì không phụ thuộc vào

phần cứng cụ thể. Chẳng hạn *Shell* và các *editors (vi, ed)* ở lớp ngoài tương tác với kernel bằng cách phát sinh ra Gọi Hệ Thống (GHT) – s*ystem calls*. GHT sẽ chỉ thị cho kernel làm những việc khác nhau mà chương trình gọi yêu cầu, thực hiện trao đổi dữ liệu (data) giữa kernel và chương trình đó. Một vài chương trình có tên trong hình là các chương trình chuẩn trong cấu hình của hệ thống và được biết tên dưới dạng các lệnh – c*ommands*. Lớp này cũng có thể bao hàm cả các chương trình của người dùng với tên là *a.out*, một loại tên chuẩn cho các tệp chạy được do bộ dịch C tạo ra. Còn có loại ứng dụng khác (APPS) được xây dựng trên lớp trên cùng của các chương trình có mức thấp hơn hiện diện ở lớp ngoài cùng của mô hình. Mặc dù mô hình mô tả hai cấp các APPS, nhưng người dùng có thể mở rộng ra các cấp thích hợp. Rất nhiều các hệ thống ứng dụng, các chương trình, cho cách nhìn ở mức cao, song tất cả đều dùng các dịch vụ cấp thấp được cung cấp bởi *kernel* qua GHT. Trong System V chuẩn có 64 GHT, trong đó có 32 GHT là thường dùng nhất (LINUX 2.x có nhiều hơn và khoản chừng 164 lệnh GHT).



Tập hợp các System calls và các thuật toán bên trong tạo thành ***thân (body) của kernel***, do vậy việc nghiên cứu Unix trong sách này sẽ giản lược để nghiên cứu chi tiết các system calls cũng như sự tương tác giữa chúng. Và khái niệm "*Unix system*", hay "*kernel*" hay "*system*" trong sách đều có ý nói tới *kernel của hệ điều hành Unix* và rõ ràng hơn ở từng bối cảnh trình bày.

2. Cách nhìn từ phía người dùng: tổ chức tệp

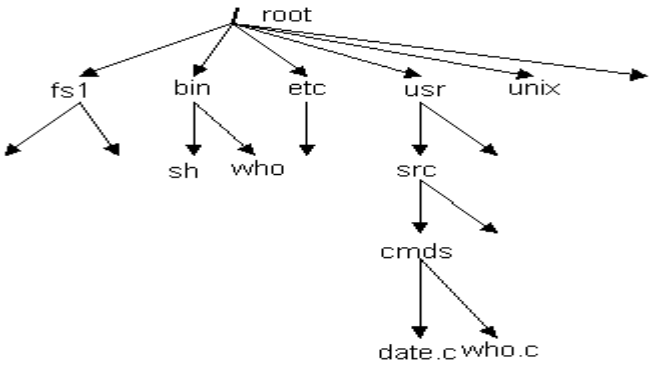
Phần này tóm lượt các nét đặc trưng ở mức cao của Unix chẳng hạn: hệ thống tệp (File system FS), môi trường xử lí, xây dựng các khối nguyên hàm, và sẽ được khai thác sau này.

2.1. Hệ thống tệp (File system – FS)

Hệ thống tệp của Unix được đặc tả bởi:

- Cấu trúc cấp bậc (cây thư mục);
- Cách xử lí nhất quán dữ liệu của tệp (chuỗi các byte *byte stream*);
- Khả năng tạo và hủy tệp (tạo mới, xóa);
- Tính tăng trưởng động của tệp (thêm bớt, cắt dán);

- Khả năng bảo vệ dữ liệu của tệp (bởi các kiểu thuộc tính như *quyền truy nhập*);
- Xử lí các thiết bị ngoại vi như xử lí các tệp (cách nhìn thiết bị bởi mô tả kiểu tệp).



FS được tổ chức như một *cây* bắt đầu từ một *nút đơn* gọi là *root*, được biểu diễn như sau: " / "; từ đó sẽ có các thư mục khác tạo thành nhánh của cây, trong các nhánh có thể có các nhánh (con) khác. Dưới các nhánh sẽ là *tệp*. Tệp có thể là *tệp bình thường (regural files*) hay cũng có thể là *tệp đặc biệt (special files)*. Tệp được truy nhập qua *đường dẫn (path name)* mô tả cách thức định vị được tệp trong FS. *Đường dẫn đầy đủ*, hay *đường dẫn tuyệt đối*, bắt đầu bởi dấu / và nó xác định sẽ tìm tệp bằng cách đi từ *root* qua cấu trúc cây thư mục theo các nhánh chỉ thị trong đường dẫn. Ví dụ trong hình ta có: ***/usr/src/cmd/date.c*** là đường dẫn tuyệt đối tới tệp date.c. Đường dẫn không bắt đầu từ root gọi là *đường dẫn tương đối*, chỉ tới thư mục hiện tại của tệp.

Các chương trình trong Unix không có hiểu biết gì về định dạng (format) bên trong của dữ liệu của tệp. Kernel lưu dữ liệu của tệp, xử lí dữ liệu tệp như một *dòng các bytes* (*byte stream*) không có định dạng. Do vậy cú pháp truy nhập dữ liệu trong tệp được định nghĩa bởi hệ thống và nhất quán như nhau cho tất cả các chương trình, nhưng ngữ nghĩa của dữ liệu thì chương trình ứng dụng phải xử lí.

Ví dụ: Chương trình ***troff*** xử lí văn bản có định dạng hoài vọng sẽ tìm thấy các kí tự "dòng mới" (" new line ") ở cuối mỗi dòng văn bản, còn chương trình kế toán ***acctcom*** hoài vọng tìm thấy những bản ghi có độ dài cố định. Cả hai chương trình dùng cùng các dịch vụ hệ thống để truy nhập dữ liệu trong tệp theo cách *byte stream*, nhưng bên trong mỗi chương trình lại dùng cách phân tích cú pháp khác nhau thích hợp cho nó. Nếu một chương trình phát hiện thấy định dạng là không đúng, thì bản thân chương trình sẽ thực hiện một hành vi khác để xử lí (chứ không phải hệ thống làm điều đó).

Thư mục cũng là một loại tệp, hệ thống xử lí dữ liệu trong thư mục cũng bằng *byte stream,* nhưng dữ liệu ở đây chứa tên các tệp trong thư mục có khuôn dạng dự đoán được, sao cho OS và các chương trình, ví dụ *ls*, có thể nhận ra các tệp trong thư mục.

Việc truy nhập tệp được kiểm soát bởi *quyền truy nhập (access permission)* kết hợp với tệp. Quyền truy nhập được lập ra một cách độc lập để kiểm soát truy nhập *đọc (read), ghi (write),* và *thực hiện (execute)* cho ba lớp người sử dụng: *người sở hữu tệp (**u** - user), nhóm người được truy nhập (**g** - group), những người khác (**o** - other).* Người dùng có thể tạo tệp nếu họ được phép và các tệp mới tạo sẽ là các *nhánh lá* của cấu trúc thư mục hệ thống.

Đối với người dùng, Unix xử lí các thiết bị như thể đó là các tệp. Các thiết bị được mô tả bởi các *tệp thiết bị đặc biệt* và nằm ở một nhánh trong cấu trúc hệ thống thư mục (**/dev**). Các chương trình truy nhập các thiết bị bằng cú pháp giống như đã dùng để truy nhập tệp bình thường, các thiết bị cũng được bảo vệ cùng phương thức như các tệp, qua việc ấn định quyền truy nhập. Bởi vì tên các thiết bị cũng giống như tên các tệp bình thường và các thao tác trên chúng là như nhau, nên hầu hết các chương trình đều không biết tới kiểu tệp bên trong của tệp mà chúng thao tác.

2.2 Môi trường xử lí

Một *chương trình* - ***program*** là một tệp thực thi và một *tiến trình (TT – **procces)*** là *một khoảnh khắc (instance)* của chương trình được thực hiện theo trục thời gian. TT bao gồm:

mã trình thực thi,

dữ liệu (data) của TT**,**

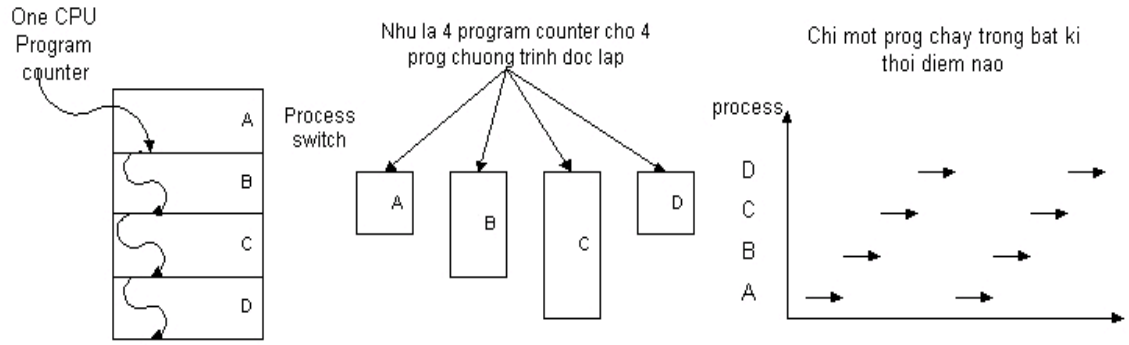
program (user) stack,

CPU program counter,

kernel stack,

CPU registers

và thông tin khác cần thiết để chạy trình. Các dữ liệu này tạo ra **bối cảnh (context) của TT**, mỗi TT có bối cảnh riêng biệt. Có rất nhiều TT được thực hiện đồng thời trên Unix (đặc tính này còn gọi là *đa trình - multiprogramming* hay *đa nhiệm - multitasking*) theo nguyên lí phân chia thời gian (**time sharing**), mà tổng số các TT về logic là không có giới hạn. Có nhiều GHT cho phép các TT tạo ra các TT mới, kết thúc các TT, đồng bộ các giai đoạn thực hiện TT, kiểm soát phản ứng với các sự kiện khác nhau. Các TT sử dụng GHT độc lập với nhau. Ví dụ chạy đa trình với 4 chương trình A, B, C, D trên một CPU:



Hãy xét ví dụ sau:

```
main (argc, argv)
        int argc;
        char *argv[];
{
        /* giả định có 2 đối đầu vào*/
```

```
        if (fork () == 0)
                execl ("copy", "copy", argv[1], argv[2], 0);
        wait((int *) 0);
        printf ("copy done\n");
}
```

Chương trình trên dùng GHT *fork()* để tạo ra một TT mới. TT mới gọi là *TT con* sẽ nhận được giá trị trả lại là 0 từ lệnh *fork* và nó kích hoạt *execl* để chạy trình copy. Lệnh *execl* sẽ phủ lên không gian địa chỉ của TT con bằng mã của trình "copy", với giả định trình "copy" nằm cùng trong thư mục hiện hành của *main*, và chạy trình *copy* với các thông số do người dùng đưa vào. Nếu *execl* hoàn tất nó sẽ không trở về địa chỉ xuất phát trong *main* vì nó chạy trong một miền địa chỉ mới khác. Trong khi đó TT bố đã kích hoạt *fork()* lại nhận được giá trị trả lại khác 0 từ GHT *wait()*, nó "treo" việc thực hiện để đợi cho đến khi "copy" kết thúc và in ra thông báo "copy done " và sau đó kết thúc thực hiện main bằng *exit* (*exit()* là ngầm định khi kết thúc *main* trong C).

Một cách tổng quát, GHT cho phép người dùng viết các chương trình thực hiện các thao tác rất tinh tế mà bản thân kernel không cần có nhiều chức năng hơn là cần thiết. Có thể đề cập tới một số các chức năng, chẳng hạn các bộ dịch (*compilers*), bộ soạn thảo (*editors*) thuộc lớp các chương trình cấp người dùng (user level) và quan trọng hàng đầu là *shell*, là *trình thông dịch* mà người dùng sử dụng ngay sau khi *log in* vào hệ thống: *shell* thông dịch các từ trong dòng lệnh thành tên lệnh máy, phát sinh TT con và TT con thực hiện lệnh đưa vào, xử lí các từ còn lại trong dòng lệnh như các thông số của lệnh.

Shell thực hiện ba kiểu lệnh:

1. Lệnh là tệp có thể thực hiện được chứa mã máy phát sinh do bộ dịch tạo ra từ mã nguồn (chương trình C chẳng hạn);
2. Lệnh là tệp chứa một xâu các dòng lệnh của *shell*;
3. Là các lệnh bên trong của *shell*. Các lệnh bên trong này làm cho *shell* trở thành một ngôn ngữ lập trình rất mạnh trong Unix.

Shell là chương trình thuộc lớp người dùng, không phải là phần của *kernel*, cho nên có thể dễ dàng biến cải cho mỗi môi trường đặc thù. Bản thân shell cũng có ba loại khác nhau thích hợp cho các nhu cầu sử dụng khác nhau và hệ thống có thể chạy các shell đó đồng thời.

Sức mạnh của mỗi kiểu shell thể hiện ở khả năng lập trình của mỗi kiểu.

Mỗi TT được thực hiện trong Unix có một môi trường (*execution environment*) thực hiện, bao gồm cả thư mục hiện hành. Thư mục hiện hành của TT là thư mục dùng để chỉ đường dẫn không bắt đầu bằng " /". Người dùng có thể thực hiện nhiều TT cùng một lúc, và các TT lại có thể tạo ra các TT khác một cách động, và đồng bộ việc thực hiện các TT đó. Đặc tính này tạo ra một môi trường thực hiện chương trình rất mạnh trong Unix.

2.3 Xây dựng các hàm chức năng cơ bản (primitives)

Như đã đề cập, tính triết lí của Unix là để cung cấp cho OS các nguyên hàm (primitives) mà người dùng sẽ sử dụng để viết các chương trình (chức năng) nhỏ, có tính modul, được dùng như các khối xây dựng để tạo ra các chương trình lớn và phức tạp. Một trong các primitive đó là khả năng tái định tuyến vào/ra (*redirect I/O*). Tiếp theo là *pipe,* một cơ chế linh hoạt cho phép truyền dữ liệu giữa các TT, hay lệnh ngay từ bàn phím. Ví dụ, khi dùng các chương

trình nhỏ để tạo các chương trình lớn và phức tạp, người lập trình sử dụng các primitives redirect I/O và pipe để hợp nhất các phần đó lại.

3. Các dịch vụ của Unix/Linux

- Trong hình mô tả các lớp của kernel, cho thấy lớp kernel nằm ngay bên dưới lớp các trình ứng dụng của người dùng. Kernel thực hiện vô số các thao tác cơ bản (primitives) thay mặt cho các TT của người dùng để hỗ trợ cho giao diện người dùng. Các thao tác đó bao hàm các *dịch vụ* mà kernel cấp:
- Kiểm soát việc thực hiện các TT gồm có: cho phép TT tạo TT mới, kết thúc TT, treo việc thực hiện và trao đổi thông điệp giữa các TT;
- Lập biểu để các TT được thục hiện trên CPU. Các TT chia xẻ CPU theo phương thức phân chia thời gian, một TT sẽ bị treo sau khi thời gian phân bổ đã hết, kernel lấy TT khác đưa vào thực hiện. Sau này kernel sẽ lại lựa chọn TT bị treo để đưa vào thực hiện trở lại.
- Cấp phát bộ nhớ cho TT đang thực hiện, cho phép TT chia sẻ không gian địa chỉ của TT dưới những điều kiện nhất định, bảo vệ miền địa chỉ riêng của TT đối với các TT khác. Nếu hệ thống chạy trong hoàn cảnh thiếu bộ nhớ, kernel sẽ giải phóng bộ nhớ bằng cách ghi lại các TT tạm thời vào bộ nhớ dự phòng (còn gọi là thiết bị swap). Nếu toàn bộ TT được ghi vào swap, thì hệ Unix gọi là *hệ tráo đổi* (*swapping system*); Nếu kernel ghi các trang của bộ nhớ lên swap, thì hệ đó gọi là *hệ lưu trang (paging system).*
- Cấp phát bộ nhớ thứ cấp để cất và tìm lại dữ liệu của người dùng có hiệu quả. Dịch vụ này cấu tạo nên hệ thống tệp. Kernel cấp vùng nhớ thứ cấp cho tệp của người dùng, khôi phục lại vùng nhớ, xây dựng cấu trúc tệp theo một cách thức hiểu được, bảo vệ tệp của người dùng trước các truy nhập bất hợp pháp.
- Cho phép các TT truy nhập các thiết bị ngoại vi, ví dụ t/b đầu cuối, đĩa, t/b mạng.
- Kernel cung cấp các dịch vụ một cách thông suốt, chẳng hạn kernel ghi nhận tệp cần thao tác thuộc loại tệp bình thường hay tệp thiết bị, nhưng ẩn điều đó đối với TT của người dùng; hay ví dụ, kernel tạo khuôn dữ liệu trong tệp để ghi (đĩa), nhưng lại ẩn khuôn dạng đó đối với TT người dùng (user). Tương tự như vậy đối với các dịch vụ hệ thống cung cấp cho các TT user dùng ở mức độ cấp người dùng. Ví dụ dịch vụ hệ thống mà shell dùng để đóng vai trò là trình thông dịch lệnh: cho phép shell đọc đầu vào từ t/b đầu cuối, phát sinh động các TT, đồng bộ việc thực hiện các TT, tạo pipe, đổi hướng I/O. Người dùng cấu tạo các phiên bản shell riêng mà không tác động tới những users khác. Các trình đó cùng dùng các dịch vụ của kernel ở mức shell chuẩn.

4. Phần cứng

Tiến trình người dùng (TT user) trên Unix được chia ra làm hai mức độ: Chế độ người dùng (***user mode)*** và chế độ nhân (***kernel mode)***. Khi TT thục hiện một GHT, chế độ thực hiện TT sẽ chuyển từ *user mode* sang *kernel mode*: OS thực hiện và cố gắng phục vụ các yêu cầu của user, trả lại kết quả và thông báo lỗi nếu có. OS lưu lại các hoạt động có liên quan tới TT user, thao tác các ngắt, lập biểu chạy TT, quản lí bộ nhớ... Có loại máy hỗ trợ nhiều mức hơn, tuy nhiên trong Unix hai mức này là đủ.

Sự khác biệt của hai mức này là:

- Các ứng dụng chạy trong chế độ xử lí không có đặc quyền, ***user mode,*** liên lạc với hệ thống qua một tập các giao tiếp giới hạn (kể cả một số lệnh của CPU), cũng như bị hạn chế truy nhập dữ liệu hệ thống. TT ứng dụng có thể truy nhập các lệnh và dữ liệu của nó, không được truy nhập lệnh và dữ liệu của kernel cũng như của các TT khác. Khi TT trong user mode thực hiện một GHT, kernel "bẩy" GHT đó, chuyển chế độ thực hiện vào kernel mode. Kernel kiểm soát TT, xác thực các đối (ví dụ quyền truy nhập, quyền thao tác dữ liệu) mà TT chuyển cho GHT và thực hiện GHT đó. Khi GHT kết thúc, Kernel chuyển TT ngược lại vào user mode trước khi trả điều khiển lại cho TT, cho phép TT tiếp tục chạy. Bằng cách đó kernel bảo vệ được chính nó cũng như các dữ liệu khỏi bị TT user làm tổn hại.
- Thực hiện mã của HĐH chạy trong chế độ đặc quyền của CPU, gọi là ***kernel mode***. Trong chế độ này HĐH chạy và thực hiện các GHT mà TT user đã gọi. TT trong kernel mode có thể truy nhập vào không gian địa chỉ của nó ở cả hai vùng kernel và user. Việc truy nhập tài nguyên hệ thống (các cấu trúc dữ liệu hê thống và phần cứng) không có giới hạn đối với kernel. Một số các lệnh máy là đặc quyền chỉ kernel mode mới dùng được.
- OS duy trì các thông tin (records) bên trong để phân biệt các TT thực hiện trên hệ thống. Mặc dù hệ thống chạy một trong hai chế độ nói trên, song kernel chạy trên danh nghĩa của TT user. Kernel không phải là tập hợp của các TT riêng biệt chạy song song với các TT user, mà là một phần của mỗi TT user. Trong văn ngữ khi nói "kernel thực hiện..." thì điều đó có nghĩa là TT chạy trong chế độ kernel thực hiện... cái gì đó. Ví dụ, shell đọc đầu vào qua GHT và được mô tả như sau: Kernel thực hiện nhân danh TT shell, kiểm soát thao tác thiết bị đầu cuối, trả lại các kí tự nhận vào cho shell. Đến đây shell, chạy trong user mode, thông dịch xâu kí tự nhận được từ người dùng và thực hiện một số hành động mà có thể các hành động đó kích hoạt GHT khác dẫn tới TT shell lại trở vào kernel mode.
- Trong môi trường đa người dùng như Unix, các thiết bị hoạt động trên cơ sở độc lập có ý nghĩa rất căn bản. Unix nhìn nhận các thiết bị như một tệp đặc biệt. Khi một t/b mới cần đưa vào hệ, người quản trị thực hiện thêm một liên kết cần thiết vào kernel. Liên kết này được biết như là phần mềm thiết bị (*device driver*) , đảm bảo rằng kernel và thiết bị được gắn lại theo cùng một phương thức mỗi khi t/b đuợc đưa vào phục vụ. Điểm mấu chốt để t/b là độc lập, liên quan tới khả năng tự thích nghi của kernel: Unix không có giới hạn số lượng của bất kì loạt t/b nào khi thích ứng vào hệ vì mỗi t/b được nhìn nhận độc lập qua liên kết riêng biệt với kernel.

4.1. Ngắt và Ngoại lệ

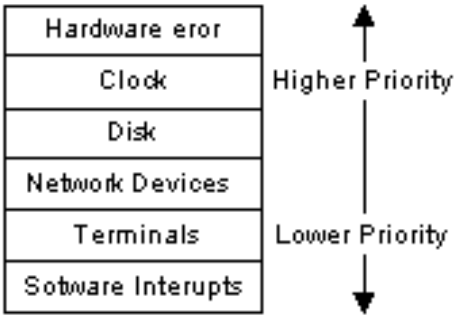
Unix cho phép các t/b như I/O, đồng hồ hệ thống ngắt CPU theo cách dị bộ. Khi chấp nhận ngắt, kernel sẽ bảo vệ bối cảnh (context) hiện tại của TT đang thực hiện, xác định lí do của ngắt, và phục vụ cho yêu cầu ngắt đó. Sau khi xử lí xong kernel khôi phục lại *context* của TT trước đó và tiếp tục thực hiện như không có gì đã xảy ra. Phần cứng thông thường có cơ chế để đặt các cấp độ ưu tiên và che các ngắt theo thứ tự mà ngắt được thao tác.

Các trường hợp *ngoại lệ* là sự kiện không trông đợi gây ra bởi một TT, ví dụ truy nhập vào vùng địa chỉ cấm, thực hiện lệnh đặc quyền, phép chia cho zero... Các *ngoại lệ* này khác với

ngắt bởi chúng phát sinh do các sự kiện bên ngoài một TT. Ngoại lệ xảy ra ở giữa chừng đang thực hiện một lệnh, và hệ thống *sẽ tái khởi động lại lệnh* sau khi đã thao tác *ngoại lệ*. Ngắt được xem là xảy ra giữa hai lệnh và hệ thống chạy lệnh tiếp theo sau xử lí ngắt. Unix dùng cùng một cơ chế để thao tác ngắt cũng như thao tác các *ngoại lệ*.

4.2. Các mức độ thực hiện xử lí

Đôi khi kernel phải ngăn chặn sự xuất hiện của ngắt trong lúc thực hiện những hoạt động có tính chất đặc biệt mà ngắt có thể làm hỏng dữ liệu hay rối loạn các con trỏ. Các máy tính thường có một số lệnh đặc biệt để làm công việc này gọi là *đặt các mức độ xử lí theo mức*, có thể che các ngắt mức thấp và cho phép ngắt mức cao.



Các mức ưu tiên ngắt

4.3. Quản lí bộ nhớ

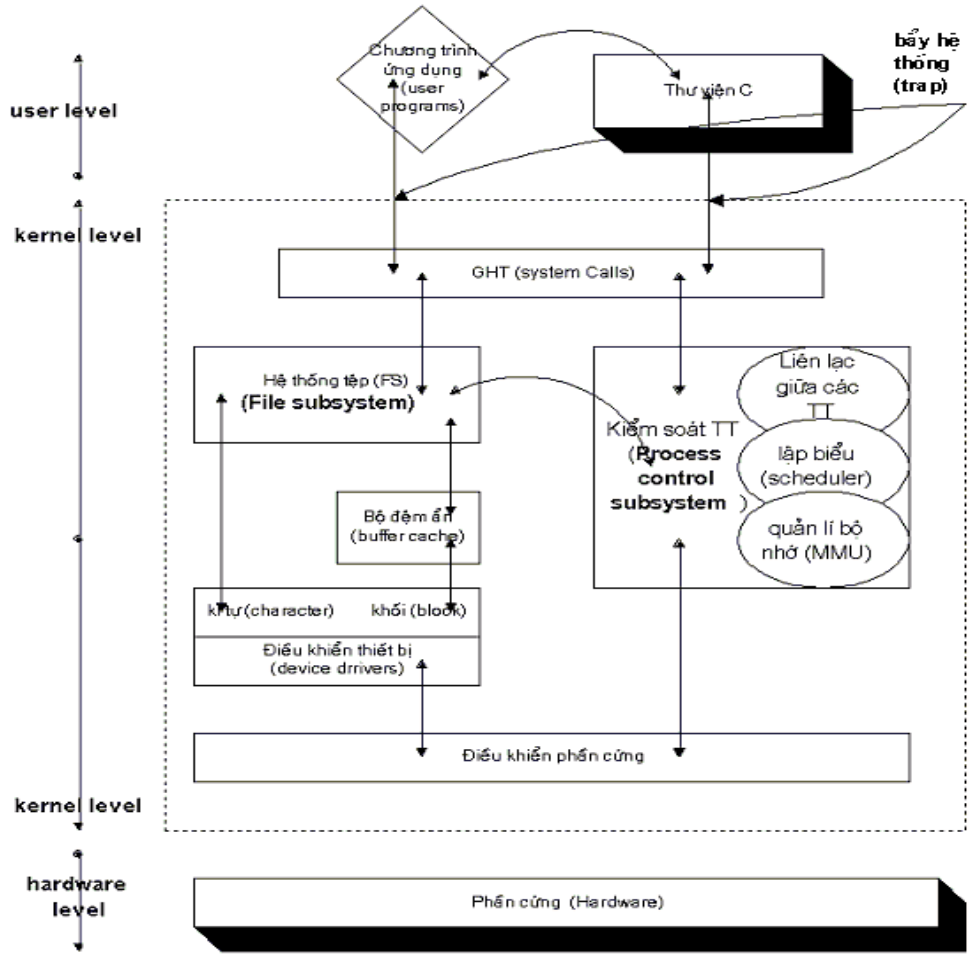
Kernel thường trú trong bộ nhớ chính và thực hiện TT hiện thời (hay ít ra là một phần của TT đó). Khi compiler dịch một chương trình, nó tạo ra tập các địa chỉ của chương trình đó cho các biến, cấu trúc dữ liệu, địa chỉ của lệnh... Compiler phát sinh ra địa chỉ cho một *máy ảo,* như thể không có chương trình nào khác sẽ được thực hiện đồng thời trên máy vật lí. Khi một chương trình chạy trong máy tính, kernel sẽ cấp cho trình một không gian địa chỉ trong bộ nhớ vật lí, nhưng không gian địa chỉ ảo này không nhất thiết phải đồng nhất với địa chỉ vật lí. Kernel phối hợp với phần cứng để ánh xạ từ địa chỉ ảo vào địa chỉ vật lí. Cách ánh xạ phụ thuộc vào đặc thù của phần cứng và các phần của Unix sẽ thích ứng theo. Ví dụ loại máy hỗ trợ theo trang (*paging*) hay theo hoán đổi (*swapping*), kernel có các hàm cơ sở tương tự cho mỗi loại cấu hình.

5. Nhân hệ điều hành (kernel)

Phần này sẽ giới thiệu tổng quát về nhân (***kernel***) của Unix theo cách nhìn kiến trúc với các khái niệm cơ bản, hỗ trợ cho các phần sau.

5.1. Kiến trúc của hệ điều hành unix

Trên Unix, hệ thống tệp (*File SystemfiFS*) có *chỗ* để cư trú và Tiến trình (TT-*Proccess*) có *cuộc đời* của nó. Tệp (*File*) và TT chính là hai khái niệm trung tâm trong mô hình của HĐH Unix. Hình sau đây là sơ đồ khối của Unix, cho thấy các *modul* và mối quan hệ giữa các *modul* đó. Phía trái là hệ thống tệp (*FS*) và bên phải là hệ thống kiểm soát TT (*procces control subsystem*), đây là hai thành phần cơ bản của Unix. Sơ đồ cho một cách *nhìn logic* về kernel, cho dù trong thực tế có thể có những khác biệt về chi tiết.



Nhân và các mức

Mô hình chỉ ra ba mức: người dùng (***user level***), nhân (***kernel level***) và phần cứng (***hardware level***). GHT và Thư viện (Lib) tạo ra ghép nối giữa chương trình của người dùng (*user programs*), còn gọi là chương trình ứng dụng, và kernel. GHT tương tự các hàm gọi trong C, và Lib ánh xạ các hàm gọi tới các hàm cơ sở (primitive) cần thiết để đi vào kernel. Các chương trình hợp ngữ (assembly language) có thể kích hoạt GHT trực tiếp không cần dùng thư viện Lib. Các chương trình thường xuyên dùng các chức năng chuẩn, ví dụ I/O của Lib, để tạo ra cách dùng tinh xảo khi GHT, và các hàm của Lib sẽ được liên kết vào chương trình vào thời điểm dịch và là bộ phận của chương trình ứng dụng. GHT chia ra thành các tập

hợp tương tác với ***file subsystem*** và với ***process control subsystem***.

File subsystem trong hình thực hiện các chức năng quản lí tệp: cấp phát vùng nhớ cho tệp, quản lí vùng trống (trong bộ nhớ, trên đĩa), kiểm soát truy nhập tệp, tìm dữ liệu cho users. TT tương tác với *File subsystem* qua một tập xác định các GHT, chẳng hạn *open* để mở tệp, *close, read, write* thao tác các kí tự của tệp, *stat* để tìm các thuộc tính tệp, *chown* thay đổi chủ sở hữu tệp, *chmod* thay đổi phép truy nhập.

File subsystem truy nhập dữ liệu bằng cơ chế đệm (buferring), điều chỉnh thông lượng dữ liệu giữa kernel và các t/b nhớ thứ cấp, tương tác với phần điều khiển (*drivers*) I/O để khởi động quá trình trao đổi dữ liệu. Các *drivers* này chính là các module của kernel, kiểm soát các t/b ngoại vi của hệ thống. Các t/b ngoại vi gồm loại truy nhập ngẫu nhiên như t/b khối (đĩa), hay liên tục (raw hay character device) như băng từ (loại này không qua cơ chế đệm).

Procces control subsystem thực hiện chức năng điều khiển đồng bộ các TT, liên lạc giữa các TT, lập biểu đưa một TT vào thực hiện và quản lí bộ nhớ. *Procces control subsystem* và *File subsystem* tác động lẫn nhau khi nạp một tệp thực thi (*executable*) vào bộ nhớ để thực hiện.

Một số trong GHT để kiểm soát TT bao gồm: *fork* (tạo TT mới), *exec* (phủ mã của chương trình kích hoạt lên vùng bộ nhớ của TT gọi exec đang chạy), *exit* (kết thúc tức thì việc thực hiện một TT), *wait* (đồng bộ thực hiện TT này với exit hay với TT trước đó đã tạo ra TT nó), *brk* (điều chỉnh kích thước bộ nhớ đã cấp cho TT), *signal* (kiểm soát đáp ứng của TT trước sự kiện khác thường).

Memory management module kiểm soát cấp phát bộ nhớ, điều chỉnh bộ nhớ qua swapping hay paging sao cho các ứng dụng có đủ bộ nhớ để thực hiện.

Scheduler tuyển chọn một TT đưa vào thực hiện: cho các TT thuê CPU để thực hiện cho tới khi TT tự động trả lại CPU để đợi một tài nguyên hoặc kernel sẽ dừng thực hiện khi lượng thời gian cấp phát cho TT đã hết. Sau đó *scheduler* sẽ chọn TT có *mức ưu tiên thích hợp nhất* để cho chạy. TT trước đó sẽ chạy lại khi nó trở thành ưu tiên thích hợp nhất.

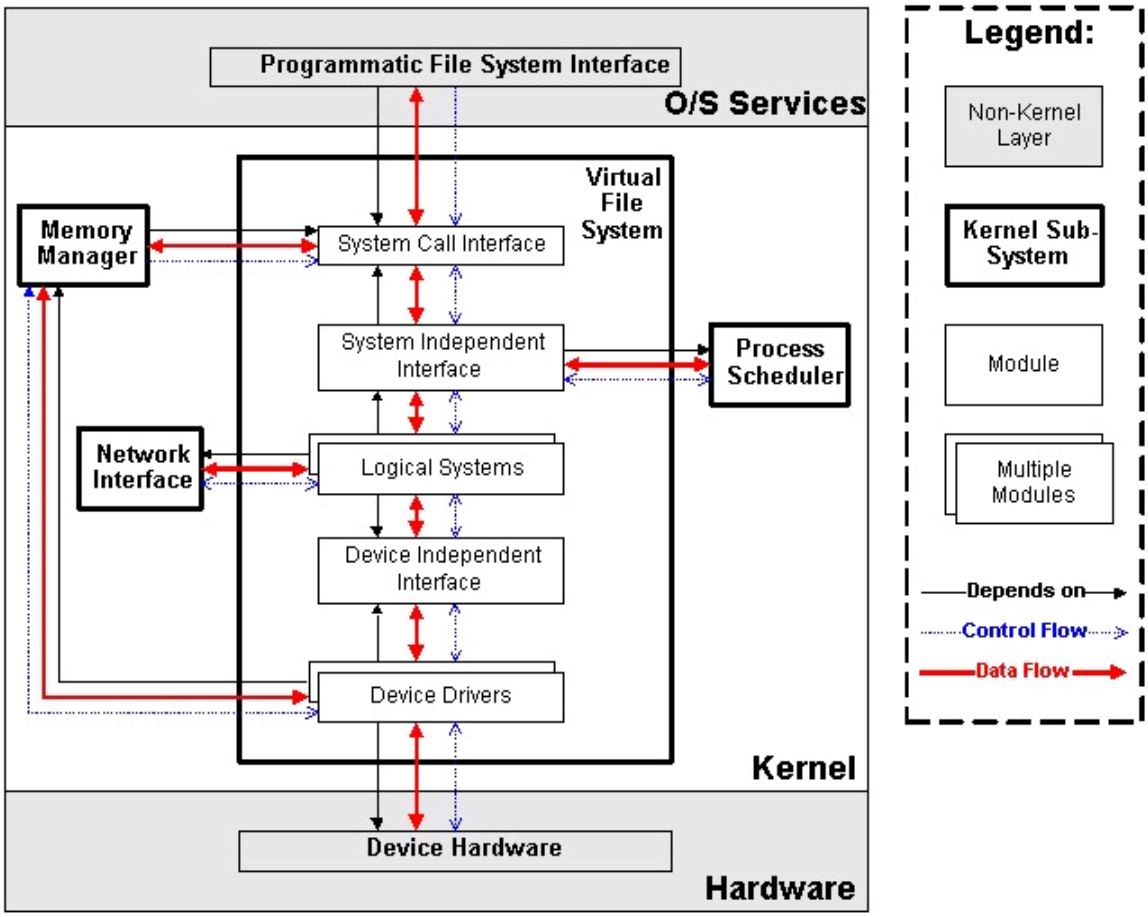
Việc *liên lạc giữa các TT* có thể thực hiện qua vài phương thức từ *đồng bộ tín hiệu của các sự kiện* hay *truyền thông điệp đồng bộ* giữa các TT.

Cuối cùng khối *hardware control* thực hiện thao tác, xử lí các ngắt (do đĩa, t/b đầu cuối...), và liên lạc với máy tính. Xử lí ngắt không thực hiện bởi một TT đặc biệt mà bởi các *chức năng đặc biệt* trong kernel được gọi tới (phát động) trong bối cảnh của TT hiện đang chạy.

Chương II. Hệ thống tệp (*file system)*

A. Tổng quan về Hệ thống tệp ảo (VFS)

VFS được thiết kế để biểu diễn cách trình bày dữ liệu trên một thiết bị cứng (đĩa cứng chẳng hạn). Hầu hết các thiết bị cứng đều thông qua phần mềm điều khiển cùng kiểu (*generic device driver*) để liên kết vào máy tính. VFS, xa hơn, cho phép người quản trị "ghép" (mount) bất kì một hệ thống tệp logic trên bất kì thiết bị nào vào hệ thống. VFS trừu tượng hoá các chi tiết của cả hai thiết bị vật lí và hệ thống tệp logic, và cho phép TT người dùng truy nhập tệp dùng các giao tiếp thông thường mà không cần biết hệ thống tệp logic hay vật lí tồn tại ở đâu.



Kiến trúc Hệ thống tệp ảo

Các modules

1. Cho mỗi thiết bị, có một trình điều khiển. Do nhiều thiết bị không tương thích nhau, nên có một số lượng lớn trình điều khiển cho mỗi loại.
2. Modul ghép nối độc lập với thiết bị (Device Independent Interface) cung cấp mô tả thích hợp của tất cả các thiết bị.
3. Đối với mỗi hệ thống tệp, có một hệ thống tệp logic tương ứng.
4. Giao diện độc lập với hệ thống (system independent interface) cho mô tả phần cứng và mô tả sự độc lập của hệ thống tệp logic với các tài nguyên phần cứng.
5. Giao diện gọi hệ thống (*system call interface*) cung cấp truy nhập có kiểm soát cho TT người dùng vào hệ thống tệp. VFS chỉ trao cho TT người dùng những chức năng nhất định.

Biểu diễn dữ liệu

Tất cả các tệp được biểu diễn bởi các *i-node*. Mỗi cấu trúc i-node cho thông tin vị trí của các khối của tệp ở thiết bị vật lí nào. Nó chứa các con trỏ của các trình thủ tục trong module hệ thống tệp logic và các trình điều khiển các thao tác đọc/ghi. Bằng cách lưu các con trỏ các hàm chức năng theo cách này, các hệ thống tệp logic và các trình điều khiển tự ghi nhận với nhân HĐH làm cho nhân HĐH không bị phụ thuộc vào bất kì module cụ thể nào.

Tính độc lập, dòng dữ liệu và hướng điều khiển

Trình điều khiển của một thiết bị đặc trưng là một mô phỏng của thiết bị trong bộ nhớ (ramdisk): thiết bị thuê một không gian của bộ nhớ để đặc tả về thiết bị, xử lí đặc tả dó như thể xử lí chính thiết bị. Như vậy nó sử dụng quả lí bộ nhớ để hoàn tất các thao tác, như vậy sẽ có sự phụ thuộc, có luồng điều khiển, dòng dữ liệu giữa trình điều khiển thiết bị của hệ thống tệp và quản lí bộ nhớ.

Một trong các hệ thống tệp logic là hệ thống tệp mạng chỉ với vai trò là khách thể (client của một máy khác). Hệ tệp này truy nhập các tệp trên một máy khác như thể đó là một phần của một máy logic. Để làm được điều này, một trong các module hệ thống tệp sử dụng hệ thống con mạng. Như vậy sẽ có sự phụ thuộc, dòng dữ liệu và luồng điều khiển giữa hai hệ thống con.

Như đã nói, quản lí bộ nhớ sử dụng VFS để thực thi hoán đổi bộ nhớ-thiết bị, đồng thời sử dụng lập biểu TT để treo TT trong khi đợi thao tác vào/ra hoàn thành và cho chạy lại khi vào/ra đã kết thúc. Giao diện gọi hệ thống cho phép TT người dùng gọi vào VFS để cát hay tìm dữ liệu. Chổ khác biệt ở đây là không có cơ chế nào để người dùng đăng kí yêu cầu không tường minh, do vậy sẽ không có luồng diều khiển từ VFS tới TT người dùng.

Các kiểu tệp và Hệ thống tệp

1. tệp chính tắc (regular) (-): là tệp phổ biến cho lưu thông tin trong hệ thống.
1. tệp thư mục (directory) (d): là tệp danh mục của các tệp;
2. tệp đặc biệt (special file) (c,f): là một cơ chế sử dụng cho cá thao tác vào / ra (I/O), fifo. Các tệp loại này nằm ở thư mục /dev.
3. liên kết (link) (l): là một hệ thống tạo ra một thư mục hay tệp nhìn thấy được trong nhiều phần của cây hệ thống tệp.
4. socket (s): là loại tệp đặc biệt cho các truyền thông mạng của một tiến trình bên trong hệ thống, và được bảo vệ bởi qui tắc truy nhập tệp.
5. ống (pipe) (p): là cơ chế để các tiến trình liên lạc với nhau.

Và một số kiểu khác.

Linux files

- **tên_tệp.bz2** file compressed with bzip2
- **tên_tệp.gz** file compressed with gzip
- **tên_tệp.tar** file archived with tar (short for tape archive), also known as a tar file
- **tên_tệp.tbz** tarred and bzipped file
- **tên_tệp.tgz** tarred and gzipped filetên_tệp
- **tên_tệ.pzip** file compressed with ZIP compression

File Formats

- **tên_tệp.au** audio file
- **tên_tệp.gif** GIF image file
- **tên_tệp.html** HTML file
- **tên_tệp.jpg** JPEG image file
- **tên_tệp.pdf** an electronic image of a document; PDF stands for Portable Document Format
- **tên_tệp.png** PNG image file (short for Portable Network Graphic)
- **tên_tệp.ps** PostScript file; formatted for printing
- **tên_tệp.txt** ASCII text file
- **tên_tệp.wav** audio file
- **tên_tệp.xpm** image file

System Files
. .conf a configuration file. Configuration files sometimes use the .cfg extension, as well. . .lock a *lock fi*le; determines whether a program or device is in use
. .rpm a Red Hat Package Manager file used to install software

Programming and Scripting Files
. .c a C program language source code file
. .cpp a C++ program language source code file
. .h a C or C++ program language header file
. .o a program object file
. .pl a Perl script
. .py a Python script
. .so a library file
. .sh a shell script
. .tcl a TCL script

Các cơ sơ dữ liệu hệ thống cho tệp

Trong hệ thống tệp, tệp được biểu diễn bằng một ***inode.*** Inode cho biết mô tả của một tệp trên đĩa cùng các thông tin khác như sở hữu tệp, quyền truy nhập, thời gian truy nhập tệp. Khái niệm *inode* có ý nghĩa là chỉ số của một *nút* (***index node, trong FS là một lá của cấu trúc cây***) và dùng thông dụng trong tài liệu về Unix. Mỗi tệp có một inode duy nhất, nhưng mỗi inode lại có thể có nhiều tên (tệp) khác nhau. Các tên tệp này đều qui chiếu tới inode đó và mỗi tên như vậy gọi là một liên kết (***link***). Khi một TT truy nhập một tệp bằng tên, kernel sẽ phân tích tên trong đường dẫn, tìm tới thư mục chứa tệp, kiểm tra phép truy nhập, tìm inode và trao inode cho TT. Khi TT tạo tệp mới, kernel gán cho tệp một inode chưa sử dụng. Các inode lưu trong FS trên đĩa, nhưng khi thao tác tệp, kernel đọc từ đĩa và đưa vào một bảng gọi là ***in - core inode table*** (gọi tắt là ***Inode table***) trong bộ nhớ hệ thống.

Kernel còn có hai cấu trúc dữ liệu khác là bảng các tệp (***File table***) và bảng các mô tả tệp của người dùng (***User file descriptor per procces table*** gọi tắt là ***File descriptor table***). ***File table*** là cấu trúc tổng thể của kernel, còn ***File descriptor table*** cấp cho TT khi TT mở một tệp. Khi TT *open* hay *creat* tệp, kernel cấp một đầu vào trong mỗi bảng tương ứng với

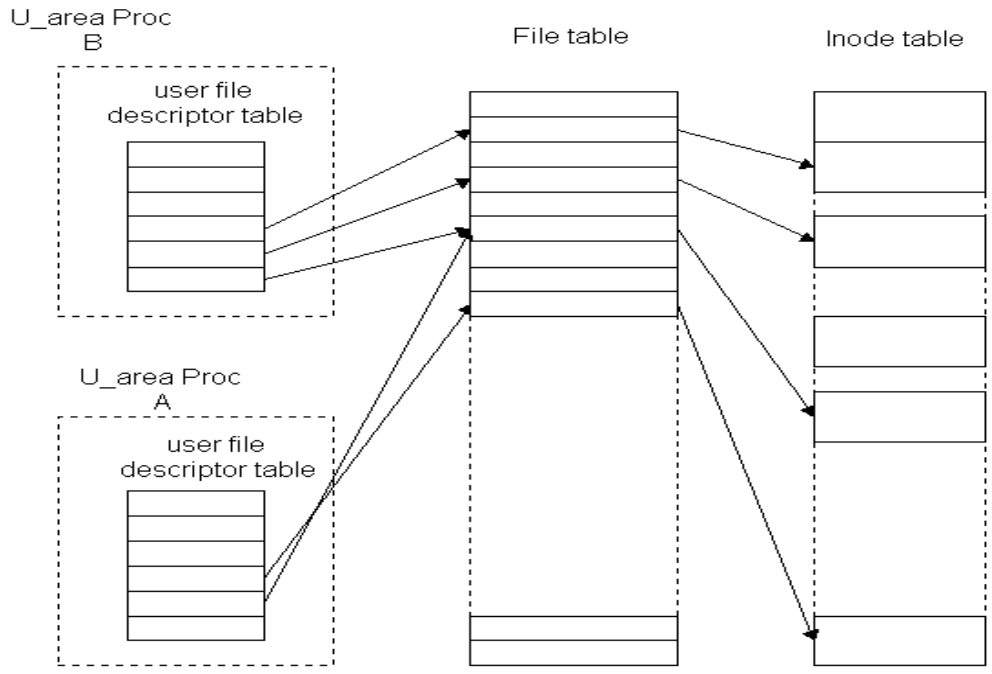
tệp đó. Các thông tin có trong các đầu vào ở ba bảng sẽ duy trì trạng thái của tệp cũng như khả năng user truy nhập tới tệp:

- **File table** theo dõi quyền truy nhập, /đọc/ghi byte tiếp theo trong tệp qua con trỏ tệp *byte offset, số đếm qui chiếu cho biết tổng số các TT truy nhập tệp, con trỏ trỏ vào inode table*;

- **File descriptor table** cho biết tất cả các tệp một TT đã mở. Hình sau cho thấy mối quan hệ giữa các bảng nói trên: kernel trả lại *mô tả tệp-file descriptor*, chỉ số trỏ tới một đầu vào của File descriptor table, của mỗi tệp khi TT GHT *open* và *creat*. Khi thực hiện GHT *read* hay *write*, kernel dùng *file descriptor* để vào *File descriptor table*, lấy con trỏ tìm tới *File table*, rồi từ đó theo con trỏ trong *File table* truy nhập vào *Inode table*. Từ *inode* sẽ có các thông tin cần thiết để truy nhập dữ liệu của tệp.

- **Inode table** là bảng của kernel, chứa các inode được đọc từ đĩa. Mỗi inode khi được đọc vào bộ nhớ sẽ được cấp một đầu vào trong bảng, mỗi đầu vào đó cho một mảng dữ liệu đặc tả về một inode đĩa (xem định nghĩa về inode).

Với *ba cấu trúc dữ liệu hệ thống* trên kernel có đủ các mức độ khác nhau để thực hiện quản lí và các thao tác chia sẻ tệp. Tệp trong Unix được để trên thiết bị khối (đĩa, băng từ). Máy tính có thể có nhiều đĩa, mỗi đĩa có thể có một hay vài phân hoạch, mỗi phân hoạch tạo thành một hệ thống tệp (***File systemfiFS***). Phân hoạch một đĩa thành nhiều phần tạo điều kiện dễ dàng cho việc kiểm soát dữ liệu, vì kernel xử lí ở mức logic với các FS chứ không phải với bản thân thiết bị. Mỗi một phân hoạch là một thiết bị logic và nhận biết nó bằng số của t/b.



File descriptors (per process, File table, Inode table

Ví dụ: SCO Unix: *hd0a*: *hd* chỉ hard disk, *0* chỉ đĩa IDE primary, *a* chỉ phân hoạch thứ nhất. Linux: *hda1*: *hd* chỉ hard disk, *a* chỉ đĩa IDE primary, 1 chỉ phân hoạch thứ nhất. (Trên PC, BIOS cho phép tạo tối da 4 phân hoạch chính (primary partions), các phân hoạch khác sẽ là phần mở rộng bên trong một phân hoạch chính đó). Do vậy ta có: hda6 sẽ là phân hoạch mở rộng của một phân hoạch chính nào đó một khi đã sử dụng hết 4 phân hoạch chính. Muốn biết cụ thể chỉ có thể ghi nhận lại trong quá trình cài đặt khi tạo các phân hoạch đĩa.

Qui ước chung: *xxyN* ,

trong đó: xx kiểu thiết bị (*major number*), y số hiệu của t/b (*miror number*), N số của phân hoạch trên t/b, ví dụ trên máy tính các nhân (PC):

1. hd = loại IDE: hda=IDE Primary 1, hda1, hda2 phân hoạch 1 và 2 của Primary IDE;

   hdb=IDE Secondary 1, hdb1, hdb2, hdb3,

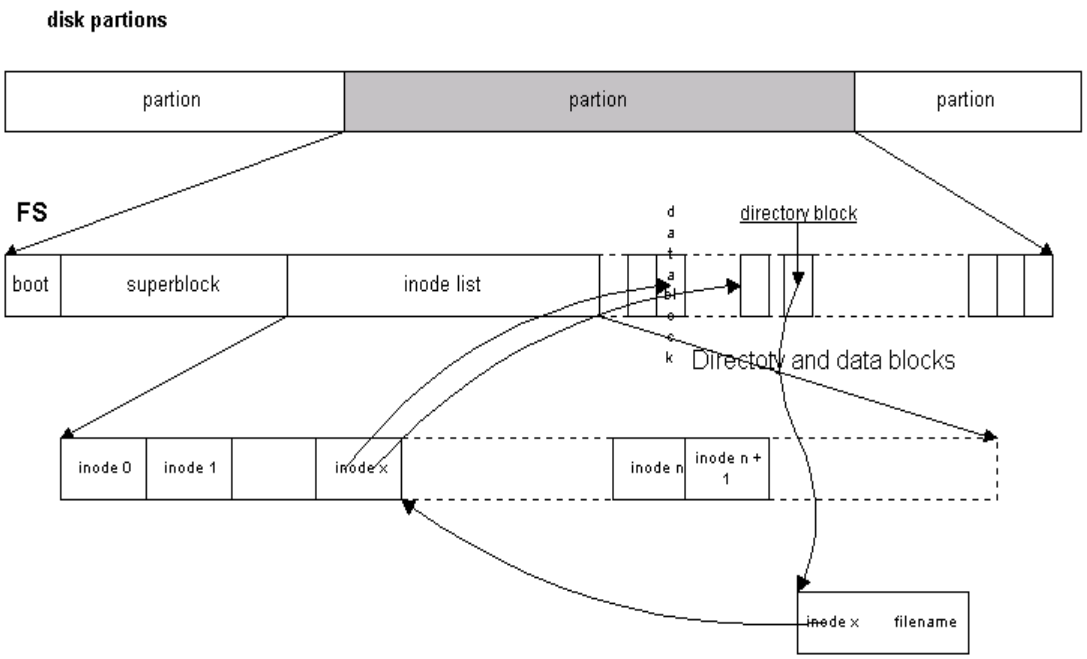
   hdc=IDE Primary 2,

   hdd=IDE Secondary 2,

2. sb = loại SCSI, sba1, sba2, …

Việc biến đổi giữa địa chỉ của thiết bị logic (magic number của FS) và địa chỉ của thiết bị vật lí (ổ đĩa) được thực hiện bởi bộ điều khiển đĩa (disk driver). Sách dùng khái niệm thiết bị là để chỉ tới thiết bị logic, trừ khi có diễn đạt cụ thể khác.

FS chứa một chuỗi các khối logic có độ lớn 512 bytes hay bội số của 512 tuỳ hệ và là đồng nhất bên trong mỗi hệ, nhưng có thể khác nhau khi áp dụng trên mỗi hệ cụ thể. Độ lớn này có ảnh hưởng nhất định về thời gian truy nhập cũng như hiệu quả sử dụng dung lượng đĩa. Sách đề cập tới "khối" có nghĩa một khối logic đĩa (*logical block*) với kích thước là 1 K bytes. FS có sắp xếp hình thức như sau: Ví dụ đĩa có 3 phân hoạch, mỗi phân hoạch là 1 FS:



Tổ chức hệ thống tệp trên đĩa

Linux ext2 FS:

- ***Boot block***, phần đầu tiên của FS đĩa, là sector đầu tiên chứa mã bootstrap được đọc vào máy và chạy để nạp HĐH.
- ***Superblock***, mô tả tình trạng của FS: độ lớn, chứa được bao nhiêu tệp (inode), không

gian còn trống (block) của đĩa để chứa nội dung tệp tìm thấy ở đâu, cũng như các thông tin khác.

Super block có các trường sau đây:

1. kích thước của FS,
2. tổng số các block còn chưa cấp phát cho tệp (*free block*) trong FS.
3. *danh sách các free block sẵn có* trên FS (xem thêm phần cấp phát block đĩa),
4. chỉ số của free block tiếp theo trong danh sách free block.
5. kích thước của danh sách inode,
6. tổng số các inode chưa cấp phát (*free inode*) trong FS,
7. *danh sách free inode* trong FS,
8. chỉ số của free inode tiếp theo trong danh sách free inode,
9. các trường khoá (*lock*) cho free block và các danh sách free inode,
10. cờ (*flag*) cho biết super block đã có thay đổi.

Phần tiếp theo sẽ nói đến cách sử dụng các trường, các chỉ số và khóa. Kernel sẽ thường xuyên cập nhật *super block* một khi có sự thay đổi sao cho nó luôn nhất quán với data trong hệ thống.

- ***Inode List,*** danh cách các inode trong FS. Người quản trị xác định kích thước khi làm cấu hình (cài đặt) hệ thống. Kernel qui chiếu các inode bằng chỉ số (index) trong inode list. Có một inode gọi là *root inode* trong mỗi FS: là inode khởi đầu để vào được FS sau khi thực hiện GHT phép *ghép* (*mount*) FS đó vào cây thư mục gốc.
- ***Data blocks,*** vùng chứa nội dung (dữ liệu) của tệp và dữ liệu quản trị hệ thống (là các block của tệp thư mục, các block nội dung của một inode). Một khối khi đã cấp cho mỗi tệp thì khối đó chỉ thuộc tệp đó mà thôi.

2.Biểu diễn bên trong của tệp

Mỗi một tệp trong UNIX có một chỉ số duy nhất (***inode)*** gán cho tệp lúc tệp được tạo**.** Inode chứa các thông tin cần thiết để một TT truy nhập tệp, ví dụ như người sở hữu tệp, quyền truy nhập, kích thước của tệp, vị trí data của tệp trong FS. Các TT truy nhập tệp bằng các GHT và xác định một tệp bằng xâu kí tự gọi là đường dẫn tên tệp. Mỗi đường dẫn lại xác định duy nhất một tệp, và kernel sẽ biến đổi đường dẫn đó thành *inode* của tệp.

Chương này đề cập tới cách tổ chức của tệp nói riêng và của FS nói chung như: inode, tệp và ghi/ đọc tệp, tệp thư mục, tệp và tổ chức đĩa: inode đĩa và block đĩa cho tệp; đồng thời cũng đưa ra các thuật toán cơ sở thao tác tệp. Các thuật toán này nằm ở lớp trên của buffer cache.

2.1 Inode (Index-node), định nghĩa và cấu trúc

Inode tồn tại ở trên đĩa (***disk inode***) còn gọi là inode dạng tĩnh và kernel đọc inode đó vào bộ nhớ (gọi là ***in - core inode***) để xử lí. *In - core inode* là một mảng trong ***Inode table.*** Một inode khi đọc vào bộ nhớ là tập hợp của hai phần gồm phần tĩnh trên đĩa và phần động

inficore inode.

Disk inode gồm có các trường sau đây:

- Nhận dạng người sở hữu tệp. Quan hệ sở hữu chia ra làm sở hữu cá thể và sở hữu nhóm định nghĩa một tập hợp các users có quyền truy nhập tệp. *Superuser* được quyền chi phối mọi tệp trong FS.
- Kiểu tệp (*file type*): tệp thường (regular), thư mục (directory), kí tự (character), kiểu khối đặc biệt (block special), hay FIFO (pipes). (Khi trường = 0, inode tương ứng chưa dùng (free)).
- Phép truy nhập tệp (Permission). Hệ thống bảo vệ tệp theo ba lớp: 1. người sở hữu (ownerfi***u***) và 2. nhóm người sở hữu (groupfi***g***) và 3. những người khác (otherfi***o***). Mỗi lớp lại được lập các quyền khác nhau một cách riêng biệt như: đọc (***r***firead), ghi (***w***fiwrite), thực hiện (***x***fiexecute) (chạy một tệp thực thi). Riêng thư mục thực hiện là tương đương với tìm tệp trên thư mục.
- Thời điểm truy nhập tệp (access time), cho biết thời điểm tệp đã có sự thay đổi vào lần truy nhập cuối cùng và khi inode đã có thay đổi.
- Số các liên kết (*link*) tới tệp. Tệp có thể có nhiều tên trong cấu trúc thư mục, đó là các link của tệp.
- Bảng *địa chỉ các block data* đĩa cấp phát cho tệp. Mặc dù user xử lí tệp như một xâu các kí tự, thì kernel cất data của tệp trên các block đĩa không liên tục, nội dung các block đĩa khi đọc và sắp xếp lại chính là nội dung của tệp.
- Độ dài của tệp. Data trong tệp được địa chỉ hoá bằng tổng số bytes, bắt đầu từ đầu tệp và khởi động bởi *offset*=0 (con trỏ tệp, ứng với byte đầu tiên) và độ dài của tệp là một số lớn hơn offset lớn nhất 1 đơn vị (*offset* max + 1).

Ví dụ:	***owner***	***john***
	group	os
	type	regular file
	permission	rwx r-x r-x
	accessed	Oct 23 1999 1:45 P.M
	modified	Oct 22 1999 10:30 A.M
	inode changed at	Oct 23 1999 1:30 P.M
	size	6030 bytes
	disk addresses (danh sách địa chỉ các khối đĩafidisk blocks)	

Tất cả các thông tin trên là nội dung của inode, cho biết chi tiết về bản thân một tệp mà nhờ đó user truy nhập tới nội dung cỉa tệp. Khi nói tới ghi đĩa, cần phân biệt ghi *nội dung của tệp* lên đĩa và ghi *nội dung của inode* lên đĩa. Nội dung của inode thay đổi khi thay đổi nội dung của tệp hay khi thay đổi các thuộc tính như owner, permission, links. Thay đổi nội dung của tệp tự động dẫn đến thay đổi inode, nhưng thay đổi inode không dẫn tới thay đổi nội dung của tệp.

Incore - inode là bản sao nội dung inode trên đĩa (disk inode) vào bộ nhớ và để tại *Inode table* trong bộ nhớ sau đó sẽ có thêm các trường sau đây:

- Trường trạng thái của *inode in - core* cho biết:
  - inode đã khoá (*locked*);
  - có một TT đang đợi inode trong khi inode bị khóa, chờ inode được giải khóa (*unlocked*);
  - bản *in - core* của inode có sự khác biệt với bản sao từ trên đĩa bởi có sự thay đổi thông tin tổ chức (data) trong inode;
  - bản *in - core* của inode có sự khác biệt với bản sao từ trên đĩa bởi có sự thay đổi nội dung (*data*) trong tệp;
  - tệp là một điểm ghép (*mount point*) của cấu trúc FS; Số thiết bị logic của FS chứa tệp.
- Số của thiết bị logic mà tệp được lưu trên thiết bị đó;
- Số của inode. Các inode được cất trong một mảng tuyến tính trên đĩa, kernel nhận biết số của inode đĩa bằng vị trí của nó trong mảng. (Inode đĩa không cần có trường này).
- Các con trỏ trỏ tới các *in - core inode* khác. Kernel liên kết các inode trên một hàng băm (*hash queue*) và trong một danh sách các inode chưa sử dụng (*free list*). Một hàng băm thì được nhận biết bởi số của thiết bị logic của inode và số của inode. Kernel có thể chứa nhiều nhất một bản sao *in - core inode* của một inode đĩa, trong khi đó các inode có thể đồng thời có mặt trong *hash queue* và trong *free list*.
- Một số đếm qui chiếu (*reference count*) cho biết tệp mở bao nhiêu lần (*actived*), ví dụ khi các TT dùng hàm open() để mở cùng một tệp, mỗi lần mở số đếm này tăng lên 1 đơn vị. Một *in – core inode* có trong free list chỉ khi nào trường số đếm này có giá trị bằng 0 và inode đó là *inactive* và kernel sẽ cấp inode đó cho một disk inode khác (khi có TT open một tệp nào đó).

Một inode bị khoá (*locked*) là để không cho các TT khác truy nhập tới inode đó. Các TT khác sẽ đặt một *flag* trong inode khi muốn truy nhập inode để thông báo rằng các TT đó sẽ được đánh thức khi inode được mở trở lại (*unlock*). Trong khi đó, kernel đặt các *flag* khác để chỉ ra sự khác nhau giữa *disk inode* và *in - core inode*. Khi kernel cần cập nhật mới disk inode từ inficore inode, kernel sẽ kiểm tra các cờ (flag) trạng thái này.

Free list của các inode được sử dụng giống như là một cache của các *inactive inode*: Khi TT truy nhập một tệp mà inode của tệp chưa có trong nhóm *in - core inode*, kernel sẽ cấp cho nó một *in - core inode* từ *free list* để sử dụng.

2.2. Truy nhập các inodes

Kernel nhận biết các inode qua FS và số của inode sau đó cấp cho inode đó một *in - core inode* và việc đó được thực hiện bởi thuật toán ***iget().*** Thuật toán này về tư duy cũng giống như getblk() tìm một block đĩa trong buffer cache. Kernel sẽ ánh xạ số của thiết bị và số của inode vào hàng băm, và tìm một hàng có inode cần tìm. Nếu không thấy inode đó (chưa có), kernel cấp một *in - core inode* trong *free list* và khoá inode đó lại. Sau đó kernel chuẩn bị để đọc bản sao của *disk inode* vào *in - core inode* vừa cấp phát: kernel biết số của inode và số của thiết bị logic, kernel tính toán block logic đĩa có chứa *disk inode*, tính ra số của block đó, đọc vào buffer hệ thống, lấy inode và sao vào *inficore inode* trong *inode table*. Phép tính dựa vào công thức sau đây:

block number = ((inode number –1) / numbrer of inode per block)

+ start block of inode list.

Trong phép chia này, chỉ *lấy phần nguyên* của thương số.

Ví dụ: block số 2 là block đầu tiên của các block dành cho inode list (xem lại hình 2.3 chương 2); mỗi block có tất cả 8 inode; Hãy tìm block chứa inode số 8, inode số 9:

Bl = ((8-1) / 8) + 2 = 2; block đĩa số 2 chứa inode số 8.

Bl = ((9-1) / 8) + 2 = 3; block đĩa số 3 chứa inode số 9.

Khi biết được số của thiết bị (đĩa cứng số:.../phân hoạch số:...) và số của block đó trên đĩa cứng, kernel dùng chức năng **bread()** để đọc block và tìm tới *offset của inode* trong block bằng thuật toán:

offset của inode = ((inode number-1) modulo (number of inodes per block)) * size of disk inode

(modulo là *lấy số dư* của phép chia của hai số).

Giả sử cấu trúc một disk inode có: 8 inodes, mỗi inode có độ dài 64 bytes. Tìm inode số 8 (byte offset bắt đầu của inode số 8):

((8-1) modulo (8)) * (64) = 448, inode số 8 bắt đầu ở byte số 448 của block đĩa inode.

Quá trình tiếp theo là:

- kernel cấp *in-core inode* (lấy ra từ *free list*), đặt *in - core inode* vào hàng băm;
- đặt (khởi động) số đếm qui chiếu = 1 (có một TT mở tệp này);
- copy toàn bộ các thông tin từ inode disk nói trên vào *in-core inode*;
- khóa (lock) *in - core inode* lại.
- trả lại cấu trúc (con trỏ) inode cho TT gọi.

Kernel dùng chức năng **iget()** tìm một inode trong FS, với các đầu vào:

input: số của inode trong FS (inode number)

output: là inode đặt trong Inode Table đã khóa lại chống tranh chấp

Ở đây **iget(), bread()** là các chức năng (hàm) cơ sở của nhân HĐH.

2.3. Giải phóng một inode

Khi một inode được giải phóng khỏi sự sử dụng của một TT (TT *close* tệp nó truy nhập), kernel giảm số đếm đi 1. Nếu số đếm trở thành 0 (không còn TT nào truy nhập tệp), kernel cập nhật (ghi) inode lên đĩa nếu bản *in - core* có sự khác biệt với disk inode (xem lại các tiêu chí về sự khác biệt trong phần trước). Kernel đặt inode vào *free list* của các inode, ẩn inode trong cache để có dùng ngay khi tái sử dụng. Kernel đồng thời giải phóng tất cả block đĩa đã dùng kết hợp cho tệp và nếu số link = 0, giải phóng luôn inode. Quá trình đó được mô tả bằng thuật toán *iput()*.

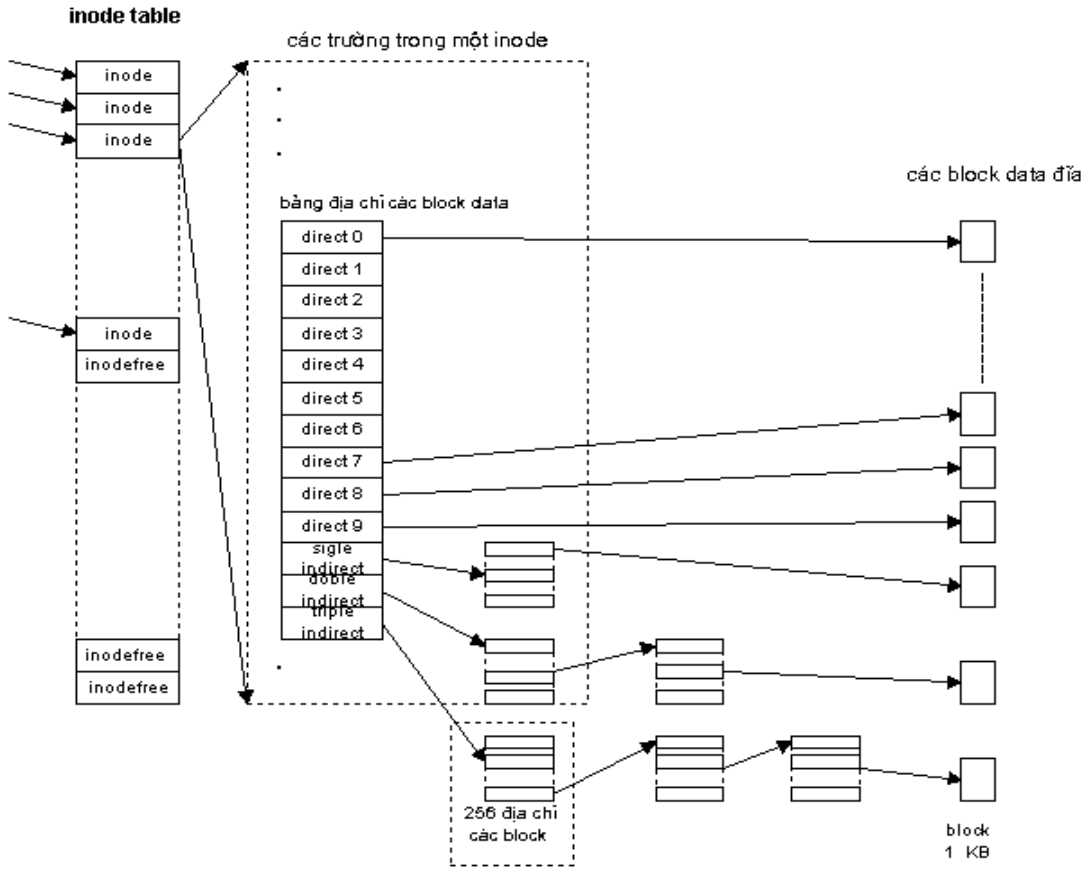
Kernel dùng chức năng ***iput()*** /*giải phóng truy nhập cho một *in - core inode*/

Input: con trỏ trỏ vào in - core inode trong bảng Inode Table;

Output: Không có mã trả lại

3. Cấu trúc của tệp thông thường (regular file hay ordinary file)

Như đã nói, inode chứa bảng *địa chỉ các block data* để định vị data trên đĩa. Mỗi block đĩa được đánh dấu bằng một số, do vậy bảng bao gồm tập hợp các số của các block đĩa. Nếu data của tệp được ghi trên một vùng liên tục của đĩa (trình tự tuyến tính các block đĩa), thì lưu địa chỉ của block khởi đầu và độ dài của tệp trong inode là đủ để truy nhập tệp. Tuy nhiên chiến lược cấp phát như vậy sẽ không cho phép mở rộng và thu nhỏ các tệp trên một hệ thống tệp khi không thực hiện biện pháp phân đoạn các vùng nhớ trống trên đĩa. Hơn nữa kernel đã có thể phải cấp phát và dành riêng những vùng đĩa liên tục trước khi cho phép các thao tác tăng độ dài của tệp.



Tổ chức địa chỉ khối đĩa trực tiếp và các kiểu gián tiếp

Ví dụ: User tạo 3 tệp A, B, C mỗi tệp dùng 10 block đĩa và được cấp các block liên tục (xem hình dưới). Nếu user cần mở rộng tệp B thêm 5 block vào giữa tệp, thì kernel phải sao chép lại vào vùng đĩa khác với 15 block liên tục. Bên cạnh việc thực hiện một thao tác đắt giá như vậy thì 10 block trước đó chỉ có thể cấp cho các tệp mới nhỏ hơn 10 block. Kernel có thể tối thiểu hoá sự phân đoạn như vậy bằng chạy định kì các thủ tục "gắn" các không gian đĩa lại nhưng điều đó bòn rút nhiều sức mạnh xử lí của hệ thống.

Để linh hoạt hơn, kernel phân phối một block mỗi lần cho tệp và cho phép data của tệp phân tán qua FS, tuy nhiên sơ đồ cấp phát như vậy sẽ là phức tạp nhiệm vụ định vị data. Bảng nội dung có thể bao gồm danh sách số các block chứa data thuộc tệp, thế nhưng bằng cách tính đơn giản chỉ ra rằng danh sách tuyến tính các block trong inode khó quản lí. Nếu

một block logic là 1 K bytes, thì tệp dài 10 K cần một chỉ số của 10 block, vậy nếu là 100 Kb thì số chỉ số sẽ là 100 số để đánh dấu block. ở đây ta thấy kích thước của inode do vậy sẽ thay đổi theo độ dài của tệp, hoặc sẽ có một giới hạn thấp sẽ áp đặt lên độ dài của tệp.

Để giữ cho cấu trúc inode nhỏ mà vẫn cho phép tệp lớn, bảng các block đĩa thích hợp với cách trình bày trên hình dưới đây. Hệ UNIX System V làm việc với 13 đầu vào của bảng các block trong một inode, nhưng nguyên tắc là độc lập với tổng số các block:

- Các đầu vào đánh dấu là "*direct*" cho số của block đĩa chứa data của tệp.
- Đầu vào đánh dấu "*single indirect*" chỉ tới một block mà nội dung của nó là danh sách số các block trực tiếp chứa data của tệp. Để lấy data qua các block gián tiếp kernel phải đọc block gián tiếp, lấy ra số của block trực tiếp, sau đó đọc block trực tiếp đó.
- Khối đánh dấu "double indirect" cho danh sách số của các block gián tiếp đôi: block gián tiếp → block gián tiếp
- Khối "*triple indirect*" cho danh sách số của các block gián tiếp ba: block gián tiếp → block gián tiếp → block gián tiếp.

