A diagram of multithreaded process

Description automatically generated

**Sự khác nhau giữa process và thread**

Bàn về luồng và thread, có rất nhiều bài viết. Các bạn có thể tham khảo tại link sau:

[**https://vivadifferences.com/13-difference-between-process-and-thread-in-os/**](https://vivadifferences.com/13-difference-between-process-and-thread-in-os/)

[**https://www.guru99.com/difference-between-process-and-thread.html**](https://www.guru99.com/difference-between-process-and-thread.html)

[**https://www.geeksforgeeks.org/difference-between-process-and-thread/**](https://www.geeksforgeeks.org/difference-between-process-and-thread/)

|  |  |
| --- | --- |
| **Process** | **Thread** |
| Một bản thể độc lập | Là một phần, là con của process |
| Sử dụng vùng nhớ độc lập | Sử dụng vùng nhớ chung với process cha, có thể chia sẻ vùng nhớ vs các thread có chung process |
| **Khi process chết (có thể bị buộc dừng bởi tác nhân khác), toàn bộ các thread con đều dừng hoat động** | **Khi một thread chết, các thread khác trong cùng process vẫn hoạt động** |
| **Giao tiếp giữa các process rất khó (ví dụ muốn chia sẻ một tham số), phải dùng kĩ thuật giao tiếp đa tiến trình (**[**inter-process communication**](https://vi.wikipedia.org/wiki/Giao_ti%E1%BA%BFp_li%C3%AAn_ti%E1%BA%BFn_tr%C3%ACnh)**)** | **Giao tiếp giữa các thread đơn giản hơn, do có cơ chế chia sẻ bộ nhớ** |
| Chi phí để tạo và hủy một process là rất đắt (Thời gian, cpu và bộ nhớ) | Chi phí để tạo và hủy một thread thì khá rẻ |
| Khó cài đặt | Dễ cài đặt |

[Lập Trình Đa Tiến Trình (codelearn.io)](https://codelearn.io/sharing/lap-trinh-da-tien-trinh)

Nhưng chưa chắc đa luồng đã giúp xử lý nhanh hơn: [Đa luồng nhanh hay chậm? (codelearn.io)](https://codelearn.io/sharing/da-luong-nhanh-hay-cham) nhưng chắc chắn dễ tổ chức và dễ cài đặt hơn (chuyên môn hóa từng nhiệm vụ). Ngoài ra nếu xử lý nhiều nhiệm vụ -> tỉ lệ crash tăng khi chúng ta sử dụng code open-source cũng như lỗi trong quá trình thiết kế phần mềm -> sử dụng multi-process có thể khoanh vùng và xử lý được các process lỗi mà ko ảnh hưởng tới process khác. Trong hình dưới với multi-threading ở bên phải, chỉ cần 1 luồng bị lỗi -> cả chương trình bị crash và phải khởi động lại toàn bộ; trong khi với bên trái, chỉ cần khởi động lại 1 tiến trình bị lỗi.

Bên cạnh đó đa tiến trình có thể cho phép ta sử dụng 1 ngôn ngữ khác nhau cho mỗi tiến trình. Tuy nhiên, nhược điểm của lập trình đa tiến trình là thời gian khởi tạo một tiến trình rất lâu, và hơi tốn CPU lẫn bộ nhớ. Để truyền tham số, giao tiếp liên lạc giữa các tiến trình vs nhau cũng là 1 bài toán khó khăn khác mà người lập trình phải giải quyết. Do đó, lập trình đa tiến trình là **kĩ năng hơi cao cấp**

Chốt lại, nếu chương trình bạn viết ổn định (ko crash), đc phát triển bởi cùng một ngôn ngữ, thì bạn có thể sử dụng đa luồng để đạt hiệu năng cao và dễ cài đặt. Ngược lại, nếu chương trình hay crash và đc phát triển bởi nhiều ngôn ngữ lập trình, bạn nên nghĩ tới hướng lập trình đa tiến trình

**Lập trình đa tiến trình như thế nào?**

Lập trình đa tiến trình là về cơ bản, sẽ có 1 tiến trình chính (main process), sau đó tiến trình này sẽ đẻ ra các tiến trình con để làm việc. Nhiệm vụ của tiến trình chính khá là đơn giản, ko thể có lỗi, thường là nhiệm vụ quản lý và giao việc cho các tiến trình con hơn là trực tiếp xử lý một công việc gì đó

Ngắn gọn lại:

* Tạo ra tiến trình chính, tiến trình chính gọi các tiến trình con
* Tiến trình con nhận nhiệm vụ (thường là nhận nhiệm vụ thông qua giao tiếp vs tiến trình chính), xử lý công việc liên tục
* Tiến trình chính quản lý tiến trình con, nếu thằng con chết/treo/hoàn thành nhiệm vụ thì xử lý (chết thì tạo mới, treo thì stop, ....)

