

BÁO CÁO LINUX SYSTEM PROGRAMING

VŨ NGỌC CƯỜNG

Mã nguồn: <https://github.com/cuongvungoc/Linux-OS-Learning.git>

MỤC LỤC

DANH SÁCH HÌNH VẼ	5
DANH SÁCH BẢNG BIỂU.....	7
CHƯƠNG 1. LINUX KERNEL	8
1.1 Linux Kernel.....	8
CHƯƠNG 2. SYSTEM PROGRAMING CONCEPT.....	10
2.1 System call.....	10
2.2 Quá trình thực thi system call	10
2.3 Library function	12
CHƯƠNG 3. FILE OPERATIONS.....	14
3.1 File descriptor (fd)	14
3.2 File permissions	14
3.3 Types of files	14
3.4 Open a file: Open() – system call.....	15
3.5 Reading from a file – read() system call.....	17
3.6 Writing to a file – writes system call.....	18
3.7 Closing a file – close() system call.....	18
3.8 Changing the file offset – lseek() – system call.....	19
CHƯƠNG 4. PROCESS	20
4.1 Type of Linux Commands:	20
4.2 Process properties	20
4.3 Type of Processes	20
4.3.1 Deamon	20
4.3.2 Zombie	21
4.3.3 Orphan	21
4.3.4 Thread vs Process	21
4.3.5 Forgeground and background process.....	21
4.4 Process vs program	21
4.5 Process ID	22
4.6 Process state	23
4.7 Process memory layout	24
4.7.1 Stack và Stack frame	25
4.7.2 Argc và argv	26
4.7.3 Environment variable	26
4.8 Thao tác với process.....	26
4.8.1 Tạo process mới.....	26

4.8.2 Chạy process: exec family	27
4.8.3 Terminating a process	27
4.8.4 System call wait()	28
4.9 Virtual memory in Linux	29
4.9.1 Page frame	30
4.9.2 Page table	30
4.10 Memory mapping	31
4.10.1 Creating a mapping	33
4.10.2 File mapping	34
4.10.3 Private file mapping	35
4.10.4 Shared file mapping	36
CHƯƠNG 5. SIGNAL	38
5.1 Concept and overview	38
5.2 Các nguồn gửi signals:	38
5.3 Các cách gửi signal đến process	38
5.4 Receive and handle signal	39
5.5 Most used signal explanations	40
5.6 Basic signal management	40
5.7 Sending a signal	41
CHƯƠNG 6. THREAD	42
6.1 Pros and cons	42
6.2 Thread vs Process	42
6.3 Thread ID	42
6.4 Create a new process	43
6.5 Thread terminate	43
6.6 Thread management	43
6.6.1 Joinable thread	43
6.6.2 Detached thread	46
6.7 Thread synchronization	48
6.7.1 Critical section	48
6.7.2 Mutex – Mutual exclusion	48
6.7.3 Mutex operation	49
6.7.4 Mutex initialization	49
6.7.5 Mutex Deadlock	50
6.7.6 Condition Variable	51
6.7.7 Condition variable initialization	51
CHƯƠNG 7. INTER PROCESS COMMUNICATION	55
7.1 PIPES	56
7.1.1 Creating and using pipes	56
7.1.2 SIGPIPE signal in PIPES	57

7.1.3 Synchronization in Pipe	57
7.2 FIFOs	58
7.2.1 Create a FIFO	58
7.3 POSIX message queue	60
7.3.1 Message queue operation	61
7.3.2 Open a message queue.....	61
7.3.3 Removing a message queue	62
7.4 POSIX Semaphore	62
7.4.1 Open a named semaphore	63
7.4.2 Closing a semaphore	64
7.4.3 Removing a semaphore	64
7.4.4 Unnamed semaphores.....	66
7.5 POSIX – Shared Memory	68
7.5.1 To use a POSIX shared memory object.....	69
7.5.2 Creating shared memory object.....	69
7.5.3 Setting shared memory size	70
7.5.4 Mapping shared memory to process virtual memory	70
TÀI LIỆU THAM KHẢO	72

DANH SÁCH HÌNH VẼ

Hình 1-1 Linux Kernel	8
Hình 2-1 Step in the execution of a system call	12
Hình 2-2 Library function interface	13
Hình 3-1 Ví dụ về open system call	16
Hình 3-2 Ví dụ về read – system call	18
Hình 4-1 List of all processes	22
Hình 4-2 Process state	23
Hình 4-3 Memory layout of a process	24
Hình 4-4 Page table in virtual memory	31
Hình 4-5 File memory mapping	33
Hình 4-6 File memory mapping	35
Hình 4-7 Shared memory mapping	36
Hình 5-1 List of signals	38
Hình 5-2 Signal delivery and handler execution	39
Hình 6-1 Ví dụ pthread_join.....	45
Hình 6-2 Ví dụ về pthread_detach.....	48
Hình 6-3 Using a mutex to protect a critical section.....	48
Hình 6-4 Trường hợp deadlock 1	50
Hình 6-5 Trường hợp deadlock 2	51
Hình 6-6 Ví dụ về condition variable	54
Hình 7-1 A taxonomy of UNIX IPC facilities	55
Hình 7-2 Using pipe to connect two process.....	56
Hình 7-3 process file descriptors after creating a pipe	56
Hình 7-4 Setting up a pipe to transfer data from a parent to child	57
Hình 7-5 Ví dụ về PIPE.....	57
Hình 7-6 Read_fifo.c	59
Hình 7-7 Write_fifo.c	60

Hình 7-8 Minh hoạ sử dụng FIFO	60
Hình 7-9 Ví dụ sử dụng message queue	62
Hình 7-10 Using semaphore to synchronize two process	65
Hình 7-11 Ví dụ sử dụng named semaphore	66
Hình 7-12 Ví dụ sử dụng unnamed semaphore	68
Hình 7-13 Shared memory	68
Hình 7-14 Ví dụ sử dụng shared memory	71

DANH SÁCH BẢNG BIỂU

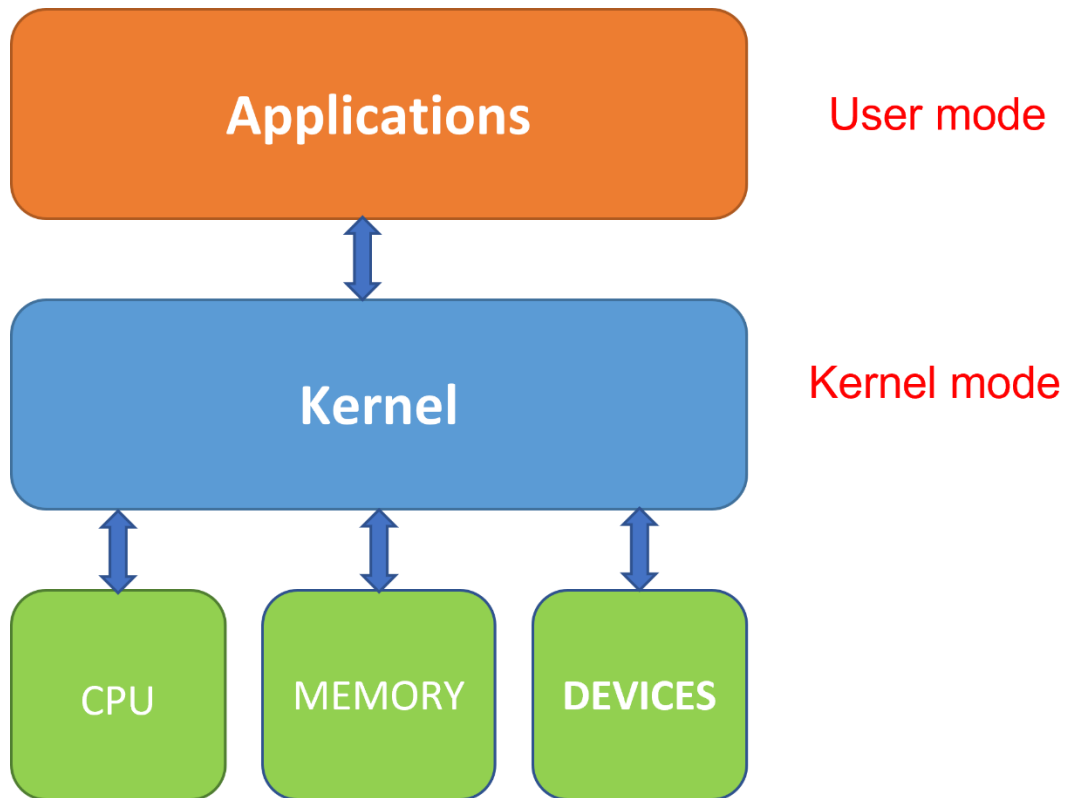
Bảng 3-1 File descriptor	14
Bảng 3-2 Oflag for open system call	15
Bảng 4-1 Exec family	27
Bảng 4-2 Memory protection values	33
Bảng 7-1 Bit value for mq_open() flag argument	61

CHƯƠNG 1. LINUX KERNEL

1.1 Linux Kernel

Linux là một hệ điều hành máy tính mã nguồn mở (open source) và tự do (free) dạng Unix-like (giống kiểu hệ điều hành Unix) được xây dựng trên nền của nhân Linux (Linux kernel).

Linux Kernel được ra đời bởi Linus Torvalds năm 1991 chính là mảnh ghép quan trọng còn thiếu có thể kết hợp với dự án GNU để trở thành một hệ điều hành hoàn chỉnh và lớn mạnh như ngày nay.



Hình 1-1 Linux Kernel

Linux kernel is a computer program at core of Operating System.

Kernal always resides in Memory.

Facilitaes interaction between hardware and software.

Handles the rest of startup, input and output from software.

Handles memory and hardware peripherals.

Advanced function:

File operations

- Memory management

- Virtual memory management
- Process management
- Thread management
- Inter process communications

CHƯƠNG 2. SYSTEM PROGRAMING CONCEPT

2.1 System call

Trong kiến trúc Linux, không gian bộ nhớ được chia thành hai phần là user space và kernel space. Theo đó, cũng tồn tại hai chế độ (mode) là user mode và kernel mode. Các chỉ lệnh được gọi từ chương trình như đóng mở file (fopen, fclose), hoặc in một thông tin (printf) chỉ có thể thực thi và truy cập vùng nhớ ở tầng user mà không được truy cập vùng nhớ của kernel.

Cơ chế phân tách user space với kernel space và không cho phép người dùng tự ý truy cập tài nguyên của kernel giúp quản lý và bảo vệ kernel cũng như toàn bộ hệ thống.

Vấn đề đặt ra là làm cách nào để user gọi xuống kernel hay thao tác điều khiển các device driver? Để đáp ứng yêu cầu này, kernel cung cấp cho user space các API (còn gọi là các dịch vụ) là system call.

System call là một cửa ngõ vào kernel, cho phép tiến trình trên tầng user yêu cầu kernel thực thi một vài tác vụ cho mình. Những dịch vụ này có thể là tạo một tiến trình mới (fork), thực thi I/O (read, write), hoặc tạo ra một pipe cho giao tiếp liên tiến trình (IPC).

Một số lưu ý về system call:

- Khi một tiến trình gọi một system call, CPU sẽ chuyển từ chế độ user mode sang kernel mode, điều này cho phép CPU truy cập các vùng nhớ và thực hiện các chỉ lệnh của kernel.
- Mỗi system call được kernel định danh bằng một số duy nhất. Tiến trình trên tầng user không biết đến các số này, thay vào đó, nó gọi một system call bằng tên hàm (ví dụ như open(), read(...)).
- Mỗi system call có thể có một số tham số truyền để cung cấp thông tin từ user truyền xuống kernel và ngược lại.

2.2 Quá trình thực thi system call

System call được gọi từ ứng dụng người dùng bằng hai cách:

- Trực tiếp gọi system call
- Code người dùng gọi library function, và nó chuyển sang gọi system call nếu được yêu cầu.

Dưới góc nhìn của lập trình viên, thực thi một system call cũng giống như gọi một hàm C thông thường. Tuy nhiên đằng sau đó còn rất nhiều bước được thực hiện từ user space xuống kernel space.

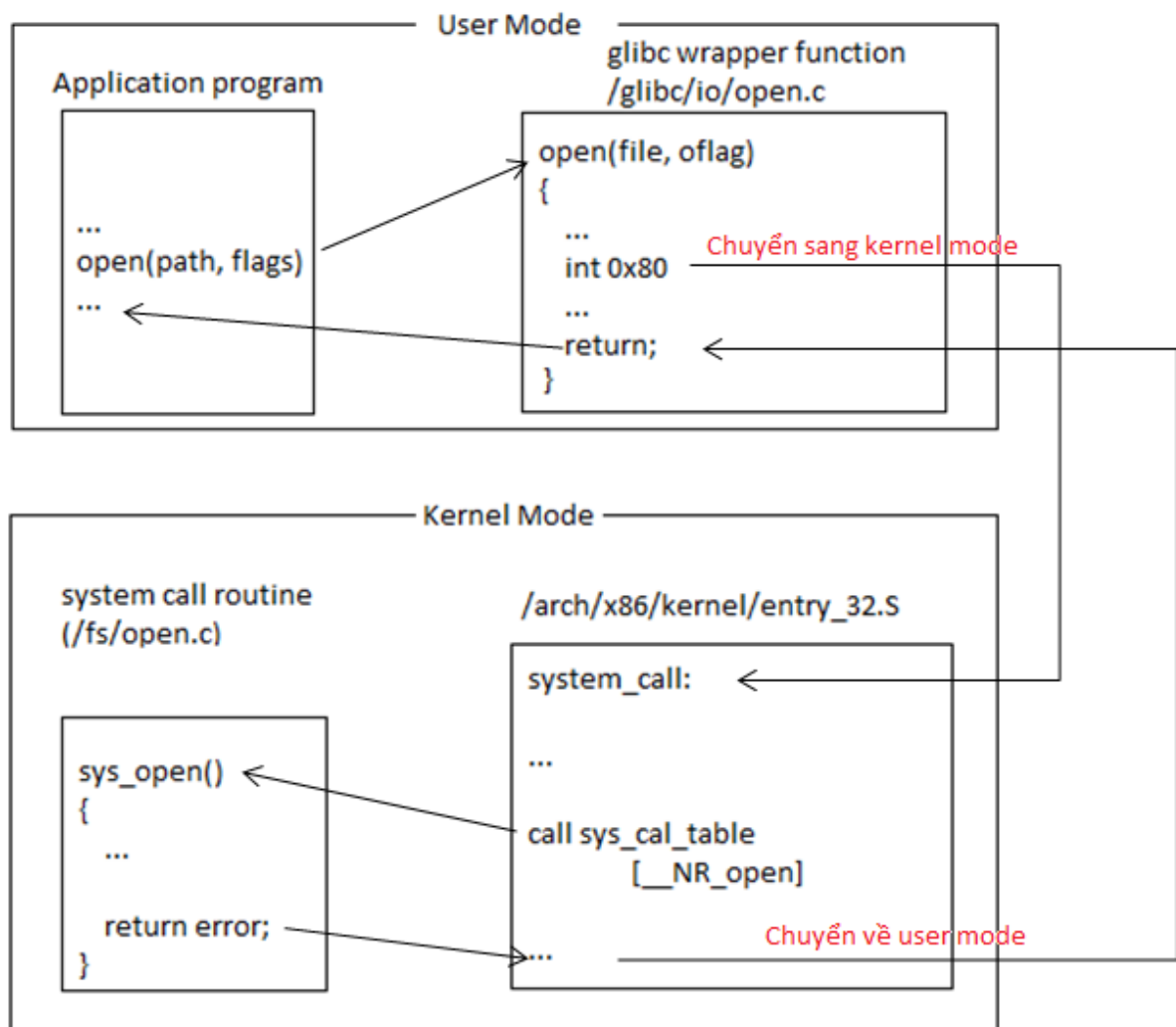
Ví dụ về các bước thực thi system call

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

```
FILE *fopen(const char *filename, const char *mode)
```

Hàm `fopen()` là một hàm thư viện (wrapper function) được dùng để thực thi việc chuyển xuống kernel mode và yêu cầu kernel mở một file dưới kernel có đường dẫn là “filename” với chế độ “mode”. Hàm `fopen()` được triển khai bằng cách gọi system call `open()`, với các bước:

1. Hàm wrapper copy các đối số (trong trường hợp này là "filename" và "mode") vào các thanh ghi, nơi mà các lệnh của luồng thực thi system call sẽ đọc và sử dụng được.
2. Hàm wrapper sao chép số system call vào một thanh ghi của CPU (%eax). Ví dụ system call number của `open()` là 5, hàm này sẽ sao chép giá trị 5 vào thanh ghi %eax.
3. Hàm wrapper thực hiện một chỉ lệnh máy gọi là trap machine instruction để chuyển chế độ CPU từ user mode sang kernel mode. Chỉ lệnh này có thể là một ngắt mềm (software interrupt) với số ngắt (interrupt number) là 0x80 (int 0x80) hoặc chỉ lệnh SYSENTER (trong các kiến trúc Intel gần đây) hoặc chỉ lệnh SYSCALL (trong AMD)
4. Kernel gọi đến luồng `system_call` (nằm trong file `arch/x86/entry_32.S`), tại đây nó sẽ làm các công việc: copy giá trị các đối số trong các thanh ghi mà đã copy vào trong bước 1 vào kernel stack; kiểm tra tính hợp lệ của các đối số; gọi đến system call service routine thích hợp bằng cách tra cứu số system call được sao chép ở bước 2 trong bảng system call routine (`sys_call_table`); gửi kết quả trả về lên cho hàm wrapper và cuối cùng là chuyển chế độ của CPU từ kernel mode sang user mode.
5. Hàm wrapper trả về giá trị là một số nguyên cho hàm gọi nó để thông báo lời gọi system call có thành công không. Nếu system call trả về giá trị lỗi, hàm wrapper sẽ set giá trị cho một biến toàn cục “`errno`” từ giá trị lỗi này.