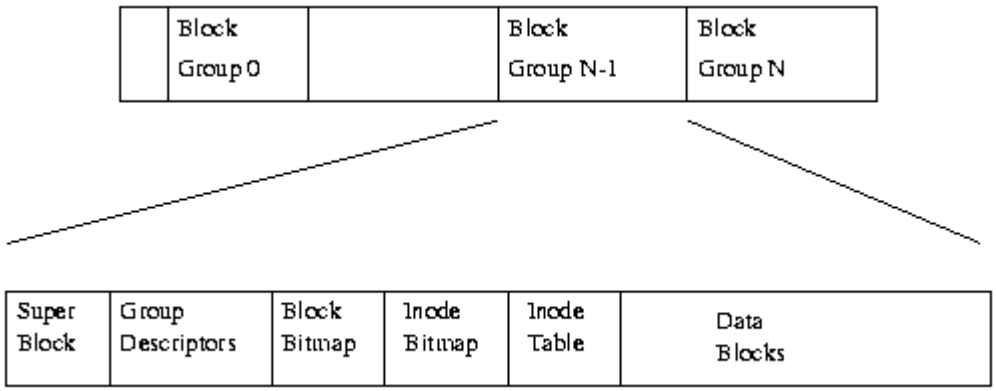
Về nguyên lí có thể mở rộng tới gián tiếp bốn, gián tiếp năm v.v...thế nhưng thực tế cấu trúc như trên là đủ. Giả định rằng một block logic là 1 K bytes và để thể hiện số nguyên của block cần 32 bit. Vậy 1 block có thể chứa được 256 số của các block. Nếu chỉ dùng cấu hình: 10 đầu vào trực tiếp, 1 đầu vào gián tiếp đơn, 1 đầu vào gián tiếp đôi và 1 đầu vào gián tiếp ba trong một inode, thì có thể tạo một tệp dài hơn 16 Gbyte:

10 đầu vào cho 10 block trực tiếp cho	10 Kbyte
1 đầu vào cho 1 block gián tiếp cho	256 x 1 K = 256 K
1 đầu vào cho 1 block gián tiếp đôi cho	256 x 256 x 1K = 64 M byte
1 đầu vào cho 1 block gián tiếp ba cho	256x256x256x1K= 16 Gbyte

Nếu trường chứa độ dài tệp trong inode là 32 bit, thì kích thước hiệu dụng của tệp sẽ giới hạn tới 4 Gbyte (2^{32}).

Các TT truy nhập data của tệp bởi *byte offset* (con trỏ tệp), làm việc trong khái niệm số đếm byte và nhìn tệp như một chuỗi các byte bắt đầu ở byte có địa chỉ bằng 0 và tiến tới cuối tệp. Kernel biến đổi cách nhìn byte của user thành cách nhìn vào block đĩa: tệp bắt đầu ở block logic 0 và tiếp tục tới block tương ứng với độ dài của tệp. Kernel truy nhập inode, biến đổi block logic vào block đĩa thích hợp. Thuật toán *bmap()* thực hiện biến đổi *file offset* thành block đĩa vật lí.

LINUX:



EXT2 Inode

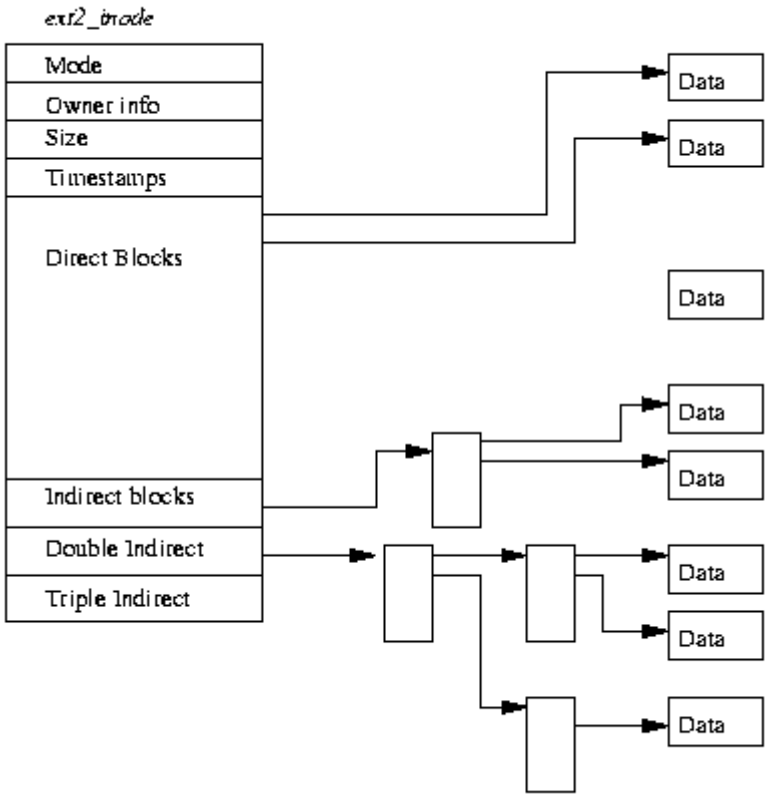


Figure 9.2: EXT2 Inode

In the EXT2 file system, the inode is the basic building block; every file and directory in the file system is described by one and only one inode. The EXT2 inodes for each Block Group are kept in the inode table together with a bitmap that allows the system to keep track of allocated and unallocated inodes. Figure 9.2 shows the format of an EXT2 inode, amongst other information, it contains the following fields:

mode

This holds two pieces of information; what this inode describes and the permissions that users have to it. For EXT2, an inode can describe one of file, directory, symbolic

link, block device, character device or FIFO.

Owner Information
The user and group identifiers of the owners of this file or directory. This allows the file system to correctly allow the right sort of accesses,

Size
The size of the file in bytes,

Timestamps
The time that the inode was created and the last time that it was modified,

Datablocks
Pointers to the blocks that contain the data that this inode is describing. The first twelve are pointers to the physical blocks containing the data described by this inode and the last three pointers contain more and more levels of indirection. For example, the double indirect blocks pointer points at a block of pointers to blocks of pointers to data blocks. This means that files less than or equal to twelve data blocks in length are more quickly accessed than larger files.

You should note that EXT2 inodes can describe special device files. These are not real files but handles that programs can use to access devices. All of the device files in `/dev` are there to allow programs to access Linux's devices. For example the mount program takes as an argument the device file that it wishes to mount.

The EXT2 Superblock

The Superblock contains a description of the basic size and shape of this file system. The information within it allows the file system manager to use and maintain the file system. Usually only the Superblock in Block Group 0 is read when the file system is mounted but each Block Group contains a duplicate copy in case of file system corruption. Amongst other information it holds the:

Magic Number
This allows the mounting software to check that this is indeed the Superblock for an EXT2 file system. For the current version of `EXT2` this is *0xEF53*.

Revision Level
The major and minor revision levels allow the mounting code to determine whether or not this file system supports features that are only available in particular revisions of the file system. There are also feature compatibility fields which help the mounting code to determine which new features can safely be used on this file system,

Mount Count and Maximum Mount Count
Together these allow the system to determine if the file system should be fully checked. The mount count is incremented each time the file system is mounted and when it equals the maximum mount count the warning message ``maximal mount count reached, running e2fsck is recommended'' is displayed,

Block Group Number
The Block Group number that holds this copy of the Superblock,

Block Size
The size of the block for this file system in bytes, for example 1024 bytes,

Blocks per Group
The number of blocks in a group. Like the block size this is fixed when the file system is created,

Free Blocks
The number of free blocks in the file system,
Free Inodes
The number of free Inodes in the file system,
First Inode
This is the inode number of the first inode in the file system. The first inode in an EXT2 root file system would be the directory entry for the '/' directory.

The EXT2 Group Descriptor

Each Block Group has a data structure describing it. Like the Superblock, all the group descriptors for all of the Block Groups are duplicated in each Block Group in case of file system corruption.

Each Group Descriptor contains the following information:

Blocks Bitmap
The block number of the block allocation bitmap for this Block Group. This is used during block allocation and deallocation,
Inode Bitmap
The block number of the inode allocation bitmap for this Block Group. This is used during inode allocation and deallocation,
Inode Table
The block number of the starting block for the inode table for this Block Group. Each inode is represented by the EXT2 inode data structure described below.
Free blocks count, Free Inodes count, Used directory count

The group descriptors are placed on after another and together they make the group descriptor table. Each Blocks Group contains the entire table of group descriptors after its copy of the Superblock. Only the first copy (in Block Group 0) is actually used by the EXT2 file system. The other copies are there, like the copies of the Superblock, in case the main copy is corrupted.

EXT2 Directories

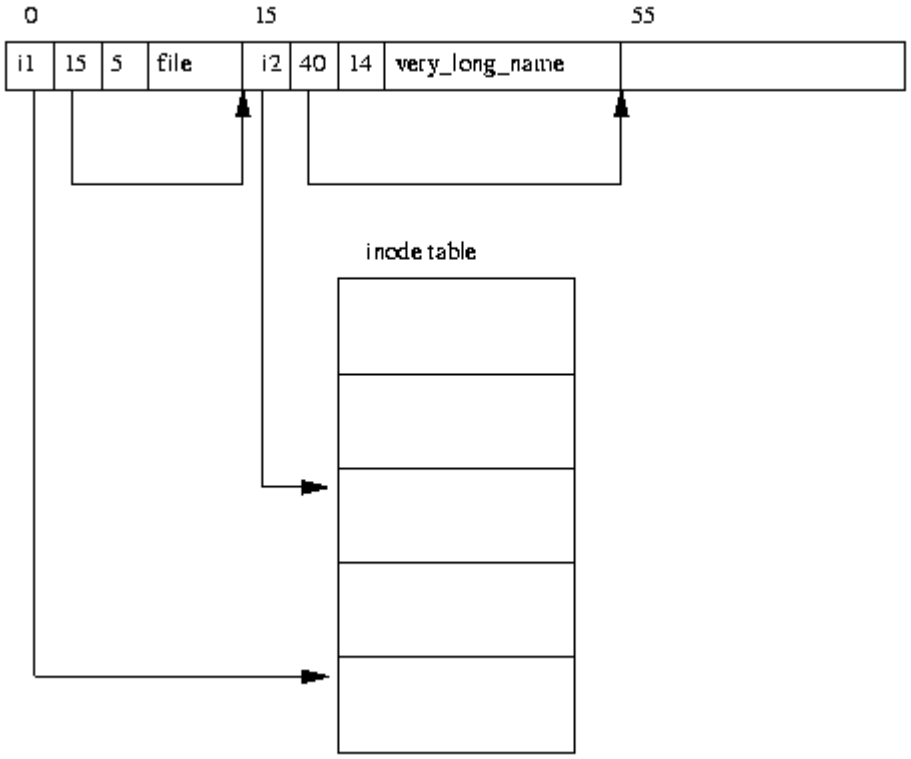


Figure 9.3: EXT2 Directory

In the EXT2 file system, directories are special files that are used to create and hold access paths to the files in the file system. Figure 9.3 shows the layout of a directory entry in memory.

A directory file is a list of directory entries, each one containing the following information:

inode
The inode for this directory entry. This is an index into the array of inodes held in the Inode Table of the Block Group. In figure 9.3, the directory entry for the file called `file` has a reference to inode number `i1`,

name length
The length of this directory entry in bytes,

name
The name of this directory entry.

The first two entries for every directory are always the standard ``.'' and ``..'' entries meaning ``this directory'' and ``the parent directory'' respectively.

Finding a File in an EXT2 File System

A Linux filename has the same format as all Unix ™ filenames have. It is a series of directory names separated by forward slashes (``/'') and ending in the file's name. One example filename would be `/home/rusling/.cshrc` where `/home` and `/rusling` are directory names and the file's name is `.cshrc`. Like all other Unix ™ systems, Linux does not

care about the format of the filename itself; it can be any length and consist of any of the printable characters. To find the inode representing this file within an `EXT2` file system the system must parse the filename a directory at a time until we get to the file itself.

The first inode we need is the inode for the root of the file system and we find its number in the file system's superblock. To read an EXT2 inode we must look for it in the inode table of the appropriate Block Group. If, for example, the root inode number is 42, then we need the 42nd inode from the inode table of Block Group 0. The root inode is for an EXT2 directory, in other words the mode of the root inode describes it as a directory and it's data blocks contain EXT2 directory entries.

`home` is just one of the many directory entries and this directory entry gives us the number of the inode describing the `/home` directory. We have to read this directory (by first reading its inode and then reading the directory entries from the data blocks described by its inode) to find the `rusling` entry which gives us the number of the inode describing the `/home/rusling` directory. Finally we read the directory entries pointed at by the inode describing the `/home/rusling` directory to find the inode number of the `.cshrc` file and from this we get the data blocks containing the information in the file.

Kernel dùng chức năng ***bmap( )****(Xem ở mã nguồn Linux)* thực hiện biến đổi *file offset* thành block đĩa vật lí.

input: *(1) inode*
(2) byte offset
output: *(1) số của khối đĩa (block number in file system)*
(2) con trỏ trỏ vào khối đĩa (byte offset into block)
(3) số byte cần thực hiện truy nhập trong jhôid đĩa
(4) đọc trước khối đĩa cho lần đọc sau

Ví dụ: biến đổi *byte offset* thành số của block đĩa.

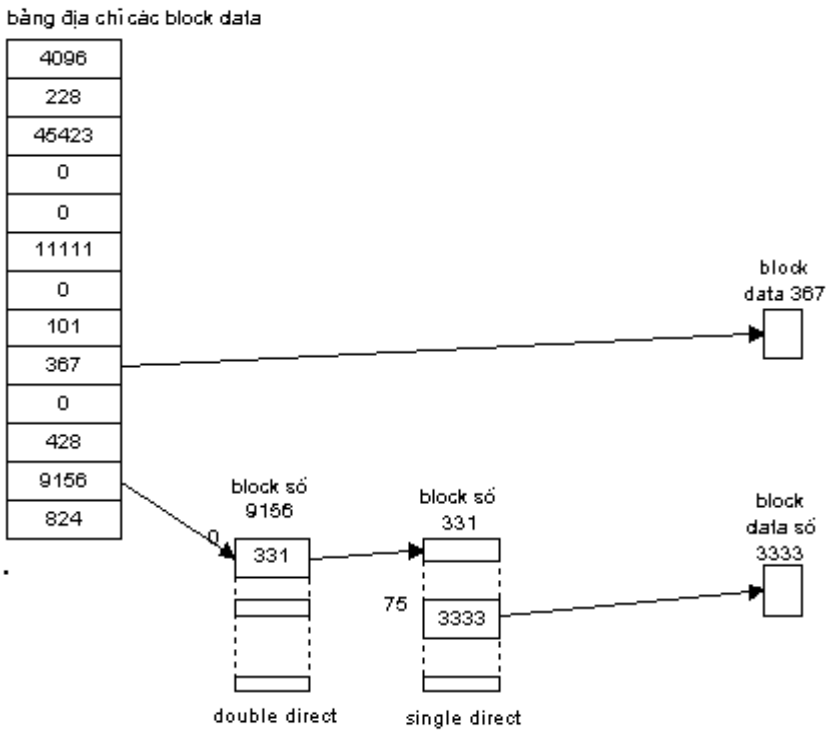
Hãy xét cách bố trí các block cho tệp ở hình sau:

Giả định rằng một block đĩa có 1024 byte. Nếu một TT muốn tìm byte thứ 9000, kernel tính thấy byte này nằm ở block trực tiếp tại đầu vào số 8 (đếm từ đầu và đánh số từ 0) trong bảng địa chỉ các block của tệp ở đó có địa chỉ của block số 367. Với 9 block cho 1024x9=9216 byte, từ đó tính ra byte thứ 808 trong block này là byte 9000 của tệp (8 block trước đó cho 8x1024=8192 byte, 9000-8192=808). Nếu TT tìm byte thứ 350.000 trong tệp, kernel tính ra rằng phải đọc block gián tiếp đôi mà địa chỉ của block là 9156. Bởi vì 1 block gián tiếp cho 256 địa chỉ các block, vậy byte đầu tiên qua block gián tiếp đôi là byte 272.384 (256K + 10K); byte 350.000 là byte thứ 77.616 của block gián tiếp đôi. Vì mỗi một block gián tiếp đơn cho 256 K bytes, byte 350.000 phải ở block gián tiếp đơn thứ 0 của block gián tiếp đôI, ví dụ đó là block số 331 (đầu vào thứ 0 của block 9156). Block 331 cho 256 block trực tiếp mỗi block 1K, vậy byte số 77.616 của block trực tiếp sẽ trong block trực tiếp thứ 75 (giả định số của block này là 3333) của block gián tiếp đơn. Cuối cùng khi đọc block 3333, kernel tìm thấy byte thứ 816 là byte 350.000 của tệp.

Xem xét hình một cách chi tiết hơn, có một vài đầu vào trong bảng địa chỉ các block là 0, có nghĩa rằng các đầu vào của các block logic này không chứa data. Điều này xảy ra khi không

có TT nào ghi data vào tệp ở bất kì vị trí nào tương ứng với các block đó, do vậy số của các block không đổi và là các giá trị ban đầu (initial). TT có thể tạo ra các block như vậy trong bảng bằng cách dùng GHT *lseek()* và *write()*.

Sự biến đổi byte offset lớn, đặc biệt khi phải qui chiếu qua gián tiếp ba (triple) là một qui trình hết sức rắc rối đòi hỏi kernel phải truy nhập thêm ba block đĩa ngay cả khi các block đã có trong cache thì phép thực hiện vẫn rất đắt giá vì kernel sẽ phải yêu cầu buffer rất nhiều lần và sẽ phải đi ngủ do buffer bị khoá. Hiệu quả thực tiễn của thuật toán này phụ thuộc vào tần xuất truy nhập các tệp lớn hay nhỏ trên hệ thống. Hãy thử xét tới khả năng tăng độ lớn của 1 block đĩa (4 K, 8K hay 12 K/block), liệu có gì khả quan hơn cũng như hiệu suất sử dụng đĩa có tồi hơn ? Hiện tượng phân đoạn có tăng lên ?



4. Tệp thư mục

Thư mục là một tệp mà data của nó là trình tự các đầu vào mà mỗi đầu vào bao gồm một số của inode và tên của tệp chứa trong thư mục. Một đường dẫn là một xâu kí tự kết thúc trống (NULL), được phân chia ra thành các thành phần ngăn bởi kí tự "/". Mỗi thành phần, trừ thành phần cuối cùng, phải là tên của một thư mục, và thành phần cuối cùng không phải là tệp thư mục. UNIX System V hạn chế tên của một thành phần chỉ tới 14 đến 256 kí tự và 2 bytes cho số của inode. Vậy ví dụ tên tệp là 14 kí tự, thì một đầu vào thư mục có 16 bytes, một tệp thư mục sẽ như sau:

Byte Offset	Inode number (2 bytes)		File name
0	83	.	(thư mục hiện tại)
16	2	..	(thư mục bố)
32	1789		init
48	1276		fsck
.	.	.	
.	.	.	
.	.	.	
224	0		crash
240	95		mkfs
.	.	.	

Mỗi một thư mục chứa các tên tệp "." (dot) và ".." (dot-dot) với số của inode là inode của thư mục đó và thư mục bố (trên một mức). Inode số "." tại offset 0 và trị số của nó là 83. Tương tự của inode ".." có offset là vị trí thứ 16 và trị số là 2. Đầu vào của thư mục có thể "rỗng" và được biểu thị bởi số inode = 0. Ví dụ đầu vào 224 "rỗng" dù đã một lần chứa tệp có tên "crash". Chương trình tạo hệ thống tệp khởi động FS sao cho "." và ".." của thư mục ***root*** có số inode của FS.

Kernel cất data của thư mục cũng giống như cho các tệp thường, sử dụng cấu trúc inode và các cách cấp trực tiếp, gián tiếp của các block. TT đọc tệp thư mục cùng cách thức như đọc tệp thường, nhưng kernel giành đặc quyền để ghi thư mục, do đó đảm bảo được sự chuẩn xác

của thư mục. Quyền truy nhập thư mục có ý nghĩa như sau:

- quyền đọc (*read*) thư mục cho phép TT đọc thư mục;
- quyền ghi (*write*) thư mục cho phép TT tạo các đầu vào mới hay xoá đầu vào cũ (*creat, mknod, link, unlink*); bằng cách đó thay đổi nội dung thư mục.
- Quyền thực hiện (*excute*) cho phép TT thực hiện tìm kiếm tên tệp trong thư mục.

5. Biến đổi tư đường dẫn thành inode

Truy nhập tệp khởi đầu bằng đường dẫn. Vì bên trong kernel làm việc với inode, chứ không bằng tên tệp qua đường dẫn, nên kernel phải chuyển đổi từ đường dẫn thành inode để truy nhập tới tệp. Thuật toán n*amei()* làm phân tích mỗi thành phần trong đường dẫn một lần, biến đổi mỗi thành phần đó thành inode trên cơ sở tên của nó và thư mục đang tìm kiếm, cuối cùng trả lại inode.

Nhắc lại trong chương trước là mỗi một TT được kết hợp với một thư mục hiện tại (mà TT thường trú ở đó). Miền bộ nhớ *ufiarea* chứa một con trỏ trỏ tới inode là thư mục hiện hành. Thư mục hiện hành của TT đầu tiên trong hệ thống, TT số 0, là thư mục *root*. Thư mục hiện hành của mỗi TT khác bắt đầu từ thư mục bố hiện hành vào lúc TT được tạo ra. TT thay đổi thư mục bằng thực hiện GHT *chdir* (đổi thư mục). Việc tìm đường dẫn xuất phát từ thư mục hiện hành của TT trừ phi có dấu "/", cho biết việc tìm thư mục phải bắt đầu từ root. Trong các trường hợp khác kernel dễ dàng tìm ra inode mà ở đó việc tìm đường dẫn bắt đầu: thư mục hiện hành có trong *ufiarea* của TT, và inode của *root* có trong biến tổng thể.

namei() */*convert path name to inode*/*

input: *đường dẫn (path name)*

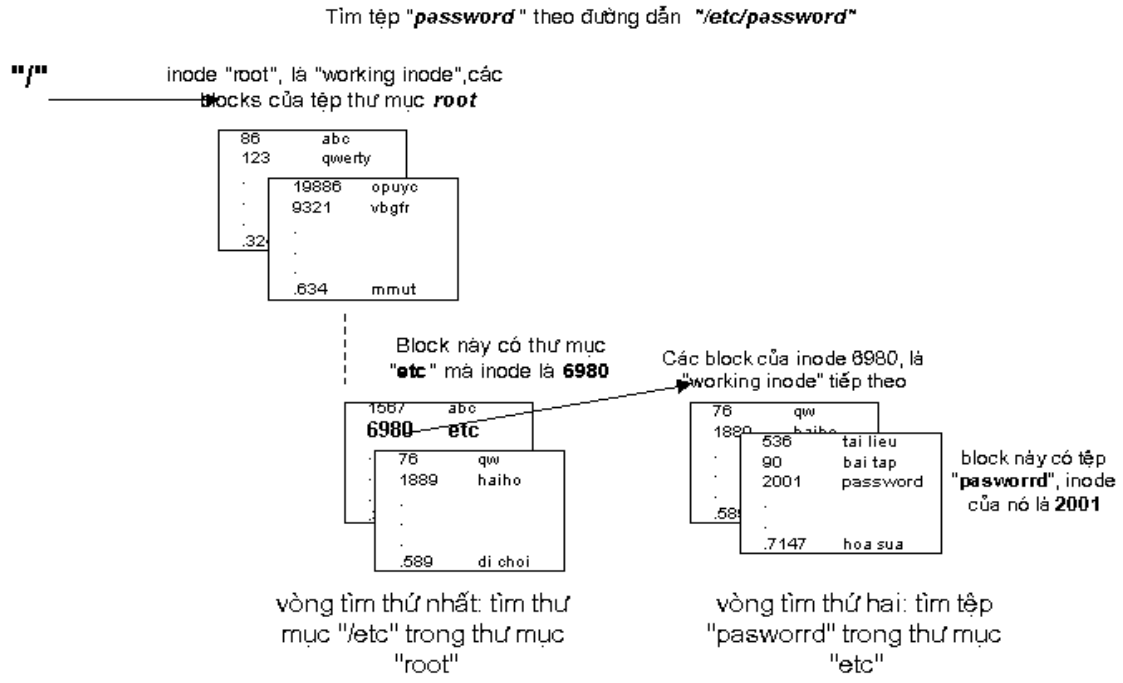
ouput: *inode đã tìm được (locked inode)*

Kernel thực hiện tìm tuyến tính một tệp thư mục, kết hợp với *working inode (thư mục tạm thời)*, thử sự tương xứng của thành phần tên đường dẫn với một đầu vào của tệp thư mục (xem lại cấu trúc của tệp thư mục). Bắt đầu với byte offset = 0 trong thư mục, chuyển đổi offset này thành block đĩa tương ứng (*bmap()*) và đọc block đó (*bread()*). Kernel xử lí nội dung của block này như tuần tự các đầu vào của thư mục, tìm từng thành phần trong đường dẫn bằng cách so sánh các đầu vào, nếu tìm được, kernel lấy ra số của inode ứng với tên đường dẫn đó, giải phóng block (*brelse()*) và working inode cũ (*iput()*), cấp phát một inode của thành phần vừa tìm được (*iget()*), và inode mới này trở thành *working inode*. Nếu không có kết quả ở block này, kernel giải phóng block, điều chỉnh lại offset bằng số byte trong block (lấy byte offset tiếp theo), biến đổi offset mới thành số của block đĩa (*bmap()*), và đọc block đó. Kernel lặp lại chu trình này cho tới khi tìm ra inode tương ứng với thành phần (tệp) yêu cầu trong đường dẫn cho tới hết các đầu vào của thư mục.

Ví dụ: Tìm tệp passwd trong thư mục /etc: "/etc/passwd".

Khi bắt đầu phân tích tên tệp, kernel gặp "/" và nhận đi lấy inode của root, inode này trở thành *working inode*. Kernel đi thu thập xâu "etc": sau khi kiểm tra đúng inode này là của thư mục "/" và TT có đủ quyền hạn truy nhập, kernel tìm tệp có tên "etc": Kernel truy nhập data trong thư mục gốc (root) hết block này tới block khác, tìm từng đầu vào của từng block cho tới khi định vị được một đầu vào "etc". Khi tìm được đầu vào này, kernel giải phóng inode root (*iput()*), lấy inode ứng với "etc" (*iget()*), sau khi biết đích xác "etc" là tệp thư mục,

kernel tìm trong các block của "etc" để đến tệp "passwd", lấy từ đó inode ứng với "passwd".



6. Cấp một inode cho một tệp mới tạo

Kernel dùng *iget()* để định vị inode đã biết (với số của inode đã được xác định trước đó trong thuật toán *namei()*). Một thuật toán khác *ialloc()* gán một inode cho một tệp mới tạo. Hệ thống tệp có một danh sách tuyến tính các inode. Một inode là *free* nếu trường chứa kiểu tệp (*type*) là 0. Khi một TT cần inode mới, kernel có thể tìm trong danh sách inode một inode free, tuy nhiên cách làm này có thể rất đắt đòi hỏi ít nhất một thao tác đọc đĩa cho mỗi inode. Để cải thiện, *super block* chứa một mảng ẩn số lượng các inode free trong FS. Hãy theo dõi cho thuật toán *ialloc()*, tìm và cấp một inode mới:

ialloc() */*allocate inode*/*

input: *FS*

output: *inode mới (locked inode)*

Thuật toán giải phóng inode đơn giản hơn.

ifree() */*inode free*/*

input: *FS inode number*

output: *none*

7. Cấp phát các block đĩa

Khi một TT ghi data lên đĩa, kernel sẽ cấp các block cho nó theo cơ chế direct và indirect như đã nói. Super block có một mảng gồm nhiều block đĩa dùng để ẩn các *số của các*

block đĩa chưa dùng (*free disk block)* của FS. Tiện ích *mkfs()* tổ chức các khối dữ liệu (*data block)* của FS vào một danh sách liên kết, mà mỗi một liên kết của danh sách là *một block đĩa*, block này chứa một *mảng số hiệu của các free disk block* và một đầu vào của mảng cho số của block tiếp theo của danh sách liên kết. Hình sau là ví dụ về danh sách liên kết: block đầu tiên là danh sách block chưa dùng của super block (*super block free list),* với các đầu vào là số của các block sẽ dùng cho cấp phát, và 1 đầu vào có số của block liên kết tiếp theo (block 109, 211, 310, 409…).

Khi kernel muốn cấp phát một block, xem thuật toán *alloc()*, kernel sẽ lấy block trong *danh sách các free block sẵn có* của *super block list*, và một khi đã dùng block không thể tái cấp cho tới khi nó trở lại là free. Nếu block đã cấp là block cuối cùng có trong mảng cache, kernel sẽ xử lí block đó như là con trỏ (109) trỏ vào block (109) chứa danh sách các free block khác. Kernel đọc block đó, nhập vào mảng Super block với danh sách mới số của các block chưa cấp, sau đó tiếp tục sử dụng số của block gốc, cấp buffer cho block, xoá data của block (điền zero vào). Block đĩa lúc này đã cấp phát cho tệp, kernel cũng đã có buffer để làm việc. Chương trình sẽ thông báo lỗi nếu FS không còn có free block.

alloc() */*FS block allocation*/*

input: FS (number)

outout: buffer for new block

Nếu TT ghi một khối lượng data lớn, TT sẽ lặp lại yêu cầu xin cấp block, trong khi kernel chỉ thực hiện cấp mỗi lần một block. Chương trình *mkfs()* sẽ tổ chức danh sách liên kết của các block sao cho gần với nhau, để giảm đi thời gian tìm kiếm trên đĩa khi TT đọc tệp tuần tự. Hình dưới mô tả số hiệu của các block với số cách quãng (*interlive code = 3*) đều đặn dựa trên cơ sở tốc độ quay của đĩa. Điều lưu ý là thứ tự sắp xếp nói trên sẽ không còn khi tần xuất sử dụng block đĩa cao (cấp phát/ giải phóng) và quá trình cập nhật lại danh sách có tính ngẫu nhiên và kernel không thực hiện sắp xếp lại thứ tự gốc.

Thuật toán giải phóng block *free()*, ngược lại với cấp phát block. Nếu *super block list* không đầy, số của block vừa được giải phóng sẽ đặt vào danh sách đó. Nhưng nếu không còn chỗ (đã đầy) thì block vừa được giải phóng sẽ thành *block liên kết*; kernel ghi *super block list* vào block đĩa đó và ghi nội dung của block này lên đĩa, sau đó đặt số của block mới giải phóng vào *super block list*. Số của block này chỉ là thành viên của danh sách.

Thuật toán gán/giải phóng inode tệp và block đĩa là tương tự ở chỗ kernel dùng super block như là một cache của các chỉ số cho nguồn tài nguyên chưa dùng (free), số của các block, số của các inode. Kernel duy trì danh sách liên kết của số hiệu các block sao cho mỗi số chưa dùng hiện diện trong một thành phần của danh sách. Điều này không có đối với free inode. Sở dĩ có sự khác nhau do các nguyên nhân dưới đây:

1. Kernel có thể biết inode là free bằng cách tham khảo trường Type, nếu = 0 thì inode free, nhưng đối với block đĩa thì không có cơ chế gì để nhận biết tương tự. Do vậy kernel cần có phương pháp khác để biết block là free, và đó là *linked list*fidanh sách liên kết.
2. Các block đĩa bản thân đã là dùng danh sách liên kết: một block đĩa có thể chứa một danh sách lớn số hiệu của các block chưa cấp phát. Với cấu trúc dữ liệu lớn cho mỗi inode, không thể thực hiện như cho block đĩa.
3. Việc khai thác block đĩa (đọc/ ghi nội dung tệp) là thường xuyên hơn khai thác inode (tạo, mở, ghi tệp), điều đó có nghĩa truy nhập block đĩa gay cấn hơn tìm inode.

8. Các kiểu tệp khác

Unix hỗ trợ hai loại tệp khác là: ***pipe*** và các tệp đặc biệt (***special files***).

Pipe còn gọi là *fifo (first-in-first-out)*, khác với tệp thường ở chỗ dữ liệu của tệp là có tính chuyển tiếp ngay: một khi data đã đọc từ pipe thì không thể đọc lại lần nữa, đồng thời data đọc theo thứ tự mà data đã được ghi vào hệ thống không cho phép làm chệch thứ tự đó. Kernel cất data cũng cùng một cách thức như tệp thường chỉ khác là chỉ dùng các block đĩa kiểu cấp trực tiếp.

Special file là các tệp kiểu khối, kiểu kí tự, các tệp loại này xác định các thiết bị. Vì vậy inode của tệp đặc biệt không qui chiếu tới dữ liệu, mà chứa hai số qui chiếu tới thiết bị gọi là *major number* và *minor number*.

major number cho kiểu thiết bị (ví dụ:terminal, disk...);

minor number cho biết số đơn vị của thiết bị đó.

Xem ví dụ phần trước về đĩa cứng.

9. Tóm tắt và bài tập

Tóm tắt

- ***Inode*** là một cấu trúc dữ liệu mô tả thuộc tính của tệp kể cả hình thức tổ chức dữ liệu của tệp trên đĩa. Có hai phiên bản của inode: bản disk inode lưu các thông tin của tệp khi tệp không đưa vào sử dụng; và phiên bản in-core inode cập nhật liên tục mọi sự thay đổi của tệp khi tệp đang được sử dụng. Các thuật toán sau thao tác inode:
- ***ialloc(), ifree(),*** kiểm soát việc cấp inode cho tệp khi thực hiện GHT *creat(), mknod(), pipe(), unlink();*
- ***iget(), iput()*** kiểm soát cấp *in - core inode* khi TT truy nhập tệp;
- ***bmap()*** định vị disk block cho tệp theo *offset* tệp đầu vào;
- *Thư mục* là loại tệp cho sự tương quan giữa các thành phần tên (*pathname*) với inode của tệp.
- Thuật toán ***namei()*** biến đổi pathname thao tác bởi TT thành inode mà kernel sẽ dùng bên trong các quá trình.
- ***alloc()*** và ***free()*** kernel dùng để kiểm soát cấp và giải phóng block đĩa cho tệp.
- Các cấu trúc dữ kiệu hệ thống đã xét gồm:

- *linked list* danh sách liên kết quản lí disk block,
- *hash queue* (hàng băm) dùng trong tổ chức buffer cache,
- *linear array* các mảng tổ chức các danh sách số block đĩa và
- các thuật toán khác hỗ trợ làm đơn giản các thao tác data hệ thống.

Tính phức tạp sẽ gia tăng bởi sự tương tác giữa các thuật toán và mã chương trình cũng đã cho thấy có những vấn đề về thời gian. Các thuật toán ở đây chưa được trau chuốt tỉ mỉ mà chỉ để minh họa cho đơn giản việc thiết kế hệ thống.

Các thuật toán mô tả là của bên trong hệ thống và dành cho kernel thực hiện và là ẩn đối với

user. Qui chiếu vào hình mô tả nhân hệ thống, các thuật toán chương này nằm ở nửa phần dưới của khối FS. Chương tiếp theo sẽ trình bày các GHT, cung cấp giao diện cho người dùng (lập trình hệ thống ứng dụng) ghép nối với FS. Các GHT này sẽ kích hoạt các thuật toán bên trong vừa mô tả.

Bài tập

1. Trong ngôn ngữ qui ước chỉ số của mảng bắt đầu từ 0, tại sao số của inode lại bắt đầu từ 1?
2. Trong *iget()*, TT đi ngủ khi thấy inode đã khoá (*locked*), khi được đánh thức, thì TT phải bắt đầu lại vòng lặp từ đầu?
3. Mô tả thuật toán cập nhật một disk inode với đầu vào là *in - core inode*.
4. Thuật toán *iget()* và *iput()* không đòi hỏi phải nâng mức ưu tiên của xử lí để ngăn chặn ngắt, tại sao như vậy?
5. Unix System V cho phép độ dài tối đa cho một thành phần của *pathname* là 14 kí tự. *Namei()* sẽ cắt đi các kí tự dài hơn trong thành phần của *pathname*. Phải thiết kế lại FS và thuật toán *namei()* như thế nào để cho phép có thể có độ dài tên tùy ý?
6. (*) Hãy xem *namei()*, trong quá trình tìm, kernel kiểm tra thấy working inode hiện tại là một thư mục. Có thể cho một TT khác hủy bỏ (*remove*) thư mục đó (bằng lệnh *unlink*) ? Làm gì để kernel ngăn chặn được điều đó?
7. (*) Hãy thiết kế cấu trúc thư mục để có thể cải thiện hiệu quả tìm tên bằng cách không dùng phương pháp tuyến tính. Hãy nghiên cứu tới hai kĩ thuật: *hash* và *n - aray tree.*
8. (*) Thiết kế sơ đồ để giảm bớt số lần phải tìm thư mục bằng cách ẩn các tên đã thường xuên dùng đến.
9. Super block là một block đĩa và bên cạnh danh sách *free block list* còn có các thông tin khác, cho nên danh sách này không thể chứa được nhiều số của các block như trong block đĩa của danh sách liên kết của các block chưa dùng (*free*). Số lượng tối ưu của số các block là bao nhiêu có thể chứa trong block trên danh sách liên kết?
10. (*) Hãy thảo luận các dùng ánh xạ bit (bit map) để kiểm soát các free disk block thay cho cách dùng *linked list* của các block. Ưu và nhược điểm của phương pháp đó?

B. Gọi Hệ Thống thao tác tệp (*System call for FS*)

(Xem lại mô tả của GHT ở phần đầu tài liệu)

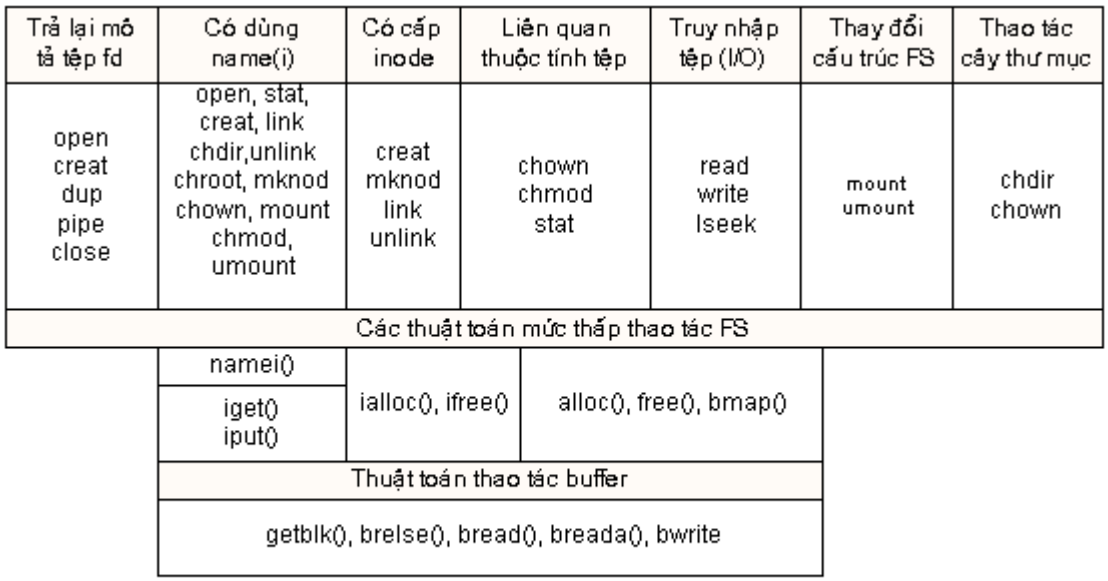
Chương trước đã mô tả cấu trúc dữ liệu bên trong cho hệ thống tệp (FS) và các thuật toán thao tác các cấu trúc đó. Chương này bàn đến các hàm hệ thống được gọi để truy nhập tệp. Điển hình cho các GHT thuộc loại này có:

- truy nhập tệp đã tồn tại trong FS: *open(), read(), write(), lseek(), close();*
- tạo tệp mới: *creat(), mknod();*
- thao tác các inode: *chdir, chroot(), chown(), stat(), fstat();*
- GHT loại nâng cao: *pipe(), dup();*

- mở rộng FS nhìn thấy được cho user: *mount(), umount();*
- thay đổi cấu trúc cây FS: *link(), ulink();*

Chương sẽ trình bày các tổng quan hệ thống khác như: quản trị, bảo trì FS, đồng thời giới thiệu ba cấu trúc dữ liệu cơ bản của kernel dùng trong quản lí tệp:

- ***File table*** (bảng các tệp) với mỗi đầu vào trong bảng cấp cho một tệp mở trong FS;
- ***User file descriptor (fd) table***: bảng mô tả các tệp mở của mỗi TT của mỗi user, với mỗi đầu vào trong bảng *fd*, dành cho một tệp;
- ***Mount table:*** bảng "ghép" một FS khác vào cấu trúc FS gốc, mở rộng FS gốc.



GHT cho FS và mối liên quan tới các thuật toán khác

Hình trên cho thấy mối quan hệ giữa các GHT và các thuật toán đã nói trong chương trước. GHT có thể phân ra làm nhiều hạng cho dù có một số GHT cùng xuất hiện nhiều hơn là trong một hạng:

- GHT trả lại *fd* để dùng trong GHT khác;
- GHT dùng *namei()* dể phân tích path name;
- GHT gán và giải phóng inode dùng *ialloc() và ifree();*
- GHT đặt và đổi thuộc tính của tệp;
- GHT thực hiện các I/O cho một TT hay từ một TT, dùng *alloc() và free()* các thuật toán cấp buffer;
- GHT thay đổi cấc trúc của FS;
- GHT cho phép một TT thay đổi cách nhìn cây FS.

1. open

GHT *open()* là bước đầu tiên một TT thực hiện khi muốn truy nhập tệp. Cú pháp như sau:

fd = open(*pathname, flags, modes*)

Các đối trong đó là: *patname* là đường dẫn tên tệp;

flags kiểu mở tệp, ví dụ mở đọc, mở ghi;

modes là quyền truy nhập tệp;

Hàm trả lại: *fd* là giá trị nguyên, còn gọi là mô tả tệp của user (*user file descriptor*).

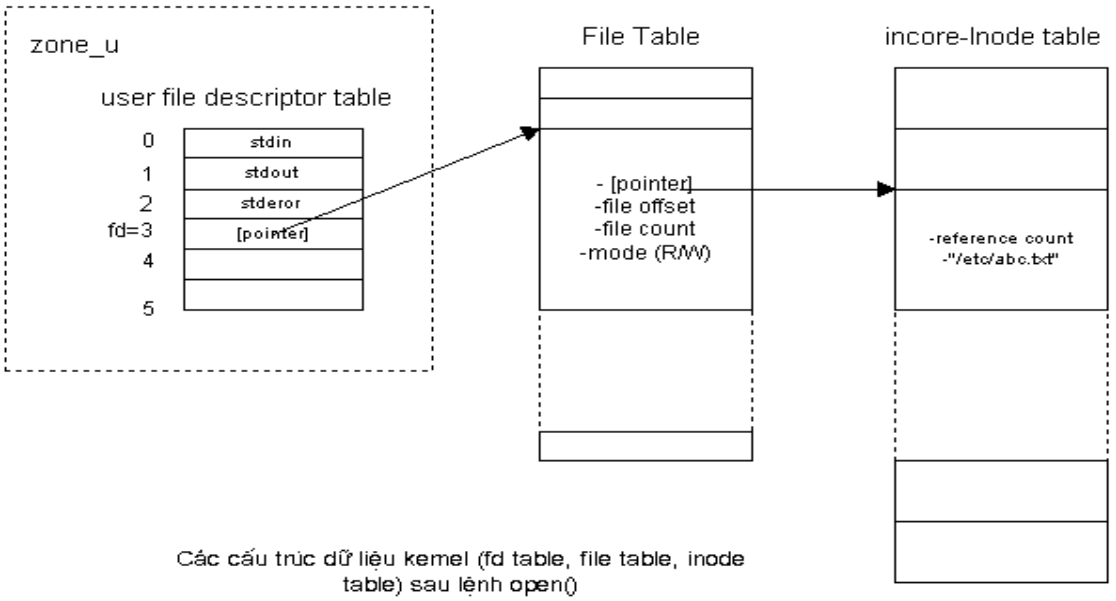
Các thao tác trên tệp như đọc, ghi, tìm, sao chép, đặt các thông số I/O, xác định trạng thái và đóng tệp đều sử dụng *fd* trả lại bởi *open()*.

Thuật toán *open()* như sau:

open() */*open file*/*

input: *đường dẫn/tên tệp, kiểu thao tác tệp (R/W),quyền truy nhập*

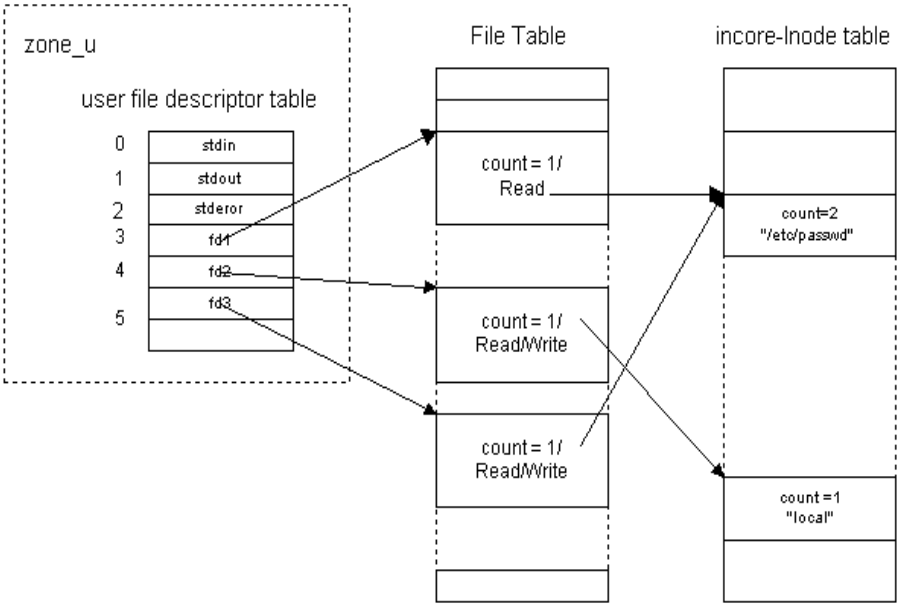
output: *con trỏ mô tả tệp (file descriptor)*



Các cấu trúc dữ liệu kernel (fd table, file table, inode table) sau lệnh open()

Các cấu trúc dữ liệu hệ thống gồm: File table, là cấu trúc tổng thể của kernel, dùng để quản lí các thao tác trên một tệp mở bởi *open()*, Inode Table: nơi để các thông tin của inde trong bộ nhớ (*in - core inode*). *File descriptor table* là của riêng từng tiến trình.

GHT *namei()* dùng để tìm file (*inode, in - core inode*) với các thông số của đầu vào, kernel kiểm tra quyền truy nhập tệp sau khi xác định *in - core inode* của tệp, cấp một đầu vào trong *File Table* cho tệp, đầu vào này chứa con trỏ trỏ vào *in - core inode* của tệp. Kernel còn khởi động một trường khác để chứa con trỏ tệp (*byte offset*) để đọc hay ghi tệp với giá trị ban đầu bằng 0, điều đó có nghĩa bắt đầu từ đầu tệp. Số đếm qui chiếu tệp (*reference count*) đặt = 1 (một TT truy nhập tệp).Nừu chế độ truy nhập là ghi-thêm vào tệp (*write - append*), offset đặt bằng độ dài tệp đã có. Kernel cấp một đầu vào trong *user file descriptor table* trong *u area* của TT, ghi nhận lại chỉ số đầu vào đó và chỉ số đó là *file descriptor fd* trả lại cho user. Nội dung tại chỉ số này là con trỏ trỏ vào *File Table*.



Giả sử TT thực hiện mã sau đây: mở tệp "/etc/passwd" hai lần, một chỉ đọc (*read - only*) và lần kia chỉ ghi (*write - only*) và tệp "local" một lần cho đọc-ghi.

fd1 = open ("etc/passwd", OfiRDONLY);

fd2 = open ("local", OfiRDWR);

fd3 = open ("etc/passwd", OfiWRONLY);

Hình chỉ mối quan hệ giữa các bảng nói trên. Mỗi một *open()* trả lại một *fd* cho TT gọi và đầu vào tương ứng *fd* trong *user file descriptor (ufd)* trỏ tới đầu vào duy nhất trong File table. Ngay cả khi một file gọi nhiều lần thì đều có các *fd* tương ứng trong *ufd* với mỗi lần gọi và các đầu vào tương ứng với mỗi fd trong file table. Các đầu vào khác nhau đó tuy nhiên đều trỏ tới một *in - core inode* trong bảng. *Refrence count* của *in - core inode* sẽ cho biết có bao nhiêu sự mở đồng thời. Ví dụ fd1 và fd3 cùng truy nhập tệp "/etc/passwd", số đếm này = 2. TT có thể *đọc* hay *ghi* tệp nhưng việc đó là phân biệt theo các fd khác nhau: fd1: mở tệp chỉ đọc, fd2: đọc/ghi tệp, fd3: mở tệp chỉ để ghi. Kernel ghi nhận khả năng đọc hay ghi tệp của TT trong mỗi đầu vào ở *file table* vào lúc gọi *open()*. Giả sử có TT thứ hai thực hiện mã sau:

fd1 = open ("etc/passwd", OfiRDONLY);

fd2 = open ("private", OfiRDONLY);

Ta có hình dưới phản ánh tình huống này, lúc này tệp "etc/passwd" lại mở một lần nữa bởi TT thứ hai (B), số đếm qui chiếu của tệp này tăng thêm 1 và tổng số có 3 lần cùng lúc mở tệp. Tóm lại *cứ mỗi một open sẽ được cấp một đầu vào duy nhất trong ufd và một đầu vào duy nhất trong file table, và với một tệp dù mở bao nhiêu lần thì cũng chỉ có nhiều nhất một đầu vào trong in - core inode table.*

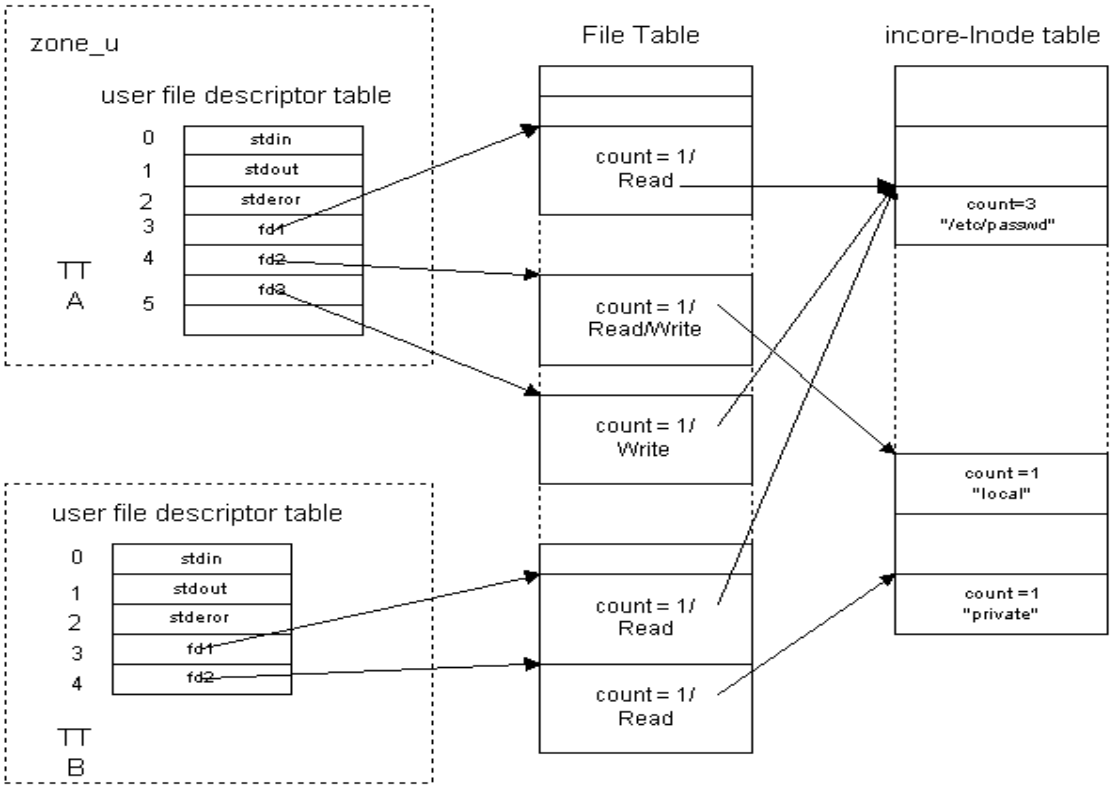
Có thể hình dung rằng đầu vào của bảng *user file descriptor* (*ufd*) có thể chứa con trỏ tệp để định vị đọc/ ghi tiếp theo và con trỏ trỏ trực tiếp tới *in - core inode* của tệp và loại trừ sự cần thiết của *file table*. Nhưng các ví dụ trên cho thấy mối quan hệ một - một giữa các đầu vào của *ufd* và *file table*, Thompsons cho biết rằng sử dụng file table riêng như trên cho phép

chia sẻ con trỏ tệp giữa nhiều *ufd*: *file offset* độc lập cho mỗi *fd*, trong khi file mở chung cho nhiều TT. Các hàm *dup()* và *fork()* thao tác các cấu trúc dữ liệu để cho phép chia sẻ con trỏ tệp này.

Ba con trỏ tệp đầu tiên trong *ufd* fd0, fd1, fd2 là các con trỏ chuẩn, trong đó:

- *fd0* cho tệp *standard input* (thường là thiết bị bàn phím, terminal) để đọc đầu vào;
- *fd1* cho tệp *standard output* (thường là thiết bị màn hình) để ghi đầu ra;
- *fd2* cho tệp *standard error*, đầu ra ghi thông báo lỗi là các thông điệp.

Hệ thống không nói gì rằng đó là các mô tả kiểu tệp đặc biệt, user có thể chọn fd 4, 5, 6 và 11 cho các tệp đặc biệt, nhưng tôn trọng qui ước của C tính từ 0 vẫn là tự nhiên hơn và thích ứng với qui ước này sẽ dễ dàng để giao tiếp với ***pipe***.



2. read

Cú pháp:

number = read (fd, buffer, count)

fd là mô tả tệp trả lại từ open(),

buffer là địa chỉ của cấu trúc dữ liệu của TT user, nơI sẽ chứa data khi lời GHT hoàn tất,

count là tổng số byte user muốn đọc từ tệp,

number là số byte thực sự đã đọc.

Read()

input: fd (trả lại từ open()), địa chỉ của bộ đệm dữ liệu, số byte cần đọc

output: số byte đọc được

- Kernel lấy từ fd con trỏ trỏ tới đầu vào tương ứng trong file table.
- Kernel chuẩn bị các thông số I/O:
  - *mode:* đọc hoặc ghi đĩa;
  - *count:* đếm số byte để đọc hoặc ghi;
  - *offset:* con trỏ định vị tệp để I/O biết bắt đầu từ đâu;
  - *address:* địa chỉ đích để copy data trong bộ nhớ của kernel hoặc của user;
  - *flag:* cho biết địa chỉ là ở miền user hay kernel;

Sau khi đã có các thông số nói trên, kernel theo con trỏ trong file table để truy nhập *in - core inode*, khóa inode lại trước khi đọc tệp.

Nhân HĐH,đọc tệp cho tới khi thỏa mãn, biến các giá trị của byte offset thành số của block cần đọc như thuật toán *bmap()* đã nói. Sau khi đọc block vào buffer, kernel copy data vào địa chỉ của TT user, cập nhật các thông số I/O mới ở *u area* theo số byte đọc, tăng byte offset và tăng địa chỉ của TT user cho lần đọc tiếp theo, giảm số đếm byte để phù hợp với yêu cầu của TT user...Nếu chưa thoả mãn đọc tệp, chu trình lặp lại các bước. Chu trình kết thúc khi thoả mãn đọc tệp, hoặc không còn data để đọc hay nếu có lỗi trong quá trình thực hiện.

Ví dụ một chương trình đọc tệp:

```
#include <fcntl.h>
main ()
{
        int fd;
        char lilbuff[20], bigbu[1024];
        fd = open("/etc/passwd", OfiRDONLY);
        read(fd, lilbuff,20);
        read(fd, bigbuf, 1024);
        read(fd, lilbuf, 20);
}
```

open() trả lại mô tả tệp gán cho biến fd và dùng để đọc liên tục sau đó. Trong *read()*, kernel kiểm tra các thông số của fd và rằng TT trước đó đã mở tệp để đọc. Các giá trị *lilbuf*, 20, 0 nhớ lại trong *ufi area* của TT là địa chỉ tương ứng với user buffer, số đếm byte phải đọc, byte offset bắt đầu của tệp. Kernel tính ra là byte offset 0 ở block 0 của tệp và tìm thấy số của block thứ 0 trong inode. Kernel đọc block từ đĩa với 1024 byte vào buffer hệ thống và copy chỉ 20 byte vào vùng của user *lilbuf*, giảm số đếm (*count*) byte phảI đọc = 0, việc đọc đã thỏa mãn, trả lại số byte đã đọc là 20. Kernel lập file offset trong file table tới giá trị 20 để chuẩn bị cho lần đọc tiếp theo …

Khi TT gọi *read()*, kernel khoá inode trong suốt thời gian thực hiện read. TT có thể sẽ đi ngủ đợi buffer có data hay móc nối các *indirect block* của inode, và nếu trong khi đó có TT khác

được phép thay đổi data của tệp, thì *read()* sẽ cho một kết quả data không nhất quán. Chính vì vậy mà TT phải khoá inode để tránh sự thay đổi nội dung của tệp do TT khác cùng truy nhập lúc TT ngủ, đợi kết quả đọc tệp.

Kernel có thể *chen ngang* (preempt) một TT đang đọc giữa các lần GHT trong user mode và cho một TT khác chạy. Bởi vì inode được giải khoá vào đoạn cuối GHT, sẽ không có gì ngăn cản các TT khác truy nhập và biến đổi nội dung tệp. Cũng sẽ không công bằng khi TT cứ khoá inode từ lúc mở cho tới lúc đóng tệp, vì như vậy sẽ ngăn cản các TT khác truy nhập tệp. Để loại trừ vấn đề này kernel giải khoá inode vào cuối mỗi GHT dùng tệp. Nếu một TT khác thay đổi tệp giữa hai lần GHT *read()* của TT đầu, thì TT này có thể sẽ đọc được data không như nó chờ đợi, nhưng cấu trúc data của kernel vẫn nhất quán.

Xem ví dụ sau đây, là mã trình mà kernel thực hiện *2 TT đồng thời*.

```
#include <fcntl.h>
/*process A*/
main()
{
        int fd;
        char buf[512];
        fd = open("/etc/passwd", OfiRDONLY);
        read(fd, buf, sizeof(buf));             /*read1*/
        read(fd, buf, sizeof(buf));             /*read2*/
}

/*process B*/
main()
{
        int fd, I;
        char buf[512];
        for (I = 0; I < sizeof(buf); I++)
        buf[I] = "a";
        fd = open("/etc/passwd", OfiWRDONLY);
        write(fd, buf, sizeof(buf));            /*write1*/
        write(fd, buf, sizeof(buf));            /*write2*/
}
```

Ví dụ các TT đọc và TT ghi tệp được thực hiện đồng thời trên cùng một tệp "/etc/paswd".

Hai TT đều truy nhập cùng một tệp chỉ khác nhau ở đích khai thác: A đọc, B ghi tệp. Giả sử cả 2 TT đều hoàn thành GHT *open()* trước khi một trong 2 TT bắt đầu đọc hay ghi. Kernel có thể thực hiện bất kì GHT *read()* và *write()* theo 6 trình tự: read1, read2, write1, write2 hoặc read1, write1, read2, write2 hoặc read1, write1, write2, read2 hoặc v.v. Data TT A đọc phụ thuộc vào trình tự mà hệ thống thực hiện GHT của hai TT; hệ thống không đảm bảo rằng data

trong tệp là như cũ sau khi đã mở tệp. Nừu dùng cơ chế file và *record locking* (xem tiếp 5.4) thì sẽ cho phép TT đảm bảo được sự nhất quán của tệp trong khi TT đã mở tệp.

Cuối cùng ta hãy xem ví dụ một TT có thể mở một tệp nhiều lần và truy nhập qua các mô tả tệp khác nhau. Kernel thao tác các trỏ tệp offset kết hợp với mỗi fd trong file table một cách độc lập ({fd1, file offsetfi1}, {fd2, file offsetfi2}), do đó các mảng buf1 và buf2 là đồng nhất khi TT kết thúc. Giả sử không có TT nào thực hiện ghi vào tệp cùng thời điểm đó.

```
#include <fcntl.h>
main()
{
        int fd1, fd2;
        char buf1[512], buf2[512];
        fd1 = open("/etc/passwd", OfiRDONLY);
        fd2 = open("/etc/passwd", OfiRDONLY);
        read(fd1, buf1, sizeof(buf1));
        read(fd2, buf1, sizeof(buf2));
}
```

3. write

```
number = write(fd, buffer, count)
```

Các đối có ý nghĩa như trong *open()*.

Thuật toán *write()* thao tác tệp bình thường (*regular file*) là tương tự như thuật toán đọc *read()*. Tuy nhiên nếu tệp không chứa block tương ứng với byte offset để ghi, kernel cấp block mới bởi *alloc()* và gán số của block vào vị trí chuẩn xác trong bảng địa chỉ của inode. Nếu byte offset ứng với block gián tiếp, kernel sẽ cấp một vài block để dùng theo cơ chế gián tiếp, gồm các block cho chỉ số và các block data cho tệp. Inode sẽ bị khoá trong suốt thời gian thực hiện *write()* bởi vì kernel có thể thay đổi inode khi cấp những block mới (cập nhật bảng địa chỉ block của inode). Việc cho phép các TT khác truy nhập tệp có thể làm hỏng inode nếu một vài TT xin cấp block đồng thời cho cùng một byte offset. Khi ghi tệp hoàn thành, kernel cập nhật trường độ dài tệp với giá trị thực tế.

Ví dụ, một TT thực hiện ghi tiếp byte số 10.240 vào tệp. Khi truy nhập byte này trong *bmap(),* kernel xác định thấy:

- tệp chưa có block để chứa byte nói trên;
- 10 đầu vào đầu tiên cho độ dài 10x1024=10240 byte với offset bắt đầu từ 0, nên byte 10240 phải cần tiếp một cấp phát gian tiếp, indirect. Kernel cấp một block để làm indirect block, và ghi số của block này vfo inode. Sau đó cấp block cho data của tệp và ghi số của block này vào vị trí đầu tiên của indirect block.

Kernel thực hiện vòng lặp bên trong, giống như *read()*, mỗi vòng lặp ghi một block đĩa về nguyên tắc, song kernel sẽ xác định ghi toàn bộ block hay ghi một phần của block. Nếu chỉ ghi một phần thì kernel đọc block đĩa, bảo toàn phần có trước và ghi tiếp vào phần còn lại của block. Nếu ghi toàn bộ block, kernel không cần phải đọc block vì sẽ ghi đè lên nội dung cũ của block. Kernel sử dụng cơ chế trì hoãn ghi (*delayed write* sẽ đề cập ở 5.12*)* để ghi tệp nhằm giảm bớt các thao tác I/O đĩa.

4. Khóa tệp và bản ghi

Unix nguyên bản do Thompson và Ritchie phát triển không có cơ chế tệp loại trừ khi truy nhập. Để giải quyết các nhu cầu với các ứng dụng thương mại có liên quan tới cơ sở dữ liệu (*database*) System V ngày nay có cơ chế khoá tệp (*file locking*) và các bản ghi (*record locking*). *File locking* là khả năng ngăn cản một TT khác đọc, ghi bất cứ phần nào của toàn bộ tệp. *Record locking* là khả năng ngăn cản một TT khác đọc, ghi các bản ghi riêng phần (các phần của tệp nằm giữa các byte offset) Bài tập 5.9 sẽ cho ví dụ cách triển khai cơ chế này.

5. Điều chỉnh vị trí của I/O tệp

Các GHT *read()* và *write()* cho cách truy nhập tệp theo tuần tự. Để TT có khả năng định vị I/O và truy nhập tệp theo kiểu ngẫu nhiên, hệ thống cung cấp GHT *lseek()* với syntax như sau:

position = lseek(fd, offset, reference, position);

fd: mô tả tệp;

offset: byte offset

reference: cho biết offset được tính so với đầu tệp, hay vị trí đọc/ ghi hiện tại hay cuối tệp: = 0 tìm từ đầu tệp, = 1 chiều tiến tới cuối tệp từ vị trí hiện tại, = 2 tính từ cuối tệp.

position: vị trí sẽ đọc/ ghi tiếp theo bắt đầu từ byte offset này.

```
#include <fcntl.h>
main (argc,argv)
        int argc;
        char *argv[];
{
        int fd, skval;
        char c;
        if (argc != 2)
                exit();
        fd = open(argv[1], OfiPDONLY);
        if(fd == -1)
                exit();
        while((skval = read(fd, &c,1)) == 1)
        {
                prinf("char %c\n",c);
                skval = lseek(fd, 1023L,1);
                prinf("new seek val %d\n", skval);
        }
}
```

Ví dụ dùng *lseek()*.

TT mở và đọc 1 byte của tệp, sau đó gọi *lseek()* để tiến tới byte 1023 của tệp với reference =

1, so với byte hiện tại. Như vậy cứ mỗi vòng lặp, TT sẽ đọc byte thứ 1024 của tệp. Nếu reference = 0, TT sẽ tính offset so với đầu tệp, nếu reference = 2 thì tính offset từ cuối tệp. Chú ý là *lseek()* thực hiện điều chỉnh con trỏ tệp (giá trị này trong file table) trong tệp chứ hoàn toàn không điều khiển gi cơ cấu định vị đầu từ của đĩa.

6. Close

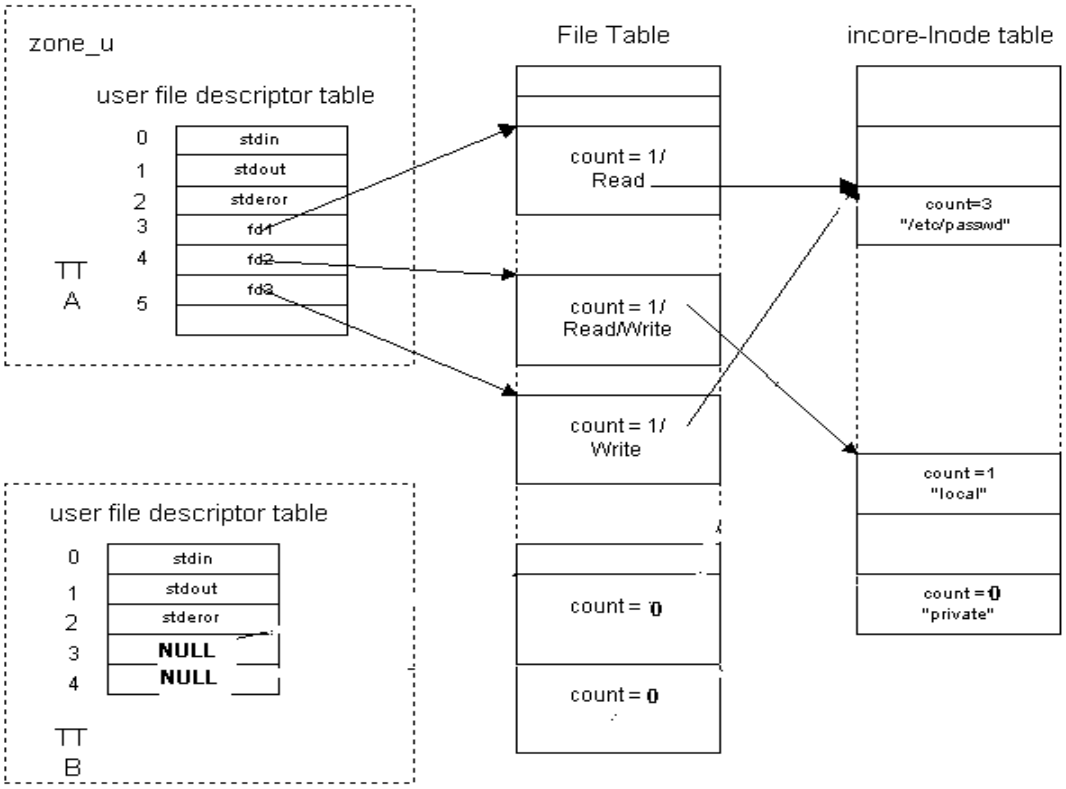
Một TT thực hiện đóng một thao tác mở tệp khi TT không còn cần truy cập tệp nữa. Cú pháp như sau:

close(fd); *fd*: mô tả tệp khi thực hiện *open()*.

Kernel thực hiện *close()* để thao tác các dữ liệu trong *user file descriptor fd*, các dữ liệu tương ứng của *file table* và *in - core inode*.

- Nếu số đếm qui chiếu *count* trong *file table* lớn hơn 1 (do có phát sinh GHT *dup(),* hay *fork()* hay có các *fds* cùng qui chiếu và tệp) thì kernel sẽ giảm giá trị đó đi 1.
- Nếu giá trị đó = 1, kernel sẽ giải phóng đầu vào này (count –1 = 0), kernel giảm số đếm qui chiếu của inode đó đi 1, đồng thời giải phóng luôn *in - core inode* cấp khi thực hiện *open()* nếu số đếm qui chiếu của nó = 0 và hoàn tất *close()*.

Như vậy khi kết thúc *close()* đầu vào trong *ufd* sẽ được dùng để cấp cho lần *open()* mới. Khi TT thực hiện một GHT *exit()* kernel các fds đang hoạt động và cũng thực hiện *close()* tất cả các tệp lại vì rằng một TT không thể duy trì mở tệp khi TT đã kết thúc.



Hình là kết quả khi TT B trong ví dụ trước, thực hiện đóng các tệp nó đã mở trước đó. Lúc

này các đầu vào số 3, 4 của bảng ufd của TT thứ 2 được giải phóng (điền là NULL). Các trường số đếm của file table sẽ có giá trị là 0, số đếm của inode "/etc/passwd" giảm đi 1, còn 2, của tệp "private" = 0. Do kernel chưa loại inode khỏi free list nên nếu có TT nào truy nhập tệp trong khi inode vẫn còn ở đó, kernel sẽ khôi phục lại tệp (xem lại phần 4.1.2).

7. Tạo tệp

Để tạo tệp dùng GHT *creat()* với cú pháp như sau:

fd = creat(pathname, modes);

Các đối có ý nghĩa giống như trong open().

Nếu tệp mới chưa có trên FS, kernel tạo mới tệp với tên xác định trong *pathname* và với các quyền truy nhập xác định ở *modes*. Nếu tệp đã tồn tại kernel sẽ giải phóng tất cả các block trước đó thuộc tệp và đặt lại độ dài tệp = 0 (*truncate*) tuỳ thuộc vào quyền truy nhập tệp.

Kernel phân tích tên thư mục và tên tệp bằng *namei()* và khi đến thành phần cuối của đường dẫn, là tên tệp sẽ tạo, *namei()* ghi nhận lại vị trí (byte offset) của đầu vào trống đầu tiên tìm thấy của thư mục và cất offset đó vào *u area*. Nếu tên tệp mới chưa có trong thư mục, kernel ghi tên mới vào chỗ trống đó. Nếu thư mục không còn có chỗ trống, kernel ghi nhớ lại offset của vị trí cuối cùng và tạo ra một đầu vào mới của thư mục. Kernel đồng thời ghi nhớ lại inode của thư mục đang tìm này trong *u area* của nó đồng thời khóa inode lại. Thư mục này sẽ là thư mục bố của tệp mới đang tạo. Với quyền được ghi trên thư mục, cuối cùng TT sẽ ghi thư mục như là phần kết quả của *creat()*.

Giả định tệp là mới trong FS, kernel cấp một inode (*ialloc()),* sau đó ghi tên tệp và inode và thư mục gốc của tệp tại vị trí offset đã lưu trước đó trong u area. Tiếp theo kernel giải phóng inode của tệp thư mục bố trước đó đã truy nhập để tìm tên tệp. Thư mục bố lúc này đã có tên và inode của tệp mới, kernel ghi inode lên đĩa (*bwrite())* trước khi ghi thư mục lên đĩa.

Nếu tên tệp đã tồn tại trước khi *creat()*, kernel tìm thấy inode ngay trong quá trình tìm tên tệp. Tệp cũ phải có quyền ghi để TT tạo tệp "mới" cũng bằng tên đó, vì kernel sẽ thay đổi nội dung của tệp bằng các công đoạn: cắt bỏ tệp, giải phóng các block data đã cấp cho tệp (*free()*), tệp cũ xem như là một tệp mới tạo với quyền truy nhập giống như tệp cũ trước đó: kernel không tái gán người sở hữu, bỏ qua các chế độ truy nhập mà TT xác định. Cuối cùng kernel cũng không kiểm tra quyền ghi thư mục vì kernel không thay đổi gì trên thư mục.

