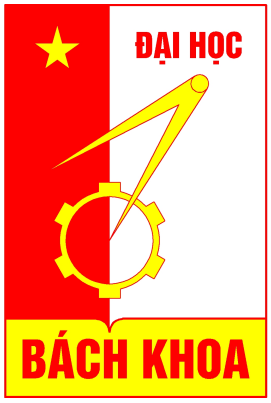
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
_____*******_____



BÁO CÁO
MÔN HỌC
Lập Trình Hệ Thống

Đề tài:

Quản lý tiến tình trong hệ điều hành Linux

Sinh viên thực hiện:

Họ và Tên	MSSV	Mã học phần
Đặng Quang Trung	20134145	IT3070

Giáo viên hướng dẫn:TS. Đỗ Quốc Huy

Hà Nội 10-05-2017

Mục lục

Danh sách hình vẽ

Danh sách bảng

Chương 1

Quản lý tiến trình trong hệ điều hành Linux

1.1 Tiến trình

Một tiến trình là một chương trình (các mã lệnh được lựu trữ ở tệp tin ngoài ổ đĩa) đang thực thi. Tuy nhiên tiến trình không chỉ mã chương trình thực hiện mà nó còn bao gồm một tập các nguồn như các files, các tín hiệu chờ, trạng thái tiến trình, không gian địa chỉ vùng nhớ với một hoặc nhiều ánh xạ vùng nhớ, một hoặc nhiều luồng thực thi, và một phần dữ liệu chứa các biến toàn cầu.

Các luồng thưc thi hay thường gọi ngắn gọn là luồng là các đối tượng hoạt động trong tiến trình. Mỗi một luồng sẽ bao gồm một bộ đếm chương trình, stack, tập các thanh ghi. Kernel lập lịch cho các luồng, không phải tiến trình. Trong các hệ thống Unix truyền thống, mỗi tiến trình bao gồm một luồng. Trong hệ thống mới,có nhiều luồng chương trình nên chứa nhiều hơn một luồng thông thường. Linux thực hiện đơn nhất các luồng, nó không phân biệt giữa luồng và tiến trình. Như vậy một luồng trong linux coi như một tiến trình đặc biệt.

Một tiến trình bắt đầu sự sống của nó khi nó được tạo. Trong linux điều này xảy ra khi hệ thống gọi lệnh fork(), nó sẽ tạo ra một tiến trình mới bằng cách nhân đôi một tiến trình đã tồn tại. Tiến trình gọi fork() là tiến trình cha(mẹ) tiến trình mới là tiến trình con. Tiến trình cha(mẹ) tiếp tục thực thi và tiến trình con thực thi cùng nơi.

Thường ngay khi một fork được gọi nó mong muốn thực thi một chương trình mới. Hàm exec() tương tự như hàm gọi tạo một không gian địa chỉ mới và load một chương trình mới vào đó. Đồng thời Linux kernels, fork() thực sự thực hiện thông qua gọi hệ thống clone().

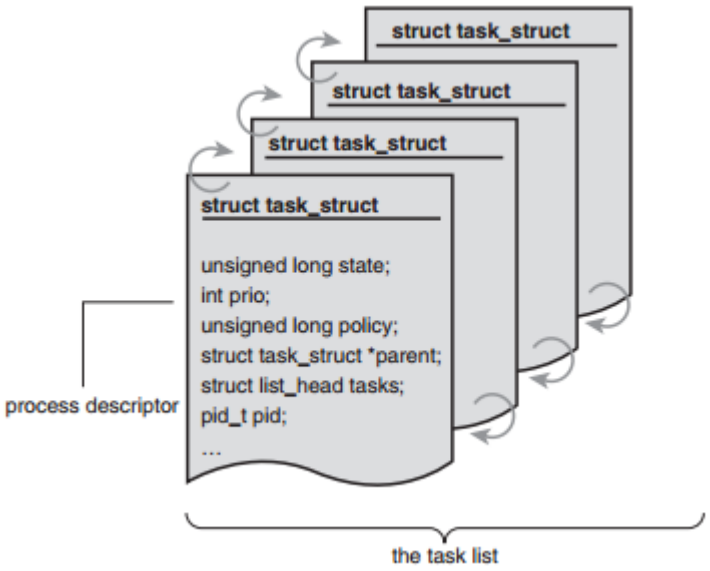
Cuối cùng, một chương trình thoát thông qua lời gọi exit(). Hàm này sẽ giải phóng tất cả các tài nguyên. Tiến trình cha có thể hỏi trạng thái của tiến trình con thông qua gọi hệ thống wait(). Khi một tiến trình thoát ra,

nó được đặt vào một trạng thái zombie đặc trưng đại diện cho các tiến trình bị chấm dứt cho đến khi cha mẹ gọi wait() hoặc waitpid().

1.2 Process descriptor và task structure

Nhân linux lưu trữ danh sách các tiến trình trong danh sách nối đôi và được gọi là task list. Mỗi một phần tử trong task list là một process descriptor kiểu **struct task_struct**. Nó được định nghĩa trong thư viện *<linux/sched.h>*. Process descriptor có chứa tất cả các thông tin riêng về một tiến trình.

Một **tast_struct** là một cấu trúc dữ liệu tương đối lớn, khoảng 1.7 kilibytes trên máy 32-bit. Tuy nhiên kích thước này lại khác nhỏ khi xem xét cấu trúc có chứa tất cả các thông tin mà nhân kernel có và cần cho tiến trình. Process descriptor chứa dữ liệu mô tả chương trình thực thi, không gian địa chỉ của tiến trình, các tín hiệu cờ , trạng thái của tiến trình, v.v ....



