1. ­­OS : Là 1 chương trình nhưng nó chạy đầu tiên của hệ thống, nó sở hữu và có toàn quyền quyết định các thành phần khác của hệ thống

* Tính năng chính của OS : có nhiều các tính năng chính, k có gì nổi trội so với những cái khác

1. Kiến trúc hệ điều hành Linux
2. Có 2 chế độ : console và graphic
3. Console : chỉ có Terminal đen trắng , ­­k có chuột, phải dùng bàn phím, tuy nhiên rất tiết kiệm tài nguyên
4. Graphic : Có chuột, thao tác trực quan hơn.
5. Chương trình : Các file Binary đc buil từ source code và nằm trên hệ thống

Tiến trình : Là các chương trình nhưng đã đc Load vào hệ thống. Chúng tiêu thụ tài nguyên của CPU

1. Đa nhiệm : Các tác vụ tưởng chừng như chạy cùng 1 lúc nhưng k phải vậy, thời gian thực hiện 1 tác vụ rất nhanh, và ta sẽ chen những tác vụ khác nhau vào thời gian chờ của mỗi tác vụ, thứ tự thực hiện ta cài đặt được

* Có thể chạy đa nhiệm trên 1 core đc , tùy thuộc vào bộ lập lịch

1. File system trong Linux

Thư mục Roots là tổng thể ( như kiểu This PC trong windows ), trong Roots chứa các Folder khác , và từ Roots có thể băm ra các Folder khác

1. Quá trình biên dịch 1 chương trình C

* Giai đoạn 1 : Tiền xử lý
* Loại bỏ comment
* Mở rông các Macro
* Mở rộng các Include File
* Biên dịch câu điều kiện
* Kết quả thu được là 1 file “.i”

VD : gcc -E main.c -o main.i

* Giai đoạn 2 : dịch ngôn ngữ bậc cao sang asm :
* Mã nguồn từ file .i sẽ được chuyển thành file .s

VD : gcc -S main.i -o main.s

* Giai đoạn 3: Dịch file .asm sang mã máy ( Assembly):
* File .s ở giai đoạn trước được sử dụng tiếp trong giai đoạn này
* Thông qua Assembler, output mà chúng ta thu được là file .o

VD : gcc -c main.s -o main.o

* Giai đoạn 4: Giai đoạn Linker
* Mỗi file .o thu được ở giai đoạn 3 là 1 phần của chương trình
* Ở giai đoạn Linker sẽ liên kết chúng để thu được 1 file thực thi hoàn chỉnh

VD : gcc -o main main.o

1. Con trỏ

*void dir()*

*{*

*char\* buff;*

*getcwd(buff, 1000);*

*printf("%s\n",buff);*

*}*

*int main()*

*{*

*char check[50];*

*// dir(check);*

*//printf("%s\n",check);*

*dir();*

*return 0;*

*}*

* Code này sai vì chưa cấp phát bộ nhớ cho buff
* void check()
* {
* char \*str;
* str = readline("-> ");
* printf("%s\n",str);
* }
* Code này đúng vì tuy ta không trực tiếp cấp phát bộ nhớ cho str, nhưng hàm readline sẽ tự cấp phát bộ nhớ cho str

    //char \*s;

    char check[10]="134444";

    char \*str = check;

   printf("%s\n",str);

* Code này đúng vì tạo `str` để trỏ đến cùng vị trí bộ nhớ với `check`. Điều này có nghĩa là `str` về cơ bản là một con trỏ tới phần đầu của mảng ký tự `check`. Trong trường hợp này, không cần cấp phát bộ nhớ bổ sung cho `str` vì nó trỏ trực tiếp vào bộ nhớ hiện có bị `check` chiếm giữ

1. Device character:

* một loại thiết bị được trình điều khiển (driver) hỗ trợ truy cập thông qua các ký tự, thường là byte. Các thiết bị ký tự bao gồm những thiết bị mà dữ liệu được truyền qua dưới dạng dãy ký tự, không cần quan tâm đến vị trí cụ thể của dữ liệu.
* Ví dụ về thiết bị ký tự bao gồm bàn phím, chuột, hoặc các thiết bị I/O như cổng nối tiếp (serial port) hoặc giao diện truyền thông USB. Những thiết bị này nhận và truyền dữ liệu dưới dạng các ký tự, và việc đọc và ghi dữ liệu thường được thực hiện theo kiểu tuần tự (sequential).
* Trong Linux kernel programming, để tạo và quản lý một thiết bị ký tự, ta sử dụng cơ sở hạ tầng của kernel để đăng ký và sử dụng struct cdev
* Các ứng dụng người dùng có thể truy cập vào các thiết bị ký tự thông qua các hệ thống tệp đặc biệt trong thư mục /dev.
* Những thiết bị ký tự thường được sử dụng để truyền tải dữ liệu theo dạng luồng (stream), không cần quan tâm đến địa chỉ cụ thể của dữ liệu, và chúng thường được sử dụng trong các ứng dụng đòi hỏi tốc độ truyền tải dữ liệu nhanh và liên tục.

Các command lưu ý

* Mount

Lệnh này được sử dụng để kết nối các thiết bị lưu trữ (ví dụ: ổ đĩa, USB, phân vùng) vào hệ thống tệp của bạn. Dưới đây là một số ví dụ cơ bản về cách sử dụng lệnh "mount":

* Xem danh sách các thiết bị đã kết nối: *lsblk*
* Kết nối một thiết bị lưu trữ vào một thư mục cụ thể: *sudo mount /dev/sdXn /đường/dẫn/thư/mục*

Trong đó, /dev/sdXn là đường dẫn đến thiết bị (ví dụ: /dev/sda1), và /đường/dẫn/thư/mục là thư mục bạn muốn kết nối.

* Ngắt kết nối một thiết bị đã được mount: *sudo umount /đường/dẫn/thư/mục*
* Xem các thiết bị đã được mount: *mount*
* Kết nối tất cả các thiết bị được liệt kê trong /etc/fstab:

*sudo mount -a*

Lưu ý rằng để sử dụng lệnh mount, bạn cần có quyền sudo hoặc là người dùng có quyền đủ để thực hiện các thao tác này.

Bài 1 : Viết File .c trên Linux

# Bật terminal lên :

Gõ “vi ….” : … là tên file muốn đặt

Sau đấy console sẽ hiện lên để soạn thảo, lưu ý trong console không dùng được chuột .