Creat() tiến triển tương tự như thuật toán *open()*. Kernel cấp đầu vào trong *file table* cho tệp đang tạo sao cho TT có thể ghi tệp, được cấp đầu vào trong bảng *ufd* và trả lại chỉ số của bảng là *fd* mô tả tệp.

8. Tạo các tệp đặc biệt

GHT *mknod()* sẽ cho phép tạo các tệp đặc biệt bao gồm *pipe, tệp các loại thiết bị, tệp thư mục*. Cú pháp như sau:

mknod (pathname, type and permissions, dev)

type and permissions: kiểu tệp tao ra là thư mục, pipe hay thiết bị và quyền truy nhập;

dev: xác định số thiết bị (*major* và *miror*) cho tệp *kiểu block* và *character.*

mknod() *tạo một nút mới*

inputs: node (file name),file type,permissions,major, minor device number (for block, character special files);

output: không.

Kernel tìm trên FS tên tệp sẽ tạo, nếu chưa có, kernel cấp cho tệp mới một inode trên đĩa và ghi tên của tệp mới, số của inode vào thư mục gốc. Kernel đặt trường kiểu tệp (type) trong inode lên giá trị để cho biết tệp là *pipe* hay thư mục (d*irectory*) hay tệp kiểu đặc biệt (*special*). Nếu tệp là kiểu thiết bị *block* hay *character* kernel ghi thêm số thiết bị major/ miror vào inode. Nếu *mknod()* là để tạo thư mục, thì node sẽ xuất hiện ngay sau khi hoàn tất lời gọi này nhưng nội dung của node thì không đúng khuôn (*format*) chuẩn (không có đầu vào nào trong thư mục cho "**.**" và "**..**". Bài tập 5.33 sẽ xét các bước cần thiết để đặt một thư mục và khuôn dạng chuẩn.

9. Chuyển đổi thư mục và root

Khi khởi động hệ thống (*booting*), TT số 0 sẽ lấy *root hệ thống* làm thư mục hiện hành trong quá trình này. TT 0 thực hiện *iget()* trên root inode, bảo vệ inode này trong *ufiarea* như là thư mục hiện hành (*current directory*) và giải tỏa việc khoá inode. Khi *fork()* tạo ra một TT mới, TT này sẽ thừa kế thư mục của TT cũ (là TT bố) đã lưu trong *u area*, và kernel tăng số đếm qui chiếu trong inode gốc.

chdir(pathname)

chdir() *chuyển thư mục*

Input: thư mục cần chuyển đến

Output: none

Với tên cung cấp kernel dùng *namei()* để xác định inode và kiểm tra inode đích có đúng là thư mục và TT có quyền truy nhập thư mục hay không. Sau đó là quá trình định vị thư mục cần tìm và việc lưu inode tương ứng của tệp thư mục trong *ufi area* của TT. Sau khi TT đã thay đổi thư mục hiện hành của nó, thuật toán *namei()* sử dụng inode để khởi động thư mục để tìm tất cả các *pathname* không bắt đầu từ *root*. Sau khi thực hiện *chdir()*, số đếm qui chiếu của inode của thư mục mới ít nhất = 1, còn của thư mục cũ có thể = 0. Inode cấp trong quá trình *chdir()* được giải phóng thực sự chỉ khi TT thực hiện một *chdir()* khác hay thực hiện *exit*.

Một TT thường dùng root để bắt đầu cho tất cả các *pathname*, bắt đầu bằnh "/". Kernel có một biến tổng thể trỏ tới inode của root hệ thống. Inode này do *iget()* cấp lúc *booting* hệ. Một TT có thể thay đổi vị trí root bằng GHT *chroot()*. Cách này rất có ích khi user muốn mô phỏng cấu trúc thông thường của FS và chạy các TT ở đây. Cú pháp như sau:

chroot(pathname); *pathname* là thư mục mà kernel sau đó xử lí như thư mục *root của TT*. Thuật toán thực hiện giống *chdir()*, sau kết quả này *root mô phỏng* có ý nghĩa như *root* cho tất cả các TT con của TT bố đã phát sinh *chroot()*.

10. Thay đổi sở hữu và chế độ truy nhập tệp

Thay đổi người sở hữu và chế độ truy nhập của một tệp là thao tác tác động trên inode của tệp đó chứ không phải trên bản thân tệp. Cú pháp như sau:

chown(pathname, owner, group);

chmod(pathname, mode);

Sau khi thay đổi quan hệ sở hữu, người "chủ" cũ sẽ mất quyền sở hữu tệp. Thay đổi mode sẽ thay đổi cách thức truy nhập tệp, các cờ (flags) trong inode thay đổi.

11. Khảo sát trạng thái tệp

Các hàm *stat()* và *fstat()* dùng để lấy các thông tin về tệp như: kiểu tệp, người sở hữu, quyền truy nhập, kích thước tệp, tổng số các liên kết, số của inode, thời điểm truy nhập tệp.

Cú pháp:

stat(pathname, statbuffer);

fstat(fd, statbuffer);

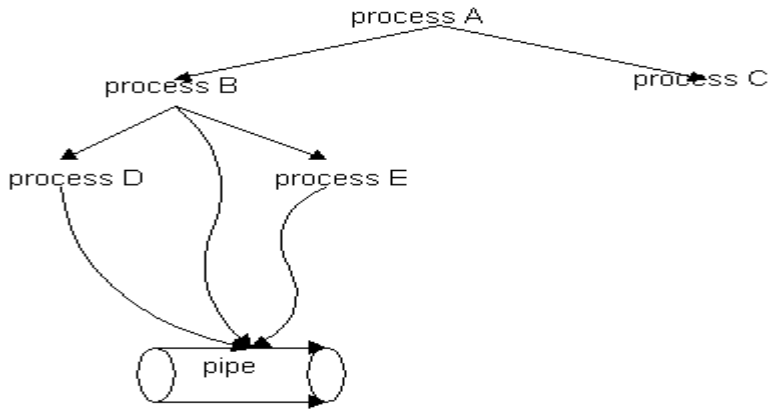
Trong đó: *statbuffer* là địa chỉ cấuu trúc data nơi sẽ chứa các thông tin trạng thái lấy từ inode.

fd là mô tả tệp do *open()* thực hiện bước trước.

12. Pipes

Pipe cho khả năng truyền data giữa các TT trong cơ chế vào trước ra trước (FIFO), và đồng thời cho phép đồng bộ việc thực hiện các TT. Sử dụng *pipe* các TT có khả năng liên lạc với nhau mà không cần biết TT nào ở đầu kia của pipe. Cách ứng dụng truyền thống là sử dụng FS để chứa data. Có hai loại pipe: *named pipe* (pipe có tên) và *unmamed pipe* (không tên). Loại này khác với *named pipe* chỉ ở cách TT khởi động truy nhập *pipe*. TT sử dụng *open()* để dùng với *named pipe*, còn dùng GHT *pipe()* để tạo một *unnamed pipe*. Tiếp theo đó TT dùng các GHT thông thường (*read(), write(), close()*); để thao tác pipes. Chỉ các TT có quan hệ huyết thống (các TT con, cháu) với TT đã tạo ra pipe mới có khả năng chia sẻ truy nhập *unamed pipe*.

Ví dụ



TT B tạo ra một pipe không tênfi*unnamed pipe* bằng GHT và sau đó sinh ra hai TT con khác

là D và E, vậy thì ba TT này sẽ chia sẻ cùng truy nhập pipe này. Trong khi đó TT A và C không thể truy nhập được. Tuy nhiên tất cả các TT có thể truy nhập named pipe bỏ qua mối quan hệ giữa chúng và chỉ thông qua quyền truy nhập tệp.

12.1. Tạo một pipe

Cú pháp tạp một pipe vô danh (***unnamed pipe)*** như sau:

pipe(fdptr);

Trong đó *fdptr* là một con trỏ trỏ vào một mảng nguyên chứa hai mô tả tệp để đọc và ghi trong pipe. Như đã đề cập, pipe là một loại tệp đặc biệt và chưa tồn tại trước khi sử dụng, nên kernel phải cấp cho pipe một inode để tạo ra pipe. Kernel đồng thời cấp một đôi mô tả tệp cũng như các đầu vào trong file table: một fd để đọc data từ pipe và một fd để ghi data và pipe. Với cách tổ chức trong file table, kernel đảm bảo sao cho giao tiếp đọc và ghi và các GHT khác trên pipe là nhất quán như đối với các tệp thường. Vì vậy TT không thể nhận biết được là TT đang làm việc với tệp hay với pipe.

pipe() ***/*create unnamed pipe*/***

input: *none*

output: *read file descriptor*

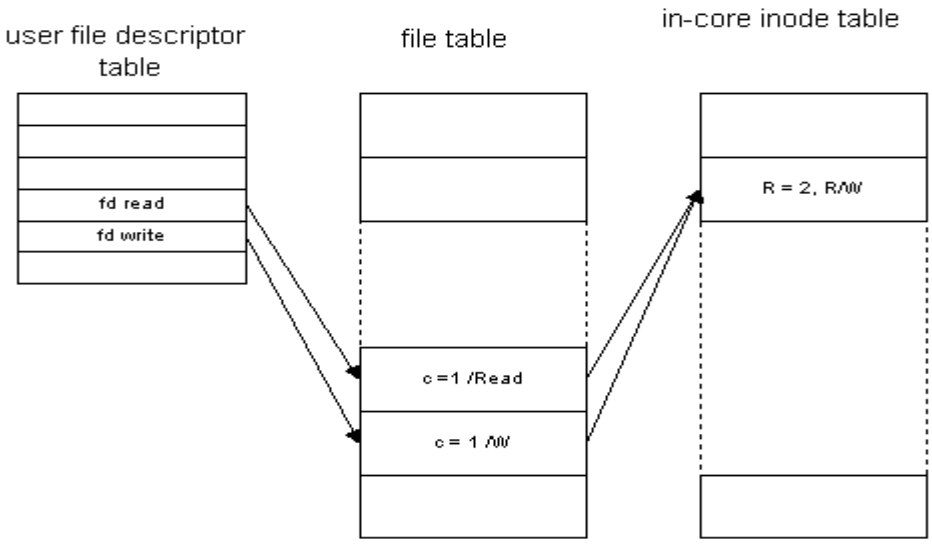
write file descriptor

Cách tạo tệp cũng giống như nhau, tuy nhiên do pipe thuộc loại tệp đặc biệt nên có ít nhiều khác biệt: inode cấp từ FS dành cho thiết bị (*pipe device*). *Pipe device* là loại mà kernel có thể cấp inode và block cho pipe, người quản trị xác định thiết bị kiểu pipe trong quá trình làm cấu hình hệ thống và tệp pipe cũng giống như các tệp khác. Trong khi pipe đang kích hoạt, kernel không thể tái gán inode và block của pipe cho tệp khác được.

Sự khác biệt còn thể hiện ở các số đếm tác động lên pipe:

- Số đếm qui chiếu truy nhập tệp (*pipe*) trong *in - core inode* khởi động = 2, vì có hai con trỏ tệp truy nhập: đọc và ghi trên pipe;
- Cùng một lúc cấp hai đầu vào trong file table: cho đọc pipe và cho ghi pipe; số đếm cách truy nhập được khởi động trong mỗi đầu vào khác nhau: cho đọc và cho ghi;
- Trong *ufd* cấp hai fd, một đọc pipe, một ghi pipe;

Duy trì *hai byte offset* là rất quan trọng để thực hiện cơ chế FIFO của pipe. TT không thể điều chỉnh *byte offset* qua *lseek()* để thực hiện truy nhập ngẫu nhiên I/O trong pipe.



12.2. Mở một pipe có tên (named pipe)

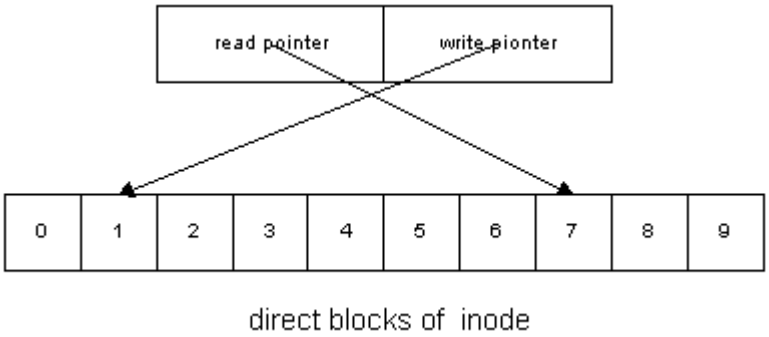
Pipe có tên-*named pipe* ý nghĩa cũng giống như *unnamed pipe*, ngoại trừ rằng pipe có tên có liệt kê trong thư mục các tệp và truy nhập tới pipe loại này bằng tên (*pathname*). Các TT mở *named pipe* giống như mở các tệp thường và các TT không có quan hệ gần với pipe có thể liên lạc với pipe. *Named pipe* luôn tồn tại trong kiến trúc của FS, trong khi đó *unnamed pipe* chỉ là nhất thời: khi tất cả các TT kết thúc sử dụng *unnamed pipe*, kernel thu hồi inode của pipe lại.

Các TT mở *named pipe* như mở các tệp thường, tuy nhiên trước khi hoàn tất lời gọi, kernel tăng các số đếm *read/ write* của inode để cho thấy số lượng các TT đã mở pipe. Một TT mở *named pipe* sẽ đi ngủ cho tới khi một TT khác mở *named pipe* để ghi và ngược lại. (Sẽ vô nghĩa nếu TT muốn đọc nhưng lại không có gì để nhận). Tuỳ thuộc vào TT mở *named pipe* để đọc hay để ghi, kernel sẽ đánh thức các TT khác đã ngủ và đang đợi để ghi hay đọc trên pipe.

Nếu TT open một *named pipe* để đọc và một TT ghi tồn tại, thì việc mở pipe hoàn tất. Hoặc nếu TT mở một *named pipe* mà có tuỳ chọn "*no delay*" thì GHT *open()* sẽ kết thúc lập tức ngay cả khi không có TT nào đang ghi. Ngược lại TT sẽ ngủ cho tới khi có một TT mở pipe để ghi. Luật này áp dụng tương tự cho một TT mở pipe để ghi.

12.3. Đọc và ghi và pipe

Làm việc trên pipe có thể xem như là khi các TT ghi vào một đầu cuối của pipe và đọc ra ở đầu cuối kia. Như đã nói TT truy nhập data ở pipe theo cơ chế FIFO. Số lượng các TT đọc pipe không nhất thiết bằng số lượng các TT ghi vào pipe. Khi số TT đọc và ghi pipe lớn hơn 1 thì các TT phải phối hợp sử dụng pipe với một cơ chế khác. Kernel truy nhập data cho pipe như cho tệp thường, kernel cất data trên thiết bị pipe và gán các block cho pipe như *write()* thực hiện. Sự khác nhau trong việc cấp block cho pipe và tệp thường là ở chỗ pipe chỉ dùng các block kiểu trực tiếp để tăng hiệu quả, cho dù có giới hạn về khối lượng data mà pipe có thể chứa mỗi lần. Kernel sử dụng các block như một hàng quay vòng, bảo trì các con trỏ đọc và ghi bên trong để bảo toàn cơ chế FIFO: vào trước ra trước.



Xem xét bốn trường hợp đọc/ ghi pipe:

1 - writing pipe: có chỗ để ghi data vào pipe,

2 - reading data từ pipe: đủ data thỏa mãn cho đọc,

3 - reading data: không đủ thỏa mãn data để đọc,

4 - writing data: không đủ chỗ trong pipe để ghi data vào.

Case 1:TT đang ghi vào pipe với giả định là pipe có đủ chỗ để ghi data: tổng số các byte đang ghi và các byte đã có trên pipe nhỏ hơn dung lượng của pipe. Kernel thực hiện theo thuật toán ghi tệp thường ngoại trừ kernel tự động tăng kích thước pipe sau mỗi lần ghi, bởi vì theo định nghĩa số data trong pipe tăng với mỗi lần ghi. Nếu con trỏ của byte tiếp theo (*byte offset*) trong pipe đã có nhu cầu dùng block gián tiếp, kernel sẽ điều chỉnh giá trị của offset của tệp trong *u area* để trỏ tới offset bắt đầu của pipe (byte offset 0) vì data không bao giờ tràn dung lượng của pipe, và kernel không bao giờ ghi đè data trong pipe. Sau khi đã ghi xong data vào pipe, kernel cập nhật inode của pipe sao cho TT sau ghi pipe sẽ tiếp tục ở vị trí cuối cùng đã dừng trước đó. Tiếp theo kernel đành thức tất cả các TT ngủ đợi đọc pipe.

Case 2:Khi TT đọc pipe, TT sẽ kiểm tra pipe có hay không có data. Nếu pipe có data, kernel đọc data như đọc từ một tệp thường theo thuật toán *read()*. Tuy nhiên offset khởi động là con trỏ đọc của pipe lưu ở inode, và nó cho biết sự mở rộng của lần đọc trước đó. Sau khi đọc mỗi block kernel giảm kích thước của pipe theo số byte đã đọc và kernel điều chỉnh giá trị của file offset trong u area để hoán đổi vào vị trí đầu của pipe. Khi kết thúc đọc pipe, kernel cập nhật con trỏ đọc (*read offset*) trong inode và đánh thức các TT ngủ đợi ghi pipe.

Case 3: đọc số byte nhiều hơn là số byte có trong pipe, kernel trả lại kết quả đọc là các byte có cho dù không thỏa mãn nhu cầu đọc theo số đếm của user. Nếu pipe rỗng, TT sẽ đi ngủ cho tới khi có TT khác ghi data vào pipe, sau đó các TT thức dậy và các TT sẽ tranh nhau đọc pipe. Tuy nhiên nếu TT mở pipe không có tùy chọn "*no delay*" thì TT sẽ kết thúc ngay nếu pipe không có data. ý nghĩa đọc/ ghi pipe giống như đọc/ ghi thiết bị đầu cuối, cho phép các chương trình bỏ qua kiểu tệp đang xử lí.

Case 4: TT ghi pipe trong khi pipe không thể chứa hết data sẽ ghi vào, kernel đánh dấu inode và đưa TT đi ngủ đợi data thoát bớt khỏi pipe. Sau đó khi có TT khác đọc pipe kernel sẽ thông báo cho biết TT ghi đang ngủ đợi để pipe có chỗ cho việc ghi data. Quá trình sẽ tiếp tục như đã nói. Trường hợp lưu ý là data ghi lớn hơn dung lượng chứa của pipe (tức dung lượng của block trực tiếp cấp cho pipe), kernel sẽ ghi số data tối đa có thể vào pipe và để TT đi ngủ cho tới khi có chỗ để ghi tiếp. Nếu vậy sẽ có thể xảy ra là data sẽ không liên tục khi một TT khác ghi data của nó vào pipe trước khi TT kia lại tiếp tục ghi data của mình.

Qua phân tích các ứng dụng pipe ta thấy kernel coi pipe như là tệp thường, nhưng thay vì quản lí con trỏ tệp (*byte offset*) ở *file table* đối với tệp thường, thì *byte offset của pipe được*

quản lí ở inode. Kernel nhớ offset đọc / ghi của *named pipe* trong inode sao cho các TT có khả năng chia sẻ các giá trị của chúng: inode mở cho nhiều TT, trong khi mỗi lần mở tệp có một đầu vào tương ứng trong file table. Đối với *unnamed pipe* các TT truy nhập qua các đầu vào trong file table như thông lệ, như đối với tệp thường.

12.4. Đóng Pipes

Khi đóng pipe các TT thực hiện các qui trình như đối với tệp thường trừ một vài thao tác đặc biệt kernel thực hiện trước khi giải phóng inode của pipe. Kernel giảm số các TT đọc/ ghi pipe theo kiểu của mô tả tệp. Nếu số đếm các TT ghi bằng 0 và có các TT ngủ đợi đọc pipe, kernel đánh thức và các TT này thoát khỏi lời gọi chúng mà không cần đọc pipe. Nếu số đếm các TT đọc pipe bằng 0 và có các TT ngủ đợi ghi pipe, kernel đánh thức và gởi *tín hiệu - signal* thông báo có error cho các TT này. Nếu không còn có TT đọc hay TT ghi pipe, kernel sẽ giảI phóng các block data của pipe và điều chỉnh lại inode để cho biết pipe không có data.

Ví dụ

Ví dụ ảo dùng pipe:

```
char string[] = ”hello”;
main()
{
        char buf[1024];
        char *cp1, cp2;
        int fds[2];
        cp1 = string;
        cp2 = buf;
        while(*cp1)
                *cp2++ = *cp1++;
        pipe(fds);
for(;;)
{
        write(fds[1], buf, 6);
        read(fds[0], buf,6);
}
}
```

Đọc và ghi một *unnamed pipe*.

TT tạo một *unnamed pipe* bằng GHT *pipe()*, sau đó TT đi vào một vòng vô hạn để ghi xâu ”hello” vào pipe và đọc xâu đó từ pipe. Trong ví dụ TT đọc và TT ghi là một.

```
#include <fcntl.h>
char string[ ] = “hello”;
main(argc, argv)
        int argc;
```

```
        char argv[];
{
        in fd;
        char buf[256];
        /*create named pipe with R/W permission for all users*/
        mknod("fifo",010777, 0);
        if (argc == 2)
                fd = open("fifo", OfiWRONLY);
        else
                fd = open("fifo", OfiRDONLY);
        for (;;)
                if (argc == 2)
                        write(fd, string, 6);
                else
                        read(fd, buf, 6);
}
```

Đọc ghi một *named pipe*

Một TT tạo ra một *named pipe* có tên "fifo" và khởi động ghi (TT ghi) xâu "hello" vào pipe nếu có 2 đối đầu vào; nếu chỉ có một đối đầu vào thì kích hoạt đọc (TT đọc) pipe. Hai TT ghi/ đọc liên lạc "bí mật" qua pipe "fifo", tuy nhiên 2 TT không nhất thiết có liên quan với nhau.

13. Sao chép một mô tả tệp (dup)

GHT *dup()* sao chép một mô tả tệp fd vào một đầu vào trống của *ufd*, trả lại fd mới. *du()* áp dụng cho tất cả các kiểu tệp.

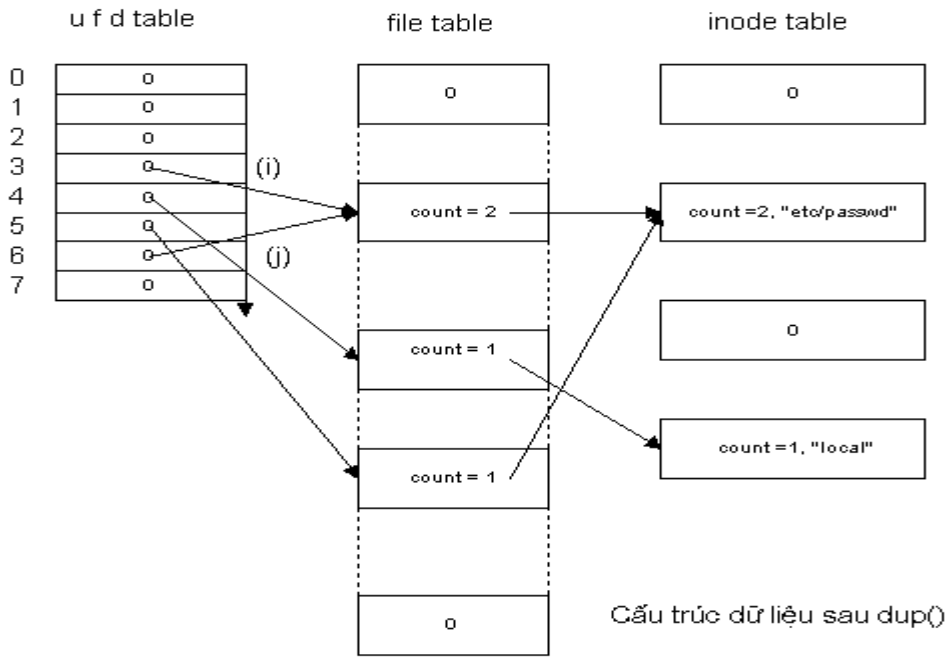
newfd = dup(fd);

Trong đó: *fd* là mô tả tệp gốc sẽ được sao chép,

newfd là mô tả tệp mới cũng qui chiếu tới tệp.

Bởi vì *dup()* sao chép fd nên nó tăng số đếm của đầu vào tương ứng của tệp trong file table, tức là có thêm một fd trỏ vào đầu vào này.

Ví dụ trong hình TT mở tệp "/etc/passwd", kết quả sẽ có fd3. Sau đó TT mở tệp "local" và fd tương ứng là fd4. Khi TT mở "/etc/passwd" một lần nữa, sẽ có fd5 trong ufd. Với 2 lần mở "/etc/passwd", trong file table có 2 đầu vào độc lập tương ứng, và số đếm qui chiếu trong inode của tệp "/etc/passwd" là 2 (2 lần tệp được mở). Nếu TT thực hiện *dup(fd3)*, ta sẽ có một bản sao mô tả tệp fd6 cùng trỏ vào nơi fd3 trỏ trong file table, tại đây số đếm các fd trỏ vào là 2 (bao gồm fd3 và fd6). Mục đích quan trọng của *dup()* là để xây dựng các chương trình tinh xảo từ các chương trình đơn giản hay các khối chương trình chuẩn. Một trong các ứng dụng điển hình là cấu tạo các kênh liên lạc của shell.



Xét ví dụ dưới đây:

```
#include <fcntl.h>
mail()
{
        int i, j;
        char buf1[512], buf2[512];
        j = open("/etc/passwd", OfiRDONLY);
        j = dup(i);
        read(i, buf1, sizeof(buf1);
        read(j, buf2, sizeof(buf2);
        close(i);
        read(j, buf2, sizeof(buf2);
}
```

Minh họa *dup()*.

Biến *i* là mô tả tệp của "/etc/passwd", *j* là mô tả tệp được tạo ra bởi *dup()* của cùng tệp TT mở. *j* làbản sao chép của *i*. Trong *ufd* của *u area* sẽ có 2 fd là *i* và *j* cùng trỏ vào một đầu vào trong *file table* và do đó cả hai dùng chung một con trỏ tệp - *byte offset*. Chú ý là hai *read()* liên tiếp nhau sau đó cho data trong *buf1* và *buf2* không như nhau. Điều này khác với trường hợp khi TT *open()* hai lần trên cùng một tệp (xem lại 5.2). Một TT có thể *close()* một fd mà TT muốn, nhưng I/O vẫn tiếp tục bình thường đối với fd kia, điều đó thể hiện trong ví dụ. Một ứng dụng khác mà *dup()* làm là đổi vai trò truyền thống của một fd: sau khi *dup(fd1)* - *standard output*, TT có thể *close(fd1)* và fd mới sẽ làm việc như fd1. Chương 7 sẽ cho thấy

tính thực tế của việc dùng *pipe()* và *dup()* khi mô tả về *shell*.

14. Ghép và tháo gở một FS (mount/unmount)

Một đơn vị đĩa thường được chia ra thành các phân hoạch và mỗi phân hoạch sẽ có tên của thiết bị đi kèm. Trên FS điều đó được thể hiện dưới dạng một tệp thiết bị tại thư mục /dev/. TT có thể truy nhập data trên phân hoạch bằng *open()* tệp thiết bị và sau đó đọc/ ghi "tệp" này, xử lí tuần tự các block đĩa. Chương 10 sẽ mô tả kĩ giao tiếp này.

14.1. Ghép (mount) FS

Một phân hoạch đĩa là một hệ thống tệp, và có *boot block*, *super block*, *inode list*, *data blocks* như bất kì một FS nào. GHT *mount()* sẽ ghép nối FS của phân hoạch trên đĩa vào cấu trúc FS đã tồn tại, sau đó users truy nhập data trên phân hoạch như thông thường, *unmout()* sẽ cắt FS đó ra.

Cú pháp:

mount(special pathname, directory pathname, options);

Trong đó:

- *special pathname:* là tên của tệp thiết bị của phân hoạch có FS sẽ ghép vào;
- *directory pathname:* là tên thư mục trên FS đã tồn tại (root), còn gọi là điểm ghép (mount point),
- *options:* cho biết kiểu truy nhập: R/W, read-only.

Ví dụ:

mount("/dev/dsk1", /"usr", 0);

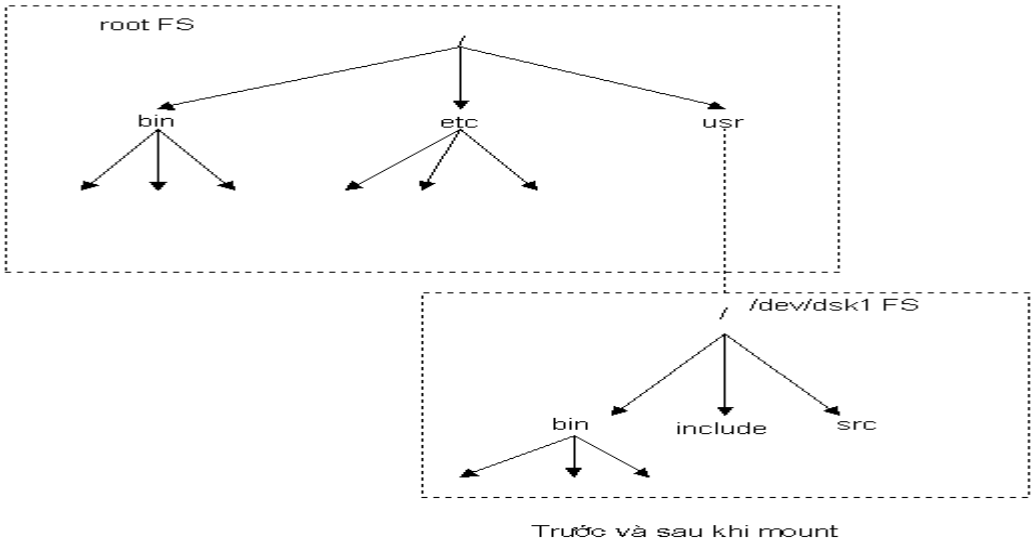
Kernel sẽ ghép FS của phân hoạch "/dev/dsk1" vào thư mục "/usr" của FS đã có. Tệp "/dev/dsk1" là một tệp đặc biệt kiểu block, có nghĩa là tên của thiết bị khối, thường là một phân hoạch đĩa. Phân hoạch có cấu trúc đầy đủ của một FS như đã nói. Sau khi đã ghép vào, root của FS ghép sẽ được truy nhập qua thư mục "/usr"."

Cây thư mục trước và sau *mount*.

Kernel có một *mount table* với các đầu vào cho mỗi một FS khi FS đó ghép vào FS đã tồn tại. Mỗi đầu vào có các trường thông tin sau đây:

- Số của thiết bị nhận biết FS được ghép (là số logic của FS);
- Con trỏ trỏ vào buffer có chứa super block của FS;
- Con trỏ trỏ vào root inode của FS được ghép ("/ " là root của "/dev/dsk1 trong hình);
- Con trỏ trỏ vào inode của thư mục là điểm ghép ("usr" là thư mục điểm ghép cho FS /dev/dsk1 vào FS đã tồn tại).

Sự kết hợp giữa inode của điểm ghép và root inode của FS được ghép trong thủ tục *mount* giúp cho kernel hoạt động trên FS uyển chuyển và người dùng không cần có sự hiểu biết gì đặc biệt.



Trước và sau khi mount

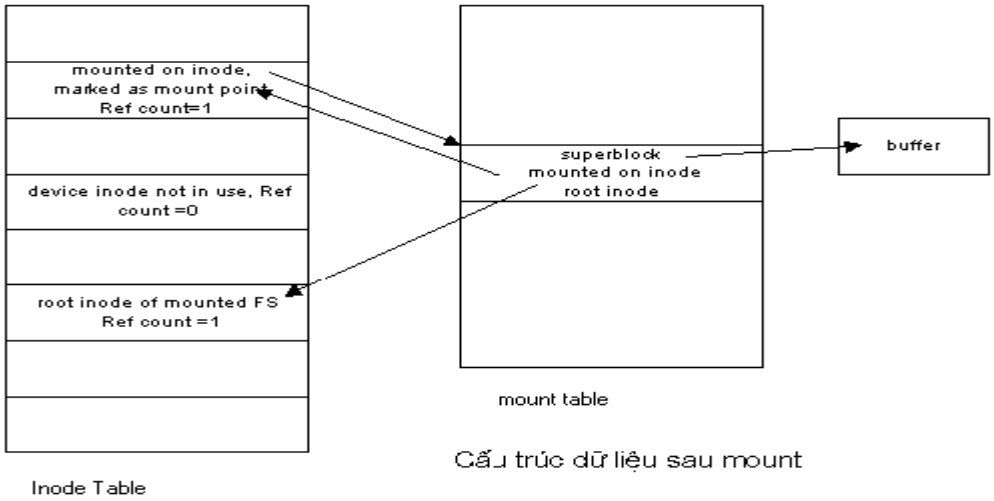
mount()

input: *file name of block special file*
directory name of mount point
options (read only)

output: *none*

Kernal cho phép chỉ các TT do super user sở hữu mới thực hiện *mount* và *unmount* FS, để tránh sự hủy hoại FS ghép. Kernel tìm inode của tệp thiết bị có FS sẽ được ghép, lấy ra số major và minor để xác định chính xác phần phân hoạch nào của đĩa chứa FS đó. Kernel sau đó tìm inode của thư mục nơi sẽ ghép FS mới và FS đã tồn tại. Số đếm qui chiếu của inode thư mục không được lớn hơn 1, nhưng ít nhất là 1 để tránh các sự cố không mong muốn có thể xảy ra (xem bài tập 5.27). Kernel tìm một đầu vào trống trong *mount table* và đánh dấu đưa vào sử dụng, gán số của thiết bị vào trường này. Quá trình này được thực hiện ngay lập tức để tránh có hiện tượng một TT khác mount FS lần nữa. Sẽ có sự bất thường xảy ra nếu cùng một FS lại được phép mount lần thứ hai (xem bài tập 5.26).

Kernel thực hiện thủ tục *open* như đối với các thiết bị block khác: kiểm tra sự hợp lí của thiết bị, khởi động các dữ liệu của bộ điều khiển (*driver*), gởi các lệnh khởi động thiết bị cho phần cứng. Các thao tác tiếp theo cần có là cấp buffer ẩn để đọc và lưu lại *super block* của FS được ghép, lưu con trỏ của thư mục điểm ghép FS của cây thư mục gốc sao cho các đường dẫn có chứa ".." có thể đi ngang qua điểm ghép; kernel tìm *inode root* của FS được ghép và lưu con trỏ trỏ vào inode này vào trường tương ứng của *mount table*. FS mới lúc này được nhìn nhận như một thư mục của FS gốc: đối với user thư mục ghép này (của FS gốc) và root của FS được ghép là tương đương về logic và sự tương đương đó được kernel xác lập bởi sự cùng tồn tại trong *mount table*. Kernel khởi động các trường giá trị thuộc tính của super block, mở khóa (*lock*) cho *free block list*, *free inode list*, đặt số các free inode trong super block về 0. Mục đích của quá trình khởi động này là để tối thiểu nguy cơ hỏng FS trong khi ghép FS sau lần hệ thống bị sự cố: Khi đặt số *free inode* = 0 sẽ buộc kernel thực hiện ialloc() để tìm các free inode đĩa cập nhật một cách chính xác sau khi đã mount FS lên. Xem hình dưới đây để thấy các cấu trúc dữ liệu khác được xây dựng vào thời điểm cuối của quá trình mount.



Cấu trúc dữ liệu sau mount

Do có sự thay đổi cấu trúc của FS với sự ghép FS, nên một số GHT có sự thay đổi để phù hợp với cấu trúc đó. Việc viết lại các GHT này là để tổng quát các trường hợp khi có sự "đi qua" điểm ghép: từ FS gốc đến FS ghép (đi từ root đến nhánh ghép vào) và ngược lại.

Ví dụ về sử dụng mount và unmount:

1. Giả sử trên máy tính có một hay vài đĩa cứng với các phân hoạch khác nhau, mỗi phân hoạch là một FS;

2. Các phân hoạch có thể là FAT (16, 32), NTFS, ext2, ext3;

Sau đây là các bước cần làm:

login vào vơi tư cách **root** hay **su**:

a. **$ su [ENTER]**

b Dùng lệnh sau để biết các phân hoạch có trên các đĩa cứng:

 # sfdisk –l

Hãy quan sát kết quả cua lệnh, chọn phân hoạch sẽ mount vào FS:

ví dụ **/dev/hda4 0 764 765 6144831 Win95 FAT32**

phân hoạch /dev/hda4 là loại FAT32 của MS Windows

b. tạo một thư mục rỗng (chổ để ghép FS và cây thư mục chính):

 # mkdir /mnt/hda4 (Có thể lấy tên bất kì sao cho có ý nghĩa, dễ hiểu).

c. Ghép vào FS gốc:

 # mount –t vfat /dev/hda4 /mnt/hda4

Bây giờ có thể truy nhập (R/W) như bình thường.

Sau khi sử dụng unount trước khi tắt máy tính.

d. umount (phải ra khỏi thư mục /mnt/hda4 trước khi umount ! Xem 14.2)

 # umount /dev/hda4

14.2. Cắt (unmount) FS

umount (special filename);

Trong đó: *special filename* cho tên của FS sẽ cắt khỏi FS gốc.

Khi cắt một FS, kernel truy nhập inode của thiết bị sẽ cắt, tìm số thiết bị của tệp đặc biệt, giải phóng inode (iput()), tìm đầu vào trong mount table các con trỏ cần thiết liên kết với tệp thiết bị. Trước khi thực sự cắt ghép FS kernel kiểm tra để chắc chắn không có tệp nào của FS ghép vẫn còn đang sử dụng bằng cách tìm trong *inode table* tất cả các tệp có số thiết bị bằng số của tệp đặc biệt sẽ cắt ra khỏi FS gốc. Buffer ẩn có thể còn có các block ở chế độ ghi "*delayed write*", kernel cập nhật lên đĩa, cập nhật super block đã có sự thay đổi, cập nhật các bản *in - core inode* lên đĩa...

umount()

input: special file name of FS to be unmounted

output: none

15. Liên kết tệp (link)

GHT *link()* sẽ tạo một tên mới cho một tệp đã có trên cấu trúc thư mục hệ thống, thực tế tạo ra một đầu vào mới cho một inode đã tồn tại. Cú pháp như sau:

link(tên tệp gốc, tên tệp đích)

Trong đó: *tên tệp gốc* là tên của tệp đã tồn tại,

tên tệp đích là tên tệp sẽ có sau khi hoàn tất GHT *link()* .

FS chứa đường dẫn cho mỗi liên kết mà tệp có sao cho các TT có thể truy nhập tệp bằng bất kì đường dẫn nào. Kernel hoàn toàn không biết tên nào của tệp là tên gốc, cho nên không cần có xử lí gì đặc biệt trên tên tệp.

Ví dụ:

*link("/usr/src/uts/sys","/usr/include/sys"); /*link thư mục*/*

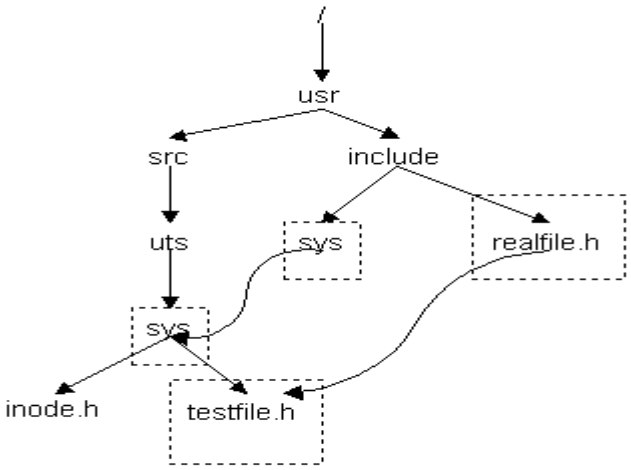
link("usr/include/realfile.h","/usr/src/uts/sys/testfile");

Sau khi thực hiện hai lệnh trên sẽ có 3 đường dẫn cùng qui chiếu tới một tệp:

"/usr/src/uts/sys/testfile.h"

"/usr/include/sys/testfile.h"

"/usr/include/realfile"



Các liên kết trong FS

Ví dụ tại console, thực hiện liên kết logic:

$ ln –s /abc/xyz /mnt/yyy

trong đó /abc/xyz là thư mục gốc tại /abc/xyz,

/mnt/yyy sẽ là đích. Kết quả sau đó nếu dùng lệnh ls –l sẽ có dòng sau:

$ ls -l

l rwx rwx rwx 1 root root xxx -> /abc/xyz

Kernel chỉ cho phép *superuser* thực hiện *liên kết thư mục*, và ngay cả *superuser* cũng phải thận trọng khi liên kết thư mục nếu không có thể dẫn đến một vòng lặp vô tận. Ví dụ khi liên kết một thư mục vào một tên bên dưới nó trong cấu trúc có thứ bậc.

Với các phiên bản mới, việc đưa vào lệnh *mkdir()* sẽ hỗ trợ cho việc tạo thư mục mới có áp dụng khả năng của *link()* và loại trừ sự cần thiết phải liên kết thư mục.

Thuật toán link()

Input: *tên tệp gốc (đã có)*

tên tệp đích (mới)

Output: *none*

Các bước trong thuật toán này như sau:

- Kernel định vị inode của tệp nguồn (*namei()*), tăng số đếm liên kết tới tệp, cập nhật mới nội dung của inode đĩa để đảm bảo sự thống nhất của cấu trúc FS và giải khóa inode đó.
- Kernel đi tìm tệp đích: nếu tệp đích ở nơi sẽ tạo đã tồn tại (trùng tên), báo lỗi, giảm số liên kết (đã làm ở bước trước). Nếu là tệp mới, kernel tạo chỗ cho inode của tệp đích, ghi vào đó tên tệp mới, số inode của tệp gốc, ghi nhận inode của tệp mới trong thư mục đích (*iput()*: giải khóa inode của tệp thư mục gốc của tệp đích), ghi nhận mới tệp gốc với số đếm liên kết tăng thêm 1 và tệp đã có thêm một tên mới và có thể truy nhập inode gốc bằng tên mới này. Số đếm link giữ số đếm của các đầu vào của tệp thư mục để qui chiếu và tệp và khác với số đếm trong inode. Nếu không có TT nào truy nhập tệp ở cuối quá trình link, thì số đếm qui chiếu của inode của tệp = 0, và số đếm liên kết của tệp ít nhất là 2.

Ví dụ: *link(“source”,”dir/target”)*

Giả định số inode của "source" là 74. Kernel tìm " source", tăng số liên kết thêm 1, nhớ lại số 74, hủy khoá " source". Kernel tìm inode của tệp thư mục " dir" (là thư mục gốc của tệp " target"), tìm trong tệp này một đầu vào trống cho tệp mới " " " " target", ghi tên " target" và số inode 74 vào chỗ này. Sau đó giải trừ khóa cho tệp " source" qua *iput()*. Số đếm liên kết đã tăng thêm 1 (nếu là 1 thì lúc này = 2). Như vậy ta thấy FS không mất thêm một inode cho tệp mà chỉ thêm một tên tệp mới trong thư mục đích mà thôi. Trong thuật toán này có một điểm rất quan trọng cần lưu ý là việc giải trừ lock của tệp gốc sau khi đã tăng số liên kết để các TT có thể truy nhập tệp này mà không gây ra tình trạng bị khóa lẫn nhau (*deadlocking*).

Ví dụ: TT A: link("**a/b/c/d**","*e/f*")

TT B: link("*e/f*","**a/b/c/d**/ee")

Điều gì sẽ xảy ra khi **cùng một thời điểm** (là trạng thái ở đó mỗi TT đã được cấp inode của tệp nó cần truy nhập để xử lí): TT A có "e/f" để tìm thư mục này nhưng TT B lại đang khai thác "f" mà không giải trừ trong GHT *link()*. Tương tự khi TT B tìm inode của thư mục "/a/b/c/**d**" mà TTA lại không giải trừ lock thư mục "d". Cả hai TT đều bị kẹt bởi đã không giải trừ lock sau khi đã thực hiện các bước đầu trong thuật toán. Kernel loại trừ kiểu kẹt cổ điển này bằng cách giải trừ khoá inode của tệp nguồn sau khi tăng số đếm liên kết của tệp. Vì rằng inode nguồn đầu tiên là tự do (*free*) khi truy nhập nguồn tiếp theo,vậy sẽ không có *deadlock* nữa.

16. Hủy liên kết tệp (unlink())

GHT *unlink()* sẽ xoá bỏ đầu vào của tệp trong thư mục, tệp sau đó không tồn tại ở thư mục đó nữa, mọi truy nhập sẽ có thông báo lỗi. Nếu thực hiện *unlink()* đối với link cuối cùng (tương đương với delete tệp), kernel sẽ giải phóng các blocks đã cấp cho tệp. Nếu còn có các links khác (với các tên khác nhau) tệp vẫn được truy nhập bình thường.

Cú pháp: *unlink(pathname)*

Thật toán *unlink()*

Input: *file name*

Output: *none*

17. Sự trừu tượng hoá FS

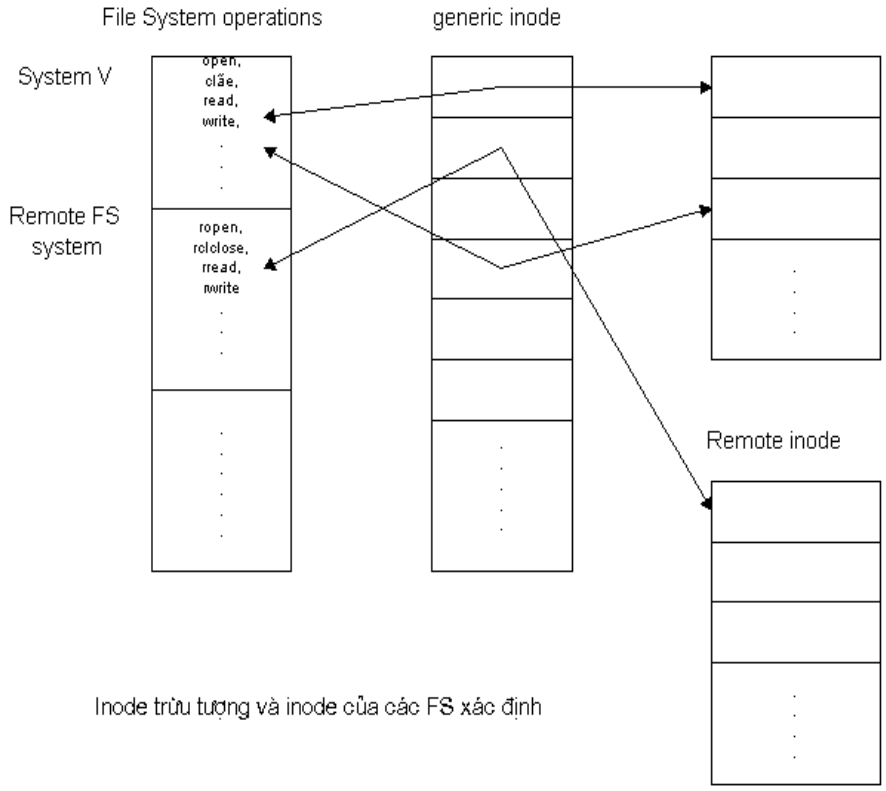
Tác giả Weinberger đã giới thiệu ***các kiểu tệp*** (***file system types***) để hỗ trợ hệ thống tệp mạng (network FS) và các phiên bản mới nhất của System V cũng hỗ trợ ý tưởng này. Với ***file system types*** kernel có thể hỗ trợ đa hệ thống tệp đồng thời (network FS, FS cho các hệ điều hành khác nhau). Trong khi các tiến trình sử dụng các GHT thông dụng của Unix để truy nhập tệp thì kernel ánh xạ tập các thao tác chung trên tệp vào các thao tác xác định cho mỗi kiểu FS.

Inode trở thành chỗ ghép nối giữa FS trừu tượng (*generic inode*) và FS cụ thể. Một *incorefiinode* trừu tượng sẽ chứa các dữ liệu (*số thiết bị, inode number, file type, size, owner, reference count*) độc lập với kiểu FS cụ thể và trỏ vào một inode của FS xác định. Inode của FS xác định này sẽ cho các chỉ dẫn cụ thể áp dụng trên FS đó (quyền truy nhập, mô hình các block data, super block cấu trúc thư mục). Một FS không nhất thiết phải có cấu trúc tựa inode, nhưng FS xác định phải thoả mãn các ngữ nghĩa của Unix FS và phải cấp inode của nó

khi kernel mở một inode trừu tượng.

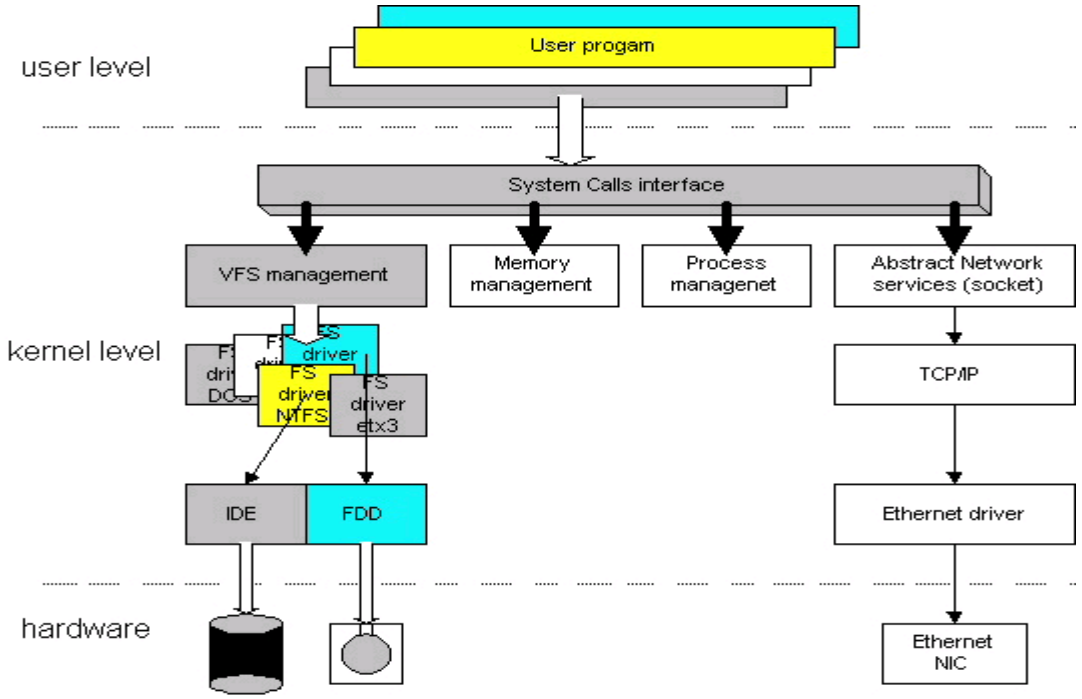
Mỗi kiểu FS có một cấu trúc chứa địa chỉ của các hàm thực hiện các thao tác trừu tượng. Khi kernel muốn truy nhập tệp, kernel thực hiện gọi hàm gián tiếp trên cơ sở của kiểu FS và cách thao tác trên FS đó. Các thao tác trừu tượng bao gồm: *open, close, read, write, return inode, release inode, update inode, ckeck permissions, set attribute (attribute), mount, umount*.

Linux là một ví dụ áp dụng FS trừu tượng, còn gọi là FS ảo (Virtual FS: VFS), hổ trợ truy nhập hầu hết các FS hiện có, trong đó có DOS, CDfiROM, NTFS (của WINDOWS 2000, XP) (Bản **Linux Mandrake 8.1** rất hoàn hảo).



Inode trừu tượng và inode của các FS xác định

Mô hình FS trên Linux:



Linux model

18. Bảo trì FS

Kernel duy trì sự bền vững của FS trong chế độ hoạt động bình thường. Nhưng khi bị sự cố (mất điện) FS có thể bị rối loạn: có tệp còn, có tệp mất. Một GHT của kernel sẽ thực hiện kiểm tra khi *reboot* máy và thực hiện truy nhập tệp bỏ qua các kiểu truy nhập thông thường để sửa lại FS. Một số các tình huống sẽ được *fsck()* thực hiện:

- Một blk đĩa có thể là thuộc nhiều inode, hay thuộc danh sách các blk tự do hay của một inode. Vào lúc tạo FS (nguyên thuỷ), các blks đĩa thuộc danh sách các blks tự do. Khi được cấp phát các blk này sẽ bị loại ra khỏi danh sách nói trên để gán cho một inode. Một blk sẽ được gán lại cho inode khác chỉ sau khi nó đã trở lại danh sách tự do. Như vậy một blk đĩa chỉ có thể là trong danh sách tự do hay đã thuộc một inode. Xem xét các khả năng khi kernel giải phóng blk của tệp trả lại số của nó vào bảng động (*in - core*) của Super block trong bộ nhớ và cấp nó cho tệp mới tạo: nếu kernel đã ghi lên đĩa inode và các blks của tệp mới tạo nhưng sự cố đã xảy ra trước khi cập nhật mới inode của tệp cũ (được giải phóng), thì một blk có thể sẽ thuộc 2 inode. Tương tự: nếu kernel đã ghi được Super block và danh sách blk tự do của nó lên đĩa và đã có sự cố trước khi cập nhật inode cũ thì blk đĩa có thể sẽ hiện diện ở cả trong danh sách tự do và trong inode.
- Nếu số của blk không trong danh sách tự do cũng không trong inode thì FS được coi là nhất quán vì các blk phải có mặt ở đâu đó. Tình huống này như sau:blk đã loại khỏi tệp và đã đặt vào danh sách tự do của Super block. Nếu tệp cũ đã được ghi đĩa và sự cố xảy ra trước khi Super block được cập nhật, thì blk có thể không xuất hiện trong bất kì danh sách nào trên đĩa.
- Nếu khuôn dạng (*format*) của inode không chuẩn (vì trường file type có giá trị không được định nghĩa), FS có vấn đề;

- Số của inode có trong đầu vào thư mục, nhưng inode là tự do, FS có vấn đề; Xảy ra là do: kernel đã tạo xong tệp mới, đã cập nhật thư mục lên đĩa nhưng chưa cập nhật danh sách inode thì có sự cố.
- Nếu số các block tự do và inode tự do ghi nhận trong Super block không phù hợp với số tồn tại trên đĩa, FS có vấn đề.

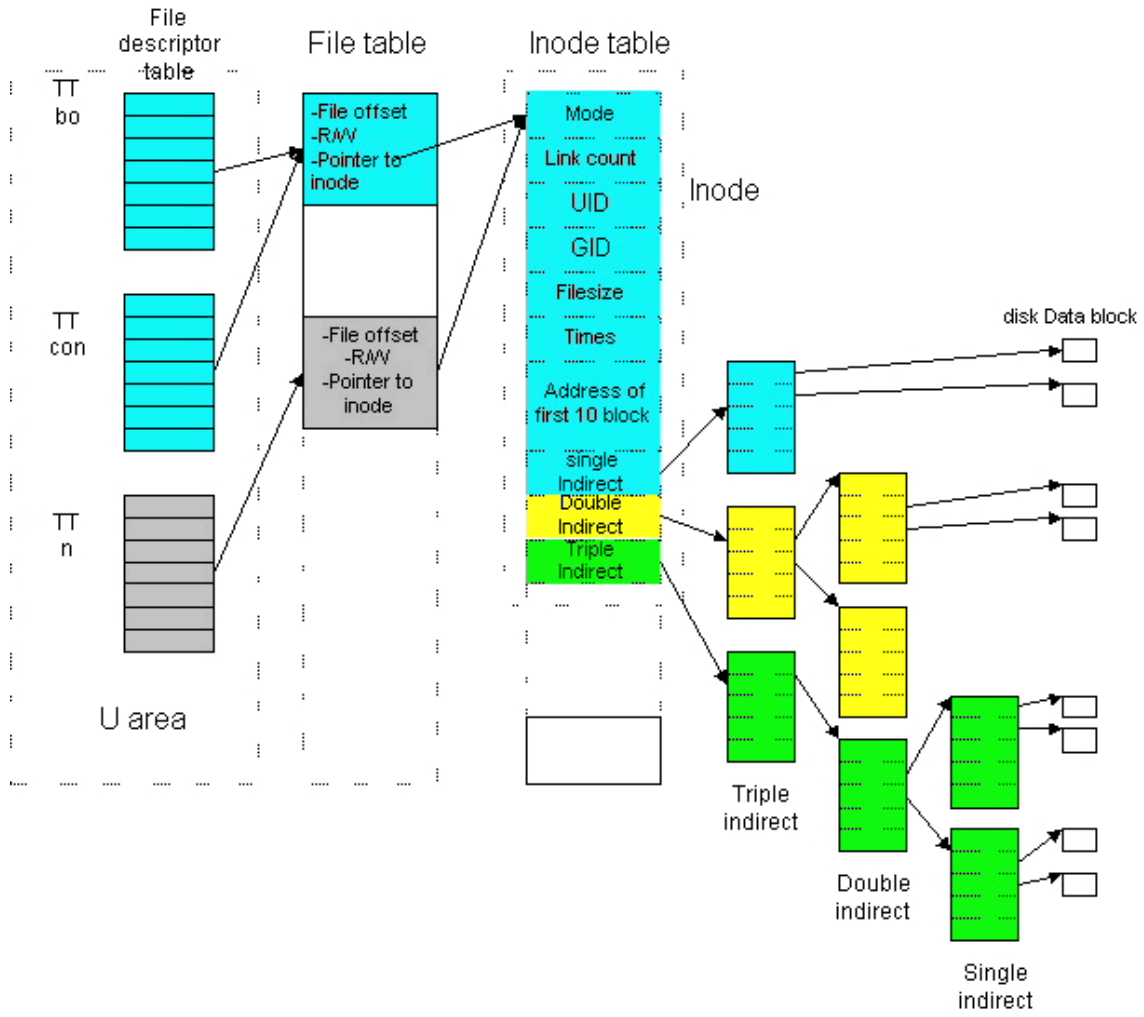
19. Tóm tắt và bài tập

Tóm tắt

Chương này giải thích về FS, giới thiệu 3 cấu trúc quản lí tệp của kernel:

- *File table,*
- *Inode table,*
- user *file descriptor table* (*ufd table*),
- *mount table*
- các thuật toán liên quan tới FS và mối tương tác giữa chúng
- *fsck()* dùng để bảo trì FS.

Mói quan hệ giữa các bảng trên có thể tóm tắt ở hình sau đây:



Ví dụ 1:

```
/*************************************************************************
* Lệnh có đối đầu vào, chuẩn C                                           *
*************************************************************************/

#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  int fd;
  struct iovec* vec;
  struct iovec* vecfinext;
  int i;
  /* cần "buffer" để chứa kí tự dòng mới.  Sử dụng biến kí tự cho mục*/
      /*đích này.  */
```

```
  char newline = '\n';
  /* Đối đầu tiên là tên tệp đầu ra.  */
  char* filename = argv[1];
  /* Bỏ qua 2 thành phần đầu của danh sách đối.  Thành phần 0 là tên
của chương trình này, thành phần 1 là tên tệp đầu ra.  */
  argc -= 2;
  argv += 2;

  /* xin cấp một trường các thành phần kiểu iovec.  Ta cần 2 cho mỗi thành
phần của danh sách đối: một cho text, một cho newline.  */
  vec = (struct iovec*) malloc (2 * argc * sizeof (struct iovec));

  /* Xst qua danh sách đối, xây dựng các thành phần cho iovec.  */
  vecfinext = vec;
  for (i = 0; i < argc; ++i) {
    /* Thành phần đầu tiên : text của bản thân đối.  */
    vecfinext->iovfibase = argv[i];
    vecfinext->iovfilen = strlen (argv[i]);
    ++vecfinext;
    /* Thành phần thứ 2 : kí tự dòng mới (newline).  Các thành phần của
struct iovec array trỉ tới cùng một vùng bộ nhớ.  */
    vecfinext->iovfibase = &newline;
    vecfinext->iovfilen = 1;
    ++vecfinext;
  }

  /* Ghi các đối mới vào tệp.  */
  fd = open (filename, OfiWRONLY | OfiCREAT);
  writev (fd, vec, 2 * argc);
  close (fd);

  free (vec);
  return 0;
}
```

Ví dụ 2:

```
/*****************************************************************
*Đọc nội dung của FILENAME vào buffer.  Kích thước của buffer để ở*
*LENGTH.  Trả lại oqr đầu ra là buffer và phải được giải phóng.
Nếu* *FILENAME không phải là tệp thường (regular), trả lại là NULL.
    *
*****************************************************************/

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
```

```
#include <unistd.h>

/*   */

char* readfifile (const char* filename, sizefit* length)
{
  int fd;
  struct stat filefiinfo;
  char* buffer;

  /* Open the file.  */
  fd = open (filename, OfiRDONLY);

  /* Get information about the file.  */
  fstat (fd, &filefiinfo);
  *length = filefiinfo.stfisize;
  /* Make sure the file is an ordinary file.  */
  if (!SfiISREG (filefiinfo.stfimode)) {
    /* It's not, so give up.  */
    close (fd);
    return NULL;
  }

  /* Allocate a buffer large enough to hold the file's contents.  */
  buffer = (char*) malloc (*length);
  /* Read the file into the buffer.  */
  read (fd, buffer, *length);

  /* Finish up.  */
  close (fd);
  return buffer;
}
```

Ví dụ 3:

```
/**********************************************************************
*   Ghi tổng số byte COUNT từ BUFFER vào descriptor FD.                *
*   Returns -1 :error, OK= Tổng số byté ghi được                       *
**********************************************************************/

#include <assert.h>
#include <unistd.h>

/*   */

ssizefit writefiall (int fd, const void* buffer, sizefit count)
{
  sizefit leftfitofiwrite = count;
  while (leftfitofiwrite > 0) {
    sizefit written = write (fd, buffer, count);
```

```
    if (written == -1)
      /* An error occurred; bail.  */
      return -1;
    else
      /* Keep count of how much more we need to write.  */
      leftfitofiwrite -= written;
  }
  /* We should have written no more than COUNT bytes!  */
  assert (leftfitofiwrite == 0);
  /* The number of bytes written is exactly COUNT.  */
  return count;
}
```

Chương III. Tiến Trình (process)

A. Tổng quan về tiến trình

1. Tiến trình

Unix là hệ đa xử lí, tức khả năng thực thi nhiều tác vụ cùng một lúc. Một chương trình máy tính là một chuỗi các chỉ lệnh (*intructions,* hay còn gọi là *lệnh máy*) mà theo đó máy tính phải thực hiện. Mặt khác tài nguyên máy tính (CPU, bộ nhớ, tệp, các thiết bị...) là hữu hạn và khi các chương trình chạy thì các chương trình đều có nhu cầu trên các tài nguyên đó. Để đáp ứng nhu cầu tài nguyên, cần có một sách lược chạy trình thật hiệu quả để đảm bảo tính đa nhiệm, nhiều người dùng. Cách phổ biến nhất là cấp tài nguyên cho mỗi chương trình trong một lượng thời gian nhất định, sao cho các chương trình đều có cơ hội thực hiện như nhau và trong thời gian thực hiện chương trình, cần kiểm soát việc thực hiện đó chặt chẻ. Để làm điều đó, ta đưa ra một khái niệm gọi là ***tiến trình* (*process*)**.

Vậy **tiến trình (TT)** là thời gian thực hiện (*instance of execution*) của một chương trình và việc thực hiện hiện đó chỉ xảy ra trong một khoản thời gian nhất định (gọi là *slice time*). Tuy nhiên để thực hiện được chương trình, TT sẽ sử dụng CPU để chạy các lệnh của nó, và bộ nhớ nơi có mã lệnh (*code* hay *text*), dữ liệu (*data*), và ngăn xếp (*stack).* Một TT khi thực hiện phải làm theo một trình tự các chỉ lệnh trong vùng *code* của TT và không nhảy tới

các chỉ lệnh của TT khác; TT chỉ có thể đọc/ghi truy nhập *data* và *stack* của nó, nhưng không thể trên *data* và *stack* của TT khác. TT liên lạc với các TT khác và phần còn lại của hệ thống bằng các Goị Hệ Thống (GHT, *system call*). Hệ thống phân chia việc thực hiện ra là hai chế độ, *user mode* và *kernel mode* như đã nói, nhưng cho dù như vậy, thì kernel vẫn là người thực hiện mã của TT (còn gọi là "nhân danh TT của người dùng"). Cần nhận thức rằng, kernel không phải là tập tách biệt các TT và chạy song song với TT người dùng, mà kernel là một phần của mỗi TT người dùng.

2. Môi trường thực hiện

2.1. Chế độ thực hiện

Việc thực hiện một TT trên Unix được chia ra làm hai mức: ***user*** (người dùng) và ***kernel*** (nhân của hệ thống). Khi một TT của user thực hiện một chức năng của nhân (thông qua gọi hệ thốngfiGHT), chế độ thực hiện của TT sẽ *chuyển từ chế độ người dùng (**user mode**) sang chế độ nhân của hệ thống (**kernel mode**):* Hệ thống sẽ thực hiện và phục vụ các yêu cầu của user, trả lại kết quả. Ngay cả khi user tạo ra các yêu cầu không tường minh, thì hệ vẫn thực hiện các kết toán có liên quan tới TT của user, thao tác các ngắt, lập biểu các TT, quản lí bộ nhớ...

Kernel mode là một chế độ đặc quyền, trong đó không có giới hạn nào đối với kernel: kernel sử dụng tất cả các lệnh của CPU, các thanh ghi của CPU, kiểm soát bộ nhớ, liên lạc trực tiếp với các thiết bị ngoại vi. Kernel tiếp nhận và xử lí các yêu cầu của các TT của user, sau đó gởi kết quả đến các TT đó.

User mode được hiểu là chế độ thực hiện bình thường của một tiến trình. Trong chế độ này, có nhiều hạn chế áp đặt lên TT: TT chỉ truy nhập được các lệnh và dữ liệu của nó, không thể truy nhập lệnh, dữ liệu của kernel và của các TT khác, một số các thanh ghi của CPU là cấm. Ví dụ: không gian địa chỉ ảo của một TT được chia ra thành miền chỉ truy nhập được trong chế độ kernel, miền khác ở chế độ user, hay TT không thể tương tác với máy vật lí, một số lệnh của CPU không được sử dụng, có thể bị ngắt trong bất kì lúc nào. Một TT trong user mode khi muốn truy nhập tài nguyên, phải thực hiện qua gọi hệ thống (GHT).

Gọi hệ thống (***GHT hay gọi thực hiện chức năng hệ thống cung cấp***) là quá trình chuyển thông số (yêu cầu qua *tên* hay *số* của các dịch vụ hệ thống) mà TT yêu cầu cho kernel thực hiện. Trong Unix, việc đó được làm qua một bẫy hệ thống (***trap***), sau đó kernel sẽ thực hiện nhu cầu của TT, đôi khi còn nói là: kernel thực hiện TT trên danh nghĩa của TT, trong môi trường của TT. Kernel không phảI là tập tách biệt của TT chạy song song với TT người dùng, mà là một phần của mỗi TT người dùng. Văn cảnh trình bày nói "kernel cung cấp tài nguyên" hay "kernel thực hiện ... " có nghĩa là TT đang chạy trong kernel mode cấp tài nguyên hay TT thực hiện . . .Bản chất của GHT để thực hiện các dịch vụ của kernel và mã thực thi các dịch vụ đó đã là một phần trong mã của TT người dùng, chỉ khác là mã đó chỉ chạy trong kernel mode mà thôi. Ví dụ: shell đọc đầu vào từ thiết bị đầu cuối bằng một GHT, lúc này kernel thực hiện nhân danh TT shell, kiểm soát thao tác của thiết bị đầu cuối và trả lại cho shell kí tự nhận được. Shell sau đó chạy trong user mode, thông dịch xâu kí tự và thực hiện các hành vi nhất định và có thể phát sinh GHT tiếp theo.

2.2. Môi trường thực hiện của Tiến trình

Như đã nói có rất nhiều TT được thực hiện đồng thời trong hệ thống, nhưng kernel cần lập

biểu để đưa vào thực hiện. Mỗi TT chỉ có một TT bố, nhưng có nhiều TT con của nó. Kernel nhận biết mỗi TT qua *số hiệu của TT* gọi là ***số định danh của TT (Procces ID: PID)***.

Khi dịch một chương trình nguồn, một tệp khả thi (*executable*) được tạo ra và nó có các phần sau:

- Tập các "*headers"* mô tả thuộc tính của tệp;
- Mã chương trình (*code* hay còn gọi là *text*);
- Một thể hiện của nguôn ngữ máy các dữ liệu được khởi động khi trình bắt đầu được thực hiện và một *chỉ báo* về kích thước (bộ nhớ) mà kernel cấp phát cho các dữ liệu chưa được khởi động.
- Và các thành phần khác, như bảng các biểu tượng.

Ví dụ: chương trình copy tệp

```
#include <fcntl.h>
char buf[1024];        /*là data chưa được khởi động*/
int version = 1;       /*là data được khởi động*/
main(argc, argv)
        int argc;
        char *argv[];
{
        int  fdold, fdnew;
        if (argc !=3)
        {
        printf ("can 2 doi dau vao cho trifnh copy\n");
        exit(1);
        }
        fdold=open(arg[1], OfiRDONLY);
        if(fdold ==-1){
                        printf("khong mo duoc tep %s\n", argv[1]);
                        exit(1);
                        }
        fdnew=creat(argv[2]. 0666);
        if(fdnew ==-1){
                        printf("khongtao duoc tep moi %s\n", argv[2]);
                        exit(1);
                        }
        copy (fdold, fdnew);
        exit(0);
}
copy(old, new)
```

```
        int old, new;
{
        int count;
        while((count = read (old, buffer, sizeof(buffer))) > 0)
                write(new, buffer, count);
}
```

Trong đó, *text* là mã tạo ra cho hàm *main*, dữ liệu được khởi động là biến *version* (int version = 1) và dữ liệu chưa khởi động có kích thước cho trước là trường *buffer*.

Kernel nạp một tệp thực thi (*executable)* bằng GHT *exec* (Unix có 6 chức năng exec là *execl(), execv(), execle(), execve(), execlp(), execvp())* và TT được nạp đó có ba phần gọi là *miền (region)*: ***mã lệnh (text*** hay ***code là xâu liên tục các lệnh), dữ liệu (data), ngăn xếp (stack).*** Hai miền đầu tương ứng với những xác định trong tệp, còn *stack* thì kernel tạo và điều chỉnh động vào thời điểm chạy trình. *Một TT trong Unix chạy trong hai chế độ nói trên nên có hai stack riêng biệt cho mỗi chế độ: user stack và kernel stack*.

User stack chứa các đối đầu vào, các biến cục bộ, các data khác cho các chức năng chạy trong chế độ *user mode*. Xem hình dưới, bên trái: Khi TT phát sinh gọi hàm copy() (frame 2) và hàm write() (frame 3) là hai khung stack liên tục trong hàm main(), trong khi frame1 là stack cho main() do khi hệ thống thực hiện exec() để gọi main(); Khi TT thực hiện một GHT (writre()), TT thực hiện một lệnh đặc biệt (lệnh *trap* cài vào mã khi dịch qua hợp ngữ), lệnh này tạo ra một "ngắt" chuyển CPU vào chế độ kernel, TT thực hiện lệnh của kernel và sử dụng kernel stack.

Kernel stack chứa các khung stack cho các chức năng (system calls) thực hiện trong *kernel mode*. Các đầu vào của hàm và dữ liệu trong kernel stack qui chiếu vào các hàm và dữ liệu bên trong của kernel, không phải các hàm và dư liệu của chương trình người dùng. Hình dưới, bên phải biểu diễn khi một TT gọi GHT write(). Tuy vậy cách cấu tạo stack thì như nhau ở cả hai.

Kernel stack của TT là *null* khi TT thực hiện trong user mode. Phần này sẽ bàn kĩ khi đề cập tới bối cảnh (***context***) của TT.