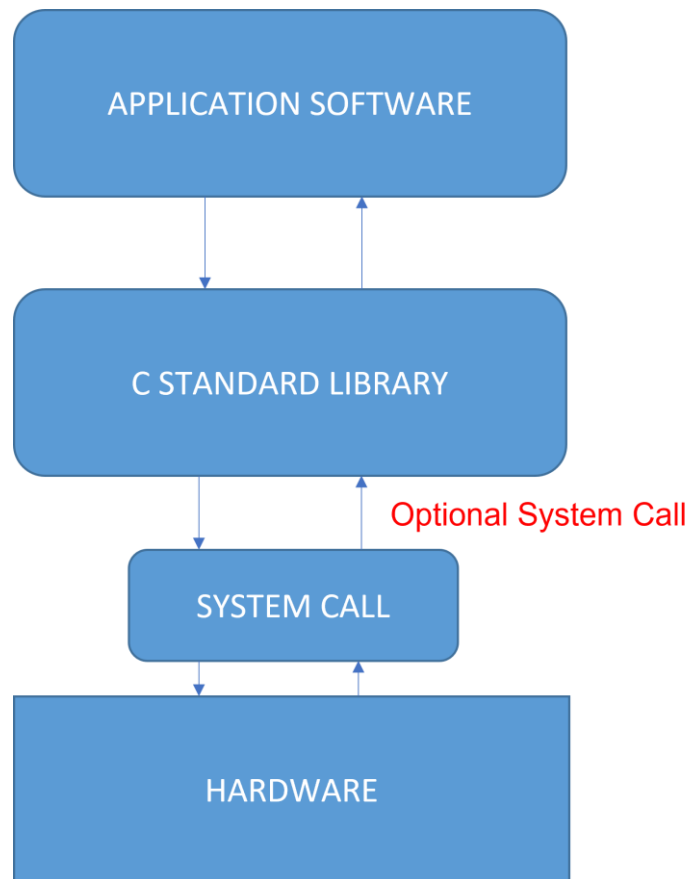
Hình 2-1 Step in the execution of a system call

2.3 Library function

Thư viện tiêu chuẩn C (C standard library - libc) có thể coi là trái tim của các ứng dụng Linux, bao gồm các hàm tiêu chuẩn từ cơ bản từ mở một file, thao tác string hay đến việc gọi system call xuống Linux kernel.

Rất nhiều hàm thư viện không tạo ra một system call nào như các hàm thao tác string (`strcmp`, `strcpy`...), nhưng có nhiều hàm thư viện là một hàm wrapper, trực tiếp gọi một system call. Lý do là các system call được tạo ra để giúp ứng dụng giao tiếp với kernel nên rất nhiều trong số đó có thể không thân thiện với lập trình viên. Đơn cử như hàm `printf()` để hiển thị một dòng text lên màn hình với các định dạng khác nhau mà tất cả chúng ta đều dùng mà không hề gặp một chút khó khăn. Nhưng để làm được việc này, hàm `printf()` phải làm thêm các bước: chuyển đổi nội dung text đó sang định dạng mà người dùng mong muốn (ví dụ `%d` hay `%s`), rồi dùng system call `write()` để ghi nội dung đã chuyển đổi đó vào mô tả file của màn hình hiển thị. Tương tự như việc sử dụng hàm `malloc()` và `free()` để cấp phát và giải phóng vùng nhớ cho một biến sẽ đơn giản hơn rất nhiều so với việc sử dụng system call `brk()`.

Vì vậy, thư viện C giúp lập trình viên chỉ cần gọi các hàm thân thuộc, thay vì phải hiểu system call nào được gọi hay kernel thực thi system call đó như thế nào.



Hình 2-2 Library function interface

C standard library được sử dụng phổ biến nhất hiện nay là GNU C library (glibc). Ngoài ra còn có thư viện khác như uClibc, diet libc với ưu điểm là yêu cầu bộ nhớ nhỏ hơn glibc nên được dùng trong một số thiết bị nhúng.

CHƯƠNG 3. FILE OPERATIONS

3.1 File descriptor (fd)

All system calls for performing I/O refer to open files using a file descriptor, a nonnegative integer

All file related operation are performed via fd

Bảng 3-1 File descriptor

Fd	Description	Posix name	Stido stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDIN_FILENO	stdout
2	Standard error	STDIN_FILENO	stderr

3.2 File permissions

Each file have below file permissions

Read (r), write (w), execute (x)

Read has '4' units (2 power 2)

Write has '2' units (2 power 1)

Execute has '1' units (2 power 0)

Each file can be accessed by different users as follows:

User, group, other

Each user type can have r/w/x access for file

3.3 Types of files

- -: regular file (data file)
- d: directory
- c: character device file
- b: block device file
- s: local socket file
- p: named pipe
- l: symbolic link

3.4 Open a file: Open() – system call

The open() system call either opens an existing file or creates and opens a new file.

```
#include <sys/stat.h>
```

```
#include <fnctl.h>
```

```
int open(const char *pathname, int flags, ... /* mode_t mode */);
```

Returns file descriptor on success, or –1 on error

The file to be opened is identified by the pathname argument. If pathname is a symbolic link, it is dereferenced.

On success, open() returns a file descriptor that is used to refer to the file in subsequent system calls. If an error occurs, open() returns –1 and errno is set accordingly.

The errors that occur which are most common are EACCES, EEXIST, EISDIR

The flags should contain atleast one of O_RDONLY, O_WRONLY, O_RDWR along with other flags

Bảng 3-2 Oflag for open system call

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read and write
O_CREAT	Create file if it doesn't exist

When open() is used to create a new file.

If the open() call doesn't specify O_CREAT, mode can be omitted.

Mode: Specifies the file creation with access permissions

To perform any operation like reading/writing file, need to open file and provide a handle to kernel. Any further operation on this file will be done using this handle

```

LSP > C open1.c > ...
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <errno.h>
6
7  void main(){
8      int fd;
9      fd = open("haha.txt", O_WRONLY);
10     if (fd == -1) {
11         printf("\nopen() was failed - errno = (%d)\n", errno);
12         perror("ERROR:");
13     }
14     else {
15         printf("\nopen() system call executed sucessfully\n");
16     }
17 }
18
cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP$ ./open
open() was failed - errno = (2)
ERROR:: No such file or directory
● cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP$ nano haha.txt
ⓧ cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP$ ./open
open() system call executed sucessfully
○ cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP$ █

```

Hình 3-1 Ví dụ về open system call

3.5 Reading from a file – read() system call

The read() system call reads data from the open file referred to by the descriptor fd.

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

```
ssize_t read(int fd, void *buffer, size_t count);
```

Returns number of bytes read, 0 on EOF, or -1 on error

Size_t: unsigned integer

Buffer: Address of the memory buffer into which the input data is to be placed. This buffer must be at least 'count' by long.

Note: Read() does not allocate any memory, user must allocate memory and pass to the read()

A successfully call to read() return the number of bytes actually read.

- 0 if end-of-file is encountered
- -1 if error

Note: Read() system calls are applited on files like regular files, PIPEs, Socket, FIFO

```

18     sz = read(fd, buf, 10);
19     printf("call 1 - called read. fd = %d,  %d bytes  were read.\n", fd, sz);
20     buf[sz] = '\0';
21     printf("Read bytes are as follows: \n<%s>\n", buf);
22
23     printf("\n Note the next set of bytes read from file, it is continuos\n");
24
25     sz = read(fd, buf, 11);
26     printf("call 2 - called read. fd = %d,  %d bytes  were read.\n", fd, sz);
27     buf[sz] = '\0';
28     printf("Read bytes are as follows:\n<%s>\n", buf);
29
30     sz = read(fd, buf, 10);
31     printf("call 3 - called read. fd = %d,  %d bytes  were read.\n", fd, sz);
32     if (sz == 0)
33     {
34         printf("EOF Reached\n");
35     }
36
37     close(fd);

```

```

● cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/FileOperation$ ./read
call 1 - called read. fd = 3,  5 bytes  were read.
Read bytes are as follows:
<haha
>

Note the next set of bytes read from file, it is continuos
call 2 - called read. fd = 3,  0 bytes  were read.
Read bytes are as follows:
<>
call 3 - called read. fd = 3,  0 bytes  were read.
EOF Reached

```

Hình 3-2 Ví dụ về read – system call

3.6 Writing to a file – writes system call

The write() system call writes data to an open file.

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

```
ssize_t write(int fd, void *buffer, size_t count);
```

Returns number of bytes written, or –1 on error

Argument: Same as read()

3.7 Closing a file – close() system call

The close() system call closes an open file descriptor, freeing it for subsequent reuse by the process. When a process terminates, all of its open file descriptors are automatically closed.

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

```
int close(int fd);
```

Returns 0 on success, or -1 on error

Error:

- Close the unopened fd
- Close the fd twice

3.8 Changing the file offset – lseek() – system call

A file can be considered as a continuous set of bytes

There is an internal indicator present, which points to the offset bytes of the file. This offset is used to read/write the next set of bytes/data from file.

This indicator is updated when we do any file operation like read() or write().

Lseek system call is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.

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

```
off_t lseek(int fd, off_t offset, int whence);
```

Returns new file offset if successful, or -1 on error

Fd: file descriptor

Offset: offset of the pointer (measured in bytes)

Whence: method in which the offset is to be interpreted

SEEK_SET: File offset is set to offset bytes

SEEK_CUR: File offset is set to its current location plus offset

SEEK_END: Size of file + offset bytes

Note:

- Always open the file (open() system call) before performing the file operations (get fd)
- Once the file operation is done → close() system call

CHƯƠNG 4. PROCESS

A running instance of a program is called a process and it runs in its own memory space. Each time you execute a command, a new process starts.

- A process is an active entity as opposed to a program, which is considered to be a passive entity.
- A new process is created only when running an executable file (not when running Shell builtin commands).

4.1 Type of Linux Commands:

- Executable file on the disk
- Shell builtin commands

4.2 Process properties

- PID (Process ID) - a unique positive integer number
- User
- Group
- Priority / Nice

4.3 Type of Processes

- Parent
- Child
- Daemon
- Zombie (defunct)
- Orphan

All the process in OS are created when another process execute `fork()` system call

➔ First process – pid = 1 , init

Process used `fork` system call – parent process create process is its child

4.3.1 Daemon

- Background process
- Contains names that finish with 'd', Ex: `sshd`

- Linux begins daemons at starting time

4.3.2 Zombie

- OS maintains a table – associates every process to the data necessary for its functioning
- When a process terminates its execution the OS, releases most of the resources and information related to that process, a terminated process whose data has not been collected is called zombie
- Remove quickly from memory and don't use any of the system resources

4.3.3 Orphan

- Whose parent process has finished or terminated though it remains running itself

4.3.4 Thread vs Process

- Multiple threads can exist within the same process and they share resources such as memory
- Other process does not share resources

Example:

Text editor: process

Autosave: thread

4.3.5 Foreground and background process

Foreground:

- Started by user and not by system services
- While they are running, user cannot start another process in same terminal
- Default: All process run in foreground
- Input: Keyboard -> output: terminal

Background:

- Can start other process in same terminal
- Use: &

4.4 Process vs program

Process: active entity

Program: positive entity – just a file on the disk

4.5 Process ID

Process ID is an unique integer

```
pid_t getpid(void);
```

In Linux startup sequence

Process 0: First process – swapper process

Process 1: Init process: create and monitor set of other process

Init process become parent of any orphan process

In Linux kernel, PID < 32767 (32 bit)

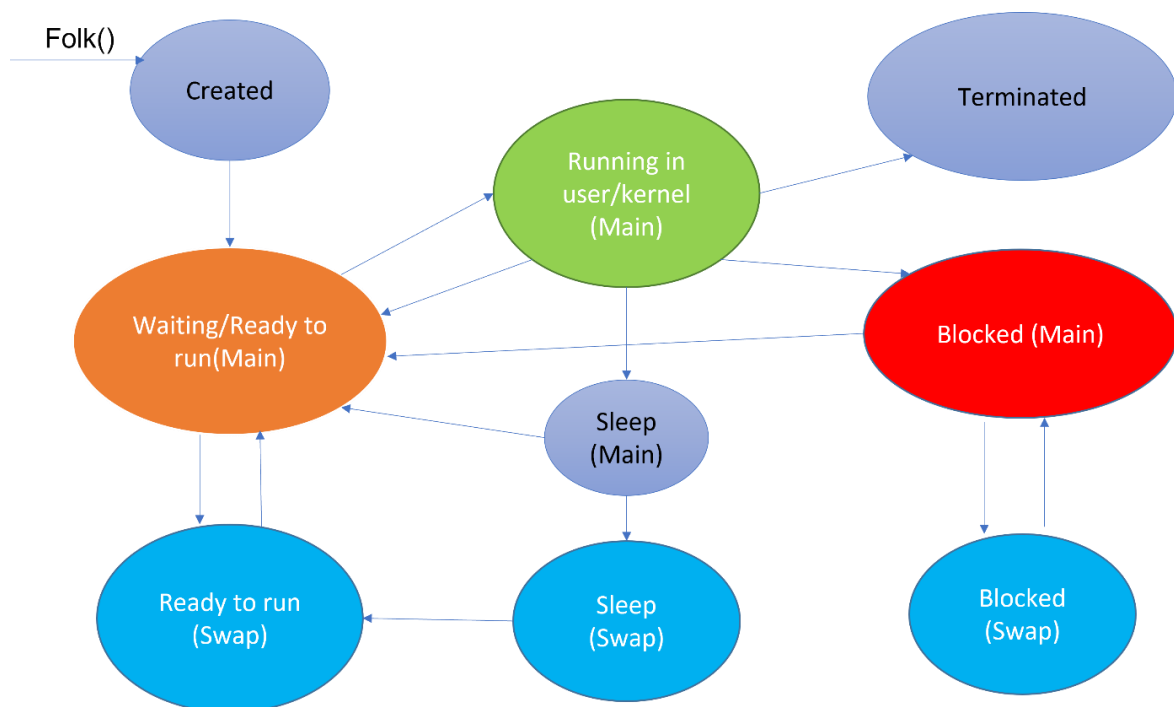
Display all process: **ps aux**

```
cuongvn@cuongvn:~$ ps aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root             1   0.5  0.3 167952 13308 ?        Ss   08:57   0:05 /sbin/init au
root             2   0.0  0.0      0     0 ?        S    08:57   0:00 [kthreadd]
root             3   0.0  0.0      0     0 ?        I<   08:57   0:00 [rcu_gp]
root             4   0.0  0.0      0     0 ?        I<   08:57   0:00 [rcu_par_gp]
root             5   0.0  0.0      0     0 ?        I<   08:57   0:00 [slub_flushwq
root             6   0.0  0.0      0     0 ?        I<   08:57   0:00 [netns]
root             7   0.0  0.0      0     0 ?        I    08:57   0:00 [kworker/0:0-
root             8   0.0  0.0      0     0 ?        I<   08:57   0:00 [kworker/0:0H
root            10   0.0  0.0      0     0 ?        I<   08:57   0:00 [mm_percpu_wq
root            11   0.0  0.0      0     0 ?        S    08:57   0:00 [rcu_tasks_ru
root            12   0.0  0.0      0     0 ?        S    08:57   0:00 [rcu_tasks_tr
root            13   0.0  0.0      0     0 ?        S    08:57   0:00 [ksoftirqd/0]
root            14   0.1  0.0      0     0 ?        I    08:57   0:01 [rcu_sched]
root            15   0.0  0.0      0     0 ?        S    08:57   0:00 [migration/0]
root            16   0.0  0.0      0     0 ?        S    08:57   0:00 [idle_inject/
root            18   0.0  0.0      0     0 ?        S    08:57   0:00 [cpuhp/0]
root            19   0.0  0.0      0     0 ?        S    08:57   0:00 [cpuhp/1]
root            20   0.0  0.0      0     0 ?        S    08:57   0:00 [idle_inject/
root            21   0.0  0.0      0     0 ?        S    08:57   0:00 [migration/1]
root            22   0.0  0.0      0     0 ?        S    08:57   0:00 [ksoftirqd/1]
root            24   0.0  0.0      0     0 ?        I<   08:57   0:00 [kworker/1:0H
root            25   0.0  0.0      0     0 ?        S    08:57   0:00 [kdevtmpfs]
root            26   0.0  0.0      0     0 ?        I<   08:57   0:00 [inet_frag_wq
root            27   0.0  0.0      0     0 ?        S    08:57   0:00 [kauditd]
root            29   0.0  0.0      0     0 ?        S    08:57   0:00 [khungtaskd]
root            30   0.0  0.0      0     0 ?        S    08:57   0:00 [oom_reaper]
root            31   0.0  0.0      0     0 ?        I<   08:57   0:00 [writeback]
root            32   0.0  0.0      0     0 ?        S    08:57   0:00 [kcompactd0]
root            33   0.0  0.0      0     0 ?        SN   08:57   0:00 [ksmd]
root            34   0.0  0.0      0     0 ?        SN   08:57   0:00 [khugepaged]
root            39   0.1  0.0      0     0 ?        I    08:57   0:01 [kworker/1:1-
root            81   0.0  0.0      0     0 ?        I<   08:57   0:00 [kintegrityd]
root            82   0.0  0.0      0     0 ?        I<   08:57   0:00 [kblockd]
```