Hình 1.1: Process descriptor và task list

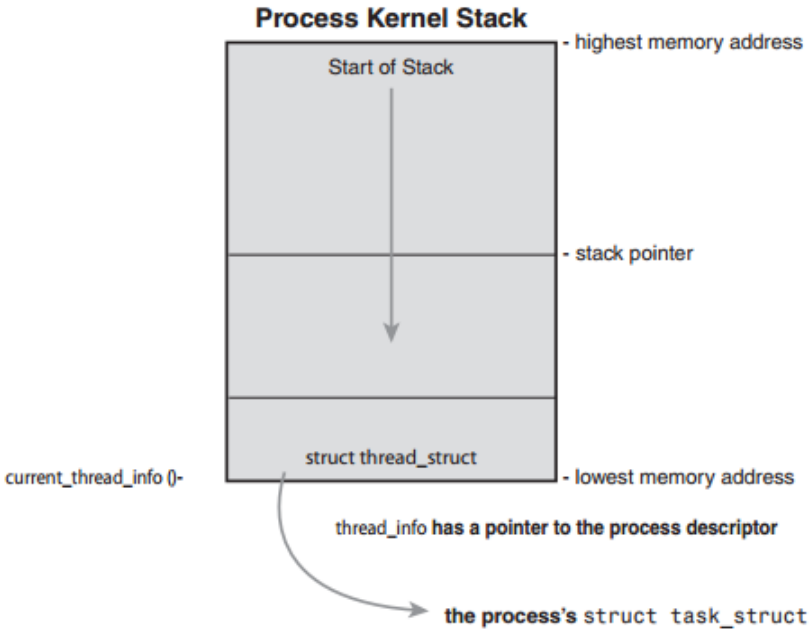
1.2.1 Allocating the Process Descriptor

Cấu trúc task_struct được phân bổ thông qua việc cấp phát slab để cung cấp tái sử dụng lại đối tượng và màu cache. Trước khi phiên bản kernel 2.6, struct task_struct được lưu trữ ở cuối mỗi ngăn kernel của mỗi tiến trình. Điều này cho phép các kiến trúc với ít thanh ghi, chẳng hạn như x86, để tính vị trí của process descriptor thông qua con trỏ ngăn xếp mà không sử dụng một thanh ghi bổ sung để lưu trữ vị trí. Với process descriptor hiện nay được tự động tạo ra thông qua việc phân bổ slab, một cấu trúc mới, struct thread_info, đã được tạo ra một lần nữa đặt ở dưới cùng của ngăn xếp (đối với các ngăn xếp

hướng đi xuống) và ở phía trên ngăn xếp (đối với các ngăn xếp hướng đi lên).

Cấu trúc thread_info được định nghĩa trên x86 trong <asm/thread_info.h> như:

```
struct thread_info {
        struct task_struct *task;
        struct exec_domain *exec_domain;
        __u32 flags;
        __u32 status;
        __u32 cpu;
        int preempt_count;
        mm_segment_t addr_limit;
        struct restart_block restart_block;
        void *sysenter_return;
        int uaccess_err;
};
```



Hình 1.2: The process descriptor and kernel stack

Mỗi cấu trúc thread_info của task được cấp phát ở phần cuối của phần tử stack của nó. Phần tử task của cấu trúc là một con trỏ tới task_struct thực tế của task.

1.2.2 Process ID

Mỗi tiến trình được đại diện bởi một định danh duy nhất, đó là process ID (thường được lưu pid). Các pid được đảm bảo là duy nhất tại bất kỳ điểm trong thời gian. Có nghĩa là trong thời gian t0 chỉ có 1 tiến trình với pid là 770 (không có bất cứ tiến trình khác tồn tại với giá trị như vậy). Không có sự đảm bảo rằng tại thời điểm t1 một tiến trình khác sẽ không tồn tại với pid 770.

Tiến trình idle - quá trình mà kernel "chạy" khi không có các tiến trình khác chạy - có pid 0. Tiến trình đầu tiên mà kernel thực hiện sau khi khởi động hệ thống, được gọi là tiến trình init, có pid 1. Thông thường, tiến trình init trên Linux là chương trình init. Chúng ta sử dụng từ "init" để tham chiếu cả quá trình ban đầu mà kernel chạy, và chương trình cụ thể được sử dụng cho mục đích đó.

Trừ khi người dùng nói cho kernel tiến trình để chạy (thông qua tham số dòng lệnh init kernel), kernel phải xác định một tiến trình init thích hợp,một ví dụ hiếm ở đây kernel đưa ra chính sách. Linux kernel thử 4 thực thi:

- /sbin/init: ưa thích và có thể đặt vị trí cho tiến trình.
- /etc/init: có khả năng đặt vị trị cho tiến trình khởi tạo.
- /bin/init: vị trí dự phòng cho tiến trình khởi tạo.
- /bin/sh: vị trí của vỏ Bourne, mà hạt nhân cố gắng để chạy nếu nó không tìm thấy một quá trình init.

Tiến trình đầu tiên của các tiến trình tồn tại được thực hiện như là tiến trình init. Nếu tất cả bốn tiến trình không thực hiện, hạt nhân Linux sẽ tạm dừng hệ thống đột ngột.

Sau khi chuyển giao từ hạt nhân, tiến trình init xử lý phần còn lại của tiến trình khởi động. Thông thường, điều này bao gồm việc khởi tạo hệ thống, bắt đầu các dịch vụ khác nhau, và khởi chạy một chương trình đăng nhập.

1.2.3 Cấp phát Process ID

Theo mặc định, kernel đặt giá trị ID tiến trình tối đa là 32768. Điều này tương thích với các hệ thống Unix cũ hơn, sử dụng các loại 16 bit nhỏ hơn cho ID tiến trình. Quản trị viên hệ thống có thể đặt giá trị cao hơn thông qua */proc/sys/kernel/pid_ max*, đổi sang một không gian pid lớn hơn để giảm khả năng tương thích.

Kernel phân bổ các ID tiến trình để xử lý theo một cách tuyến tính chặt chẽ. Nếu pid 17 là số cao nhất được phân bổ, pid 18 sẽ được phân bổ tiếp theo, ngay cả khi quy trình cuối cùng được giao pid 17 không còn chạy khi quá trình mới bắt đầu. Kernel không tái sử dụng các giá trị ID tiến trình cho đến khi nó kết thúc tốt đẹp từ phía trên - nghĩa là các giá trị trước đó sẽ không được sử dụng lại cho đến khi giá trị trong */proc/sys/kernel/pid_ max* được cấp phát. Vì vậy, trong khi Linux không đảm bảo tính duy nhất của các ID tiến trình trong một thời gian dài, hành vi phân bổ của nó sẽ cung cấp ít nhất sự thoải mái ngắn hạn trong sự ổn định và tính duy nhất của các giá trị pid.