Mỗi TT có một đầu vào (*entry*) trong *Bảng các Tiến trình (**Process Table**)*. Mỗi TT lại được cấp một vùng dữ liệu gọi là ***ufiarea*** (*user area*) dùng để lưu các dữ liệu riêng của TT mà kernel sẽ xử lí. Thông tin trong cấu trúc này gồm có:

1. Các thanh ghi của CPU. Khi có bẫy vào kernel xuất hiện, các thanh ghi (kể cả đơn vị dấu phẩy động) được lưu lại tại đây.registers (including the floating-point ones, if used) are saved here.
2. Trạng thái gọi hệ thống: thông tin về GHT hiện tại, bao gồm cả các thông số lời gọi và cáckết quả thực hiện lời gọi đó.
3. Bảng mô tả các tệp của TT với **fd** là chỉ số trỏ vào bảng để định vị cấu trúc *in-core data* (i-node) tương ứng với tệp.
4. Con trỏ tới bảng kết toán thời gian sử dụng CPU của mỗi TT, cũng như các giá trị giới hạn hệ thống (max stack size, max page frames, v.v…).
5. **Kernel stack**. Là stack riêng của kernel

Process table có các thông tin:

1. Các thông số lập biểu chạy TT, mức ưu tiên của TT, lượng thời gian hiện đã dùng CPU, thời lượng ở trạng thái "sleep". Các thông số này cần để tuyển TT nào sẽ chạy tiếp theo trong lịch trình.

2. Nội dung bộ nhớ chứa mã thực thi. Các con trỏ trỏ tới bộ nhớ: *text, data, stack* , bảng các trang bộ nhớ của TT.

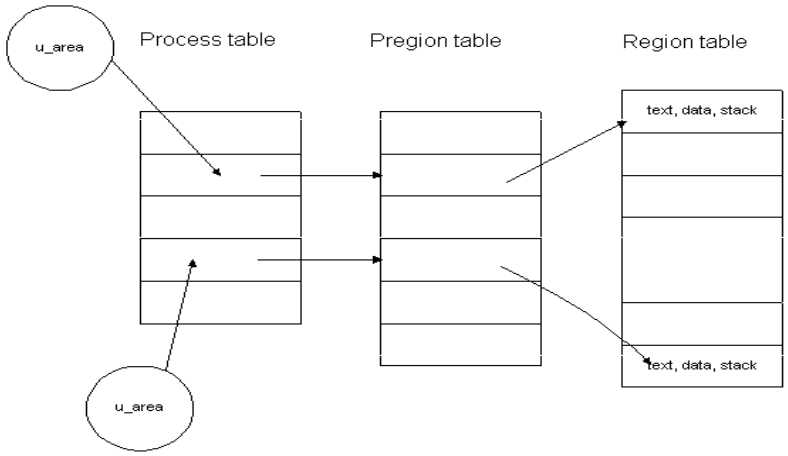
3. Tín hiệu và mặt nạ (mask) tín hiệu. Mask chỉ ra tín hiệu nào sẽ bỏ qua, tín hiệu nào sẽ xử lí, tín hiệu nào tạm thời để lại và tín hiệu nào TT đã gởi đi.

4. Các trạng thái hiện tại của TT, các sự kiện đang đợi sẽ xuất hiện, các timeouts, các số hiệu (PID) của TT , PID của TT bố, số hiệu tiến trình gắn với người dùng (UPID) và nhóm người dùng (GPID).

Khi TT gọi chức năng để thoát (***exit)***, kernel giải phóng các miền TT đã sử dụng.

Ví dụ chi tiết các cấu trúc dữ liệu của một TT đang thực hiện:

- Con trỏ trong *Procces table* trỏ tới Per procces region table (*pregion*);
- Con trỏ tại *Per proccess region table* (*pregion*) trỏ tới các đầu vào của *Region table* để trỏ tới các miền của text, data, stack của TT.
- Đầu vào của *Process table* và *u_area* (*user_area)* chứa thông tin điều khiển TT. *u_area* là phần mở rộng của đầu vào của *Proccess table*, ghi mới và cập nhật những thông tin trong suốt cuộc đời của TT.



Các trường trong Process table là:

- trường trạng thái;
- các nhận dạng cho biết người sở hữu TT: user IDs, EUIDs, GPID;
- tập mô tả các sự kiện khi treo TT (TT ở trạng thái ngủ).

Các trường quan trọng trong u_area và chỉ truy nhập được khi TT đang được thực hiện là:

- con trỏ trỏ tới đầu vào trong *Procces table* của TT đó;
- các thông số của GHT hiện tại: giá trị trả lại, mã lỗi;

- các mô tả tệp của tất cả tệp mở;
- các thông số I/O bên trong, bảng các khối đĩa chứa tệp trong FS đĩa;
- thư mục hiện hành và thư mục root;
- các giới hạn hệ thống trên tệp và TT.

Kernel có thể truy nhập vào *u_area* của TT đang được thực hiện, nhưng không thể được đối với các TT khác. Bên trong kernel qui chiếu tới biến *u (biến tổng thể của kernel)* của *u_area* của TT đang thực hiện và khi một TT khác đi vào thực hiện kernel sắp xếp lại không gian địa chỉ ảo của TT đó sao cho cấu trúc biến u sẽ trỏ tới *u_area* của TT mới (xem *context switch* sau này). Cách thức này giúp kernel dễ dàng nhận dạng một TT hiện tại bằng cách theo dõi con trỏ từ *u_area* tới đầu vào của TT trong Proccess table.

3. Bối cảnh của Tiến trình (Procces context)

Bối cảnh (context) *của TT là tập hợp các thông tin mô tả trạng thái của TT đó. Bối cảnh được định nghĩa bởi mã lệnh, các giá trị của các biến và các cấu trúc tổng thể của user, giá trị của các thanh ghi của CPU mà TT đang dùng, các giá trị trong proccess table và u_area của TT và nội dung các stack (ngăn xếp) trong user và kernel mode.* Phần mã, cũng như các cấu trúc dữ liệu tổng thể của kernel tuy có chia sẻ cho TT nhưng không thuộc bối cảnh của TT.

Khi thực hiện một TT, ta nói hệ thống được thực hiện trong bối cảnh của TT đó. ***Khi kernel quyết định cho chạy một TT khác, kernel sẽ chuyển đổi bối cảnh (switch context) sao cho hệ thống sẽ thực hiện trong bối cảnh của TT mới***. Kernel cho phép chuyển bối cảnh chỉ dưới những điều kiện nhất định. Khi chuyển bối cảnh, kernel bảo vệ các thông tin cần thiết để khi trở lại TT trước đó, kernel có thể tiếp tục thực hiện TT đó. Tương tự như vậy cũng xảy ra cho quá trình chuyển từ user mode sang kernel mode và ngược lại. Tuy nhiên *chuyển từ user mode sang kernel mode của một TT là sự thay đổi chế độ thực hiện, chứ không phải chuyển đổi bối cảnh. Bối cảnh thực hiện của TT không đổi, vì không có sự chuyển đổi thực hiện TT khác.*

Kernel phục vụ cho ngắt trong bối cảnh của TT bị ngắt cho dù TT đó không hề gây ra ngắt. TT bị ngắt có thể đã đang được thực hiện trong user mode hay kernel mode, kernel bảo vệ các thông tin cần để sau đó trở lại thực hiện TT bị ngắt và đi phục vụ cho ngắt trong kernel mode. Kernel không phát sinh hay lập biểu cho một TT đặc biệt nào đó để xử lí ngắt. *Trong bối cảnh đó, kernel tạo ra một layer mới trong kernel stack của TT để xử lí ngắt, sau khi hoàn tất ngắt, kernel trở về layer trước đó của TT đang thực hiện.*

4. Trạng thái của TT

Cuộc đời của TT có thể phân chia ra các trạng thái, mỗi trạng thái có các đặc tính mô tả về TT. Chương 6 sẽ nói đầy đủ các trạng thái tuy nhiên lúc này cần hiểu các trạng thái sau đây:

1. TT đang chạy trong user mode;
2. TT đang chạy trong kernel mode;
3. TT không được thực hiện (không chạy) nhưng sẵn sàng chạy khi bộ lập biểu chọn để thực hiện. Có rất nhiều TT trong trạng thái này, nhưng *scheduler* chỉ chọn một TT.
4. TT ngủ (*sleeping*): TT không thể thực hiện tiếp vì những lí do khác nhau, ví dụ đang đợi

để hoàn tất nhu cầu tài nguyên I/O.

Bởi vì CPU chỉ thực hiện một TT trong một thời điểm cho nên nhiều nhất chỉ có một TT trong trạng thái 1 hay 2. Hai trạng thái này tương ứng với user mode hay kernel mode.

Thực tế số các TT còn nhiều hơn như mô tả và sẽ đề cập chi tiết sau này.

5. Chuyển đổi trạng thái của TT

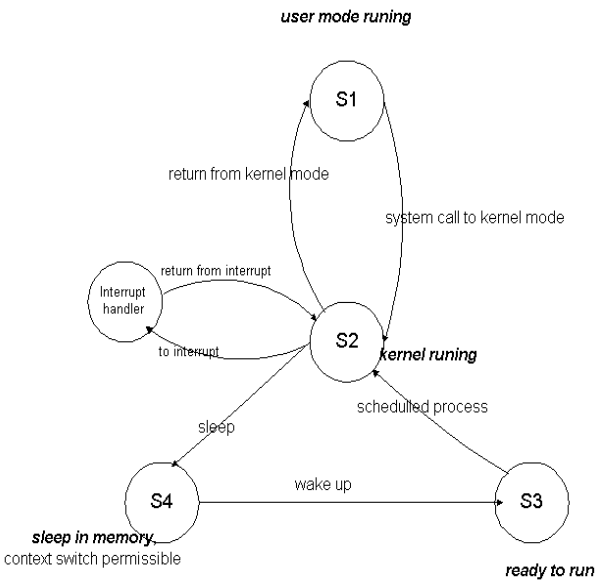
Phần trên là cách nhìn tĩnh (static) trạng thái của TT, tuy nhiên TT chuyển trạng thái liên tục và theo một qui luật được định nghĩa rõ ràng. Sơ đồ chuyển trạng thái là đồ thị mà các nút (*node*) biểu diễn trạng thái TT sẽ đi vào, thoát ra, các cạnh nối biểu diễn sự kiện làm cho TT chuyển đổi trạng thái. Sự chuyển đổi giữa hai trạng thái là hợp lí nếu có mũi tên nối từ trạng thái thứ nhất đến trạng thái thứ hai. Có nhiều chuyển đổi bắt nguồn từ một trạng thái nhưng TT chỉ theo một và chỉ một lí do, phụ thuộc vào sự kiện xuất hiện trên hệ thống. Hình dưới cho đồ thị theo định nghĩa nói trên.

Theo cách phân chia thời gian (*time - shared*), vài TT có thể thực hiện đồng thời trong kernel mode. Nếu TT chạy trong kernel mode mà không có các cưỡng chế, chúng có thể làm hỏng cấu trúc dữ liệu tổng thể của kernel. Bằng cách cấm chuyển đổi trạng thái một cách tuỳ tiện và kiểm soát nguyên nhân các ngắt, kernel bảo vệ được sự nhất quán của dữ liệu và tất nhiên cả bản thân kernel.

Kernel cho phép đổi bối cảnh chỉ khi TT chuyển từ trạng thái "đang chạy trong kernel" vào "ngủ trong bộ nhớ". Các TT chạy trong kernel mode không bị chen ngang (*preempted*) bởi các TT khác, do đó đôi khi ta nói kernel không thể chen ngang, trong khi đó ở user mode TT có thể bị chen ngang việc thực hiện. Việc không thể bị chen ngang làm cho kernel bảo trì sự nhất quán dữ liệu, và giải quyết được vấn đề loại trừ lẫn nhau (*mutual exclution*) - đảm bảo rằng các phần nhạy cảm (critical) của mã là được thực hiện bởi nhiều nhất một TT ở một thời điểm. (Phần mã gọi là nhạy cảm nếu việc thực hiện các xử lí ngắt tuỳ tiện có thể gây ra các tổn thất (làm hỏng) sự nhất quán dữ liệu).

Ví dụ: Xử lí ngắt của đĩa thao tác các hàng đợi đệm, thì phần mã lệnh mà kernel thực hiện thao tác trên hàng đợi đó là phần mã nhạy cảm khi nhìn tới sự liên quan với xử lí ngắt đĩa.

Nhìn lại ta thấy *kernel bảo vệ sự nhất quán bằng cách cho phép đổi bối cảnh chỉ khi TT tự đặt vào trạng thái ngủ và ngăn chặn một TT thay đổi trạng thái của một TT khác*. Kernel đồng thời gia tăng mức độ thực hiện (*run level*) của CPU đối với các vùng mã nhạy cảm để ngăn cản các ngắt có thể gây nên sự mất nhất quán dữ liệu. TT lập biểu (***scheduler process)*** một cách định kì, chen ngang các TT trong user mode để loại trừ trường hợp các TT có thể độc chiếm sử dụng CPU, và tuyển chọn một TT khác (theo các tiêu chuẩn nhất định) đưa vào hoạt động.



Trạng thái (giản lược) và sự chuyển đổi trạng thái của TT

6. TT ngủ (sleep) và thức dậy (wake up)

Một TT thực hiện trong kernel mode có khả năng tự trị trong việc quyết định sẽ làm gì khi phản ứng với các sự kiện của hệ thống. Các TT có thể liên lạc với nhau, đề xuất các giải pháp nhưng cuối cùng tự các TT sẽ quyết định. Có một tập hợp các luật (*rule*) mà các TT phải tuân thủ khi gặp các hoàn cảnh khác nhau, nhưng các TT sẽ theo các luật này để chủ động xử lí.

Một TT *đi ngủ* vì TT đang đợi sự xuất hiện của một sự kiện nào đó, chẳng hạn hoàn thành một yêu cầu I/O của t/b ngoại vi, hay đợi một TT kết thúc thực hiện, nguồn tài nguyên yêu cầu đã được giải trừ khoá... Nói rằng TT ngủ do một sự kiện có nghĩa rằng TT trong trạng thái ngủ cho tới khi sự kiện xuất hiện, và vào lúc đó TT *thức dậy* và chuyển vào trạng thái *sẵn sàng chạy*. Nhiều TT có thể cùng chờ chung một sự kiện (đều ngủ) và khi sự kiện đó đến tất cả các TT cùng thức dậy và chuyển từ trạng thái "ngủ" sang trạng thái "sẵn sàng chạy", tức sẽ chờ được *scheduler* sắp xếp lịch chạy. Các TT đang ngủ không sử dụng tài nguyên của CPU: kernel không phải kiểm tra để xem TT vẫn đang ngủ, nhưng sau đó sẽ đánh thức TT dậy khi sự kiện xuất hiện.

Đến đây ta có một số điểm cơ bản để mô tả về một TT:

- Mô tả về TT (định nghĩa, đặc tả các cơ sở dữ liệu về TT);
- Trạng thái của TT (đời sống của TT, qui luật TT hoạt động);
- Chế độ làm việc của TT (*user mode* và *kernel mode*) (ai chạy một chương trình, ở chế độ nào);
- Bối cảnh (context) của TT và quá trình chuyển bối cảnh của TT (các mô tả của TT thay đổi khi TT chuyển trạng thái);

  Các điểm này sẽ được nhắc lại chi tiết hơn ở phần tiếp theo

B. Cấu trúc của Tiến trình

Chương này đề cập đến:

- mở rộng các trạng thái của TT;
- các cấu trúc dữ liệu hệ thống quản lí TT;
- các nguyên tắc quảm lí bộ nhớ ảo, bộ nhớ vật lí của máy tính;
- làm rõ bối cảnhfi*context* của TT, thao tác *context* trong quá trình TT chuyển trạng thái (*save, switch, restore*);

1. Các trạng thái của tiến trình

Các trạng thái mà TT sẽ có trong suốt cuộc đời được mô tả chi tiết như sau, trong đó <...> là tác nhân gây ra sự chuyển trạng thái:

S1: TT thực hiện trong chế độ user;

S2: TT đang thực hiện trong chế độ kernel;

S3: TT không thực hiện, nhưng sẵn sàng thực hiện khi được *scheduler* chọn;

S4: TT "ngủ" trong bộ nhớ;

S5: TT sẵn sàng thực hiện nhưng TT đang ở trên *swap* (đĩa);

S6: TT "ngủ", nhưng TT hiện đang trên *swap*;

S7: TT đã chuyển từ kernel mode về user mode nhưng kernel đã chen ngang (*preempt*) TT, và đã tác động sự chuyển bối cảnh để lựa chọn một TT khác;

S8: TT được tạo ra ("*forked*") và đang ở trạng thái chuyển đổi: TT tồn tại, nhưng chưa sẵn sàng thực hiện, cũng không phải đi ngủ. *Đây là trạng thái khởi đầu của tất cả các TT*, trừ Tiến trình số 0. *Ta nói rằng TT bước vào mô hình trạng thái khi TT bố thực hiện fork(): tạo ra một TT;*

S9: TT đã thực hiện một *exit*, TT do vậy không còn tồn tại nữa nhưng TT vẫn còn để lại mã lệnh *return* và một số giá trị thời gian và sẽ được TT bố thu dọn. Trạng thái này là trạng thái cuối cùng của một TT, còn gọi là *zombie* (xác). (zombie: khi một TT chết, Unix không thể loại bỏ TT đó một cách hoàn tất khỏi hệ thống cho tới khi TT bố có sự hiểu biết về cái chết đó. TT bố biết được điều này khi thực hiện GHT *wait()*).

-Trên sơ đồ các mũi tên chỉ ra sự chuyển trạng thái còn bên cạnh <...> là các tác nhân gây ra. Một TT bước vào mô hình trên một khi TT bố thực hiện tạo mới (*forked*) một TT, sau đó sẽ đi vào S3 hay S5. Giả sử TT vào S3, và được chọn để chạy, TT vào S2 (kernel running) và tại đây TT hoàn tất phần tạo TT của GHT fork().

-Khi TT hoàn thành một GHT, nó *có thể* sẽ chuyển vào S1 (user running) và chạy trong user mode. Nếu lúc này, đồng hồ hệ thống ngắt CPU (sau một chu kì thời gian), CPU đI xử lí ngắt đồng hồ, khi kết thúc xử lí ngắt, bộ lập biểu (scheduler) có thể chọn một TT khác để chạy, TT trước đó không tiếp tục và ta nói nó bị chen ngang (preempted, S7). Trạng thái S7 thực tế giống như S3 nhưng được mô tả tách biệt để nhấn mạnh rằng một TT đang chạy trong kernel mode chỉ có thể bị chen ngang khi nó đang vận động vào user mode S1. Hệ quả của việc này là kernel có thể chuyển TT ở S7 lên swap nếu cần, hay nếu cuối cùng được chọn, TT sẽ *tiếp tục* chuyển vào S1 và thực hiện ở user mode.

-Khi TT thực hiện một GHT, TT đi từ *user mode* S1 vào *kernel mode* S2. Giả định GHT thuộc lớp các nhu cầu I/O, TT phảI đợi kết quả, TT đi vào S4 và cho tới khi I/O hoàn tất, TT sẽ được đánh thức và chuyển vào S3.

-TT có thể sẽ vào các trạng thái hoán chuyển fi *swap out, swap in* (bộ nhớ S4 → đĩa S6, đĩa S5 → bộ nhớ S3) do kernel cần bộ nhớ để chạy các TT khác. Như vậy một TT có hai hình thái của "sẵn sàng chạy" tùy thuộc vào vị trí mà TT đang "ở" đâu: trên đĩa hay trong bộ nhớ. Việc chuyển vị trí như vậy do TT 0 *swapper* lựa chọn, còn bản thân TT sẽ được thực hiện hay chưa lại phụ thuộc vào *scheduler*. *Scheduler* chọn TT để chạy lại trên cơ sở của các chính sách mà kernel vạch ra.

-TT có thể kiểm soát một vài sự chuyển trạng thái ở user mode.

1. TT có thể tạo ra một TT khác nhưng việc chuyển trạng thái vào S3 hay S5 phụ thuộc vào kernel: TT không kiểm soát được các trạng thaI này;
2. TT có thể phát sinh một GHT để chuyển từ S1 vào S2 và vào kernel mode nếu muốn, nhưng không thể kiểm soát việc ra khỏi kernel mode: các sự kiện có thể khiến TT không thể thoát ra và đưa TT vào S9 (sự phụ thuộc vào các tín hiệu signals);
3. TT có thể thoát ra khỏi sự tự ý nói trên do các sự kiện bên ngoài buộc mà không thực hiện tường minh một GHT *exit()*.

Tất cả các quá trình chuyển trạng thái đều theo một mô hình cứng nhắc đã mã hoá trong kernel, phản ứng trước các sự kiện theo cách thức đã dự báo trước theo các luật đã hình thành. Ví dụ như luật: *không TT nào được chen ngang một TT đang chạy trong chế độ kernel* như đã đề cập.

2. Các cấu trúc dữ liệu của tiến trình

Có hai cấu trúc dữ liệu của kernel dùng để mô tả một TT, đó là các thành phần của ***Bảng các TT (Proces Table)*** và một vùng nhớ cấp cho mỗi TT ***u_area*** *(viết tắt của user area**).***

Process Table có các đầu vào (*entry* hay *slot*) dành cho mỗi TT khi TT được tạo ra. Mỗi entry là một cấu trúc dữ liệu của mỗi TT mà các trường trong đó kernel luôn có thể truy nhập được, trong khi u_area với các trường chỉ có thể truy nhập được khi TT đang thực hiện. *U_area* chỉ phát sinh khi tạo ra một TT. Dưới đây là đặc tả các trường của một đầu vào (entry) cho một TT:

Mỗi đầu vào (entry) cho mỗi TT của Process Table có các trường sau:

1. *Trường trạng thái: cho biết trạng thái hiện tại của TT;*
2. *Trường định vị TT: cho biết vị trí của TT, của u_area của TT trong bộ nhớ hay trên swap. Kernel dùng các thông tin tại đây để chuyển đổi bối cảnh của TT khi TT từ "sẵn sàng thực hiện trong bộ nhớ "vào" thực hiện trong kernel ":S3 → S2; thực hiện hoán đổi giữa bộ nhớ ↔ swap; cho biết độ dài của TT sao cho kernel có thể cấp đủ không gian bộ nhớ cho TT;*
3. *Các định danh về người sử dụng (user IDs hay UIDs) cho biết user có các quyền hạn khác nhau trên TT;*
4. *Các định danh của TT (PIDs), cho biết mối quan hệ của TT với các TT khác (TT bố, TT con). Trường này sẽ phát sinh khi TT thực hiện một fork() để tạo ra TT (con) mới.*
5. *Trường mô tả các sự kiện mà TT hoài vọng sẽ đến khi TT ở trạng thái ngủ; ví dụ ứng dụng trường này có trong thuật toán sleep() và wakeup();*
6. *Các thông số lập biểu giúp kernel xác định thứ tự TT chuyển vào trạng thái " thực hiện trong kernel mode" và " thực hiện trong user mode".*
7. *Trường số hiệu các tín hiệu (signals) đã gởi cho TT nhưng chưa được xử lí.*
8. *Các số đếm qui chiếu thời gian TT đã dùng CPU và các nguồn tài nguyên khác của kernel. Các số liệu này giúp cho việc kết toán, tính toán mức độ ưu tiên khi lập biểu thực hiện TT. Một bộ đếm mà user xác lập giá trị được dùng để gởi signal thông báo (alarm signal) cho một TT khác;*

U_area (mở rộng của entry của TT) có các thông tin sau:

1. *Con trỏ trỏ vào đầu vào của một TT trong Process table cho biết u_area thuộc TT đó;*
2. *Định danh người dùng thực (real user ID) và định danh người dùng hiệu dụng (effective user ID), cho biết các quyền hạn khác nhau đối với từng loại user trên TT (ví dụ quyền truy nhập một tệp);*
3. *Các bộ đếm ghi nhận thời gian một TT (và các TT con của nó) đã trôi qua khi thực hiện trong user mode và kernel mode;*
4. *Bảng chỉ ra (trỏ tới) cách thức TT sẽ phản ứng khi có các signal;*
6. *Thiết bị đầu cuối (terminal) kết hợp với TT nếu có;*
7. *Các thông báo lỗi gặp phải khi thực hiện một GHT;*

8. *Chứa giá trị trả lại là kết quả (cho hàm gọi) khi thực hiện GHT;*
9. *Các thông số vào ra (I/O) cho biết khối lượng data chuyển đổi, địa chỉ nguồn data trong không gian địa chỉ của user, số đếm định vị (offset) của I/O, v.v.v.*
10. *Thư mục hiện tại, thư mục gốc của FS mà TT truy nhập tệp;*
11. *Các mô tả tệp (file descripters) các tệp TT đã mở;*
12. *Các giới hạn về kích thước của TT sẽ tạo hay của tệp mà TT có thể ghi vào;*
13. *quyền tạo tệp khi TT tạo tệp (ví dụ: tạo tệp chỉ đọc, hay W/R);*

3. Tổ chức bộ nhớ hệ thống

(Không đề cập)

4. Hoán đổi (swaping)

Việc lập biểu chạy TT phụ thuộc rất nhiều vào chiến thuật quản lí bộ nhớ. Với các TT ít nhất phải có một phần mã trong bộ nhớ chính để chạy, tức là CPU không thể chạy một TT mà mã của nó nằm toàn bộ trên bộ nhớ phụ (đĩa), trong khi đó bộ nhớ chính là tài nguyên đắt giá và thường không đủ lớn để chứa tất cả các TT đang hoạt động. Như vậy hệ quản lí bộ nhớ (*Memory Managment Subsystem MMS*) sẽ quyết định TT nào sẽ tồn tại (hay thường trú) trong bộ nhớ và quản lí các phần địa chỉ của không gian địa chỉ ảo không nằm trong bộ nhớ của TT. MMS kiểm soát bộ nhớ chính, ghi các TT vào bộ nhớ phụ gọi là *thiết bị hoán đổi (swap device)* để có nhiều bộ nhớ chính hơn và sau này khi TT được chạy lại sẽ nạp lại vào bộ nhớ. Unix truyền thống chuyển toàn bộ TT giữa bộ nhớ và *swap device* chứ không chuyển từng phần của TT một cách độc lập, trừ phần text chia sẻ. Sách lược quản lí như vậy gọi là hoán đổifi *swaping*. Với kĩ thuật này kích thước của một TT được gói gọn trong khuôn khổ số lượng bộ nhớ vật lí hệ có (ví dụ áp dụng trên PDP 11). Cho tới hệ Unix BSD phiên bản 4.0 và cho tới các phiên bản ngày nay đã triển khai kĩ thuật mới gọi là chuyển trang theo yêu cầu (*demand paging*), trong đó MMS hoán đổi các trang bộ nhớ thay vì cả TT lên *swap device* và ngược lại vào bộ nhớ: toàn bộ TT không thường trú trong bộ nhớ để thực hiện, kernel nạp các trang chỉ khi nào TT yêu cầu, trừ khi TT qui chiếu vào miền nhớ mà TT cần có để chạy. Kĩ thuật này mang lại các thuận lợi lớn cho phép quản lí và sử dụng bộ nhớ mềm dẻo và linh hoạt khi ánh xạ địa chỉ ảo của TT vào địa chỉ vật lí của hệ thống. Chính nhờ vậy mà kích thước của TT không bị hạn chế, đôi khi còn lớn hơn cả bộ nhớ thật và cho phép thường trú nhiều TT đồng thời trong bộ nhớ.

4.1. Thuật toán hoán đổi

Thuật toán hoán đổi gồm 3 phần:

- quản lí không gian trên *swap device*
- chuyển TT ra khỏi bộ nhớ lên *swap device*
- đưa TT trở lại bộ nhớ

5. Bối cảnh của TT (context of a Process)

Bối cảnh của TT có thể hình dung như "môi trường" trong đó TT sống và hoạt động. Trong "môi trường" như vậy bất kì sự thay đổi hay các tác động lên TT đều được ghi nhận, xử lí và chuyển giao khi TT có sự thay đổi trạng thái.

Như vậy, context của TT sẽ bao gồm *nội dung* của không gian địa chỉ của TT (lớp user) và *nội dung* của các thanh ghi phần cứng (mà TT đang sử dụng), *nội dung* các cấu trúc dữ liệu của kernel có liên quan tới TT.

Trên cơ sở định nghĩa trên, về mặt hình thức ta có thể chia *context* của TT ra thành:

- *User level context*: bao gồm *text, data, user stack*, vùng nhớ chia sẻ. Các vùng này thuộc không gian địa chỉ ảo của TT;
- *Register context*: gồm các thành phần sau đây:
  - *Bộ đếm chương trình (PC):* xác định địa chỉ tiếp theo CPU sẽ thực hiện. Địa chỉ đó có thể là địa chỉ ảo của kernel hay của không gian địa chỉ ảo của user;
  - *Thanh ghi trạng thái (PS) của CPU:* trạng thái của phần cứng liên quan tới TT, ví dụ như kết quả thực hiện các phép toán với giá trị âm, hay dương, hay 0, có bị tràn ô...
  - *Stack pointer:* địa chỉ hiện tại của vị trí tiếp theo trong *kernel stack* hay *user stack* được xác định bởi chế độ thực hiện;
  - *General purpose register:* các thanh ghi vạn năng chứa *data* do TT tạo ra trong quá trình thực hiện của TT.
- *System level context*: bối cảnh hệ thống có hai phần:
- **Static:** là phần tồn tại trong suốt cuộc đời của TT:
  - Các trường trong đầu vào cho mỗi TT của *process table*, cho biết trạng thái của TT và các thông tin kernel truy nhập để kiểm soát TT;
  - *U_area* chứa các thông tin của TT chỉ truy nhập được trong bối cảnh của TT;
  - *Pregion, region table, page table* của TT định nghĩa cách ánh xạ địa chỉ ảo tới địa chỉ vật lí. Chú ý là nếu có một miền nào đó chia sẻ chung cho các TT, thì miền đó được xem như là bộ phận của TT vì rằng mỗi TT thao tác miền đó một cách độc lập.
- **Dynamic:** TT có thể có nhiều phần động và coi như các lớp (*stack layers*) mà kernel đẩy (*push*), hay lấy lại (*pop*) nội dung trên sự hiện diện của các sự kiện (*events*):

  Kernel stack lưu các khung stack của các xử lí trong kernel mode, mỗi khung ứng với một *layer* (TT phát sinh của TT đó). Các TT đều có bản copy của *kernel stack* xác định từng kiểu kích hoạt các chức năng của kernel. Ví dụ TT kích hoạt GHT *creat()*, đi ngủ đợi kernel cấp một ifinode, TT khác kích hoạt GHT *read()*, đi ngủ đợi kết quả đọc từ đĩa xuống buffer. Cả hai TT thực hiện các hàm chức năng của kernel và mỗi TT có stack riêng để lưu các trình tự TT thực hiện. Kernel phải có khả năng khôi phục lại nội dung của *kernel stack* và vị trí của con trỏ stack sao cho có thể tiếp tục thực hiện TT trong kernel mode. Kernel stack của TT thường được đặt trong *U_area* của TT đó, và rỗng khi TT chạy trong user mode.

Phần động của *system level context* của TT là tập của các lớp (*layers*) bối cảnh của TT đó sắp xếp theo LIFO. Mỗi *layer* chứa các thông tin cần thiết để trở về *layer* trước đó kể cả *registerficontext* của *layer* trước đó.

Kernel đẩy (thay đổi) *layer* khi *interrupt* xuất hiện hay khi TT thực hiện một GHT hay khi TT chuyển đổi bối cảnh và lấy lại (*pop*) bối cảnh khi trở về sau xử lí ngắt, hay khi trở lại user mode sau khi đã hoàn tất thực hiện một GHT hay khi TT chyển đổi bối cảnh. Chuyển bối cảnh thực tế là kernel đẩy *context layer* "cũ" (TT cũ) vào stack và tạo *layer* mới cho TT mới. Kernel quản lí các TT qua các đầu vào cho mỗi TT trong *Process table*. Cấu trúc *stack* này là một thành phần của mỗi TT và đặt trong *u_area* của TT đó.

6. Ghép nối với GHT

GHT ghép vào kernel đã được đề cập như gọi một chức năng hệ thống. Nhưng với cách gọi như vậy thì không thể thay đổi chế độ thực hiện TT từ *user mode* vào *kernel mode*. Bộ dịch C dùng thư viện các chức năng đã định nghĩa (*C - library*) có tên của các GHT để giải quyết các qui chiếu GHT trong chương trình của user tới những tên đã được định nghĩa. Các chức năng của thư viện C thông thường sẽ kích hoạt một lệnh (*instruction*) và lệnh này sẽ làm thay đổi chế độ thực hiện của TT từ *user mode* vào *kernel mode* và làm cho kernel đi thực hiện mã của GHT. Lệnh này thường được gọi là bẫy hệ điều hành fi *operating system trap*. Các hàm thư viện sẽ chuyển cho kernel số của GHT theo cách thức phụ thuộc vào từng loại máy tính, hoặc như một thông số cho *trap* qua một thanh ghi xác định hay để nó ở stack. Từ đó kernel sẽ tìm ra được GHT mà user đã kích hoạt.

Thuật toán *syscall*

```
Input:   system call number (số của GHT)
Output:  kết quả thực hiện GHT
{
    .find entry in system call table corresponding to system call number;
    .determine number of parameters to system call;
    .copy parameters from user address space to u_area;
    .save current context for abortive return if there is;
    .invoke system call code in kernel;
    .if (error during execution of system call)
            {
                .set reg0 in user saved register context to error number;
                .turn on carry bit in status reg PS in user saved register context;
            }
    else
            .set reg0, reg1 in user saved register context to return values from system call;
}
```

ở đây *reg0* và *reg1* là hai thanh ghi vạn năng của CPU.

7. Chuyển bối cảnh (context swich)

Qui vào mô hình chuyển trạng thái của TT, ta thấy kernel cho phép chuyển bối cảnh trong các hoàn cảnh sau:

S2 → S4: TT tự đưa vào trạng thái ngủ (*sleep*), vì rằng sẽ phải đợi 1 khoảng thời gian trôi qua cho nhu cầu có được tài nguyên (I/O: đọc/ghi data ↔ đĩa, inode đang locked chưa được giải khoá...), thời gian này là đáng kể, và kernel chọn TT khác có thể được thực hiện;

S2 → S9: TT thực hiện một *exit()* và việc chuyển bối cảnh sẽ xảy ra lúc kết thúc *exit* vì sau đó không còn gì phải làm với TT đó, kernel tuyển chọn một TT khác để thực hiện;

S2 → S1: Hoàn tất một GHT hay khi kernel hoàn tất xử lí ngắt nhưng bản thân TT không phải là TT có nhiều khả năng nhất để tiếp tục chạy, nhờ vậy các TT khác có mức ưu tiên cao hơn có cơ hội để thực hiện;

Kernel đảm bảo sự thống nhất toàn vẹn của các cơ sở dữ liệu bằng cách cấm chuyển đổi bối cảnh một cách tuỳ tiện: trạng thái của cấu trúc dữ liệu là bền vững trước khi chuyển bối cảnh (tất cả các dữ liệu được cập nhật xong, các hàng đợi liên kết chuẩn xác, các khoá (*lock*) được lập, nhằm ngăn cản sự truy nhập bởi các TT khác v.v...).

Các bước chuyển bối cảnh:

{

1.Decide whether to do a context switch and whether a context switch is permissible now; /*sách lược*/

2.Save context of"old" process;

3.Find the "best" process to schedule for execution;

*4.If any, restore its context;/*chuyển sang context của*/*

*/*TT được chọn: khôi phục */*

*/*lại layer cuối cùng lúc TT*/*

/ này chuyển bối cảnh */*

}

8. Thao tác không gian địa chỉ của TT

Cho tới đây ta đã có được mô tả cách kernel thay đổi bối cảnh thực hiện giữa các TT và cách kernel tạo các lớp (hay khungfi*layer)* của stack. Đối với các miền trong không gian địa chỉ của TT, kernel cung cấp các GHT để thao tác các miền đó. Để làm được việc này, cần có các cấu trúc dữ liệu mô tả miền cũng như định nghĩa các thao tác tương ứng trên miền. Có một thành phần trong bảng các miền (r*egion table)* cho các thông tin cần thiết để mô tả một miền, đó là:

1. con trỏ trỏ vào inode của tệp mà nội dung của tệp đã nạp vào miền đó;
2. kiểu của miền (là *code*, hay *stack*, hay *data*, hay miền nhớ chia sẻ hoặc của riêng của TT);
3. kích thước của miền;

4. vị trí của miền trong bộ nhớ vật lí (qua *pages table*);
5. Trạng thái của miền (đã khoá(*locked*), miền đang yêu cầu cấp cho TT, đang được nạp vào bộ nhớ; miền hợp pháp, đã nạp vào bộ nhớ);
6. Số đếm qui chiếu cho biết tổng số các TT có truy nhập đến miền (miền có kiểu (thuộc tính) chia sẻ).

Các thao tác trên miền bao gồm:

9.1. Khoá (lock) và giải trừ khoá (unlock) miền (bộ nhớ)

Kernel cho phép thao tác khoá và giải khoá một miền (tương tự như các thao tác trên inode). Kernel có thể khoá và cấp một miền sau đó giải khoá mà không cần tới giải phóng miền. Kernel thao tác miền đã cấp, khoá lại để ngăn cản cácTT sẽ truy nhập vào miền sau đó.

9.2. Cấp phát miền (allocreg()).

Khi phát sinh GHT *fork(), exec()* hay *shmget()* (chia sẻ miền), kernel sẽ cấp một miền mới cho TT. Kernel dùng bảng các miền (*region table*) với các thành phần (đầu vào) liên kết tới một danh sách các miền còn tự do (*free*) hay danh sách đang sử dụng (*active*). Khi cấp phátfi*allocreg()*, một miền còn tự do tìm thấy đầu tiên trong danh sách *free* sẽ được đưa và danh sách *active* và gán các thuộc tính cần thiết. Thông thường mỗi TT đều liên kết với một tệp thực thi do *exec()* thực hiện, cho nên *allocreg()* sẽ đặt con trỏ inode trong *region table* trỏ vào inode của tệp thực thi đó. Khi miền có tính chia sẻ, kernel tăng số đếm qui chiếu đến miền khi có TT khác truy nhập, ngăn cản một TT thực hiện giải phóng miền khi chạy *link()*. Hàm *allocreg()* trả lại miền được cấp và đã khóa.

9.3. Ghép một miền vào TT (attachreg())

Khi thực hiện *fork(), exec()*, hay *shmat()* để ghép miền vào không gian địa chỉ của TT. Miền ghép có thể là một miền mới, hay một miền chia sẻ với các TT khác. Kernel mở một đầu vào trong *pregion table* của TT, xác định kiểu của miền, ghi nhận địa chỉ ảo của miền trong không gian địa chỉ ảo của user.

9.4. Thay đổi kích thước miền của TT (growreg())

Khi TT mở rộng miền bộ nhớ kernel phải thực hiện một số việc:

- Không để chồng chéo địa chỉ ảo (*overlaping*),
- Miền địa chỉ tổng cộng không vượt quá giới hạn cho phép trong không gian địa chỉ ảo của hệ thống.

9.5. Nạp một miền (loadreg())

Đối với các hệ dùng kĩ thuật phân trang (*paging*), kernel sẽ ánh xạ nội dung của một tệp vào không gian địa chỉ ảo của TT trong quá trình thực hiện *exec()* và sắp xếp để truy nhập các trang vật lí riêng biệt khi có yêu cầu (*in demand*).

Nếu hệ không dùng *paging*, kernel phải *copy* tệp thực thi vào bộ nhớ: nạp các thông số các miền của TT vào các địa chỉ ảo khác nhau mà tại đó tệp sẽ được nạp. Kernel có thể sẽ nối một miền ở các vùng địa chỉ ảo khác nhau nơi sẽ nạp nội dung của tệp, và do vậy sẽ có những chỗ cách quãng (*gap*) trong *page table*. Để nạp một tệp vào một miền, hàm *loadreg()* sẽ tính độ lớn của *gap* chen giữa các miền địa chỉ ảo sẽ nối vào cho TT, địa chỉ ảo khởi đầu của miền, mở rộng miền theo lượng bộ nhớ miền cần có. Tiếp theo đặt miền vào trạng thái " đang nạp nào bộ nhớ", đọc dữ liệu từ tệp vào miền (bằng GHT *read()*).

Nếu miền *text* đang nạp là chia sẻ cho nhiều TT, có thể miền sẽ bị truy nhập vào thời điểm này, trong khi nội dung chưa nạp hoàn tất, do vậy kernel sẽ kiểm tra cờ (*flag*) trạng thái của miền, đưa TT vào *sleep*. Vào thời điểm cuối cùng của *loadreg(),* kernel sẽ đổi trạng thái miền thành sẵn có (*valid*) trong bộ nhớ, đánh thức các TT đợi miền để các TT truy nhập.

9.6. Giải phóng một miền của TT (freereg())

Khi một miền không còn nối vào TT, kernel sẽ giải phóng miền đó, đưa miền vào danh sach liên kết của các miền tự do, bao gồm:

- Giải phóng miền địa chỉ ảo đã cấp,
- Giải phóng các trang địa chỉ vật lí kết hợp với địa chỉ ảo.

9.7. Tách một miền khỏi TT (detachreg())

Là chức năng ngược lại của *attachreg()*, thực hiện khi kết thúc *exec()*, thực hiện *exit(),* hủy chia sẻ *shmdt()*:

- Cập nhật mới các *register*, các đầu vào của *pregion table*;
- Nếu số đếm qui chiếu vào miền = 0, và không còn lí do gì để tồn tại (chia sẻ chẳng hạn) kernel sau đó sẽ giải phóng miền (*freereg()*)
- Nếu số đếm qui chiếu vào miền ≠ 0 thì giải trừ *lock* cho miền và inode tương ứng để TT khác có thể truy nhập.

9.8. Nhân bản miền của TT (dupreg())

Khi thực hiện *fork()*, kernel sẽ nhân bản *context* của TT gọi *fork()* (TT bố) để tạo *context* cho TT con.

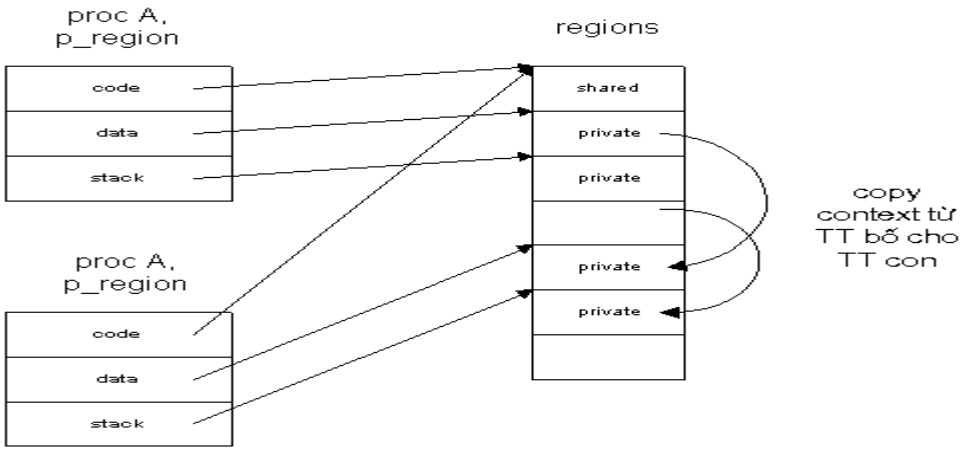
- Nếu miền là chia sẻ (*code, data*), kernel sẽ không nhân bản địa chỉ của miền, chỉ cần tăng số đếm qui chiếu tới miền,
- Nếu miền không chia sẻ, kernel thực hiện copy miền:

  .cấp đầu vào mới trong *region table*,

  .cấp các đầu vào mới trong *page table*,

  .cấp vùng nhớ vật lí mới cho TT.

Ví dụ: TTA (bố) *fork* ra TT B (con), ta sẽ có:



dupreg(): Nhân đôi miền, ví dụ khi fork()

10. Tiến trình ngủ (sleep)

Phần trên đã đề cập đến các chức năng cấp cơ sở liên quan đến sự chuyển trạng thái của TT từ "sẵn sàng thực hiện trong bộ nhớ" đến/hay trở lại "thực hiện trong kernel". Tiếp theo sẽ là thuật toán mô tả TT đi vào trạng thái "ngủ trong bộ nhớ": S2 → S4. Sau đó sẽ là thuật toán đánh thức TT, đưa TT vào trạng thái "sẵn sàng thực hiện trong bộ nhớ" S4 → S3 hay S3 → S5 "sẵn sàng thực hiện" nhưng TT ở trên *swap* (đĩa).

Khi một TT đi ngủ (thường là khi thực hiện một GHT), TT thực hiện một bẫy hệ thống (*trap*).

11. Sự kiện (event) và địa chỉ (addres)

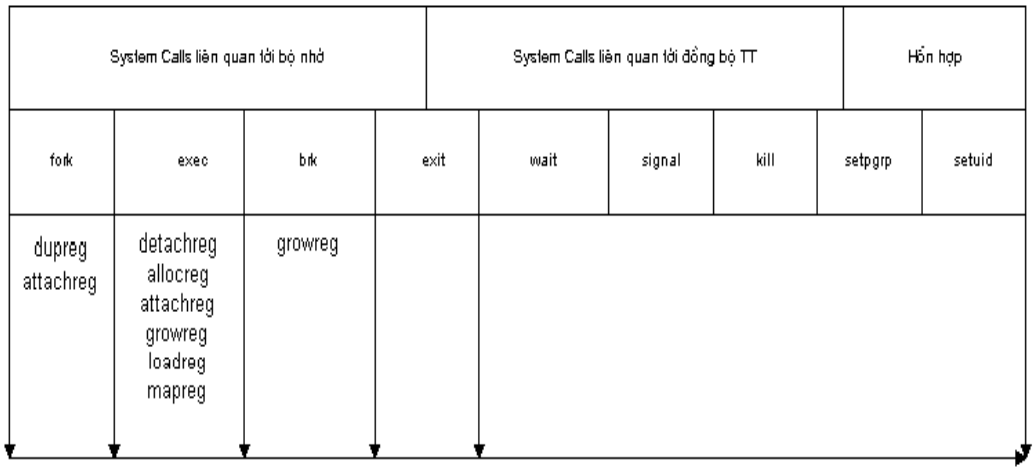
Nói rằng TT đi ngủ do chờ đợi một sự kiện nào đó, thông thường là cho tới khi sự kiện xuất hiện và vào lúc đó TT thức tỉnh và đi vào trạng thái " sẵn sàng thực hiện " hoặc trong bộ nhớ hoặc trên *swap out*. Mặc dù hệ thống nói tới "ngủ" một cách trừu tượng, nhưng cách áp dụng là để ánh xạ tập các sự kiện vào tập các địa chỉ ảo của kernel. Địa chỉ để biểu diễn các sự kện được mã hoá trong kernel, và kernel chờ đợi sự kiện để ánh xạ nó vào một địa chỉ cụ thể. Tính trừu tượng dẫn đến vấn đề là không thể phân biệt được có bao nhiêu TT đang đợi một sự kiện và dẫn đến hai điều không bình thường: khi một sự kiện xuất hiện, thì tất cả các TT ngủ đợi đều được đánh thức, và tất cả đều chuyển vào trạng thái " sẵn sàng chạy ". Điều thứ hai là có vài sự kiện lại cùng ánh xạ tới một địa chỉ (đích). Để giải quyết, kernel sẽ sử dụng giải pháp ánh xạ một - một (*one to one*), và rằng thực tế cho thấy ánh xạ đa sự kiện vào một địa chỉ rất ít xảy ra vì một TT đang chạy thường giải phóng tài nguyên đã khóa, trước khi có TT khác được lập biểu để chạy. TT lập biểu nếu không nhận được tài nguyên thì lại chuyển vào sleep.

C. Kiểm soát tiến trình

Phần trước đã định nghĩa *context* và giải thích các thuật toán thao tác những gì liên quan tới *context*. Phần này sẽ mô tả việc sử dụng và áp dụng các GHT dùng để *kiểm soát bối cảnh của TT*, hay ngắn gọn là kiểm soát TT.

Các GHT sẽ đề cập bao gồm:

- *fork()*: tạo ra một TT mới,
- *exit():* kết thúc thực hiện một TT,
- *wait():* cho phép TT bố đồng bộ việc thực hiện của mình với việc kết thúc thực hiện của TT con,
- *signal():* thông báo cho các TT về các sự kiện (không đồng bộ) xuất hiện, kernel lại dùng *signals* để đồng bộ việc thực hiện của *exit* và *wait*,
- *exec():* cho phép một TT kích hoạt một chương trình "mới". Bản chất của exec là biến TT gọi thành TT mới. TT mới là một tệp thực thi và mã của nó sẽ "ghi đè" lên không gian của TT gọi, ví vậy sẽ không có giá trị trả về khi exec thành công
- *brk():* TT thuê thêm bộ nhớ một cách linh động
- các kiến trúc chu trình chính của *shell() và init().*

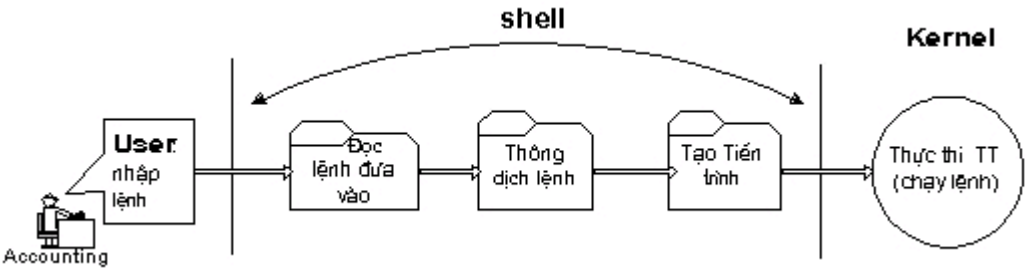


1. Tạo tiến trình

Tạo ra các TT là một điểm mạnh trong Unix. Vậy tạo ra các TT để làm gì. Người lập trình luôn quan tâm tới khả năng thực hiện nhiều tác vụ đồng thời trong khi phát triển một ứng dụng, trong khi đó cũng muốn sử dụng lại các tiện ích, các hàm đã có để nâng cao năng lực, hiệu quả tính toán của máy tính. Tạo TT là giảI pháp đáp ứng yêu cầu trên, bởi vì một TT được tạo ra sẽ chạy song song với TT đã tạo ra nó. Đó cũng chính là sức mạnh đa nhiệm mà Unix có. Ta hãy theo dõi quá trình sau:

Khi thực hiện một lệnh máy, còn gọi là command (**cmd**), một qui trình được thực hiện bao gồm:

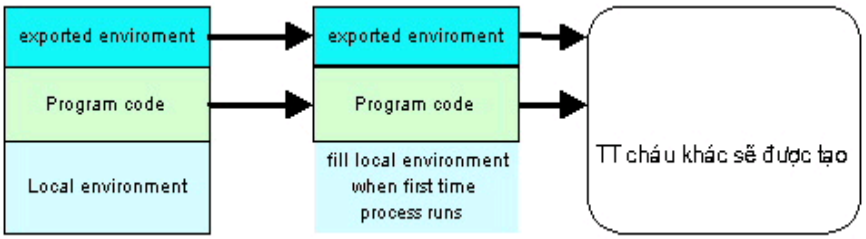
Giao diện người – máy, *shell,* nhận lệnh (cmd) user đưa vào, sau đó shell tạo ra một TT (con) để thực hiện cmd đó. Quá trình này xảy ra qua hai bước:



1. shell tạo ra một TT con để chạy cmd, tức tạo ra một bối cảnh sao cho lệnh có thể thực hiện được. Quá trình này thực sự là việc sao chép bối cảnh của TT bố (shell) cho TT mới (TT con). Như vậy khi thực hiện cmd thì cmd này sử dụng các môi trường của TT bố đã sao chép cho nó. Tuy nhiên không phải tất cả môi trường của TT bố được sao chép cho TT con, mà chỉ có các phần tổng thể được sao chép, và phần này bao gồm:

   -*môi trường xuất* (exported environment): -UID, GID của TT bố, các đầu vào/ra chuẩn (stdin, stdout), các tệp đã mở, thư mục root, thư mục hiện hành, và các thông tin hệ thống khác, danh sách các biến tổng thể (global variables) và một vài biến trong môI trường riêng của TT bố cũng có thể xuất cho TT con.

   -*mã chương trình*.



Chuyển thông tin từ TT bố cho TT con, các phần fork sao chép

TT gốc TT tạo mới

2. Thực hiện cmd trong bối cảnh của TT mới được tạo (sẽ do scheduler sắp xếp: S8->S3 hay S8->S5). TT con sẽ trả lại các kết quả khi kết thúc thực hiện cmd qua **exit()**.

Sự tạo lập một tiến trình mới thực hiện bằng lệnh gọi hệ thống *fork. Fork()* cho phép một tiến trình lập một bản sao của nó, trừ bộ định dạng tiến trình. Tiến trình gốc tự nhân bản chính nó được gọi là tiến trình bố và bản sao tạo ra được gọi là tiến trình con.

Cú pháp như sau:

pid = fork();

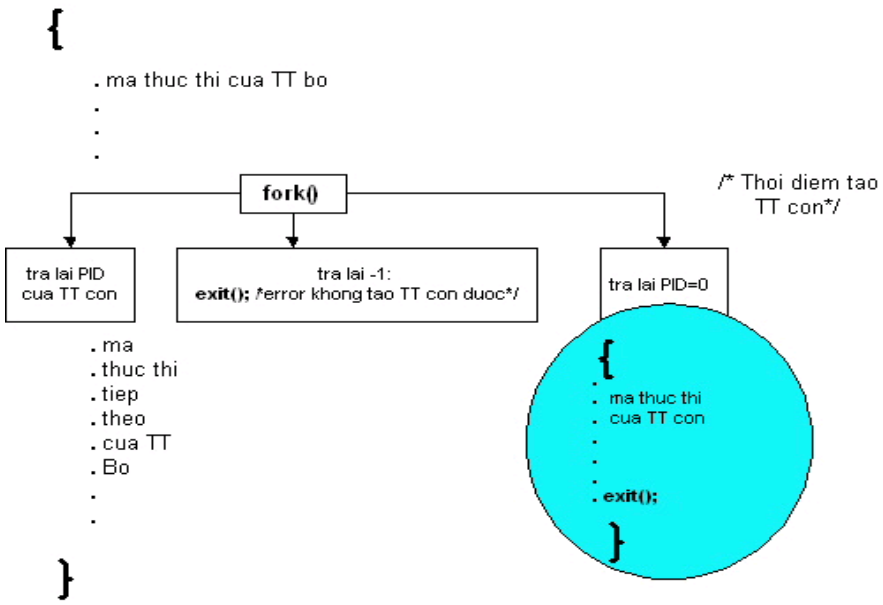
Khi thực hiện xong, hai TT nói trên có hai bản sao ***user level context như nhau*** (*exported environment* ở mô hình trên), và với các giá trị trả lại *pid* khác nhau:

- trong bản thân mã của TT bố, *pid* là số định danh của TT con;
- trong bản thân mã của TT con, *pid* = 0 thông báo kết quả tạo TT là tốt;

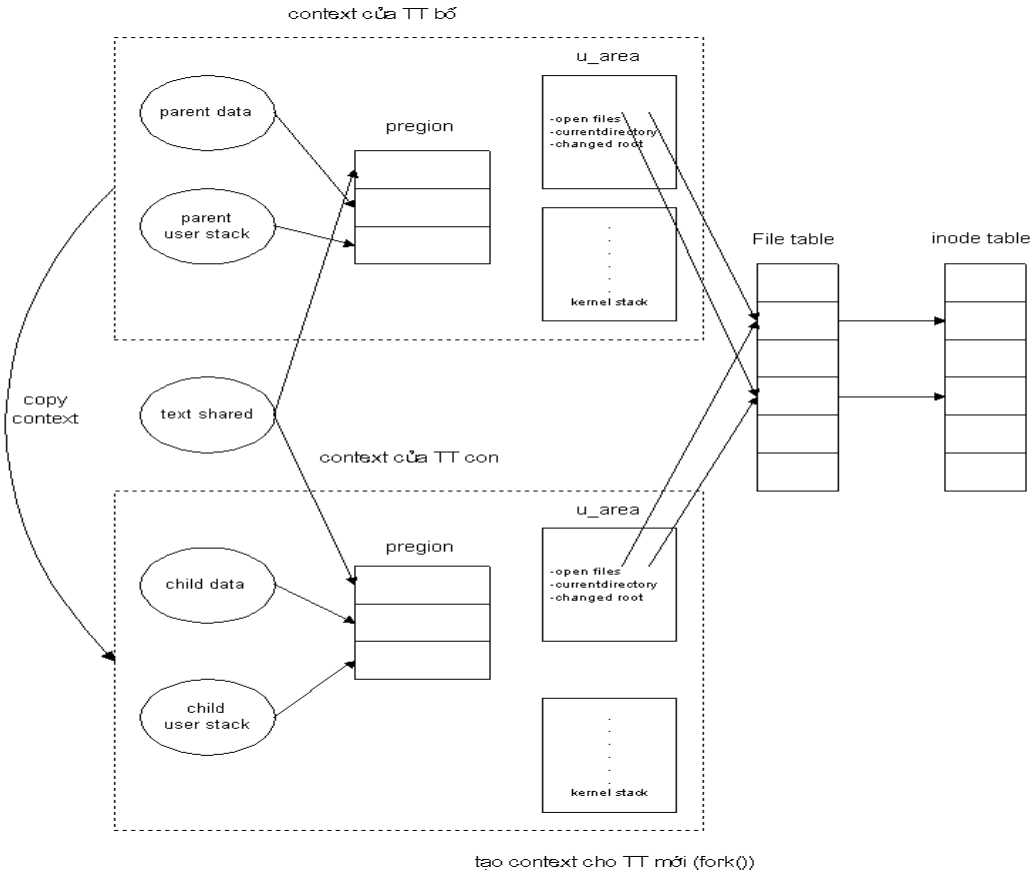
(Chú ý: ở đây không phải là TT 0 *Init* như đã nói, TT duy nhất không tạo ra bằng *fork()*).
Kernel thực hiện các bước sau khi *fork()*:

Các bước thực hiện fork như sau:

1. Cấp cho TT mới một đầu vào (*entry*) trong *procces table*;
2. Gán cho TT con một số định danh duy nhất;
3. Tạo ra bản sao *logic* bối cảnh của TT bố cho TT con. Vì một số phần của TT có thể chia sẻ (ví dụ miền *text* chẳng hạn) giữa các TT, kernel sẽ tăng các số đếm qui chiếu vào các miền đó thay vì sao chép miền trong bộ nhớ vật lí.
4. Tăng số đếm qui chiếu trong bảng các tệp mở (*file table)* kết hợp với TT (mà TT bố đã mở, TT con thừa kế sử dụng, tức là cả hai thao tác trên cùng một tệp), bảng inode (*inode table*) vì TT con tồn tại trong cùng thư mục hiện hành của TT bố, nên số truy nhập tệp thư mục +1.
5. Trả lại số định danh *pid* của TT con cho TT bố.



Các cơ sở dữ liệu của kernel sau khi fork() :



fork() tạo ra bối cảnh cho TT mới

Thực tế thuật toán *fork()* không đơn giản vì TT mới có thể đi vào thực hiện ngay và phụ thuộc vào hệ thống kiểu *swaping* hay *demand paging*. Để đơn giản chọn hệ với *swaping* và giả định rằng hệ có đủ bộ nhớ để chứa TT con, thuật toán sẽ như sau:

fork()

input: none

output: - PID for parent process / Process ID*/*

- 0 to child process;

{

.check for available kernel resources;

.get free process table entry (slot), unique ID number;

.check that user not running too many processes;

.mark child state is " being created";

*.copy data from parent process table entry to new child entry;/*dup*/*

.increment counts on current directory inode and changed root (if applicable);

*.increment open file counts in file table;/*Ref. count of fds = +1*/*

.make copy of parent context (u_area, text, data, stack) in memory;

*/*create user level context (static part) now in using dupreg(), attachreg(). Except that the content of child's u_area are initally the same as of parent's, but can diverge after completion of the fork: the pionter to its process table slot. Note that, if nows parent open new file, the child can't access to it !!! */*

*.push dummy system level context layer onto child system level context; /*Now kernel create the dynamic context for child: copy parent context layer 1 containing the user saved register context and the kernel frame stack of the fork system cal. If kernel stack is part of u_area, kernel automaticlly create child kernel stack when create the child u_area. If other method is used, the copying is needed to the private memory associated with chile process. Next kernel creates dummy context layer 2 for child containing the saved register context for lauer 1, set program counter . other regs to restore the child context, even it had never executed before and so that the child process to recognize itself as child when it runs.*

Now kernel tests the value register reg0 to decide if process is parent (reg0=1) or child (reg0=0)./;*

.if (executing process is parent process)

{ / After all, kernel set child to "ready to run in memory", return PID to user */*

.change child state to "ready to run";

*.return(child PID); /*from system to user*/*

}

.else

{ / and it start running from here when scheduled executing process is the child process: Child executes part of the code for fork() according to the program counter that kernel restored from the saved regiter context in context layer 2, and return a 0 from the fork()*/*

initialize u_area timing field;

*return(0); /*to user*/*

}

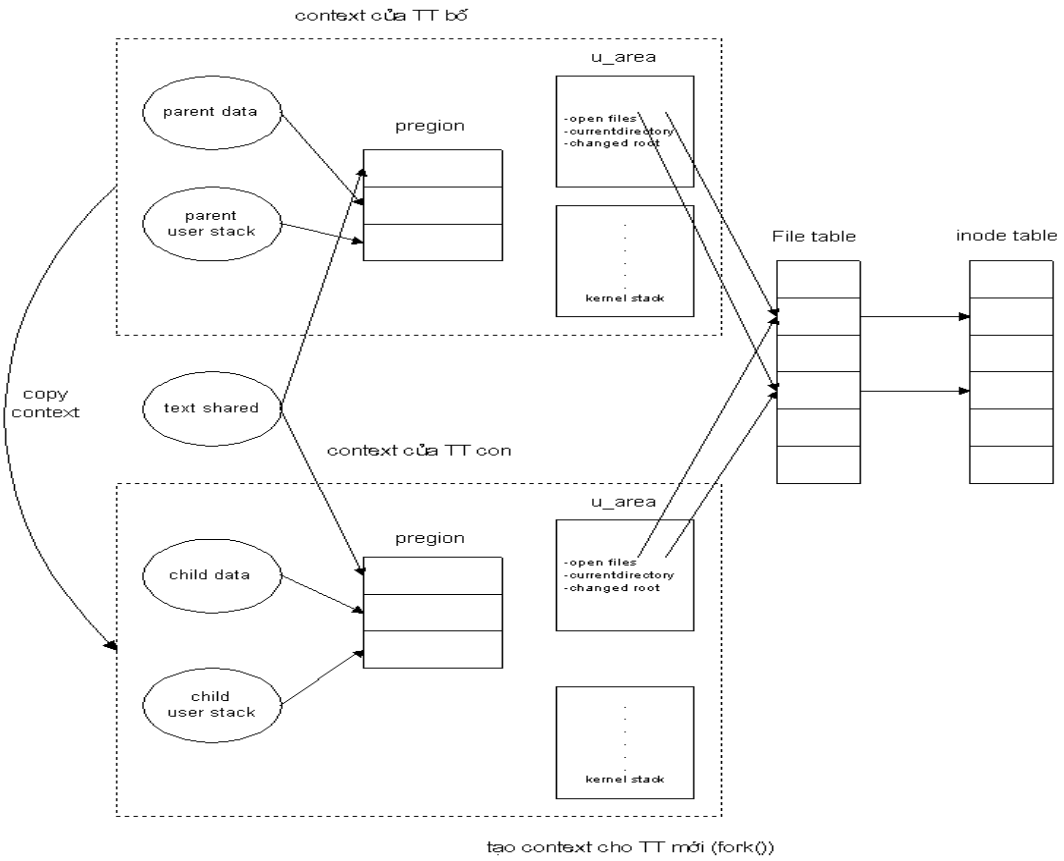
}

Trong thuật toán này cần lưu ý một số điểm: đối với hệ *swaping* cần có chỗ để tạo TT con hoặc trong bộ nhớ hoặc trên *swap* (đĩa). Đối với hệ *paging* phải cấp phát các trang trong *page table*. Nếu không đủ bộ nhớ, *fork()* sẽ không thành công, TT con không được tạo ra. Số PID là số duy nhất cho mỗi TT, thường là số lớn hơn 1 so với số cuối cùng đã dùng. Tuy nhiên do một số TT có cuộc đời ngắn nên PID có thể sẽ được dùng lại. Hệ thống đồng thời đặt một số giới hạn trên tổng số TT cho một user sao cho hệ không bị kẹt (*deadlock*) do phải đợi tài nguyên.

Kernel khởi động cấu trúc TT của TT con bằng cách sao chép các trường từ cấu trúc của TT bố trong Process Table. TT con "thừa kế" từ TT bố các loại user (*real user, effective user ID, group user ID*), số ưu tiên (*nice priority*) dùng để thống kê sắp xếp lại mức ưu tiên. Kernel gán PID của TT bố vào cấu trúc của TT con, dưa TT con vào cây phả hệ của TT bố, khởi động các thông số lập biểu (như gía tri mức ưu tiên khởi đầu, thông số sử dụng CPU khởi đầu, các trường thời gian), trạng tháI của TT lúc này là " đang được tạo S8".

Kernel điều chỉnh lại các số đếm qui chiếu vào các tệp TT bố đã mở, +1, vì TT con sẽ tự động kết hợp tới. Vì TT con tồn tại trong thư mục của TT bố, nên tổng số các TT truy nhập thư mục tăng 1, tương tự như vậy số đếm qui chiếu của inode cũng tăng 1. Nếu TT bố , hay một trong các tổ tiên đã thực hiện một GHT *chroot* để đổi đường dẫn, TT con mới sẽ thừa kế và số đếm inode qui chiếu tăng 1. Kernel cuối cùng tìm đến số đếm qui chiếu các mô tả tệp của mỗi tệp TT bố đã mở trong *file table* và tăng 1. TT con không chỉ thừa kế cả các quyền trên tệp TT bố đã mở mà còn chia sẻ truy nhập tệp với TT bố bởi cả hai TT đều cùng thao tác các đầu vào trong *file table*. Hiệu quả của *fork()* là tương tự của *dup()* (nhân bản) theo quan điểm mở tệp, chỉ khác là có hai bảng con trỏ tệp (*file descriptor table*), mỗi bảng trong *u_area* của mỗi TT, đều trỏ tới cùng một tệp trong *file table*. (Xem hình dưới).

Lúc này kernel đã sẵn sàng tạo *user_level context* (*static*) cho TT con (*u_area*, các miền, *pages*) bằng việc nhân đôi từng miền của TT bố cho TT con bằng *dupreg()* và ghép vào cho TT con bằng *attachreg()*. Sau đó kernel tạo tiếp phần *dynamic* của *context*: copy *layer1 context* của TT bố, layer này chứa *register context* của user đã được bảo vệ, cũng như các lớp *kernel stack* của GHT fork()... Cơ chế thực hiện các bước lúc này tương tự như khi thực hiện chuyển bối cảnh của TT. Khi context của TT con đã chuẩn bị xong, TT bố hoàn tất *fork()* bằng việc thay đổi trạng thái của TT con thành "*ready to run (in memory)*" và trả lại PID cho user. TT con sẽ được thực hiện theo cách lập biểu thông thường bởi *scheduler*. Hai TT bố và con thực sự là hai TT chạy độc lập trong hệ, thông thường mã thực thi của TT con được người lập trình xác định khi thực hiện một kiểm tra với PID=0. Kernel kích hoạt mã này từ bộ đếm chương trình mà kernel đã lưu trong khi tạo bối cảnh cho TT con từ TT bố và để ở lớp saved register context trong layer 2 như đã đề cập. Hình dưới mô tả quá trình tạo bối cảnh cho TT con trong mô hình với kernel stack là một phần của u_area của mỗi TT. Nếu là mô hình khác thì TT bố sẽ sao chép kernel stack của nó vào vùng nhớ riêng kết hợp của TT con. Còn các mô hình khác kernel tack của TT bố và TT con là đồng nhất như nhau.

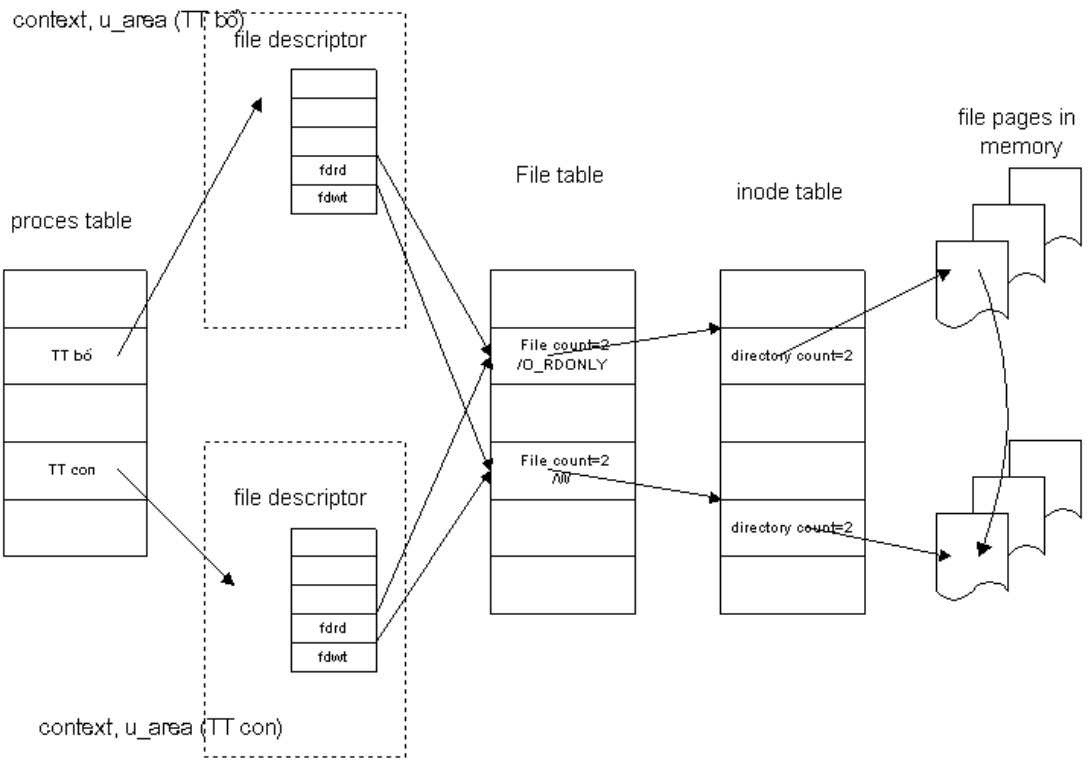


Ví dụ: TT bố và TT con cùng chia sẽ tệp (do TT bố đã mở): copy.c ($copy tep1 tep2)

```
#include <fcntl.h>
int fdrd, fdwt;
char c;
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc !=3)
                exit(1);
        if ((fdrd = open(argv[1], O_RDONLY)) ==-1)
                exit(1);
        if ((fdwt = creat(argv[2], 0666)) ==-1)
                exit(1);
        fork();
        /*cả hai TT cùng thực hiện code như nhau:*/
        rdwrt();
        close(fdrd);
```

```
        close(fdwt);
        exit(0);
}
rdwrt()
{
        for (;;)
        {
                if(read(fdrd,&c,1)!=1)          /*end of file*/
                        return;
                write(fdwt,&c,1);
        }
}
```

Chương trình trên thực hiện copy tệp khi user kích hoạt với hai đối đầu vào là tên tệp đã có và tên tệp sẽ tạo. Bên trong kernel sao chép *context* của TT bố cho TT con. Mỗi TT thực hiện trong một không gian địa chỉ khác nhau, truy nhập bản copy riêng của các biến tổng thể *fdrd* và *fdwt*, *c*, và bản copy riêng stack các biến *argc,* argv và gọi hàm rdwrt() độc lập. Bởi vì kernel đã sao chép *u_area* của TT bố cho TT con nên TT con thừa hưởng truy nhập tệp mà TT bố đã mở: ở đây các mô tả tệp *fdrd*, fdwt của cả hai TT *đều qui chiếu và cùng các đầu vào trong file table*: *fdrd* (tệp nguồn), *fdwt* (tệp đích), số đếm qui chiếu vào mỗi tệp tăng lên thành 2, cả hai TT dùng chung các gía trị của *file offset (*thay đổi mỗi lần thực hiện rdwrt()), nhưng các giá trị lại không giống nhau, vì kernel thay đổi giá trị đó sau mỗi lần gọi *read()* và *write()* của mỗi TT và mặc dù có những hai lần *copy* tệp do thực hiện chung mã lệnh (các lệnh sau fork(): hàm rdwrt()), kết quả sẽ phụ thuộc vào trình tự TT nào sẽ thực hiện cuối cùng (do *scheduler* sắp đặt): kernel không đảm bảo rằng tệp đích có nội dung giống hoàn toàn tệp gốc (thử nghỉ tới kịch bản như sau: hai TT đang thực hiện read() hai kí tự "ab" trong tệp nguồn. Giả sử TT bố readt() tự "a" (con trỏ tệp file offset tăng để trỏ vào "b" sau khi xong read()) và kernel chuyển sang thực hiện TT con, trước khi nó kịp ghi "a" vào tệp đích. TT con read() (sẽ đọc "b" theo giá trị của file offset hiện tại, và sau đó tăng 1) và giả sử nó thực hiện được write() ghi xong "b" vào tệp đích. TT bố trở lại chạy, nó ghi "a" đã đọc trước khi bị treo thực hiện (theo file ofset TT con đã tăng. Kết quả lúc này trong tệp đích sẽ là xâu "ba" chứ không phảI là "ab" như tệp gốc !.) Điều gì đã xảy ra: đó là do quá trình thực hiện hai TT không luân phiên như sụ mong muốn mà do kernel sắp đặt theo hoàn cảnh chung của hệ thống.