# Gõ “i” và bắt đầu viết chương trình

VD : viết chương trình hello world

# Sau khi viết xong, nhấn ESC và shift + insert , sau ấy nhấn shift + “;” rồi gõ “wq” để lưu chương trình

# Gõ gcc “tên chương trình “ để bulit chương trình

# Gõ ls để check

# Khi này, mặc định file vừa viết mặc định ra “a.out”, ta phải xóa file này đi bằng cách gõ “rm a.out”

# Đổi tên ( VD tên cũ là hello.c và tên mới là hello ) bằng cách gõ

# gcc hello.c -o hello

# Sau khi đổi tên, ta chạy chương trình bằng cách gõ “./hello” ( hello là tên file mới “

# Ta có thể thay từ bước 4 đến 7 bằng lệnh : gcc -o hello hello.c

* Các ký tự đặc biệt trong Linux
* Bộ soạn thảo vi chạy ở hai chế độ khác nhau:

+ Chế độ dòng lệnh command mode, những gì được gõ vào sẽ được hiểu như là lệnh của vi. Vi có rất nhiều lệnh như: tìm kiếm, thay thế, xóa, lưu tâp tin…

+ Chế độ nhập văn bản insert mode, những gì được gõ vào được hiểu là nội dung của tập tin đang soạn thảo.

* Khi bắt đầu sử dụng lệnh**vi**, **vi** mặc định ở command mode. Ấn phím lệnh**i**, **a**, **o** hoặc Inserrt từ chế độ command mode để chuyển sang insert mode.

**i**để thêm ký tự trước con trỏ,

**a** để thêm ký tự sau con trỏ,

**o** để chèn thêm dòng mới sau con trỏ.

* Ấn Esc để chuyển đổi qua lại từ command mode với insert mode.
* Một số lệnh với vi:

**:set nu** hiện thị số dòng

**:set nonu** bỏ hiện thị số dòng

**Sử dụng phím mũi tên hoặc các phím h,l,j,k để dịch trái, phải, lên, xuống**

**:1 để nhảy đến dòng đầu tiên của file**

**:n nhảy đến dòng n**

**$ nhảy xuống cuối dòng**

**:$ nhảy đến dòng cuối của file.0 nhảy về đầu dòng**

**:0 nhảy về dòng đầu tiên của file.**

**dd xóa một dòng hiện tại**

**ndd xóa n dòng**

**/ hay ? để tìm kiếm**

**:w! lưu tập tin**

**:x! lưu tập tin và thoát**

**:wq ZZ lưu tập tin và thoát**

**:q! không lưu và thoát**

# Thuật ngữ :

* Các tiến trình chạy trên fork ground : các tiến trình khi chạy chiếm hết terminal
* Các tiến trình chạy trên back ground : khi chạy k chiếm terminal, để thực hiện, ta để & sau file chạy. VD : ./exam &

Các command trong Linux Terminal

* ls : check các thành phần
* cd …./ : di chuyển đến folder …
* cd : trở về Rootfs chung
* cat ….. : view file …. ( Muốn cat được File thì phải cd đến folder chứa file đó trước)
* touch : tạo 1 file mới
* Viết file trực tiếp từ terminal : echo “……” > ‘Ten file”
* pwd : check vị trí đang đứng
* rm -rf : xóa hẳn file
* cd ../ : lùi lại 1 thư mục
* mkdir ….: tao 1 thu muc moi
* move ….. “ten thu muc’/ : di chuyen file … den thu muc
* để chạy 1 chương trình C ( bao gồm thu viện tự tạo ) , ta cần xác định thư mục chứa thư viện ( tạo 1 folder lưu bằng lệnh mkdir và move ), sau đấy dùng lệnh

gcc -o example main.c hello.c -I include/

Với example và file chạy ( màu xanh lục ), main.c và hello.c là các file source, include là thư mục chứa file header

MakeFile

1. Makefile là 1 script bên trong có chứa các thông tin :

* Cấu trúc của 1 project (file,dependency)
* Các command line dùng để tạo-hủy file
* Chương trình make sẽ đọc ND trong makefile và thực thi nó

1. Cấu trúc của 1 lệnh MakeFile



* Cách sử dụng Makefile
* VD : ta chạy 1 chương trình có 3 file là main.c hello.c và hello.h, trong đó hello.h là file header
* Ta lần lượt tạo và viết 3 file trên bằng lệnh vim
* Để chạy , ta sử dụng lệnh : “gcc -o exam main.c hello.c -I .”
* File chạy là exam, để xóa file exam , ta dùng lệnh “rm -rf exam”

Đó là quy trình viết và chạy 1 file

* Để sử dụng Makefile :

VD :

all:

gcc -o exam hello1.c main1.c -I .

clear:

rm -rf exam

* Target là all và clear
* Để chạy lệnh “gcc -o exam main.c hello.c -I .”, thay vì gõ cả câu lệnh đó, ta sử dụng lệnh “make all “ ( với all là target trong makefile)
* Để chạy lệnh “rm -rf exam”, thay vì gõ cả câu lệnh đó, ta sử dụng lệnh “make clear “ ( với all là target trong makefile)
* Nguyên lý hoạt động của 1 makefile

Sau khi gọi lệnh make target , kiểm tra xem các dependency có mặt hay k, nếu có thì thực hiện action, nếu không thì báo lỗi

1. Biến trong makefile

* Cách gán 1 : đệ quy
* Vd : var1 = $(var) -> bất cứ khi nào var thay đổi giá trị thì var2 cũng thay đổi giá trị
* **Cách gán 2 : Trực tiếp**
* VD : var2 := $(var) -> gán trực tiếp giá trị của var2 là var , giá trị của var2 được giữ nguyên cho đến khi nó được gán lại
* Cách gán 3 :
* VD : var3 ?= $(var)-> kiếm tra xem var3 có giá trị hay chưa , nếu có rồi thì k gán lại nữa, nếu chưa có thì gán theo cách 1

Lưu ý : để lấy giá trị 1 biến trong Linux , ta dùng $(ten\_bien), để command 1 dòng, ta dùng #

1. Nếu trong 1 folder có nhiều makefile , để chạy makefile chỉ định , dùng lênh “make -f ten\_file”
2. Để chỉ định đến 1 rule trong 1 makefile : make -f ten\_makefile ten\_rule
3. Để ngăn ngừa lỗi có 1 file nào đó trùng tên rule trong makefile, ở đầu make file, ta dùng biến PHONY : .PHONY: liệt\_kê\_tên\_rule
4. Các biến đặc biệt :

VD : test: hello.c hello.h

@echo $@

@echo $<

@echo S^

* $@ : giá trị đầu bên trái của dấu :
* $< giá trị bên phải đầu tiên sau dấu :
* $^ toàn bị bên phải dấu :

Khi chạy chương trình rule test trên -> kết quả là

test

hello1.c

hello1.c hello1.h

1. Cách viết các rule

VD : ta có e file main.c check.c và check.h trong 1 folder

Ta viết 1 rule để build file main.c

Ta viết như sau:

CC := gcc

CCR := -I .

buil: main.o check.o

$(CC) -o $@ main.o check.o $(CCR)

Kết quả là :

gcc -c -o main.o main.c

gcc -c -o check.o check.c

gcc -o buil main.o check.o -I .