1.2.4 Phân cấp các tiến trình

Tiến trình tạo ra một tiến trình mới được gọi là cha mẹ; tiến trình mới được gọi là tiến trình con. Mỗi tiến trình được sinh ra từ tiến trình khác (ngoại trừ, tất nhiên, tiến trình init). Do đó, mọi tiến trình con đều có cha mẹ. Mối quan hệ này được ghi lại trong tiến trình cha mẹ của tiến trình (ppid) của mỗi tiến trình, đó là pid của cha mẹ của tiến trình con.

Mỗi tiến trình được sở hữu bởi một người dùng và một nhóm. Quyền sở hữu này được sử dụng để kiểm soát quyền truy cập vào tài nguyên. Đối với kernel, người dùng và nhóm chỉ là các giá trị nguyên. Thông qua các tập tin */etc/passwd* và */etc/group*, các số nguyên này được ánh xạ tới các tên người dùng có thể đọc được mà người dùng Unix quen thuộc, chẳng hạn như root hoặc bánh xe nhóm (nói chung, kernel của Linux không quan tâm đến chuỗi có thể đọc được của con người, và muốn xác định các đối tượng với số nguyên). Mỗi tiến trình con được thừa hưởng quyền sở hữu của người dùng và nhóm của cha mẹ.

Mỗi tiến trình cũng là một phần của một nhóm tiến trình, chỉ đơn giản thể hiện mối quan hệ của nó với các tiến trình khác và không nên nhầm lẫn với khái niệm / nhóm người dùng nói trên. Tiến trình con thường thuộc cùng nhóm tiến trình với cha mẹ. Ngoài ra, khi một trình bao (shell) bắt đầu một đường ống (ví dụ như khi người dùng nhập ls | less), tất cả các lệnh trong đường ống đi vào cùng nhóm tiến trình. Khái niệm của một nhóm tiến trình giúp dễ dàng gửi tín hiệu đến hoặc nhận thông tin trên toàn bộ đường ống, cũng như tất cả tiến trình con của các tiến trình trong đường ống. Từ quan điểm của người dùng, một nhóm tiến trình có liên quan chặt chẽ đến công việc.

1.2.5 pid_t

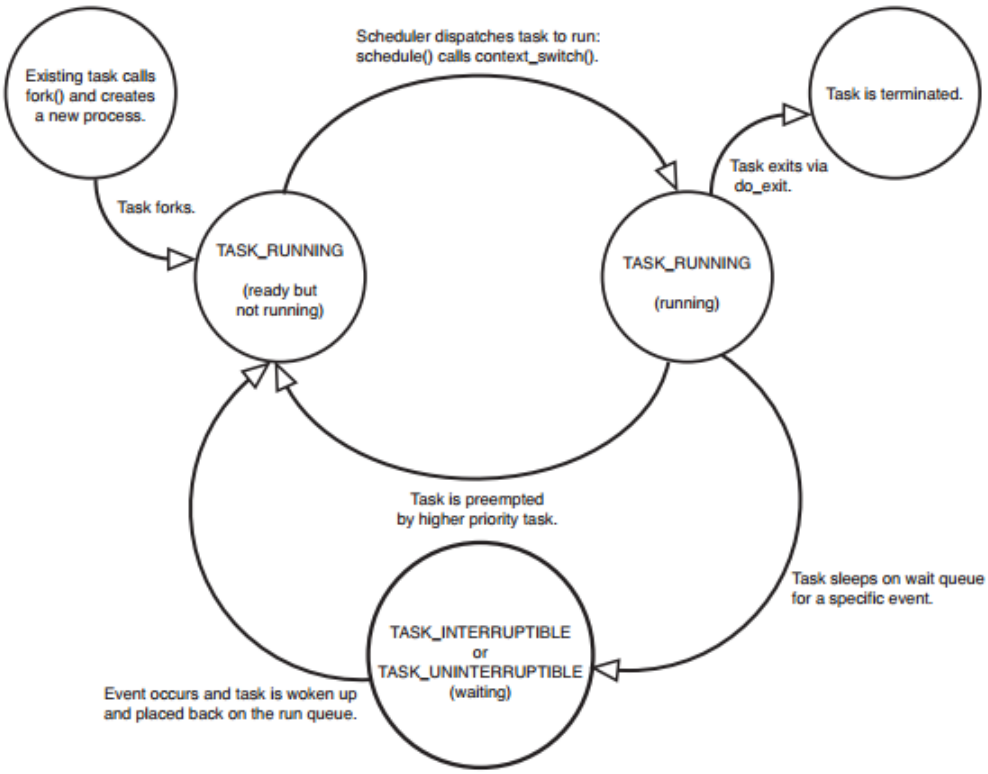
Theo chương trình, ID tiến trình được đại diện bởi loại pid_t, được định nghĩa trong tệp tin tiêu đề *<sys / types.h>*. Loại C sao lưu chính xác là kiến trúc cụ thể và không được định nghĩa bởi bất kỳ tiêu chuẩn C nào. Trên Linux, tuy nhiên, pid_t thường là typedef với kiểu int C.
Ví dụ: lấy Process ID và Parent Process ID trong C/C++