TT con thừa kế tệp do TT bố đã mở, hai TT chia sẻ (shared) tệp

Ví dụ: Tạo một TT con, hai TT bố và con cùng chia sẽ phần mã thục thi ngay sau khi fork thành công:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
main()
{
        printf( "TT bố:  "Bắt đầu tạo TT con". (Dòng lệnh này in ra từ mã của TT bố)\n");
        fork(); /* Tạo TT con, không xác định mã riêng*/
        printf( "PID= %d \n", getpid());       /* Bắt đầu mã chung (shared code)*/
        execl("/bin/ls/","ls", "-l",0);
        printf("Nếu in ra được dòng này, execl() không thành công !!!\n");
}
```

Phần in đậm là shared code của hai TT, và kết quả thực hiện sẽ là:

- In ra hai dòng với PID khác nhau, là PID của TT bố và PID của TT con;
- Hai kết quả danh sách thư mục do hai TT thực hiện lệnh "ls –l".

2. *Tín hiệu (signals)*

Tín hiệu (*signals*) là loại ngắt mềm, thông báo cho các TT về sự xuất hiện của các sự kiện (*events*) không đồng bộ, cũng như cho một phương thức để xử lí các sự kiện đó. Các TT có thể gởi cho mỗi TT khác các tín hiệu bằng GHT *kill()*, hoặc kernel gởi tín hiệu bên trong hệ thống. Mô hình về signal trên các hệ Unix có khác nhau và không tương thích (như giữa 4.3 BSD và SVR3), tuy nhiên với chuẩn POSIX.1 các thủ tục đã được chuẩn hoá và đảm bảo có độ tin cậy cao. Unix System V Realse 4 và 4.3+BSD có 31 loại *signals* (định nghĩa trong <signal.h>) và được phân chia như sau:

1. *signal* thực hiện kết thúc của một TT, được gởi đi khi TT gọi *exit()* hay khi TT kích hoạt GHT *signal()* với thông số "*death of child*"- TT con kết thúc;
2. *signal* thông báo các trường hợp bất thường do TT gây ra, như khi TT qui chiếu vào miền địa chỉ ảo không phải của nó, TT ghi vào miền địa chỉ "chỉ đọc", TT thực hiện các lệnh đặc quyền mà TT không được phép, hay các sự cố phần cứng;
3. *signal* thông báo các điều kiện không thể khắc phục được khi thực hiện GHT, như hết nguồn tài nguyên khi thực hiện *exec()* mà không gian địa chỉ TT gốc không đủ để nạp mã của chương trình được kích hoạt;
4. *signal* phát sinh do lỗi không dự đoán được khi thực hiện GHT, ví dụ thực hiện một GHT không có trong hệ (số hiệu GHT không tồn tại), hay ghi vào một *pipe* mà không có TT nào đọc *pipe* đó... Hệ thống sẽ bền vững hơn khi thoát khỏi các lỗi như vậy thay vì phát sinh ra các *signals*, nhưng sử dụng *signals* để thoát khỏi các xử lí hỗn tạp lại có tính thực tế hơn;
5. *signal* sinh ra từ các TT trong *user mode*, chẳng hạn khi TT muốn nhận tín hiệu *alarm* sau một chu kì thời gian, hay khi TT gởi *signals* bất kì cho mỗi TT khác bằng *kill()*;
6. *signal* liên quan tới tương tác với thiết bị đầu cuối khi người dùng treo máy ("*hung up*"), hay khi tín hiệu sóng mang đường truyền ("*carrier*") bị mất, hay khi người dùng gõ "*break*","*delete*" trên bàn phím;
7. *signal* dùng để theo dõi thực hiện TT (*tracing*).

Để gởi một signal cho một TT, kernel lập *signal bit* trong trường signal tại đầu vào của TT trong *process table,* tương ứng với kiểu signal sẽ nhận. Nếu TT đang ngủ ở mức ngắt có ưu tiên, kernel sẽ đánh thức TT. Công việc của người gởi tín hiệu (một TT hay kernel) coi như hoàn tất. Một TT có thể nhớ được các kiểu *signal* khác nhau, nhưng không thể ghi lại có bao nhiêu *signal* nó nhận được của mỗi kiểu. Ví dụ nếu TT nhận tín hiệu "hungup" và " kill", nó sẽ lập các bits tương ứng với các tín hiệu đó trong trường nói trên, nhưng không thể biết có bao nhiêu lần các tín hiệu này đã đến.

Kernel kiểm tra để tiếp nhận một *signal* khi đang chuyển trạng thái từ *kernel mode* về *user mode* (S2 → S1) và khi TT đi vào hay ra khỏi trạng thái ngủ (từ S2 vào trạng thái ngủ S4 trong bộ nhớ , hay từ S3 → trở lại S2 khi TT được chọn để thực hiện) ở một mức ưu tiên lập biểu thấp thích hợp.

Vì kernel thao tác các *signal* chỉ khi một TT từ *kernel mode* trở về *user mode*, nên *signal* không có hiệu ứng tức thì trên TT đang chạy trong *kernel mode*. Nếu TT đã đang chạy trong *user mode*, và kernel đang thao tác một ngắt sẽ sinh ra một *signal* gởi cho TT đó, kernel sẽ ghi nhận và thao tác *signal* ấy khi kết thúc xử lí ngắt. Cho nên một TT không bao giờ thực hiện trong user mode trước khi kernel xử lí các *signal* còn đang đợi.

Dưới đây là thuật toán kernel sẽ thực hiện để xác định khi một TT đã nhận một *signal*. Một TT có thể quyết định từ chối nhận *signal* hay không bằng thực hiện GHT *signal(),* do vậy trong thuật toán này kernel sẽ đơn giản bỏ qua các chỉ báo đối với các tín hiệu mà TT muốn

bỏ qua, nhưng ghi nhận sự tồn tại của *signals* mà TT để ý tới.

issg() *:Kernel thực hiện chức năng này ở các thời đIểm: S2->S4, S3->S2, S7->S1 và S2->S1,* ***kiểm tra để biết TT đã nhận một tín hiệu****: trường signal trong đầu vào của TT ở proces table khác 0.*

input: none

output: true: ìf process received signal that it does not ignore,

false: otherwise

```
{
   .while(received signal field in process table entry not 0)
   {
        .find a signal number sent to the process;
        .if (signal is "dead of child") /* là kiểu signal TT con đã két thúc, "còn
                                          zombie" nhận được*/
        {
             if (ignoring "dead of child" signal)
                  free process table entries of zombie child;
             else if (catching "dead of dead child" signals)
                  return(true);/*không bỏ qua, nhận để xử lí*/
        }
        .else if (not ignoring signal) /*nhận các loạI signals khác*/
             return(true);
        .turn off signal bit in received signal field in process table;
             /*lập lại gía trị ban đầu=0 ( reset field)*/
   }
   .return(false);/*không có signal nào đã gởi đến cho TT*/
}
```

2.1. Xử lí tín hiệu

Như cách phân loại trên có thể thấy rằng signal là lớp các sự kiện dị bộ, xuất hiện ngẫu nhiên đối với TT. TT do đó không thể kiểm tra để xem signal đã đến, mà thay vi TT "nói" với kernel theo cách "*nếu và khi signal đến, thì làm như sau ...*". Có ba việc khác nhau để TT nói cho kernel và để kernel thực hiện, đó là "chuẩn bị" và "hành động" kết hợp với signal:

- TT thực hiện mặc định, tức là phản ứng tự nhiên đối với mỗi tín hiệu, đó là để kết thúc TT.
- TT bỏ qua *signal*, là trường hợp cho hầu hết các tín hiệu, trừ SIGKILL và SIGTOP cung cấp cho superuser biết để quyết định huỹ diệt hay dừng chạy tiếp một TT. Thêm nữa nếu bỏ qua một vài signal phát sinh từ phần cứng (qui chiếu sai bộ nhớ, chia cho 0), thì xử sự của TT sẽ không dự đón được.

– Đón nhận và nói cho kernel gọi một hàm chức năng xử lí mà user định nghĩa (user function). Ví dụ khi một TT kết thúc, tín hiệu SIGCHLD sẽ phát sinh và gởi cho TT bố, như vậy TT bố sẽ dùng hàm chức năng, như waitpid() chẳng hạn, để đón nhận SIGCHLD, tìm PID của TT đó và kiểm tra mã trạng thái TT trả lại. Một ví dụ khác, Khi TT đã tạo ra một tệp tạm thời và nếu muốn xoá tệp đó đi sau khi sử dụng, cần một tạo một hàm để nhận SIGTERM khi TT gởi kill() để xoá tệp đó. Nhắc lại rằng, kernel thao tác tín hiệu trong bối cảnh của TT nhận tín hiệu, do đó TT phải được chạy hay đúng hơn là khi TT vừa ra khỏi kernel mode bắt đầu chạy trong user mode như đã nói, để xử lí tín hiệu.

Chế độ mặc định là thực hiện gọi *exit()* trong kernel mode (để S2->S9), tuy nhiên TT có thể xác định một hành động đặc biệt để chấp nhận một số *signals*. GHT *signal()* là giao diện đơn giản nhất để tiếp cận các signals, ý nghĩa sử dụng là: lập hàm xử lí nếu TT nhận signal.

Cú pháp GHT_signal() (làm gì khi nhận một signal) như sau:

```
#include <signal.h>
oldfunction = signal(signum, function)
```

Trong đó:

- *signum*: số hiệu của signal, xác định hành động sẽ thực hiện,

ví dụ:	***tín hiệu:***	***số hiệu***	***mô tả:***	***phản ứng mặc định:***
	#define SIGHUP	1	Hang up ;	kết thúc TT
	#define SIGINT	2	Ctrl_C;	kết thúc ngay tức khắc TT
	#define SIGQUIT	3	QUIT;	
	#define SIGILL	4	Illegal instuction	
	#define SIGTRAP	5	Lỗi phần cứng;	kết thúc và tạo tệp core
	#define SIGABRT	6	Crtl-\; Kết thúc ngay tức khắc TT	
	#define SIGIOT	6	asynch; I/O trap kết thúc hoặc bỏ qua	
	#define SIGBUS	7		
	#define SIGFPE	8		
	#define SIGKILL	9	kill; Diệt TT ngay tức khắc	
	#define SIGUSR1	10		
	#define SIGSEGV	11		
	#define SIGUSR2	12		
	#define SIGPIPE	13		
	#define SIGALRM	14		
	#define SIGTERM	15	Crtl_DEL; exit(), Chấm dứt tất cả các TT đang chạy	
	#define SIGSTKFLT	16		
	#define SIGCHLD	17		
	#define SIGCONT	18		
	#define SIGSTOP	19		
	#define SIGTSTP	20	Crtl_Z;	Tạm dừng TT
	#define SIGTTIN	21		

```
#define SIGTTOU         22
#define SIGURG          23
#define SIGXCPU         24
#define SIGXFSZ         25
#define SIGVTALRM       26
#define SIGPROF         27
#define SIGWINCH        28
#define SIGIO           29
#define SIGPOLL         SIGIO
#define SIGLOST         29
#define SIGPWR          30
#define SIGSYS          31
```

• *function* = địa chỉ của hàm xử lí của user mà TT sẽ kích hoạt, hay các hằng sau đây:

= 1 (hay SIG_IGN), TT sẽ bỏ qua sự hiện diện lần tới của *signal*, tuy nhiên có hai signal không được bỏ qua là SIGKILL và SIGSTOP);

= 0 (hay SIG_DFL), hành động kết hợp với signal là mặc định (xem *man* signal liệt kê cá signal và hành động mặc định). Phần lớn hành động là kết thúc TT và ghi tệp ảnh (*core*) của TT để degug.

• *oldfunction*: giá trị trả lại là con trỏ của *function* tương ứng hành động trước đó của signal, thường dùng để khôi phục lại chức năng trước đó.

• *u_area* của TT có một danh sách với các trường các con trỏ trỏ vào các xử lí tín hiệu của hệ thống. Kernel lưu địa chỉ hàm xử lí của user trong một trường tương ứng với số hiệu của *signal* đó. Quá trình xử lí một *signal* không tác động tới *signal* khác. Thuật toán thao tác *signal* như sau:

psig() *(*kernel primitive): thực hiện chức năng này ở thời điểm TT vừa ra khỏi kernel, vào user mode S2-S1, S7-S1, khi ghi nhận sự hiện diện của signal đó nếu xử lí, chuẩn bị context.

input: *none*

output: *none*

```
{
    .get signal number set in process table entry;
    .clear signal number in process entry;/*cho lần nhận tiếp sau đó*/
    .if (user had called signal system call to ignore this signal)
            return;         /*done*/
    if (user specified function to handle signal) /*lấy các đối mà signal() cung cấp, chuẩn
                                                    bị môI trường để chạy chức năng xử li
                                                    signal*/
    {
            .get user virtual address of signal catcher stored in u_area;
            .clear u_area entry after get virtual address;
```

.modify user level context: (-create user stack, -call function handling signal);

.modify system level context: write address of signal function into program counter field of user saved register context;

.return;

}

*.if (signal is type that system should dump core image of process) /*ví dụ:SíGTRAP "hardware fault"*/*

*/*function set=0, default ->exit, dump core image of the proccess, then exit. Từ tệp cỏe image, người lập trình dùng để debug nếu muốn: như TT thực hiện illegal function, hay outside virtual address space (thuộc diện các lỗi). ở đây kernel chỉ dump các signal làm hỏng chương trình mà thôI, còn các sìgnal khác như: user gõ "del:, "break" để kết thúc vĩnh cữu chương trình hay "hungup" thông báo t/b cuối không nối vào hệ thống nữa, sẽ không tác dụng dump*/*

{

.create file named " core" in current directory;

.write contents of user level context to "core" file;

}

*.invoke exit() immediately; /*default*/*

}

Khi TT nhận *signal* mà trước đó TT đã quyết định bỏ qua, TT tiếp tục chạy như khi không có *signal* đến. Vì kernel không xoá trường ghi nhận trong u_area (=1) trước đó, TT sẽ lại bỏ qua nếu *signal* lại xuất hiện. Nếu TT nhận *signal* đã quyết định nhận, TT sẽ thực hiện hàm xử lí tương ứng với signal đó ngay khi TT trở lại trong user mode, sau khi kernel thực hiện các bước sau:

- kernel truy nhập *user saved register context*, tìm lại giá trị của *progam counter* và *stack pointer* đã lưu để trở về TT của user;
- xoá trường xử lí signal trong u_area, đặt lại bằng mặc định;
- tạo khung *stack* mới trong *user stack* (trong *user - context level*). Ghi vào đó giá trị *progam counter* và *stack pointer nói trên*, xin cấp vùng bộ nhớ mới nếu cần;
- kernel thay đổi *user saved register context*: nạp *progam counter* = địa chỉ của hàm xử lí *signal*, lập giá trị tiếp theo cho *user stack pointer trong* user stack.

Sau bước chuẩn bị bối cảnh này, ***lúc TT trở về user mode, TT sẽ thực hiện hàm xử lí signal***. Khi kết thúc xử lí *signal*, kernel sẽ trở về vị trí trong mã thực thi của user nơi một GHT hay một *interrupt* đã xuất hiện.

2.1.1 Bỏ qua, không xử lí tín hiệu:

signal(SIGINT, SIG_IGN), sẽ vô hiệu hoá t/h, nếu có t/h ngắt đến (ví dụ gây ra bởi interrupt KEY, "SIGQUIT" do ấn phím quit, "HUNGUP"); Khi gõ interupt key trên console để kết thúc một chương trình, thi hiệu ứng đó sẽ lan đến cả các chương trình chạy nền của user đó. Để tránh trường hợp này, sử dụng SIG_IGN khi có SIGINT đến với TT. Ví dụ sau minh họa SIGINT chỉ có hiệu lực với TT bố, trong khi TT con vẫn tiếp tục chạy:

#include <signal.h>

```
main()
{
        .
        .
        if (fork() == 0)
                {
                signal (SIGINT, SIG_IGN);
                .
                .
                }
}
```

2.1.2 Khôi phục lại phản ứng mặc định:

*signal(**signum**, SIG_DFL),* xác dịnh xử lí đối với signum là như mặc định.

Ví dụ:

```
#include <signal.h>
#include <stdio.h>
main()
{
        FILE * fp;
        char record [ BUFSIZE], filename [100];
        signal (SIGINT, SIG_IGN); /*Sẽ bỏ qua interrupt signal trong khi ghi tệp*/
        fp = fopen (filename, " a");
        fwrite (record, BUF,1,fp);
        signal (SIGINT, SIG_DFL); /*KhôI phục lại phản ứng mặc định cho ngắt*/
}
```

2.1.3 Nhận signal để xử lí: xác định hàm xử lí cho tín hiệu, hay thay đổi từ phản ứng mặc định sang xử lí xác định:

```
#include <signal.h>
main()
{
        int catch();
        printf( " ẤN Ctrl_C để dừng chạy trình.\n");
        signal (SIGINT, catch); /*Cài để nhận signal và sẽ xử lí theo catch()*/
        while(){
                /*các lệnh của trình*/
                }
```

```
}
/*Hàm xử lí:*/
catch()
{
        printf (" Chương trình kết thúc.\n");
}
```

2.1.4 Khôi phục lại chức năng của signal trước đó:

Ví dụ: Khôi phục lại chức năng phụ thuộc vào giá trị trả lại của hàm keytest():

```
#include <signal.h>
main()
{
        int catch1(), catch2();
        int (*savesig)(); /* là con trỏ hàm*/
        if (keystest() ==1)
        signal(SIGINT, catch1); /*Khi có phím interrupt thì catch1 hay catch2*/
        else
                signal(SIGINT, catch2);
        savesig = signal (SIGINT, SIG_IGN);/*Nếu bỏ qua, thì lấy lại hàm trước đó*/
        computer();
        signal (SIGINT, savesig); /*Để khôI phục lại phản ứng với interrupt key*/
}
```

Ví dụ: Đoạn mã mô tả hệ thống sẽ vô hiệu hoá tất cả các signal SIGINT và SIGQUIT trong khi TT con chạy, cho tới khi nó kết thúc, hệ sẽ khôi phục lại các chức năng xử lí đã xác đjinh cho các signal đó. Việc khôI phục được thực hiện trong TT bố. Nếu wait() trả lại code -1 (không còn TT con nào nữa) thì TT bố sử dụng luôn làm giá trị trả lại của nó cho hệ thống.

```
#include <stdio.h>
#include <signal.h>

system(s)       /*Chạy dòng lệnh*/
char *s;
{
int status, pid,w;
register int (*istat)(), (*qstat)();
```

```
if ((pid = fork()) == 0)
        {
        execl("/bin/sh", "sh", "-c", s, NULL);
        exit(127);
        }
istat = signal (SIGINT, SIG_IGN); /*bỏ qua không xử lí nhưng láy lại con trỏ hàm mặc định*/
qstat = signal (SIGQUIT, SIG_IGN);
while ((w = wait(&status)) != pid && w != -1)
;
if( w == -1)
        status = -1;
signal (SIGINT, istat);          /*KhôI phục lại phản ứng của signal như trước*/
signal (SIGQUIT, qstat);
return( status);
}
```

2.1.5 Đón nhận vài tín hiệu

Ví dụ dùng một hàm xử lí để nhận nhiều tín hiệu, sử dụng số hiệu của tín hiệu do hệ thống chuyển đến như là thông số:

```
#include <signal.h>
main()
{
        int I;
        int catch();
        for (i=1; i <= NSIG; i++)
                signal (i, catch);
        /*
        mã lệnh chương trình
        */
}
catch(sig)
{
        signal(sig, SIG_IGN);
        if (sig != SIGINT && sig != SIGOUT && sig != SIGHUP)
                printf("Oh, oh. Tín hiệu số  %d đã nhận được. \n".sig);
        unlink (tmpfile);
        exit(1);
```

```
}
```

Trong đó hằng NSIG là tổng số các tín hiệu được định nghĩa trong *signal.h*. Lưu ý là phản ứng đầu tiên của hàm catch() để bỏ qua tín hiệu xác định đã nhận được, là cần thiết vì hệ tự động llạp lại phản ứng mặc định.

2.1.6. Dùng signal kiểm soát thực hiện chương trình

Tín hiệu (t/h) không nhất thiết chỉ dùng để kết thúc thực hiện một chương trình,một số t/h có thể định nghĩa lại để trf hoản hành động hay tác động để kết thúc một phầb nào đó của chương trình, chứ không phảI toàn bộ. Sau đây là các cách dùng t/h để kiểm soát thực hiện chương trình.

a) Trì hoãn tác động của signal

Bằng cách bắt t/h và định nghĩa lại hành động của t/h qua cờ (flag) tổng thể, sao cho t/h sẽ không làm gì cả, thay vào đó chương trình vẫn chạy và sẽ đi kiểm tra cờ xem có signal nào đã nhận hay không, trên cơ sở đó sẽ trả lời theo giá trị của cờ. Điểm cơ sở ở đây là, những hàm đã dùng để nhận t/h sẽ trở lại thực hiện chính xác ở chổ mà chương trình đã bị ngắt. Nếu hàm thoát ra bình thường thì chương trình tiếp tục chạy như chưa hề có t/h xuất hiện. Làm trể t/h có ý nghĩa đặc biệt đối với các chương trình không được dừng ở bất kì thời điểm nào. Ví dụ trong khi cập nhận danh sách liên kết, t/h không thể tác động làn quá trình này bị gián đoạn, vì như vậy sẽ dẫn đếnviệc danh sách sẽ bị huỹ hoại. Đoạn mã sau đây dùng hàm delay() để bắt t/h ngắt sẽ lập lại cờ tổng thể "sìgflag" và trở về nagy lập tức điểmtrình bị ngắt.

```
#include <signal.h>
int sìgflag;
main()
{
        int delay();
        int (*savesig)();

        signal(SIGINT, delay) /*Khong xu li t/h, chi lam tre lai*/
        updatelist(); /* Là chức năng không thể gián đoạn*/
        savesig = signal(SIGINT,SIG_IGN); /* Cấm (disable) t/h để lại trừ sigflag thay đổi*/
                                          /*trong khi kiểm tra */
        if(sigflag)
        {
                /* Đặ mã xử lí t/h ngắt nếu đã xuất hiện*/
        }
}
 delay()
        {
        signal(SIGINT, delay); /*Một lần nữa đặt lại , vì hệ đặt t/h về xử lí mặc định: kết
thúc*/
```

```
                    /*thực hiện chương trình do INT*/
    sigflag = 1; /* Đã có t/h xuất hiện, set sigflag = 1*/
    }
```

b) Dùng t/h trể với các hàm của hệ thống

Khi một chương trình dùng t/h đã bị làm trể trong khi gọi các hàm hệ thống, chương trình phảI có phần kiểm tra giá trị trả lại của hàm gọi để đảm bảo rằng lỗi trả lại không gây ra bởi t/h ngắt. Điiều này cần thiết vì khi dùng t/h trể để ngắt các hàm hệ thống (như read(0, write()), hệ thống sẽ khiến các hàm này kết thúc và trả lại lỗi. Và vấn đề là nếu chương trình diễn đạt lỗi hàm hệ thống gây ra bởi t/h trể lại như lỗi thông thường, thì sẽ dẫn đến sai lần nghiêm trọng. Ví dụ khi chương trình đọc các kí tự từ bàn phím mà nhận được t/h ngắt, thì tất cả các kí tự nhận được trước đó sẽ bị mất (do hệ tác động kết thúc đọc bàn phím, báo lỗi) và như vậy giống như chưa có kí tự nào đã gõ từ bàn phím.

Ví dụ sau chương trình sẽ kiểm tra giá trị hiện tại của cờ ngắt "ìntlag" và giá trị trả lại EOF của getchar() lđể khẳng định là kết thúc tệp thực sự.

```
if(getchar() == EOF)
        if (inflag)
                /* EOF là do INT gây ra*/
        else
                /* Đây mới là EOF thực sự*/
```

c) Dùng signal với các chương trình tương tác (interactive)

Các trình tương tác thường sử dụng nhiều lệnh (commmand), nhiều thao tác (operation) khác nhau. Ví dụ trong xử lí văn bản, để ngắt một thao tác hiện tại (như hiển thị tệp), và trở về chương trình gốc (đợi nhận lệnh), ta có thể dùng t/h. Để làm được điều này, hàm dùng định nghĩa lại hành động của t/h, phảI có khả năng trả lại việc thực hiện của chương trình ở vị trí có ý nghĩa chứ không phảI vị trí bị gián đoạn. Hai hàm thư viện C **setjmp()** và **longjmp()** hổ trợ công đoạn này.

setjmp(buffer): bảo vệ bản sao trạng tháI thực hiện của trình;

longjmp(buffer): chuyển trạng thái đang thực hiện về lại trạng thái đã được bảo vệ trước đó với các giá trị, các trạng thái của các thanh ghi như chưa có gì đã xảy ra.

buffer: biến để lưu trạng tháI thực hiện, khai báo theo kiểu **jmp_buf** trước khi dùng.

Ví dụ bảo vệ trạng tháI thực hiện của một chương trình vào biến *oldstate*:

```
jmp_buf oldstate;
setjmp(oldstate);
```

biến oldstate sau đó sẽ chứa các giá trị như của bộ đếm chương trình, dữ liệu, thanh ghi địa chỉ, trạng tháI của CPU. Các giá trị này tuyệt đối không được biến đổi.

Ví dụ:

```
#include <signal.h>
#inlude <setjmp.h>
```

```
jmp_buf sjbuf;
main()
{
        int onintr();
        setjmp(sjbuf);
        signal(SIGINT, onintr);

        /*Mã của chương trình chính, ngắt có thể xảy ra trong khi thực hiện*/

}
onintr()
{
        printf("\Interrupt đến đây!\n");
        longjmp(sjbuf);
}
```

2.1.7. Dùng các signal trong nhiều TT

Về nguyên tắc khi có t/h tại một terminal, thì t/h đó sẽ tói tất cả các chương trình đã kích hoạt tại terminal đó, kể cả là trình chạy ở chế độ nền hay các TT con. Phần này sẽ giảI thích việc sử dụng t/h trong nhiều TT.

a) Bảo vệ các TT chạy nền

Các chương trình chạy nền từ shell (với kíu tự & sau lệnh) đều được shell bảo vệ tránh bị dừng thực hiện khi có ngắt từ terminal. Có những trường hợp TT chạy nền cũng muốn bắt t/h của nó. Nếu thực hiện được, thì việc bảo vệ chống ngắt của shell cho nó thất bại, t/h mà các chương trình khác mong đợi sẽ ngắt được nó. Để ngăn chặn được đuều này, bất kì chương trình nào sẽ là TT chạy nền, phảI kiểm tra trạng tháI hiện tại của t/h, trước khi xác định lại phản ứng của t/h. Chương trình chỉ có thể định nghĩa lại phản ứng của t/h khi t/h chưa bị cấm. Ví dụ sau cho tháy, phản ứng của t/h thay đổi chỉ nếy t/h hiện tại không bị bỏ qua:

```
#include <signal.h>
main()
{
        int catch();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN) /* Nếu shell chưa thực hiện vô hiệu t/h
                                                thì*/
                signal(SIGINT, catch);
        /* Mã của chương trình*/
}
```

b) Bảo vệ các TT bố

Một chương trình (là TT bố) có thể đợi TT con kết thúc bằng wait() chỉ khi nó có thể tự bảo vệ bản thân bằng cách vô hiệu t/h ngắt, trước khi gọi GHT wait(), Bằng cách này, TT bố ngăn chặn các t/h đến với TT con có thể làm kết thúc lời gọi wait(). Làm như vậy ngăn cản được lỗi nghiêm trọng có thể xảy ra khi TT bố tiếp tục thực hiện trước khi TT con kết thúc. Ví dụ:

```
#include <signal.h>
main()
{
        int (*saveintr)();
        if (fork() == 0)
                {
                        execl(...);
                }
        saveintr = signal( SIGINT, SIG_IGN); /*TT bố vô hiệu hoá t/h ngắt ngay sau */
                                                /*khi tạo TT con*/
        wait(&status);
        signal(SIGINT, saveintr); /*khoi phục lại xử lí ngắt*/
}
```

2.1.8. Tương tranh trong xử lí ngắt

Thuật toán xử lí tín hiệu cũng còn có một số vấn đề, chẳng hạn khi TT đang xử lí một signal thì signal khác đến. Signal này có thể cúng hay khác kiểu và có thể cùng tác động đến TT. Ta nói có sự tương tranh giữa các s*ignal*.

Như đã đề cập, kernel xoá trường chứa địa chỉ của hàm xử lí ngắt trong *u_area* của TT trước khi TT trở về *user mode* để xử lí tín hiệu bằng hàm đã định. Vậy nếu TT muốn nhận và xử lí *signal* lần nữa, thì trong hàm xử lí TT phải gọi lại *signal()* trước. Nhưng nếu *signal* đến trước khi TT kích hoạt *signal()* thì sẽ xảy ra sự chạy đua (tương tranh): trong khi TT đang trong user mode, kernel có thể chuyển bối cảnh để TT có cơ hội nhận *signal* trong khi TT chưa kích hoạt được *signal()*.

Ví dụ: TT dùng GHT *signal()* để nhận tín hiệu ngắt và thực hiện chức năng *sigcatcher()*. Sau đó TT tạo một TT con, kích hoạt *nice()* để tự hạ thấp mức ưu tiên của nó xuống so với TT con và chuyển vào vòng lặp vô tận. TT con dừng thực hiện trong 5 giây để TT bố thực hiện xong *nice()* và xếp lại mức ưu tiên. Sau đó TT con đi vào chu trình, gởi tín hiệu ngắt bằng *kill()* cho TT bố trong mỗi chu kì tương tác. Nếu *kill()* trả lại lỗi (-1), có thể TT bố không còn tồn tại, TT con sẽ kết thúc bằng *exit()*. Ý tưởng ở đây là TT bố có thể đã kích koạt *signal()* mỗi lần nó nhận tín hiệu ngắt: Hàm xử lí ngắt in thông báo và gọi lặp lại *signal()* để đón nhận tín hiệu lần tiếp sau.

```
#include <signal.h>
sigcatcher()
        {
```

```
                printf("PID %d caught one \n",getpid());
                signal(SIGINT,sigcatcher);
        }
main()
        {
                int ppid;
                signal(SIGINT,sigcatcher);/* TT bố lập hàm xử lí cho tín hiệu ngắt SIGINT*/
                if (fork() == 0)    /*TT bố tạo TT con*/
                                /*code của TT con: */
                {       /*để một khoảng thời gian đủ cho 2 TT xác lập*/
                sleep(5);
                ppid = getpid();   /* lấy pid của TT bố, TT con dùng kill() gởi SIGINIT*/
                                /* cho TT bố nếu bố còn sống:*/
                for (;;)
                        if (kill(ppid,SIGINIT) == -1)/* Nếu TT bố không còn tồn tại*/
                                exit();
                }
        /*TT bố hạ mức ưu tiên của mình: */
        nice(10);
        for(;;); /* TT bố chạy lòng vòng để tồn tại*/
}
```

Trình tự các sự kiện có thể xuất hiện như sau:

1. TT con gởi t/hiệu ngắt cho TT bố;
2. TT bố bắt được t/hiệu, thực hiện *sigcatcher()*, nhưng kernel chen ngang và chuyển bối cảnh trước khi TT bố có thể cho chạy *sigcatcher()* lần tiếp theo;
3. TT con lại được chạy và gởi t/hiệu ngắt cho TT bố;
4. TT bố nhận t/hiệu ngắt lần thứ 2, nhưng không xử lí được (do kết quả ở 2.), Khi tiếp tục chạy, TT kết thúc (*exit*).

Các cách giải quyết tương tranh

Để giảI quyết sự tranh chấp này, Unix hệ cung cấp các hàm sau:

```
        #include <signal.h>
        sigmask();
        sigsetmask();
Linux:
        #include <signal.h>
        sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Cấu trúc lưu các t/h dùng thêm hay loại bỏ khỏi mask, hàm dùng để xử lí biến cấu

trúc:

Khai báo: **sigset_t mask_set;**
Khởi tạo biến: **sigemptyset (&mask_set);**
Thêm một signal vào cấu trúc: **sigaddset ((&mask_set, SIGTSTP);**
sigaddset ((&mask_set, SIGINT);
Loại bỏ một signal khỏi cấu trúc: **sigdelset ((&mask_set, SIGTSTP);**
Xác định signal có hay không trong cấu trúc:
sigismenber ((&mask_set, SIGTSTP);
Đặt tất cả các signal của hệ vào cấu trúc:
sigfillset ((&mask_set);

Ví dụ:

Đến số lần Crtl-C, đến lần thứ 5, hỏi người dùng về ý định thực tế, và nếu có Crtl_Z sẽ in ra số lần ấn Crtl_C:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int ctrl_c_count = 0;
#defineCTRL_C_MAX          5

void catch_int(int sig_num) /*hàm xử lí phản ứng của INT khi có Crtl_C*/
{
   sigset_t mask_set;              /* Khai báo*/
   sigset_t old_set;

   signal(SIGINT, catch_int); /* Bắt Crtl_C*/
   sigfillset(&mask_set);                  /* Khóa các t/h khác để xử lí INT*/
   sigprocmask(SIG_SETMASK, &mask_set, &old_set);

   ctrl_c_count++;
  if (ctrl_c_count >= CTRL_C_MAX) {
     char answer[30];

        printf("\nRealy Exit? [y/N]: ");
        fflush(stdout);
        gets(answer);
        if (answer[0] == 'y' || answer[0] == 'Y') {
           printf("\nExiting...\n");
           fflush(stdout);
           exit(0);
        }
        else {
           printf("\nContinuing\n");
           fflush(stdout);
           ctrl_c_count = 0;
        }
```

```
   }
   sigprocmask(SIG_SETMASK, &old_set, NULL);/*KhôI phục lại như cũ*/
}

void catch_suspend(int sig_num)       /* Chặn bắt Crtl_Z*/
{
   sigset_t mask_set;
   sigset_t old_set;

   signal(SIGTSTP, catch_suspend);  /* Các bước như trên*/
    sigfillset(&mask_set);
   sigprocmask(SIG_SETMASK, &mask_set, &old_set);

   printf("\n\nSo far, '%d'
         Ctrl-C presses were counted\n\n", ctrl_c_count);

   fflush(stdout);

   sigprocmask(SIG_SETMASK, &old_set, NULL);
}

/* Dịch và chạy trình này*/
int main()
{
   signal(SIGINT, catch_int);
   signal(SIGTSTP, catch_suspend);

    printf("Please random press Ctrl-C or Ctrl-Z");
    while (1) {
                sleep(1);
            }

   return 0;
}
```

Tổng hợp xử lí tương tranh bằng hàm ***sigaction():***

sigaction(int sig, const struct sigaction *act, const struct sigaction *oldact);

trong đó cấu trúc

sigaction cho thông tin về hàm xử lí:

```
struct sigaction {
                    void (*) int sa_handler; /*hàm xử lí t/h*/
                    sigset_t sa_mask;       /* Tập t/h cần khoá*/
                    int sa_flags;           /*trạng tháI khôI phục sau khi xử lí t/h*/
                    .
                    .
                    .
                }
```

Lưu ý chung khi xử lí signal:

1. Hàm tạo ra phải ngắn gọn, chạy nhanh, để trách tương tranh signal;
2. Ngăn chặn tương tranh ngay nếu có thể;
3. Không bỏ qua các signal hệ thống báo lỗi nghiêm trọng(SIGSEGV, SIGBUS) theo tư duy phù hợp với lập luận khi là trình, nếu cần dọn dẹp tàI nguyên và kết thúc (abort()) thực hiện đẻ tránh gây đổ vỡ thêm ở các phần khác.
4. Cẩn thận khi cài timer, mỗi TT chỉ có một timer, tắt bật hợp lí.
5. Xử lí signal là dị bộ, đơn giản, không có sự chờ đợi, chỉ có xử lí hay bỏ qua

2.2. Các GHT bị ngắt bởi signal

Trong hệ thống có một số các GHT thuộc loại "chậm" và trong khi thực hiện, các GHT này có thể bị ngắt bởi signal và GHT sẽ trả lại thông báo lỗi thuộc loại EINTR (error interupt). Hệ chia các GHT thành hai loại: "chậm" và các GHT còn lại. Các GHT chậm có thể bị gián đoạn thực hiện vĩnh viễn, là các GHT:

- đọc tệp, nếu chưa có dữ liệu (pipe, terminal, network);
- ghi vào cùng một tệp (hay nhiều tệp như nhau) nếu dữ liệu không được chấp nhận ngay;
- mở tệp bị kẹt cho tới khi có điều kiện vào đó xuất hiện;
- pause() và wait(), đợi cho tới khi signal xuất hiện;
- một số thao tác I/O (ioctl());
- Một số IPC;

Để xử lí cần viết mã như sau:

```
again:
        if (( n = read(fd, buf, BUFFSIZE)) < 0) /*GHT là open()*/
            {
                if (errno = EINTR)
                    goto  again;
            }
```

Trong Unix 4.2BSD tạo ra cái gọi là tự động khởi động lại GHT bị ngắt, gồm có: ioctl(),read(),write(), wait(), waitpid(), và writev(). SVR4 không có đặc tính này, nhưng hàm sigaction() với tuỳ chọn SA_RESTART dùng để khởi động lại GHT bị ngắt.

2.3. Nhóm các TT

Hệ điều hành Unix duy trì các nhóm tiến trình. Một nhóm tiến trình là một tập hợp gồm nhiều tiến trình. Tất cả các tiến trình thuộc một nhóm và các tiến trình con bất kỳ đều thuộc cùng nhóm theo mặc định. Một tiến trình có thể quyết định tạo ra một nhóm mới và trở thành trưởng nhóm đó. Một nhóm được nhận biết bởi số nhóm của nó, số này bằng với số của trưởng nhóm.

Khái niệm này về các nhóm cho phép gửi các thông điệp đến tất cả các tiến trình là thành viên của cùng nhóm để thực hiện kiểm soát công việc. Các trình thông dịch lệnh (bash,

esh, ksh, v.v...) sử dụng các nhóm này để cho phép người dùng đình chỉ thực hiện một tiến trình và tiếp tục nó ở phía trước hoặc phía sau. Các nhóm tiến trình cũng được sử dụng ở hệ quản lý các thiết bị đầu cuối.

Các kỳ hoạt động

Một kỳ hoạt động là một tập hợp gồm một hoặc nhiều nhóm tiến trình. Mỗi kỳ hoạt động được gán cho một thiết bị đầu cuối điều khiển. Thiết bị đầu cuối điều khiển này là một thiết bị ngoại vi hoặc một giả thiết bị (khi có một liên kết mạng đến một thiết bị đầu cuối ở xa). Khi tiến trình tạo ra một kỳ hoạt động mới:

- nó trở thành trưởng quá trình.

- một nhóm mới các tiến trình được tạo ra và tiến trình gọi trở thành trưởng của nó.

- tiến trình gọi không có thiết bị đầu cuối kiểm tra.

Một kỳ hoạt động được nhận biết theo số của nó, số này bằng với số của trưởng tiến trình. Thông thường, một kỳ hoạt động mới được tạo ra bởi chương trình login khi người sử dụng được nối kết. Tất cả các tiến trình sau đó sẽ trở thành các thành viên của cùng kỳ hoạt động. ở trường hợp các lệnh được liên kết (chẳng hạn bằng các ống dẫn), trình thông dịch lệnh (shell) tạo ra các nhóm tiến trình mới để thực hiện kiểm soát công việc.

Dù một TT được nhận biết bằng số ID, nhưng đôi khi kernel lại nhận biết các TT bằng "nhóm". Ví dụ tất cả các TT phát sinh sau *login shell* đều có liên quan tới *login shell* đó, cho nên khi user gõ "*break*" hay "*delete*" tất cả các TT nói trên đều nhận được tín hiệu. Kernel dùng ***procces group ID*** (GPID) để nhận biết nhóm các TT (có cùng thuỷ tổ) có thể cùng nhận chung một tín hiệu của một vài sự kiện nhất định. Số GPID lư trong cấu trúc của TT trong process table, các TT trong cùng nhóm đều có cùng số định danh nhóm.

GHT *setpgrp()* sẽ khởi động số nhóm TT của một TT và đặt số đó bằng số PID của nó.

grp = setpgrp();

Trong đó: *grp* là số nhóm TT mới được trả lại. TT con giữ nguyên số nhóm của TT bố trong quá trình thực hiện fork().

2.4. TT gởi tín hiệu

Hàm **kill()** gởi một signal cho một TT hay nhóm các TT. Và **raise()** cho phép một TT gởi signal cho chình mình. Cú pháp như sau:

1. Gởi từ dòng lệnh tại console:

#kill – [SIGNAME] [PID]

2. Khi lập trình:

#include <sys/types.h>

#include <signal.h>

kill(pid, signum);

raise(signum);

Trong đó: *Pid* là cho biết tập hợp các TT sẽ nhận *signal* và được xác định như sau:

- nếu *pid* là số nguyên dương, kernel gởi *signal* cho TT có số pid đó;
- nếu *pid* = 0 kernel gởi *signal* cho tất cả các TT có số nhóm TT (GPID) bằng số nhóm (GPID) của TT gởi *signal*;

  KháI niệm "tất cả" loại trừ các TT hệ thống (swapper (PID=0), init (PID=1), pagedaemon (PID=2)).
- nếu *pid* = -1 kernel gởi cho tất cả các TT mà UID thực (*real user ID*) bằng số ID hiệu dụng (*effective user ID*) của TT gởi signal. Nếu TT gởi có số ID hiệu dụng là của *superuser*, thì *signal* sẽ được gởi cho tất cả TT, trừ TT 0 (*swaper*) và TT 1 (*Init*).
- nếu *pid* < 0, khác –1, kernel gởi *signal* cho tất cả các TT có số nhóm PID bằng giá trị tuyệt đối của |*pid*|, với điều kiện TT gởi có quyền gởi signal đến.
- trong các trường hợp nếu: Eff. UID của TT gởi khác Eff.UID của superuser, hoặc Real UID hoặc Eff. UID của TT gởi khác với Eff. UID của TT nhận, *kill()* sẽ không thực hiện được.

Ví dụ:

TT bố lập số của nhóm *setpgrp()* TT (bằng chính *pid* của nó) và tạo ra 10 TT con khác. Sau khi tạo, mỗi TT con có cùng số nhóm như TT bố, tuy nhiên với phép & (*bitwise*), các TT tạo ra khi *i* lẻ sẽ có số nhóm khác (và bằng chính số *pid* của chúng) khi thực hiện *setpgrp()* tiếp theo sau. Hàm *getpid()* và *getpgrp()* cho số ID và số của nhóm của một TT đang chạy, sau đó TT dừng (*pause()*) thực hiện cho tới khi TT nhận được một t/hiệu mà TT gởi. TT bố gởi t/hiệu cho các TT đã không thay đổi nhóm (trong nhóm của TT bố): tức 5 TT "chẵn" đã không đổi số nhóm. 5 TT "lẻ" không nhận được tín hiệu, tiếp tục chu trình.

```
#includ <signal>
main()
{
        register int i;
        setpgrp(); /*lạp gpid nhóm = 0, TT bố*/
        for(i=0; i<10; i++)
        {
                if (fork() ==0)/* sau fork() TT con có grp ID của bố*/
                {
                        if (i & 1)/* cho 1,3,5,7,9*/
                        setpgrp();/*để gpr ID khác grpID của bố và= PID TT con đó*/
                        printf("pid=%dpgrp=%d\n",getpid(),getpgrp());
                        pause(); /*dừng thực hiện TT, đợi signal*/
                }
        }
        kill(0,SIGINIT);/*pid=0:TT bố gởi t/h ngắt cho các TT đã không đổi gpid, */ /*cùng
                        nhóm của nó*/
}
```

Ví dụ 2: Dùng lệnh ***kill*** trên terminal: user tạo ra hai signal (SIGUSR1 và SIGUSR2) và hàm xử lí sig_usr() sẽ đón nhận và in ra số của signal nhận được:

```
#include    <signal.h>
#include    "ourhdr.h"          /*tệp header do user tạo*/

static void     sig_usr(int);   /* hàm xử lí dùng cho cả hai signal */

int
main(void)
{
        if (signal(SIGUSR1, sig_usr) == SIG_ERR)
                err_sys("can't catch SIGUSR1");
        if (signal(SIGUSR2, sig_usr) == SIG_ERR)
                err_sys("can't catch SIGUSR2");

        for ( ; ; )
                pause();
}

static void
sig_usr(int signo)/* đối vào là signal number */
{
        if (signo == SIGUSR1)
                printf("received SIGUSR1\n");
        else if (signo == SIGUSR2)
                printf("received SIGUSR2\n");
        else
                err_dump("received signal %d\n", signo);
        return;
}
```

Giả sử sau khi dịch, chương trình có tên sigusr, và cho chạy ở chế độ nền (background với *$ sigusr &* cuối lệnh), và trên console dùng lệnh *$ kill(1)* và kill(2) để gởi signal cho TT hay nhóm TT. Kết quả như sau:

```
$ sigusr &              /* cho chương trình chạy ở nền*/
$jobs                   /* xem TT chạy ngầm với số hiệu sẽ dùng trong kill*/
[1]     4270            /*PID=4270*/
$ kill –USR1    4270    /* gởi cho TT 4270 signal SIGUSR1*/
reveived SIGUSR1
$ kill –USR2    4270    /* gởi cho TT 4270 signal SIGUSR2*/
reveived SIGUSR2
$ kill 4270             /*gởi cho TT signal kết thúc SIGTERM*/
[1] + Terminated        sigusr &
```

2.6. alarm() và pause()

#include <inistd.h>

alarm(seconds): Đặt cho định thời (timer) một khoản thời gian, khi hết thời gian đó, kernel sẽ phát sinh ra signal SIGALRM. (cộng với độ trể để TT chạy hàm xử lí signal); Chỉ có một đồng hồ báo động (alarm clock) cho một TT. Nêu gọi lần tiếp theo, mà lần trước thời gian chưa trôI qua, giá trị trả lại sẽ là định lượng còn lại, giá trị mới sẽ được thay vào. Nếu giá trị mới đặt =0, alarm() sẽ huỹ, giá trị trả lại sẽ là số thời gian còn lại bỏ dỡ. Hành động mặc định cho signal này là kết thúc TT. Nhiều TT dùng đồng hồ báo động này để bắt tín hiệu SIGALRM: ví dụ dọn dẹp trước khi TT kết thúc sau một khoản thời gian.

Trả lại: 0, hay giá trị như nói trên.

pause(void): Treo (tạm dừng thực hiện) TT gọi hàm cho tới khi bắt được signal. Hàm trả lại thời gian nếu hàm xử lí signal dược thực hiện và hàm này trở về. Các trường hợp khác sẽ là -1 với thông báo errno=EINTR

Ví dụ: Dùng alarm() và pause() để tự đi ngủ trong một thời gian. Dịch ra với tên sleep1, đối vào là thời lượng (số thời gian ngủ):

$sleep1(10) /* ngủ 10 s*/

```
#include    <signal.h>
#include    <unistd.h>

static void
sig_alrm(int signo)
{
    return; /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs);       /* start the timer */
    pause();                /* next caught signal wakes us up */
    return( alarm(0) );     /* turn off timer, return unslept time */
}
```

Ví dụ còn một số vấn đề tương tranh khi lần đầu gọi alarm() và pause(). Trên hệ quá bận, phảI dợi để alarm kết thúc và hàm xử lí signal có thể sẽ chạy trước khi gọi pause(). Nêu xảy ra trình gại pause() có thể bị treo.

Ví dụ: (read1.c)

```
#include    <signal.h>
#include    "ourhdr.h"
```

```
static void     sig_alrm(int);

int
main(void)
{
        int             n;
        char    line[MAXLINE];

        if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                err_sys("signal(SIGALRM) error");
        alarm(10);
        if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
                err_sys("read error");
        alarm(0);

        write(STDOUT_FILENO, line, n);

        exit(0);
}

static void
sig_alrm(int signo)
{
        return; /* nothing to do, just return to interrupt the read */
}
```

Vấn đề gần giống như ví dụ trước: chạy đua giữa lần gọi alarm() đầu tiên và gọi read(). Nếu kernel ngăn cản một TT giữa lần gọi hai hàm ở đây lâu hơn là chu kì của alarm, hàm read() sẽ bị treo. Vậy phảI đặt chu kì alarm đủ dàI để tránh bị kẹt read.
Để stránh các hiện tượng trên , chương trình trên viết lại với sử dụng longjmp() dùng để đặt các giới hạn thời gian trên các xử lí I/O. hàm được dùng từ bên trong các chức năng xử lí signal để trở về vòng main() của chương trình thay vì trở về tư chức năng xử lí. Lưu ý là theo ANSI C các chức năng xử lí signal có thể trở về trình chính nhưng cũng có thể abort() (từ bỏ hàm, kết thúc bất thường) hay exit() và longjmp(), và bằng cách này ta không còn quan tâm trên các hệ chạy chậm các GHT có bị ngắt hay không.

```
#include        <setjmp.h>
#include        <signal.h>
#include        "ourhdr.h"

static void             sig_alrm(int);
static jmp_buf env_alrm;

int
main(void)
{
        int             n;
        char    line[MAXLINE];
```

```
        if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                err_sys("signal(SIGALRM) error");

        if (setjmp(env_alrm) != 0)
                err_quit("read timeout");

        alarm(10);
        if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
                err_sys("read error");
        alarm(0);

        write(STDOUT_FILENO, line, n);

        exit(0);
}

static void
sig_alrm(int signo)
{
        longjmp(env_alrm, 1);
}
```

3. Kết thúc TT

TT trên Unix có thể kết thúc bình thường theo ba cách và không bình thường theo hai cách và mô tả như sau:

1. Kết thúc bình thường khi gọi
   - (a) Kết thúc một main(), tương đương với gọi ẩn exit().
   - (b) Gọi exit(). exit() do ANSI C định nghĩâ bao hàm gọi các xử lí của exit bằng kích hoạt atexit() và đóng tất cả các xâu I/O (fclose() được gọi để đóng tất cả các xâu đã mở, ghi buffer lên tệp). Do không xử lí các mô tả tệp (fd), nên các TT (bố và các con), các điều khiển công việc sẽ không hoàn tất (như đã đề cập khi TT là zombie).
   - (c) Gọi _exit(),kết thúc và về kernel ngay. Hàm này sẽ do exit() gọi và có các xử lí đặc biệt của bản thân HĐH Unix thực hiện. _exit() do POSIX.1 định nghĩa. exit(3) là của thư viện C chuẩn Unix, còn _exit(2) lại là một GHT.
2. Kết thúc không bình thường.
   - (a) Gọi abort(), bỏ việc, tạo ra SIGABRT signal.
   - (b) Khi TT nhận một số signal nhất định từ bản thân nó (bằng abort()), hay từ các TT khác hay từ kernel (TT chia cho số 0, qui chiếu địa chỉ ngoàI không gian của TT).

Một TT kết thúc sẽ đi vào trạng tháI *zombie* (S9) sẽ *bỏ lại* các nguồn tài nguyên TT đã dùng, hủy bối cảnh của nó, chỉ còn lưu lại *entry* trong *procces table*.

exit(status)

Status là giá trị trả lại cho TT bố để TT thực hiện kiểm tra. TT có thể thực hiện GHT *exit()* một cách tường minh hay ẩn. Khi một chương trình thoát ra khỏi *main()* thì đều tự động có *exit()*. Kernel tự phát sinh *exit()* cho TT khi TT nhận được một t/hiệu không có lí do "rõ ràng" như đã nói và trong trường hợp này, *status* sẽ là số của *signal* đó.

Hệ thống tuy không đặt giới hạn thời gian cho việc thực hiện TT, nhưng TT sẽ kết thúc nếu thời gian đó quá lâu. TT 0 (*swapper*) và TT 1 (*Init*) là hai TT vĩnh cữu của hệ.

exit()

```
    input:   return code for parent process
    output:  none
{
      .ignore all signals;
      .if(process group leader with associated control terminal)
      {
            .send hungup signal to all members of process group;
            .reset process group for all member to 0;
      }
      .close all open files;
      .release current directory (iput());
      .release current (changed) root, if exists (iput());
      .free regions, memory associated with process (freereg());
      .write accounting record;
      .make process state zombie;
      .assign process ID of all child process to be Init process;
      .if any children were zombie. Send dead of child signal to Init process;
      .send dead of child signal to parent process;
      .context switch;
}
```

TT bỏ qua các *signal* vì không có lí do gì để xử lí khi TT kết thúc cuộc đời. Nếu TT đang kết thúc là TT đứng đầu các TT (là TT *shell login*, kết hợp với *terminal*), thì kernel sẽ gởi *SIGHUP* cho tất cả TT trong nhóm (ví dụ nếu user gõ CTRL-DEL (*tức End of File*), tại *login shell* trong khi các TT kết hợp đang chạy, thì các TT sẽ nhận được *hungup signal*); đồng thời số định danh nhóm sẽ đổi thành 0 cho tất cả các TT thành viên. Kernel sau đó thực hiện đóng các tệp TT đã mở, trả lại các inodes cho thư mục (hiện tại, cũng như thư mục đã có chuyển đổi bằng lệnh *cd*). Tiếp theo kernel giải phóng các miền bộ nhớ TT đã dùng và đưa TT vào trạng thái còn xác (*zombie*). Các số liệu kết toán thời gian sử dụng của TT cũng như của các TT hậu bối của nó trong *user mode* và *kernel mode* được ghi lại trong *proccess table*. Đồng thời các số liệu đó cũng ghi vào tệp kết toán của hệ thống (bao gồm: user ID, run time CPU, sử dụng memory, các I/O... có ích cho việc điều chỉnh hoạt động của hệ). Cuối cùng kernel cắt TT ra khỏi cấu trúc cây của TT, đưa TT *init(1)* tiếp nhận các TT hậu bối của TT đã kết thúc. Như vậy các TT hậu bối do TT tạo ra đang còn sống, sẽ có bố mới là TT *init*. Cố gắng

cuối cùng của TT đang kết thúc là thông báo cho TT bố của nó về sự ra đi của mình, và nếu có còn TT con nào của nó là *zombie*, TT sẽ thông báo cho *init* để thu dọn khỏi *proccess table*, bản thân TT trở thành *zombie*, chuyển bối cảnh để kernel tuyển TT khác vào thực hiện. Trong thực tế lập trình ta dùng *wait()* để đồng bộ TT bố với TT kết thúc của TT con.

Ví dụ: TT bố tạo TT con, sau đó TT con tạm treo thực hiện, còn TT bố kết thúc. TT con còn sống và đợi *signal* ngay cả khi TT bố đã "khuất":

```
main()
{
        int child;
        if ((child ==fork()) == 0)
        {
                printf(" child PID= %d\n",getpid);
                pause() /* TT con treo, chờ signal*/
        }
        /*parent*/
        printf(" child PID= %d\n", child);
        exit(child);
}
```

Ví dụ: Các gí trị trả lại của exit()

```
#include        <sys/types.h>
#include        <sys/wait.h>
#include        "ourhdr.h"

int
main(void)
{
        pid_t   pid;
        int             status;

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)                      /* child */
                exit(7);

        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)                      /* child */
                abort();                        /* generates SIGABRT */
```

```
        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)                      /* child */
                status /= 0;            /* divide by 0 generates SIGFPE */

        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */

        exit(0);
}

/*pr_exit(status)*/
void pr_exit(int status)
{
        if (WIFEXITED (status))
                printf("nornal termination, exit status = %d\n", WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
                printf("abnornal termination, exit status = %d\n", WTERMSIG(status));
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generarted)" : "");
#else
                "");
#endif
        else if (WIFSTOPPED          (status))
                printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
```

WIFEXITED(status), WIFSIGNALED(status), WIFSTOPPED(status) là các macro định nghĩa trong ***<sys/wait>*** *theo POSIX.1*

4. wait(), pidwait(), đợi một TT kết thúc

Khi một TT kết thúc (bình thường hay không bình thường) thì TT bố cũng sẽ nhận được thông báo từ kernel bằng signal SIGCHLD. Do kết thúc TT là một sự kiện dị bộ (xảy ra bất kì lúc nào khi TT đang chạy), nên tín hiệu này mang tính thông báo cho TT bố. TT bố sẽ

bắt tín hiệu để xử lí hay cũng có thể bỏ qua. Mặc định là bỏ qua, còn nếu quan tâm đến TT dùng wait() hay pidwait() để đợi TT con kết thúc và sau dó chạy tiếp. TT khi dùng wait(), pidwait() có thể:

(a) ngăn cản tất cả các TT con nếu hãy còn đang chạy, hoặc

(b) trở về ngay lập tức với thông báo trạng thái kết thúc của TT con (nếu TT con đã kết thúc và đâng đợi nhận trạng tháI kết thúc), hoặc

(c) Trở về ngay với thông báo lỗi.

Nếu TT đang gọi wai() do nhận được SIGCHLD, ta hoàI vọng wait() để thát ra ngay lập tức. Còn nếu gọi wait() tại bất kì thời điểm nào, thì wait() có thể sẽ bị kẹt.

#include<sys/types>

#include <sys/wait.h>

pid = wait(stat_addr);

waitpid(pid, stat_addr, options)

Trong đó: *pid* là PID của TT con đã kết thúc (*zombie*), nếu OK, -1 có lỗi;

atat_addr là địa chỉ chứa mã trả lại của *exit()* củaTT con.

Giá trị của ***pid*** trong waitpid() như sau:

pid = -1: đợi bắt kì TT con nào. waitpid() tương đương vớii wai().

pid > 0 : đợi TT có số PID bằng pid;

pid == 0: đợi bất kì TT nào có số nhóm TT (GPID) bằng GPID của TT gọi.

pid < -1: đợi bất kì TT nào có GPID bằng |pid|

wait() có thể dừng thực hiện trình gọi (TT bố) cho tới khi TT con kết thức (dùng để đồng bộ thực hiện TT bố và TT con), còn với options trong waitpid(), có thể huỹ việc TT bố đợi TT con đầu tiên kết thúc hay xác điịnh TT con nào nó sẽ đợi kết thúc.

waitpid() cho ba đặc tính khác với wait() như sau:

- cho phép đợi nhiều hơn là một TT (wait(0 trả lại trạng tháI của TT kết thúc);

- cung cấp khả năng không ngăn cản TT gọi tạm dùng để đợi TT con, dùng khi muốn biết trạng tháI của TT con nhưng không dùng thực hiện TT bố.

- hổ trợ kiểm soát công việc (job).

Thuật toán cơ bản của wait():

```
Input:   none
Output:  child ID, child exit code
{
    .if (waiting process has no child process)
        return(error);
    .for(;;)
    {
        .if (waiting process has zombie child)
        {
```

```
                    .pick arbitrary zombie child;
                    .add child CPU usage to parent;
                    .free child process table entry;
                    .return (child ID, child exit code);
                }
        .if (waiting process has child process but none zombie)
        {
                sleep at interruptible priority(event:child process exit);
        }
        }
}
```

Thuật toán cho thấy hai trường hợp cơ bản:

- TT bố có TT con là *zombie*, nó sẽ làm công việc "thu dọn" và thoát ra với PID và mã trả về khi TT con kết thúc sự tồn tại.
- TT bố có TT con nhưng đang hoạt động, chưa kết thúc cuộc đời, do vậy TT bố tự đi ngủ, và thức dậy bằng t/hiệu ngắt do TT kết thúc bằng *exit*. Khi nhận được t/h "*death of child*" TT sẽ có các phản ứng khác nhau:
  - Trong trường hợp mặc định, TT sẽ thức từ trong *wait*. Thuật toán *sleep()* kích hoạt *issig()* để kiểm tra nhận *signal*, ghi nhận trường hợp đặc biệt của "*death of child*" và trả lại "*false*". Kết quả là kernel trở về *wait*, quay lai chu trình tìm thấy *zombie*... và thoát khỏi *wait*.
  - Nếu là "*death of child*" kernel sắp xếp để thực hiện hàm xử lí ngắt của user.
  - Nếu Tt bỏ qua "*death of child*" kernel tái thực hiện *wait*, giải phóng tài nguyên của *zombie* và tìm các TT con khác.

Ví dụ:

Chương trình cho các kết quả khác nhau lúc chạy khi có hay không có đối đầu vào:

1. Không có đối đầu vào (*argc* = 1, chỉ có tên của chương trình): TT bố tạo ra 15 TT con, mỗi TT kết thúc bằng exit() với ***i*** trả về. Kernel thực hiện *wait()*, tìm *zombie* của TT con kết thúc, Kernel không chắc chắn TT con nào nó sẽ tìm thấy, biến *ret_val* sẽ cho PID của TT con tìm được.
2. Khi có đối đầu vào (*argc* > 1, có tên chương trình và một xâu khác), TT bố dùng *signal()* để bỏ qua "*death of child*". Giả sử TT bố đi ngủ trong *wait()* trước khi có một TT con *exit()*: TT con gởi "*death of child*" cho TT bố, TT bố thức dậy, nhận thấy t/hiệu gởi đến là "*death of child*", TT bố loại *zombie* của TT con ra khỏi *proccess table* và tiếp tục thực hiện *wait()* như không có t/hiệu nào xảy ra. Công việc đó lặp lại cho tới TT bố đi qua hết *wait()* và khi TT bố không còn TT con nào, và *wait()* trả lại –1.

Sự khác nhau cho hai trường hợp là: trường hợp 1, TT bố đợi kết thúc của *bất kì* TT con nào; còn trường hợp 2 TT bố đợi kết thúc của *tất cả* các TT con.

```
#include <signal.h>
main(argc, argv)
{
```

```
        int i, ret_val, ret_code;
        if (argc >= 1)
        signal(SIGCLD, SIG_IGN); /* bỏ qua " death of child"*/
        for(i = 0; i < 15; i++)
        if (fork() == 0)
        {       /* TT con ở đây*/
                printf(" child proc %x\n, " getpid());
                exit(i);
        }
        ret_val = wait(&ret_code);
        printf(" wait ret_val %x, ret_code %x\n", ret_val, ret_code);
}
```

Nếu áp dụng thuật toán này để "dọn dẹp" các *zombie* cho *procces table* thì bắt buộc TT bố phải thực hiện *wait(),* nếu không các đầu vào của *procces table* có thể sẽ hết.

Ví dụ với waitpid(): (*fork2.c*)

```
#include        <sys/types.h>
#include        <sys/wait.h>
#include        "ourhdr.h"

int
main(void)
{
        pid_t   pid;

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {                /* first child */
                if ( (pid = fork()) < 0)
                        err_sys("fork error");
                else if (pid > 0)
                        exit(0);/* parent from second fork == first child */

                /* We're the second child; our parent becomes init as soon
                   as our real parent calls exit() in the statement above.
                   Here's where we'd continue executing, knowing that when
                   we're done, init will reap our status. */

                sleep(2);
                printf("second child, parent pid = %d\n", getppid());
                exit(0);
        }
```

```
        if (waitpid(pid, NULL, 0) != pid)       /* wait for first child */
                err_sys("waitpid error");

                        /* We're the parent (the original process); we continue
                        executing,knowing that we're not the parent of the second child. */

        exit(0);
}
```

5. Sự tương tranh

Sự chạy đua (race) xuất hiện khi nhiều TT cố thực hiện một việc gì đó trên dữ liệu chia sẻ mà kết quả cuối cùng phụ thuộc vào thứ tự trong đó các TT chạy. Tạo TT bằng fork() có vẽ như là sự gieo rắc tương tranh nếu có một logic nào đó ẩn hay tường minh, phụ thuộc vào TT bố hay TT con sẽ chạy trước sau khi fork(). Thực tế không thể dự đoán trước TT nào sẽ chạy trước và cho dù là biết, thì điều gì sẽ xảy ra khi TT bắt đầu chạy lại phụ thuộc vào tảI hệ thống cũng như vào thuật toán lập biểu của kernel.

Trong ví dụ trên (fork2.c) tương tranh có xu thế xảy ra khi TT con thứ hai đã in xong số PID của TT bố. Nếu TT con thứ hai chạy trước TT con thứ nhất thì TT bố sau đó sẽ là TT con thứ nhất. Nừu TT con thứ nhất chạy trước và có đủ thời gian để exit() thì TT bố của TT con thứ hai sẽ là TT init của hệ thống. Cho dù dùng sleep() cũng chưa có gì đảm bảo. Nừu hệ quá bận TT con thứ hai sẽ tiếp tục thực hiện sau khii sleep(0 kết thúc, trước khi TT con thứ nhất có cơ hội để chạy.

Một TT muốn đợi TT con kết thúc, sẽ phảI dùng wait(). Nừu TT con muốn đợi TT bố kết thúc, (như trong ví dụ fork2.c), vòng đợi sau đây có thể sử dụng đến:

```
    while (getppid() != 1)
        sleep(1);
```

Vấn đề kỉ thuật ở đây là CPU bị dùng nhiều cho quay vòng (polling) vì sau mỗi giây, trình gọi sẽ dược đánh thúc 1 lần để thực hiện kiểm tra (!= 1). Để loại trừ tương tranh và quay vòng sẽ cần đến một kiểu tín hiệu, hay một kiểu liên lạc giữa các TT nào đó (cơ chế IPC chẳng hạn).

Kịch bản thường diễn ra giữa TT bố và TT con như sau: sau fork(), cả hai TT sẽ làm việc gì đó (ví dụ: TT bố cập nhật tệp log với số hiệu mới của TT con, TT con đI tạo một tệp cho TT bố). Trong trường hợp này ta cần để mỗi TT nói cho TT kia việc nó làm đã xong và mỗi TT đợi TT kia xong việc trước khi mỗi TT tiếp tục.
Mô hình đó như sau:

(tellwait2.c)

```
#include        <sys/types.h>
#include        "ourhdr.h"

static void charatatime(char *);

int
main(void)
{
```

```
        pid_t   pid;

        TELL_WAIT();

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {            /*TT con sẽ làm gì đấy …*/
                TELL_PARENT(getppid); /*Nói cho TT bố là đã làm xong*/
                WAIT_PARENT();          /*Dợi TT bố,  parent chạytrước */
                charatatime("output from child\n"); /*TT con tiếp tục việc của mình*/
                exit(0);
        } else
        {       /* TT bố làm việc của mình*/
                charatatime("output from parent\n");
                TELL_CHILD(pid);        /*nói cho TT : … con đã làm xong*/
                WAIT_CHILD();                   /*đợi TT con*
                charatatime("output from parent\n");  /*TT tiếp tục công việc/*/
                exit(0);
        }
        exit(0);
}

static void
charatatime(char *str)
{
        char    *ptr;
        int             c;

        setbuf(stdout, NULL);                   /* set unbuffered */
        for (ptr = str; c = *ptr++; )
                putc(c, stdout);
}
```

Khi chạy trình trên,ta sẽ có đầu ra như mong muốn, mà không có sự xen kẻ khi in ra màn hình từ hai TT. Nếu phần in đậm thay đổi như sau:

```
  if (pid == 0) {               /*TT con sẽ làm gì đấy …*/
                charatatime("output from child\n"); /*TT con tiếp tục việc của mình*/
                TELL_PARENT(getppid); /*Nói cho TT bố là đã làm xong*/
                exit(0);
        } else
        {       /* TT bố làm việc của mình*/
                WAIT_CHILD();                   /*đợi TT con*/
                charatatime("output from parent\n");
                exit(0);
        }
```

thì TT con sẽ chạy trước.

TELL_WAIT(), TELL_PARENT(), TELL_CHILD(), WAIT_PARENT(), WAIT_CHILD() là

các hàm sau, sử dụng cơ chế IPC pipe để trao đổi thông điệp giữa các TT:

```
#include        "ourhdr.h"

static int      pfd1[2], pfd2[2];

void
TELL_WAIT()
{
        if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
                err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
        if (write(pfd2[1], "c", 1) != 1)
                err_sys("write error");
}

void
WAIT_PARENT(void)
{
        char    c;

        if (read(pfd1[0], &c, 1) != 1)
                err_sys("read error");
        if (c != 'p')
                err_quit("WAIT_PARENT: incorrect data");
}

void
TELL_CHILD(pid_t pid)
{
        if (write(pfd1[1], "p", 1) != 1)
                err_sys("write error");
}

void
WAIT_CHILD(void)
{
        char    c;

        if (read(pfd2[0], &c, 1) != 1)
                err_sys("read error");
        if (c != 'c')
                err_quit("WAIT_CHILD: incorrect data");
}
```

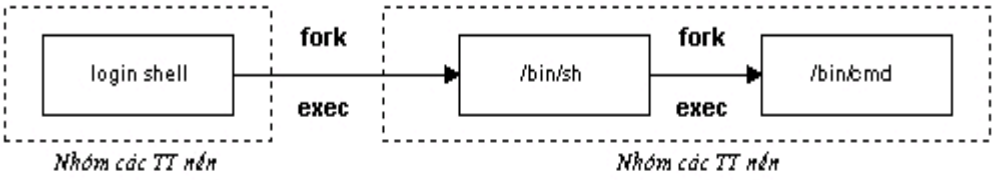
6. GHT system()

Một cách tiện lợi để cho chạy một lệnh bên trong một chương trình, là dùng hàm function(). Tuy nhiên hàm function() có một số đặc điểm, là hoạt động của hàm phụ thuộc vào hệ thống, không có giao diện với hệ điều hành, chỉ giao diện với *shell*.

#include <stdlib.h>
int system (const char *cmd string);

Nếu *cmdstring* là con trỏ NULL, thì hàm sẽ trả về giá trị khác 0 chỉ khi lệnh xác định tồn tại.

Mô hình kiến tạo khi chạy *system()* như sau:



Khi thực hiện system()

Thực tế, system() là triển khai áp dụng của fork() và exec() cũng như waitpid(), do đó giá trị trả lại sẽ có khác nhau:

(a) Nếu fork() không thành công hay waitpid() trả lại error mà error đó khác với EINTR, system(0 sẽ trả lại -1 với số hiệu errono cho biết lỗi đó;
(b) Nếu exec() có lỗi (suy ra là shell không thể thực hiện được), giá trị trả lại sẽ như thể shell thực hiện một exit(127);
(c) Trong các trường hợp còn lại, ba hàm (fork(), exec() và waitpid()) đều thực hiện được, giá trị trả về từ system() là kết quả kết thúc của shell, theo định dạng xác định trong waitpid().

Ví dụ: (system.c)
Sử dụng system() để chạy một lệnh trong hệ

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)        /* version without signal handling */
{
    pid_t   pid;
    int             status;

    if (cmdstring == NULL)
        return(1);              /* always a command processor with Unix */
```

```
    if ( (pid = fork()) < 0)
        {
            status = -1;    /* probably out of processes */

        }
    else
        if (pid == 0)
            {                           /* child */
                execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
                _exit(127);             /* execl error */

            }
        else
            {                                                   /* parent */
                while (waitpid(pid, &status, 0) < 0)
                if (errno != EINTR)
                 {
                    status = -1; /* error other than EINTR from waitpid() */
                    break;
                }
            }

    return(status);
}
```

Gỉa sử tên chương trình là system, tại dấu nhắc ta có:
$ system – ls
thì:

1. shell hệ thống tạo ra một TT mới (fork()) để chạy lệnh "system –ls" đưa vào;
2. để chạy được cần có một shell của người dùng, phảI exec("bin/sh");
3. bin/sh tạo một TT khác (fork()) đề chạy "ls –l" bằng exec("/bin/ls –l").

Ví dụ trên chưa đề cập tới sử lí signal. Trong lệnh execl() có tuỳ chọn –c là để lấy đối đầu vào tiếp theo (-l), (char *) 0) thông báo kết thúc xâu lệnh đưa vào.

Có gì khác biệt khi ta không dùng shell để chạy lệnh? Trong trường hợp này ta phảI dùng *execlp()* (*execlp (const char *filename, const chả *arg0, ...,/* (char *) 0*/)*), thay cho execl() và sẽ phảI dùng biến PATH (tựa như dùng shell), phảI xử lí kết thúc xâu để sau đó gọi execlp(), và không thể sử dụng các kí tự metal của shell. Công việc xem ra có phần phức tạp hơn !