Để lập trình đc, cần biết các kĩ thuật chính như:

* Tiến trình chính tạo ra tiến trình con như thế nào?
* Tiến trình con nhận tham số, công việc từ tiến trình chính ra làm sao?
* Làm sao để tiến trình chính biết chương trình con hoạt động ntn (còn sống hay đã chết, có bị treo hay không, ....)

Chú ý rằng mỗi tiến trình sẽ là một chương trình/không gian chạy độc lập. Do đó, nếu lập trình đa tiến trình, bạn sẽ có nhiều hàm main. Một hàm main thể hiện tiến trình chính, một vài hàm main thể hiện tiến trình phụ. Tất cả các đoạn mã cần đc build thành công và có thể chạy độc lập vs nhau

A screenshot of a screen

Description automatically generated

A white text on a white background

Description automatically generated

A computer code with black text

Description automatically generated

A diagram of a memory process

Description automatically generated

A text on a white background

Description automatically generated

Sau khi xài hàm fork(), lúc này cả 2 process con và cha đều chạy song song với fork() return 0 (process con) và return id của process con (process cha). Khi này ta có thể triển khai bằng cách check if (process\_id\_get\_from\_fork == your process num) để xử lý

Thoát 1 process bằng hàm #include <signal.h> + int kill(childPid,SIGKILL);

Hoặc kill bằng command : kill -9 process\_id

I 🐔: Ví dụ, tôi có 1 biến có giá trị rồi. Bạn thử giải thích giá trị của biến đó sau khi fork() ở cha và con. Nếu 1 người sửa biến đấy thì người còn lại sẽ nhìn thấy giá trị của biến đó như thế nào?

M 🐸: Cơ chế nhân bản của hàm fork() là lazy copy hoặc là copy on write. Tức là khi nó nhân bản thì nó chỉ nhân bản dải địa chỉ địa chỉ bộ nhớ ảo thôi. Còn dải bộ nhớ vật lý thì nó để nguyên. Sau khi nhân bản thì cả cha và con đều trỏ về cùng 1 vùng nhớ vật lý. Tuy nhiên, nếu có người tiến hành chỉnh sửa giá trị trên vùng nhớ vật lý thì nó sẽ tiến hành nhân bản và vùng nhớ vật lý đó và tạo ra 2 vùng nhớ vật lý riêng biệt cho mỗi người. Vì vậy, nếu như 1 người sửa giá trị của 1 biến thì người còn lại sẽ vẫn nhìn thấy giá trị cũ của biến đó

**Process monitor**

**A white background with black text

Description automatically generated**

A close up of a text

Description automatically generated

Pid\_t wait(int \*status); hàm wait ở đây sẽ trả về 1 giá trị dựa vào exit status

A computer screen with blue text

Description automatically generated

* Nếu exit(exitStatus) thành công, thì exit status sẽ ghi vào bit thứ 8-15

VD exit(3) -> no error -> 00000011 | 00000000 = 768

A close-up of a credit card

Description automatically generated

* Nếu bị kill –(num) thì 7 bit từ 0 -6 sẽ chứa giá trị num, còn bit thứ **7** sẽ bật thành 1 tượng trưng cho flag

VD bị kill bởi : kill -8 process\_id -> mã lỗi 136 tương ứng với 00000000 | **1** 0001000

Nếu có nhiều hơn 1 child process, thì khi wait hàm sẽ đợi thông tin từ process nào xong gần nhất, nếu muốn đợi cụ thể process nào thì dùng waitpid(pid, &status, flag like : WNOHANG (this is non-blocking-call to waitpid))