Hình 4-1 List of all processes

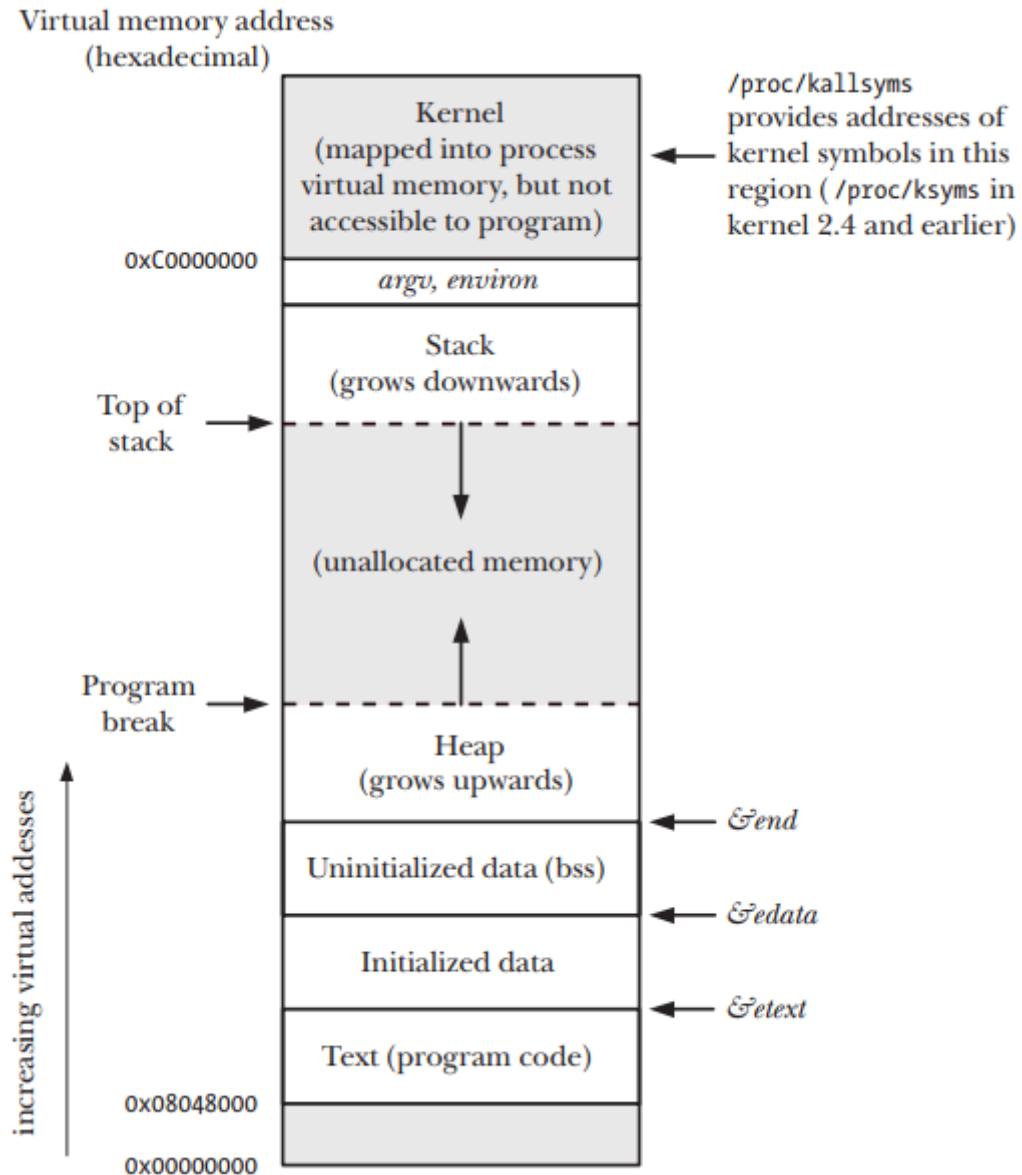
4.6 Process state

- Create: Using `fork()` system call
- Running: Running in main memory
- Ready to run in main memory
- Ready to run in swap memory
- Sleep in main memory
- Sleep in swap memory
- Blocked in main memory
- Blocked in swap memory
- Terminated state



Hình 4-2 Process state

4.7 Process memory layout



Hình 4-3 Memory layout of a process

- **Text segment:** Chứa các chỉ lệnh ngôn ngữ máy (machine-language instruction) của chương trình mà tiến trình đó chạy. Text segment chính là các chỉ lệnh được biên dịch từ source code của lập trình viên cho chương trình đó. Nội dung của phân vùng này không thay đổi trong suốt quá trình process tồn tại nên được kernel thiết lập chế độ read-only để bảo vệ khỏi sự truy cập vô tình hay cố ý của người dùng. Như đã nói ở trên, vì nhiều tiến trình có thể chạy một chương trình nên text segment cũng được thiết lập sharable để các tiến trình có thể sử dụng chung để tiết kiệm tài nguyên.
- **Initialized data segment:** Vùng này chứa các biến toàn cục (global) và biến static mà đã được khởi tạo từ code của chương trình. Giá trị của các biến này

được đọc từ các file thực thi khi chương trình được tải vào RAM. Ví dụ khi lập trình viên khai báo biến tĩnh “*static var = 10;*”, biến var này sẽ được lưu vào vùng nhớ của initialized data segment.

- **Uninitialized data segment:** Còn được gọi là vùng bss segment. Segment này chứa các biến toàn cục và static mà chưa được khởi tạo từ source code. Ví dụ khi lập trình viên khai báo biến tĩnh “*static var;*”, biến var sẽ được chứa ở vùng này và được khởi tạo giá trị 0. Nếu bạn thắc mắc tại sao biến var không được lưu vào vùng initialized data segment cho đơn giản. Câu trả lời là khi một chương trình được lưu trữ vào ổ cứng, không cần thiết phải cấp phát tài nguyên cho uninitialized data segment; thay vào đó chương trình chỉ cần nhớ vị trí và kích thước biến được yêu cầu cho vùng này, các biến này sẽ được cấp phát run time khi chương trình được tải vào RAM.
- **Stack segment:** Chứa stack frame của tiến trình. Chúng ta sẽ tìm hiểu sâu hơn về stack và stack frame ở phần dưới đây. Tạm thời hiểu là khi 1 hàm được gọi, một stack frame sẽ được cấp phát cho hàm đó (các biến được khai báo trong hàm, các đối số truyền vào hay giá trị return) và sẽ bị thu hồi khi hàm đó kết thúc. Vì vậy, stack segment có thể giãn ra hoặc co lại khi tiến trình cấp phát/thu hồi các stack frame.
- **Heap segment:** Là vùng bộ nhớ lưu các biến được cấp phát động (dynamic allocate) tại thời điểm run time. Tương tự như stack segment, heap segment cũng có thể giãn ra hoặc co vào khi một biến được cấp phát hoặc free.

4.7.1 Stack và Stack frame

Stack segment của tiến trình có thể được giãn ra hoặc co lại mỗi khi một hàm được gọi hoặc return. Trong kiến trúc x86-32 có một thanh ghi đặc biệt là stack pointer (sp) lưu thông tin địa chỉ đỉnh stack. Khi một hàm được gọi, một stack frame của hàm đó được cấp phát và push vào stack; và stack frame này sẽ được thu hồi khi hàm đó return.

Mỗi stack frame chứa các thông tin sau:

Đối số (argument) của hàm và biến cục bộ (local variable):

Các biến này còn được gọi là biến tự động automatic variable vì chúng sẽ tự động được tạo ra khi hàm gọi và tự động biến mất khi hàm đó return (vì stack frame cũng biến mất).

Call linkage information: Các hàm khi chạy sẽ sử dụng các thanh ghi của CPU, ví dụ thanh ghi program counter (pc) lưu địa chỉ tiếp theo được thực thi. Mỗi lần một hàm (ví dụ hàm X) gọi hàm khác (ví dụ hàm Y), giá trị các thanh ghi mà hàm X đang dùng sẽ được lưu vào stack frame của hàm Y; và sau khi hàm Y return, các giá trị thanh ghi này sẽ được phục hồi cho hàm X tiếp tục chạy.

4.7.2 Argc và argv

Mọi chương trình C đều phải có hàm main(), được gọi là entry point của chương trình vì là hàm đầu tiên được thực thi khi tiến trình chạy. Thông thường, một hàm main() của chương trình thường có 2 đối số sau:

int main (int argc, char *argv[])

Đối số đầu tiên argc chỉ số đối số command-line được truyền nào. Đối số tiếp theo char *argv[] là 1 mảng con trỏ trỏ đến các đối số command-line, mỗi command-line là 1 chuỗi kết thúc bởi ký tự NULL. Trong mảng con trỏ đối số đó, chuỗi đầu tiên là argv[0] luôn phải là tên của chính chương trình. Mảng đối số luôn phải kết thúc bởi con trỏ NULL (argv[argc] = NULL).

4.7.3 Environment variable

Mỗi tiến trình có 1 danh sách các biến ở dạng string gắn với nó được gọi là các biến môi trường (environment list). Mỗi chuỗi trong số này được định nghĩa dưới dạng name=value, các biến này được dùng để lưu trữ 1 thông tin bất kỳ mà tiến trình muốn giữ. Hiểu một cách đơn giản, biến môi trường là biến của tiến trình đang chạy đó, không phải là biến của một hàm nào cả và được lưu trữ trong không gian bộ nhớ của tiến trình đó.

Khi 1 chương trình được tạo ra, nó sẽ kế thừa các biến môi trường của tiến trình cha. Vì đặc điểm này, sử dụng biến môi trường cũng có thể coi là 1 cách rất đơn giản cho giao tiếp liên tiến trình (IPC) giữa tiến trình cha và con. Ví dụ khi bạn tạo 1 biến môi trường từ tiến trình cha rồi tạo ra một tiến trình con, tiến trình con này sẽ lưu giữ giá trị của biến đó. Lưu ý là việc lưu giữ này chỉ là một chiều và một lần, nghĩa là sau đó nếu tiến trình cha hoặc con thay đổi giá trị của biến đó thì nó sẽ không được cập nhật sang cho tiến trình còn lại.

4.8 Thao tác với process

4.8.1 Tạo process mới

Một tiến trình Linux tạo ra các tiến trình con nhằm mục đích chia công việc của mình cho các tiến trình con.

Sử dụng fork() system call

```
#include <unistd>
pid_t fork(void);
```

In parent: returns process ID of child on success, or -1 on error;
in successfully created child: always returns 0

System call `fork()` làm việc bằng cách tạo ra 1 tiến trình mới với PID mới, và nhân bản dữ liệu từ tiến trình cha sang tiến trình con (các segment stack, data và heap), riêng text segment không cần sao chép mà được sử dụng chung (sharable) bởi cả 2 tiến trình.

Vì vậy, tiến trình con sẽ kế thừa toàn bộ các biến (bao gồm cả biến môi trường), giá trị hiện tại của các biến, các mô tả file và stack frame của tiến trình cha. Sau lời gọi `fork()`, 2 tiến trình sẽ đồng thời tồn tại và chúng sẽ tiếp tục chạy sau thời điểm `fork()` return.

4.8.2 Chạy process: *exec family*

`execve(pathname, argv, envp);`

`execve` system call loads a new program (pathname, with argument list argv, and environment list envp) into process memory.

Example:

```
execl("/usr/bin/vi", "vi", "home/work/hello.txt", NULL);
```

- l: đối số dạng list (`execl`)
- v: đối số dạng vector (`execv`)
- p: đường dẫn gốc
- e: biến môi trường

Bảng 4-1 Exec family

Function	Specification of program file (-, p)	Specification of argument (v, l)	Source of environment (e, -)
<code>execve()</code>	Pathname	Array	Envp arg
<code>execle()</code>	Pathname	List	Envp arg
<code>execlp()</code>	Filename + path	List	Caller's array
<code>execvp()</code>	Filename + path	Array	Caller's array
<code>execv()</code>	Pathname	Array	Caller's array
<code>execl()</code>	Pathname	List	Caller's array

4.8.3 Terminating a process

Một tiến trình được kết thúc khi nó kết thúc công việc của mình hoặc có lỗi xảy ra (ví dụ `segfault`) với tiến trình đó.

Khi tiến trình kết thúc, các tài nguyên của nó (bộ nhớ, các mô tả file đang mở,...) bị thu hồi và có thể được cấp phát cho tiến trình khác

System call `_exit()`

```
#include <unistd.h>

void _exit(int status);
```

Đối số status truyền vào `_exit()` định nghĩa trạng thái kết thúc (termination status) của tiến trình, nó có thể được tiến trình cha dùng khi gọi system call `wait()` mà chúng ta sẽ học dưới đây. Theo quy ước, giá trị của đối số status là 0 nghĩa là tiến trình được kết thúc thành công, khác 0 nghĩa là kết thúc không thành công.