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 int main(int argc, char **argv) {
4         printf ("My␣pid=%jd\n" , (intmax_t) getpid ());
5         printf ("Parent's␣pid=%jd\n" , (intmax_t) getppid ());
6 }
```

1.2.6 Trạng thái process

Trạng thái mô tả của tiến trình mô tả tình trạng hiện tại của tiến trình. Mỗi tiến trình trên hệ thống xác định bởi một trong năm trạng thái khác nhau.

Giá trị này được biểu thị bằng một trong năm cờ:



Hình 1.3: Luồng biểu đồ trạng thái tiến trình

- TASK_RUNNING - Tiến trình có thể chạy, tiến trình hoặc là hiện tại đang chạy hoặc là đang ở trong hàng chờ chạy. Đây là trạng thái duy nhất có thể cho tiến trình thực hiện trong không gian người dùng; nó cũng có thể áp dụng cho một tiến trình mà nó đang hoạt động tích cực.
- TASK_INTERRUPTIBLE - Quá trình đang ngủ (có nghĩa là nó bị block) chờ đợi một điều kiện nào đó tồn tại. Khi điều kiện này tồn tại, kernel sẽ đặt trạng thái cờ của tiến trình là TASK_RUNNING. Tiến trình cũng sẽ đánh thức và trở thành runnable nếu nó nhận được một tín hiệu.
- TASK_UNINTERRUPTIBLE - Trạng thái này giống hệt với TASK_IN TERRUPTIBLE ngoại trừ việc nó không thức dậy và trở thành runnable nếu nó nhận được tín hiệu. Điều này được sử dụng trong trường hợp tiến trình phải chờ đợi mà không bị gián đoạn hoặc khi sự kiện này dự kiến sẽ xảy ra khá nhanh. Bởi vì công việc không phản hồi các tín hiệu trong trạng thái này, TASK_UNINTERRUPTIBLE ít thường được sử dụng hơn TASK_INTERRUPTIBLE.
- TASK_TRACE - Tiến trình này đang được theo dõi bởi một tiến trình khác, chẳng hạn như một trình debug, thông qua ptrace.
- TASK_STOPPED - Tiến trình thực hiện đã ngừng,công việc không chạy và cũng không đủ điều kiện để chạy. Điều này xảy ra nếu công việc nhận

được tín hiệu SIGSTOP, SIGTSTP, SIGTTIN, hoặc SIGTTOU hoặc nếu nó nhận được bất kỳ tín hiệu nào trong khi đang debugged.

1.2.7 Thao tác trạng thái tiến trình hiện tại

Mã kernel thường cần thay đổi trạng thái của một tiến trình. Cơ chế ưa thích sử dụng là
set_task_state(task, state);
Hàm này set task cho trước vào trạng thái nhất định. Nếu có thể, nó cũng cung cấp một rào cản bộ nhớ để buộc xếp trên bộ vi xử lý khác.(Điều này chỉ cần trên hệ thống SMP) Nếu không nó tương đương với
task -> state = state; Phương thức set_current_state(state) đồng nghĩa với set_task_state(current, state). Thấy *<linux/sched.h>* để thực hiện các hàm này và hàm liên quan.

Prcess Context

Một trong những phần quan trọng nhất của tiến trình là mã chương trình thực thi. Mã này được đọc từ một tệp thực thi và được thực hiện trong vùng địa chỉ của chương trình. Thông thường chương trình thực thi xuất hiện trong user-space. Khi mà một chương trình thực thi một lời gọi hệ thống hoặc triggers một ngoại lê,nó sẽ đi vào kernel-space. Tại thời điểm này, kernel được cho là "thực hiện thay mặt cho tiến trình" và đang trong process context. Khi mà trong process context, macro hiện tại là hợp lệ. Khi rời khỏi kernel, tiến trình sẽ tiếp tục thực hiện trong không gian người dùng, trừ khi tiến trình ưu tiên cao hơn đã trở thành runnable thay thế, trong trường hợp đó trình lập lịch được gọi để chọn tiến trình ưu tiên cao hơn.

Các lời gọi hệ thống và các trình xử lý ngoại lệ là những giao diện được xác định rõ ràng trong kernel. Một quá trình có thể bắt đầu thực hiện trong không gian kernel thông qua một trong các giao diện này tất cả các quyền truy cập vào kernel là thông qua các giao diện này.

The Process Family Tree

Một hệ thống phân cấp phân biệt tồn tại giữa các tiến trình trong các hệ thống Unix, và Linux không phải là ngoại lệ. Tất cả các tiến trình là con của tín trình init, PID của nó là một. Kernel bắt đầu init ở bước cuối cùng của tiến trình khởi động. Tiến trình init lần lượt đọc các initscripts của hệ thống và thực hiện nhiều chương trình hơn, cuối cùng là hoàn thành quá trình khởi động.

Mỗi tiến trình trên hệ thống có chính xác một cha(mẹ). Tương tự như vậy, mọi quá trình đều có từ 0 hoặc nhiều tiến trình con. Các tiến trình là tất cả trẻ em trực tiếp của cùng một cha mẹ được gọi là anh chị em ruột. Mối quan hệ giữa các quá trình được lưu trong bộ mô tả quá trình. Mỗi task_struct có một con trỏ tới task_struct của cha(mẹ), tên cha(mẹ) và một danh sách

các con. Do đó, với tiến trình hiện tại, ta có thể có được bộ mô tả quá trình của cha mẹ với mã sau:
struct task_struct *my_parent = current->parent;
Tương tự ta có thể lặp qua các tiến trình con:

```
struct task_struct *my_parent = current->parent;
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children) {
task = list_entry(list, struct task_struct, sibling);
/* task now point to one of current's children */
}
```

Bộ mô tả tiến trình của nhiệm vụ init được phân bổ tĩnh như init_task. Một ví dụ điển hình về mối quan hệ giữa tất cả các quy trình là một thực tế là mã này sẽ luôn thành công:

```
struct task_struct *task;
for (task = current; task != &init_task; task = task->parent);
/* task now points to init */
```

Trong thực tế, chúng ta có thể theo hệ thống phân cấp quá trình từ bất kỳ quá trình nào trong hệ thống tới bất kỳ hệ thống nào khác. Tuy nhiên, thường thì chỉ cần lặp lại tất cả các tiến trình trong hệ thống. Điều này rất dễ dàng bởi vì danh sách task là một danh sách liên kết,liên kết nối đôi. Để có được task tiếp theo trong danh sách, cho bất kỳ task hợp lệ, sử dụng:
list_entry(task->tasks.next, struct task_struct, tasks)

Lấy nhiệm vụ trước đó hoạt động theo cùng một cách.
list_entry(task->tasks.prev, struct task_struct, tasks)

Hai cách này được cung cấp bởi các macro next_task (nhiệm vụ) và prev_task (nhiệm vụ), tương ứng. Cuối cùng, macro for_each_process (nhiệm vụ) được cung cấp, lặp lại toàn bộ danh sách tác vụ. Trên mỗi lần lặp, task chỉ ra task kế tiếp trong danh sách:

```
struct task_struct *task;
for_each_process(task) {
        /* this pointlessly prints the name and PID of each task */
        printf("%s(%d)\n",task->comm, task->pid);
}
```

1.3 Process Creation

Tiến trình tạo trong Unix là duy nhất. Hầu hết các hệ điều hành thực hiện cơ chế sinh để tạo ra một tiến trình mới trong không gian địa chỉ mới, đọc trong một thực thi, và bắt đầu thực hiện nó. Unix có cách tiếp cận khác thường để tách các bước này thành hai chức năng riêng biệt: fork() và exec(). Đầu tiên, fork(), tạo một tiến trình con là một bản sao của task hiện tại. Nó khác với

cha(mẹ) chỉ PID của nó (đó là duy nhất), PPID của nó (PID của cha mẹ, được thiết lập để tiến trình ban đầu), và các tài nguyên và thống kê nhất định, chẳng hạn như các tín hiệu đang chờ giải quyết, không được cung cấp. exec(), tải một thực thi mới vào không gian địa chỉ và bắt đầu thực hiện nó. Sự kết hợp của fork() theo sau là exec() tương tự như các chức năng duy nhất mà hầu hết các hệ điều hành cung cấp.

1.3.1 Copy-on-write

Theo cách truyền thống, tất cả nguồn tài nguyên của cha(mje) được sao chép và nhân bản đủa cho tiến trình con. Cách tiếp cận này là ngây thơ và không hiệu quả vì nó sao chép nhiều dữ liệu có thể được chia sẻ. Tồi tệ hơn, nếu tiến trình mới ngay lập tức thực thi một hình ảnh mới, tất cả những gì sao chép sẽ bị lãng phí. Trong Linux, fork() được thực hiện thông qua việc sử dụng các trang copy-on-write. Copy-on-write (hoặc COW) là một kỹ thuật để trì hoãn hoặc hoàn toàn ngăn chặn việc sao chép dữ liệu. Thay vì trùng lặp không gian địa chỉ tiến trình,tiến trình cha(mẹ) và tiến trình con có thể chia sẻ một bản sao.

Tuy nhiên, dữ liệu được đánh dấu theo cách sao cho nếu nó được ghi vào, một bản sao được tạo ra và mỗi tiến trình nhận được một bản sao duy nhất. Do đó, việc sao chép các nguồn lực chỉ xảy ra khi chúng được viết; cho đến khi đó, chúng được chia sẻ chỉ đọc. Kỹ thuật này làm chậm sự sao chép của mỗi trang trong không gian địa chỉ cho đến khi nó thực sự được ghi vào.Trong trường hợp các trang không bao giờ được viết - ví dụ, nếu exec() được gọi ngay sau khi fork() - chúng không bao giờ cần phải được sao chép.

Chi phí duy nhất phát sinh bởi fork() là trùng lặp bảng trang của cha(mẹ) và tạo ra một descriptor tiến trình duy nhất cho tiến trình con. Trong trường hợp thông thường, một tiến trình thực hiện một hình ảnh thực thi mới ngay sau khi fork, tối ưu hóa này ngăn cản sao chép lãng phí số lượng lớn dữ liệu (với không gian địa chỉ, hàng chục megabyte). Đây là một sự tối ưu hóa quan trọng bởi vì triết lý Unix khuyến khích nhanh chóng quá trình thực hiện.

1.3.2 Forking

Linux thực hiện tìm kiếm fork() thông qua cuộc gọi hệ thống clone(). Cuộc gọi này có một loạt các cờ xác định các tài nguyên nào, nếu có, tiến trình cha(mẹ) và con sẽ chia sẻ. Các cuộc gọi của thư viện fork(), vfork() và __clone() tất cả các cuộc gọi hệ thống gọi clone() với các flags.Lời gọi clone() hệ thống, lần lượt,gọi các do_fork().

Phần lớn các công việc trong forking được xử lý bởi do_fork(), được định nghĩa trong kernel / fork.c. Chức năng này gọi copy_process() và bắt đầu

quá trình chạy. Tác phẩm thú vị được thực hiện bởi copy_process():

1. Nó gọi dup_task_struct, Nó tạo ra một kernl stack mới, thread_info structure và task_struct cho tiến trình mới. Các giá trị mới là giống nhau với các task hiện tại. Ở điểm này, tiến trình con và cha có process descriptors là giống nhau.
2. Sau đó kiểm tra xem tiến trình con mới sẽ không vượt quá giới hạn tài nguyên về số lượng tiến trình cho người dùng hiện tại.
3. Tiến trình con cần phân biệt nó với cha(mẹ) của nó. Các thành phần khác của process descriptor được xóa hoặc thiệt lập lại giá trị mới. Các thành phần của process descriptor không được thừa kế chủ yếu là thông tin thống kê. Phần lớn các giá trị trong task_struct không thay đổi.
4. Trạng thái của tiến trình con được đặt thành TASK_UNINTERRUPTIBLE để đảm bảo rằng nó chưa chạy.
5. copy_process() gọi copy_flags) để cập nhật các thành phần cờ của task_struct. Cờ PF_SUPERPRIV, biểu thị cho dù một task được sử dụng các đặc quyền superuser hay không, được xóa. Biểu tượng PF_FORKNOEXEC, biểu thị một tiến trình không gọi là exec(), đã được thiết lập.
6. Nó gọi alloc_pid() để gán một PID có sẵn cho nhiệm vụ mới.
7. Tùy thuộc vào các cờ được truyền vào clone(), copy_process() sao chép hoặc chia sẻ các tập tin mở, thông tin hệ thống tập tin, xử lý tín hiệu, không gian địa chỉ tiến trình, và không gian tên.Những tài nguyên này thường được chia sẻ giữa các chủ đề trong một quy trình nhất định; nếu không họ là duy nhất và do đó sao chép ở đây.
8. Cuối cùng, copy_process() dọn dẹp và trả về người gọi một con trỏ tới đứa trẻ mới.

Quay trở lại trong do_fork(), nếu copy_process() trả về thành công, đứa trẻ mới được đánh thức và chạy. Cố ý, hạt nhân chạy quá trình con đầu tiên. Trong trường hợp thông thường của tiến trình con chỉ đơn giản gọi exec() ngay lập tức, điều này sẽ loại bỏ bất kỳ copy-on-write trên đầu mà có thể xảy ra nếu cha(mẹ) chạy đầu tiên và bắt đầu viết vào không gian địa chỉ.

1.3.3 vfork()

Lời gọi hệ thống vfork() có tác dụng tương tự như fork (), ngoại trừ các mục của bảng trang của tiến trình cha(mẹ) không được sao chép. Thay vào đó, tiến trình con thực hiện như là một luồng trong không gian địa chỉ của tiến trình cha(mẹ), và cha mẹ bị khóa cho đến khi đứa trẻ gọi exec () hoặc thoát.

Lời gọi vfork() được thực hiện thông qua cờ đặc biệt lời gọi hệ thống clone():

1. Trong copy_process(), task_struct thành phần vfork_done được set bằng NULL.