Sử dụng WNOHANG sẽ làm cho process cha không chờ để nhận giá trị exit của process con (ko có thì bỏ qua luôn)

Orphan, Zombie and sleeping process

Orphan process: parent process got killed before child process, and child process still working (ophan but still alive :D if it’s dead, it becomes zombie)

What happens if parent processes terminate before their children? In such a case, the system could be flooded with zombie processes whose process descriptors would stay forever in RAM. As mentioned earlier, this problem is solved by forcing all orphan processes to become children of the init process (PID = 1). In this way, the init process will destroy the zombies while checking for the termination of one of its legitimate children through a wait( )-like system call (Understanding The Linux Kernel 3rd edition)

Zombie process: [Một Zombie Process (hay còn gọi là Defunct Process) là một process đã hoàn thành việc thực thi và đã kết thúc bằng cách sử dụng hệ thống gọi exit, nhưng vẫn còn một mục trong process](https://www.linuxfordevices.com/tutorials/linux/defunct-zombie-process)\_table[1](https://www.linuxfordevices.com/tutorials/linux/defunct-zombie-process). [Zombie Process thường xảy ra cho các child process, vì parent process vẫn cần đọc trạng thái thoát của child process2](https://www.tutorialspoint.com/what-is-zombie-process-in-linux) thông qua wait hoặc waitpid

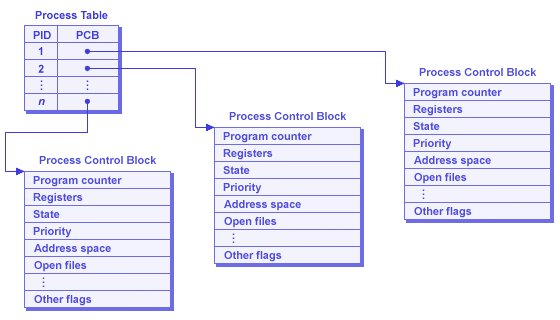
1. **Tại sao lại xảy ra Zombie Process?**
   * [Khi một child process chết, parent process được thông báo để nó có thể dọn dẹp, như giải phóng bộ nhớ3](https://www.howtogeek.com/119815/htg-explains-what-is-a-zombie-process-on-linux/). [Tuy nhiên, nếu parent process không biết về cái chết của child process, thì child process sẽ trở thành zombie3](https://www.howtogeek.com/119815/htg-explains-what-is-a-zombie-process-on-linux/). [Đối với parent process, child process vẫn tồn tại nhưng thực tế thì child process đã chết4](https://itsfoss.com/kill-zombie-process-linux/).
2. **Các tình huống thực tế gây ra Zombie Process và cách giải quyết**

[Zombie Process thường xảy ra khi parent process **không đọc trạng thái exit của child process (thông qua wait hoặc waitpid)** trước khi child process hoàn thành5](https://www.geeksforgeeks.org/difference-between-zombie-orphan-and-daemon-processes/). [Điều này có thể xảy ra nếu parent process bị giết, kết thúc do lỗi, hoặc kết thúc do sự cố5](https://www.geeksforgeeks.org/difference-between-zombie-orphan-and-daemon-processes/).

Để tránh tạo ra Zombie Process, quan trọng là phải đảm bảo rằng các child process được kết thúc đúng cách trước khi parent process kết thúc. [Điều này có thể được thực hiện bằng cách sử dụng hàm wait() hoặc waitpid(), hoặc bằng cách sử dụng một trình xử lý tín hiệu để bắt tín hiệu SIGCHLD và kết thúc child process khi parent process kết thúc5](https://www.geeksforgeeks.org/difference-between-zombie-orphan-and-daemon-processes/).

[Nếu có Zombie Process, bạn không thể giết chúng bằng cách sử dụng tín hiệu SIGKILL như bạn có thể giết các process bình thường – Zombie Process đã chết3](https://www.howtogeek.com/119815/htg-explains-what-is-a-zombie-process-on-linux/). Tuy nhiên, bạn có thể gửi tín hiệu SIGCHLD đến parent process. [Tín hiệu này yêu cầu parent process thực hiện hệ thống gọi wait() và dọn dẹp các child process zombie4](https://itsfoss.com/kill-zombie-process-linux/)

[What is the linux process table ? What does it consist of? - Stack Overflow](https://stackoverflow.com/questions/4880555/what-is-the-linux-process-table-what-does-it-consist-of#:~:text=Process%20table%20is%20a%20data,used%20in%20the%20process%2C%20ect.&text=The%20process%20table%20major%20information,Process%20ID)



Ngoài ra khi process cha không gọi hàm wait hay waitpid thì process con vẫn sẽ trở thành zombie process:

nếu process cha kết thúc trước process con và không gọi hàm wait() hay waitpid(), khi process con hoàn thành xong, nó sẽ trở thành một **process zombie**.