Trong thực tế, lập trình viên thường không dùng trực tiếp system call `_exit()` mà dùng hàm thư viện `exit()`, hàm này sẽ làm 1 số việc cần thiết (thoát các hàm handler và flush các stdio buffer) trước khi gọi system call `_exit()`.

```
#include <stdlib.h>

void exit(int status)
```

Thực tế dùng `return n` trong lập trình cũng tương tự gọi `exit(n)`

4.8.4 System call `wait()`

```
#include <unistd.h>

pid_t wait(int *status);
```

Return child PID or -1 if error

Hoặc

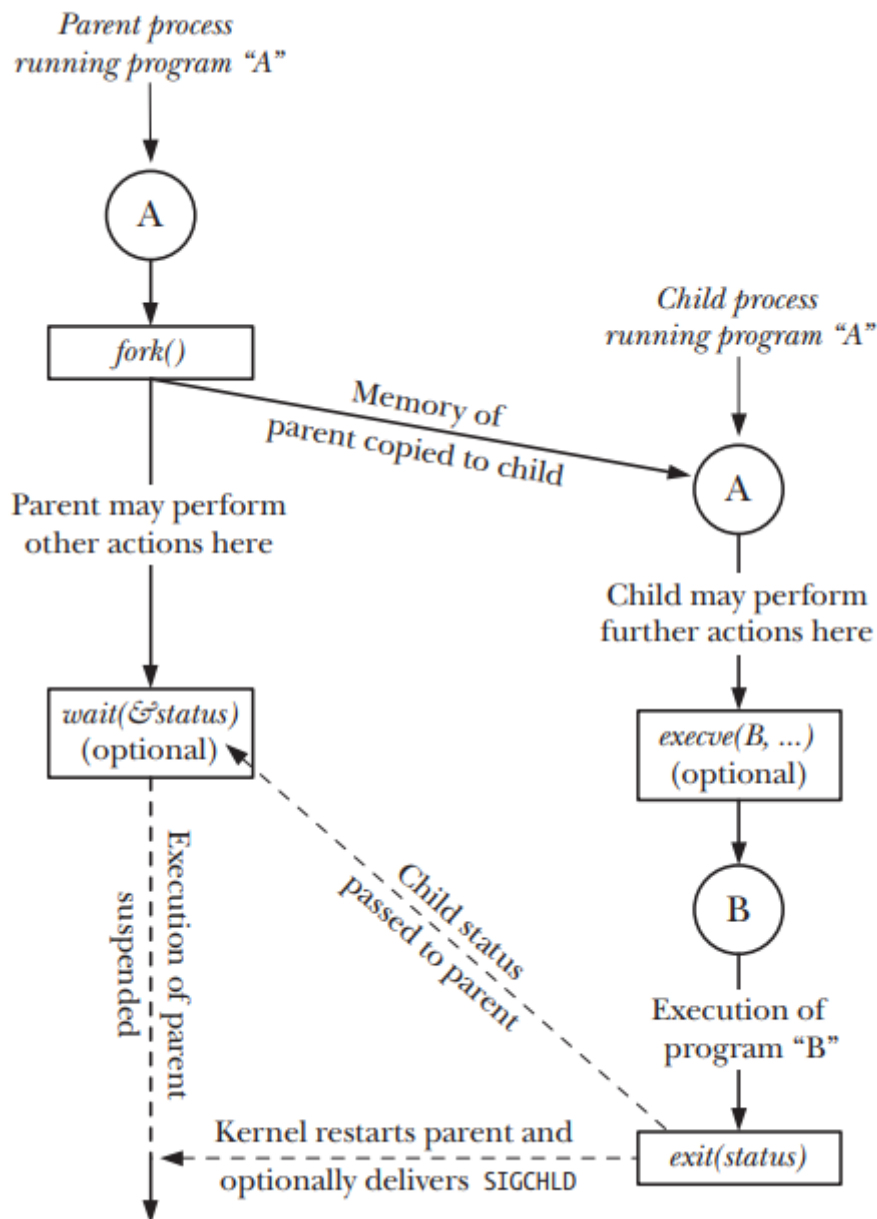
```
pid_t waitpid(pid_t pid, int *status, int options);
```

Được gọi ở tiến trình cha chờ đến khi một trong các tiến trình con bị kết thúc và trong trạng thái kế thúc, mang thông tin trạng thái kết thúc của tiến trình con.

Cách thức hoạt động chi tiết:

- If no (previously unwaited-for) child of the calling process has yet terminated, the call blocks until one of the children terminates. If a child has already terminated by the time of the call, `wait()` returns immediately.
- If status is not NULL, information about how the child terminated is returned in the integer to which status points.

- The kernel adds the process CPU times and resource usage statistics to running totals for all children of this parent process.
- As its function result, wait() returns the process ID of the child that has terminated.



4.9 Virtual memory in Linux

Like most modern kernels, Linux employs a technique known as virtual memory management. The aim of this technique is to make efficient use of both the CPU and RAM (physical memory) by exploiting a property that is typical of most programs: locality of reference.

Each process hold instruction/code, data, stack, heap in the imaginary virtual memory, is copied to real memory at time of execution

Each process has its own memory space (4GB) on a 32bit system

Each process has its own private user space memory, this means one process can not access memory of other process directly

The process virtual memory is configurable, but usually 3GB user space and 1GB kernel space.

Most programs demonstrate two kinds of locality:

- Spatial locality is the tendency of a program to reference memory addresses that are near those that were recently accessed (because of sequential processing of instructions, and, sometimes, sequential processing of data structures).
- Temporal locality is the tendency of a program to access the same memory addresses in the near future that it accessed in the recent past (because of loops).

4.9.1 Page frame

A virtual memory scheme splits the memory used by each program into small, fixed-size units called pages.

Correspondingly, RAM is divided into a series of page frames of the same size. At any one time, only some of the pages of a program need to be resident in physical memory page frames; these pages form the so-called resident set.

Copies of the unused pages of a program are maintained in the swap area—a reserved area of disk space used to supplement the computer's RAM—and loaded into physical memory only as required.

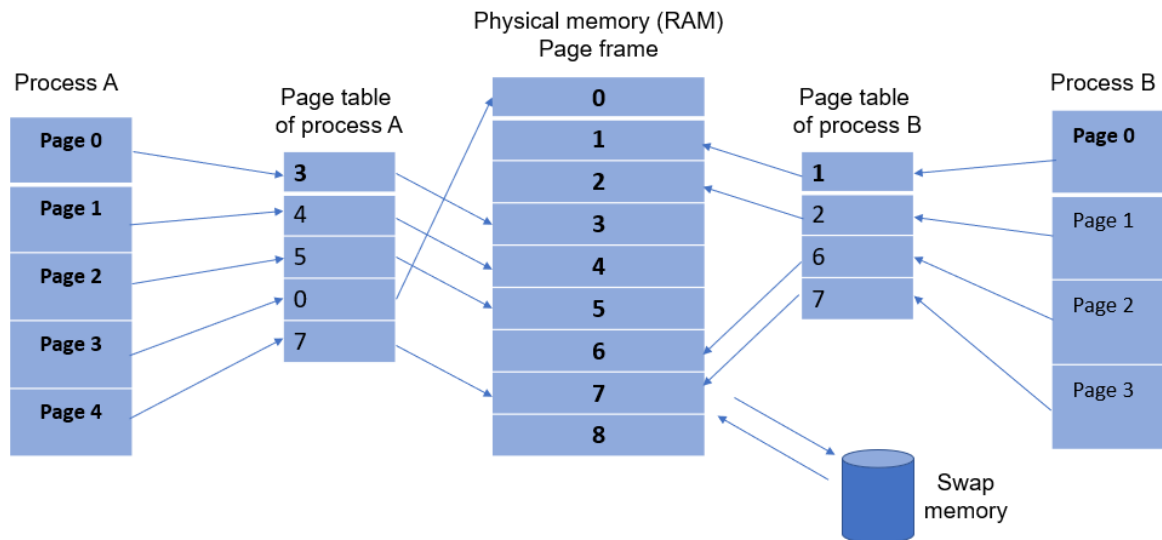
When a process references a page that is not currently resident in physical memory, a page fault occurs, at which point the kernel suspends execution of the process while the page is loaded from disk into memory.

Frame size varies – generally 4K, 8K or 16K and is configurable.

4.9.2 Page table

Two or more process can share memory. This is possible by having page-table entries of different processes refer to the same pages of RAM. Memory sharing occurs below circumstances:

- Multiple processes executing the same program can share a single (read-only) copy of the program code.
- Processes can use the shared memory with other processes to exchange data's.



Hình 4-4 Page table in virtual memory

A process range of valid virtual address can change over its lifetime, as the kernel allocates and deallocates pages for process.

- Stack grow downward beyond limits pre reached.
- Memory is allocated or deallocated on heap
- System V share memory regions – `shmat()`, `shmdt()`
- Memory mapping are created – `mmap()`, `munmap()`

4.10 Memory mapping

The `mmap()` system call creates a new memory mapping in the calling process's virtual address space. A mapping can be of two types:

- **File mapping:** A file mapping maps a region of a file directly into the calling process's virtual memory. Once a file is mapped, its contents can be accessed by operations on the bytes in the corresponding memory region. The pages of the mapping are (automatically) loaded from the file as required. This type of mapping is also known as a file-based mapping or memory-mapped file.
- **Anonymous mapping:** An anonymous mapping doesn't have a corresponding file. Instead, the pages of the mapping are initialized to 0.

The memory in one process's mapping may be shared with mappings in other processes (i.e., the page-table entries of each process point to the same pages of RAM).

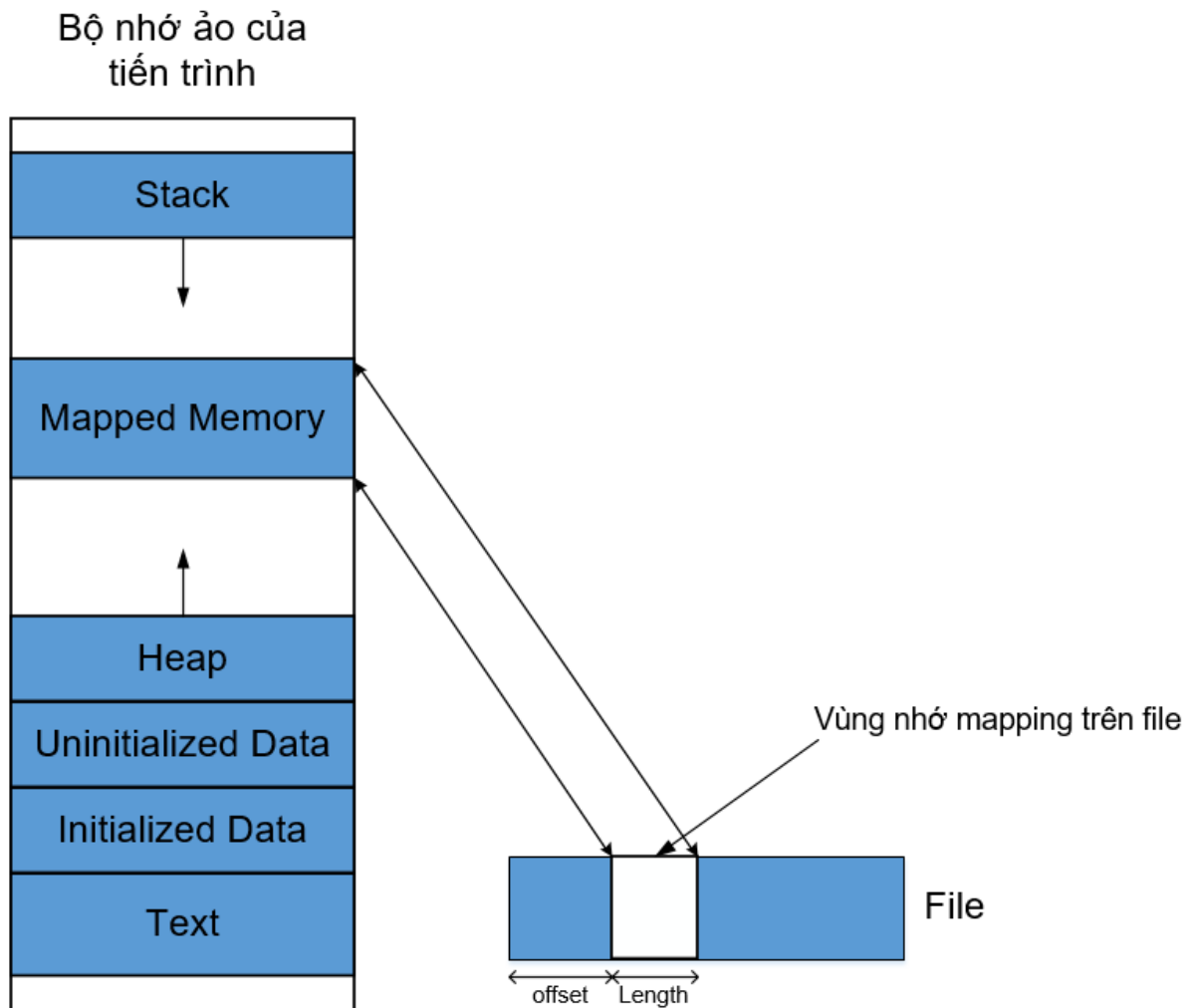
This can occur in two ways:

- When two processes map the same region of a file, they share the same pages of physical memory.

- A child process created by `fork()` inherits copies of its parent's mappings, and these mappings refer to the same pages of physical memory as the corresponding mappings in the parent.

When two or more processes share the same pages, each process can potentially see the changes to the page contents made by other processes, depending on whether the mapping is private or shared:

- Private mapping (`MAP_PRIVATE`): Modifications to the contents of the mapping are not visible to other processes and, for a file mapping, are not carried through to the underlying file. Although the pages of a private mapping are initially shared in the circumstances described above, changes to the contents of the mapping are nevertheless private to each process. The kernel accomplishes this using the copy-on write technique. This means that whenever a process attempts to modify the contents of a page, the kernel first creates a new, separate copy of that page for the process (and adjusts the process's page tables). For this reason, a `MAP_PRIVATE` mapping is sometimes referred to as a private, copy-on-write mapping.
- Shared mapping (`MAP_SHARED`): Modifications to the contents of the mapping are visible to other processes that share the same mapping and, for a file mapping, are carried through to the underlying file.



Hình 4-5 File memory mapping

4.10.1 Creating a mapping

```
#include <sys/mman.h>
```

```
Void *mmap (void *addr, size_t length, int prot, int flags, int fd, int off_t offset);
```

Returns starting address of mapping on success, or MAP_FAILED on error

The addr argument: the virtual address at which the mapping is to be located. If specify addr as NULL, the kernel choose a suitable address for the mapping

The length: size of the mapping in bytes.

The prot argument is a bit mask specifying the protection to be placed on the mapping.

Bảng 4-2 Memory protection values

Value	Description
PROT_NONE	The region may not be accessed

PROT_READ	The contents of the region can be read
PROT_WRITE	The contents of the region can be modified
PROT_EXEC	The contents of the region can be executed

The flags argument is a bit mask of options controlling various aspects of the mapping operation.