2. Trong do_fork(), nếu cờ đặc biệt được cho, vfork_done sẽ trỏ đến địa chỉ đặc biệt.
3. Sau khi tiến trình con được chạy lần đầu tiên, tiến trình cha(mẹ) - thay vì trả lại - chờ đợi cho tiến trình con báo hiệu nó qua con trỏ vfork_done
4. Trong hàm release(), được sử dụng khi một task thoát khỏi một không gian địa chỉ bộ nhớ, task được thực hiện sẽ được kiểm tra xem nó có là NULL không. Nếu không, cha(mẹ) được báo hiệu.
5. Quay lại do_fork(), tiến trình cha(mẹ) được đánh thức và trả về.

Nếu xảy ra theo kế hoạch, tiến trình con hiện đang thực hiện trong một không gian địa chỉ mới, và tiến trình cha(mẹ) thực hiện lại trong không gian địa chỉ ban đầu của nó. Chi phí trên thấp hơn, nhưng việc triển khai không phải là khá.

1.4 The Linux Implementation of Threads

Các luồng là một chương trình trừu tượng phổ biến hiện đại. Chúng cung cấp nhiều luồng thực hiện trong cùng một chương trình trong không gian bộ nhớ chia sẻ. Chúng cũng có thể chia sẻ các tập tin mở và các tài nguyên khác. Các tập tin này cho phép lập trình đồng thời, và trên nhiều hệ thống xử lý, song song thực.

Linux thực hiện duy nhất của luồng. Đối với kernel Linux, không có khái niệm về một luồng. Linux thực hiện tất cả các luồng như các quy trình chuẩn. Linux kernel không cung cấp bất kì lập lịch đặc biệt hoặc cấu trúc dữ liệu đại diện cho luồng. Thay vào đó, một luồng chỉ đơn thuần là một quá trình chia sẻ tài nguyên nhất định với các quá trình khác. Mỗi luồng có một task_struct duy nhất và xuất hiện với kernel như là một tiến trình bình thường - luồng chỉ xảy ra để chia sẻ tài nguyên, chẳng hạn như không gian địa chỉ, với các quá trình khác.

Cách tiếp cận này đối với các luồng tương phản rất tốt với các hệ điều hành như Microsoft Windows hay Sun Solaris, có hỗ trợ kernel rõ ràng cho các luồng (và đôi khi gọi các luồng xử lý nhẹ). Đối với các hệ điều hành khác, luồng là một sự trừu tượng để cung cấp một đơn vị thực hiện nhẹ hơn, nhanh hơn tiến trình nặng. Đối với Linux, luồng chỉ đơn giản là chia sẻ tài nguyên giữa các tiến trình. Ví dụ, giả sử bạn có một tiến trình bao gồm bốn luồng. Trên các hệ thống với sự hỗ trợ luồng rõ ràng, một process descriptor có thể tồn tại, và ngược lại chỉ ra bốn luồng là khác nhau. Process descriptor mô tả các tài nguyên chia sẻ, chẳng hạn như không gian địa chỉ hoặc các tệp mở. Các đề tài sau đó mô tả các tài nguyên mà họ sở hữu . Ngược lại, trong

Linux, chỉ có bốn tiến trình và do đó có bốn task_struct thông thường. Bốn tiến trình được thiết lập để chia sẻ các tài nguyên nhất định.

1.4.1 Creating Threads

Các luồng được tạo giống như các tác vụ thông thường, ngoại trừ việc gọi hệ thống clone() được thông qua các cờ tương ứng với các tài nguyên cụ thể để chia sẻ:
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

Đoạn code trước dẫn đến hành vi tương tự như một fork() bình thường, ngoại trừ không gian địa chỉ, tài nguyên hệ thống tập tin, trình mô tả tập tin, và bộ xử lý tín hiệu được chia sẻ. Nói cách khác, task mới và cha(mẹ)của nó là những gì thường được gọi là luồng.

Ngược lại, một fork() bình thường có thể được thực hiện như:
clone(SIGCHLD, 0);

và vfork() được thực hiện như:
clone(CLONE_VFORK|CLONE_VM|SIGCHLD, 0);

Các cờ cung cấp cho clone() giúp xác định hành vi của tiến trình mới và chi tiết các nguồn tài nguyên mà cha(mẹ) và con sẽ chia sẻ. Bảng liệt kê các cờ clone, được định nghĩa trong <linux/sched.h>, và ảnh hưởng của chúng.

Flag	**Meaning**
CLONE_FILES	Parent and child share open files
CLONE_FS	Parent and child share filesystem information
CLONE_IDLETASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Child is to have same parent as its parent.
CLONE_PTRACE	Continue tracing child.
CLONE_SETTID	Write the TID back to user-space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals
CLONE_SYSVSEM	Parent and child share System V SEM_UNDO semantics.
CLONE_THREAD	Parent and child are in the same thread group.

CLONE_VFORK	vfork() was used and the parent will sleep until the child wakes it.
CLONE_UNTRACED	Do not let the tracing process force CLONE_PTRACE on the child.
CLONE_STOP	Start process in the TASK_STOPPED state.
CLONE_SETTLS	Create a new TLS (thread-local storage) for the child.
CLONE_CHILD_CLEARTID	Clear the TID in the child.
CLONE_CHILD_SETTID	Set the TID in the child.
CLONE_PARENT_SETTID	Set the TID in the parent.
CLONE_VM	Parent and child share address space.

1.4.2 Kernel Threads

Nó thường hữu ích cho kernel để thực hiện một số hoạt động ngầm. Kernel hoàn thành điều này thông qua các luồng kernel - tiêu chuẩn này mà chỉ tồn tại trong kernelspace. Sự khác biệt đáng kể giữa các luồng kernel và tiến trình thông thường là các luồng kernel không có một không gian địa chỉ.(Con trỏ mm của chúng, chỉ ra không gian địa chỉ của chúng, là NULL) Chúng chỉ hoạt động trong không gian kernel và không chuyển đổi ngữ cảnh sang không gian người dùng. Các luồng kernel có thể được lập lịch và có thể được dự phòng trước, giống như các tiến trình thông thường.

Linux ủy quyền một số tác vụ cho các luồng kernel, đáng chú ý nhất là các flush task và ksoftirqd task. Bạn có thể xem các luồng kernel trên hệ thống Linux của bạn bằng cách chạy lệnh ps -ef. Có rất nhiều luồng! Kernel được tạo ra trên hệ thống khởi động bởi các luồng hạt nhân khác. Thật vậy, một luồng kernel chỉ có thể được tạo ra bởi luồng kernel khác. Kernel xử lý điều này tự động bằng cách forking tất cả các luồng kernel off mới của tiến trình kernel kthreadd. Giao diện, được khai báo trong <linux / kthread.h>, để sinh ra một luồng kernel mới từ một trong những tiến trình hiện có là:

```
1 struct task_struct *kthread_create(int (*threadfn)(void *data),
2 void *data,
3 const char namefmt[],
4 ...)
```

Task vụ mới được tạo ra thông qua cuộc gọi hệ thống clone() thông qua tiến trình hạt nhân kthread. Tiến trình mới sẽ chạy hàm threadfn, nó được thông qua đối số dữ liệu. Tiến trình sẽ được đặt tên là namefmt, trong đó có các đối số định dạng kiểu printf trong danh sách đối số biến. Tiến trình được

tạo ra trong một trạng thái không thể tháo rời; nó sẽ không bắt đầu chạy cho đến khi nó được đánh thức thông qua wake_up_process(). Một tiến trình có thể được tạo và chạy được với một chức năng duy nhất, kthread_run():

```
struct task_struct *kthread_run(int (*threadfn)(void *data),
void *data,
const char namefmt[],
...)
```

Thủ tục này được thực hiện như là một macro, chỉ cần gọi cả kthread_create() và wake_up_process():

```
#define kthread_run(threadfn, data, namefmt, ...)
({
struct task_struct *k;
k = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__);
if (!IS_ERR(k))
wake_up_process(k);
k;
})
```

Khi bắt đầu, một kernel thread tiếp tục tồn tại cho đến khi nó gọi do_exit() hoặc một phần khác của kernel gọi kthread_stop(), đi qua địa chỉ của cấu trúc task_struct được trả về bởi kthread_create():
int kthread_stop(struct task_struct *k)

1.5 Process Termination

Khi tiến trình chấm dứt, kernel sẽ giải phóng tài nguyên thuộc quyền sở hữu của tiến trình và thông báo cho cha(mẹ) của tiến trình con về việc chấm dứt.

Nói chung, sự hủy tiến trình là tự gây ra. Nó xảy ra khi tiến trình gọi hệ thống gọi exit(), hoặc là rõ ràng khi nó đã sẵn sàng để chấm dứt hoặc ngầm trả về từ chương trình con chính của chương trình bất kỳ. (Nghĩa là trình biên dịch C đặt một cuộc gọi đến exit() sau khi trả về main()). Một tiến trình cũng có thể chấm dứt vô tình. Điều này xảy ra khi mà tiến trình nhận được một kí hiệu hoặc ngoại lệ nó không thể xử lý hoặc không quan tâm. Bất kể tiến trình kết thúc như thế nào, phần lớn công việc được xử lý bởi do_exit(), được định nghĩa trong kernel/exit.c, hoàn thành một số việc vặt khác:

1. Nó thiết lập cờ PF_EXITING trong thành phần cờ của task_struct.
2. Nó gọi del_timer_sync() để loại bỏ bất kỳ bộ tính giờ kernel nào. Khi trở về, nó được đảm bảo rằng không có hẹn giờ là xếp hàng đợi và không có xử lý hẹn giờ đang chạy.
3. Nếu bộ đếm chương trình được bật, do_exit() gọi acct_update_integrals() để ghi ra thông tin bộ đếm chương trình.

4. Nó gọi exit_mm() để giải phóng mm_struct được giữ lại bởi tiến trình này. Nếu không có tiến trình khác đang sử dụng không gian địa chỉ này, nếu không gian địa chỉ không được chia sẻ-kernel sau đó phá hủy nó.
5. Nó gọi exit_sem(). Nếu tiến trình này được xếp hàng chờ đợi một semaphore IPC, nó được lấy ra ở đây.
6. Sau đó nó gọi exit_files() và exit_fs() để giảm số lần sử dụng các đối tượng liên quan đến bộ mô tả tập tin và dữ liệu hệ thống tập tin. Nếu một trong hai sử dụng tính đến số không, đối tượng không còn được sử dụng bởi bất kỳ tiến trình, và nó bị phá hủy.
7. Nó thiết lập mã thoát của task, được lưu trữ trong thành viên exit_code của task_struct, đến mã được cung cấp bởi exit() hoặc bất cứ cơ chế kernel nào buộc phải chấm dứt. Mã thoát được lưu ở đây để truy xuất tùy chọn bởi cha mẹ.
8. Nó gọi exit_notify() để gửi tín hiệu đến cha(mẹ) của task, đặt lại cha(mẹ) cho bất kỳ con của công việc đến một luồng khác trong nhóm luồng của họ hoặc tiến trình init, và thiết lập trạng thái thoát của task, được lưu trữ trong exit_state trong cấu trúc task_struct, để EXIT_ZOMBIE.
9. do_exit() gọi schedule() để chuyển sang một tiến trình mới. Bởi vì tiến trình này bây giờ không schedulable, đây là đoạn mã cuối cùng công việc sẽ không bao giờ thực hiện. do_exit () không bao giờ trả về.

Tại thời điểm này, tất cả các đối tượng liên quan đến nhiệm vụ (giả sử nhiệm vụ là người dùng duy nhất) được giải phóng. Task không phải là runnable (và không còn khoảng trống địa chỉ để chạy nữa) và chỉ ở trạng thái exit EXIT_ZOMBIE. bộ nhớ nó chiếm là stack kernel của nó, cấu trúc thread_info, và cấu trúc task_struct.Các nhiệm vụ tồn tại chỉ để cung cấp thông tin cho cha mẹ của nó.Sau khi cha(mẹ) lấy thông tin, hoặc thông báo cho kernel rằng nó không quan tâm, bộ nhớ còn lại được giữ bởi tiến trình được giải phóng và trả về hệ thống để sử dụng.

1.5.1 Removing the Process Descriptor

Sau khi do_exit() hoàn thành, process descriptor cho tiến trình kết thúc vẫn còn tồn tại, nhưng tiến trình này là một zombie và không thể chạy. Như đã nói, điều này cho phép hệ thống có được thông tin về tiến trình con sau khi nó đã chấm dứt. Do đó, các hành vi dọn dẹp sau quá trình và gỡ bỏ process descriptor của nó là riêng biệt. Sau khi cha mẹ đã thu thập thông tin về tiến trình con bị chấm dứt của nó, hoặc được biểu thị cho kernel mà nó không quan tâm, task_struct của tiến trình con sẽ được deallocated.

Chuỗi các chức năng wait() được thực hiện thông qua một cuộc gọi hệ thống đơn (và phức tạp), wait() .Các hành vi tiêu chuẩn là tạm ngưng việc thực hiện tác vụ gọi cho đến khi một trong số các con của nó thoát, lúc đó hàm trả về với PID của đứa trẻ xuất cảnh. Ngoài ra, một con trỏ được cung

cấp cho hàm mà khi trả lại giữ mã thoát của con bị chấm dứt.

Đến lúc deadlocate process descriptor, release_task() được gọi:

1. Nó gọi __exit_signal(), gọi __unhash_process(), nó sẽ gọi detach_pid() để loại bỏ tiến trình từ pidhash và loại bỏ tiến trình khỏi danh sách công việc
2. __exit_signal() giải phóng bất kỳ tài nguyên còn lại nào được sử dụng bởi tiến trình đã kết thúc.
3. Nếu task vụ là thành phần cuối cùng của nhóm luồng, và tiến trình đầu của một zombie, sau đó release_task() thông báo cho cha mẹ của tiến trình đầu của zombie.
4. release_task() gọi put_task_struct() để giải phóng các trang chứa tiến trình kernel và cấu trúc thread_info và giải phóng bộ nhớ cache slab chứa task_struct.

Tại thời điểm này, process descriptor và tất cả các tài nguyên thuộc về tiến trình này đã được giải phóng.

1.6 The Dilemma of the Parentless Task

Nếu tiến trình cha(mẹ) thoát ra trước tiến trình con của nó, một số cơ chế phải tồn tại để đặt lại tiến trình cha(mẹ) cho bất kỳ task tiến trình con nào trong tiến trình mới, hoặc nếu các tiến trình chấm dứt không có cha mẹ sẽ mãi mãi là zombie, lãng phí bộ nhớ hệ thống. Giải pháp là đặt lại cha(mẹ) cho tiến trình con của task ở chỗ thoát ra trong nhóm luồng hiện tại hoặc, nếu không thành công, quá trình init. do_exit() gọi exit_notify(), gọi là forget_original_parent(), và lần lượt các cuộc gọi find_new_reaper() để thực hiện đặt lại cha(mẹ):