* File main.o là file build ra của main.c
* File check.o là file build ra của check.c
* Sau khi build ra 2 file trên, chương trình sẽ dùng chính chúng để build file buil

Tuy nhiên, nếu chỉ viết như trên, khi thay đổi trong file .h thì chương trình k detect ra được, nên ta cần viết thêm rule bằng cách

HFILE := check.h

%.o: %.c $(HFILE)

$(CC) -c -o $(@) $<

# Phân chia thư mục trong 1 folder làm việc

Gồm các folder sau

1. Bin : chứa file thực thi (màu xanh lục )
2. Inc : Chứa các file Header
3. Lib : Gồm StaticLib và ShareLib
4. Obj: Gồm các File .o
5. Src : gồm các thư viện .c
6. File main.c
7. Makefile
8. Readme.md

VD : Ta có chương trình gồm file main.c và hello.c , hello.h

* Các bước tạo Static Lib và chạy chương trình
* B1: Tạo các file .o và nhét nó vào thư mục obj, các câu lệnh sử dụng

gcc -c main.c -o obj/main.o -I inc/

gcc -c src/hello.c -o obj/hello.o -I inc/

* B2 : Tạo ra static Lib và nhét nó vào thư mục Lib, câu lệnh sử dụng

ar rcs lib/static/libhello.a obj/hello.o

* B3 : tạo file thực thi và nhét nó vào trong thư mục bin bằng cách Link các file .o lại với nhau, câu lệnh sử dụng :

gcc obj/main.o -L lib/static -l hello -o bin/exam

File Trong Linux

# Tổng quan

* Regular File : Các file thông thường như text file, exe file
* Dirẻctories File : file chứa danh sách các file khác
* Character Devices File : File đại diện cho các thiết bị không có địa chỉ vùng nhớ
* Block Device File: File đại diện cho các thiết bị có địa chỉ vùng nhớ
* Link file : file đại diện cho 1 file khác
* Socket file : file đại diện cho 1 socket
* Pipe File : file đại diện cho 1 pipe
* Để lấy ra thông tin toàn bộ file , ta dùng ls -l
* Số hard link của file : số file cùng trỏ đến 1 vùng nhớ ( ta có thể kiểm tra bằng lệnh ls -l)
* Loại file và quyền của file ( kiểm tra bằng ls -l)
* ở cột ngoài cùng bên trái , nếu là dấu “-“ thì là regular file
* nếu là chữ “d” thì là Direct file
* chứ “b” là Block Device file

…

* quyền của file

User permission : r-w-x: 111(7)

Group permission: r-w-x: 111(7)

Others permission: r-w-x: 111(7)

* để sửa quyền của file -> lệnh chmod :

VD : chmod o-r : lấy đi quyền read của Others

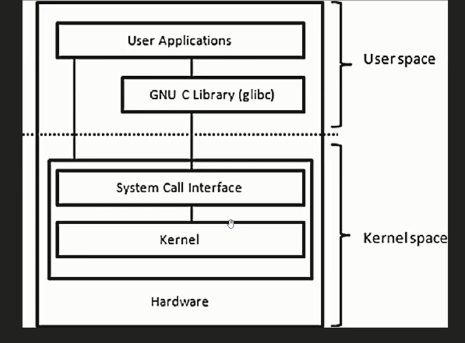
Chmod u+w : thêm quyền write cho User

Để thay đổi User , ta dùng : sudo chown ten\_user ten\_file

Để thay đổi Group , ta dùng : sudo chown ten\_user:ten\_group ten\_file

# Thao tác với file

Các system call cơ bản :



1. open :

**int open(const char \****pathname***, int** *flags***);**

**int open(const char \****pathname***, int** *flags***, mode\_t** *mode***);**

* \*const char \*pathname : tên file cần tạo , VD : test.txt
* Flags : các cờ đc quy định trước:

VD : O\_RDONLY, O\_WRONLY, or O\_RDWR. These request opening the file read-only, write-only, or read/write

* O\_CREAT : nếu file chưa có sẵn thì tạo ra nó
* Mode : tạo quyền cho file , VD : 0666
* Giá trị trả về của hàm open là 1 File Descriptor

1. Write

**ssize\_t write(int** *fd***, const void** *buf***[.***count***], size\_t** *count***);**

* **Fd là** giá trị của File Descriptor của file cần ghi
* **Buff** là chuỗi cần ghi vào
* Count là kích thước chuỗi ký tự cần ghi vào
* Hàm này trả về kích thước thực tế ghi đc vào file

1. Read
2. Lseek : di chuyen con tro file

**off\_t lseek(int** *fd***, off\_t** *offset***, int** *whence***);**

* Fd : ….
* Offset : Giá trị bù vào, nếu offset bằng 2 thì con trỏ file sẽ ở vị trí 2
* Whence :

SEEK\_SET

Offset tính từ đầu bên trái

SEEK\_CUR

Offset tính từ vị trí hiện tại

SEEK\_END

Offset tính từ cuối file

1. Close: truyen vao fd

Giải phóng tài nguyên cho process

# Quản lý file trong Linux

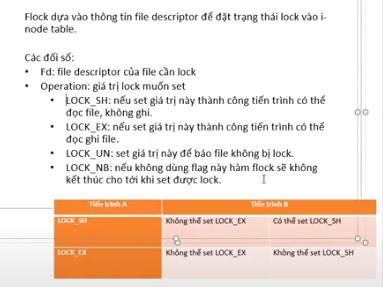
* Kernel điều khiển việc tương tác giữa các tiến trìn và file thông qua 3 bảng:

+ File descriptor table

+ Open file table

+ I-node table

* flock(): một cơ chế ngăn chặn nhiều tiến trình hoặc luồng truy cập đồng thời vào một tệp hoặc một phần của tệp. Điều này rất quan trọng trong các tình huống mà tính nhất quán của dữ liệu là rất quan trọng, chẳng hạn như khi nhiều quy trình cần đọc hoặc ghi vào một tệp được chia sẻ.
* cú pháp : flock (fd, operation)