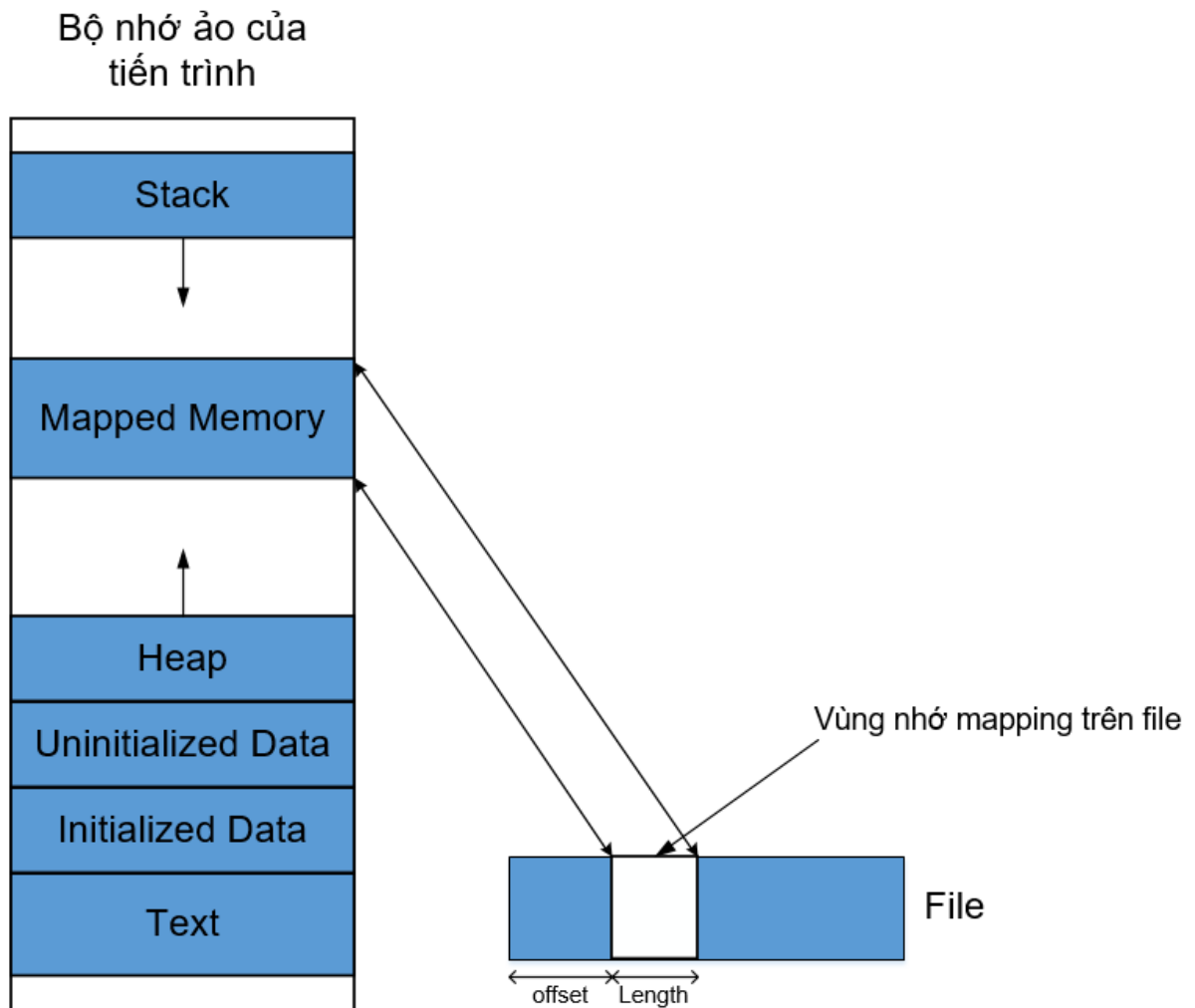
The fd and offset are used with file mappings (they are ignored for anonymous mappings). Fd is file descriptor and offset is the starting point of the mapping in the file.

4.10.2 File mapping

To create a file mapping, we perform the following steps:

- Obtain a descriptor for the file, typically via a call to open().
- Pass that file descriptor as the fd argument in a call to mmap()

As a result of these steps, mmap() maps the contents of the open file into the address space of the calling process. Once mmap() has been called, we can close the file descriptor without affecting the mapping.



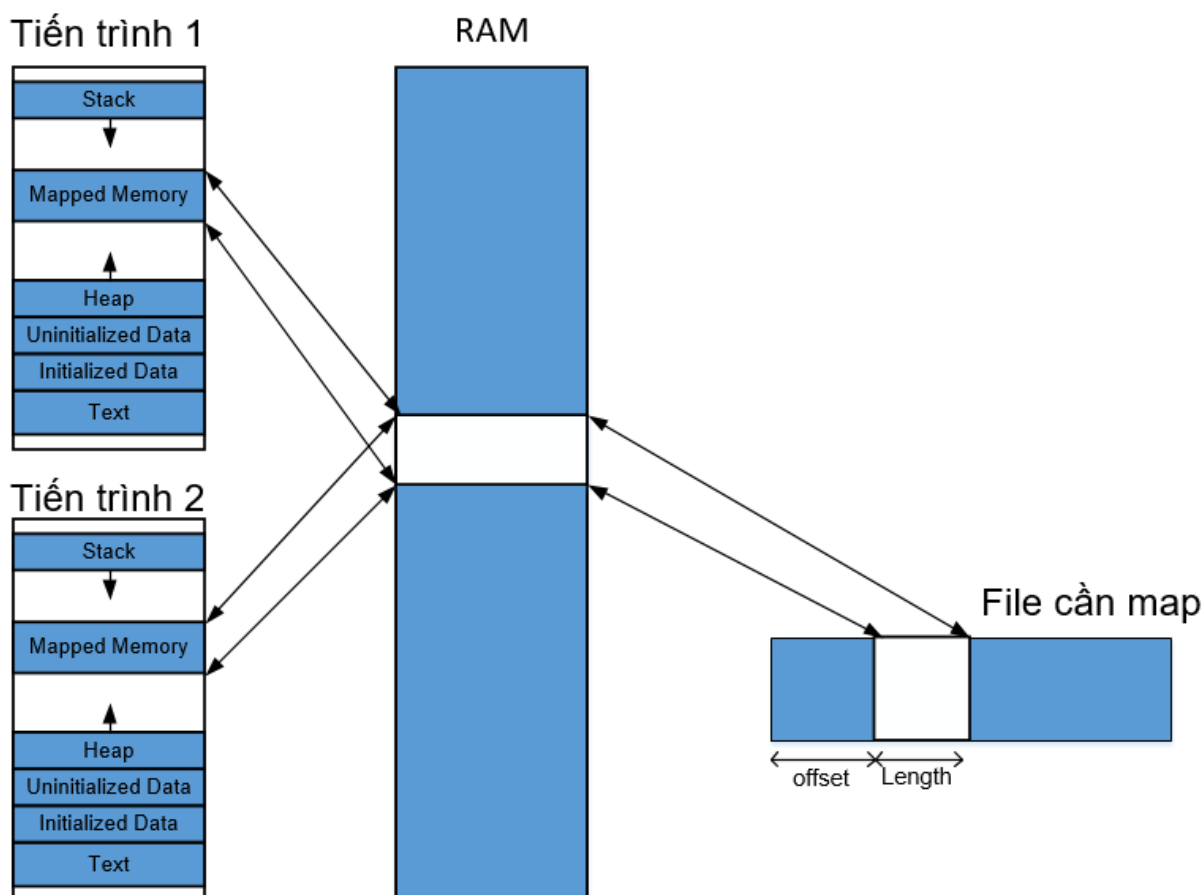
Hình 4-6 File memory mapping

4.10.3 Private file mapping

The two most common uses of private file mappings are the following:

- To allow multiple processes executing the same program or using the same shared library to share the same (read-only) text segment, which is mapped from the corresponding part of the underlying executable or library file.
- To map the initialized data segment of an executable or shared library. Such mappings are made private so that modifications to the contents of the mapped data segment are not carried through to the underlying file.

4.10.4 Shared file mapping



Hình 4-7 Shared memory mapping

When multiple processes create shared mappings of the same file region, they all share the same physical pages of memory. In addition, modifications to the contents of the mapping are carried through to the file. In effect, the file is being treated as the paging store for this region of memory.

Shared file mappings serve two purposes: memory-mapped I/O and IPC.

a) Memory mapped I/O

Since the contents of the shared file mapping are initialized from the file, and any modifications to the contents of the mapping are automatically carried through to the file, we can perform file I/O simply by accessing bytes of memory, relying on the kernel to ensure that the changes to memory are propagated to the mapped file.

This technique is referred to as memory-mapped I/O, and is an alternative to using `read()` and `write()` to access the contents of a file.

Memory-mapped I/O has two potential advantages:

- By replacing `read()` and `write()` system calls with memory accesses, it can simplify the logic of some applications.

- It can, in some circumstances, provide better performance than file I/O carried out using the conventional I/O system calls.
- b) IPC using a shared file mapping

Since all processes with a shared mapping of the same file region share the same physical pages of memory, the second use of a shared file mapping is as a method of (fast) IPC.

Anonymous mapping

An anonymous mapping is one that doesn't have a corresponding file

On Linux, there are two different, equivalent methods of creating an anonymous mapping with `mmap()`.

- Specify `MAP_ANONYMOUS` in flags and specify `fd` as `-1`. (On Linux, the value of `fd` is ignored when `MAP_ANONYMOUS` is specified. However, some UNIX implementations require `fd` to be `-1` when employing `MAP_ANONYMOUS`, and portable applications should ensure that they do this.)
- Open the `/dev/zero` device file and pass the resulting file descriptor to `mmap()`.

CHƯƠNG 5. SIGNAL

5.1 Concept and overview

Signal is notification to a process that an event has occurred.

Signals are sometimes described as software interrupts.

- Can be employed as a synchronization technique, even as a primitive form of inter process communication (IPC).
- The usual source of many signals sent to a process is kernel.

Types of events that cause kernel to generate a signal for a process

- Hardware exception occurred (Ex: dividing for 0).
- User type one of the terminal special characters that generate signals (Ex: interrupt ctrl + C, suspend ctrl + Z)
- Software event occurred (Ex: terminal resize, timer went off)

Each signal is defined as a unique integer, starting sequentially from 1 in /usr/include/signal.h

List of signals: [kill -l](#)

```
cuongvn@cuongvn:~$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM     15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD    18) SIGCONT    19) SIGSTOP     20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH   29) SIGIO        30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

Hình 5-1 List of signals

5.2 Các nguồn gửi signals:

- From hardware
- From kernel
- From user
- From process to process

5.3 Các cách gửi signal đến process

By keyboard:

- Ctrl C: SIGINT – interrupt
- Ctrl Z: SIGSTP – suspend
- Ctrl /: SIGABRT – abort

By command line

- Kill thường dùng để ngừng thực thi một tiến trình
- Default: 15 – SIGTERM
- Fg: gửi các tín hiệu CONT đến tiến trình, đánh thức các tiến trình tạm dừng do TSTP trước đó

By system function: kill()

5.4 Receive and handle signal

When a process received a signal, it can:

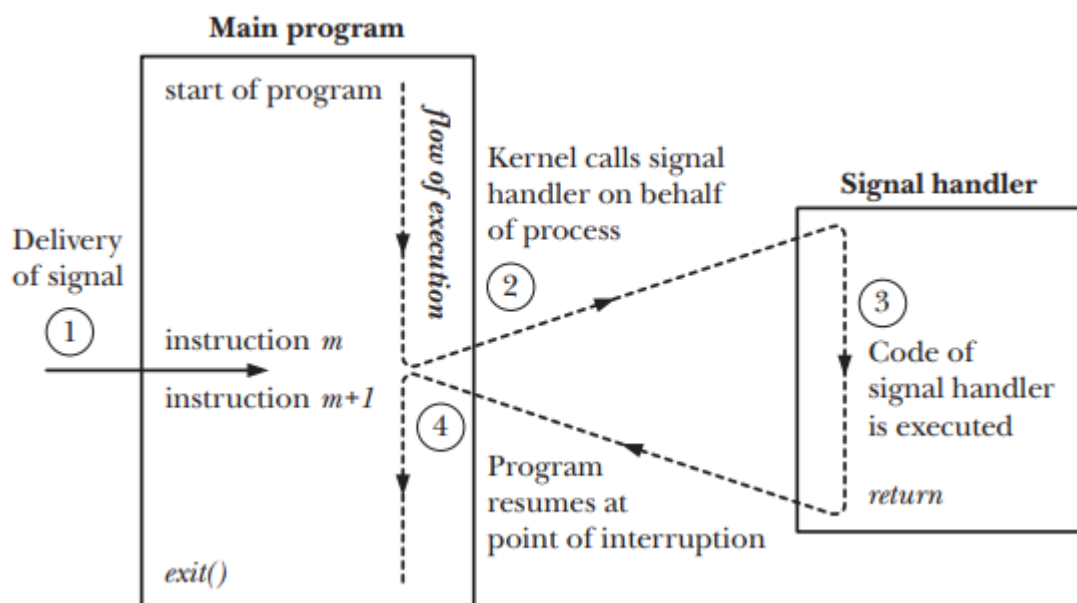
- Ignore the signal
- Handle signal by default
- Receive and handle by program's special way

Note: There are two signals cannot be ignored: SIGKILL and SIGSTOP

Signals have a very precise life cycle

- First, a signal is raised, kernel then stores the signal until it is able to deliver it.
- Finally, signals are delivered to corresponding process

Signal handler



Hình 5-2 Signal delivery and handler execution

Signal handler called when a specified signal is delivered to process

Header file: <signal.h>

5.5 Most used signal explanations

SIGABRT: abort() function send this signal to the process that invoked it. The process then terminates and generates a core dump file

SIGALRM: alarm() and setitimer(), function send this signal to the process that invoked them when an alarm expires.

SIGBUS: (bus error) this signal is raised when process has a memory access error

SIGCHLD: Whenever a process terminates or stops, the kernel sends this signal to the process's parent

SIGCONT: Kernel sends this signal to process when this process needs to be resumed which is in a stopped state

SIGFPE: represent any arithmetic exception

SIGKILL: kernel send this signal when a process attempts to execute an illegal machine instruction

SIGINT: interrupt signal, this occurs when user type Ctrl C

SIGKILL: this signal is sent from the kill() system call

SIGPIPE: if a process writes to a pipe but the reader has terminated, the kernel raises this signal.

SIGSEGV: this signal, whose name delivered from segmentation violation, is sent to a process when it attempts an invalid memory access

SIGUSR1 and **SIGUSR2;** are available for user defined purposes, the Kernel never raises them

5.6 Basic signal management

```
#include <signal.h>

sighandler_t signal(int signo, sighandler_t mgh);

void my_handler(int signo);
```

When a signal is delivered to a process, using signal(), the signal can either perform default action or ignore the particular signal.

SIG_DFL: Set the behavior of the signal given by signo to its default

Ex: SIGPIPE, process will terminate

SIG_IGNL: ignore the signal given by signo

5.7 Sending a signal

The `kill()` signal call, the basic of the common `kill` utility, send a signal from one process to another

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
```

CHƯƠNG 6. THREAD

Thread là một thành phần của process

Một process có thể có một hoặc nhiều thread

Mỗi process bắt đầu chạy luôn có một thread chính

Các thread trong process chia sẻ vùng nhớ toàn cục của tiến trình: initialized, uninitialized, heap.

Dùng chung global memory nhưng có vùng nhớ stack riêng

Các thread có thể thực thi đồng thời và độc lập

6.1 Pros and cons

Pros:

- Easy to share memory
- Create a new thread is faster than process

Cons:

- Because shared memory, if a process has an error → affected with other thread
- Process memory is limited

6.2 Thread vs Process

- Threads dễ dàng để tạo hơn là process vì chúng không cần yêu cầu thêm bộ nhớ riêng.
- Multithreading yêu cầu lập trình cẩn thận hơn khi các threads chia sẻ cấu trúc dữ liệu. Nghĩa là nó chỉ được thay đổi cấu trúc dữ liệu bởi một thread tại một thời điểm. Không giống như thread, các process không thể chia sẻ chung một địa chỉ ô nhớ.
- Các thread được xem là nhẹ (lightweight) bởi vì chúng được sử dụng ít tài nguyên hơn so với process.
- Các process là độc lập với nhau. Các thread thì chia sẻ chung địa chỉ ô nhớ.
- Một process có thể chứa nhiều threads.