Tại đây, dùng _exit() để loại trừ việc tạo tệp từ nội dung của các các bộ đệm I/O chuẩn đã sao chép từ TT bố sang cho TT con.

Ví dụ tiếp theo của system() có quan tâm tới xử lí signal:, kích hoạt lệnh soạn thảo văn bản "/ed", là một trình ứng dụng có sử dụng cơ chế tương tác qua việc đón sígnl và thoát khỏi signal khi gõ từ bàn phím DEL, Ctrl_C:

```
#include    <sys/types.h>
#include    <signal.h>
#include    "ourhdr.h"
```

```
static void     sig_int(int), sig_chld(int);

int
main(void)
{
        int             status;

        if (signal(SIGINT, sig_int) == SIG_ERR)
                err_sys("signal(SIGINT) error");
        if (signal(SIGCHLD, sig_chld) == SIG_ERR)
                err_sys("signal(SIGCHLD) error");

        if ( (status = system("/bin/ed")) < 0)
                err_sys("system() error");
        exit(0);
}

static void
sig_int(int signo)
{
        printf("caught SIGINT\n");
        return;
}

static void
sig_chld(int signo)
{
        printf("caught SIGCHLD\n");
        return;
}
```

7. Kích hoạt một chương trình

GHT *exec* sẽ kích hoạt một chương trình khác, phủ lên không gian bộ nhớ của TT gốc bằng bản sao của tệp thực thi. Nội dung của *user - level context* có trước *exec* sau đó không truy nhập được nữa, ngoại trừ các thông số của *exec* mà kernel đã sao chép từ không gian địa chỉ cũ sang không gian địa chỉ mới. PID của TT củ không đổi và TT mới do *exec* tạo ra sẽ nhận PID đó.

execve (filename, argv, envp)

filename:tên tệp thực thi sẽ kích hoạt,

argv: trường các con trỏ kí tự trỏ vào các xâu kết thúc bằng kí tự NULL. Các xâu này tạo thành danh sách đối đầu vào cho TT mới được tạo.

envp: con trỏ trỏ vào các xâu kí tự tạo thành môi trường của tệp thực thi.

Trong thư viện C có sáu *exec* như sau:

*execl (const char * pathname, const char *arg0, ... / * (char *) 0 */);*

*execv (const char * pathname, char *const argv[ ]);*

*execle (const char * pathname, const char *arg0, ... /*(char *) 0, char *const envp [ ] */);*

*execve (const char * pathname, char *const argv[ ], char *const envp [ ]);*

*execlp (const char * filename, const char *arg0, ... / * (char *) 0 */);*

*execvp (const char * filename, char *const argv [ ]);*

Để dễ nhớ dùng bảng sau:

Hàm	Đường dẫn	Tên tệp thực thi	Danh sách đối	argv [ ]	environ	envp [ ]
execl	√		√		√	
execlp	√	√	√		√	
execle			√			√
execv	√			√	√	
execvp		√		√	√	
execve	√			√		√
		p	l	v		e

Các chử cái có ý nghĩa như sau:

p: hàm lấy đối filename và dùng biến mmôI trường PATH để tìm tệp thực thi;

l: Hàm lấy một danh sách các đối là sự loại trừ lẫn nhau với v;

v: hàm lấy đối ở argv[ ];

e: Hàm lấy đối môI trường từ envp[ ] thay cho môI trường hiện hành.

Khi một chương trình dùng dòng lệnh:

main(argv, argv)

thì trường *argv* là bản sao của thông số *argv* cho *exec*. Các xâu kí tự trong envp có dạng " *name* = *value*" và chứa các thông tin hữu ích cho chương trình (chẳng hạn *user's home directory*, đường dẫn tìn tệp thực thi). TT truy nhập các biến môi trường của nó qua biến tổng thể *environ* được khởi động bằng chu trình thực thi (*routine*) của C.

exec()

input: 1. file name

2. parameter list

3. environment variables list

output: none

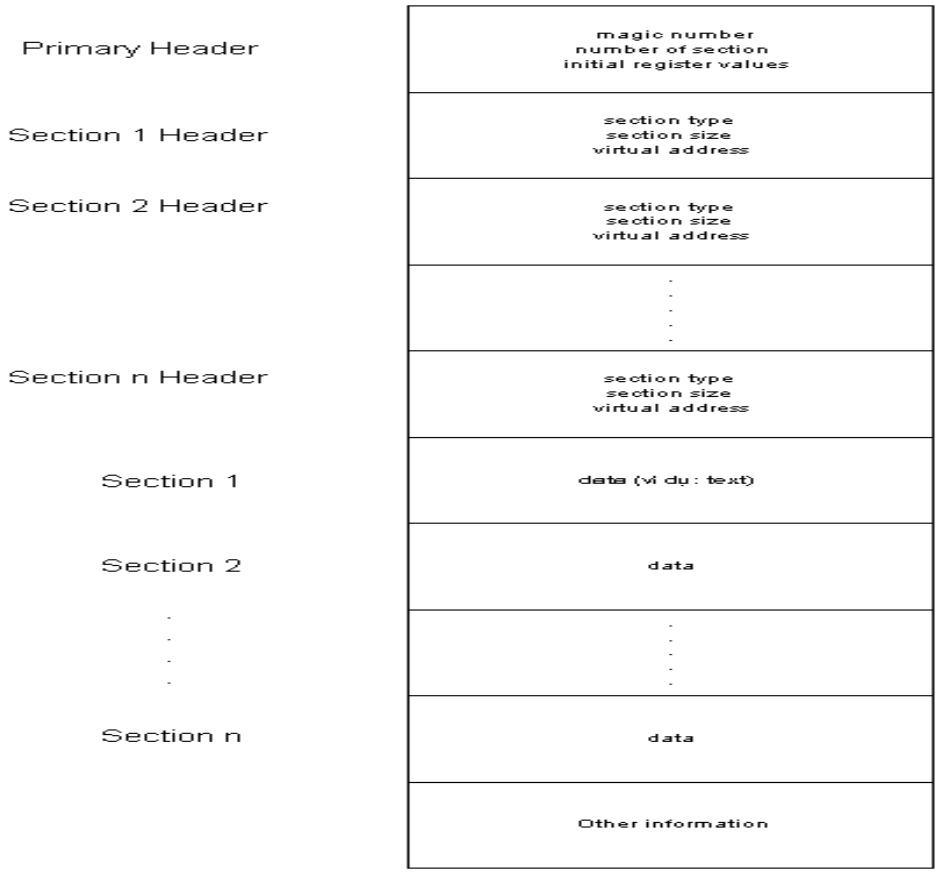
{

```
        .get file inode(namei());
        .verify file is executable, user has permission to execute;
        .read file header, check that it is a load module;
        .copy exec parameters from old address space to system space;
        .for (every region attached to process)
                detach all old region;
        .for (very region specified in load module)
        {
                allocate new regions;
                attach the regions;
                load region into memory if applicable;
        }
        .copy exec parameters into new user stack regions;
        .special processing for setuid programs, tracing;
        .initialize user register save area for return to user mode;
        .release inode of file;
}
```

Dưới đây là dạng của tệp thực thi trong FS thường được tạo ra bằng *hợp ngữ (assembler) hay trình nạp (loader):*

1. *Primary Header:* cho biết có bao nhiêu phần (*section*) trong tệp, địa chỉ đầu để thực hiện, *magic number* là một số nguyên ngắn cho biết kiểu của tệp thực thi, (là tệp khả nạp...) giúp kernel phân biệt các đặc trưng chạy trình (*run time*) của tệp.
2. *Section header:* mô tả mỗi phần trong tệp, cho kích thước của phần đó, địa chỉ ảo khi phần đó chạy trong hệ và các thông tin khác.
3. *Phần data:* chẳng hạn là text, được nạp vào không gian địa chỉ của TT.
4. *Other information:* các bảng biểu tượng (*symbols*), các dữ liệu khác (cho *debug* chẳng hạn).

Bởi vì các thông số cho *exec* là địa chỉ của các xâu kí tự trong không gian địa chỉ của user, kernel copy các địa chỉ đó cũng như các xâu kí tự vào không gian của kernel (thông thường là vào *kernel stack*).



Cách đơn giản nhất để sao chép các thông số vào *user_level context* mới là dùng *kernel stack*, tuy nhiên do các thông số có độ dài không biết trước được trong khi stack có giới hạn, nên có thể sẽ dùng một trang nào đó để thực hiện.

Sau khi copy các thông số kernel sẽ giải phóng các miền bộ nhớ cũ của TT, cấp miền mới cho chương trình mới, nạp nội dung của tệp thực thi sẽ kích hoạt vào các miền đó, khởi động các data của chương trình. Cuối cùng kernel cấp phát miền *user stack* cho TT, copy các thông số tạm lưu nói trên vào *user stack*, chuẩn bị *user context* cho *user mode*. Kernel thực hiện tiếp một số các xử lí đặc biệt để thực hiện *setuid*: đặt các trường *effective user ID* trong *procces table entry* tương ứng với TT, và trong *u_area* vào số định danh của người sở hữu tệp thực thi. (Xem *setuid()* ở phần sau).

Cuối cùng kernel chuẩn bị các giá trị của các thanh ghi trong *register context* của TT để sau này trở về *user mode*. Kernel kết thúc thuật toán, *exec*, TT chuyển sang thực hiện mã của chương trình mới.

exec thực tế biến đổi TT gọi (*calling procces*) thành một TT mới. TT mới được xây dựng từ tệp thực thi. Không gian địa chỉ của TT cũ bị ghi đè bởi TT mới nên TT gọi không có *return* khi *exec* thành công, mà thực hiện mã của chương trình mới. *Tuy nhiên PID không thay đổi, cũng như vị trí của TT trong File table cũng không thay đổi, mà chỉ* có *user_level context* của TT đã thay đổi để phù hợp với môi trường của chương trình được kích hoạt.

Ví dụ:

main()

```
{
        int status;
        if (fork() == 0)
                execl(" /bin/date","date",0);
        wait(&status);
}
```

Ngay lập tức sau khi TT bố và TT con thực hiện xong GHT *fork()*, chúng sẽ độc lập thực hiện. Khi TT con đi vào kích hoạt *exec()* thì miền mã của nó gồm các lệnh chương trình, miền data có xâu "*/bin/date*", "*date*", stack có các giá trị cho nó, sẽ bị đẩy đi (*pushed*) để thực hiện GHT *exec*. Tại *user_level*, kernel tìm tệp thực thi "*date*", kiểm tra các tiêu chí (trong *header*) và sao xâu "/bin/date" và "date" vào hệ thống, giải phóng các miền nhớ TT con đã dùng, xin cấp phát không gian mới cho chương trình mới, copy mã của tệp "date" vào miền text, data vào miền data (= nạp chương trình mới vào bộ nhớ). Kernel tái xây dựng danh sách các thông số và đặt vào stack. Công việc mà *exec* cần làm đã xong, GHT *exec* kết thúc, TT con không thực hiện mã chương trình cũ nữa mà thực hiện chương trình mới "date". Trong khi đó TT bố *wait()* để TT con chạy xong "date" và nhận kết quả qua biến *status*.

Các TT thông thường kích hoạt *exec* sau khi thực hiện *fork*. TT con sau *fork* sao chép không gian địa chỉ của TT bố, và loại bỏ địa chỉ đó khi *exec*, thực hiện một chương trình khác với cái mà TT bố thực hiện. Liệu có nên phối hợp hai GHT này làm một để kích koạt một chương trình và chạy chương trình đó như một TT mới ? Hãy xem thêm Ritchie, D. M., "*The evolution of Unix Time-sharing System*". Tuy nhiên về chức năng, *fork* và *exec* tách biệt lại rất quan trọng. Bởi các TT có thể thao tác các mô tả tệp vào và mô tả tệp ra chuẩn một cách độc lập để tạo *pipe* thuận lợi hơn là khi hai GHT này được kết hợp làm một.

Cho tới nay ta đã giả định rằng miền *text* và *data* của TT đang chạy là tách biệt. Điều đó mang lại các thuận lợi như: bảo vệ miền và chia sẻ miền: kernel có thể phối hợp với hardware (*MMU*) để loại trừ miền bị ghi đè (do sai trong quản lí địa chỉ). Khi nhiều TT cùng thực hiện một chương trình, chúng sẽ chia sẻ text để chạy với các con trỏ lệnh riêng của mỗi TT, tiết kiệm bộ nhớ. Trong quá trình thực thi *exec*, kernel sẽ kiểm tra tệp thực thi khả năng *share* qua *magic number* và *share text* cho các TT khác.

Ví dụ của exec: (exec1.c)

```
#include        <sys/types.h>
#include        <sys/wait.h>
#include        "ourhdr.h"

char    *env_init[ ] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
        pid_t   pid;

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {      /* specify pathname, specify environment */
                if (execle("/home/SV1/bin/echoall",
```

```
                        "echoall", "myarg1", "MY ARG2", (char *) 0,
                        env_init) < 0)
                err_sys("execle error");
        }
        if (waitpid(pid, NULL, 0) < 0)
                err_sys("wait error");

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {      /* specify filename, inherit environment */
                if (execlp("echoall",
                        "echoall", "only 1 arg", (char *) 0) < 0)
                err_sys("execlp error");
        }
        exit(0);
}
```

8. Định danh người dùng của một TT

8.1. Số định danh người dùng

- Định danh của người dùng thực (real user UID): đây là mã hiệu người dùng trên hệ, đã khởi động tiến trình.

- Định danh của người dùng hiệu quả (effective UID); đây là mã hiệu được hệ thống sử dụng để kiểm soát truy nhập, nó có thể khác với định danh của người dùng thực, đặc biệt trong trường hợp các chương trình có lập bit "setuid".

- Định danh của nhóm thực (real group user): mỗi thành viên thực phảI thuộc một nhóm thực, mỗi nhóm có một số định danh là real group user (RGID).

- Định danh của nhóm hiệu quả (effective UID): đây là mã hiệu được hệ thống sử dụng để kiểm soát truy nhập: nó có thể khác với nhóm thực, đặc biệt trong trường hợp các chương trình có lập bit "setgid".

- Danh sách các định danh nhóm: một người sử dụng đồng thời có thể thuộc nhiều nhóm và kernel giữ một danh sách gồm các nhóm kết hợp với mỗi tiến trình để thực hiện kiểm soát truy nhập. (ở hệ điều hành Linux, một tiến trình có thể có đến 32 nhóm).

Số định danh người dùng (*user ID, viết tắt UID*) trên hệ là một giá trị số, mà hệ dùng để nhận biết người dùng. Số này phát sinh khi người quản trị tạo một người dùng mới và sao cho không có số trùng lặp, và không thể thay đổi. Kernel sử dụng UID để kiểm soát các quyền hạn mà user có trên hệ thống. Root hay *superuser* có UID = 0 và có các đặc quyền mà các người dùng thông thường không có. Thực tế user có hai số định danh và có liên quan khi chạy chương trình và kernel kết hợp hai số định danh đó vào cho một TT độc lập với số định danh của TT (PID):

real user ID và effective user ID

- số định danh người dùng thực (*real user ID)* cho biết ai cho chạy một TT;
- số định danh người dùng có hiệu lực (*effective user ID*) dùng để xác định quyền sở

hữu cho một tệp mới tạo, để kiểm tra quyền truy nhập, quyền gởi tín hiệu cho một TT bằng *kill()*.

Kernel cho phép một TT thay đổi *effective UID* khi TT kích hoạt (*exec*) chương trình *setuid()* hay thực hiện GHT *setuid()*. Thông thường *effective user ID = real user ID*.

setuid() là một tệp thực thi thực hiện đưa *bit setuid* trong trường quyền truy nhập của TT lên: kernel lập trường *effective user ID* trong *proccess table* của TT và *u_area* của TT thành ID của người sở hữu tệp. Để phân biệt hai trường này, hãy gọi trường trong *proccess table* là *saved user ID* và thực hiện minh hoạ:

setuid(uid)

Trong đó: *uid* là một user ID mới và kết quả phụ thuộc vào giá trị hiện tại của *effective UID:*

- Nếu *effective user ID* của TT gọi là superuser, kernel sẽ lập lại trường *effective user* và *real user* trong proccess table và u_area.
- Nếu *effective user ID* của TT gọi không phải là *superuser*, kernel sẽ lập *effective user ID* trong u_area là *uid* nếu *uid* có giá trị của *real user ID* hoặc các giá trị của *saved user ID*;
- Các trường hợp khác báo lỗi.

Nói chung TT con thừa kế *real user* ID và *effective user ID* từ TT bố trong quá trình *fork* và duy trì các giá trị của TT qua GHT *exec*.

effective user ID trong u_area là kết quả của GHT *setuid()* hoặc của việc thực hiện *setuid* trong *exec* và *setuid()* chịu trách nhiệm định đoạt quyền truy nhập tệp. Các TT dùng giá trị ở *saved user ID* để khôi phục lại *effective user ID*.

Trình *login* dùng để vào hệ là chương trình điển hình sử dụng GHT *setuid()*. *Login* lập *user root (superuser)* và chạy bằng *root effective user ID*, sau đó đợi user bất kì gõ tên *login* và mật khẩu, tiếp theo đó kích hoạt *setuid()* để lập *real user* ID và *effective user ID* cho user đang login vào máy (user đó có trong danh sách /etc/passwd). Cuối cùng trình login dùng *exec* để chạy shell bằng *real user* ID và *effective user ID* đã lập cho user đó.

Trình *mkdir()* cũng dùng *setuid()*, nhưng chỉ cho *effective user ID superuser* mà thôi. Để các user khác có khả năng tạo thư mục, thì *mkdir* là một trình có *setuid* sở hữu bởi root, khi chạy sẽ dùng quyền hạn của root tạo thư mục cho user, sau đó thay đổi người sở hữu, quyền truy nhập thư mục cho user đó (*real user*).

Hãy thử với trình sau đây bằng hai user khác nhau, một người tên **maury** có tệp của mình là tệp **maury**, người kia **mjb** có tệp là **mjb** và đều là tệp có thuộc tính "chỉ đọc". Chương trình được dịch bởi **maury** và là người sở hữu trình đó.

```
#include <fcntl.h>
main()
{
        int uid; /*value of real user ID*/
        int euid; /*value of effective user ID*/
        int fdmjb, fdmaury; /* file descriptors*/
        uid = getuid();/*get real UID*/
        euid = geteuid(); /*get effective UID*/
        printf("uid = %d euid = %d \n",uid, euid);
        fdmjb = open("mjb", O_RDONLY);
```

```
    fdmaury = open("maury", O_RDONLY);
    printf("fdmjb = %d fdmaury = %d \n",fdmjb, fdmaury);
    setuid(uid);
    printf(""after setuid(%d): uid = %d euid = %d \n", uid, getuid(), geteuid());
    fdmjb = open ("mjb",O_RDONLY);
    fdmaury = open("maury",O_RDONLY);
    printf("fdmjb = %d fdmaury = %d \n",fdmjb, fdmaury);
    /*reset effective UID*/
    setuid(euid);
    printf(""after setuid(%d): uid = %d euid = %d \n", uid, getuid(), geteuid());
}
```

Giả sử ID của abc là: 5088, ID của xyz là 8319.

Khi user tên là **mjb** chạy trình, kết quả như sau:

```
Uid = 5088, euid = 8319
fdmjb = -1 fdmaury = 3        /*=-1 không có quyền truy nhập tệp*/
after setuid(5088): uid = 5088 euid = 5088
fdmjb = 4 fd fdmaury = -1
after setuid(8319): uid= 5088 euid = 8319
```

Khi user tên là **maury** chạy trình, kết quả mhư sau:

```
uid = 8319      euid=8319
fdmjb = -1 fdmaury = 3        /*=-1 không có quyền truy nhập tệp*/
after setuid(8319): uid = 8319 euid = 8319
fdmjb = -1 fdmaury = 4
after setuid(8319): uid= 8319 euid = 8319
```

Chương trình cho thấy hiệu ứng do *setuid()* tạo ra trên các tệp có chế độ sở hữu khác nhau, và có ích khi chia sẽ các tệp cho người khác sử dụng.

8.2. Số định danh nhóm người dùng

Nhóm trong Unix đực dùng để nói lên sự lựa chọn hay tập hợp của một số người dùng có khả năng chia sẻ tài nguyên (tệp, tư liệu, ứng dụng . . .), sau đó hệ thống sẽ gán cho một giá trị số.

9. Các lệnh liên quan tới sở hữu và quyền trên tệp

91. Thay đổi quyền trên tệp:

chmod u+x *têntệp*

chmod g+wx o+wx *têntệp*

chmod g-rwx *têntệp*

9.2 Đối tượng truy xuất tệp: u (user), g (group), o (other)

Nhóm của đối tượng: khi người quản trị tạo một user mới, thì người đó sẽ gán user

vào một nhóm nào đó bằng các cách sau đây:

groupadd *tên_nhóm_mới_tạo*

useradd *tên_user_mới*

useradd *tên_của_user* **-g** t*ên_nhóm* **-d** *tên_thư_mục_home_của_user* **-p** *mật_khẩu*

1. Cấp quyền sử dụng thư mục cho user, tác dụng cho tất cả user

chmod o-w /home/abc

2. Chia sẽ thư mục cho nhóm các user, tác dụng: chỉ cho một nhóm

chgrp *tên_nhóm* /home/abc

3. Thay đổi người sở hữu, tác dụng: duy nhất một user cụ thể

chown *tên_user* /home/abc

10. Thay đổi kích thước của TT

TT có thể điều chỉnh miền dữ liệu (*data region*) bằng

brk(endds)

Trong đó: *endds* là giá trị địa chỉ ảo cao nhất của miền dữ liệu của TT.

Phiên bản khác của lệnh này là:

oldendds = sbrk(increment)

Trong đó*:* *increment* sẽ xác định số bytes thay đổi,

oldendds giá trị trước khi thực hiện *sbrk()*;

```
brk()
      input:    new break value
      output:   old break value
{
        .lock process data region;
        .if (region size increasing)
                if(new region is illegal)
                {
                        .unlock data region;
                        .return(error);
                }
        .change region size (growreg());
        .zero out addresses in new data space;
        .unlock process data region;
}
```

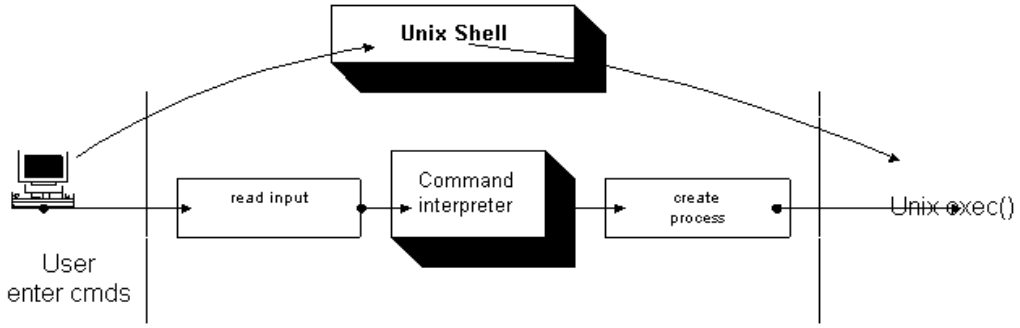
11. Shell

Chúng ta đã có các hàm chức năng để thao tác một TT, và cũng có nghĩa có thể mô tả

hoạt động của *shell*. Shell đã có đề cập ở đầu phần về TT, tuy nhiên shell tương đối phức tạp.

Hãy tìm hiểu qua ví dụ: Shell đọc dòng lệnh từ đầu vào chuẩn (*stdin, stdout* là các tệp có cho mọi TT trong *file descriptor* của mỗi TT, và gán cho terminal để thực hiện shell login), thông dịch theo luật cố định. Nếu xâu đầu vào là các lệnh bên trong của hệ (*built - in commands*), shell tạo TT để thực hiện, nếu không phải, shell giả định đó là tên của một tệp thực thi của user, shell cũng tạo TT để thực hiện.

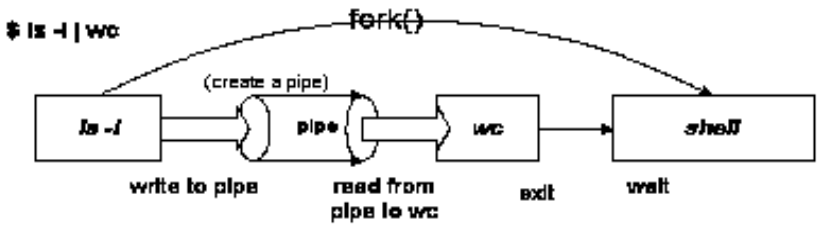
shell tạo TT con (*fork()*), TT con này sẽ gọi một trong các *execs,* kích hoạt lệnh user đưa vào. TT bố, tức shell, mà user sử dụng sẽ *wait()* cho tới khi TT con *exit* và vòng lặp nhận lệnh bắt đầu từ đầu.



Ví dụ:

Shell xử lí lệnh đưa vào với pipe: đầu ra của lệnh *ls* chuyển qua *pipe* tới đầu vào của lệnh *wc*:

$ ls -l | wc



Chu trình chính của shell: mô tả quá trình thực hiện dòng lệnh trên

```
/* shell đọc dòng lệnh do user đưa vào cho tới khi có End of File*/
/* shell làm phân tích dòng lệnh đưa vào để ở stdin:*/
while (read (stdin, buffer, numchars))
{
        /*phân tích lệnh*/
        if (/* dòng lệnh có chứa &, chạy background*/)
                amper = 1;   /*có &, biến amper đặt = 1, shell sẽ start từ*/ /*đầu*/
        else
                amper = 0;
                /*lệnh không thuộc ngôn ngữ của shell:*/
```

```
if (fork() ==0) /* tạo TT con để thực hiện các bước tiếp theo*/
{
        /* 1. Lệnh có đổi hướng vào ra I/O?: */
        if (/* redirect output */)
        {
                fd = create (newfile, fmask);
                close(stdout);
                dup(fd);
                close(fd);
                /* đầu ra đã đổi hướng*/
        }
        if (/* piping */) /*lệnh có dùng pipe: ls -l | wc */
        {
                pipe(fildes);
                if (fork() == 0) /* tạo TT cháu để thực hiện lệnh */
                /*đầu: ls -l*/
                {
                        close(stdout);
                        du(fildes[1]);
                        close(fildes[1]);
                        close(fildes[0]);
                        /*đầu ra stdout của lệnh thứ nhất chuyển vào pipe*/
                        /*TT con kích hoạt exec để thực hiện lệnh:*/
                        execlp(command1, command1,0);
                }
                / *TT bố tiếp tục lệnh thứ hai wc*/
                close(stdin);
                (fildes[0]);
                close(fildes[0]);
                close(fildes[1]);
                /*đầu ra của pipe -> stdin*/
        }
        execve(command2, command2, 0);    /* thực hiện lệnh wc */
        }
        /*TT bố tiếp tục chạy tại đây... hay đợi TT con (wc) kết thúc nếu cần*/
        if (amper ==0) /*không phải lệnh có &*/
        retid = wait(&status);
```

```
        }
/*shell quay về đầu nhận lệnh khác của user*/
```

12. TT boot và init

Để đưa hệ vào hoạt động cần qua bước khởi động, gọi là "***bootstrap***":

- bật náy,
- nạp chương trình mồi (*hardcoded bootstrap*: vi lệnh có mã ở phần cứng);
- bootstrap chạy, mồi một chương trình khác (*loader*) (*Unix: đọc boot block: 0 của đĩa và nạp vào bộ nhớ*);
- *loader* thực hiện: nạp Hệ điều hành từ FS xuống (Unix: "/unix");
- *bootstrap* hoàn tất, chuyển điều khiển cho HĐH (Unix: kernel chạy):
  - kernel khởi động các cấu trúc dữ liệu hệ thống: *buffers, inodes, page table, file table, process table, region table,...*;
  - *mount FS* trên *root* ("/");
  - tạo môi trường cho TT số 0 (tạo u_area, khởi động đầu vào cho TT 0 trong *procces table*, root là thư mục của TT 0...;
  - chuyển sang chạy TT 0 trong *kernel mode*, tạo TT 1 với *context* của nó (*user và kernel context*), TT 1 chạy trong *kernel mode*, copy mã của kernel, tạo *user - level context* cho nó; TT 0 đi ngủ và thành *swapper.*
  - TT 1 "*return*" từ *kernel* sang *user mode*, thực hiện mã vừa sao chép từ kernel. TT 1 trở thành TT ở *user_level*, TT 0 là *kernel_level* chạy trong kernel; TT 1 gọi *exec* thực hiện chương trình "/etc/init". Từ đây TT 1 có tên là *TT init*.

```
Boot system start
        Input:    none
        Output:   none
{
        init all kernel data structures;
        pseudo - mount of root;
        hand - craft envirinment of proccess 0;
        fork proccess 1:
        {
                /*TT 1 ở đây*/
                allocate regions;
                attach region to init address space;
                grown region to accommodate code about copy;
                copy code from kernel space to init user space to exec init;
                change mode: return from kernel to user mode;
```

```
            /*init không quay lại đây nữa vì đã chuyển mode,*/
            /*init thực hiện " /etc/init" bằng exec và trở thành user process*/
            /*bình thường*/
        }
        /*TT 0 tiếp tục*/
        fork kernel processes;
        /*TT 0 kích hoạt swapper để quản lí bộ nhớ và thiết bị nhớ (đĩa)*/
        /*TT 0 thường là ngủ nếu không có gì để làm*/
        execute code for swawapper algorithm;
}
```

TT ***init*** là một TT điều vận (*dispatcher*), khởi động các TT khác qua "/etc/initab", khởi động các chế độ chạy máy (ví dụ: *multi-user state (2)*), khởi động các TT *gettys*, *login shell*... *init* còn thực hiện *wait()* kiểm soát các TT đã chết, thu hồi tài nguyên...

Các TT trên Unix:

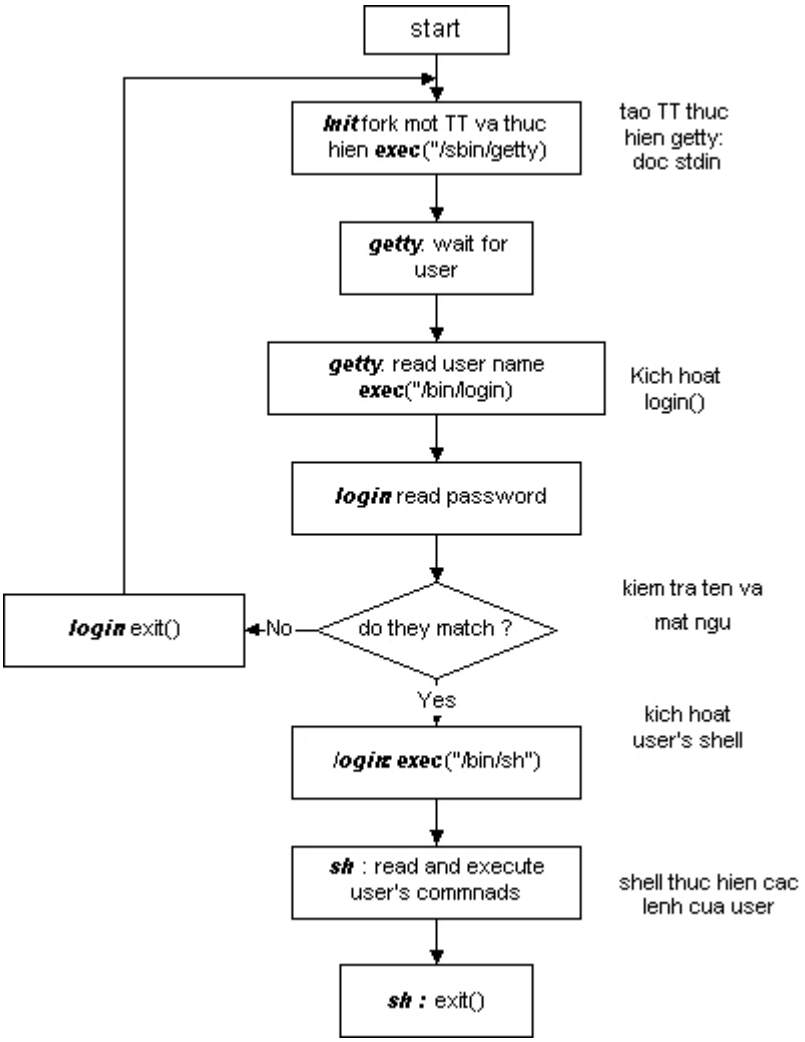
- thông thường là TT của user, kết hợp với user tại terminal
- hay *daemons* (các trình dịch vụ không kết hợp với user nào chạy ngầm) nhưng cũng giống như các TT user khác, chạy ở user mode thực hiện các GHT để truy nhập các dịch vụ hệ thống,
- hay TT kernel chỉ chạy trong kernel mode. TT 0 phát sinh ra các TT kernel để sau đó chuyển thành TT *swapper*. Các TT kernel tương tự các TT *daemon* ở chỗ chúng cung cấp rộng rãi các dịch vụ hệ thống, nhưng các TT này lại có khả năng kiểm soát được mức độ ưu tiên thực hiện bởi mã của chúng là một phần của kernel. Chúng có thể truy nhập trực tiếp vào các thuật toán và cơ sở dữ liệu của kernel mà không dùng GHT, chính vì vậy các TT kernel rất mạnh. Tuy nhiên mỗi lần sửa đổi kernel, thì phải thực hiện dịch lại kernel, trong khi các TT *daemon* không cần dịch lại.

init (TTsố 1)

```
    Input:    none
    Output:   none
{
    fd = open ("/etc/init", O_RDONLY);
    while (line_read(fd, buffer))
    {
            /*read every line of file*/
            if (invoked state != buffer state)
            continue; /* loop back to while*/
            /*state matched*/
            if (fork() == 0)
            {
```

```
                execl("process specified in buffer");
                exit();
            }
            /*init process does not wait*/
            /*loop back to while*/
        }
        while((id = wait((int *) 0)) != -1)
        {
            /* check if spawned child died;*/
            /*consider respawning it*/
            /*otherwise, just continue*/
        }
}
```

Ví dụ ***init*** với quá trình **login** tại t/b đầu cuối:



13. Tóm tắt và bài tập

Tóm tắt

Chương này đề cập các GHT thao tác bối cảnh của TT và kiểm soát quá trình thực hiện (chạy) một TT.

- *fork()* tạo ra một TT mới bằng cách nhân bản tất cả các miền của TT bố cho TT con. Phần tinh vi hơn cả của fork() là để khởi động *saved register context* của TT con sao cho nó khởi động thực hiện bên trong GHT fork() và ghi nhận rằng nó là TT con.
- *exit()* dùng để các TT kết thúc, trả lại các miền (địa chỉ, bộ nhớ) cho hệ thống mà TT đã dùng và gởi tín hiệu "death of child" cho TT bố.
- TT bố có thể đồng bộ thực hiện với TT con bằng *wait();*
- *exec()* cho phép một TT kích hoạt một chương trình, mã (nội dung) của chương trình xác định trong exec() phủ lên không gian địa chỉ của TT gốc: kernel giải trừ các miền trước đó của TT, cấp phát các miền mới tương ứng cho chương trình được kích hoạt, sau khi thực hiện xong, không thể trở về chương trình gốc được. Ví dụ:

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
        printf (" chạy lệnh ps bằng GHT execlp()\n");
        execlp ("ps", "ps", " –ax",0);
        prìntf (" Sẽ không bao gờ thực hiện đến lệnh này nữa !!!"\n);
        exit(0);
}
```

- Kernel cho phép một user bình thường thực hiện chương trình bằng các quyền hạn của các user khác bằng *setuid()*.
- Các TT kiểm soát việc thực hiện của chúng qua GHT signal(). Khi TT chấp nhận (xử lí) một tín hiệu, kernel thay đổi *user stack và user saved register context* để chuyển tới địa chỉ của hàm chức năng xử lí tín hiệu. Các TT có thể dùng *kill()* để gởi tín hiệu đi đến từng TT hay nhóm các TT qua GHT *setpgrp()*.
- *Shell* và *init* dùng các GHT chuẩn để tạo ra các chức năng tinh xảo có trong kernel của hệ. Shell dùng GHT để thông dịch lệnh đầu vào, chuyển hướng các *stdin stdout stderror*, sinh ra các TT, tạo *pipe* giữa các TT phát sinh, đồng bộ thực hiện với các TT con, ghi nhận trạng thái khi các lệnh kết thúc. *Init* tạo ra các TT khác nhau, đặt biệt là vai trò kiểm soát kết thúc thực hiện của TT.

Bài tập

1. Chạy chương trình sau tại terminal và sau đó đổi hướng đầu ra từ màn hình thành tệp, so sánh kết quả:

```
main()
{
    printf("hello\n");
    if(fork == 0)
        printf(" Đây là mã của TT con"\n");
}
```

2. Liệu có thể xảy ra trường hợp mất các thông báo nếu TT nhận nhiều signals tức thời trước khi TT có cơ hội để phản ứng ? (Ví dụ bằng chương trình TT đếm các signal ngắt nó nhận được). Liệu có cách giải quyết để loại bỏ tình huống đó ?
3. Khi TT nhận signal mà không xử lí, kernel tạo ra một tệp toàn cảnh (image) của TT đó (dump image của TT: *dump core*), tệp đó gọi là "core" tại thư mục hiện hành của TT. Kernel copy các miền *u_area, code, data, stack* của TT vào tệp này. User sau đó dùng để debug TT. Hãy tạo thuật toán (các bước) kernel thực hiện để tạo ra tệp này. PhảI làm gì nếu tại thư mục hiện hành đã có tệp "core"? Kernel sẽ phải làm gì nếu có nhiều TT cùng dump core ở cùng một thư mục ?
4. Ví dụ dưới cho thấy một TT " tấn công" một TT khác bằng các signal mà TT kia sẽ chặn bắt. Hãy thảo luận xem điều gì sẽ xảy ra nếu *thuật toán xử lí tín hiệu* thay đổi theo một trong hai cách sau đây:

   . kernel không thay đổi hàm xử lí tín hiệu cho tới khi user yêu cầu rỏ ràng phải thay đổi;

   . kernel tác động đến TT để TT bỏ qua tín hiệu cho tới khi user thực hiện GHT *signal()* một lần nữa.

```
#include <signal.h>
sigcatcher()
{
    printf("PID %d caught one \n",getpid());
    signal(SIGINT,sigcatcher);
}
main()
{
    int ppid;
    signal(SIGINT,sigcatcher);/* TT bố nhận tín hiệu ngắt*/
    if (fork() == 0) /*TT bố tạo TT con*/
    /*code của TT con: */
    {/*để một khoảng thời gian đủ cho 2 TT xác lập*/
        sleep(5);
    ppid = getpid(); /* lấy pid của TT bố*/
                        /*TT con dùng kill() gởi SIGINIT*/
                        /* cho TT bố nếu bố còn sống:*/
```

```
    for (;;)
        if(kill(ppid,SIGINIT) == -1)
            exit();
    }
/*TT bố hạ mức ưu tiên của mình: */
nice(10);
for(;;);    /* TT bố chạy lòng vòng để tồn tại*/
}
```

5. Một TT đi kiểm tra signals khi TT đi vào hay ra khỏi trạng thái ngủ, và khi trở về user mode từ kernel mode sau khi hoàn tất một gọi hệ thống hay sau khi hoàn tất thao tác ngắt. Tại sao TT không thể kiểm tra signal trong khi đang chuyển vào thực hiện một gọi hệ thống ?
6. Cho một GHT mới: ***nowait(pid)*** trong đó pid cho số hiệu của TT con của TT phát sinh ra GHT này. Khi thực hiện *nowait(pid)*, TT gọi (bố) sẽ thông báo cho kernel đừng bao giờ đợi TT con kết thúc (*exits*), do vậy kernel sẽ dọn dẹp ngay *slot* đầu vào của *process table* khi TT con chết (không để lại "*zombie*". Hãy thảo luận giá trị của *nowait()* và so sánh với việc dùng *"death of child"* signal. Kernel sử dụng giảI pháp này như thế nào?
7. *wait()* sẽ tìm thấy thông tin gì khi TT con kích hoạt *exit()* mà không có thông số ? (Tức là khi TT con dùng *exit()* thay vì dùng *exit(n)).* Nếu người lập trình dùng *exit()*, thì giá trị gì *wait()* chờ đợi để kiểm tra?
8. *Superuser* là người duy nhất được quyền ghi lên tệp "/etc/passwd". Chương trình *passwd()* cho phép user thay đổi mật khẩu của mình, và không cho thay đổi mật khẩu của user khác. Vậy chương trình này làm việc như thế nào?
9. Khi shell tạo ra một TT mới để thực hiện một lệnh (*command*), làm thế nào shell biết được đó là tệp thực thi? Nếu là tệp thực thi được thì làm thế nào shell có thể phân biệt được đó là tệp loại shell script và tệp do comliler tạo ra ? Trình tự chuẩn xác để kiểm tra các trường hợp trên là như thế nào?
10. Khi user gõ "*delete*" hay "*break*" trên bàn phím, thì *terminal driver* sẽ gởi *interrupt signal* cho tất cả các TT trong nhóm của login shell. User thực tế đã kết thúc cả các TT phát sinh bởi shell nhưng lại không muốn logoff. Vậy chương trình shell trong chương này phải cải tiến thế nào?
11. Chỉ có một TT init là TT số 1 trên hệ. Tuy nhiên system adminitrator có thể thay đổi trạng thái hệ thống bằng phát động init. Ví dụ lúc khởi động hệ thống chạy ở chế độ single user (console hoạt động, các terminals bị khóa), do vậy adminitrator đưa vào lệnh:

    # init 2

    để chuyển sang multi-user mode. Shell sẽ *forks* (tạo TT mới) và *execs* (kích hoạt) init với thông số là 2. Điều gì xảy ra với một hệ nếu chỉ có một TT init được phép kích hoạt ?

Chương IV. Lập biểu và thời gian

Trong các hệ phân chia thời gian, kernel sẽ phân phối CPU cho các TT trong một khoảng thời gian nhất định (gọi là ***time slice***). Khi lượng thời gian này kết thúc kernel sẽ lập biểu (***schedule***) chọn TT khác và cấp CPU cho TT đó. Chức năng lập biểu của Unix dùng thời gian tương đối để thực hiện mã (code) của một TT, làm thông số xác định TT nào sẽ tiếp theo sẽ được chạy. Mỗi TT hoạt động đều có mức lập biểu ưu tiên. Khi kernel thực hiện chuyển bối cảnh, kernel sẽ chuyển bối cảnh từ TT đang chạy sang bối cảnh của TT có mức ưu tiên cao nhất. Kernel sẽ tính toán lại mức ưu tiên của TT đang thực hiện khi TT ra khỏi kernel mode để về lại user mode và kernel thực hiện đều đặn việc điều chỉnh mức ưu tiên của các TT trong *user mode* khi TT đang ở trạng thái "*ready to run*".

Khi bàn về lập biểu, các TT được phân ra làm hai lớp theo nhu cầu thời gian, là :

- *"hướng I/O"* (**I/O bound**) có nghĩa TT sử dụng nhiều tới I/O và sử dụng nhiều thời gian để đợi kết quả I/O;
- Lớp thứ hai là " hướng CPU" (**CPU-bound**), là lớp TT yêu cầu nhiều thời gian

CPU.

Tuy nhiên cũng có cách phân biệt khác theo lớp ứng dụng như:

- Lớp TT tương tác (**interactive**), tương tác thường xuyên với user, do đó chi phí nhiều thời gian cho nhận bàn phím, hay chuột. Khi có đầu vào TT phảI thức dậy thật nhanh, thời gian trễ trung bình là 50 đến 150 ms, nếu quá chận, user co hệ có vấn đề; Các ứng dụng như soạn văn bản, shell, đồ hoạ (táI tạo hình ảnh) thuộc lớp này;
- Lớp TT xử li lô (**batch**), không cần có sự can thiệp của user, và thường chạy nền (background). Do không cần phảI có đáp ứng nhanh, nên thường không được xếp ưu tiên cao, Ví dụ loại này là các trình dịch (compiler) hay cơ cấu tìm dữ liệu (database search engine), các tính toán khoa học;
- TT thời gian thực (**real time**), có yêu cầu rất khắc khe, không bao giừi bị cản trở bởi các TT mức ưu tiên thấp, cần đáp ứng thời gian thật nhanh, và quan trọng hơn cả là thời gian đáp ứng chỉ được thay đổi ở mức tối thiểu. Các ứng dụng như video, âm thanh, điều khiển robot, thu nhặt số liệu vật lí thuộc lớp này.

Các chương trình thời gian thực thường được nhận biết ẩn trong thuật toán lập biểu, chứ không có cách nào phân biệt giữa ứng dụng tương tác và ứng dụng xử lí lô. Để có đáp ứng tốt về thời gian cho các ứng dụng tương tác, Unix thiên về hướng I/O.

Các TT trong Unix có đặc thù *chen ngang* (preemptive), nên khi TT đi vào trạng thái "sẳn sàng chạy", kernel sẽ kiểm tra nếu số ưu tiên của nó lớn hơn của TT đang chạy, thì TT đang chạy sẽ bị ngắt (bị chen ngang_preempted), scheduler sẽ chọn TT khác trong các TT như trên cho chạy. TT bị chen ngang không có nghĩa bị treo, chỉ đơn giản là TT không được dùng CPU, vì TT vẫn còn trong danh sách các TT chạy. Nhắc lại (như mô hình các trạng thái), TT chỉ bị chen ngang khi chạy trong user mode. (Tuy nhiên trong hệ thời gian thực_real time kernel, TT chạy trong kernel mode có thể bị chen ngang sau bất kì chỉ lệnh nào, giống như trong user mode).

Một số các TT lại cần biết về thời gian, ví dụ lệnh *time, date* cho thời gian ngày tháng và giờ của ngày. Các chức năng liên quan tới thời gian sẽ hổ trợ cho các TT khi có nhu cầu cập nhật thời gian (TT cần đặt hay tìm lại các giá trị thời gian TT đã sử dụng CPU v.v ...). Đồng hồ thời gian thực sẽ ngắt CPU trong các khoảng 50 hay 100 lần/giây (PC intel là 18,2 lần/s = 54.9 ms). Mỗi lần có một ngắt như vậy gọi là một ***clock tick.***

Chương này sẽ đề cập tới các hoạt động có liên quan tới thời gian trên Unix, xem xét cách lập biểu cho TT, các hàm thao tác thời gian và ngắt đồng hồ.

1. Lập biểu cho tiến trình

Lập biểu chạy trình trong Unix phải giải quyết các vấn đề sau đây: đáp ứng thời gian nhanh cho TT, thông suốt cho các xử lí nền (*background jobs*), loại trừ một TT bị kẹt không được chạy, điều hoà mức ưu tiên của các TT có mức ưu tiên cao và các TT có mức ưu tiên thấp và các vấn đề khác nữa. Tập các qui luật sử dụng để xác định khi nào và lựa chọn một TT như thế nào để chạy, gọi là sách lược lập biểu (***scheduling policy***). Sách lược lập biểu còn đặt nền tảng trên sự sắp xếp và mức ưu tiên của các TT. Các thuật toán phức tạp được sử dụng để lấy ra được mức ưu tiên hiện tại của một TT, đó là giá trị kết hợp với TT cho biết việc TT sẽ được tiến chọn để sử dụng CPU (TT chạy). Trong khi đó mức ưu tiên của TT lại có tính động, được thuật toán điều chỉnh liên tục. Nhờ có vậy các TT bị từ chối sử dụng CPU trong thời gian dàI sẽ được đẩy lêm mức ưu tiên cao, ngược lại TT đã dùng CPU lâu sẽ chuyển xuống mức thấp. Lập biểu đồng thời cũng xem xét tới sự xếp loại TT (như nêu trên)

và đặc thù chen ngang để chọn TT chạy.

Lượng thời gian chạy TT cũng là một thông số ảnh hưởng rất lớn tới năng lực thực thi (performance) của toàn hệ thống. Sự lựa chọn độ lớn (time slice duration) sau đó có tính nhân nhượng, với luật là chọn đủ lâu nếu có thể (để giảm thiểu tiêu tốn thời gian cho việc thực hiện chuyển bối cảnh từ TT này sang TT khác), nhưng phảI đảm bảo đáp ứng nhanh cho các TT (tức các TT cho dù là lớp nào cũng có cơ hội thực hiện mau chóng). Trên Linux, các TT khác nhau có lượng thời gian khác nhau, nhưng khi TT đã hết thời gian của nó thì TT khác sẽ thay thế.

Mức ưu tiên cho một TT có hai kiểu:

- Mức ưu tiên tỉnh : người dùng có thể gán cho các TT thời gian thực và khoản xác định từ 1 đến 99, và bộ lập biểu sẽ không bao giờ thay đổi giá trị này;
- Mức ưu tiên động: cho các TT thông thường; là tổng của các lượng thời gian (base priority của TT) và tổng số tick của CPU đã trôI qua cho TT trước khi lượng thời gian cho TT kết thúc.
- Bộ lập biểu sẽ chọn TT thông thường nếu không có TT thời gian thực nào trong hàng đợi thực hiện. Có nghĩa mức ưu tiên tỉnh cho TT thời gian thực đặt cao hơn TT thông thường.

Một macro định nghĩa lương thời gian cơ sở (base time quantum) như sau:

```
#define DEF_PRIORITY      (20*Hz/100)
```

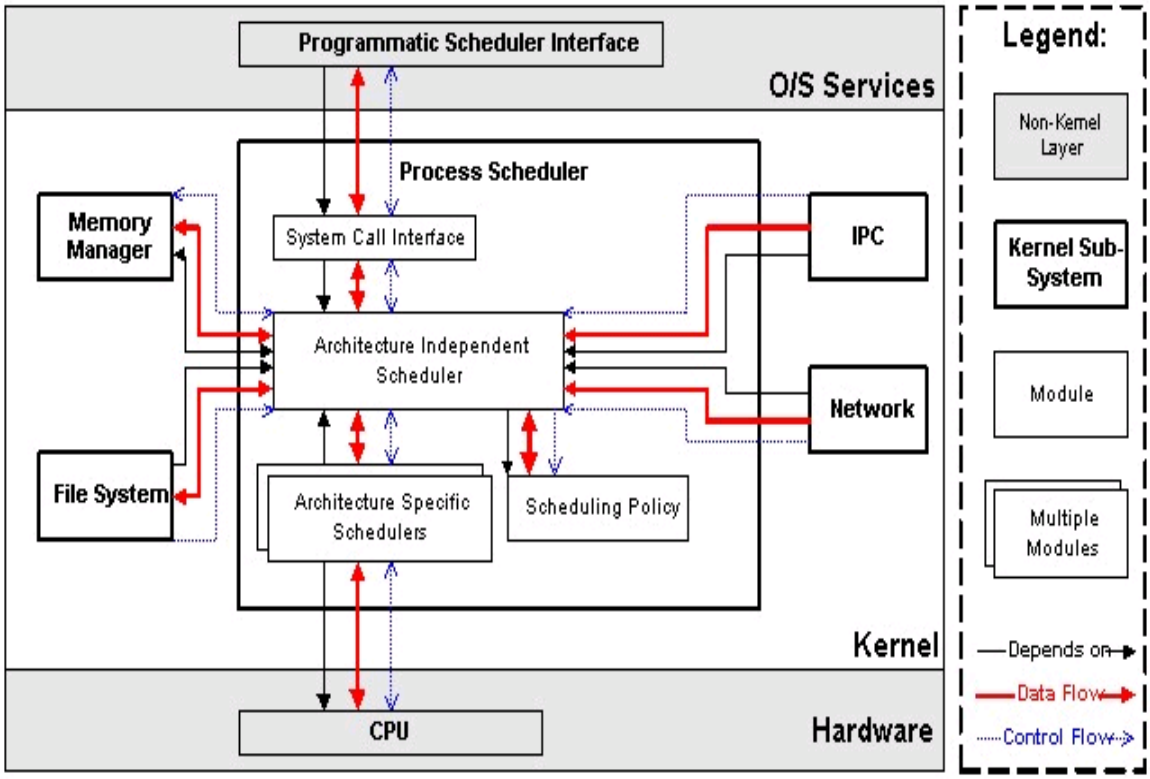
Hz do đồng hồ địng thời trong máy tính (timer, timer interrupt) xác lập, và đặt = 100 cho IBM PC, do đó DEF_PRIORITY = 20 tick, tức khoản 210 ms.
Người dùng có thể thay đổi thời lượng này bằng hàm *nice()* hay *setpriority()*.

Bộ lập biểu trong Unix thuộc lớp cơ sở của các bộ lập biểu của HĐH và làm việc theo cách ***luân chuyển quay vòng đa mức*** (***round robin with multilevel feedback)*** , có nghĩa là kernel sẽ cấp cho TT một lượng thời gian, không cho TT sử dụng quá thời gian đã cho, đưa TT vào trở lại một trong các hàng đợi ưu tiên, sau đó lại chọn TT nếu đủ điều kiện để chạy trở lại. Do đó TT có thể có nhiều lần tương tác qua "vòng luân hồi " này, trước khi nó kết thúc cuộc đời của mình. Khi kernel thực hiện *chuyển đổi và khôi phục bối cảnh*, TT lại chạy tiếp tại vị trí TT đã bị treo trước đó.

Dưới đây là ví dụ mô hình kiến trúc của module lập biểu và mối quan hệ với các module chức năng khác trong nhân hệ **Linux**. Các module nhỏ hơn bao gồm:

1. Module chính sách lập biểu (*scheduling policy*) nắm quyền lựa chọn TT nào sẽ sử dụng CPU. Nó được thiết kế để thực thi chính sách chạy TT sao cho tất cả các TT đều có cơ hội sử dụng CPU.
2. Module kiến trúc đặc trưng (*Architecture-specific*) được thiết kế để trừu tượng hoá các chi tiết của bất kì kiểu kiến trúc máy tính nào. Module này chịu trách nhiệm liên lạc với CPU để treo hay tiếp tục chạy một TT. Các thao tác ở đây thực hiện tập các mã máy để chuyển đổi bối cảnh của mỗi TT (procces *context*) khi TT bị chen ngang (*preempted*) cũng như sẽ chạy trở lại.
3. Module kiến trúc độc lập (*architecture-independent*) liên lạc với module chính sách lập biểu để xác định TT nào sẽ chạy trong thời gian tới, sau đó gọi module *architecture-specific* để phục hồi lại TT đã chọn, đồng thời module này sẽ gọi module *memory manager* để đảm bảo rằng việc khôi phục lại bộ nhớ cho TT là chuẩn xác.
4. Module ghép nối Gọi Hệ thống (*System Call Interface*) cho phép TT người dùng chỉ

có thể truy nhập tới các tài nguyên mà nhân HĐH cho phép. Điều này sẽ hạn chế sự phụ thuộc của TT người dùng vào các giao diện đã định nghĩa chặt chẽ và ít khi sửa đổi bất chấp các sửa đổi ở các module khác của nhân HĐH. Như vậy TT người dùng (hay chương trình mã nguồn C) có thể chạy trên các nền tảng khác nhau (tính *portable*).



Hệ thống con Lập biểu tiến trình

Với các module trên, bộ lập biểu sẽ thực hiện các thao tác:

- nhận các ngắt, chuyển các ngắt đến các module tương ứng của kernel để xử lí;
- gởi signal tới cho các TT của user;
- quản lí bộ định thời gian (timer) phần cứng;
- xác định TT nào sẽ chạy tiếp theo;
- thu dọn tài nguyên khi một TT đã kết thúc hoàn hảo;
- Cho phép một TT nhân bản khi gọi fork();

1.1 Thuật toán lập biểu cho TT

Dưới tác động của ngắt timer, thuật toán sẽ hoạt động

input: none
output: none

```
{
    while (no process picked to execute)
    {
        for (every process on run queue)
```

```
                pick higest priority process that is loaded in memory;
            if (no process eligible to execute)
                idle the machine;
                /* interrupt takes machine out of idle state*/
        }
        remove chosen process from run queue;
        switch context to that of chosen process, resume its execution;
}
```

Ta nhận thấy các TT có xu hướng được chọn để chạy phải ở các trạng thái "*ready to run in memory*" và "*preempted*" như đã đề cập trước đây, và TT được chọn là TT có mức ưu tiên cao nhất. Còn các TT ở các trạng thái khác đều không có ý nghĩa để lựa chọn. Nếu có nhiều TT cùng có mức ưu tiên như nhau, kernel sẽ chọn TT đã có thời gian đợi lâu nhất và tuân thủ theo sách lược luân chuyển (*round robin)*. Nếu không có TT nào đủ điều kiện chạy, hệ sẽ nghỉ cho tới khi có ngắt tiếp theo xuất hiện xảy ra, nhiều nhất là sau một *tick* của đồng hồ. Sau khi thao tác ngắt này, kernel lại thử sắp xếp cho một TT chạy.

Linux 2.4.7-10 (Version 7.2), Xem trong mã nguồn.

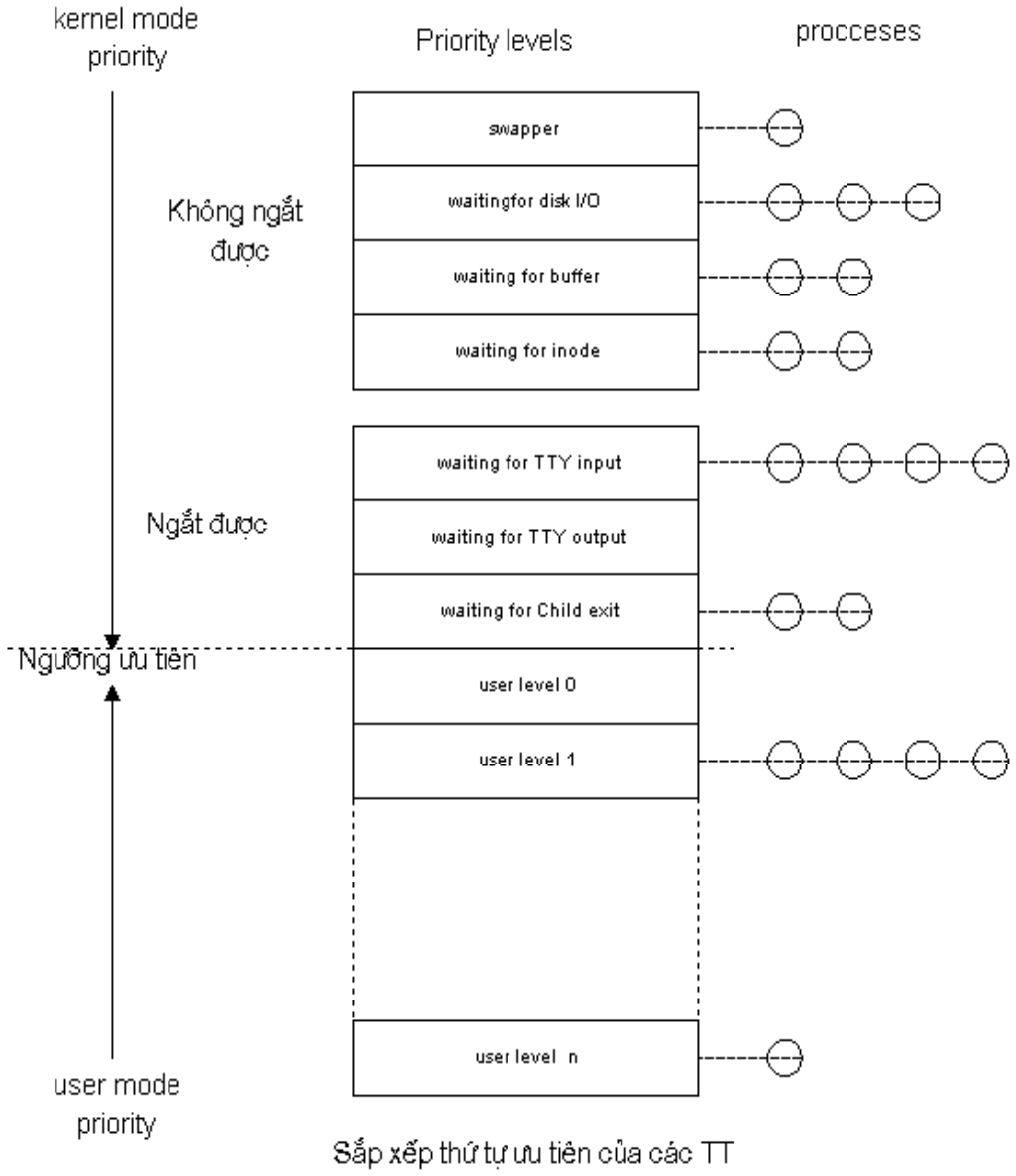
```
/*
 * /usr/linux-x.y.x/kernel/sched.c
 *
 *  Kernel scheduler and related syscalls
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 *
 *  1996-12-23  Modified by Dave Grothe to fix bugs in semaphores and
 *              make semaphores SMP safe
 *  1998-11-19     Implemented schedule_timeout() and related stuff
 *                 by Andrea Arcangeli
 *  1998-12-28  Implemented better SMP scheduling by Ingo Molnar
 */
   /* 'sched.c' is the main kernel file. It contains scheduling *primitives
    *(sleep_on, wakeup, schedule etc) as well as a number of simple system
  * call functions (type getpid()), which just extract a field from
  * current-task
  */
```

2 Các thông số lập biểu

Mỗi đầu vào cho từng TT trong *proces table* có một trường ghi nhận mức ưu tiên để lập biểu chạy TT. Mức ưu tiên của một TT trong user_mode là hàm số của việc TT hiện sử dụng CPU. Các TT vừa sử dụng CPU xong thì xếp loại có mức ưu tiên thấp hơn. Khoảng xác định mức ưu tiên có thể phân ra làm hai lớp: *user_priority* và *kernel_priority*. Mỗi lớp có một vài giá trị, mỗi mức ưu tiên có một hàng đợi cho các TT kết hợp với hàng đó. Các TT với mức ưu tiên của người dùng (user_priority) dành quyền ưu tiên khi từ kernel mode trở về user mode, còn các TT với mức ưu tiên trong kernel (kernel_priority) dành được ưu tiên ẩn có trong thuật toán ngủ (*sleep*). Mức user_priority nằm dưới một giá trị ngưỡng, còn mức

kernel_priority thì trên giá trị ngưỡng đó và còn được chia nhỏ như sau: TT với mức kernel_priority thấp sẽ được đánh thức (wake up) trên cơ sở nhận được tín hiệu (signal), còn mức kernel_priority cao hơn thì tiếp tục ngủ.

Hình dưới cho thấy ngưỡng ưu tiên giữa user mode và kernel mode, ranh giới ở đây là "*waiting for Child exit*" và "*user level 0*". Các ưu tiên có tên "swapper", "waiting for disk I/O" . . . với các TT đi cùng 1, 3, 2, với mức ưu tiên đó được thể hiện trên hình. Các mức user priority được phân biệt là "user level 0", "user level 1" … với mức 0 là cao nhất.



Sắp xếp thứ tự ưu tiên của các TT

Kernel tính mức ưu tiên của TT trong các trạng thái xác định như sau: Gán mức ưu tiên cho TT đang đi vào trạng thái ngủ tương ứng với một giá trị cố định liên quan tới lí do mà TT đi ngủ. Sự ưu tiên không phụ thuộc vào các đặc tính thời gian chạy TT , thay vì, đó là một hằng giá trị đã mã hoá cứng cho mỗi GHT sleep() tương ứng với lí do làm TT ngủ. Các TT đi ngủ khi chạy các thuật toán mức thấp và không trở lại hoạt động trong thời gian càng

lâu càng có xu hướng tạo nên sự tắc nghẽn hệ thống. Vì thế các TT loại này nhận mức ưu tiên cao hơn các TT ít có khả năng gây tắc nghẽn. Ví dụ, các TT *ngủ* và *đợi* hoàn tất truy nhập đĩa (disk I/O) có mức ưu tiên cao hơn là các TT đợi để có được buffer (vì lí do nào đó). Đó là vì các TT I/O này đã có buffer, và một khi thức dậy chúng có đủ điều kiện để xử lí (data từ đĩa) và giải phóng buffer cũng như, có thể cả các tài nguyên khác. Hơn nữa nếu càng nhiều tài nguyên mà TT sẽ giải phóng (trả lại cho hệ thống), thì càng tăng thêm cơ hội tốt cho các TT đang đợi tài nguyên, làm giảm đi xu thế hệ bị nghẻn, kernel càng ít phải thực hiện chuyển bối cảnh của các TT, đáp ứng thời gian cho TT nhanh hơn, hệ thống thông suốt hơn. Còn có trường hợp là một TT đang đợi có buffer mà buffer đó lại đang sử dụng bởi TT khác đang đợi hoàn tất truy nhập đĩa, cả 2 TT đều đang ngủ, với lí do: cần buffer, còn TT kia chưa xong việc. Khi I/O hoàn tất cả 2 thức dậy vì chúng cùng trông đợi vào 1 địa chỉ (buffer). Nếu TT đợi buffer chạy được trước thì TT này sẽ lại đi ngủ cho tới khi TT kia trả lại buffer. Vì vậy mức ưu tiên của nó sẽ thấp hơn.

Kernel đồng thời điều chỉnh ức ưu tiên của một TT khi TT chuyển từ kernel mode về user mode. Một TT đã vào trạng thái ngủ trước đó, đang thay đổi mức ưu tiên của nó vào mức ưu tiên trong chế độ kernel thì TT phải hạ thấp mức ưu tiên của nó trong chế độ user khi trở về user mode. Kernel làm vậy là để đảm bảo sự công bằng cho tất cả các TT khi các TT có nhu cầu các nguồn tài nguyên có giá trị của hệ thống.

. Bộ xử lí đồng hồ (clock handler) điều chỉnh mức ưu tiên của tất cả các TT trong user mode đều đặn sau 1 giây (trên Sytem V) và khiến kernel phải đi qua thuật toán schedule và như vậy sẽ ngăn chặn một TT độc chiếm CPU.

Clock có thể ngắt một TT vài lần trong khoản thời gian cấp cho nó. Và mỗi lần như vậy clock handler sẽ tăng thêm 1 vào giá trị của trường thời gian của TT đó trong *process table* để ghi nhận thời gian TT đã sử dụng CPU và điều chỉnh việc sử dụng CPU theo hàm số sau:

decay(CPU) = CPU/2

Khi tính toán lại việc sử dụng CPU, clock handler cũng tính lại mức ưu tiên của mỗi TT đang ở trạng thái "đã bị chen ngang nhưng sẵn sàng chạy" (S3, S7) và thực hiện theo công thức:

priority = ("recent CPU usage" / 2) + (base level user priority)

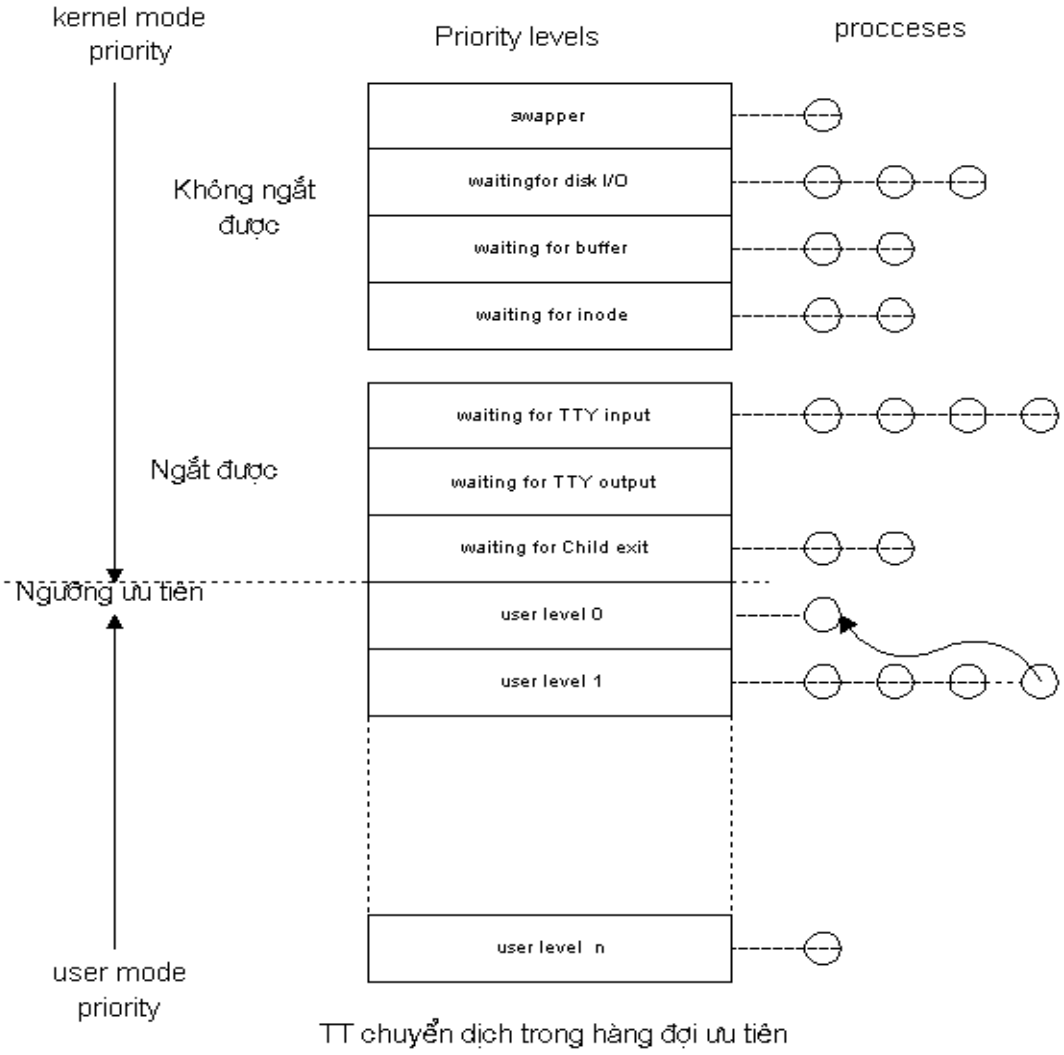
Trong đó *base level user priority* là ngưỡng ưu tiên đã nói trên. ở đây giá trị số càng thấp tương ứng với mức ưu tiên càng cao. Hai công thức trên cho thấy mối liên hệ giữa việc sử dụng CPU và mức ưu tiên của một TT và rằng khi tỉ lệ suy giảm dùng CPU (decay(CPU)) thấp hơn, TT càng lâu đạt được mức cơ bản. Hậu quả các TT ở trạng thái "*ready to run*" sẽ có xu hướng dành được nhiều mức ưu tiên hơn.

Tác động của việc tính lại mức ưu tiên mỗi lần trong 1 giây là ở chỗ TT với user level priority sẽ chuyển dịch giữa các hàng đợi ưu tiên (hình dưới).

Trong hình, một TT chuyển từ hàng đợi cho mức ưu tiên user_level 1 lên user_level 0. Trong các hệ thực tế , có nhiều TT chuyển sang các hàng đợi của các mức, nhưng hình vẽ chỉ biểu thị 1 TT. Kernel không thay đổi mức ưu tiên của TT trong kernel mode, cũng không cho phép các TT có mức ưu tiên trong user mode vượt ngưỡng sang mức ưu tiên trong kernel

mode. TT chỉ có thể thay đổi khi phát sinh GHT để đi vào trạng thái ngủ (như đã nói trên).

Kernel tính toán lại mức ưu tiên của tất cả các TT hoạt động mỗi lần / giây, nhưng khoảng thời gian có thể thay đổi ít nhiều. Nếu ngắt clock đến vào lúc kernel đã và đang thực hiện một miền mã nhạy cảm (critical codes: mức xử lí đã nâng cao hơn nhưng chưa đủ để bỏ qua ngắt clock), kernel sẽ không tính lại mức ưu tiên , thay vì kernel sẽ ghi nhận và sẽ thực hiện ở ngắt clock ngay tiếp theo khi mà mức thực hiện xử lí trước đó đủ thấp. Việc tính lại mức ưu tiên định kì đảm bảo chiến lược luân chuyển việc thực hiện các TT trong user mode. Kernel phản ứng tự nhiên đối với các yêu cầu tương tác, chẳng hạn khi một TT có tỉ số thời gian nghỉ dùng CPU cao, thì mức ưu tiên của TT sẽ nâng lên khi TT sẵn sàng chạy. Khoảng thời gian của cơ chế lập biểu có thể thay đổi từ 0 giây đến 1 giây tuy thuộc vào tải (nhiều hay ít TT hoạt động) của hệ thống.



TT chuyển dịch trong hàng đợi ưu tiên

1.3 Ví dụ việc lập biểu

Giả sử trên System V có 3 TT: A, B và C, hoạt động với các tình huống như sau: Chúng đồng thời được tạo ra với mức ưu tiên khởi đầu là 60, mức ưu tiên cao nhất của

user-level là 60, đồng hồ ngắt hệ thống 60 lần/giây, các TT không phát sinh ra GHT, không có TT nào khác ở trạng thái "ready to run".

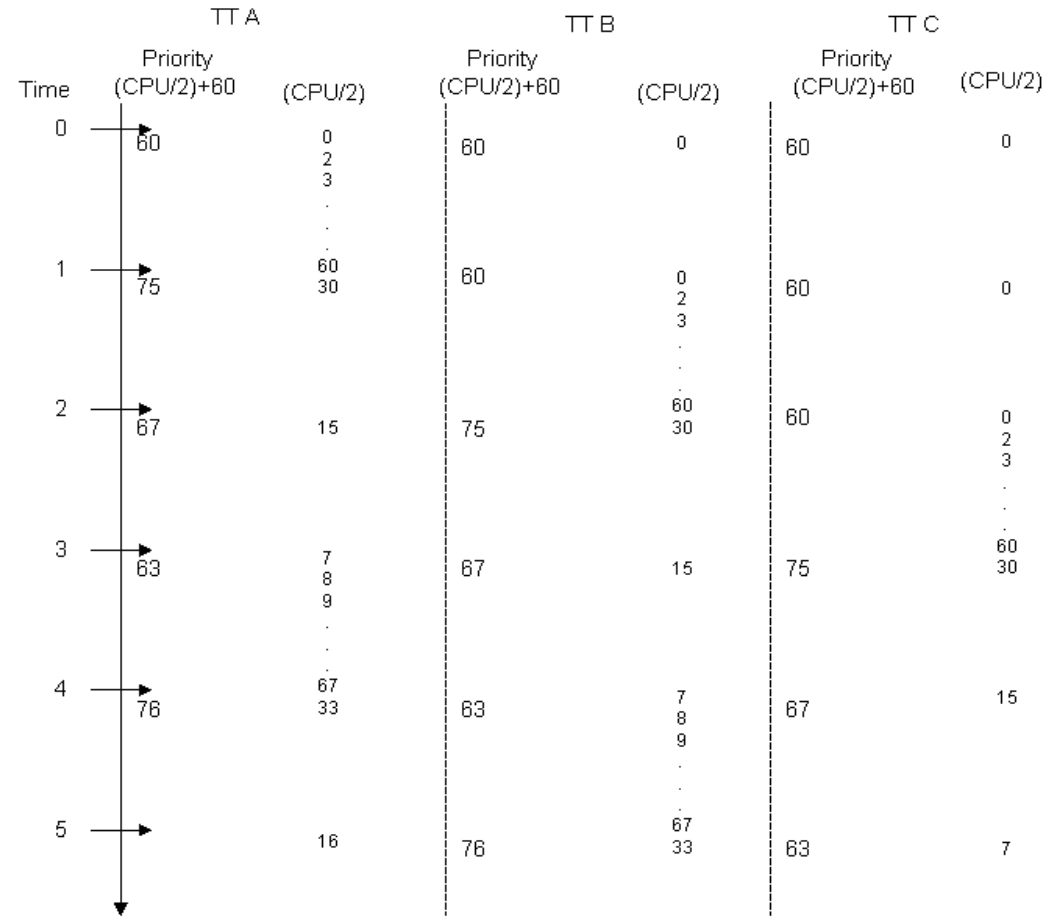
Kernel tính giá trị suy giảm dùng CPU:

CPU = decay(CPU) = CPU/2;

và mứ ưu tiên của TT:

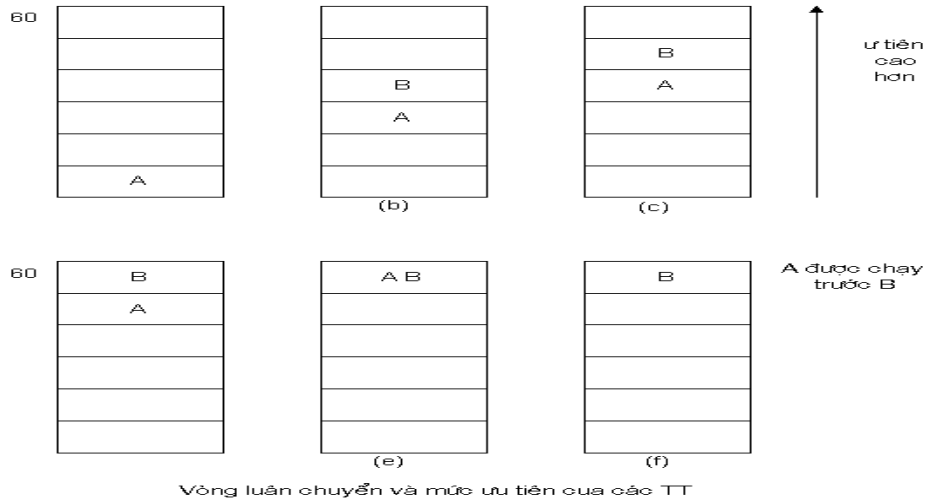
priority = (CPU/2) + 60;

Giả sử A chạy đầu tiên và A bắt đầu chạy ở đầu của khoảng thời gian của nó, A chạy trong 1 giây: trong khoảng 1 giây này hệ bị ngắt 60 lần và xử lí ngắt tăng số sử dụng CPU lên tới 60. Kernel chuyển bối cảnh sau 1 giây và sau khi thực hiện tính toán mức ưu tiên cho tất cả các TT, chọn B chạy. Sự việc cũng diễn ra như trên … Quá trình lặp lại và các TT lần lượt được chạy.



Lập biểu chạy các TT

Hãy quan sát các TT trong hình tiếp theo:



Vòng luân chuyển và mức ưu tiên của các TT

Hệ có TT A, (a), và sau đó có TT khác là B, (b). Kernel có thể chen ngang cho A, đặt A vào "ready to run" sau khi A nhận được lượng thời gian chạy kế tiếp trên CPU, và mức ưu tiên ở user-level có thể thấp (a). Thời gian tiến triển, B vào "ready to run", và mức ưu tiên ở user-level của B cao hơn A (b). Nếu kernel không lập biểu cho A và B (mà cho một TT khác nào đó), thì cả A và B có thể sẽ cùng mức cho dù B đã vào mức này trước vì mức khởi đầu của B gần với mức nói tới hơn (c) và (d). Tuy nhiên kernel có thể chọn A trước B, vì A đã ở trong trạng thái "ready to run" đã lâu hơn (e). Điều này gọi là quyết định ai khi mà các TT có mức ưu tiên ở user-level như nhau (tie-breaker). Nhắc lại là kernel lập biểu cho TT vào lúc kết thúc của chuyển bối cảnh: TT phải chuyển bối cảnh khi TT đi ngủ hoặc khi TT thực hiện *exit* và TT có cơ hội chuyển bối cảnh khi từ kernel mode về lại user mode. Một TT có mức ưu tiên cao trong user mode không thể chạy được nếu có TT có mức ưu tiên cao hơn đang ở kernel mode. Xử lí clock cho biết TT đã dùng hết thời gian của nó, và có nhuều TT đã thay đổi mức ưu tiên, nên kernel thực hiện chuyển bối cảnh để tái lập biểu cho các TT.

1.4 Kiểm soát mức ưu tiên

TT có thay đổi việc lập biểu của nó bằng GHT *nice()*: *nice(value);* trong đó *value* sẽ cộng vào giá trị lập biểu:

priority = ("recent CPU usage"/constant) + (base priority) + (nice value)

GHT *nice()* tăng hay giảm trường *nice* của mỗi TT trong *process table* bằng giá trị của thông số *value*, và chỉ có superuser có thể cho *giá trị nice* để tăng priority của TT lên cũng

như giá trị dưới một ngưỡng nhất định. User phát sinh ra nice() để giảm mức ưu tiên của mình khi thực hiện các tính toán đòi hỏi nhiều sự ưu tiên thì user đó được ưu tiên hơn các users khác (giá trị nice càng cao, giá trị priority càng cao, mức ưu tiên càng thấp). Các TT con sau fork() thừa kế giá trị nice từ TT bố, và GHT nice() hoạt động chỉ cho các TT đang chạy, và một TT không thể đặt lại (reset) giá trị nice của TT khác và đIều đó nói lên rằng nếu adminitrator muốn giảm mức ưu tiên của các TT do dùng qúa nhiều thời gian máy (CPU), thì con đường cần làm là giết (kill) các TT đó đi.

1.5 Xử lí thời gian thực

Xử lí thời gian thực là việc áp dụng khả năng mang lại đáp ứng tức thì cho những sự kiện bên ngoài riêng biệt và lập biểu các TT riêng biệt để chạy trong một giới hạn thời gian nhất định sau khi sự kiện xuất hiện. Thuật toán lập biểu đã đề cập được thiết kế cho môi trường phân chia thời gian và không thích hợp cho môi trường thời gian thực, vì không thể bảo đảm rằng kernel có thể lập biểu cho một TT nhất định trong một khoảng thời gian giới hạn cố định. Một cản trở khác là kernel không thể chen ngang một TT : kernel không thể lập biểu một TT thời gian thực trong user mode nếu đồng thời đang có TT khác đang thực hiện trong kernel mode. Để hỗ trợ thời gian thực, cần có các sửa đổi: người lập trình cần đưa các TT thời gian thực vào kernel để có được đáp ứng thời gian thực. Giải pháp thực sự là để cho phép các TT thời gian thực kết thúc một cách động, và khả năng thông báo cho kernel về những ràng buộc của TT thời gian thực. Cho tới nay các hệ Unix chuẩn không có khả năng này, mà OS thời gian thực là loại OS tương đối khác biệt.

2. Các GHT dùng với thời gian

Các GHT liên quan tới thời gian bao gồm:

1.stime(): cho phép superuser đặt các giá trị vào các biến tổng thể đẻ có thời gian hiện tại:

stime(pvalue);

trong đó *pvalue* tro tới một số nguyên dài (long integer) cho thời gian tíng bằng giây tính từ nửa đêm (00:00:00), GMT. Xử lí ngắt đồng hồ sẽ tăng biến này bằng +1 một giây một lần.

2. time(): hiển thi lại thời gian :

time(tloc);

tloc trỏ vào vị trí của TT user dành cho gía trị trả lại: time() tìm lại thời gian tích lũy trong user mode và kernel mode của TT gọi đã sử dụng và thời gian tích luỹ mà tất cả các TT con đang ở trạng thái zombie đã thực hiện trong user mode và kernel mode. Cú pháp để gọi như sau:

times(tbuffer)
*struct tms *tbuffer;*

ở đây cấu trúc *tms* chứa thời gian cần tìm và được định nghĩa bởi:

```
struct tms {
        /* time_t là cất trúc dữ liệu cho time*/
    time_t tms_utime;    /*user time of process*/
    time_t tms_stime;    /*kernel time of process*/
    time_t tms_cutime;   /*user time off children*/
    time_t tms_cstime;   /*kernel time off children*/
    };
```

times trả lại thời gian đã trôi qua, thường là từ lúc boot hệ. Thực tế chung cho tất cả các GHT

liên quan tới thời gian là sự trông cậy vào đồng hồ hệ thống (system clock): kernel thao tác các bộ đếm thời gian khác nhau khi xử lí ngắt đồng hồ và khởi động các hành động thích hợp.

3. Đồng hồ tạo ngắt

Clock hay còn gọi là **timer** là hết sức cần trong các hệ phân chia thời gian. Có hai kiểu clock được tạo ra trên hệ: loại lập trình được bao gồm một xung ngắn (**one short**) được tạo ra bằng cách đếm lùi một giá trị khi có mỗi xung của thạch anh (crystal) trong máy, cho tới khi hết số đếm thì phát sinh ra xung hẹp (one short), kích hoạt một ngắt cứng. Sau đó dừng làm việc cho tới khi phần mềm khởi động lại. Kiểu thứ hai là tạo ra một xung vuông đều đặn (**square wave**): sau khi bộ đếm về 0, ngắt được tạo ra, bộ đếm nạp lại giá trị và tự động nhắc lại quá trình đếm lùi và quá trình đó là không đổi. Ngắt tuần hoàn như vậy gọi là **clock tick**. Giá trị của bộ đếm là lập trình được. Thuận lợi của kiểu tạo clock này là ngắt phát sinh kiểm soát được bằng phần mềm. Loại thứ hai là dùng ngay đàu ra của thạch anh, đưa và bộ đếm. Bộ đếm sẽ đếm cho tới 0 thì tạo ngắt CPU, qúa trình này thuần túy điện tư với độ chính xác rất cao.

Chức năng của bộ xử lí ngắt đồng hồ là để:

- khởi động lại đồng hồ,
- lập biểu kích hoạt các chức năng bên trong của kernel trên cơ sở các định thời gian (timer) bên trong,
- cung cấp khả năng định hình sự thực hiện cho kernel và cho các TT của user,
- thu thập các số liệu kết toán của hệ thống và của TT,
- giữ nhịp thời gian,
- gởi các tín thiệu thông báo cho các TT khi có yêu cầu,
- định kì đánh thúc TT swap,
- kiểm soát việc lập biểu TT.

Một số các thao tác (operation) được thực hiện trong mỗi ngắt đồng hồ, các thao tác khác sau một vài ticks. Xử lí ngắt đồng hồ thực hiện trên CPU với mức cao ưu tiên cao, ngăn các sự kiện khác (như ngắt từ các thiết bị ngoại vi) xảy ra trong khi nó đang chạy. Xử lí ngắt đồng hồ phải nhanh sao cho thời gian cấm các ngắt khác nhỏ nhất nếu được.

Thuật toán xử lí ngắt đồng hồ (*clock handler*) như sau:

```
Input:  none
Output:         none
{
        restart clock;          /*so that it will interrupt again*/
        if (callout table not empty)
                {
                        adjust callout time;
                        schedule callout function if time elapsed;
                }
        if (kernel profiling on)
                note program counter at time of interrupt;
        gather system statistics;
        gather system statistics per procces;
```

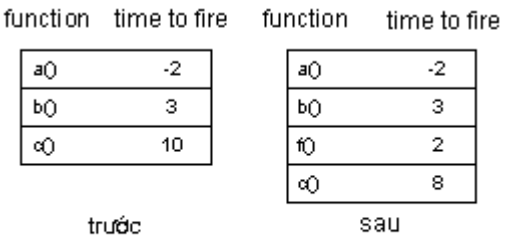
```
        adjust measure of process CPU utilization;
        if ( 1 seconde or more since last here and interrupt not in critical region of
        code)
                {
                        for (all procceses in the system)
                        {
                                adjust alarm time if active;
                                adjust measure of CPU utilization;
                                if (process to execute in user mode)
                                        adjust process priority;
                        }
                        wake up swapper process is neccesary;
                }
}
```

3.1 Khởi động lại đồng hồ

Khi đồng hồ đã ngắt hệ thống, thì hầu hết các máy (tính) đều có yêu cầu sao cho đồng hồ sẽ lại ngắt hệ sau một khoản thời gian thích hợp. Các lệnh thực hiện yêu cầu này phụ thuộc vào phần cứng, và sẽ không đề cập ở đây.

3.2 Timeout bên trong hệ thống

Khi kernel chạy, có một số các hoạt động cần kích hoạt những chức năng của kernel trên cơ sở thời gian thực, ví dụ chức năng quản lí thiết bị, các giao thức mạng và một số thao tác khác. Ví dụ : TT đặt terminal vào chế độ đọc một khối dât vào sao cho thỏa mãn yêu cầu của read() ở một khoản thời gian cố định thay vì đợi để user gõ phím xuống dòng. Kernel lưu các thông tin cần thiết trong bảng gọi là *callout*, bảng cho một hàm sẽ kích hoạt khi khoản thời gian đã hết, các thông số cấp cho hàm, lượng thời gian tính theo ticks.



Bảng callout và đầu vào mới cho chức năng f()

User không thể kiểm soát *callout table* được, các thuật toán của kernel sẽ tạo ra các đầu vào của bảng khi cần thiết. Kernel sắp xếp phân loại các đầu vào của bảng trên sự tôn trọng "thời điểm phát hỏa" (*time to fire*), độc lập với thứ tự các chức năng đã được đặt vào bảng. Theo trình tự thời gian, trường thời gian cho mỗi đầu vào lưu lại lượng thời gian để phát hỏa sau khi thành phần trước đó đã khai hỏa. Tổng thời gian để phát hoả cho một thành phần trong bảng là tổng của thời gian để phát hoả của tất cả đầu vào tính lên trên, kể cả thời gian của thành phần đó.

Hình trên cho ví dụ tức thì của *callout table* trước và sau khi thêm một đầu vào mới

trong bảng cho chức năng *f*. Khi tạo mới 1 đầu vào mới, kerenl tìm thấy vị trí chính xác (timed) cho đầu vào đó và điều chỉnh thích hợp trường thời gian của đầu vào ngay liền phía sau đó. Trong hình, kernel sắp xếp để kích hoạt chức năng *f* sau 5 ticks đồng hồ:như sau: -tạo đầu vào cho f sau đầu vào cho b với thời lượng =2 (tổng thời lượng cho b và f =5), -điều chỉnh thời lượng cho *c* =8 (*c* sẽ phát hoả sau 13 ticks). Cách thực hiện quá trình này có thể dùng danh sách liên kết cho mỗi đầu vào hay đIều chỉnh vị trí các đầu vào trong bảng khi thay đổi bảng (cách này không đắt giá nếu kernel không dùng nhiều tới bảng).
ở mỗi ngắt đồng hồ, xử lí đồng hồ sẽ kiểm tra xem có đầu vào nào trong bảng callout và nếu có, thì giảm trường thời gian của đầu vào đầu tiên. Theo cách mà kernel quản lí thời gian trong bảng, việc giảm thời lượng của đầu vào đầu tiên cũng giảm thời lượng của tất cả các đầu vào khác. Nếu lượng thời gian của đầu vào đầu tiên là bé hơn hay bằng 0 thì chức năng tương ứng sẽ đã kích hoạt. Xử lí đồng hồ không kích hoạt chức năng trực tiếp, sao cho chức năng không tình cờ cản trở ngắt đồng hồ sau này: Mức ưu tiên của bộ xử lí hiện được đặt để ngăn ngắt đồng hồ đến, tuy vậy kernel lại không thể biết là chức năng sẽ hoàn thành trong bao lâu. Nếu chức năng mà lâu hơn 1 tick đồng hồ, thì ngắt đồng hồ (và các ngắt khác) tới sẽ bị ngăn lại. Thay vì làm như vậy, xử lí đồng hồ thông thường sắp xếp chạy chức năng bằng cách tạo "ngắt mềm" (soft interrupt) hay cong gọi là ngắt được lập trình, vì nó tác động để thực hiện các lệnh máy đặc biệt. Như đã nói ngắt mền ở mức ưu tiên thấp hơn là các ngắt khác, cho nên ngắt mêm sẽ bị chặn lại cho tới khi kernel kết thúc thao tác các ngắt khác. Có một số ngắt , kể cả ngắt đồng hồ có thể xuất hiện vào giữa khoản thời gian kernel sẵn sàng để kích hoạt một chức năng trong bảng callout và thời gian ngắt mềm xuất hiện, do đó trường thời gian của đầu vào đầu tiên trong bảng có thể có giá trị âm (xem hình). Cuối cùng, khi ngắt mềm xảy ra, xử lí ngắt sẽ lại trừ các đầu mà trường thời gian của nó đã hết hạn khỏi bảng và kích hoạt chức năng tương ứng. Vì rằng trường thời gian của các đầu vào đầu tiên là 0 hay giá trị âm, xử lí ngắt đồng hồ phảI tìm đầu vào đầu tiên có trường thời gian là dương và giảm gia trị đó. Chức năng *a* có giá trị thời gian là -2, có nghĩa rằng hệ đã bắt được 2 ngắt đồng hồ sau khi *a* có cơ hội kích hoạt, kernel bỏ qua *a* và giảm thời lượng của *b*.

3.3 Làm lược sử (*profiler*)

Bằng cách theo dõi lịch sử (*profiling*), kernel có được thước đo mức độ hệ chạy trong chế độ user hết bao lâu so với chế độ kernel, cũng như từng thủ tục (routine) chạy trong chế độ kernel hết bao nhiêu thời gian. Kernel kiểm soát thời gian tương đối thực hiện các module của kernel bằng cách lấy mẩu sự hoạt động của hệ vào lúc có ngắt đồng hồ. Bộ điều khiển lược sử (*profile driver*) có danh sách các địa chỉ của kernel để lấy mẩu mà thông thường là địa chỉ của các chức năng của kernel. Nếu chức năng profile được phép hoạt động (enable), xử lí ngắt đồng hồ sẽ kích hoạt xử lí ngắt của *profile driver* và *profile driver* sẽ xác định ở thời điểm ngắt này bộ xử lí đang là chế độ user hay kernel. Nếu là chế độ user, *profile driver* sẽ tăng bộ đếm thực hiện trong chế độ user, nếu là chế độ kernel, thì tăng một bộ đếm bên trong tương ứng với bộ đếm chương trình (program counter). Các TT của user có thể đọc được các thông số của *profile driver* để lấy các giá trị đếm của kernel và thực hiện các số liệu thống kê.

Ví dụ cho các địa chỉ giả định của các chức năng của kernel và các gí trị đếm tương ứng:

Algoritms	Addres	Count
bread()	100	5
bread()	150	0
bwrite()	200	2
User()	-	2

Người dùng (user) có thể thực hiện gọi hệ thống kể có các giá trị ở mức user như sau:

profil(buff, bufsize, offset, scale);

buff: địa chỉ của một mảng trong không gian của user,
bufsize: kích thước mảng;
offset: địa chỉ (ảo) của chương trình xử lí của user;
scale: là một hệ số (factor) để ánh xạ đị chỉ ảo của user vào mảng.

3.4 Kết toán và thống kê

Khi clock ngắt hệ, hệ có thể đang chạy trong chế độ user hay kernel, hay không làm gì cả (idle: không chạy một TT nào, các TT đang ngủ đợi sự kiện hoài vọng). Kernel lưu các (giá trị) của các bộ đếm (cho biết mỗi trạng thái của CPU), điều chỉnh các gía trị này trong mỗi lần ngắt đồng hồ, ghi nhận chế độ máy (user/kernel) và TT của user sau đó có thể dùng các số liệu này để làm phân tích. Mỗi TT có 2 trường trong u_area của mình để lưu thời gian đã trôi qua trong chế độ user và kernel. Khi thao tác ngắt đồng hồ , kernel sẽ cập nhật các giá trị mới cho TT đang được thực hiện và tuỳ vào TT đang ở chế độ nào. Các TT bố tập hợp các số liệu của các TT con trong khi thực hiện *wait()* đợi TT kết thúc (*exit*).

Kernel tăng biến thời gian (*timer*) ở mỗi ngắt đồng hồ để lưu lại thời gian kể từ khi hệ khởi động. Giá trị của *timer* sẽ là đầu ra của chức năng *time()* và để tính tổng thời gian (thực) thực hiện một TT. Kernel lưu thời gian bắt đầu của TT trong u_area khi TT được tạo ra bằng *fork()* và trừ đi giá trị thời gian lúc TT kết thúc.

Chức năng *stime()* sẽ cập nhật mỗi giây một lần để tính thời biểu lịch.

4. Tóm tắt về TT và sự lập biểu chạy TT

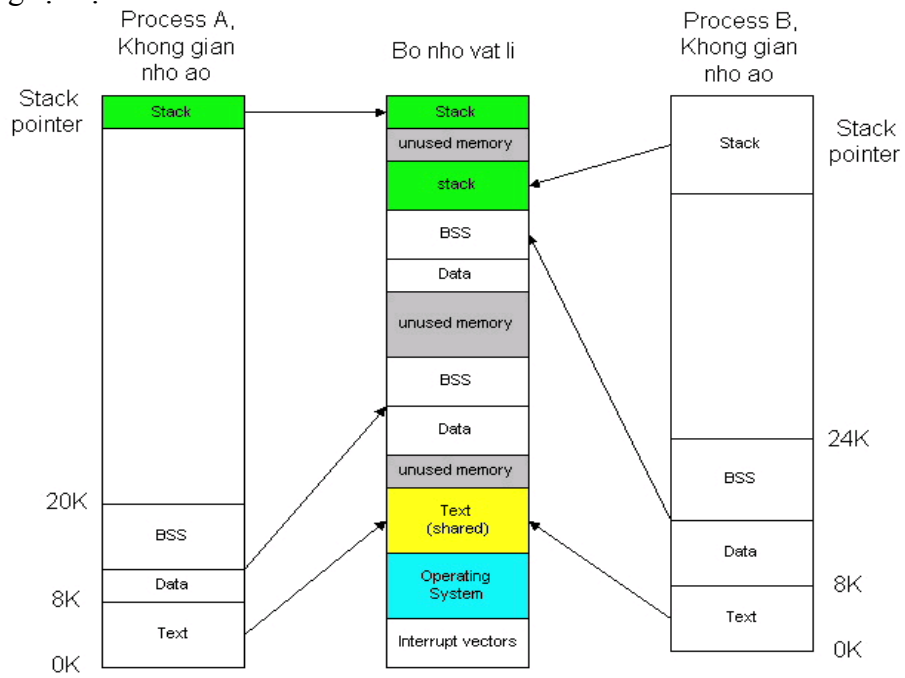
Trong Unix, chỉ có TT là các thực thể hoạt động. Mỗi TT chạy một chương trình và có một luồng (*thread*) kiểm soát. Nói một cách khác, TT có một bộ đếm chương trình, các thanh ghi của CPU và không gian địa chỉ với các thành phần: text (mã chương trình), data (user data và kernel data), stack (user stack và kernel stack). Các thành phần đó tạo ra *bối cảnh* (context) của TT.

Mỗi TT trong Unix có không gian địa chỉ gồm ba segment: text, data stack. Text segment chứa các lệnh máy tạo thành chương trình (tệp thực thi) với trạng tháI chỉ đọc (read only). Data segment chứa data bao gồm các biến của chương trình, xâu, trường (đa chiều), và các dữ liệu khác. Data có hai phần: data đã được khởi động (các biến, hằng cần thiết khi chương trình bắt đầu chạy) và dữ liệu chưa khởi động (***BSS: Block Started by Symbols***). Unix có khả năng chia sẻ mã thực thi khi có nhiều người dùng cùng chạy một trình ứng dụng.

Ví dụ như trên hình, TT A và TT B cùng chia sẻ text segment, việc ánh xạ từ không gian nhớ ảo của từng TT đến text dùng chung do HĐH và phần cứng thực hiện.

Là hệ đa trình, nên có vô số các TT độc lập chạy đồng thời, mỗi user có thể có vài TT hoạt động, do vậy con số có thể lên đến hàng trăm hay hàng ngàn. Ngay cả khi user chưa thực hiện gì thì trên hệ cũng đã có các TT hoạt động , gọi là *daemon*, được khởi động tự động khi boot máy. Cron daemon là một ví dụ, nó được đánh thức cứ sau 1 phút để kiểm tra xem có công việc gì phảI làm, nếu không thì đi ngũ, daemon khác dùng để định thời gian lập biểu cho các TT chạy, v.v…

TT được tạo ra bởi GHT **fork(),** fork() tạo ra một bản sao chính xác của TT gốc. TT được tạo ra là TT con và TT thực hiện GHT fork() là TT bố. *Hai TT có không gian bộ nhớ riêng biệt*, như vậy nếu TT bố thay đổi các biến của nó thì TT con không thể biết được, và ngược lại.



Các tệp đã mở là chia sẽ giữa hai TT, có nghĩa nếu có tệp mở trước khi TT bố fork, thì tệp tiếp tục mở cho cả hai sau đó. Mọi thay đổi trên tệp thực hiện bởi TT nào đều được TT kia nhận biết, và các TT khác không huyết thống tham gia mở tệp này cũng đều (phải) biết đến các thay đổi đó.

Sự giống hệt nhau (identical) của ảnh bộ nhớ, các biến, các thanh ghi, cũng như các các giá trị khác ở TT bố và TT con, dẫn đến một khó khăn: Các TT làm sao biết được, TT nào chạy mã của TT bố và TT nào chạy mã TT con. Điều bí mật là ở chổ fork() trả lại giá trị 0 cho TT con, và giá trị khác 0, còn gọi là PID cho TT bố. Cả hai TT do đó thường kiểm tra giá trị trả về và thực thi hành động thích hợp, như ví dụ sau:

```
.
.
.
pid = fork();    /* Nếu GHT fork() thành công, pid > 0 trong TT bố*/
if (pid < 0)
```

```
        { /* fork () không được: thiếu bộ nhớ, Process Table hết đầu vào…*/
        .
        .               /* Mã xử lí…*/
        .
         exit(n);
}
else
        if( pid > 0)
        { /* mã TT bố ở đây*/
          .....
        }
        else            /*tương đương với if (pid == 0)*/
        { /* mã TT con ở đây*/
        .
        .
        .
        }
        .
        .
        .
```

TT được nhận biết bằng PID, một TT có thể biết PID của nó bằng GHT getpid(). PID được dùng trong nhiều hoàn cảnh, ví dụ khi TT con kết thúc, TT bố sẽ nhận được PID của TT con vừa kêt thúc (TT bố có thể có nhiều TT con). Các TT con có thể tạo ra các TT con khác, TT bố sẽ có TT cháu, TT chắt, v.v… Điều đó dẫn đến sự hình thành cây TT (tree of process).

Các TT trong Unix liên lạc với nhau bằng cơ chế trao đổi thông điệp (message) và qua kênh truyền data gọi là ống dẫn (**pipe**). Nguyên tắc làm việc của pipe là một TT ghi một xâu các bytes vào pipe và TT kia đọc các bytes đó và nó chỉ được đọc khi pipe không rỗng, kernel sẽ treo (không cho chạy tiếp) TT ghi nếu pipe đã đầy. Quá trình này gọi là sự đồng bộ khi thực hiện pipe.

Các TT còn liên lạc qua ngắt mềm mà thực chất là TT sẽ gởi một tín hiệu (**signal**) cho một TT khác. Để xử lí các signal, các TT nói cho kernel biết, các TT muốn gì khi có các signal đến và sự lựa chọn là để bỏ qua signal hay đón nhận signal hay TT sẽ hũy diệt signal đó. Nếu đón nhận, TT phải xác định một thủ tục để xử lí signal, sao cho điều khiển sẽ chuyển đến thủ tục này. Quá trình sau đó giống như xử lí ngắt I/O đã biết. TT chỉ gởi signal đến được các TT thành viên cùng nhóm (**process group**) của nó : TT bố và các tiền bối tổ tông, anh chị em ruột, các con và các cháu. TT cũng có thể gởi signal tới tất cả các thành viên trong nhóm của mình bằng một GHT đơn gảin.

Signal còn được dùng cho mục đích khác, chẳng hạn như trường hợp xử lí lỗi : chia cho số 0 trong phép tính dấu phẩy động. Cacsignal mà POSIX yêu cầu bao gồm:

SIGABRT	Sent to abort a process and force a core dump
SIGALARM	The alarm clock has gone off
SIGFPE	A floating point error has occurred
SIGHUP	The phone line the process was using has been hung up

SIGILL	The process has excuted an illegal machine instruction
SIGINT	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignore)
SIGPIPE	The process has written on a pipe with no readers
SIGSEGV	The process has referenced an invalid memory ddress
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purpose
SIGUSR2	Available for application-defined purpose

Các bản Unix có các signal của mình song không thể sử dụng trên các hệ Unix khác (not portable).

Mỗi người dùng trong Unix được nhận biết bởi một số nguyên gọi là *số nhận dạng người dùng* ***uid*** (**user identification**), số này được đặt trong tệp /etc/passwd. Khi một TT được tạo, thì TT sẽ tự động lấy **uid** của người đã tạo ra nó. Với **uid = 0**, thì đó là **superuser** hay còn gọi là **root**. Root có quyền truy nhập (R/W/X) tất cả các tệp trên FS cho dù ai đã tạo ra tệp cũng như rệp được bảo vệ như thế nào. Các TT với uid = 0 có quyền tạo ra một số GHT được bảo vệ và các GHT này sẽ từ chối thực hiện đối với các user bình thường. Do có sự khác biệt về hiệu lực trên Unix giữa superuser và các người dùng khác, và để tạo điều kiện cho người dùng có một số khả năng nhất định, Unix có một hàm chức năng gán các bit đặc biệt kết hợp với các tệp thực thi, đó là **setuid bit**. bit này là một phần của các chế độ bảo vệ trên tệp. Khi một chương trình với *setuid bit = ON*, được thực hiện, thì **effective uid** (uid hiệu lực) cho TT đó sẽ là **uid** của người sở hữu tệp thực thi, thay cho **uid** của người dùng đã kích hoạt chương trình đó. Bằng cách đó, superuser cho phép các người dùng bình thường chạy nhiều chương trình trên hệ bằng chính quyền lực của mình, nhưng theo các mức độ giới hạn và kiểm soát được. Cơ chế **setuid** dùng phổ biến trên Unix để hạn chế một số các GHT có khả năng truy nhập sâu vào hệ thống va tài nguyên mà kernel quản lí.

Mỗi TT trong Unix có hai phần, phần user (level) và phần kernel (level). Phần kernel chỉ hoạt động khi có một GHT kích hoạt, nó có stack và bộ đếm chương trình riêng, và điều này rất quan trọng, vì một GHT có thể làm dừng thực hiện TT, để thực hiện một công việc khác (như đợi đọc số liệu từ đĩa). Kernel duy trì hai cấu trúc dữ liệu liên quan tới TT: bảng các TT (**process table**) và cấu trúc của người dùng (**user structure u_area**). **Process table** thường trú trong hệ thống và chứa các thông tin cần thiết cho tất cả các TT cho dù TT có hay không (swap out) trong bộ nhớ. **User structure** cũng được hoán đổi khi TT kết hợp với cấu trúc không trong bộ nhớ (để sử dụng bộ nhớ hiệu quả hơn). Các thông tin trong **process table** gồm các hạng mục sau:

1. Các thông số lập biểu (scheduling parameters): mức ưu tiên của TT, lượng thời gian đã dùng CPU. Các thông số này dùng để chọn một TT đưa vào chạy.
2. Ảnh bộ nhớ: các con trỏ vào các phân đoạn text, data, stack, các bảng các con trỏ trỏ vào địa chỉ vật lí trong bộ nhớ (nếu là cơ chế cấp phát theo phân trang_paging). Nếu text segment là chia sẽ, thì các con trỏ trỏ vào các trang chia sẽ. Khi TT không trong bộ nhớ thì các thông tin cần để tìm lại TT cũng chứa ở đây.
3. Tín hiệu: Mặt nạ signal cho biết tín hiệu nào sẽ được bỏ qua, tín hiệu nào sẽ tiếp nhận, tín hiệu nào tạm thời sẽ ngăn lại và tín hiệu nào TT đang gởi đi.
4. Các thông tin khác: trạng tháI hiện tại của TT, sự kiện đang hoàI vọng, PID

cuat TT, PID của TT bố, UID và GUID của TT.

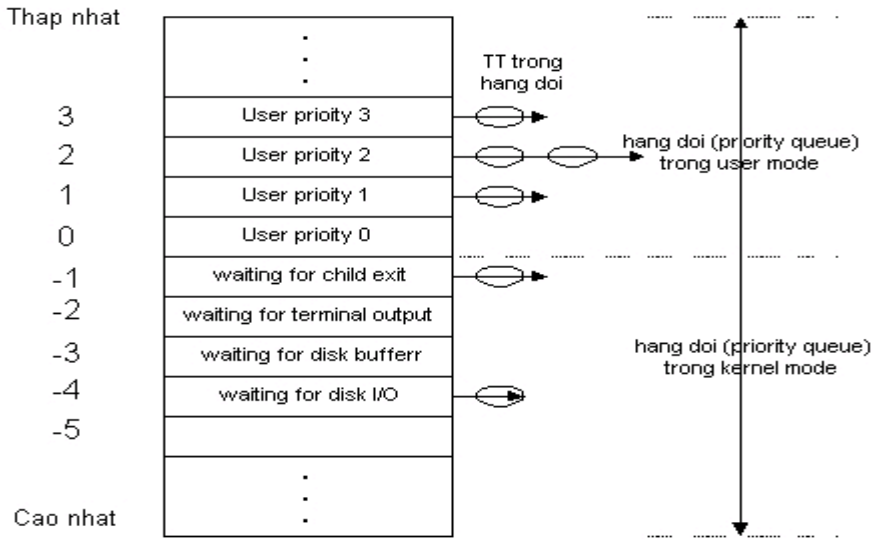
Thông tin trong **user structure u_area** bao gồm:

1. Các thanh ghi của CPU: khi TT bị ngắt, các thanh ghi TT đang dùng lưu ở đây
2. Trạng thái của GHT. Khi một GHT có phát sinh theo một mô tả tệp (file descriptor fd) thì fd là chỉ số để tìm đến con trỏ mà từ đó (qua file table) định vị inode trong incore-inode table.
3. Con trỏ chỉ đến bảng có các thông tin về thời gian CPU, thời gian hệ thống mà TT đã dùng, để làm kết toán, độ dàI tối đa của stack cho TT, tổng số trang bộ nhớ TT có thể được cấp.
4. Kernel stack: là stack cố định để phần kernel của TT sử dụng.

Là hệ phân chia thời gian, thuật toán lập biểu (**scheduler**) được thiết kế để đáp ứng với các TT tương tác. Thuật toán có hai mức: *mức thấp* sẽ chọn một TT trong các TT trong bộ nhớ sẵn sàng chạy để chạy tiếp theo. *Mức cao* của thuật toán thực hiện hoán đổi các TT giữa bộ nhớ và đĩa (swap), sao cho các TT có cơ hội trong bộ nhớ để chạy. Thuật toán mức thấp khai thác các hàng đợi khác nhau mà mỗi hàng kết hợp với một giá trị (mức) ưu tiên nhất định. Các TT thực hiện trong **user mode** có giá trị dương, còn TT thực hiện trong **kernel mode** (TT thực hiện một GHT) có giá trị âm. Các giá trị càng âm thì mức ưu tiên kết hợp với nó càng cao, giá trị dương càng lớn mức ưu tiên càng thấp. Chỉ các TT đang trong bộ nhớ và sẵn sàng chạy mới đặt trong hàng đợi vì đó là nơiI TT sẽ được chọn. Khi scheduler chạy, nó tìm các hàng đợi bắt đầu với mức ưu tiên cao nhất, và từ đó "nhặt" ra TT đầu tiên của hàng và cho TT đó chgạy. TT sẽ chạy trong một lương thời gian tối đa nhất định (100ms chẳng hạn, khoảng 5 tics với tần số 50 Hz) cho tới khi hết. Mỗi lần có một tics, bộ đếm sử dụng CPU của TT trong process table tăng lên 1. Giá trị này sẽ cộng vào mức ưu tiên của TT để chuyển TT xuống mức thấp hơn. Khi hết thời gian của mình, TT sẽ được đặt vào cuối hàng đợi và quá trình sẽ tái diễn. Chính vì vậy ta gọi thuật toán chia sẽ CPU là luân chuyển (round robin). Cứ sau mỗi giây mức ưu tiên của các TT sẽ được tính lại một lần:

mức ưu tiên mới = (thời gian đã dùng CPU)/2 + mức cơ bản + nice valule

trong đó mức cơ bản thường là 0. Tuy nhiên người dùng sẽ điều chỉnh thành số dương qua GHT nice().



Khi TT bị treo (TT thực hiện một GHT liên quan tới tài nguyên như I/O đĩa), TT sẽ bị loại ra khỏi hàng đợi. Cho tới khi sự kiện TT hoài vọng đến, nó sẽ được đặt vào hàng với một giá trị âm. Việc chọn hàng nào sẽ phụ thuộc vào sự kiện mà TT hoài vọng (I/O đĩa thường cao nhất). ẩn ý của thuật toán này là làm cho TT thoát ra khỏi kernel mode thật nhanh, trả TT về user mode (kết thúc GHT).

Chương V. Liên lạc giữa các tiến trình

(Inter Process Communication_IPC)

1. Giới thiệu

Chương trước đã đề cập các chức năng cơ bản để kiểm soát TT, từ việc tạo ra TT, đồng bộ hoạt động giữa các TT, sử dụng cơ chế tín hiệu để các TT thông báo các sự kiện. Tuy nhiên các cơ chế đó vẫn mới chỉ là cơ bản. Chương này sẽ đề cập đến các phát triển cao hơn và khả năng ứng dụng khi phát triển các ứng dụng từ đơn giản đến phức tạp.

System V IPC có các cơ chế như sau:

-*pipe* (đường ống): không tên, *FIFO:* có tên;

-*thông điệp (message)*: cho phép các TT gởi các khuôn dữ kiệu có khuôn dạng tới bất kì TT nào;

-*vùng nhớ chia sẽ (shared memory)*: các TT chia sẻ một phần không gian địa chỉ ảo của mình;

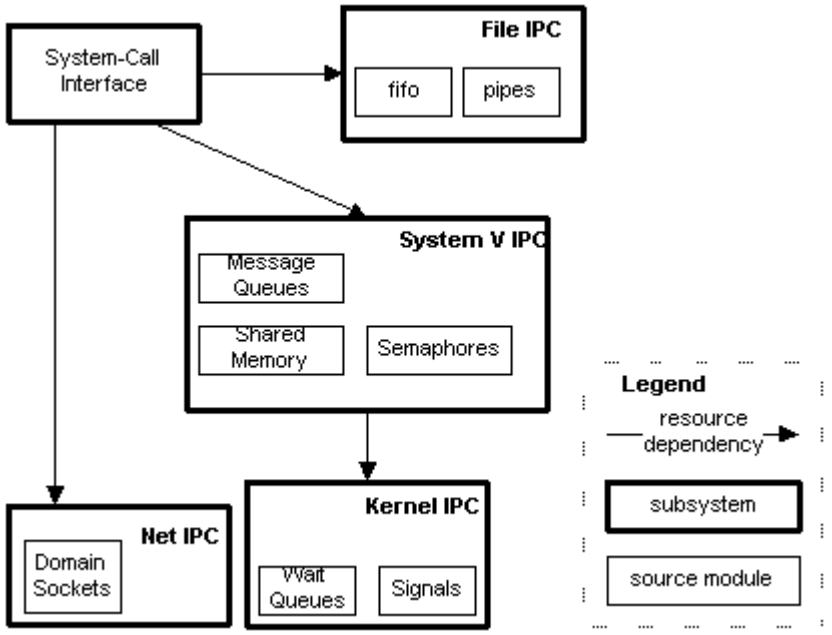
-*đánh tín hiệu (semaphore)*: các TT dùng để đồng bộ việc thực hiện.

Được sử dụng thống nhất, các cơ chế trên chia sẻ các thuộc tính như sau:

1. Mỗi cơ chế có một bảng với các đầu vào mô tả (còn gọi lá các *descriptor*) tất cả các thời điểm (instance) của cơ chế đó;
2. Mỗi đầu vào chứa một khoá số (key) mà tên của khoá do user chọn;
3. Mỗi cơ chế chứa một gọi hệ thống "get" để tạo ra một đầu vào mới hoặc để tìm lại một đầu vào đã có , và các thông số cho "get" kể cả khoá (key) và cờ (flags). Kernel tìm bảng tương ứng để tìm một đầu vào có tên xác định bằng khoá. Các TT GHT "get" với khoá IPC_PRIVATE để đảm bảo nhận được một đầu vào chưa sử dụng. Các TT có thể dựng bit IPC_CREATE ở trường chứa cờ (flag) để tạo một đầu vào mới khi đầu vào này với khoá trao cho mà chưa tồn tại, hay dựng cờ IPC_EXCL và IPC_CREAT để có thông báo lỗi một khi đầu vào với khoá trao cho đã tồn tại. GHT "get" trả lại một *mô tả* kernel lựa chọn để sử dụng trong các GHT khác.
4. Đối với mỗi cơ chế IPC, kernel dùng công thức sau đây để tìm chỉ số trong bảng cấu trúc dữ liệu của một *mô tả* nói trên*:*
   index = descriptor modulo (number of entries in table)
5. Mỗi đầu vào có cấu trúc để quản lí quyền truy nhập bao gồm user ID (UID) và group ID (GID) của TT đã tạo ra đầu vào đó (tương tự như khi thực hiện làm R/W/X trên tệp cho u(user), g(group), o(other).
6. Mỗi đầu và còn có các thông tin trạng thái, ví dụ PID của TT cuối cùng đã cập nhập mới đầu vào đó (TT đã gởi / nhận thông điệp, đã ghép vùng nhớ chia sẻ, ...) và thời gian TT đã truy nhập, cập nhật đầu vào.
7. Mỗi cơ chế có hệ thống kiểm soát để truy vấn trạng thái của một đầu vào, để đặt trạng thái, huỹ đầu vào. Khi TT truy vấn trạng tháI của một đầu vào, kernel

kiểm tra TT có được phép đọc hay không và sau đó copy dữ liệu từ đầu vào đó của bảng vào không gian địa chỉ của user. Tương tự, khi lập các thông số vào đầu vào, kernel kiểm tra UID của TT, xem có trùng hợp với user ID (hay UID của người đã tạo ra đầu vào) hay không, có phải superuser chạy TT ? Kernel sao chép dũ liệu của user vào đầu vào của bảng, lập các UID, GID, quyền R/W/X cũng như các trường khác của đầu vào đó, tuỳ thuộc vào từng kiểu cơ chế. Kernel không thay đổi trường ID của người đã tạo ra đầu vào, và GID có user đó, vì vậy người đã tạo ra đầu vào vẫn kiểm soát đầu vào như trước. User có thể huỹ đầu vào nếu đó là superuser hoặc nếu UID trùng hợp với trường UID của đầu vào đó, kernel sau đó tăng số của descriptor sao cho lần tạo đầu vào mới tiếp theo sẽ có được số descriptor khác. Chính vì vậy nếu TT truy nhập đầu vào bằng số descriptor cũ, sẽ có thông báo lỗi.

Ví dụ mô hình hệ thống IPC trên Linux:



2. Kỉ thuật liên lạc giữa các TT trong Unix System V

Phần này sẽ đề cập các kỉ thuật áp dụng cho mỗi cơ chế đã nêu và các phiên bản Unix hổ trợ , với bảng tóm tắt sau:

Kiểu IPC	SVR4	4.3 BSD	4.4+BSD
pipe (no named, half duplex) FIFO(named pipe	• •	•	• •
stream pipe (full duplex)	•	•	•

naned stream pipe	•	•	•
message queue semaphore shered memory	• • •		
éamocket stream	• •	•	•

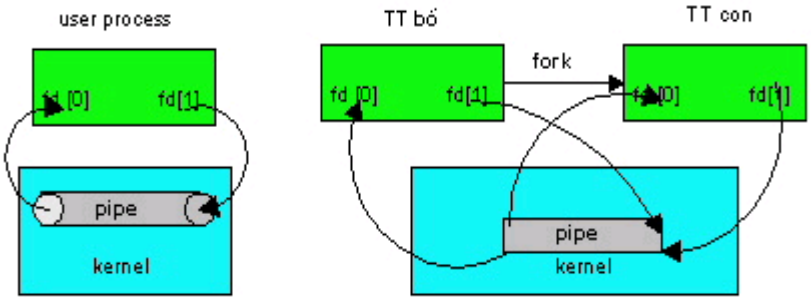
2.1.1. PIPE, nguyên tắc

Cơ chế liên lạc giữa các TT cho phép các TT bất kì liên lạc và đồng bộ việc thực hiện với nhau. Chúng ta cũng đã xem xét một vài phương thức, chẳng hạn

-pipe (bán song công halff duplex) là loại pipe vô danh (unnamed pipe) có hạn chế là *chỉ dùng với các TT thuộc con cháu của TT phát sinh ra GHT pipe, do vậy các TT không có cùng quan hệ huyết thống sẽ không thể liên lạc qua pipe được*.

-p*ipe có tên (named pipe: FIFO)* mở rộng phạm vi sử dụng, loại trừ hạn chế ở pipe vô danh;

- hạn chế chung của pipe: các TT lại không thể liên lạc với nhau qua mạng, cũng như không hổ trợ dễ dàng liên lạc đa đường dẫn cho các tập hợp khác nhau của các TT tham gia liên lạc: không thể thực hiện một quá trình dồn kênh (multiplex) trên một *named pipe* để tạo ra các kênh riêng cho các đôi TT liên lạc.



1.TT bố tạo pipe[2] trước 2. sau đó tạo TT con: pipe sau khi fork()

Sau khi fork, các mô tả tệp fd[0], và fd[1] của TT bố sẽ có trong TT con và điều gì sẽ xảy ra phụ thuộc vào hướng thông tin mong muốn. Để có chiều từ TT bố đến TT con, TT bố đóng fd[0]_read của TT bố, TT con đóng fd[1]_write của TT con. Kết quả TT con đọc thông tin từ fd[0] của TT con, TT bố ghi thông tin vào fd[1] của TT bố. Ngược lại từ TT con đến TT bố: TT bố đóng fd[1], TT con đóng fd[0].

Khi có một đầu cuối của pipe đóng, hai luật sau sẽ áp đặt:

Nếu đọc (read()) pipe mà đầu ghi (fd[1]) đã đóng sau khi tất cả dữ liệu đã đọc hết, thì read() sẽ trả lại 0, để thông báo hết tệp (EOF). Có thể nhân bản fd (mô tả pipe) cho nhiều TT có thể ghi vào pipe, tuy thông thường chỉ một đọc/một ghi.

Nếu ghi (write()) vào pipe mà đầu đọc (fd[0]) đã đóng, thì tín hiệu SIGPIPE sẽ phát ra. Nếu bỏ qua hay bắt và xử lí signal và trở về từ hàm xử lí signal, thì write() sẽ trả lại lỗi EPIPE.

Biến hằng PIPE_BUF xác định độ lớn của pipe trong kernel, cho nên nếu một TT ghi số bytes nhỏ hơn độ lớn của pipe thì sẽ không có sự cách quãng dữ liệu khi nhiều TT cùng ghi vào pipe. Ngược lại dữ liệu của các TT sẽ bị đan xen cách quãng.

Cú pháp tạo pipe vô danh:

#include <unistd.h>
int pipe (int filedes[2]);
return:=0 nếu thành công,= -1 khi không tạo được.

Ví dụ: tạo pipe để gởi dữ liệu từ TT bố đến TT con. (*pipe1.c*) và gởi dữ liệu cho nhau.

```
#include        "ourhdr.h"

int main(void)
{
        int     n, fd[2];
        pid_t   pid;
        char    line[MAXLINE];

        if (pipe(fd) < 0)                   /* tạo pipe[2] trước khi fork()*/
                err_sys("pipe error");

        if ( (pid = fork()) < 0) /* 1. kiểm tra xem fork() OK !*/
                err_sys("fork error");

        else if (pid > 0) {        /* 2. Cho code của TT bố: TT bố (parent) đóng fd[0] của nó  */
                close(fd[0]);
                write(fd[1], "hello world\n", 12);      /* sau đố ghi một dữ liệu vào pipe*/

                        }
            else {        /* 3. (pid==0) Cho code TT con:, TT con (child) đống fd[1] của nó  */
                close(fd[1]);
                n = read(fd[0], line, MAXLINE); /* và đọc dữ liệu từ pipe*/
                write(STDOUT_FILENO, line, n); /*và  hiển thi nội dung ra stdout*/
                }

        exit(0);
}
```

Ví dụ: TT bố đọc dữ liệu ở stdin và chuyển vào pipe, TT con đọc dữ liệu từ pipe và xử lí. Hoạt động của pipe: một chiều: TT bố -> TT con.

```
#include <stdio.h>
#include <unistd.h>
/* Hàm xử lí của TT con*/
void do_child(int data_pipe[]) {
   int c;
```

```
   int rc;

   close(data_pipe[1]);

   while ((rc = read(data_pipe[0], &c, 1)) > 0) {
            putchar(c);
   }

   exit(0);
}

/* Hàm xử lí của TT cha*/

void do_parent(int data_pipe[])
{
   int c;
   int rc;

   close(data_pipe[0]);

   while ((c = getchar()) > 0) {
      rc = write(data_pipe[1], &c, 1);
           if (rc == -1) {
              perror("Parent: pipe write error");
              close(data_pipe[1]);
              exit(1);
      }
   }


   close(data_pipe[1]);
   exit(0);
}

int main()
{
   int data_pipe[2];
   int pid;       int rc;
   rc = pipe(data_pipe);
   if (rc == -1) {
            perror("pipe create error ");
            exit(1);
   }

   pid = fork();

   switch (pid) {
         case -1:
```

```
            perror("fork error");
            exit(1);
        case 0:
            do_child(data_pipe);

        default:
            do_parent(data_pipe);
    }
    return 0;
}
```

2.1.2. Các hàm nâng cao

#include <stdio.h>

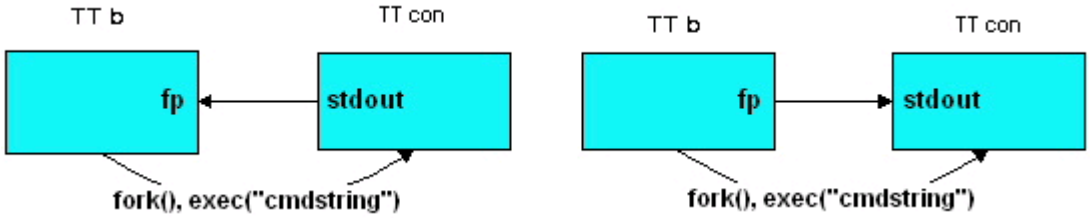
FILE *popen (const char *cmdstring, const char *type)

return: con trỏ tệp nếu OK, NULL nếu không thành công;

int pclose(FILE *fp);

return: trạng thái kết thúc của cmdstring, hoặc -1 nếu lỗi.

Hàm thực hiện tạo pipe, tạo TT con, kích hoạt exec shell để thực hiện "cmdstring", là một dòng lệnh của shell sẽ thực hiện bằng /bin/sh, sau đó đợi lệnh kết thúc.



Kết quả của fp = popen (cmdstring, "r") Kết quả của fp = popen (cmdstring, "w")

Ví dụ: Copy tệp vào môt chương trình more (hay còn gọi pager) ở (/"usr/bin/more). Lệnh shell ${PAGER:-more} nói rằng hãy dùng giá trị của biến shell PAGER nếu được định nghĩa và khác NULL, còn thì dùng biến **more**.

(popen2.c)

```
#definePAGER        "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
        char    line[MAXLINE];
        FILE    *fpin, *fpout;

        if (argc != 2)
                err_quit("usage: a.out <pathname>");
```

```
        if ( (fpin = fopen(argv[1], "r")) == NULL)
                err_sys("can't open %s", argv[1]);

        if ( (fpout = popen(PAGER, "w")) == NULL)
                err_sys("popen error");

                /* copy argv[1] to pager */
        while (fgets(line, MAXLINE, fpin) != NULL) {
                if (fputs(line, fpout) == EOF)
                        err_sys("fputs error to pipe");
        }
        if (ferror(fpin))
                err_sys("fgets error");
        if (pclose(fpout) == -1)
                err_sys("pclose error");
        exit(0);
}
```

Dịch và chạy:

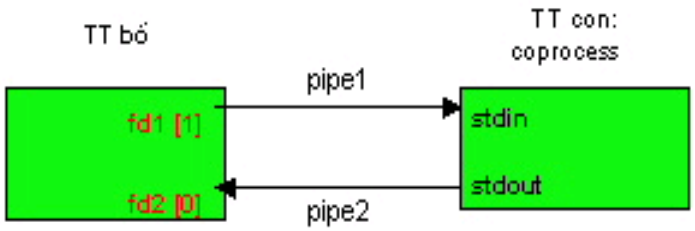
$popen2 /../../têntệp
Kết qủa giống như khi dùng "… | more"

2.1.3. Đồng Tiến trình (coprocess): Tạo pipe hai chiều

Một đồng TT là TT chạy ở chế độ nền (background) từ một shell mà đầu vào chuẩn (*stdin*) và đầu ra chuẩn (*stdout*) của nó nối tới một chương trình khác bằng một pipe. Trong khi popen() cho kết nối một chiều tới stdin hay từ một stdout của một TT khác, thì coprocess sẽ cho hai pipe một chiều nối tới một TT khác: môt nối vào stdin của coprocess, một nối tới stdout của coprocess. Ta có thể ghi vào stdin của nó, để đẩy cho coprocess xử lí, sau đó đọc ở stdout của nó.

Xem ví du sau:

1. Một TT tạo ra hai pipe: môt pipe là stdin của coprocess, và pipe kia là stdout của coprocess đó;
2. Thao tác: ghi vào stdin và đọc từ stdout của coprocess



Ví dụ 1: minh hoạ quá trình trên:

1. Tạo một chương trình (coprocess) thực hiện một công việc, ví dụ công hai số: đọc hai

số ở stdin, tính tổng, ghi ra tại stdout của nó.

2. Tạo một trình khác sẽ sử dụng coprocess nói trên.

Chương trình tạo hai pipe với fd1[ ] và fd2[ ]. Một cho stdin, và một cho stdout của ddd2. TT con gọi dup2() để chuyển các mô tả của pipe vào các stdin, stdout của TT con trước khi gọi execl("add2") để coprocess làm stdin, stdout của nó.Dịch chương trình với tên pipe4 và chạy. Nừu kill add2 trong khi pipe4 chạy, thì signal sẽ phát sinh.

1.Tạo coprocess: (add2.c: cộng hai số). Dịch ra thành tệp thực thi với tên **add2**

```
#include <stdio.h>
#include <unistd.h>
#include "ourhdr.h"

int
main(void)
{
        int     n, int1, int2;
        char    line[MAXLINE];

        while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0)
         {
                line[n] = 0;            /* null terminate */
                if (sscanf(line, "%d%d", &int1, &int2) == 2)
                {
                        sprintf(line, "%d\n", int1 + int2);
                        n = strlen(line);
                        if (write(STDOUT_FILENO, line, n) != n)
                                err_sys("write error");
                } else
                 {
                        if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                                err_sys("write error");
                }
        }
      exit(0);
}
```

2. Tạo một trình khác, dặt tên là pipe4.c, sẽ sử dụng add2:

```
#include        <signal.h>
#include        "ourhdr.h"

static void     sig_pipe(int);          /* our signal handler */

int
main(void)
{
        int     n, fd1[2], fd2[2];
```

```
pid_t   pid;
char    line[MAXLINE];

if (signal(SIGPIPE, sig_pipe) == SIG_ERR) /* cài xử lí signal*/
        err_sys("signal error");

if (pipe(fd1) < 0 || pipe(fd2) < 0)       /* Tạo ra hai pipe*/
        err_sys("pipe error");

if ( (pid = fork()) < 0)
        err_sys("fork error");
else if (pid > 0) {                       /* parent: đóng các fd không dùng**/
        close(fd1[0]);
        close(fd2[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
                n = strlen(line);
                if (write(fd1[1], line, n) != n)
                        err_sys("write error to pipe");
                if ( (n = read(fd2[0], line, MAXLINE)) < 0)
                        err_sys("read error from pipe");
                if (n == 0) {
                        err_msg("child closed pipe");
                        break;
                }
                line[n] = 0;    /* null terminate */
                if (fputs(line, stdout) == EOF)
                        err_sys("fputs error");
        }
        if (ferror(stdin))
                err_sys("fgets error on stdin");
        exit(0);

} else {                        /* child : đóng các fd không dùng*/
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
                if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                        err_sys("dup2 error to stdin");
                close(fd1[0]);
        }
        if (fd2[1] != STDOUT_FILENO) {
                if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                        err_sys("dup2 error to stdout");
                close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *) 0) < 0)    /* Child kích hoạt add2
coprocess*/
                err_sys("execl error");
```

```
        }
}

static void
sig_pipe(int signo)
{
        printf("SIGPIPE caught\n");
        exit(1);
}
```

Ví dụ2 Pipe hai đường

```
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>

void user_handler(int input_pipe[], int output_pipe[])
{
   int c;       int rc;

   close(input_pipe[1]);
   close(output_pipe[0]);


   while ((c = getchar()) > 0) {
      rc = write(output_pipe[1], &c, 1);
           if (rc == -1) {
              perror("user_handler: pipe write error");
              close(input_pipe[0]);
              close(output_pipe[1]);
              exit(1);
      }
         rc = read(input_pipe[0], &c, 1);
         if (rc <= 0) {
            perror("user_handler: read error");
            close(input_pipe[0]);
            close(output_pipe[1]);
            exit(1);
    }
         putchar(c);
  }

   close(input_pipe[0]);
   close(output_pipe[1]);
   exit(0);
}

void translator(int input_pipe[], int output_pipe[])
```

```
{
   int c;
   int rc;


   close(input_pipe[1]);

   close(output_pipe[0]);

   while (read(input_pipe[0], &c, 1) > 0) {
         if (isascii(c) && islower(c))
         c = toupper(c);
      rc = write(output_pipe[1], &c, 1);
      if (rc == -1) {
         perror("translator: write");
         close(input_pipe[0]);
         close(output_pipe[1]);
         exit(1);
      }
   }

   close(input_pipe[0]);
   close(output_pipe[1]);
   exit(0);
}

int main()
{
   int user_to_translator[2];
   int translator_to_user[2];

   int pid;
   int rc;

   rc = pipe(user_to_translator);
   if (rc == -1) {
         perror("main: pipe user_to_translator error");
         exit(1);
   }
   rc = pipe(translator_to_user);
   if (rc == -1) {
            perror("main: pipe translator_to_user error");
            exit(1);
   }

   pid = fork();

   switch (pid) {
```

```
        case -1:
           perror("main: fork error ");
           exit(1);

        case 0:
           translator(user_to_translator,
                      translator_to_user);

        default:

           user_handler(translator_to_user,
                        user_to_translator);

  }
   return 0;
}
```

2.2 FIFO

FIFO đôi khi còn gọi là *named pipe*. Với FIFO, các TT không có cùng huyết thống vẫn dùng chung được. Vì FIFO là tệp đặc biệt, nên cách tạo cũng giống như khi tạo tệp. Cú pháp tại FIFO như sau:

```
#include <sys/types>
#include <sys/stat.h>

int mkfifo ( const char *pathname, moe_t mode);
return:    0 nếu thành công, -1 có lỗi.
           mode:  O_RDONLY,
                  O_WRONLY,
                  O_RDWR
           Pathname: tên tệp sẽ tạo trong FS
```

Khi đã tạo được FIFO, các thao tác thực hiện bằng các hàm I/O trên tệp : close(), read(), write(), link(), unlink(), …

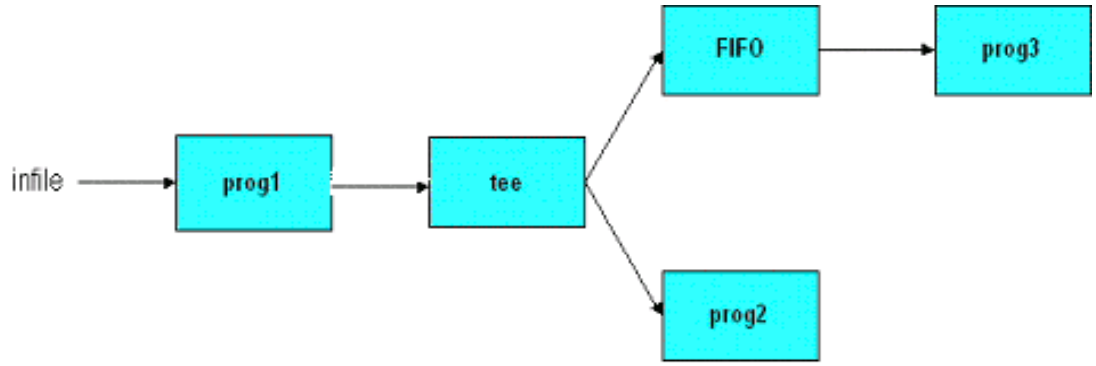
Có hai cách sử dụng:

1. các lệnh shell sử dụng để chuyển dữ liệu từ pipeline của nó cho pipeline của lệnh shell khác mà không cần tạo ra tệp tạm thời;
2. các ứng dụng client/server dùng để chuyển dữ liệu với nhauấuCc ví dụ:

Ví dụ:
1.Tạo ra một FIFO có tên fifo1,
2.Chạy chương trình có tên prg3 chế độ nền, với dữ liệu vào lấy từ fifo1 (dùng redirect),
3.Cho chạy chương trình có tên prg1, với dữ liệu vào lấy từ tệp ìnfile, lệnh **tee** sẽ copy stdin

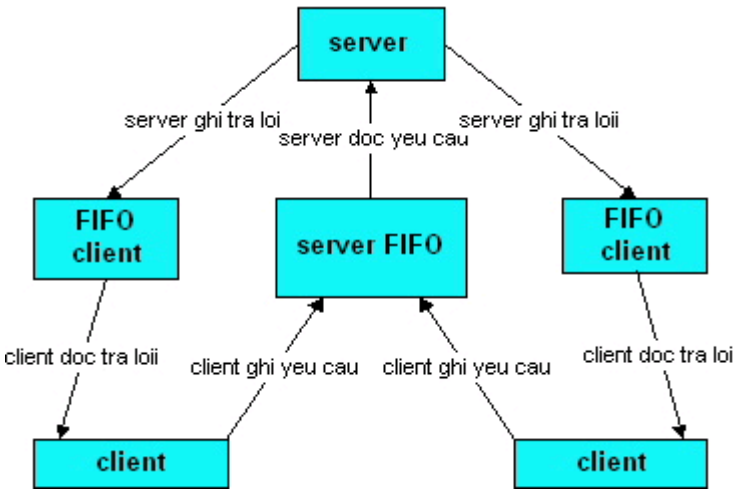
của nó cho cả stdout của nó và cho tệp có tên trong dòng lệnh.