Nếu lock không thành công , giá trị trả về của hàm flock() là -1

Process

# Khái niệm

* Phân biệt process (tiến trình) và program(chương trình) :
* Chương trình là 1 nhóm câu lệnh, các câu lệnh đấy tạo ra 1 file thực thi trên ổ cứng
* Tiến trình là 1 chương trình đg được thực thi , và đc phép dùng các tài nguyên của hệ thông. Mỗi 1 process có 1 mã định danh ID (PID), đây là 1 số nguyên dương và duy nhất cho mỗi process trên hệ thống
* Để tìm PID của 1 tiến trình , dùng command “ps aux | grep …” với “…” là tên file running
* Memory layout

1. **Text segment** : chứa các lệnh ngôn ngữ máy -> read only
2. **Initialzed data segment** : bao gồm các biến **global** và **static** được khởi tạo 1 cách tường minh

Vd : int a = 7;-> Tường minh

Segment này có quyền Read-WRite

1. **UnInitialzed data segment** : Bao gồm biến global và static không tường minh

VD : int a;

Segment này có quyênhf Read-write

1. **Stack Segment** :Chứa các biến cục bộ

Có thể co – dẫn vùng nhớ bằng cách cấp phát/giải phóng stack frame

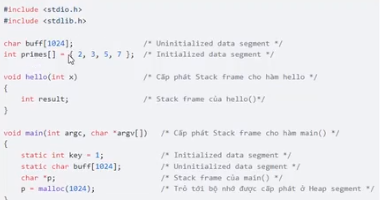
* Có quyền read-write

1. **Heap Segment** : dành cho việc cấp phát động, sử dụng các hàm như alloc(), calloc(),malloc(),

Heap cũng có thể co dãn như stack. Điểm kết thúc của Heap gọi là program break

* Có quyền read-write

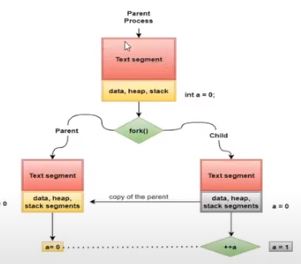
VD :



# Thao tác với process

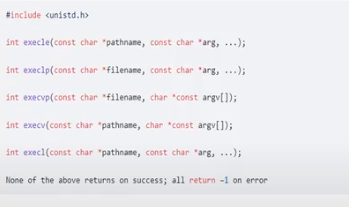
1. Tạo 1 tiến trình mới

* Sử dungk system call fork()
* Tiến trình gọi fork() đgl tiến trình cha ( parent process)
* Tiến trình mới đc tạo ra đgl tiến trình con (chill process)
* Tiến trình init là tiến trình đầu tiên đc chạy, là cha của mọi tiến trình khác với pid là 1



1. Chạy 1 chương trình mới

* Trong nhiều trường hợp bạn đg có 1 tiến trình A đang thực thi và bạn muốn chạy 1 chương trình B nào đó từ tiến trình A hoặc con của nó. Ta dùng họ hàm exec



* exel() function

*int execl(const char \*path, const char \*arg0, const char \*arg1, ..., const char \*argn, (char \*) NULL);*

* path: Đường dẫn file của file thực thi mà bạn muốn thực thi.
* arg0, arg1, ..., argn: Các đối số được truyền cho quy trình mới. Đối số đầu tiên ( arg0) thường là tên của chương trình đang được thực thi.
* (char \*) NULL: Một trọng điểm cho biết sự kết thúc của danh sách đối số*.*

*VD0 : include <unistd.h>*

*int main(void) {*

*char \*file = "/usr/bin/echo";*

*char \*arg1 = "Hello world!";*

*execl(file, file, arg1, NULL);*

*return 0;*

*}*

VD1 : char \*path = "./hello";

execl(path,"hello",NULL);

VD2 : char \*path ="../test/check";

execl(path,"check",NULL);

* Execlp()

*#include <unistd.h>*

*int execlp(const char \*file, const char \*arg0, ..., (char \*)NULL);*

execlp() rất giống với execl() . Tuy nhiên, execlp() sử dụng biến môi trường PATH để tìm tệp. Do đó, đường dẫn đến tệp thực thi là không cần thiết:

* file: Tên của file thực thi. Nếu nó không chứa dấu gạch chéo (/), hàm execlp sẽ tìm kiếm tệp trong các thư mục được liệt kê trong biến môi trường PATH.
* arg0: Tên của chương trình, thường giống với file.
* Đối số biến (...): Danh sách các đối số đại diện cho đối số dòng lệnh của chương trình mới. Danh sách phải được kết thúc bằng con trỏ NULL.

VD :

*include <unistd.h>*

*int main(void) {*

*char \*file = "echo";*

*char \*arg1 = "Hello world!";*

*execlp(file, file, arg1, NULL);*

*return 0;*

*}*

* Execle()

Điểm khác biệt chính là execle cho phép bạn chỉ định rõ ràng các biến môi trường cho chương trình mới.

*#include <unistd.h>*

*int execle(const char \*path, const char \*arg0, ..., char \*const envp[]);*

* path: Đường dẫn tới file thực thi.
* arg0: Tên chương trình, thường giống với đường dẫn.
* Đối số biến (...): Danh sách các đối số đại diện cho đối số dòng lệnh của chương trình mới. Danh sách phải được kết thúc bằng con trỏ NULL.
* envp: Một mảng các chuỗi biểu thị các biến môi trường cho chương trình mới. Phần tử cuối cùng của mảng này phải là con trỏ NULL.
* Execvp()

Tính năng chính của execvp là nó tìm kiếm tệp thực thi được chỉ định trong các thư mục được liệt kê trong biến môi trường PATH.

*#include <unistd.h>*

*int execvp(const char \*file, char \*const argv[]);*

* file: Tên của file thực thi. Nếu nó không chứa dấu gạch chéo (/), execvp sẽ tìm kiếm tệp trong các thư mục được liệt kê trong biến môi trường PATH.
* argv: Một mảng các chuỗi biểu thị các đối số dòng lệnh của chương trình mới, trong đó phần tử cuối cùng phải là con trỏ NULL.

VD : *char \*args[] = {"ls", "-l", NULL};*

*execvp("ls", args);*

1. Kết thúc 1 tiến trình

* Kết thúc bình thường : gọi hàm exit()
* Kết thúc bất thườnglệnh kill() :

Câu lệnh thường dùng : kill -9 …..