6.3 Thread ID

Mỗi thread trong process có một số định dạng duy nhất là threads ID

Kiểm tra thread ID:

```
#include <pthread.h>
```

```
pthread_t pthread_self(void); // return threads id
```

6.4 Create a new process

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void*),  
void *arg);
```

Return 0 if success, positive number if error

Pthread_create() tạo một thread mới trong process

Entry point của thread mới: start()

Main thread của process tiếp tục thực thi các câu lệnh sau hàm pthread_create().

6.5 Thread terminate

Các cách để kết thúc một thread đang thực thi

- Hàm bắt đầu thread gặp lệnh return
- Gọi hàm pthread_exit()
- Bị huỷ bởi hàm pthread_cancel()
- Thread bất kì gọi hàm exit() hoặc thread chính gọi return

```
void pthread_exit(void *retval)
```

Việc gọi hàm pthread_exit() có tác dụng giống với việc gọi return của hàm bắt đầu của thread đó, chỉ khác là pthread_exit() có thể gọi từ bất kỳ hàm nào trong thread còn return bắt buộc phải gọi ở hàm bắt đầu của thread. Đối số retval là giá trị return của hàm. Lưu ý rằng hàm pthread_exit() không return giá trị nào cho hàm gọi nó.

6.6 Thread management

Problem: Khi thread kết thúc gọi pthread_exit, kernel không tự động thu hồi tài nguyên của thread đó, điều này tương tự như zombie process gây lãng phí tài nguyên.

6.6.1 Joinable thread

Được kernel theo dõi trạng thái kết thúc và thu hồi tài nguyên.

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

Return 0 on success, or a positive error number on error

Hàm `pthread_join()` sẽ block chương trình và chờ cho thread với ID là “**thread**” kết thúc, và giá trị return của thread đó được lưu vào biến con trỏ “**retval**”. Nếu thread đã kết thúc trước khi gọi `pthread_join()`, thì hàm sẽ return ngay lập tức. Việc gọi hàm `pthread_join()` 2 lần với cùng một thread ID có thể dẫn đến lỗi “unpredictable behavior”.

Việc gọi hàm `pthread_join()` sau khi tạo ra thread mới bằng hàm `pthread_create()` cũng tương tự như việc gọi system call `wait()` sau khi tạo ra tiến trình con bằng `fork()`. Chỉ khác là `pthread_join()` có thể được gọi từ bất kỳ thread nào trong tiến trình, còn `wait()` phải được gọi bởi tiến trình cha đã tạo ra nó. Ví dụ, thread A tạo ra thread B và thread B tạo ra thread C thì thread A có thể gọi `pthread_join()` với thread C và ngược lại. Ngoài ra, `pthread_join()` phải join vào 1 thread cụ thể, không có khái niệm join với tất cả các thread giống như system call `wait()`.

```
#include <stdio.h>
#include <pthread.h>
#include <errno.h>

void *thread_start(void *args)
{
    pthread_exit(NULL);
}

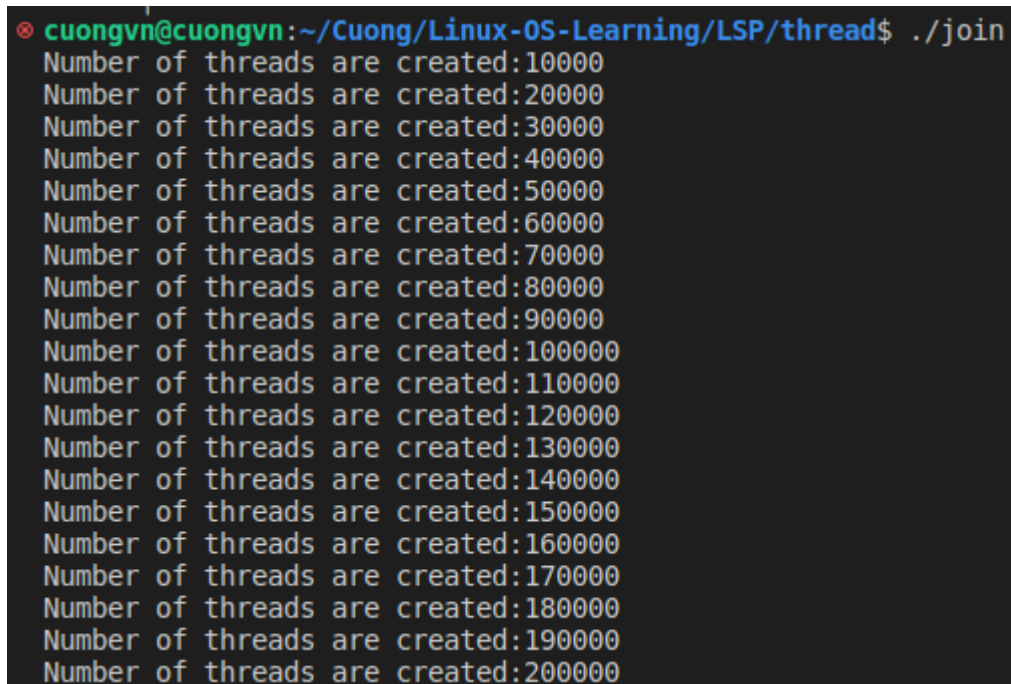
int main (void)
{
    pthread_t threadID;
    int ret;
    long count = 0;
    void *retval;

    while(1)
    {
        if(ret = pthread_create(&threadID, NULL, thread_start, NULL))
        {
```

```

        printf("pthread_create() fail ret: %d\n", ret);
        perror("Fail reason:");
        break;
    }
    pthread_join(threadID, &retval);
    count++;
    if (0 == count % 10000)
    {
        printf("Number of threads are created:%ld\n", count);
    }
}
printf("Number of threads are created:%ld\n", count);
return 0;
}

```



```

cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/thread$ ./join
Number of threads are created:10000
Number of threads are created:20000
Number of threads are created:30000
Number of threads are created:40000
Number of threads are created:50000
Number of threads are created:60000
Number of threads are created:70000
Number of threads are created:80000
Number of threads are created:90000
Number of threads are created:100000
Number of threads are created:110000
Number of threads are created:120000
Number of threads are created:130000
Number of threads are created:140000
Number of threads are created:150000
Number of threads are created:160000
Number of threads are created:170000
Number of threads are created:180000
Number of threads are created:190000
Number of threads are created:200000

```

Hình 6-1 Ví dụ pthread_join

Note: pthread_join() tương tự với gọi system call wait(), tuy nhiên

- Pthread_join() được gọi từ bất kì thread nào trong process, trong khi wait() phải gọi từ process cha.

- Pthread_join() phải join vào một thread cụ thể, không thể join tất cả thread giống như wait()

6.6.2 Detached thread

Không cần quan tâm đến trạng thái kết thúc

Kernel tự động thu hồi tài nguyên của detached thread khi nó kết thúc.

Một thread mới mặc định sẽ là joinable thread.

```
#include <pthread.h>

int pthread_detach(pthread_t thread);

Return 0 on success, or a positive error number on error
```

Đối số truyền vào: ID của thread

Một thread có thể tự thiết lập nó thành detached thread bằng cách sử dụng hàm pthread_self() để truyền thread ID của nó vào hàm pthread_detach():

```
Pthread_detach(pthread_self());
```

Khi một thread là detach thread

- Không thể truy vấn trạng thái kết thúc
- Không thể chuyển thành joinable thread

Note: detached thread chỉ xác định cách hành xử của hệ thống khi thread đó kết thúc, nó không thể tránh được việc kết thúc thread khi tiến trình chứa nó kết thúc.

```
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>

void *thread_start(void *args)
{
    if(pthread_detach(pthread_self()))
    {
        printf("pthread_detach error\n");
    }
}
```

```

    }
    pthread_exit(NULL);
}

int main (void)
{
    pthread_t threadID;
    int ret;
    long count = 0;

    while(1)
    {
        if(ret = pthread_create(&threadID, NULL, thread_start, NULL))
        {
            printf("pthread_create() fail ret: %d\n", ret);
            perror("Fail reason:");
            break;
        }
        usleep(10);
        count++;
        if (0 == count % 10000)
        {
            printf("Number of threads are created:%ld\n", count);
        }
    }
    printf("Number of threads are created:%ld\n", count);
    return 0;
}

```

```

cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/thread$ ./detach
Number of threads are created:10000
Number of threads are created:20000
Number of threads are created:30000
Number of threads are created:40000
Number of threads are created:50000
Number of threads are created:60000
Number of threads are created:70000
Number of threads are created:80000
Number of threads are created:90000
Number of threads are created:100000
Number of threads are created:110000
Number of threads are created:120000
Number of threads are created:130000
Number of threads are created:140000
Number of threads are created:150000

```

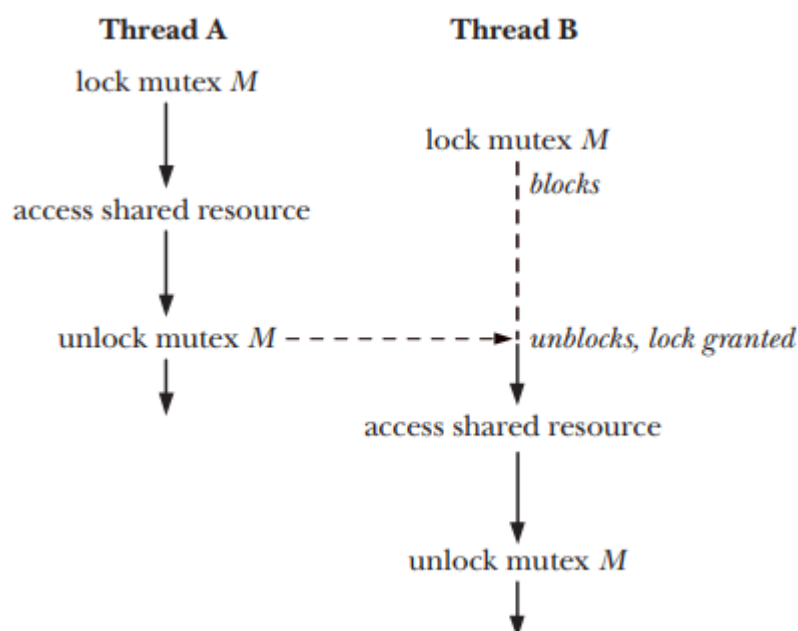
Hình 6-2 Ví dụ về pthread_detach

6.7 Thread synchronization

6.7.1 Critical section

Refer to a section of code that access a shared memory resource and whose execution should be atomic, else synchronization issues arise.

6.7.2 Mutex – Mutual exclusion



Hình 6-3 Using a mutex to protect a critical section

Kỹ thuật mutual exclusion (gọi tắt là mutex) dùng để bảo vệ tài nguyên chia sẻ (shared variable) mà chủ yếu là biến chia sẻ (shared variable) giữa các thread.

Có thể hình dung cách hoạt động của mutex giống như là một cái khóa:

Mỗi vùng critical section sẽ được bảo vệ trong phòng và có một cái khóa ở cửa, một thread trước khi chạy vào critical section sẽ khóa lại rồi vào làm những gì mình muốn làm; sau khi xong việc nó sẽ mở khóa để cho các thread khác truy cập.

Các thread khác nếu thấy cửa đang khóa thì phải chờ (đi vào trạng thái blocked) cho đến khi thread trước mở khóa; sau đó nó tiếp tục khóa lại rồi chạy vào critical section và mở khóa ra sau khi xong việc.

Cần lưu rằng chỉ có thread nào khóa mutex thì mới mở được mutex đó

6.7.3 Mutex operation

- Lock the mutex for the shared resource
- Access the shared resource, perform operations on shared resource as required
- Unlock the mutex

6.7.4 Mutex initialization

In POSIX Thread:

Data type: pthread_mutex_t

Macro: PTHREAD_MUTEX_INITIALIZER

Statically allocation

```
Pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```

Dynamically initializing

```
Int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Khi khởi tạo động, cần hủy mutex khi không cần thiết

```
Int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Lock and unlock mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Nếu mutex đang ở trạng thái unlock, hàm này sẽ khoá mutex sau đó return.
- Nếu mutex đang bị lock bởi một thread khác, hàm này sẽ block cho đến khi unlock
- Nếu thread khoá một mutex mà chính nó đang giữ khoá sẽ xảy ra trường hợp deadlock.

Ngoài ra, chuẩn Posix còn cung cấp hai hàm lock mutex sau đây:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec
*restrict abs_timeout);
```

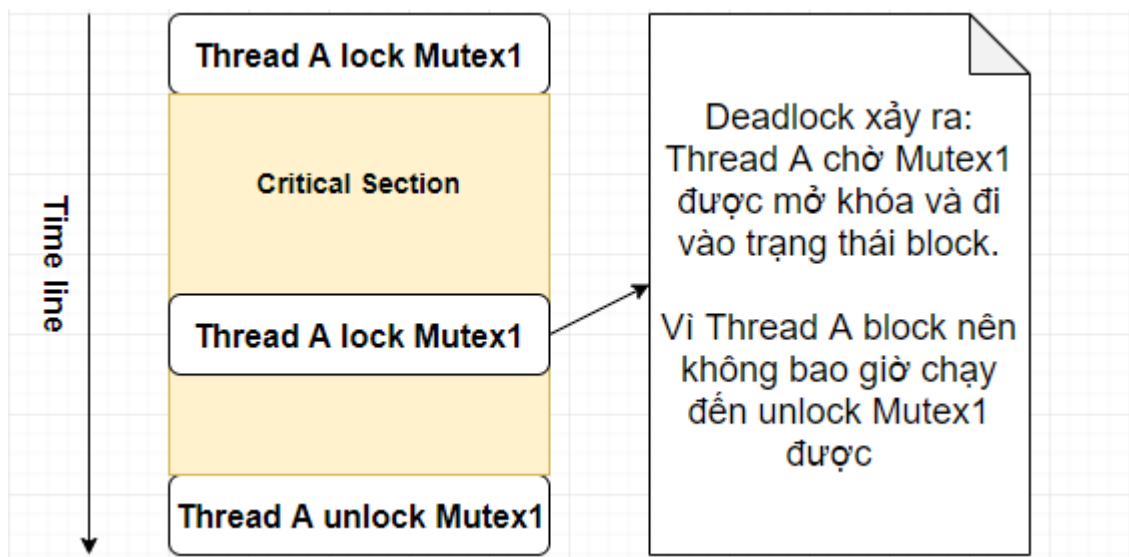
Hàm pthread_mutex_trylock() hoạt động khác pthread_mutex_lock() ở chỗ:

Nếu mutex đang bị khóa, nó sẽ không block thread mà sẽ return ngay lập tức với mã lỗi là EBUSY. Còn hàm pthread_mutex_timedlock() được thêm vào đối số abs_timeout để thiết lập thời gian tối đa thread có thể chờ; nếu sau khoảng thời gian "abs_timeout" mà thread đó chưa sở hữu được mutex, nó sẽ return và trả về mã lỗi ETIMEDOUT.

6.7.5 Mutex Deadlock

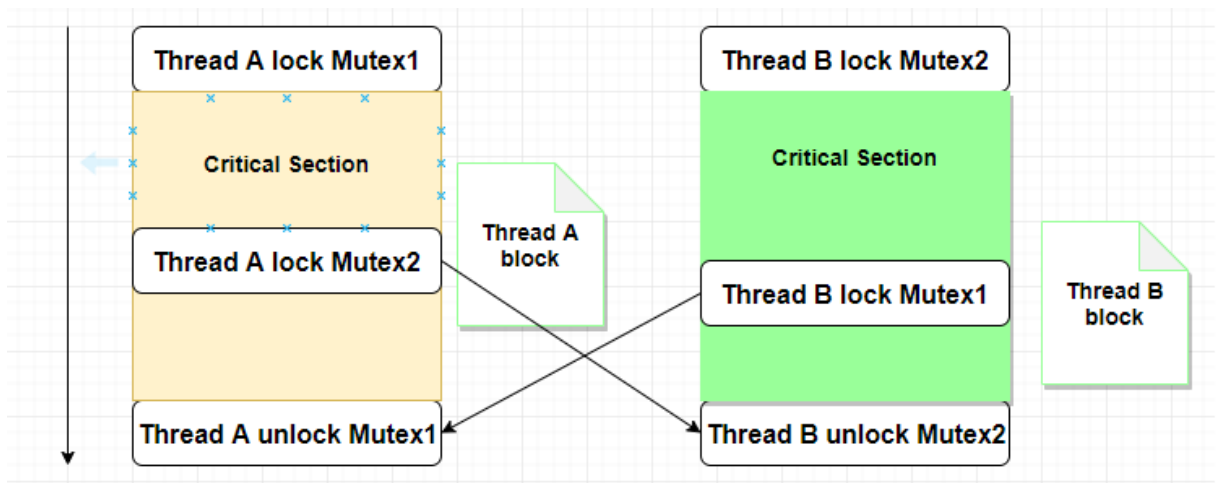
Trong lập trình multi-thread, deadlock xảy ra trong những trường hợp sau đây (giả sử có 2 thread A và B gây ra deadlock):

TH1: Thread A lock một mutex trước đó đã được lock bởi chính nó



Hình 6-4 Trường hợp deadlock 1

TH2: Thread A khóa Mutex1 và vào critical section, sau đó cố gắng khóa một Mutex2 mà thread B đang giữ. Trong khi đó, thread B đang khóa Mutex2 và lại cố gắng khóa Mutex1 ở trong critical section.



Hình 6-5 Trường hợp deadlock 2

Để tránh deadlock trong trường hợp 2, chúng ta có hai cách sau:

- Đảm bảo sao cho tất cả các thread khóa các mutex theo một thứ tự xác định trước. Ví dụ trong trường hợp trên, Thread B cần phải khóa Mutex1 trước, sau đó mới khóa Mutex2, khi đó nếu thread A đang giữ Mutex1 thì thread B chỉ bị block cho đến khi thread A nhả khóa chứ không bị deadlock.
- Sử dụng phương pháp "khóa thử, nếu không được thì quay lại sau" bằng cách dùng hàm `pthread_mutex_trylock()` để khóa mutex. Như đã giải thích ở trên, hàm `pthread_mutex_trylock` thay vì block thread nếu mutex đang bị khóa sẽ return ngay, qua đó tránh được việc block cũng như deadlock. Nếu `pthread_mutex_trylock()` return fail, thread release tất cả các mutex đang khóa và thử lại từ đầu. Cách này thường ít được sử dụng hơn cách trên vì yêu cầu vòng lặp khá tốn kém thời gian và phức tạp.

6.7.6 Condition Variable

A condition variable allows signaling from one thread to other thread, about changes in the state of shared variable

Condition variable help to define the sequence of thread execution

In condition variable, the thread waiting for shared resource will be made to sleep, and as soon as it gets signal from other thread, the thread wakes up and executes.

A condition variable is always used in conjunction with a mutex:

The purpose of the condition variable is to help the thread sleep while waiting for a certain condition of the global variable

6.7.7 Condition variable initialization

Statically allocation

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER
```