Khi một process con kết thúc, nhưng process cha không kiểm tra trạng thái của nó thông qua hàm wait() hay waitpid(), thì process con sẽ không được hệ điều hành giải phóng hoàn toàn. Thay vào đó, nó sẽ chuyển sang trạng thái zombie, vẫn còn giữ lại trong bảng process để chứa một số thông tin như mã thoát và thời gian chạy.

Điều này có thể gây ra tình trạng lãng phí tài nguyên nếu có quá nhiều process zombie. Để tránh điều này, process cha nên luôn gọi wait() hoặc waitpid() để đảm bảo rằng khi process con kết thúc, tất cả tài nguyên của nó đều được giải phóng hoàn toàn.

Kiểm tra zombie process bằng lệnh trên terminal: ps aux | grep Z



Zombie process sẽ được gắn thẻ Z+ và [zombie]

So sánh orphan và zombie:

Dưới đây là định nghĩa chính xác hơn:

* **Orphan Process (Quá trình mồ côi)**: Đây là quá trình con mà quá trình cha đã kết thúc nhưng quá trình con vẫn đang chạy. Khi quá trình cha kết thúc, quá trình con sẽ được “adopted” bởi quá trình init (process ID 1). Quá trình init sẽ lấy trạng thái kết thúc của quá trình con khi nó kết thúc, ngăn chặn nó trở thành quá trình zombie.
* **Zombie Process (Quá trình zombie)**: Đây là quá trình con đã kết thúc nhưng thông tin về quá trình con vẫn còn trong bảng quá trình vì quá trình cha chưa gọi hàm wait()/waitpid() để lấy trạng thái kết thúc của nó. Quá trình zombie không sử dụng tài nguyên hệ thống nhưng vẫn chiếm một vị trí trong bảng quá trình.
* **Sleeping Process** : mô tả quá trình đang trong trạng thái wait để đợi / xử lý zombie process, còn được gọi là quá trình ngủ

**Advanced Process Programing**

A white text on a white background

Description automatically generated

Trong đó có hàm Execl(): Khi chạy hàm này, nó sẽ chạy file khác và thay thế toàn bộ bộ nhớ ảo của chương trình hiện tại sang cho chương trình mới, những hàm sau dòng này ở chương trình cũ sẽ được bỏ qua

A screenshot of a computer

Description automatically generated

Các hàm execl(), execlp(), execle(), execv(), execvp(), và execve() đều thuộc gia đình hàm exec trong lập trình C. [Chúng đều có chức năng chính là thay thế chương trình hiện tại bằng một chương trình mới1](https://stackoverflow.com/questions/55743496/difference-between-exec-execvp-execl-execv). Tuy nhiên, chúng có sự khác biệt về cách tìm kiếm chương trình, cách truyền tham số, và nguồn gốc của biến môi trường.

1. **execl() và execv()**: Không tìm kiếm PATH. [execl() yêu cầu danh sách tham số, trong khi execv() yêu cầu một mảng tham số2](https://stackoverflow.com/questions/5769734/what-are-the-different-versions-of-exec-used-for-in-c-and-c).
2. **execlp() và execvp()**: Tìm kiếm PATH. [execlp() yêu cầu danh sách tham số, trong khi execvp() yêu cầu một mảng tham số2](https://stackoverflow.com/questions/5769734/what-are-the-different-versions-of-exec-used-for-in-c-and-c).
3. [**execle() và execve()**: Không tìm kiếm PATH và cho phép chỉ định biến môi trường cho quá trình mới3](https://stackoverflow.com/questions/20823371/what-is-the-difference-between-the-functions-of-the-exec-family-of-system-calls). [execle() yêu cầu danh sách tham số, trong khi execve() yêu cầu một mảng tham số2](https://stackoverflow.com/questions/5769734/what-are-the-different-versions-of-exec-used-for-in-c-and-c).