“…” là PID của tiến trình

1. Wait()

* System call wait() được sử dụng để theo dõi trạng thái kết thúc của 1 tiến trình con mà tiến trình cha tạo ra

System call wait() được gọi trong tiến trình cha để chờ cho đến khi 1 trong các tiến trình con của nó bị kết thúc và trả về trạng thái kết thúc của tiến trình con đó vào con trỏ “status”.

Tại thời điểm wait() được gọi, nếu chưa có tiến trình con nào kết thúc, wait() sẽ block cho đến khi có 1 tiến trình con bị kết thúc. Nếu có 1 tiến trình con đã kết thúc từ trước khi wait() được goi, nó sẽ return ngay lập tức. Nếu con trỏ “status” không NULL, nguyên nhân kết thúc của tiến trình con sẽ được lưu vào số nguyên mà “status” trỏ đến.

System call wait() tồn tại 1 số hạn chế khi nó chỉ có thể theo dõi 1 tiến trình bị kết thúc tiếp theo trong số tất cả các tiến trình con, và sẽ block tiến trình nếu chưa có tiến trình con nào bị kết thúc. Trong nhiều trường hợp chúng ta chỉ muốn theo dõi 1 tiến trình cụ thể.

*#include <sys/wait.h>*

*pid\_t wait(int \*status);*

*/\*Trả về PID của tiến trình con kết thúc, hoặc –1 nếu lỗi\*/*

* . Trong nhiều trường hợp chúng ta chỉ muốn theo dõi 1 tiến trình cụ thể -> waitpid()

Giá trị trả về cũng như biến nguyên nhân kết thúc “status” của waitpid() giống với wait(). Trong đó đối số “pid” xác định tiến trình con mà chúng ta muốn theo dõi, với quy ước như sau:

*#include <sys/wait.h>*

*pid\_t waitpid(pid\_t pid, int \*status, int options);*

*/\*Trả về PID của tiến trình con, 0 (see text), hoặc –1 nếu lỗi\*/*

Nếu pid >0, chờ tiến trình con có định danh là pid

Nếu pid = 0, chờ bất kỳ tiến trình con nào nằm trong nhóm với tiến trình cha

Nếu pid < -1, chờ bất kỳ tiến trình con có process group ID (chúng ta sẽ học ở bài sau) bằng giá trị tuyệt đối với pid. Ví dụ pid ==-200 thì sẽ chờ tiến trình con có pid 200

Nếu pid == -1, chờ bất kỳ tiến trình con nào (giống với wait())

Đối số “options” là 1 bit mask có thể OR với 0 hoặc 1 trong các flag WUNTRACED, WCONTINUED, WNOHANG.

* Ta có thể kiểm tra tiến trình con kết thúc ra sao bằng cách sau

   if (WIFEXITED(status)) {

            printf("Child process  exited with status: %d\n", WEXITSTATUS(status));

        } else {

            printf("Child process did not exit normally and status is : %d\n", WEXITSTATUS(status));

        }

1. Zombie process vs orphan process

* Một tiến trình zombie là một tiến trình con đã kết thúc nhưng trạng thái thoát của nó vẫn chưa được tiến trình cha thu thập.

Khi một tiến trình con thoát ra, nó sẽ trở thành một zombie cho đến khi tiến trình cha gọi lệnh gọi hệ thống wait()hoặc waitpid()để thu thập trạng thái thoát của nó.

Các tiến trình zombie vẫn có một mục trong bảng tiến trình, chứa thông tin về tiến trình, nhưng chúng không tiêu tốn tài nguyên hệ thống ngoài mục này.

Các chương trình được thiết kế phù hợp phải đảm bảo rằng quy trình cha nhanh chóng thu thập trạng thái thoát của các quy trình con để ngăn chúng trở thành zombie.

* Để ngăn chặn zombie process, ta thường để wait() hoặc waitpid() ở tiến trình cha
* Luư ý : không thể kill trực tiến tiến trình zombie, nếu muốn kill zombie, phải kill thằng cha của nó

* Một tiến trình mồ côi là một tiến trình con mà tiến trình cha của nó đã kết thúc trước khi tiến trình con đó hoàn thành việc thực thi của nó.

Các tiến trình mồ côi tiếp tục thực thi ở chế độ nền trong tiến trình init (ID tiến trình 1 trong các hệ thống giống Unix), tiến trình này sẽ trở thành tiến trình mẹ mới của chúng.

Các tiến trình mồ côi không trở thành zombie vì chúng vẫn có tiến trình cha (tiến trình init) để thu thập trạng thái thoát khi chúng kết thúc.

Các quy trình mồ côi có thể được các quy trình khác áp dụng nếu chúng kết thúc trước quy trình cha mới của chúng.

Thread

# Khái niệm

Tương tụ như process, thread được tạo ra với nhiều mục đích đồng thời xử lý nhiều công việc 1 lúc ( Multi thread)

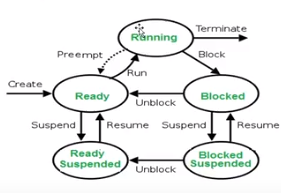
* Thread ID : định danh thread trong process . Là 1 số nguyên hoặc 1 struct( thường sử dụng struct)

Tuy nhiên , cần lưu ý vài điều :

* Process ID là duy nhất trong hệ thống nhưng Thread ID là duy nhất trong 1 process
* Thread ID được đại diện bởi kiểu : pthread\_t
* Để so sánh các ID Thread với nhau, ta cần sử dụng 2 hàm sau : pthread\_self() và pthread\_equal()
* pthread\_self(): tương đương với hàm getpid()
* pthread\_equal(…) : truyền vào ID của 2 thread và thực hiện so sánh. Hàm trả về 0 nếu Id của thread khác nhau, trả về khác 0 nếu ID của 2 thread bằng nhau

# Nguyên lý hoạt động

* Trên multi cỏe , nhiều thread hoạt động song song
* Nếu 1 thread bị block, các thread khác vẫn hoạt động bình thường
* Mỗi khi thread đc tạo, chúng sẽ được đặt trong stack segment
* Context switching ( chuyển đổi ngữ cảnh)



# Thao tác với Thread

* Chương trình ( program) được khởi chạy và trở thành 1 tiến trình ( process) . Khi ấy ,bản thân tiến trình đó là single\_thread
* Tiến trình tạo nhiều hơn 1 thread đgl Multi Thread
* Ta có thể kết luận rằng mọi tiến trình đều có ít nhất 1 thread. Trong đó thread chưa hàm main đc gọi là main thread  
  a. Tạo 1 thread mới

#include <pthread.h>

*int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,*

*void \*(\*start\_routine)(void \*), void \*arg);*

thread: Một con trỏ tới một pthread\_tbiến nơi ID luồng sẽ được lưu trữ sau khi luồng được tạo.

attr: Một con trỏ tới pthread\_attr\_tcấu trúc chứa các thuộc tính luồng. Bạn có thể chỉ định xem NULLbạn có muốn sử dụng các thuộc tính mặc định hay không.

start\_routine: Một con trỏ tới hàm sẽ được thực thi bởi luồng mới. Hàm này sẽ nhận một đối số duy nhất thuộc loại void\*và trả về a void\*. Nó đại diện cho điểm vào của thread mới.

arg: Một con trỏ tới đối số sẽ được truyền cho hàm start\_routine khi luồng được tạo.