Dynamically allocation

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);  
int pthread_cond_destroy(pthread_cond_t *cond);
```

Signaling and waiting on condition variable

- The principal of condition variable is “signal and wait”
- The signal operation is a notification to one or more waiting threads that a shared variable’s state has changed.
- Wait operation is the method of blocking until such a notification is received from other thread
- Pthread_cond_signal() – specified by cond
- Pthread_cond_wait() – blocks thread until the condition variable cond n signaled

```
#include <pthread.h>  
  
int pthread_cond_signal(pthread_cond_t *cond);  
  
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Step 1: The thread calling pthread_cond_wait unlocks MUTEX M

Step 2: Block on the condition variable ‘cond’ to receive the signal from other thread

Step 3: As soon as it receives required signal, it further locks MUTEX M

Example of condition variable:

```
#include<stdio.h>  
#include<string.h>  
#include<pthread.h>  
#include<stdlib.h>  
#include<unistd.h>  
  
#define THRESHOLD 5  
  
pthread_t tid;
```

```

/*Counter la bien toan cuc duoc 2 thread su dung*/

int counter;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //khai báo mutex
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;    // khai bao bien dieu
kien

void threadFunc(void)
{
    pthread_mutex_lock(&mutex); /*Khoa mutex de bao ve bien counter*/
    printf("pthread has started\n");

    while(counter < THRESHOLD)
    {
        counter += 1;
        sleep(5);
    }

    pthread_cond_signal(&cond);
    printf("pthread has finished, counter = %d\n", counter);
    pthread_mutex_unlock(&mutex); /*Nha khoa mutex*/

    pthread_exit(NULL);
}

int main(void)
{

```

```

int ret = 0;

ret = pthread_create(&tid, NULL, &(threadFunc), NULL);
if (ret != 0)
{
    printf("pthread created fail\n");
}

pthread_mutex_lock(&mutex); /*Khoa mutex de bao ve bien counter*/
while(counter < THRESHOLD) /*Kiem tra gia tri cua counter co dung nhu mong
doi khong*/
{
    pthread_cond_wait(&cond, &mutex); /*Cho signal tu thread khac*/
    printf("Global variable counter = %d.\n", counter); /*Truy cap bien counter*/
}

pthread_mutex_unlock(&mutex); /*Nha khoa mutex*/
pthread_join(tid, NULL);

return 0;
}

```

```

● cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/thread$ ./condition
pthread has started
pthread has finished, counter = 5
Global variable counter = 5.

```

Hình 6-6 Ví dụ về condition variable

CHƯƠNG 7. INTER PROCESS COMMUNICATION

IPC are used to send/receive data between process

IPC are also used to synchronize between processes

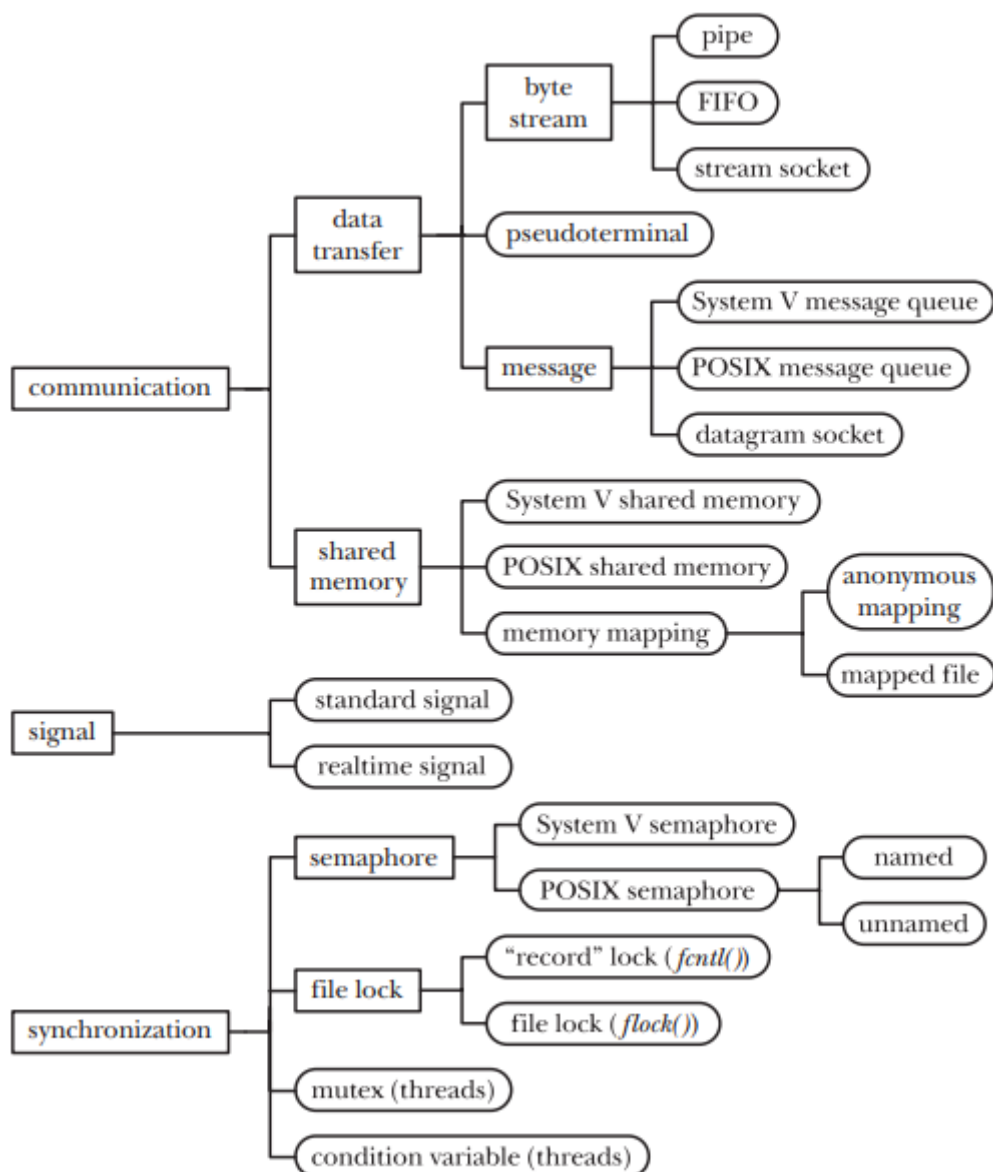
IPC are usually communication based on Synchronize based

Communication based IPC are:

Data transfer based: PIPE, FIFO, message queue, socket

Memory sharing based: Shared memory

Synchronization based: semaphore, mutex (thread), condition variable (thread).

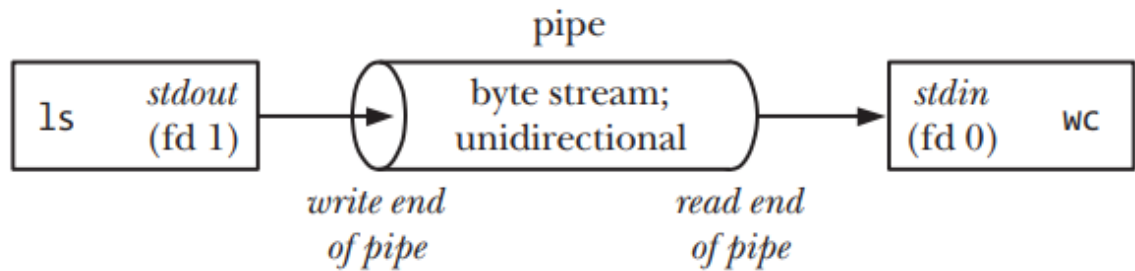


Hình 7-1 A taxonomy of UNIX IPC facilities

7.1 PIPES

A pipe is a byte stream used for IPC

A pipe has 2 ends: read end and write end



Hình 7-2 Using pipe to connect two process

Pipes are unidirectional: Data can travel only in one direction through a pipe. One end of the pipe is used for writing, and the other end is used for reading.

7.1.1 Creating and using pipes

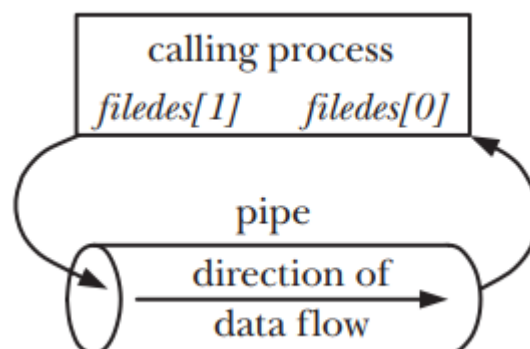
The pipe() system call creates a new pipe.

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

```
int pipe(int fd[2])
```

Return 0 on success, return -1 on error

A successful call to pipe() returns two open file descriptors in the array `filedes`: one for the read end of the pipe (`filedes[0]`) and one for the write end (`filedes[1]`).



Hình 7-3 process file descriptors after creating a pipe

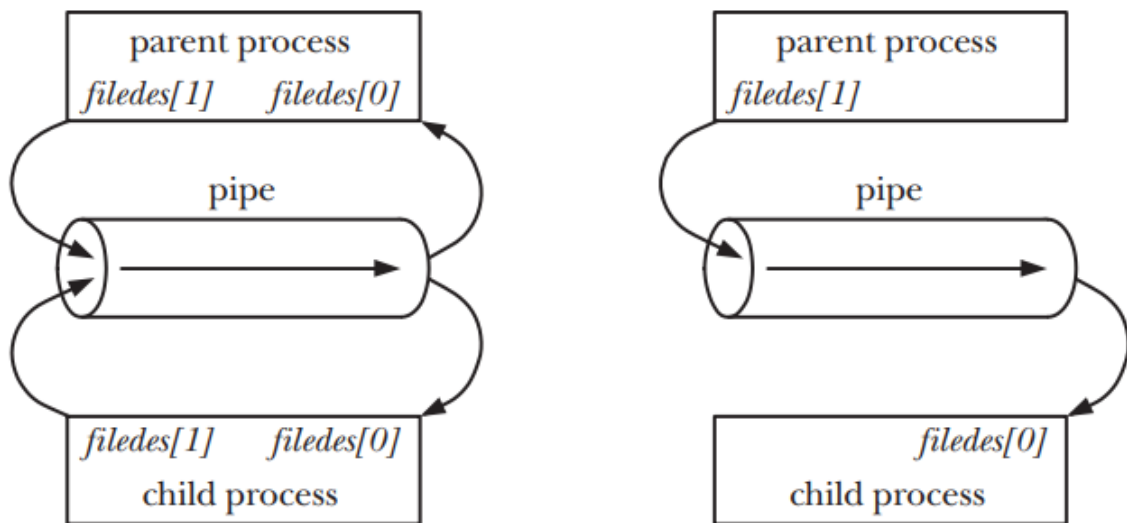
Application of pipes

Used to communicate across related process

Ex: Parent – child process

After fork() – the child process inherits copies of its parent's fd

Pipe after fork()



Hình 7-4 Setting up a pipe to transfer data from a parent to child

```
cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/pipe$ ./pipe3
Parent: sending user input string to child process - ((null))
child: parent closed write end
Segmentation fault (core dumped)
cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/pipe$ ./pipe3 HAHA
Parent: sending user input string to child process - (HAHA)
child process: data received in child process is (HAHA)
child process: closing read end
parent: about to write data to pipe
```

Hình 7-5 Ví dụ về PIPE

7.1.2 SIGPIPE signal in PIPEs

SIGPIPE signal (broken pipe signal) is sent to write end process when read end is closed

Example Program of PIPEs

Note: Two process unaware of existence of the pipe, they just read from and write to the standard fd.

By the normal, PIPE has the limited size (1096 character)

7.1.3 Synchronization in Pipe

Tiến tình đọc pipe sẽ khác nếu pipe trống → đợi đến khi pipe có dữ liệu truy xuất

Tiến trình ghi pipe sẽ bị khoá nếu pipe đầy → đợi đến khi có chỗ trống để ghi dữ liệu.

```
write(int fd, const void *buf, size_t count);  
read(int fd, const void *buf, size_t count);
```

7.2 FIFOs

Semantically, a FIFO is similar to a pipe.

The principal difference is that a FIFO has a name within the file system and is opened in the same way as a regular file.

This allows a FIFO to be used for communication between unrelated processes (e.g., a client and server).

FIFOs are also sometimes known as named pipes.

When all descriptors referring to a FIFO have been closed, any outstanding data is discarded.

Once a FIFO has been opened, we use the same I/O system calls as are used with pipes and other files (i.e., `read()`, `write()`, `close()`).

Just as with pipes, a FIFO has a write end and a read end, and data is read from the FIFO in the same order as it is written.

7.2.1 Create a FIFO

The `mkfifo()` function creates a new FIFO with the given pathname.

```
#include <sys/stat.h>
```

```
int mkfifo(const char *pathname, mode_t mode);
```

Return 0 on success, or -1 on error

```

3  #include <stdio.h>
4  #include <string.h>
5  #include <fcntl.h>
6  #include <sys/stat.h>
7  #include <sys/types.h>
8  #include <unistd.h>
9
10 int main()
11 {
12     int fd1;
13     char str1[80];
14     // FIFO file path
15     char * myfifo = "/tmp/myfifo";
16
17     // First open in read only and read
18     fd1 = open(myfifo,O_RDONLY);
19
20
21     while (1)
22     {
23         read(fd1, str1, 80);
24         // Print the read string and close
25         printf("User1: %s\n", str1);
26     }
27     close(fd1);
28     return 0;
29 }

```

Hình 7-6 Read_fifo.c

```

10  int main()
11  {
12      int fd;
13
14      // FIFO file path
15      char * myfifo = "/tmp/myfifo";
16      char arr2[80];
17
18      // Creating the named file(FIFO)
19      // mkfifo(<pathname>, <permission>)
20      mkfifo(myfifo, 0660);
21      // Open FIFO for write only
22      fd = open(myfifo, O_WRONLY);
23
24      while (1)
25      {
26          printf("\nEnter string to be sent via fifo\n");
27          fgets(arr2, 80, stdin);
28
29          // Write the input arr2ing on FIFO
30          // and close it
31          write(fd, arr2, strlen(arr2)+1);
32      }
33      close(fd);
34      return 0;
35  }

```

Hình 7-7 Write_fifo.c

```

cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/fifo$ ./write
Enter string to be sent via fifo
hello
Enter string to be sent via fifo

cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/fifo$ ./read
User1: hello

```

Hình 7-8 Minh hoạ sử dụng FIFO

7.3 POSIX message queue

Message queue can be used to pass message between process

Unlike FIFO and PIPE, which are ‘byte’ oriented IPC message queue is ‘message’ oriented IPC

Readers and writers communicate each other in units of message

POSIX message queue permit each message to be assigned a priority

Priority allows high-priority messages to be queued ahead of low priority messages

Message queue entry is present in file system in /dev/mqueue

7.3.1 Message queue operation

The mq_open() function creates a new message queue or opens an existing queue, returning a message queue descriptor for use in later calls.