[Điểm chung của các hàm này là chúng đều thay thế chương trình hiện tại bằng một chương trình mới1](https://stackoverflow.com/questions/55743496/difference-between-exec-execvp-execl-execv). [Sự khác biệt chính giữa chúng là cách chúng xử lý tham số và biến môi trường1](https://stackoverflow.com/questions/55743496/difference-between-exec-execvp-execl-execv)[3](https://stackoverflow.com/questions/20823371/what-is-the-difference-between-the-functions-of-the-exec-family-of-system-calls)[2](https://stackoverflow.com/questions/5769734/what-are-the-different-versions-of-exec-used-for-in-c-and-c). [Cụ thể, các hàm có chữ ‘l’ (như execl(), execlp(), execle()) yêu cầu danh sách tham số, trong khi các hàm có chữ ‘v’ (như execv(), execvp(), execve()) yêu cầu một mảng tham số2](https://stackoverflow.com/questions/5769734/what-are-the-different-versions-of-exec-used-for-in-c-and-c). [Các hàm có chữ ‘p’ (như execlp(), execvp()) sẽ tìm kiếm PATH để tìm chương trình4](https://linux.die.net/man/3/execle). [Các hàm có chữ ‘e’ (như execle(), execve()) cho phép chỉ định biến môi trường cho quá trình mới](https://stackoverflow.com/questions/20823371/what-is-the-difference-between-the-functions-of-the-exec-family-of-system-calls)

A screenshot of a computer program

Description automatically generated

Execv : truyền vào 1 array (nhớ kết thúc = NULL)

A screenshot of a computer program

Description automatically generatedexecl: truyền vào 1 list (nhớ kết thúc = NULL)

A screenshot of a computer program

Description automatically generated

Execve giống với execv nhưng có thể thay đổi enviroment variable của 2nd program. Như hình trên ta truyền thêm các env var mới từ chuỗi \*envVec[]

!để thêm vào biến environ, ta cần cú pháp VARNAME = var-value

Thứ tự xử lý sẽ là [sau khi execve được gọi, toàn bộ vùng nhớ của quá trình gọi (bao gồm cả vùng nhớ chứa argv và envp) sẽ bị ghi đè bởi chương trình mới2](https://stackoverflow.com/questions/12016441/execve-and-environment-variables). [Tuy nhiên, kernel sẽ sao chép argv và envp trước khi execve được gọi, và sau đó chép chúng vào vùng nhớ của chương trình mới](https://stackoverflow.com/questions/12016441/execve-and-environment-variables)

Chúng ta có thể kết hợp exec() và fork() để thực hiện đa tiến trình với mỗi tiến trình là 1 main khác nhau -> thực hiện nhiều công việc cùng lúc

A white paper with black text

Description automatically generated

Khi một hàm fork() được thực hiện trong một chương trình, một quá trình con (child process) mới được tạo ra. Quá trình con này là một bản sao của quá trình cha (parent process) từ đó nó được tạo ra. Điều này có nghĩa là tất cả các mô tả tệp (file descriptors) mà quá trình cha sở hữu đều được sao chép cho quá trình con.

Mô tả tệp là một con số được hệ thống sử dụng để tham chiếu đến một tệp hoặc một dòng dữ liệu (stream). Khi quá trình con nhận được các mô tả tệp từ quá trình cha, cả hai quá trình đều có thể đọc và ghi vào các tệp này. Tuy nhiên, họ không chia sẻ vị trí con trỏ tệp - nghĩa là, nếu một quá trình di chuyển con trỏ tệp, quá trình kia sẽ không thấy sự thay đổi.