1. Kết thúc 1 thread

* int pthread\_exit(void \*retual)
* Retual trả về giá trị kết thúc của thread
* Int pthread\_cancel(pthread\_t thread)
* Gửi yêu cầu kết thúc đến 1 thread cụ thể nào đó, trả về 0 nếu thành công, nhỏ hơn 0 nếu k thành công

1. Quản lý thread

Có các kiểu kết thúc thread :

- Kết thúc bình thường

- Kết thúc khi gọi pthread\_exit()

- Kết thúc khi gọi pthread\_cancel(thread\_id)

-> Bất kỳ 1 thread nào khi gọi exit() , hoặc main thread kết thúc thì all các thread khác kết thúc ngay lập tức

* Joinable Thread

Để thu đc giá trị kết thúc của 1 thread khác, ta dùng hàm pthread\_join()

* Hoạt động Joining
* Pthread\_join(pthread\_t thread, void \*retual): sẽ block cho đến khi 1 thread kết thúc ( ID của thread đc truyền vào làm đối số của hàm), nếu thread đó đá kết thúc thì hàm join return ngay
* Khi 1 thread kết thúc , về cơ bản nó sẽ đc xử lý như 1 zombie process, nếu lượng zombie process càng ngày càng lớn, thì có thể ta k thể tạo thêm thread đc nữa. Vai trò của hàm join() giống waitpid()-> Để tránh zombie , ta nên dùng pthread\_join()
* Detach a thread
* Mặc định khi 1 thread đc tạo ra, nó là joinable, tức khi thread kết thúc thì 1 thread khác có thể thu đc trạng thái trả về qua hàm pthread\_join()
* Tuy nhiên, nhiều TH ta không cần quan tâm trạng thái kết thúc của 1 thread mà chỉ cần hệ thống tự động clean và remove thread tự động
* Đặt thread vào trạng thái detach thông qua gọi hàn pthread\_detached(pthread\_t id)
* Điểm khác biệt mấu chốt giữa pthread\_join() và pthread\_detach() là :
* Với pthread\_join(id\_x) : luồng chứa hàm này phải đợi thread có id\_x thực hiện xong thì mới được chạy tiếp
* Với pthread\_detach(id\_x) : được sử dụng để thông báo cho hệ thống rằng luồng đang gọi nó phải được tách ra, nghĩa là các tài nguyên được liên kết với luồng sẽ tự động được giải phóng khi luồng thoát ( không cần chờ đợi gì hết)

VD : Đoạn code sau

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

pthread\_t id1,id2;

typedef struct{

    char name[20];

    char Idol[20];

}data\_check;

static void \*func1(void \*para)

{

    data\_check \*data = (data\_check \*)para;

    printf("I am thead 1\n");

   // pthread\_join(id2,NULL);

    while(1)

    {

    printf("%s %s\n",data->Idol, data->name);

    sleep(1);

    }

   // pthread\_exit(NULL);

}

static void \*func2(void \*para)

{

   printf("I am thread 2\n");

   int t=5;

   while(t--)

   {

    printf("Hello Tuan\n");

    sleep(1);

   }

    pthread\_exit(NULL);

    //exit(1);

}

int main()

{

    data\_check data={0};

    strncpy(data.name, "TuanTNT19",sizeof(data.name));

    strncpy(data.Idol, "Irene Bae",sizeof(data.Idol));

    if (!pthread\_create(&id1,NULL,&func1,&data))

    {

        printf("Create thread1\n");

    }

    if (!pthread\_create(&id2,NULL,&func2,NULL))

    {

        printf("Create thread2\n");

    }

   // sleep(5);

   // pthread\_cancel(id2);

   // sleep(2);

    //pthread\_cancel(id1);

    //pthread\_join(id2,NULL);

   // pthread\_detach(id2);

    printf("OK ??\n");

    while(1);

    return 0;

}

* Nếu trong hàm main, pthread\_join(id2) được gọi thì hàm main phải đợi thread 2 thực thi xong mới in ra chữ “OK”
* Nếu trong hàm main, pthread\_detach(id2) được gọi thì hàm main k cần đợi thread 2 kết thúc mà nó sẽ in ra chữ “OK” , sau ấy thực hiện các nhiệm vụ khác của 2 task trong while(1)

1. Thread synchonization

* Một trong các điểm mạnh của thread là việc chia sẻ dữ liệu thông qua các biến global

Tuy nhiên tồn tại vài vấn đề về đồng bộ

* Điều gì xảy ra nếu nhiều thread cùng chỉnh sửa 1 biến vào cùng 1 thời điểm hay 1 thread đang cố đọc giá trị của 1 biến nào đó trong khi thread khác đang sửa đổi biến đó
* Khái niệm Atomic / Nonatomic
* Atomic : Tại 1 thời điểm chỉ có 1 thread duy nhất đc truy cập vào tài nguyên đc chia sẻ ( share resouces ) -> Atomic an toàn
* Nonatomic : Nhiều thread có thể truy cập vào share reseouces tại cùng 1 thời điểm-> Nonatomic không an toàn
* Mutex: 1 kỹ thuật được dùng để bảo đảm rằng 1 thời điểm chỉ có 1 thread mới đc phép truy cập vào share resouces.
* Triển khai mutex theo các bước
* Khởi tạo khóa Mutex
* Thực hiện khóa mutex cho các share resouces trước khi vào critical section
* Thực hiện truy cập vào share resouces
* Mở khóa / giải phóng Mutex
* Khóa Mutex có 1 kiểu dữ liệu là pthread\_mutex\_t. Trước khi sử dụng ta phải khởi tạo khóa Mutex ( Cấp phát tĩnh hoặc động)