The mq_send() function writes a message to a queue.

The mq_receive() function reads a message from a queue.

The mq_close() function closes a message queue that the process previously opened.

The mq_unlink() function removes a message queue name and marks the queue for deletion when all processes have closed it.

7.3.2 Open a message queue

```
#include <fcntl.h>

#include <sys/stat.h>    /* Defines O_* constants */
#include <mqueue.h>      /* Defines mode constants */

mqd_t mq_open(const char * name , int oflag , ...
               /* mode_t mode , struct mq_attr * attr */);

Returns a message queue descriptor on success, or (mqd_t) -1 on error
```

Bit value for mq_open() flag argument

Bảng 7-1 Bit value for mq_open() flag argument

Flag	Description
O_CREAT	Create queue if it doesn't already exist
O_EXCL	With O_CREAT , create queue exclusively
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing

O_NONBLOCK	Open in nonblocking mode
------------	--------------------------

Closing a message queue

```
#include <mqueue.h>

int mq_close(mqd_t mqdes );
```

Returns 0 on success, or -1 on error

Note:

- A message queue descriptor is automatically closed when a process terminates or call exec()
- As close() for file, closes doesn't delete it. For that purpose, we need mq_unlink(), which is the message queue analog of unlink().

7.3.3 Removing a message queue

```
#include <mqueue.h>

int mq_unlink(const char * name );
```

Returns 0 on success, or -1 on error

Example:

[msg_q_rec.c](#)

[msg_q_send.c](#)

```
cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/tpc/message_queue$ ./send /queue 'Hello'
message queue mq_maxmsg = (10), mq_msgsize is (8192)
cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/tpc/message_queue$ ./rec /queue
Received message in msg queue is (Hello)
```

Hình 7-9 Ví dụ sử dụng message queue

7.4 POSIX Semaphore

Posix semaphore allows process and threads to synchronize access to shared resources

Semaphores are used to execute the critical section in an atomic manner

Named semaphore – has a name as specified in `sem_open()` (on linux present in `/dev/shm`)

Unnamed semaphores: doesn't have a name, it resides at a location in memory

Named semaphore function

- The `sem_open()` function opens or creates a semaphore, initializes the semaphore if it is created by the call, and returns a handle for use in later calls.
- The `sem_post(sem)` and `sem_wait(sem)` functions respectively increment and decrement a semaphore's value.
- The `sem_getvalue()` function retrieves a semaphore's current value.
- The `sem_close()` function removes the calling process's association with a semaphore that it previously opened.
- The `sem_unlink()` function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

7.4.1 Open a named semaphore

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>         /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char * name , int oflag , ...
                /* mode_t mode , unsigned int value */ );

Returns pointer to semaphore on success, or SEM_FAILED on error
```

Note: when program is opening a existing semaphore we need to take care of oflag, both 'read' and 'write' permission is required.

Value: an integer – the init value assign to new semaphore

When a child is created via `fork()`, it inherits references to all of named semaphore that open in its parent. After `fork()`, parent and child can use these semaphore to synchronize their action.

7.4.2 Closing a semaphore

```
#include <semaphore.h>

int sem_close(sem_t * sem );
```

Returns 0 on success, or -1 on error

Open named semaphores are also automatically closed on process termination or if the process performs an `exec()`.

Closing a semaphore does not delete it. For that purpose, we need to use `sem_unlink()`.

7.4.3 Removing a semaphore

```
#include <semaphore.h>

int sem_unlink(const char * name );
```

Returns 0 on success, or -1 on error

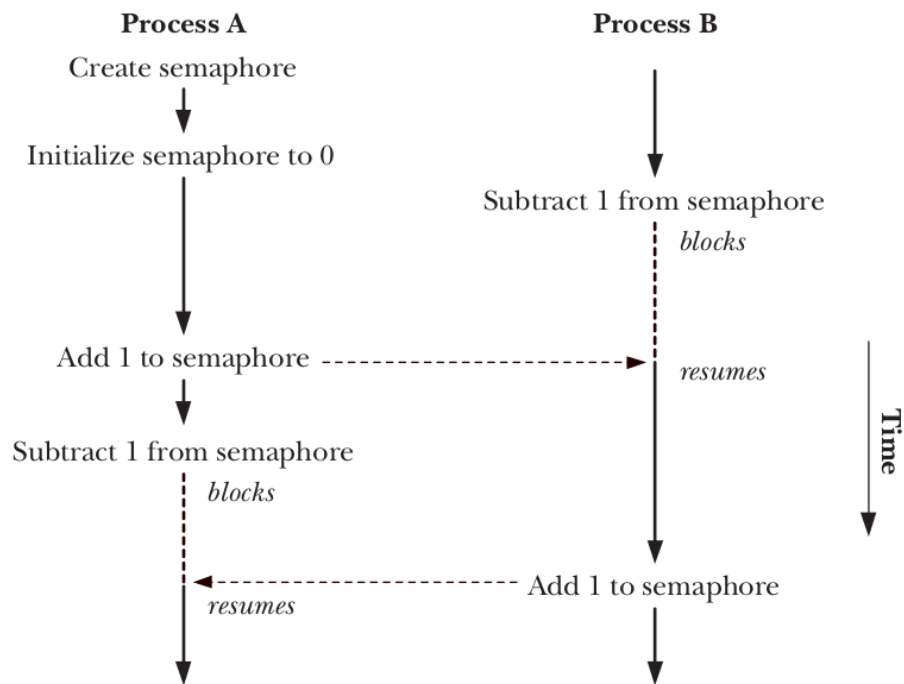
The `sem_unlink()` function removes the semaphore identified by name and marks the semaphore to be destroyed once all processes cease using it

POSIX semaphore is an integer ≥ 0

`Sem_wait()`:

If semaphore currently value $> 0 \Rightarrow$ return immediately

If semaphore currently value $= 0 \Rightarrow$ at that time, semaphore is decremented and `sem_wait()` return



Hình 7-10 Using semaphore to synchronize two process

[LSP/ipc/semaphore](#)

```

cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/semaphore$ ./p1

sem_open success

before sem_wait semaphore value = (2)

after sem_wait semaphore value = (1)

process 1: Executing Critical section

critical section executed

after sem_post semaphore value = (2)

cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/semaphore$ ./p2

sem_open success

semaphore val = (1)

Process 2 executing critical section

ret is (0)

process 2:

Process 2 executed critical section

cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/semaphore$ ./p3

sem_open success

sval = (0)

ret is (0)

process 3:

```

Hình 7-11 Ví dụ sử dụng named semaphore

7.4.4 Unnamed semaphores

Unnamed semaphore (also known as memory-based semaphore) are variables of type `sem_t` that are stored in memory allocated by the application

The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they shared.

Note:

- Named semaphore was present in file system similar to a regular file
- Unnamed semaphore does not exist on file system rather on volatile memory like RAM.

Unnamed semaphore operation (Same as named semaphore)

Two further functions are required:

`Sem_init()`: initialize a semaphore and inform the system between process or between the threads of a single process.

`Sem_destroy(sem)`: destroy a semaphore

7.4.4.1 Initializing an unnamed semaphore

```
#include <semaphore.h>
```

```
int sem_init(sem_t * sem , int pshared , unsigned int value );
```

Returns 0 on success, or -1 on error

Pshared argument indicates whether the semaphore is to be shared between threads or between process

If pshare = 0, shared between threads

Pshare \neq 0, shared between processes

7.4.4.2 Destroy an unnamed semaphore

```
#include <semaphore.h>
```

```
int sem_destroy(sem_t * sem );
```

Returns 0 on success, or -1 on error

The `sem_destroy()` function destroys the semaphore `sem`, which must be an unnamed semaphore that was previously initialized using `sem_init()`.

Good practice to destroy a semaphore only if no processes or threads are waiting on it

Example:

[LSP/ipc/semaphore/unnamed.c](https://lsp.icpc.org/semaphore/unnamed.c)

```

cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/semaphore$ ./unamed

threadFunc (thr1) running
semaphore availed in (thr1)
threadFunc End for (thr1)
threadFunc (thr2) running
semaphore availed in (thr2)
threadFunc End for (thr2)
threadFunc (thr3) running
semaphore availed in (thr3)
threadFunc End for (thr3)

value of glob after both thread running is = 30000

```

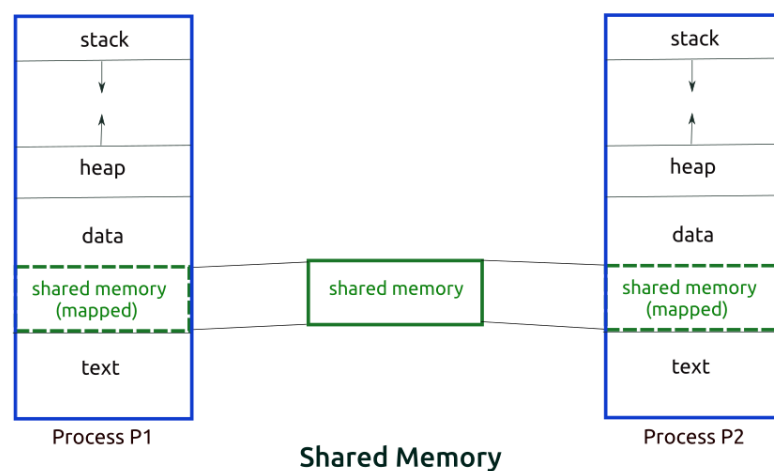
Hình 7-12 Ví dụ sử dụng unnamed semaphore

7.5 POSIX – Shared Memory

Step 1: Create shared memory

Step 2: Define size of shared memory

Step 3: Map shared memory on individual process



Hình 7-13 Shared memory

Trong các cơ chế IPC như PIPE hay message queue, cần thực hiện đầy đủ các bước gửi dữ liệu từ tiến trình này đến tiến trình khác, tuy nhiên với shared memory, không có

bất kì hành vi truyền dữ liệu nào cần thực hiện, các tiến trình đều có thể truy cập vào bộ nhớ chung.

Ở đây các tiến trình chia sẻ một vùng nhớ vật lý thông qua trung gian không gian địa chỉ của chúng.

Một shared memory tồn tại độc lập với các tiến trình khi một tiến trình muốn truy xuất đến vùng nhớ này, tiến trình phải gắn kết vùng nhớ chung đó vào không gian địa chỉ riêng của từng tiến trình và thao tác trên đó như một vùng nhớ riêng của mình.

Shared memory entries are present in 'dev/shm'

Shared memory is the fastest IPC mechanism, as no extra kernel data structures are involved.

7.5.1 To use a POSIX shared memory object

Step 1:

- Use the shm_open() function to open an object with a specified name
- The shm_open function is analogous to the open() system call
- It either create a new shared memory object or opens an existing one
- Shm_open() return a file descriptor referring to the shared memory

Note: The shared memory created newly is of length '0' bytes.

Step 2:

Define the length of shared memory

Step 3:

File descriptor obtained in the previous step is referenced in mmap()

This maps the shared memory object into the process's virtual address space

Note: Any read and write process's virtual memory will be actually read/write from shared memory (Part of physical memory)

7.5.2 Creating shared memory object

The shm_open() function creates and opens a new shared memory object or opens

an existing object. The arguments to `shm_open()` are analogous to those for `open()`.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
```

```
int shm_open(const char * name , int oflag , mode_t mode );
```

Returns file descriptor on success, or `-1` on error

7.5.3 Setting shared memory size

Newly create – size zero

User need to assign size required for the shared memory this is similar to creating a buffer with given size.

```
#include <unistd.h>
Int ftruncate (int fd, off_t length);
```

Fd is file descriptor obtained from `shm_open()` and length is the required length in bytes of shared memory

7.5.4 Mapping shared memory to process virtual memory

`Mmap()` is used to map the shared memory created to the process virtual map.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Input value:

Addr: address of process virtual memory to which the shared memory is mapped

If addr is NULL, then kernel chooses the address at which to create the mapping.

Length: length of the created shared memory

Prot: Describe the memory protection of mapping (`PROT_READ` – read, `PROT_WRITE` – write)

Flags: for shared memory flags use value 'MAP_SHARED' describing that the memory can be share.

Fd: file descriptor obtained from shm_open()

Return values of mmap()

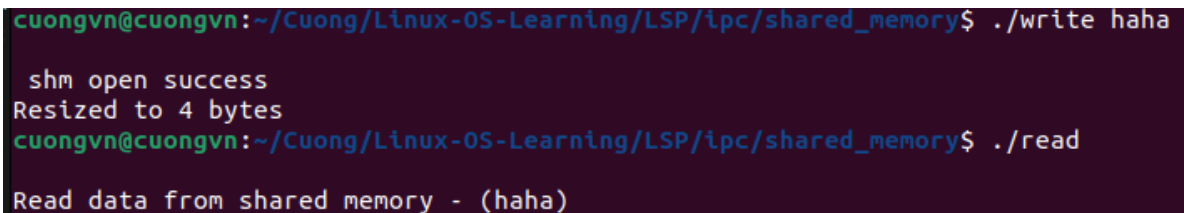
On success: The VM of process, to which the shared memory is mapped

On error: The value MAP_FAILED ((void*) -1) is returned, and errno is set to indicate the cause of error

Note: Refer to below command on terminal for more information: man 2 mmap

Example:[1]

[LSP/ipc/shared_memory](#)

A terminal window with a dark background and green text. The prompt is 'cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/shared_memory\$'. The user enters './write haha', and the output is 'shm open success' followed by 'Resized to 4 bytes'. Then the user enters './read', and the output is 'Read data from shared memory - (haha)'.

```
cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/shared_memory$ ./write haha
shm open success
Resized to 4 bytes
cuongvn@cuongvn:~/Cuong/Linux-OS-Learning/LSP/ipc/shared_memory$ ./read
Read data from shared memory - (haha)
```

Hình 7-14 Ví dụ sử dụng shared memory

TÀI LIỆU THAM KHẢO

- [1] M. Kerrisk, *The Linux programming interface: a Linux and UNIX system programming handbook*. San Francisco: No Starch Press, 2010.
- [2] “c - Why do pthreads’ condition variable functions require a mutex? - Stack Overflow.” <https://stackoverflow.com/questions/2763714/why-do-pthreads-condition-variable-functions-require-a-mutex> (accessed Jan. 31, 2023).
- [3] “Chi tiết bài học Thread Synchronization-Biến điều kiện.” <https://vimmentor.com/vi/lesson/thread-synchronization-bien-dieu-kien> (accessed Jan. 31, 2023).
- [4] “Chi tiết bài học Thread Synchronization-Biến điều kiện.” <https://vimmentor.com/vi/lesson/thread-synchronization-bien-dieu-kien> (accessed Jan. 31, 2023).
- [5] “Giao tiếp giữa các tiến trình trong Linux (Phần 1): Sử dụng Signal và Pipe,” Oct. 09, 2020. <https://viblo.asia/p/giao-tiep-giua-cac-tien-trinh-trong-linux-phan-1-su-dung-signal-va-pipe-Qpmlejxr5rd> (accessed Jan. 31, 2023).
- [6] “Giao tiếp giữa các tiến trình trong Linux (Phần 2): Sử dụng Share memory và Message Queue.” <https://viblo.asia/p/giao-tiep-giua-cac-tien-trinh-trong-linux-phan-2-su-dung-share-memory-va-message-queue-djeZ1yyYZWz> (accessed Jan. 31, 2023).
- [7] “Linux System Programming - A programmers/Practical Approach,” *Udemy*. <https://www.udemy.com/course/linux-system-programming-f/> (accessed Jan. 31, 2023).
- [8] “Processes vs Threads,” Sep. 04, 2017. <https://viblo.asia/p/processes-vs-threads-naQZR7Xv1vx> (accessed Jan. 31, 2023).