```
1  static struct task_struct *find_new_reaper(struct task_struct *father)
2  {
3  struct pid_namespace *pid_ns = task_active_pid_ns(father);
4  struct task_struct *thread;
5  thread = father;
6  while_each_thread(father, thread) {
7  if (thread->flags & PF_EXITING)
8  continue;
9  if (unlikely(pid_ns->child_reaper == father))
10 pid_ns->child_reaper = thread;
11 return thread;
12 }
13
14 if (unlikely(pid_ns->child_reaper == father)) {
15 write_unlock_irq(&tasklist_lock);
16 if (unlikely(pid_ns == &init_pid_ns))
17 panic("Attempted to kill init!");
18 zap_pid_ns_processes(pid_ns);
19 write_lock_irq(&tasklist_lock);
20 /*
21 * We can not clear ->child_reaper or leave it alone.
```

```
22 * There may by stealth EXIT_DEAD tasks on ->children,
23 * forget_original_parent() must move them somewhere.
24 */
25 pid_ns->child_reaper = init_pid_ns.child_reaper;
26 }
27 return pid_ns->child_reaper;
28 }
```

Đoạn mã này cố gắng tìm và trả lại một tác vụ khác trong nhóm luồng của tiến trình. Nếu một tác vụ khác không nằm trong nhóm luồng, nó sẽ tìm và trả về tiến trình init. Bây giờ là một tiến trình cha(mẹ) phù hợp mới cho tiên trình con được tìm thấy, mỗi tiến trình con cần phải được đặt vị trí và đặt lại cha mẹ:

```
1 reaper = find_new_reaper(father);
2 list_for_each_entry_safe(p, n, &father->children, sibling) {
3 p->real_parent = reaper;
4 if (p->parent == father) {
5 BUG_ON(p->ptrace);
6 p->parent = p->real_parent;
7 }
8 reparent_thread(p, father);
9 }
```

ptrace_exit_finish() sau đó được gọi là để thực hiện cùng một bồi thường nhưng đối với một danh sách tiến trình con bị đe doạ:

```
1 void exit_ptrace(struct task_struct *tracer)
2 {
3 struct task_struct *p, *n;
4 LIST_HEAD(ptrace_dead);
5 write_lock_irq(&tasklist_lock);
6 list_for_each_entry_safe(p, n, &tracer->ptraced, ptrace_entry) {
7 if (__ptrace_detach(tracer, p))
8 list_add(&p->ptrace_entry, &ptrace_dead);
9 }
10 write_unlock_irq(&tasklist_lock);
11 BUG_ON(!list_empty(&tracer->ptraced));
12 list_for_each_entry_safe(p, n, &ptrace_dead, ptrace_entry) {
13 list_del_init(&p->ptrace_entry);
14 release_task(p);
15 }
16 }
```

Với tiến trình thành công reparented, không có nguy cơ tiến trình zombie đi lạc.Các tiến trình init thường gọi wait() trên con của mình, làm sạch bất kỳ zombies được giao cho nó.

Tài liệu tham khảo

[1] Technical Report No. 2005-499 Scheduling Algorithms for Real-Time Systems of *Arezou Mohammadi and Selim G. Akl*

[2] Scheduling Algorithms for Real-Time Systems of *Fredrik Lindh and Thomas Otnes and Jessica Wennerström*

[3]