VD 1: Pthread\_mutex\_t mtx = PTHEAD\_MUTEX\_INITIALIZER // khởi tạo tĩnh

VD2 : Int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr)

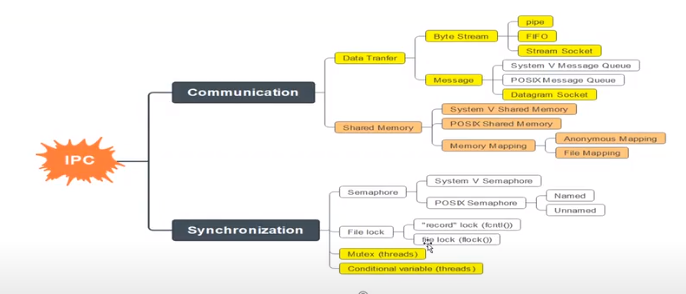
// khởi tạo đông

* Nếu k dùng Mutex nữa , ta phải hủy bằng hàm pthread\_mutex\_destroy(). Khởi tạo tĩnh k cần gọi hàm này
* Sau khi khởi tạo, khóad mutex ở trạng thái unlock

Nếu muốn khóa cần hàm pthread\_mutex\_lock(pthread\_mutex\_t \*mutex) , mở khóa cần hàm pthread\_mutex\_unlock()

* Khi khóa mutex ở trạng thái unlock, pthread\_mutex\_lock() sẽ return ngay lập tức. Ngược lại, nếu mutex ở trạng thái locked bởi 1 mutex khác thì pthread\_mutex\_lock() sẽ bị block cho đến khi đc unlock

IPC



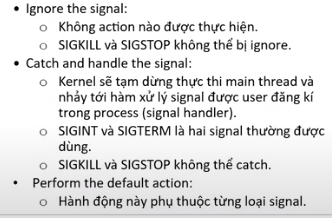
# Khái niệm : IPC ( giao tiếp liên tiến trình)

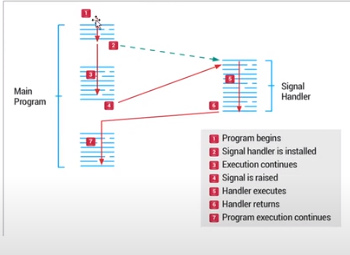
1. Signal

# Signal là 1 trong những phương thức IPC lâu đời nhất

# Signal là 1 sofrware interrupt, là 1 cơ chế xử lý sự kiện bất đồng bộ(async)

* 1 vài signal quan trọng : để kiểm tra các signal -> kill -l
* Giết đi 1 tiến trình -> kill -9
* Stop 1 tiến trình -> kill -19
* Tiếp tục 1 tiến trình : kill -18
* Signal Lifecycle ( vòng đời của signal)
* Generation : 1 tín hiệu đc tạo ra ( raised, sent, generation)
* Delivery: signal đc pending cho tới khi nó đc phân phối
* Processing: 1 tín hiệu đc phân phối, nó có thể được xử lý bởi nhiều cách
* Processing





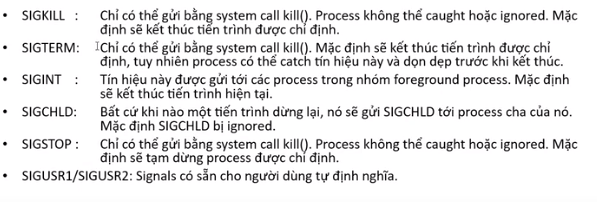
* Ta đăng ký việc xử lý 1 signal thông qua system call signal()
* Signal là 1 software interrupt nên nhạy cảm về mặt thời gian thực thi. Khi signal handler thực thi nó sẽ chiếm hoàn toàn cpu của process
* Cần hoàn thành và thoát ra khỏi signal handler nhanh nhất có thể
* Các đối số của hàm signal

Sighandler\_t signal( int signo, sighandler\_t handler)

Signo : signal number

Handler: signal handler

* Các signal thường xuyên sử dụng

  
- Lưu ý : SIGINT : chỉ kết thúc đc các tiến trình fork ground

* Blocking và Unblocking signal
* Khái niệm : signal làm gián đoạn process -> blocking muốn giải quết vấn đề này
* Mỗi process có thể chỉ định signal cụ thể nào muốn block. Signal bị block vẫn xảy ra nhưng nó đc kernel giữ trong hàng đợi ( pending)
* Tín hiệu đc gửi tới process chỉ sau khi nó đc unblocking
* Danh sách signal bị block đgl “signal mask”
* Signal Mask : được đại diện bởi 1 struct là sigset\_t

*typedef struct {*

*usigned long sig[\_NSIG\_WORDS];*

*} sigset\_t;*

* 1 vài hàm thao tác vs signal mask
* sigfillset(newset) : Toàn bộ các trường trong newset mask thành 1
* sigemptyset ( newset) : ........................................0
* sigaddset ( newset, SIGINT ) : trường SIGINT trong newset đc điền là 1
* sigdelset ( newset, SIGINT ) : ......................................0
* sigismember ( newset, SIGILL) : kiểm tra , nếu hàm trả về 1 thì SIGILL đang được set trong mask của mình nếu trả về 0 thì SIGILL k đc set trong mask của mk, trả về -1 nếu lỗi
* sigprocmask( int how, const sigset\_t \*newset, sigset\_t \*oldset)
* how :

+ SIG\_SETMASK : signal mask của process sẽ bị thay đổi thành newset-> thay thế

hoàn toàn các trường trong signal mask bằng newset

+ SIG\_BLOCK : newset sẽ được thêm vào signal mask ( phép OR của newswt vs

process set)-> set các trường mang bit 1 trong newset thành trạng thái block trong

signal mask

+ SIG\_UNBLOCK : newset sẽ bị xóa khỏi signal mask (các trường = 1 trong cả newset và process set đc ghi là 0 tức unblock) -> sét các trường mang bit 1 trong newset thành trạng thái unblock

trong signal mask

* oldset : Nếu oldset khác NULL , sigprocmask sẽ lấy ra đc signal mask hiện tịa và lưu

vào oldset

* Nếu newset là NULL , sigprocmask sẽ bỏ qua việc thay đổi quá trị của signal mask , nhưng nó sẽ lấy ra được signal mask hiện tại và lưu nó vào oldset. Nói cách khác, truyền NULL vào set như 1 cách để lấy ra signal mask hiện tại
* Để kiểm tra signal bất kỳ có đg bị block hay không - Lấy ra mask của process-> sigismember (oldset,SIGINT)
* -> để lấy ra mask của 1 process sigprocmask(SIG\_UNBLOCK/BLOCK/SETMASK, NULL,oldset);