Dưới đây là một ví dụ cụ thể:

#include <stdio.h>

#include <unistd.h>

int main() {

int fd[2];

pipe(fd);

if (fork() == 0) {

// This is the child process

close(fd[0]); // Close the read end of the pipe

write(fd[1], "Hello from child", 16);

} else {

// This is the parent process

char buffer[20];

close(fd[1]); // Close the write end of the pipe

read(fd[0], buffer, sizeof(buffer));

printf("%s\n", buffer);

}

return 0;

}

Trong ví dụ trên, chúng ta tạo ra một (pipe) và sau đó gọi fork(). Quá trình con viết vào phần cuối ghi của pipe , và quá trình cha đọc từ phần cuối đọc của pipe (cái này do chúng ta tự quy định). Việc close fd[0] giúp cho code an toàn và dễ hiểu hơn (nếu đóng fd[0] và write fd[1] thì process đó đảm nhận việc ghi và không đọc, nếu đọc sẽ dời con trỏ trong file descriptor)

Hàm pipe nhận vào đúng 1 mảng chứa 2 file descriptor fd[2] trong đó *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe theo cơ chế FIFO

Tại sao phải sử dụng pipe ?

Khi một pipe được tạo ra bằng hàm pipe(), một buffer được tạo trong **không gian kernel**. Khi một tiến trình ghi vào đầu ghi của pipe (fd[1]), dữ liệu được chuyển vào buffer này. Khi một tiến trình khác đọc từ đầu đọc của pipe (fd[0]), nó nhận dữ liệu từ buffer này.

Trong khi đó, khi một tiến trình đọc hoặc ghi vào một tệp thông thường (không phải pipe), nó thường sử dụng một buffer trong không gian người dùng của chính nó. Dữ liệu được đọc vào buffer này hoặc được ghi từ buffer này vào tệp. Nhưng mỗi tiến trình có buffer riêng của nó, và các thao tác đọc và ghi không ảnh hưởng đến buffer của các tiến trình khác.

**Review**

A screenshot of a computer

Description automatically generated

A diagram of a table

Description automatically generated

Trong hệ thống tập tin Unix, mọi truy cập đến dữ liệu tệp tin đều thông qua inode. Mỗi tệp tin được đại diện bởi một inode duy nhất, và các thao tác đọc, ghi, thay đổi quyền truy cập, thậm chí cả việc xóa tệp tin đều được thực hiện thông qua inode tương ứng.

Khi bạn muốn truy cập một tệp tin, hệ điều hành sẽ sử dụng đường dẫn tệp tin để tìm inode của nó trong bảng inode (inode table). Sau đó, thông qua inode, hệ thống có thể biết được nơi lưu trữ dữ liệu thực sự của tệp tin trên đĩa cứng và các thông tin liên quan khác.

Do đó, mặc dù trong mã nguồn của chương trình hoặc ứng dụng, chúng ta sử dụng đường dẫn tệp tin để thao tác với tệp tin, nhưng trong hệ thống tập tin, mọi hoạt động thực sự đều dựa trên inode.

**Tương tự ta có process table**

A diagram of a process

Description automatically generated

PCB là process control block

A white text on a black background

Description automatically generated

Why we need to have process control block ? Chúng ta cần Process Control Block (PCB) để lưu trữ tất cả thông tin cần thiết về một tiến trình trong hệ thống đa tiến trình như Linux. Khi CPU chuyển từ một tiến trình này sang một tiến trình khác, PCB giúp hệ thống lưu trữ và khôi phục trạng thái của tiến trình.

Ví dụ, giả sử bạn đang chạy một ứng dụng như trình duyệt web và đồng thời bạn muốn mở một ứng dụng khác như trình soạn thảo văn bản. Trong trường hợp này, CPU sẽ chuyển đổi giữa hai tiến trình này. Khi CPU chuyển từ trình duyệt web sang trình soạn thảo văn bản, nó cần lưu trữ trạng thái hiện tại của trình duyệt web (như vị trí con trỏ, trạng thái của các tab, v.v.) vào PCB tương ứng. Khi CPU quay lại trình duyệt web, nó sẽ sử dụng thông tin trong PCB để khôi phục trạng thái của trình duyệt web như nó đã được để lại.

Đây chỉ là một ví dụ đơn giản để giải thích tại sao chúng ta cần PCB. Trên thực tế, PCB chứa nhiều thông tin hơn về tiến trình, bao gồm trạng thái tiến trình (chạy, chờ, v.v.), giá trị của các thanh ghi CPU, bộ đếm chương trình (để biết tiến trình đang thực hiện lệnh nào), độ ưu tiên của tiến trình, v.v. Trong Linux, PCB được tham chiếu bởi cấu trúc dữ liệu được gọi là task\_struct.