```
$mkfifo fifo1
$prog3 < fifo1 &
$prog1 < ìnfile | tee fio1 | prog2
```



Sử dụng FIFO và **tee** để gởi một xâu tới hai TT khác nhau.

Client/server sử dụng FIFO:



Client – server liên lạc qua FIFO

2.3 Hàng thông điệp (message queue)

Hàng thông điệp là một danh sách liên kết của các thông điệp được lưu trong kernel và nhận biết bởi nhận dạng hàng thông điệp (message queue ID). Hàm mssget() dùng để mở hay tạo mới một hàng. Mỗi thông điệp có một trường kiểu nguyên dương, có độ dài không âm và các bytes dữ liệu. Kernel định nghĩa mỗi cấu trúc cho các kiểu IPC (message queue, semaphore, shared memory) và qui chiếu tới bằng một số nguyên định dạng không âm (gọi là ID, *identifier*). Khi một cấu trúc IPC được tạo ra, thì một khóa (*key*) phải được xác định (là

tên người dùng tự chọn). Kiểu dũ liệu của key là một số nguyên dài (*long integer* định nghĩa trong <sys/types.h>) và kernel sẽ biến đổi *key* thành ID. Còn có trường cờ trạng thái (flag) với các giá trị: bit IPC_CREAT, … để tạo mới một đầu vào trong hàng thông điệp với key chưa tồn tại.

Có bốn GHT để gởi thông điệp:

***msgget()*:** lấy (,hay để tạo) một mô tả thông điệp có chỉ rỏ hàng thông điệp dùng cho các GHT khác;

msgctl(): có tuỳ chọn lập các thông số và trả về các thông số kết hợp với một mô tả thông điệp, và tuỳ chọn để huỹ các mô tả;

msgsnd(): gởi một thông điệp;

msgrcv(): nhận một thông điệp.

Kernel lưu các thông điệp trên một hàng liên kết cho mỗi một mô tả và dùng *msgqid* làm chỉ số (index) để vào trường các đầu đề hàng đợi thông điệp. Thêm vào các phép truy nhập đã nói, cấu trúc hàng đợi có thêm các trường sau đây:

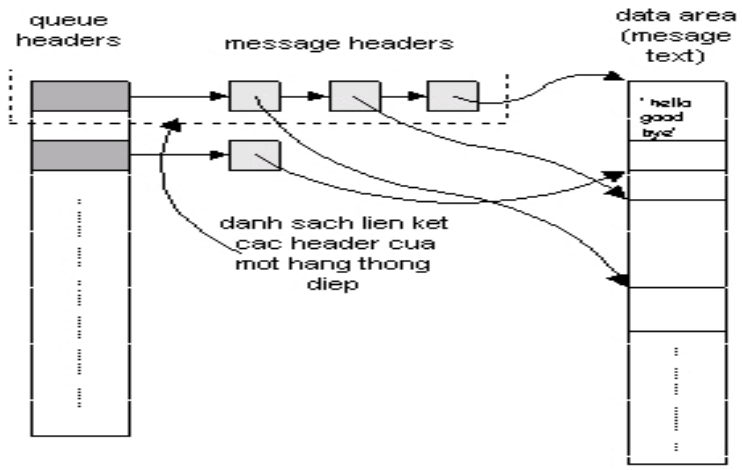
-các con trỏ trỏ vào thông điệp đầu tiên và thông điệp cuối cùng của danh sách liên kết;

-số các thông điệp và tổng số các bytes dữ liệu trên danh sách liên kết;

-số lượng tối đa các bytes dư liệu có thể trong danh sách;

-PIDs của các TT cuối cùng gởi và nhận thông điệp;

-Tem thời gian cho biết các thao tác thông điệp cuối cùng như *msgsnd, msgrcv, msgctl*.



Cau truc du lieu cho cac thong diep

- Để *tạo một mô tả thông điệp mới*, user thực hiện GHT *msgget()*, kernel tìm trường các hàng thông điệp để kiểm tra xem mô tả với key đi kèm đã có chưa. Nếu chưa có đầu vào nà ứng với key đó, kernel sẽ cấp một hàng thông điệp mới, khởi động các cấu trúc của hàng và trả lại mô tả của hàng. Trong các trường hợp khác, kernel kiểm tra quyền truy nhập hàng thông điệp và thoát khỏi GHT.

msgget*(key, flag);*

trong đó key là khoá gán cho hàng thông điệp, key có thể là IPC_PRIVATE cho biết

hàng thông điệp là riêng của TT và ẩn đối với các TT khác.

- Một TT khi cần *gởi một thông điệp*, sẽ thực hiện:

msgsnd*(msgqid, msg, count, flag)*

trong đó

msg là con trỏ trỏ vào cấu trúc chứa thông điệp như sau:

```
{
  long  mtype   /*kiểu thông điệp user tự chọn, nguyên dương*/
  char mtext    /*mảng văn bản thông điệp*/
}
```

count: số bytes cực đại tuỳ hệ;
flag: hành động kernel sẽ làm gì khi vượt quá buffer bên trong hệ thống.

Thuật toán gởi thông điệp như sau:

msgsnd()

```
input:  (1) message queue descriptor
        (2) address of message structure
        (3) size of message
        (4) flags
output: number of bytes sent
{
        . check  legality of descritor, permissions;
        . while ( not enough space to strore message)
                {
                        if (flags specify not to wait)
                                return;
                        sleep (until event enough space available);
                }
        . get message header;
        . read message text from user space to kernel;
        . adjust data structure: enqueue message header, message header point to data,
        count, time stamps, process ID;
        .wakeup all processes waiting to read message from queue;
}
```

Sau khi thực hiện kiểm tra độ dài thông điệp, kiểu thông điệp (nguyên dương) kernel cấp vùng nhớ cho thông điệp và copy data từ vùng nhớ của user vào đó. Kernel cấp đầu của một hàng thông điệp (queue header), đặt header thông điệp vào cuối danh sách liên kết các header của hàng thông điệp, lập con trỏ trong header đó chỉ vào văn bản thông điệp (data message trong hình). Sau đó cập nhật các trường số liệu (số thông điệp, số bytes trong hàng thông điệp, tem thời gian, ID của TT gởi thông điệp) trong header của hàng thông điệp (queue header trong hình). Sau cùng kernel đánh thức các TT đang đợi nhận thông điệp gởi tới hàng đó. Khi số bytes trong hàng nhiều hơn giới hạn cho phép, TT đi ngủ cho tới khi có thông điệp khác lấy ra khỏi hàng. Nếu TT xác định không cần đợi (lập cờ IPC_NOWAIT), TT thoát ngay

GHT với báo lỗi.
Sau đây là các ví dụ cho thấy quá trình tạo hàng thông điệp và các sử dụng:

Ví dụ 1:

1.Tạo hàng thông điệp:

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

int queue_id = msgget(IPC_PRIVATE, 0600);
int queue_id = msgget(1234, 0777);
```

2. Cấu trúc hàng thông điệp, phải khai báo trước khi chuyển thông điệp

```
struct msgbuf {
        long int mtype;
        char mtext[ 1 ]; /*mtype nguyên dương, biểu diễn thông điệp, dữ liệu là
    trường */
        };          /* biến thiên xác định độ dàI thông điệp*/
```

ví dụ:
Tạo thông điệp *char * msg_text = ” Xin chào”*, và xin cấp vùng nhớ:

```
struct msgbuf *msg = (struct msgbuf *) malloc (sizeof(struct msgbuf) + strlen(msg_text));
```

3. Gán một số nguyên tuỳ chọn biểu diễn ý nghĩa cho kiểu thông điệp:

```
msg->mtype = 23;
```

4. Sau đó copy nội dung vào cấu trúc cỉa thông điệp

```
strcpy(msg->mtext, msg_text);
```

5. Sau đó gởi thông điệp và hàng các thông điệp.

Ví dụ gởi thông điệp giữa các TT:

```
/* queue_defs.h:       Tệp định nghĩa các biến sẽ dùng:*/

#ifdef QUEUE_DEFS_H
#define QUEUE_DEFS_H
#define QUEUE_ID 137
#define MAX_MSG_SIZE 200
#define NUM_MESSAGES 100
#endif /* QUEUE_DEFS_H */

/*sender.c:     TT Gởi thông điệp*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>

#include "queue_defs.h"

struct msgbuf {
  long int mtype;
  char mtext[1];
};

int main(int argc, char* argv[])
{
   int queue_id;
   struct msgbuf* msg;

  int i;
  int rc;

   queue_id = msgget(QUEUE_ID, IPC_CREAT |  0777);
   if (queue_id == -1) {
                        perror("main: msgget error");
                        exit(1);
                        }
   printf("message queue created, queue id '%d'.\n", queue_id);

   msg = (struct msgbuf*)malloc(sizeof(struct msgbuf)+MAX_MSG_SIZE);
   for (i=1; i <= NUM_MESSAGES; i++)
     {
      msg->mtype = (i % 3) + 1;
      sprintf(msg->mtext, "hello world - %d", i);
      rc = msgsnd(queue_id, msg, strlen(msg->mtext)+1, 0);
      if (rc == -1) {
                     perror("main: msgsnd error");
                     exit(1);
                    }
     }
   free(msg);
   printf("generated %d messages, exiting.\n", NUM_MESSAGES);

   return 0;
}

/*TT nhận thông điệp: reader.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
#include "queue_defs.h"

struct msgbuf{
                long int mtype;
                char mtext[1];
              };

int main(int argc, char *argv[])
{
   int queue_id;
   struct msgbuf* msg;
   int rc;
   int msg_type;

   if (argc != 2) {
                fprintf(stderr, "Usage: %s <message type>\n", argv[0]);

                fprintf(stderr, "       <message type> must be between 1 and 3.\n");
                exit(1);
                }
   msg_type = atoi(argv[1]);
   if (msg_type < 1 || msg_type > 3)
                {
                 fprintf(stderr, "Usage: %s <message type>\n", argv[0]);
                 fprintf(stderr, "       <message type> must be between 1 and 3.\n");
                 exit(1);
                }

   queue_id = msgget(QUEUE_ID, 0);
   if (queue_id == -1) {
                        perror("main: msgget error");
                      exit(1);
                       }
   printf("message queue opened, queue id '%d'.\n", queue_id);

   msg = (struct msgbuf*) malloc(sizeof(struct msgbuf)+MAX_MSG_SIZE);
   while (1) {
              rc = msgrcv(queue_id, msg, MAX_MSG_SIZE+1, msg_type, 0);

              if (rc == -1) {
                              perror("main: msgrcv error");
                              exit(1);
                             }
             printf("Reader '%d' read message: '%s'\n", msg_type, msg->mtext);
             sleep(1);
            }
```

```
  return 0;
}
```

Ví dụ 2:
TT khách: TT gọi *msgget()* để nhận được một mô tả, với *key* đưa vào "MSGKEY":

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

#define     MSGKEY    75

struct msgform {
            long    mtype;
            char    mtext[256];
        };
main()
    {
        struct msgform msg;
        int msgid, pid, *pint;

        msgid = msgget(MSGKEY, 0777);

        pid = getpid();
        pint = (int *) msg.mtext;
        *pint = pid;    /*copy pid into message*/
        msg.mtype = 1; /*set message type=integer positive*/

        msgsnd(msgid, &msg, sizeof(int), 0);
        msgrcv(msgid, &msg, 255, pid,0);    /*pid is used as the msg type*/
        printf("client: receive from pid %d\n", *pint);
    }
```

TT tạo một thông điệp dài 256 bytes (tuy dùng không hết), copy PID của nó vào phần nội dung của thông điệp, gán kiểu thông điệp = 1 (là số nguyên dương), sau đó gởi thông điệp đi bằng GHT *msgsnd()*.

TT nhận thông điệp bằng:

count = ***msgrcv****(id, msg, maxcount, type, flag);*

id:	là mô tả thông điệp;
msg:	địa chỉ cấu trúc data của user chứa thông điệp nhận được;
maxcount:	kích thước mảng data trong msg,
type:	xác địnhkiểu thông điệp user muốn nhận;
flag:	cho kernel biết phải làm gì nếu không có thông điệp trong hàng;
count:	số bytes của thông điệp trả lại cho user.

Và thuật toán nhận thông điệp như sau: ***msgrcv()***

```
Input:  (1) message descriptor
        (2) address of data array for incoming message
        (3) size of data array
        (4) requested message type
        (5) flags
output: number of bytes in returned message
{
        . check permissions;
loop:
        . check legality of  message descriptor;
        /*now find message to trurn to user*/
        . if (requested message type = 0)
                consider first message on queue;
        . else if (requested message type > 0)
                consider first message on queue with given type;
        . else  /*type <0*/
                -consider first of  the lowest typed message on queue,
                such that its type is < = absolute value of requested type;
        . if (there is a message)
                {
                        -adjust message size or return error if user size too small;
                        -copy  message type, text from kernel space to user space;
                        -unlink message from queue;
                        -return;
                }
        /*no message*/
        . if (flags specify not to sleep)
                return with error;
        . sleep (event message arrives on queue);
        . goto loop;
}
```

Các trường hợp xử lí như sau:

Type = 0, lấy thông điệp đầu tiên trong danh sách liên kết các headers của hàng đợi và nếu độ dài thông điệp <= array[ ] của user, chuyển thông điệp vào cho user; kernel điều chỉnh lại cấu trúc dữ liệu: giảm số đếm các thông điệp, giảm số đếm các bytes trên hàng, dán tem thời gian nhận thông điệp, cập nhật PID của TT nhận, điều chỉnh danh sách liên kết, trả lại không gian bộ nhớ kernel đã dùng, đánh thức các TT trước đó (đang ngủ) muốn gởi thông điệp nhưng do chưa có chổ chứa. Trong trường hợp độ dài của thông điệp > maxcount do user lập, trả lại thông báo lỗi, thông điệp giữ lại trong hàng. Nếu TT bỏ qua lỗi nói trên (khi bit MSG_NOERROR lập trong *flag*), kernel cắt thông điệp (= maxcout)và trả lại số bytes user yêu cầu, huỹ toàn bộ thông điệp khỏi danh sách.

TT có thể lấy thông điệp theo *type* TT muốn. Nếu *type* là nguyên_dương, TT sẽ nhận được thông điệp đầu tiên của kiểu *type* đó. Nếu *type* <0, kernel tìm *type* thấp nhất của tất cả các thông điệp trên hàng nhỏ hơn hoặc bằng giá trị số tuyệt đối của *type* cho và trao cho TT thông điệp đầu tiên của *type* kiểu đó. Ví dụ, hàng thông điệp có 3 thông điệp với các kiểu là 3, 1, 2 tương ứng. Nếu TT (user) lấy thông điệp với type = -2, kernel sẽ trao thông điệp kiểu 1

(1<= | 2|). Trong các trường hợp khi không có thông điệp nào thoả mãn mà TT yêu cầu nhận, kernel đặt TT vào trạng thái ngủ, trừ khi TT đã lập bit IPC_NOWAIT trong flag để thoát ra ngay lập tức.

Hãy xem ví dụ sau đây, gọi là **TT phục vụ**:

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

#define     MSGKEY      75

struct msgform {
                long    mtype;
                char    mtext[256];
            }msg;
main()
    {
            int i; msgid, pid, *pint;
            extern cleanup();
            for( i=0; i <20; i++)
                signal(i, cleanup);
            msgid = msgget( MSGKEY, 0777 | IPC_CREAT);

            for (; ; )
            {
                msgrcv(msgid, &msg, 256, 1,0);
                pint = (int*)msg.mtext;
                pid = *pint
                printf("server: receive from pid  %d\n",pid);
                msg.mtype = pid;
                *pint =  getpid();
                msgsnd(msgid, &msg, sizeof(int), 0);
            }
    }

cleanup()
    {
            msgctl(msgid, IPC_RMID, 0);
            exit();
    }
```

Chương trình *TT phục vụ* cho cấu trúc của server cung cấp dịch vụ cho *TT client* trước đây. *TT phục vụ* nhận yêu cầu từ các *TT client,* cung cấp thông tin từ một database; *TT phục vụ* là điểm đơn giản để truy nhập vào database, hỗ trợ dễ dàng để làm cho batabase bền vững và an toàn. *TT phục vụ* tạo cấu trúc thông điệp bằng cách lập cờ IPC_CREAT trong msgget() và nhận tất cả các thông điệp theo kiểu 1, yêu cầu từ các *TT client*. *TT phục vụ* đọc văn bản thông điệp, tìm PID của *TT client* và đặt kiểu thông điệp bằng số PID của *TT client* đó. Trong

ví dụ này, *TT phục vụ* gởi trả lại PID của mình cho TT client trong văn bản thông điệp, TT client nhận thông điệp với kiểu thông điệp bằng số PID của nó (các lệnh trong vòng for(;;)). Bởi vậy *TT phục vụ* nhận chỉ các thông điệp gởi cho nó bởi các *TT client*, và các *TT client* nhận chỉ các thông điệp gởi đến từ *TT phục vụ*. Các TT này phối hợp với nhau để tạo ra cơ chế đa kênh trên một hàng thông điệp.

Các thông điệp được tạo ra theo định dạng kiểu cặp dữ liệu mà ở đó tệp dữ liệu là một dòng các bytes (byte stream). Tiền tố *type* cho phép các TT lựa chọn các thông điệp theo các kiểu riêng biệt nếu muốn, là một đặc tính không có trong cấu trúc của hệ thống tệp (FS). TT do đó có thể lấy ra các thông điệp kiểu cá biệt từ hàng thông điệp theo trình tự các thông điệp đã đến và kernel bảo trì chính xác thứ tự đó. Hoàn toàn có thể thực hiện trao đổi thông điệp ở mức user kết hợp với hệ thống tệp, nhưng với cơ chế thông điệp, các TT trao đổi dữ liệu hiệu quả hơn nhiều.

TT có thể truy vấn trạng thái của mô tả thông điệp, đặt trạng thái, loại bỏ mô tả thông điệp bằng *msgctl()* với cú pháp như sau:

msgctl(id, cmd, mstatbuf)

id:	nhận dạng của mô tả thông điệp,
cmd:	kiểu lệnh,
mstatbuf:	địa chỉ cấu trúc dữ liệu của user sẽ chứa các thông số kiểm soát hay kết quả truy vấn (Xem hổ trợ lệnh này ở HĐH để có chi tiết và các thông số).

Trong ví dụ của *TT phục vụ* TT chặn để nhận tín hiệu (signal()) và gọi cleanup() để loại thông điệp khỏi hàng. Nếu TT không thực hiện lệnh này, hoặc nhận một tín hiệu SIGKILL, thì thông điệp sẽ tồn tại ngay cả khi không có TT nào qui chiếu tới thông điệp đó. Điều đó sẽ gây lỗi khi tạo ra một hàng thông điệp mới cho một key đã cho và chỉ tạo được khi thông điệp kiểu này bị huỹ khỏi hàng.

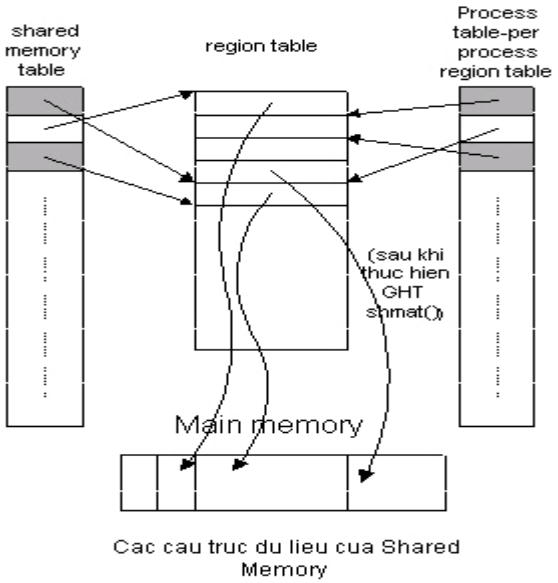
2.4 Vùng nhớ chia sẻ (shared memory region)

Các TT có thể liên lạc với nhau qua một phần trong không gian địa chỉ ảo của mình gọi là vùng nhớ chia sẻ (*shared memory),* sau đó thực hiện đọc / ghi dữ liệu vào đó. GHT ***shmget()*** sẽ tạo một miền mới của *vùng nhớ chia sẻ* hay trả lại miền nhớ nếu đã có tồn tại. ***shmat()*** sẽ gắn một miền nhớ vào không gian địa chỉ ảo của một TT, ***shmdt()*** làm việc ngược lại và ***shmctl()*** thao tác các thông số khác nhau kết hợp với mỗi vùng nhớ chia sẻ. TT đọc/ghi vào *shared memory* bằng các lệnh như thông thường với bộ nhớ nói chung. Sau khi một miền được gắn vào không gian của TT, thì miền đó như một phần của bộ nhớ của TT và không đòi hỏi thao tác gì đặc biệt.
Cú pháp của *shmget()* như sau:

shmid = shmget(key, size, flag);
size: là số bytes của miền.

Kernel dò trong bảng vùng nhớ chia sẻ để tìm *key*, nếu có *key* (= entry trong bảng) và chế độ truy nhập cho phép, kernel trả lại mô tả (*shmid*) của đầu vào đó; nếu không có và nếu trong GHT user đặt *flag*=IPC_CREAT để tạo vùng mới, kernel kiểm tra lại nếu kích thước (*size*)

yêu cầu là trong các giới hạn hệ thống thì cấp vùng nhớ cho TT gọi (bằng alloreg() đã nói trước đây). Các thông số kiểm soát như chế độ truy nhập, kích thước, con trỏ vào vùng nhớ vào trong bảng bộ nhớ chia sẻ. đặt cờ để cho biết chưa có bộ nhớ (của TT) nào gắn với vùng. Khi một TT gắn vùng vào bộ nhớ của TT, kernel cấp các page cho vùng, lập cờ để cho biết vùng sẽ không được giải trừ khi còn một TT cuối cùng gắn vào thực hiện exit(). Nhờ đó data trong bộ nhớ chia sẻ không mất đi ngay cả khi không có TT nào gán vào không gian nhớ của mình.



Một TT gắn vùng nhớ chia sẻ vào không gian địa chỉ ảo của TT với GHT *shmat()*:

virtaddr = shmat(id, addr, flags)

id do *shmget()* trả lại cho biết miền bộ nhớ chia sẻ, *addr* là miền địa chỉ ảo mà user muốn gắn bộ nhớ chia sẻ vào, *flags* xác định vùng sẽ chỉ đọc hay kernel có thể làm tròn (mở rộng hơn) địa chỉ user xác định, *viraddr* là địa chỉ ảo nơi kernel đã gắn vùng nhớ chia sẻ vào (không cần TT phải xác định).

Thuật toán gắn vùng nhớ chia sẻ vào không gian địa chỉ của một TT như sau:

```
Input:      (1) shared memory descriptor
            (2) virtual address to attach shared memory memory to
            (3) flags
output:     virtual address where shared memeory was attached.
{
      .check validity of descriptor, permission;
      .if ( user specified virtual address)
            {
                  .round off virtual address, as specified by flags;
                  .check legality of virtual address, size of region;
```

```
                }
        .else   /*user wants kernel to find good address*/
                .kerrnel picks virtual address, error if none available;
        .attach region to process address space (attachreg());
        .if (region being attached for first time)
                .allocate page table, memory for region (growreg());
        .return (virtual address where attached);
}
```

Vùng nhớ chia sẻ không được phủ lên các vùng khác trong không gian địa chỉ của TT, đồng thời vậy phải chọn rất thận trọng (ví dụ không gắn vào gần với vùng data của TT, TT mở rộng vùng data với địa chỉ liên tục, không gần với đỉnh của vùng stack) sao cho các vùng của TT khi tăng trưởng không chen vào đó.

Một TT gở vùng nhớ chia sẻ khỏi không gian địa chỉ ảo của TT với GHT *shmat()*:

shmat(addr)

Khi thực hiện kernel sẽ dùng GHT *detachreg()* đã đề cập. Vì bảng miền không có con trỏ ngược trỏ vào bảng vùng nhớ chia sẻ, nên kernel tìm trong bảng này đầu vào có con trỏ trỏ vào miền gắn và điều chỉnh luôn trường thời gian để biết lần cuối cùng vùng đã gở ra (*stamp*).

Ví dụ: Một TT tạo ra 128 Kbytes vùng nhớ chia sẻ và hai lần gắn vào vào không gian địa chỉ của nó ở các vị trí khác nhau. TT ghi vào vùng gắn "thứ nhất" và đọc ở vùng "thứ hai" (mà thực tế chỉ có một):

```
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/shm.h>
#define         SHMKEY      75
#define         K       1024

int     shmid;
main()
{
        int i, *pint;
        char *addr1, *addr2;
        extern char *shmat();
        extern cleanup();

        for (i = 0; i < 20; i++)
                signal(i, cleanup);
        shmid = shmget(SHMKEY, 128*K, 0777 | IPC_CREAT);
        addr1 = shmat(shmid, 0 ,0);
        addr2 = shmat(shmid, 0, 0);
        printf("addr1 0x%x  addr2 0x%x\n",addr1, addr2);
        pint = (int *)addr1;
        for(i=0;i<256;i++)
```

```
                *pint++ = i;
        pint = (int *) addr1;
        *pint = 256;

        pint = (int *) addr2;
        for(i=0; i<256; i++)
                printf("index %d\tvalue %d\n", i, *pint++);
        pause();
}
```

/*Huỹ vùng nhớ chia sẻ, cmd=IPC_RMID*/

```
cleanup()
{
        shmctl(shmid, IPC_RMID);
        exit();
}
```

Tiếp theo chạy một TT khác gắn vào cùng bộ nhớ chia sẻ (cùng khoá *SHMKEY*), chỉ dùng 46 K trong tổng số 128 Kbytes, TT này đợi cho TT đầu ghi các giá trị khác 0 vào từ đầu tiên của vùng nhớ chia sẻ, sau đó đọc nội dung. TT đầu sẽ nghỉ (*pause())* tạo điều kiện cho TT kia thực hiện. Khi TT đầu bắt được tín hiệu nó sẽ gở (*signal(i, cleanup)*)vùng nhớ chia sẻ ra khỏi không gian của mình.

```
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/shm.h>
#define         SHMKEY      75
#define         K       1024

main()
{
        int i, *pint;
        char *addr;
        extern char *shmat();

        shmid = shmget(SHMKEY, 64*K, 0777);

        addr = shmat(shmid, 0, 0);
        pint =  (int*) addr;
        while(*pint == 0)
        ;
        for(i=0;i<256;i++)
        printf("%d\n", *pint++);
}
```

TT dùng

shmctl(id, cmd, shmstatbuf)

để truy vấn trạng thái và đặt các thông số cho bộ nhớ chia sẻ. *Id* chỉ tới đầu vào trong bảng bộ nhớ chia sẻ, *shmstatbuf* là địa chỉ cấu trúc dữ liệu của user chứa các thông tin trạng thái của vùng nhớ chia sẻ khi truy vấn trạng thái hay khi đặt các thông số, *cmd* là lệnh cần thực hiện. Ví dụ trong hàm *cleanup()* nói trên.

2.5 Cờ hiệu (*semaphore*)

Các gọi hệ thống dùng cờ hiệu hổ trợ các TT đồng bộ việc thực hiện, bằng các thao tác tên tập các cờ hiệu. Trước khi áp dụng cờ hiệu, TT tạo ra việc khóa tệp (lock) bằng GHT creat(). Nếu có lỗi do: tệp đã có trong FS, hay có thể tệp đã khóa do một TT khác. Chổ bất tiện của cách tiếp cận này là các TT không biết khi nào thì có thể thử lạivà khóa tệp có thể tình cờ bỏ lại đằng sau khi hệ thống bị sự cố, hay khởi động lại.

Thuật toán Dekker mô tả áp dụng cờ hiệu như sau:

Các đối tượng có giá trị nguyên có hai thao tác cơ bản nhất định nghĩa cho chúng là P và V. Thao tác P giảm giá trị của cờ hiệu nếu giá trị đó > 0, thao tác V tăng gía trị đó. Vì rằng P và V là các thao tác thuộc loại nguyên tố (atomic) nên ít nhất P hay V thành công trên cờ hiệu ở bất kì thời điểm nào. Trên *System V*, P và V với vài thao tác có thể được làm đồng thời và việc tăng hay giảm các gía trị của các thao tác vì vậy có thể > 1. Vì kernel thực hiện tất cả các thao tác đó như là các nguyên tố, nên không một TT nào có thể điều chỉnh giá trị của thao tác cho tới khi thao tác hoàn tất. Nếu kernel không thể thực hiện *tất cả* các thao tác, có nghĩa kernel *không làm bất kì thao tác nào;* TT đi ngủ cho tới khi có thể làm tất cả các thao tác.

Cờ hiệu trong *System V* có các thành phần như sau:

- giá trị của cờ hiệu,
- PID của TT cuối cùng thao tác cờ hiệu,
- Tổng số các TT đang đợi để tăng giá trị của cờ hiệu lên,
- Tổng số các TT đang đợi giá trị của cờ hiệu sẽ bằng 0.

Để thao tác cờ hiệu các TT dùng các GHT sau đây:

semgt(): tạo và gán quyền truy nhập tập các cờ hiệu;
semctl(): thực hiện các thao tác kiểm soát tập các cờ hiệu;
semop(): thao tác giá trị của các cờ hiệu.

semgt() *tạo một mảng các cờ hiệu* như sau:

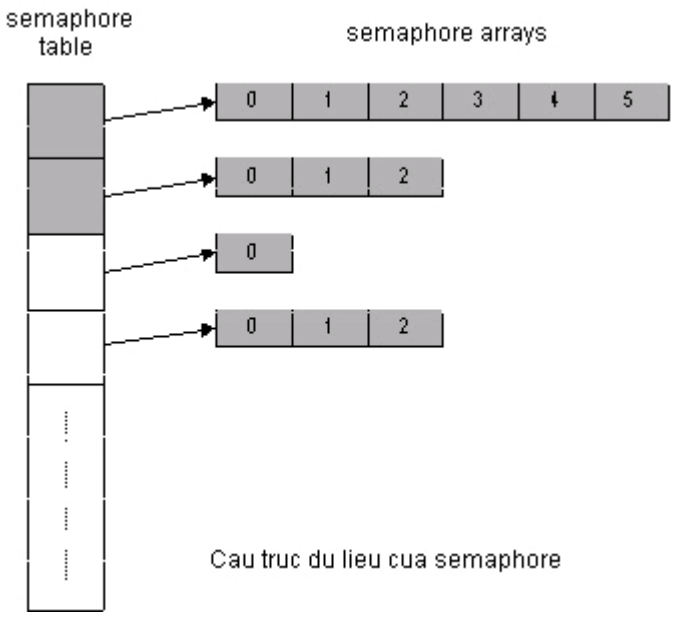
id = semget(key, count, flag);

Các đối trong hàm như trước đây đã định nghĩa cho các GHT thông điệp.Kernel cấp một đầu vào trong bảng các cờ hiệu với con trỏ trỏ vào trường cấu trúc cờ hiệu với *count* thành phần, đồng thời cho biết tổng số các cờ hiệu có trong mảng, thời điểm cuối cùng đã thực hiện *semop(), semctl()*.

Các TT thao tác cờ hiệu với semop():

oldval = semop(id, oplist, count);

id do semgt() trả lại, *oplist* là con trỏ trỏ vào mảng các thao tác cờ hiệu, *count* là kích thước của mảng. o*ldval* là giá trị của cờ hiệu cuối cùng đã thao tác trên tập trước khi thao tác (do gọi *semop()*) hoàn tất.



Mỗi thành phần của *oplist* như sau:

-số của cờ hiệu cho biết đầu vào của mảng các cờ hiệu sẽ được thao tác;
-thao tác (gì);
-các cờ.

Thuật toán như sau:

semop():
inputs: *(1) semaphore descriptor;*
(2) array of semaphore operations,
(3) number of element s in array.
Output: *start value of last semaphore operated on.*
{
.check legality of semaphore descriptor;
start: *.read array of semaphore operations fron user to kernel space;*
.check permissions for all semaphore operations;

.for (each semaphore in array)
{
-if(semaphore operation is positive)
{
-add "operation" to semaphore value;
-if (UNDO flag set on semaphore operation)
update process undo structure;
-wakeup all processes sleeping (event semaphore value
increases);
}

```
                -else if (seamphore operation is negative)
                        {
                         .if ("operation" + semaphore value > = 0)
                                {
                                -add "operation" to semaphore value;
                                -if(UNDO flag set)
                                        .update process undo structure;
                                -if(semaphore value 0)
                                        .wakeup all procces sleeping (event seamphore
                                                value become 0);
                                -continue;
                                 }
                                .reverse all semaphore operations already done this system
                                        call (previuos iteractions);
                                .if (flags specify not to sleep)
                                        return with error;
                                .sleep (event semaphore value increases);
                                .goto start; /*start loop from beginning*/
                        }
                -else   /*semaphore operation is 0*/
                        {
                         .if (semaphore value non 0)
                                {
                                -reverse all semaphore operations done this system
                                        calls;
                                -if (flags specify not to sleep)
                                        return error;
                                -sleep(event semaphore value ==0);
                                -goto start;
                                }
                        }
                } /*for loop end here*/
                /*semaphore operations all succeded*/
        .update time stamp, processe ID's;
        .return value of last semaphore operated on before all succeded;
}
```

Kerrnel đọc mảng các thao tác cờ hiệu, *oplist*, từ địa chỉ của user và kiểm tra các số cờ hiệu là đúng và TT đủ quyền để đọc hay thay đổi cờ hiệu. Có thể xảy ra trường hợp là đang vào lúc đầu này (thao tác oplist), kernel phải đi ngủ cho tới khi sự kiện chờ đợi xuất hiện, kernel sẽ khôi phục lại (đọc lại mảng từ địa chỉ của user) các giá trị của các cờ hiệu đã đang làm dỡ vào lúc đầu và khởi động lại GHT semop(). Công việc này thực hiện hoàn toàn tự động và cơ bản nhất.

Kernel thay đổi giá trị của một cờ hiệu theo *giá trị của thao tác (operation)*: nếu dương, tăng giá trị của cờ hiệu và đánh thức tất cả các TTđang đợi giá trị cờ hiệu tăng.; nếu là 0, kernel kiểm tra giá trị cờ hiệu và như sau: -nếu giá trị cờ hiệu là 0, kernel tiếp tục với các thao tác khác trong mảng, ngược lại kernel tăng tổng số các TT đang ngủ đợi giá trị có hiệu bằng 0, và đi ngủ (đúng ra là TT semop() đi ngủ, kernel đang thực hiện nhân danh semop() !).

Nếu thao tác cờ hiệu là âm, và giá trị tuyệt đối của nó <= giá trị của cờ hiệu,kẻnl cộng thêm giá trị của thao tác (hiện là âm) vào giá trị cờ hiệu. Nếu kết quả cộng = 0, kernel đánh thức tất cả các TT ngủ đang đợi giá trị của cờ hiệu sẽ = 0. Nếu giá trị cờ hiệu < giá trị tuyệt đối của thao tác cờ hiệu, kernel đưa TT đi ngủ với hoài vọng (event) rằng giá trị cờ hiệu sẽ tăng. Hể khi nào TT phải đi ngủ vào lúc giữa chừng của thao tác cờ, thì điều đó sẽ phụ thuộc vào mức ưu tiên ngắt; TT sẽ thức khi nhận được tín hiệu.

Ví dụ: ***Các thao tác (operation) khóa (Locking) và giải khóa (Unlocking):***

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

#defineSEMKEY    75
int semid;
unsigned int count;
/*definition of sembuf in the file sys/sem.h
*    struct sembuf {
*                unsigned short_num;
*                short sem_op;
*                short sem_flg;
                 };
*/
struct sembuf psembuf, vsembuf;    /*operations for P and V*/

main(argc, argv)
    int argc;
    char *argv[];
{
    int i, first, second;
    short initarray[2], outarray[2];
    extern cleanup();

    if (argc == 1)
    {
        for (i=0; i<20;i++)
            signal(i,cleanup);
        swmid=semget(SEMKEY, 2, 0777 | IPC_CREAT);
        initarray[0]=intarray[1]=1;
        semctl(semid, 2, SEALL, initarray);
        semctl(semid, 2, GETALL, outarray);
        printf("sem int vals  %d\n", outarray[0], outarray[1]);
        pause(); /* TT di ngu cho toi khi co signal*/
    }
    else if (argv[1][0] =='a')
    {
        first = 0;
        second = 1;
```

```
        }
        else
        {
                first =1;
                second = 0;
        }
        semid= semget(SEMKEY, 2, 0777);
        psembuf.sem_op = -1;          /*set semaphore P operation value*/
        psembuf.sem_flg= SEM_UNDO;
        vsembuf.sem_op = 1;           /*set semaphore V operation value*/
        vsembuf.sem_flg= SEM_UNDO;

        for(count=0; ;cout++)
        {
                psembuf.sem_num=first;
                semop(semid, &psembuf, 1);
                psembuf.sem_num=second;
                semop(semid, &psembuf, 1);
                printf("proc %d count %d\n", getpid(), count);
                vsembuf.sem_num=second;
                semop(semid, &vsembuf, 1);
                vsembuf.sem_num=first;
                semop(semid, &vsembuf, 1);
        }
}

cleanup()
{
        semctl(semid, 2,IPC_RMID,0);
        exit();
}
```

Sau khi dịch chương trình, ta có tệp thực thi là *a.out*. User cho chạy 3 lần theo trình tự sau đây trên terminal:

```
$ a.out &
$ a.out a &
$ a.out b &
```

Khi chạy không có đối đầu vào, TT tạo ra tập cờ hiệu với 2 thành phần và khởi đọng vớc các giá trị =1. Sau đó TT ngủ (pause()), và thức dậy khi có tín hiệu, huỹ cờ hiệu trong cleanup().
Khi chạy với đối "a", TT A thực hiện 4 thao tác cờ trong chu trình: giảm giá trị của cờ hiệu 0, giảm giá trị của cờ hiệu 1, thực hiện lệnh in, sau đó tăng giá trị của cờ hiệu 1 và cờ hiệu 0. TT sẽ đi ngủ nếu nó cố giảm giá trị của một cờ hiệu khi cờ hiệu đó là 0, vì vậy ta gọi cờ hiệu đó *bị khoá (locked)*. Vì rằng các cờ hiệu đều đã khởi động =1 và không có TT nào đang sử dụng cờ hiệu, TT A sẽ không bao giờ đi ngủ và các gí trị của cờ hiệu sẽ giao động giữa 1 và 0.

Khi chạy với "b", TT B giảm cờ hiệu 0 và 1 theo trình tự ngược lại của TT A. Khi TT A và TT B cùng chạy đồng thời, sẽ xảy ra tình huống rằng TT A đã khóa cờ hiệu 0 và muốn mở khóa cờ hiệu 1, nhưng TT Bđã khóa cờ hiệu đó và muốn mở khóa cờ hiệu 0. Không đạt được ý muốn, cả hai TT đi ngủ không thể tiếp tục được nữa. Ta nói cả hai TT bị "kẹt" (deadlock*ed*) và chỉ thoát khỏi hoàn cảnh này khi có tín hiệu.

Để loại trừ tình trạng như vậy, các TT có thể thực hiện nhiều thao tác cờ hiệuđồng thời bằng cách khởi động cấu trúc *sembuf* như sau:

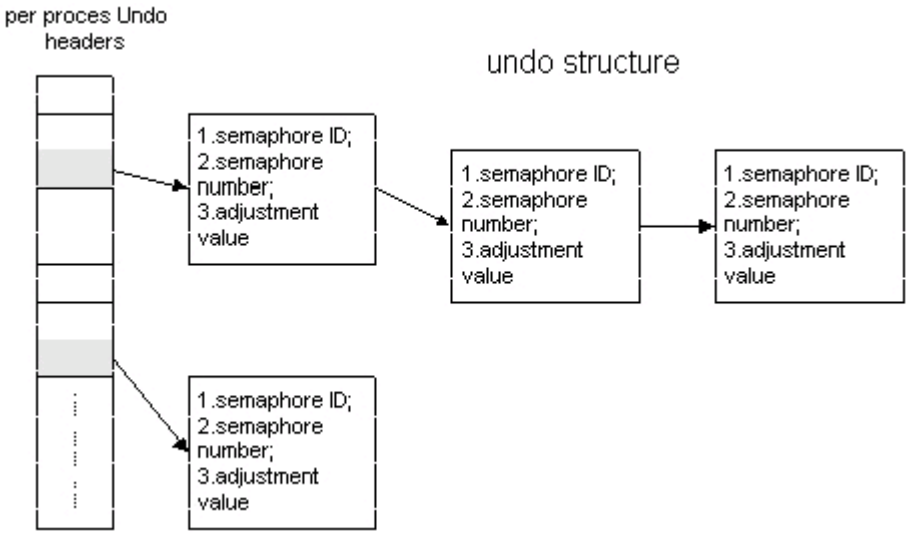
```
struct sembuf psembuf[2]:

psembuf[0].sem_num = 0;
psembuf[1].sem_num = 1;
psembuf[0].sem_op = -1;
psembuf[1].sem_op = -1;
semop(semid, psembuf, 2);
```

psembuf là mảng các thao tác cờ hiệu, tăng giá trị cờ hiệu 0 và 1 đồng thời. Nếu như chẳng hạn thao tác không thành công, TT ngủ cho tới khi cả hai thành công. Ví dụ, nếu giá trị cờ hiệu 0 là 1, và giá trị cờ hiệu 1 là 0, kernel có thể để các giá trị đó không bị thay đổi cho tới khi có thể giảm giá trị của cả hai.

TT có thể đặt cờ IPC_NOWAIT trong *semop();* nếu khi kernel rơi vào trường hợp ở đó TT ngủ vì phải đợi giá trị của cờ hiệu vượt quá một giá trị nào đó, hay có giá trị =0, kernel thoát ra khỏi GHT *semop()* với thông báo lỗi. Do vậy có thể áp dụng một kiểu cờ hiệu điều kiện mà bằng cờ hiệu đó TT sẽ không đi ngủ khi TT không thể thực hiện được việc gì.

Trường hợp nguy hiểm là khi TT thực hiện thao tác cờ hiệu, chẳng hạn khóa một nguồn tài nguyên nào đó, sau đó thoát ra (exit) mà không đặt lại giá trị cờ hiệu. Điều này có thể do lỗi của người lập trình, hay do nhận được tín hiệu dẫn đến việc đột ngột kết thúc một TT. Trong thuật toán *lock và unlock* đã đề cập nếu TT nhận được *tín hiệu kill* sau khi đã giảm giá trị của cờ hiệu thì TT không còn có cơ hội nào để tăng các giá trị đó bởi vì không thể còn mhận được *tín hiệu kill* nữa. Hậu quả là các TT khác nhận thấy cờ hiệu đã khóa và TT khóa cờ hiệu từ lâu không còn tồn tại nữa. Để loại trừ vấn đề, một TT đặt cờ SEM_UNDO trong *semop()*, để khi TT thoát, kernel sẽ đảo ngược hiệu quả của từng thao tác cờ hiệu mà TT đã làm. Để thực thi điều này, kernel duy trì một bảng với một đầu vào cho mỗi TT trong hệ. Đầu vào sẽ trỏ vào tập *các cấu trúc undo*, mà mỗi cấu trúc cho một cờ hiệu mà TT sử dụng. Mỗi cấu trúc *undo* là một mảng bộ ba gồm nhận dạng của cờ hiệu (semaphore ID), số của cờ hiệu trong tập nhận biết được bởi ID, và một giá trị điều chỉnh.



Kernel cấp động *cấu trúc undo* khi TT lần đầu tiên thực hiện semop() với cờ SEM_UNDO được lập. Sau đó với semop() có cờ SEM_UNDO lập, kernel sẽ tìm cấu trúc undo cho một TT có cùng nhận dạng của cờ hiệu (semaphore ID), cùng số cờ hiệu: khi tìm ra cấu trúc đó, kernel trừ giá trị của thao tác cờ hiệu với giá trị điều chỉnh. Do đó, cấu trúc undo chứa một tổng phủ định của tất cả các thao tác cờ hiệu mà TT đã làm trên cờ hiệu mà cho cờ hiệu đó cờ SEM_UNDO đã lập. Nếu cấu trúc như vậy không tồn tại, kernel tạo ra, sắp xếp một danh sách cấu trúc bằng nhận dạng của cờ hiệu (semaphore ID), cùng số cờ hiệu. Nếu giá trị điều chỉnh = 0, kernel huỹ cấu trúc đó. Khi một TT thoát ra, kernel gọi một thủ tục đặc biệt xử lí cấu trúc kết hợp với TT và thực hiện hành động xác định trên cờ hiệu đã chỉ ra.

Trở lại với thuật toán *lock_unlock* trước đó, kernel tạo cấu trúc undo mỗi lần khi TT giảm gía trị cờ hiệu và huỹ cấu trúc mỗi lần TT tăng giá trị cờ hiệu bởi vì giá trị điều chỉnh là 0.

Dưới đây là cấu trúc undo khi kích hoạt chương trình với thông số " a".

semaphore ID		sem ID
semaphore num		0
adjustment		1

sau thao tác thứ nhất

semaphore ID		sem ID	sem ID
semaphore num		0	1
adjustment		1	1

sau thao tác thứ hai

semaphore ID		sem ID
semaphore num		0
adjustment		1

sau thao tác thứ ba

Nếu TT đã phải thoát đột ngột kernel sẽ đi qua bộ ba nói trên và thêm giá trị 1 vào mỗi cờ hiệu, khôi phục lại giá trị của chúng =0. Trong điều kiện thông thường, kernel giảm giá trị

điều chỉnh của cờ hiệu số1 ở lần thao tác thứ ba, tương ứng với việc tăng giá trị của cờ hiệu và huỹ bỏ bộ ba giá trị bởi vì trị số của điều chỉnh =0. Sau lần thao tác thứ 4, TT sẽ không còn bộ ba nào nữa vì rằng các giá trị của điều chỉnh có thể đều =0.

Các thao tác mảng trên cờ hiệu cho phép TT loại trừ vấn đề "kẹt" (deadlock), nhưng lại rắc rối cung như nhiều ứng dụng không cần tới toàn bộ sức mạnh của các thao tác đó. Các ứng dụng sử dụng nhiều cờ hiệu có thể giải quyết deadlock ở mức user và kernel sẽ không chứa các GHT đã phức tạp hoá như vậy.

GHT semctl() chứa đựng vô số các thao tác kiểm soát cờ hiệu. Cú pháp như sau:

semctl(id, number, arg);
arg được khai báo là một *union*:

```
union semunion {
                int val;
                struct semid_ds  *semstat; /*xem them lenh*/
                unsigned short  *array;
        } arg;
```

Kernel thông dịch arg trên cơ sở giá trị của cmd.

6 Lưu ý chung

1. Kernel không có bản ghi nào về TT nào truy nhập cơ chế IPC, thay vì các TT có thể truy nhập cơ chế này nếu TT chắc chắn được chính xác ID và khi quyền truy nhập phù hợp;
2. Kernel không thể huỹ một cấu trúc IPC không sử dụng vì kernel không thể biết khi nào thì IPC không còn cần nữa, do vậy trong hệ có thể tồn tại các cấu trúc không còn cần đến;
3. IPC chạy trên môi trường của một máy đơn lẻ, không trên môi trường mạng hay môi trường phân tán; Sử dụng ***key*** trong IPC (chứ không phải là tên tệp), có nghĩa rằng các tiện ích của IPC gói gọn trong bản thân nó, thuận lợi cho các ứng dụng rỏ ràng, tuy vậy không có các khả năng như ***pipe*** và ***tệp***. IPC mang lại tốt cho việc thực hiện cho các ứng dụng phối hợp chặt chẻ, hơn là các tiện ích của hệ thống tệp (dùng tệp làm phương tiện liên lạc giữa các TT).

2.7 Dò theo một tiến trình (*tracing*)

Hệ Unix cung cấp một tiện ích nguyên thủy cho việc liên lạc giữa các TT dùng để theo dõi và kiểm soát việc thực hiện TT và rất tiện lợi cho làm gở rối (debug). Một TT debug sẽ kích hoạt một TT, theo dõi và kiểm soát việc thực hiện TT đó bằng GHT *ptrace(),* sẽ đặt điểm dừng trình, đọc ghi data vào không gian địa chỉ ảo. Vậy việc theo dõi TT bao gồm việc đồng bộ TT đi dò tìm (debugger) và TT bị dò theo (gở rối cho TT đó) cũng như kiểm soát việc thực hiện của TT đó.

Ta thử mô tả một cấu trúc điển hình của trình ***debugger*** bằng thuật toán sau:

```
if ((pid = fork()) ==0) /*debugger tạo TT con để kích hoạt trình cần trace*/
{
        /*child là TT bị theo dõi, traced*/
        ptrace(0,0,0,0);
        exec("put the name of traced processs here");
                                /* ví dụ: exec(a.out)*/
}
/*debugger process continues here*/
for(;;)
{
        wait((int *) 0);
        read(input for tracing instruction);
        ptrace(cmd, pid,...);
        if(quiting trace)
                break;
}
```

Trình debugger sinh ra một TT con, TT con thực hiện GHT *ptrace*, kernel đặt bit trace ở đầu vào của TT con trong *proces table* hệ thống, và nó kích hoạt (*exec()*) chương trình cần quan sát. Kerrnel thực hiện *exec* như thường lệ, tuy nhiên cuối cùng kernel nhận thấy bit trace đã dựng (set), và do đó gởi tín hiệu "TRAP" cho TT con. Khi kết thúc *exec*, kernel kiểm tra tín hiệu (như làm với tất cả các GHT khác) và tìm thấy "TRAP" mà kernel vừa gởi cho chính mình, kernel thực hiện mã xử lí (như khi thao tác các tín hiệu) nhưng là trường hợp đặc biệt. Vì trace bit dựng, TT con (đúng hơn là TT đang bị debug_traced proces) đánh thức TT bố (là trình debugger ngủ và đang trong vòng lặp ở mức user) khi thực hiện *wait()*, TT con chuyển vào *trạng thái trace đặc biệt* (tương tự như trạng thái ngủ nhưng không chỉ ra trên lưu đồ chuyển trạng thái). Debugger trở ra khỏi *wait(),* đọc lệnh do user đưa vào, chuyển thành một loạt liên tiếp *ptrace()* để kiểm soát TT đang debug (con).

Cú pháp của *ptrace()* như sau:

ptrace(cmd, pid, addr, data)

trong đó:

cmd: các lệnh như read (đọc data), write (ghi data), continue (tiếp tục) …
pid: số định danh của TT làm debug,
data: là một giá trị nguyên sẽ ghi (trả) lại.

Khi thực hiện *ptrace()*, kernel thấy rằng debugger có TT con với *pid* của nó và TT con đó đang trong trạng thái *trace*, kernel dùng các cấu trúc dữ liệu dò được tổng thể để trao đổi data giữa hai TT: copy *cmd, addr, data* vào cấu trúc dữ liệu (khoá cấu trúc để ngăn chặn các TT khác có thể ghi đè dữ liệu), đánh thức TT con, đặt TT con vào trạng thái "ready to run" còn bản thân đi ngủ chờ đáp ứng của TT con. Khi TT con trở lại thực hiện (trong chế độ kernel), nó thực hiện chính xác các lệnh *cmd*, ghi kết quả và cấu trúc dữ liệu nói trên và sau đó đánh thức TT trình bố (debugger). Tuỳ thuộc vào kiểu lệnh (*cmd*), TT con có thể sẽ quay trở lại trạng thái dò, đợi lệnh mới, hay thoát khỏi xử lí tín hiệu và tiếp tục thực hiện. Cho tới khi debugger tái chạy, kernel lưu gía trị trả lại mà TT bị dò thao trao, bỏ khoá các cấu trúc dữ liệu, và trở về user.

Hãy theo dõi hai trình, một trình gọi là dò theo (*trace*) và một trình gở rối (*debug*).

*trace: (là TT sẽ bị dò theo, tên trình là **tracce.c**)*

```
int data[32];
main()
{
        int i;
        for i=0;i<32;i++)
                printf("data[%d] = %d\n", i, data[i]);
        printf("ptrace data addres 0x%x\n".data);
}
```

Khi chạy trace tại terminal, các giá trị của trường data sẽ là 0.

*debug: (là TT thực hiện dò (dò theo dấu vết của trình **trace**, tên trình là **debug.c**)*

```
#define TR_SETUP 0
#define TR_WRITE 5
#define TR_RESUME 7
int addr;
main(argc, argv)
        int argc;
        char *argv[];
{
int i, pid;

sscanf(argv[1], "%x", &addr);
if ((pid = fork()) == 0)
        {
                ptrace(TR_SETUP,0,0,0);
                execl("trace","trace",0); /*là dò theo trình trace nói trên*/
                exit(0);
        }
        for(i=0;i<32;i++)
        {
                wait((int *) 0);
                /*ghi giá trị của i vào addr của TT có pid*/
                exit(0);
                addr + =sizeof(int);
        }
        /*TT bị dò theo có thể tiếp tục thục hiện tại đây*/
        ptrace(TR_RESUME, pid,1,0);
}
```

Bây giờ chạy trình debug với các giá trị do *trace* đã in ra trước đó. Trình *debug* sẽ lưu các thông số trong *addr* , tạo TT con. TT con này sẽ kích hoạt *ptrace()* để thực hiện các bước dò theo *trace*, sau đó cho chạy *trace* bằng *execl()*. Kernel gởi cho TT con (*trace*) tín hiệu SIGTRAP vào cuối *exec()l* và *trace* đi vào trạng thái dò theo (bắt đầu bị theo dõi), đợi lệnh do

debug chuyển vào. Nếu *debug* đã đang ngủ trong *wait()*, nó thức dậy và tìm thấy TT con đang bị dò theo, *debug* thoát ra khỏi *wait()*. *debug* gọi *ptrace()*, ghi *i* vào không gian địa chỉ của *trace* tại *addr*, tăng con trỏ của *addr* trong *trace*. (addr là địa chỉ của 1 thành phần trong mảng *data*). Cuối cùng *debug* gọi *ptrace()* để cho *trace* chạy, và vào thời điểm này *data* chứa giá trị 0 tới 31.

Dùng ptrace() để dò theo một TT là có tính sơ đẳng và chịu một số các hạn chế:

1.Kernel phảI thực hiện 4 chuyển bối cảnh để trao đổi 1 từ dữ liệu (word ò data) giữa debugger và TT bị dò theo: kernel -> debugger() khi gọi *ptrace()* cho tới khi TT bị dò theo trả lời các truy vấn, kernel <- > TT bị dò theo và kernel -> debugger: trở về debugger với kết qủa cho ptrace(). Tuy có chậm như là cần vì debuggerr không còn có cách nào truy nhập vào địa chỉ của TT bị dò theo.

2.debugger có thể dò theo nhiều TT con đồng thời. Trường hợp tới hạn là khi chỉ dò theo các TT con: Nếu TT con bị dò theo lại fork(), debugger sẽ không thể kiểm soát TT cháu, vì phảI thực hiện gở rối kiểu chương trình rất tinh vi: sẽ xảy ra việc thực hiện nhiều *exec* trong khi đó debugger không thể biết đuợc tên của các trình đã liên tục kích hoạt (là *images* của mã trình).

3.debugger không thể dò theo một TT đã và đang thực hiện nếu TT đó lại không phát sinh ptrace() để làm cho kernel biết rằng TT đồng ý để kernel thực hiện gở rối. Nhắc lại là TT đang gở rối đôi khi phải huỹ đi và khởi động lại, và điu này lại không làm được khi nó đã chạy.

4.Có một số trình không thể dò theo được, chẳng hạn trình *setuid()*, vì như vậy user sẽ vi phạm qui tắc an toàn khi ghi vào không gian địa chỉ qua *ptrace()* và thực thi các lệnh cấm.

3. Liên lạc trên mạng: client/server

Các chương trình ứng dụnh như thư điện tử (*mail*), truyền tệp từ xa (*ftp*), hay login từ xa (*telnet, rlogin*) muốn kết nối vào một máy khác, thường dùng các phương pháp nói chuyện (ad hoc) để lập kết nối và trao đổi dữ liệu. Ví dụ các chương trình mail lưu văn bản thư của người gởi vào một tệp riêng cho người đó. Khi gởi thư cho người khác trên cùng một máy, chương trình mail sẽ thêm tệp mail vào tệp địa chỉ. Khi người nhận đọc thư, chương trình sẽ mở tệp thư của người đó và đọc nội dung thư. Để gởi thư đến một máy khác, chương trình mail phải khéo léo tìm tệp thư ở máy đó. Vì rằng chương trình mail không thể thao tác các tệp ở máy kia (remote) trực tiếp, sẽ có một chương trình ở máy kia đóng vai trò một nhân viên phát hành thư (agent) cho các tiến trình thư. Như vậy các tiến trình của máy gởi (local) cần cách thức để liên lạc với (agent) máy kia (nhận) qua ranh giới của các máy. Tiến trình của máy gởi (local) gọi là tiến trình khách (*client*) của TT chủ (*server*) máy nhận thư.

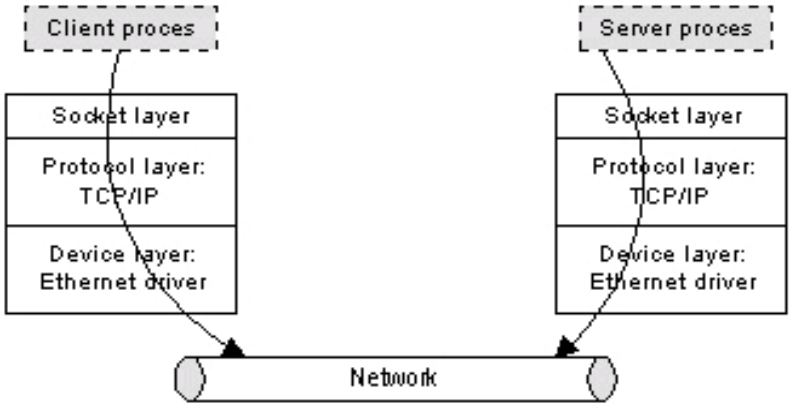
Bởi vì Unix tạo ra các TT bằng *fork()*, nên *TT server* phải tồn tại trước khi *TT client* cố lập kết nối. cách thức thông thường là thực hiện qua init và sau khi tạo TT server sẽ sẽ luôn đọc kênh liên lạc cho tới khi nhận được yêu cầu phục vụ và tiếp theo tuân theo các thủ tục cần thiết để lập kết nối. Các chương trình client và server sẽ chọn môi trường mạng và thủ tục theo thông tin trong các cơ sở dữ liệu của ứng dụng hay dữ liệu có thể đã mã cố định trong chương trình.

Với liên lạc trên mạng, các thông điệp phải có phần dữ liệu và phần kiểm điều khiển. Phần điều khiển chứa thông tin địa chỉ để xác định nơi nhận. Thông tin địa chỉ được cấu trúc theo kiểu và thủ tục mạng được sử dụng. Phương thức truyền thống áp dụng cho kết nối mạng liên quan tới *GHT ioctl()* để xác định các thông tin điều khiển, nhưng việc sử dụng không thuần nhất trên các loại mạng. Đó là chổ bất cập khi các chương trình thiết kế cho mạng này

lại không chạy cho mạng kia. Đã có nhiều nổ lực để cải thiện giao diện mạng cho Unix, sử dụng luồng (stream) là một cơ chế thích hợp hổ trợ mạng, vì các modul thủ tục có thể phối hợp linh hoạt bằng cách đẩy vào strem mở và cách sử dụng chuyển cho mức người dùng. Socket là một trong các thể hiện và được bàn ở phần tiếp theo.

4. Sockets

Để tạo ra phương pháp chung cho việc liên lạc giữa các tiến trình cũng như hổ trợ các giao thức mạng tinh xảo, Unix của BSD cung cấp một cơ chế gọi là ***socket***. Ta sẽ lược tả các khía cạnh của socket ở mức độ người dùng (user level).



Mô hình của Socket

Socket layer cung cấp ghép nối giữa GHT và các lớp thấp hơn;
Protocol layer có các modul giao thức (thủ tục) sử dụng để liên kết, ví dụ TCP/IP;
Device layer có các trình điều khiển thiết bị, kiểm soát hoạt động của thiết bị mạng, ví dụ Ethernet.

Các tiến trình liên lạc với nhau theo *mô hình client_server*: liên lạc được thực hiện qua đường dẫn hai chiều: một đầu cuối, TT server lắng nghe một socket, còn TT client liên lạc với TT server bằng socket ở đàu cuối đằng kia trên máy khác. Kernel duy trì mối liên kết và dẫn dữ liệu từ client đến server.

Các thuộc tính liên lạc chia sẻ chung cho các socket như các qui ước gọi tên, định dạng địa chỉ ..., được nhóm lại thành *domain*. BSD Unix hổ trợ " *Unix system domain*" cho các TT thực hiện liên lạc trên một máy và "*Internet domain*" cho các tiến trình thực hiện liên lạc trên mạng dùng giao thức DARPA (*Defence Advanced Reseach Project Agent*). Mỗi socket có một kiểu của nó, gọi là *virtual circuit* (hay *stream* theo thuật ngữ của BSD) hoặc *datagram*. *Virtual circuit* (mạng ảo) cho phép phát các dữ liệu tuần tự và tin cậy, *gatagram* không bảo đảm tuần tự, chắc chắn hay phát đúp, nhưng rẻ tiền hơn bởi không cần các thiết đặt đắt giá, nên cũng có ích cho một số kiểu liên lạc. Mỗi hệ thống có một giao thức mặc định, ví dụ giao thức TCP (Transport Connect Protocol) cung cấp dịch vụ mạng ảo, UDP (User Datagram Protocol) cung cấp dịch vụ datagram trong vùng Internet.

Cơ chế socket bao gồm một số GHT để thực hiện các liên lạc, bao gồm:

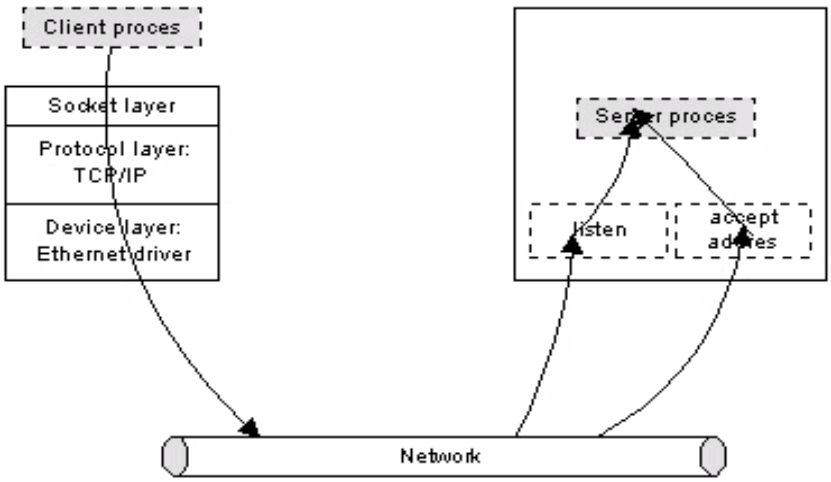
1. lập một điểm đầu cuối của liên kết liên lạc:
   sd = socket(format, type, protocol);

trong đó: *sd* là mộ mô tả của socket mà các TT sử dụng trong các GHT khác,
format định dạng vùng liên lạc là Unix domain hay Interet domain,
type kiểu liên lạc trên socket: virtual network, hay UDP,
protocol cho biết loại thủ tục kiểm soát sự liên lạc.

2. ***close()*** sẽ đóng socket lại.
3. ***bind (sd, address, length)*** kết hợp (đóng gói) một tên với mô tả socket,
trong đó: *sd* là mô tả socket.
address chỉ tới một cấu trúc xác định một định dạng vùng liên lạc và thủ tục xác định trong GHT socket(),
length độ dài của cấu trúc *addres*, nếu không có giá trị này kernel sẽ không biết addres sẽ dài bao nhiêu vì addres thay đổi theo vùng và thủ tục.

Ví dụ, một địa chỉ trong vùng Unix là một tên tệp. Các TT server sẽ bind() các địa chỉ này vào các socket và "thông báo" các tên của các địa chỉ để tự nhận biết cho các TT client.

4. ***connect(sd, addres, length)*** yêu cầu kernel tạo ra một kêt nối tới socket. ý nghĩa các đối của GHT như trên, tuy nhiên addres là địa chỉ của socket đích, mà socket này sẽ tạo ra một socket cuối khác của tuyến liên lạc. Cả hai socket phải dùng cùng một vùng liên lạc và cùng một thủ tục và kernel sắp xếp để các tuyến liên kết được lập chính xác. Nếu kiểu của socket là datagram, GHT *connect()* sẽ thông báo cho kernel địa chỉ sẽ dùng trong *send()* sau đó. Không có kết nối nào được lập vào thời điểm gọi *connect()*.
5. Khi kernel sắp xếp để chấp nhận một kết nối trên mạch ảo, kernel phải đưa vào xếp hàng các yêu cầu đến cho tới khi kernel có thể phục vụ các yêu cầu đó. Hàm listen() sẽ xsac địng đọ dài tối đa của hàng đợi này:
listen(sd, qlength);
qlength là số cực đại các yêu cầu chưa được phục vụ.
6. ***nsd = accept(sd, addres, addrlen)*** nhận một yêu cầu nối với TT server, với
sd là mô tả của socket,
addres trỏ vào mảng dữ liệu của user mà kernel sẽ nạp vào địa chỉ của TT client đang kết nối;
addlen cho kích thước của mảng nói trên.
Khi kết thúc và trở ra khỏi GHT, kernel sẽ ghi phủ lên nội dung của *addlen* bằng một số, cho biết lượng không gian mà địa chỉ đã lấy. *accept()* trả lại một mô tả của socket mới, *nsd,* khác với *sd*. TT server có thể tiếp tục lắng nghe socket đã nối trong khi vẫn liên lạc với TT client trên một kênh liên lạc khác như hình vẽ dưới.
7. Dữ liệu truyền trên socet đã kết nối bằng *send()* và *recv():*
count = send(sd, msg, length, flags);
sd, mô tả của socket;
msg con trỏ tới vùng dư liệu sẽ truyền đI;
length là độ dài dữ liệu;
count là số bytes thực sự đã truyền.
flags có thể đặt giá trị SOF_OOB để gởi đI dữ liệu "out-of-band", để thông báo rằng dữ liệu đang gởi đI, sẽ không xem như phần của thứ tự chính tắc của dữ liệu trao đổi giữa các tiến trình đang liên lạc với nhau. Ví dụ, trình login từ xa, rlogin, gởi thông điệp "out-of-band" để mô phỏng user ấn phím "delete" ở terminal.



Tiến trình server nhận và phục vụ

8. Nhận dữ liệu thực hiện bởi GHT
   count = recv(sd, buf, length, flags);
   buf là mảng cho dữ liệu nhận,
   length là độ dài hoài vọng sẽ nhận,
   count là số bytes cao lại cho chương trình của user,
   flags có thể đặt "peek" tại thông đIửp đến và kiểm tra nội dung mà không loại ra khỏi hàng, hay nhận "out-of-band".
9. Phiên bản cho *datagram* (UDP) là ***sendto()* và *recvfrom()*** có thêm các thông số cho các địa chỉ. Các TT có thể dùng *read()* hay *write()* thay cho *send() và recv()* sau khi kết nối đã lập. Do vậy, server sẽ quan tâm tới việc đàm phán thủ tục mạng sử dụng và kích hoạt các TT, mà các TT này chỉ dùng *read(), write()* như sử dụng trên tệp.
10. Để đóng kết nối của socket, dùng
    shtdown(sd, mode);
    mode cho biết là đó là bên nhận hay bên gởi, hay cả hai bên sẽ không cho phép truyền dữ liệu; nó thông báo cho các thủ tục đóng liên lạc trên mạng, nhưng các mô tả của socket hãy còn nguyên đó. Mô tả của socket sẽ giải trừ khi thực hiện *close().*
11. Để có được tên của socket đã đóng gói bởi *bind()*, dùng
    getsockname(sd, name, length);
12. ***getsockopt()* và *setsockopt()*** nhận lại và đặt các tuỳ chọn kết hợp với socket theo vùng (domain) và thủ tục (protocol) của socket.

```
struct sockaddr {
    unsigned short  sa_family;    /* address family, AF_xxx       */
    char                        sa_data[14];  /* 14 bytes of protocol address */
```

```
};

struct sockaddr_in {
    short int          sin_family;  /* Address family          */
    unsigned short int sin_port;    /* Port number             */
    struct in_addr     sin_addr;    /* Internet address        */
    unsigned char      sin_zero[8]; /* Same size as struct sockaddr */
};

/* Internet address (a structure for historical reasons) */

struct in_addr {
                unsigned long       s_addr;
            };
```

```
struct hostent {
    char *h_name;         /* official name of host */
    char **h_aliases;     /* alias list */
    int  h_addrtype;      /* host address type (e.g. AF_INET) */
    int  h_length;        /* length of address */
    char **h_addr_list;   /* list of addresses, null terminated */
};

#define h_addr h_addr_list[0] /* 1st address, network byte order */
```

The *gethostbyname()* function takes an internet host name and returns a `hostent` structure, while the function *gethostbyaddr()* maps internet host addresses into a `hostent` structure.

```
struct netent {
    char *n_name;         /* official name of net */
    char **n_aliases;     /* alias list */
    int n_addrtype;       /* net address type */
    unsigned long n_net;  /* network number, host byte order */
};
```

The network counterparts to the host functions are *getnetbyname()*, *getnetbyaddr()*, and *getnetent()*; these network functions extract their information from the `/etc/networks` file.

For protocols, which are defined in the `/etc/protocols` file, the `protoent` structure defines the protocol-name mapping used with the functions *getprotobyname()*, *getprotobynumber()*, and *getprotoent()*:

```
struct protoent {
    char *p_name;         /* official protocol name */
    char **p_aliases;     /* alias list */
    int  p_proto;         /* protocol number */
};
```

* Services available are contained in the `/etc/services` file. A service mapping is described by the `servent` structure:

```
struct servent {
     char *s_name;          /* official service name */
     char **s_aliases;      /* alias list */
     int s_port;            /* port number, network byte order */
     char *s_proto;         /* protocol to use */
};
```

The *getservbyname()* function maps service names to a `servent` structure by specifying a service name and, optionally, a qualifying protocol.

Ví dụ về một chương trình tạo ***TT server*** trong vùng Unix (Unix domain):

```
#include    <sys/types.h>
#include    <sys/socket.h>

main()
{
        int sd, ns;
        char buf[256];
        struct sockaddr sockaddr;
        int fromlen;

        sd = socket(AF_UNIX, SOCK_STREAM, 0);
        /*bind name- khong duoc co ki tu null trong ten*/
        bind(sd, "sockname", sizeof(*sockname) –1);
        listen(sd, 1);

        for(; ;)
        {
                ns = accept(sd, &sockaddr, &fromlen);
                if (fork == 0)
                {
                        /*child*/
                        close(sd);
                        read(ns, buf, sizeof(buf));
                        printf("server read '%s'\n", buf);
                        exit();
                }
        }
        close(ns);
}
```

TT tạo ra một socket trong vùng của Unix (Unix system Domain) và đóng gói *sockname* vào socket, sau đó kích hoạt listen() để xác định một hàng đợi dành cho các thông điệp gởi đến, sau đó tự quay vòng (for(;;)), để nhận thông điệp đến. Trong chu trình này accept() sẽ ngủ yên cho tới khi thủ tục xác định nhận ra rằng một yêu cầu kết

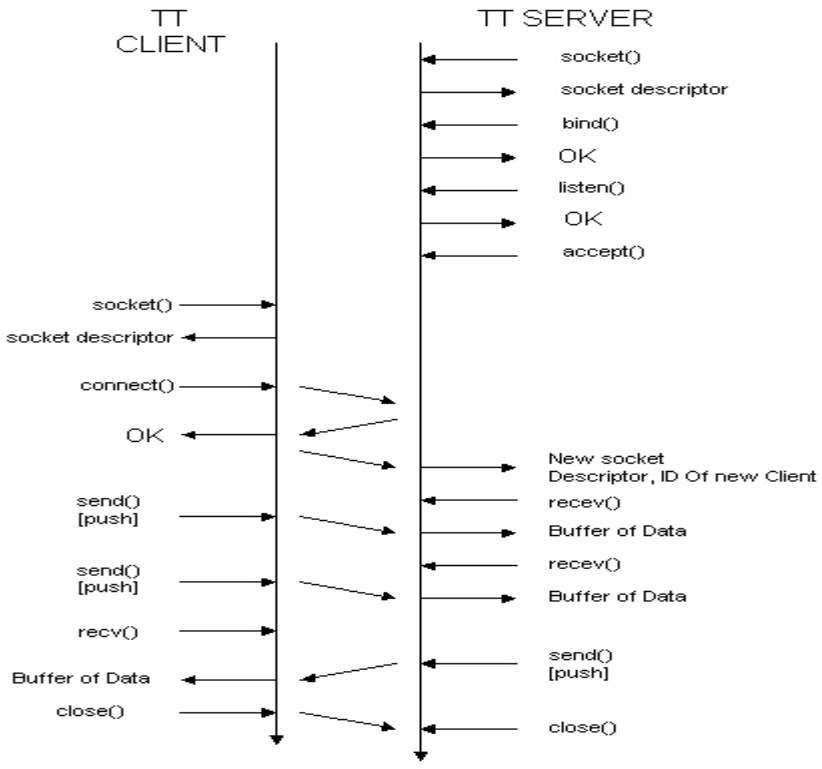
nối được dẫn thẳng tới socket với tên đã đóng gói. Sau đó accept() trả lại một mô tả mới (ns) cho yêu cầu sẽ đến. TT server tạo TT con để liên lạc với TT client: TT bố và TT con cùng đóng các mô tả tương ứng với mỗi TT sao cho các TT sẽ không cản trở quá trình truyền thông (traffic) của các TT khác. TT con thực hiện hội thoại với TT client (read()) và thoát ra, trong khi TT server quay vòng đợi một yêu cầu kết nối khác (accept()).

Một cách tổng quát, TT server được thiết kế để chạy vĩnh cữu và gồm các bước:

1. Tạo ra một socket của nó với khối điều khiển (*Transmision Control Block*_TCB) và trả lại một số nguyên, sd;
2. Nạp các thông tin về địa chỉ vào cấu trúc dữ liệu;
3. Đóng gói (bind): copy địa chỉ của socket và TCB, hệ gán cho server một cổng;
4. Lập hàng đợi có thể phục vụ cho 5 client;

Các bước sau đây lặp lại khong dừng:

5. Đợi TT client, Khi có tạo một TCB mới cho client, nạp địa chỉ socket , và các thông số khác của client vào đó;
6. Tạo một TT con để phục vụ TT client, TT con thừa kế TCB mới tạo và thực hiện các liên lạc với TT client: đợi nhận thông điệp, ghi và thoát ra.



Tạo ***TT client***:

```
#include      <sys/types.h>
#include      <sys/socket.h>

main()
{
```

```
        int sd, ns;
        char buf[256];
        struct sockaddr sockaddr;
        int fromlen;

        sd = socket(AF_UNIX, SOCK_STREAM, 0);
        /* connect to -name. Ki tu NULL khong thuoc vao name*/
        if(connect(sd, "sockname", sizeof("sockname") –1) == -1)
                exit();
        write(sd, "hi my love",10);
}
```

Sau khi tạo ra một socket có cùng vùng tên (Uinx name) như của server, TT phát sinh yêu cầ kết nối (connect()) với sockname đã đóng gói như của bên TT servẻ. Khi connect() thoát và kết nối thành công, TT client sẽ có một mạng (kênh) ảo liên lạc với TT server, sau đó gởi đi một thông điệp (write()) và thoát. Qúa trình xác lạp liên lạc giữa TT client và TT server thông qua gọi các sịch vụ của socket được mô tả như sau:

Khi TT server phục vụ *kết nối trên mạng*, vùng tên khai báo là "Internet domain", sẽ là:

socket(AF_INET, SOCK_STREAM,0);

và bind() sẽ đóng gói địa chỉ nạng lấy từ máy chủ phục vụ tên (name server). Hê thóng BSD có thư viện GHT để thực hiện các chức năng này. Tương tự như vậy cho đối thứ hai của connect() trong TT client sẽ chứa thông tin địa chỉ cần để nhận biết máy tính trên mạng, hay địa chỉ định tuyến (routing addres) để gởi thông điệp chuyển qua các máy tính mạng, và các thông tin hổ trợ khác để nhận biết các socket cá biệt trên máy đích. Nếu server lắng nghe mạng, và cả các TT cục bộ, TT servẻ sẽ dùng hai socket và GHT elect() để xác định TT client nào yêu cầu kết nối.

Các ví dụ khác

```
/*client1.c*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
   int sockfd;
   int len;

   struct sockaddr_un address;
   int result;
   char ch = 'A';


   sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
   address.sun_family = AF_UNIX;
```

```
    strcpy(address.sun_path, "server_socket");
    len = sizeof(address);

    result = connect(sockfd, (struct sockaddr *)&address, len);
    if(result == -1) {
        perror("oops: client1 problem");
        exit(1);
    }
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}

/server1.c*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;

    struct sockaddr_un server_address;
    struct sockaddr_un client_address;


    unlink("server_socket");
    server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    server_address.sun_family = AF_UNIX;
    strcpy(server_address.sun_path, "server_socket");

    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
    listen(server_sockfd, 5);
    while(1) {
        char ch;
        printf("server waiting\n");

        client_sockfd = accept(server_sockfd, (struct sockaddr *)&client_address, &client_len);
        read(client_sockfd, &ch, 1);
        ch++;
       write(client_sockfd, &ch, 1);
```

```
    close(client_sockfd);
  }
}
```

5. Tóm tắt và bài tập

Chương này đề cập các kỉ thuật kiểm soát và gở rối TT chạy trên máy, và các phương thức các TT liên lạc với nhau. *Ptrace()* theo dõi TT là một tiện ích tố nhưng đắt giá đối với hệ thống vì tốn nhiều thời gian và xử lí tương tác, thực hiện nhiều quá trình chuyển bối cảnh, liên lạc chỉ thực hiện giữa TT bố và TT con …

Unix Systen V cho các cơ chế liên lạc bao gồm thông điệp, cờ hiệu, và vùng nhớ chia sẻ. Có điều tất cả chỉ dùng cho các mục đích đặc biệt và cũng không thể áp dụng trên môi trường mạng. Tuy nhiên tính hữu ích rất lớn và mang lại cho hệ thống một tính năng cao hơn so với các cơ chế khác.

Unix hổ trợ liên kết mạng mạnh, hổ trợ chính thông qua tập GHT ioctl() nhưng sử dụng lại không nhất quán cho các kiểu mạng, do đó BSD đưa ra socket, dùng vạn năng hơn trên mạng. Stream là công cụ được dùng chính trên Unix System V.

Bài tập:

1. Viết một chương trình để so sánh chuyển data sử dụng bộ nhớ chia sẻ và sử dụng thông điệp. Trình dùng bộ nhớ chia sẻ có sử dụng cờ hiệu để đòng bộ việc hoàn tất đọc và ghi.
2. Chương trình (“nghe trộm”) sau đây làm gì ?

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/mes.h>
#define                 ALLTYPES    0

main()
{
        struct msgform
                        {
                                long mtype;
                                char mtext[1024];
                        } msg;
        register unsigned int id;

        for (id=0; ; id++)
                while(msgrecv(id, &msg, 1024, ALLTYPES, IPC_NOWAIT) > 0)
                ;
}
```

3. Viết lại chương trình ***Các thao tác (operation) khóa (Locking) và giải khóa (Unlocking)*** ở phần 2.2 dùng cờ ***IPC_NOWAIT*** sao cho các thao tác cờ hiệu (semaphore) là có điều kiện. Cho biết như vậy làm sao loại trừ được tình trạng kẹt (deadlock).

%

Hết phần cơ bản về Unix .%