SOCKET

# Tổng quan

* Cư chế truyền thông cho phép các tiến trình giao tiếp trên cùng / khác thiết bị
* Socket đc đại diện bởi 1 file socket descriptor
* Thông tin mô tả file socket : Domain, Type, protocol

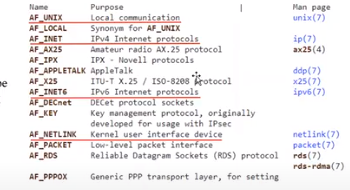
1. Domain ( Tên miền ): Tiến trình giao tiếp trên cùng / khác thiết bị

* Internet domain : chia thành IPV4 vs IPV6
* UNUX Domain:

1. Type Mô tả cơ chế truyền nhận dữ liệu

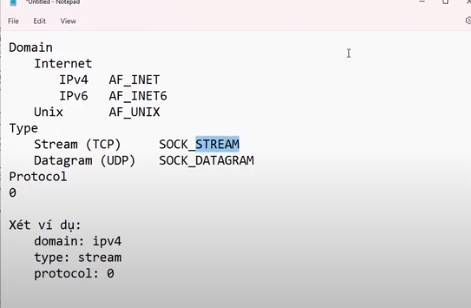
* Stream : Dựa trên giao thức TCP -> Dùng khi truyền chuỗi bit ( Tin nhắn , file)
* Datagram : UDP -> Dùng khi truyền gói tin ( audio, video ) vì tôc độ nhanh hơn stream

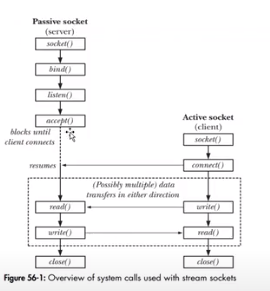
1. Protocol : Cách thức đóng gói dữ liệu ( Thường hay để là 0 khi lập trình )

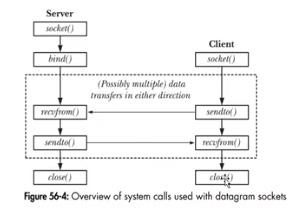


# Flow hoạt động

* Stream socket yêu cầu 1 kết nối trc khi giao tiếp







# Cac ham hay dung

1. Socket(): Hàm này trả về -1 nếu k thành công. Nếu thành công thì trả về giá trị của bộ mô tả têp (bộ mô tả tệp của máy chủ nếu nó đặt trong chương trình máy chủ/ bộ mô tả tệp máy khách nếu nó đặt trong chương trình máy khách)

*#include <sys/types.h>*

*#include <sys/socket.h>*

*int socket(int domain, int type, int protocol);*

* domain: Chỉ định miền liên lạc, chẳng hạn như AF\_INET cho IPv4 hoặc AF\_INET6 cho IPv6.
* type: Chỉ định loại ổ cắm, chẳng hạn như SOCK\_STREAM cho ổ cắm luồng (TCP) hoặc SOCK\_DGRAM cho ổ cắm datagram (UDP).
* protocol: Chỉ định giao thức cụ thể sẽ được sử dụng (hoặc 0 cho giao thức mặc định trong miền và loại được chỉ định).

1. Bind() : Hàm liên kết socket vs 1 địa chỉ IP trên máy cục bộ. Trả về 0 nếu thành công, -1 nếu thất bại

*#include <sys/types.h>*

*#include <sys/socket.h>*

*int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);*

* sockfd: Bộ mô tả tệp của socket được socket()hàm trả về. ( bộ mô tả tệp của máy chủ )(
* addr: Một con trỏ tới a struct sockaddrchứa thông tin chi tiết về địa chỉ cục bộ (địa chỉ IP và số cổng) mà ổ cắm sẽ được liên kết.
* addrlen: Kích thước của cấu trúc địa chỉ.
* Struct sockaddr\_in : cấu trúc dữ liệu được sử dụng trong bối cảnh lập trình mạng trên Linux (và các hệ điều hành giống Unix khác) để thể hiện địa chỉ ổ cắm IPv4. Nó là một phần của API socket và thường được sử dụng với các hàm như bind(), connect(), accept()và các hàm khác.

*struct sockaddr\_in {*

*sa\_family\_t sin\_family; // Address family (AF\_INET for IPv4)*

*in\_port\_t sin\_port; // Port number (in network byte order)*

*struct in\_addr sin\_addr; // IPv4 address*

*char sin\_zero[8]; // Padding to make the structure the same size as struct sockaddr*

*};*

* **sin\_family**: Chỉ định họ địa chỉ và đối với địa chỉ IPv4, nó được đặt thành **AF\_INET**.
* **sin\_port**: Chứa số cổng theo thứ tự byte mạng (dùng **htons()**để chuyển đổi).
* **sin\_addr**: Biểu thị địa chỉ IPv4 bằng cấu trúc kiểu **struct in\_addr**.
* **sin\_zero**: Phần đệm để làm cho cấu trúc có cùng kích thước với **struct sockaddr**

**VD :**

*#include <netinet/in.h>*

*struct sockaddr\_in server\_addr;*

*// Set up the server address struct*

*server\_addr.sin\_family = AF\_INET;*

*server\_addr.sin\_port = htons(8080); // Port number (in network byte order)*

*// Now, you can use this structure in functions like bind(), connect(), etc.*

1. listen(): . Hàm này listen()được sử dụng trong lập trình socket để tạo một socket thụ động, cho biết rằng nó sẽ được sử dụng để chấp nhận các kết nối đến. Sau khi socket() được tạo bằng cách sử dụng socket()và liên kết với một địa chỉ cục bộ bằng cách sử dụng bind(), listen()hàm này sẽ được gọi để đặt ổ cắm ở trạng thái nghe. Trả về 0 nếu thành công, -1 nếu thất bại

*#include <sys/types.h>*

*#include <sys/socket.h>*

*int listen(int sockfd, int backlog);*

* sockfd: Bộ mô tả tệp của socket đã được tạo và liên kết bằng cách sử dụng socket()và bind().
* backlog: Độ dài tối đa mà hàng đợi kết nối đang chờ xử lý có thể tăng lên. Nó cho biết số lượng máy khách tối đa có thể chờ để được máy chủ phục vụ. Các kết nối vượt quá giới hạn này có thể bị từ chối.

1. accept(): để chấp nhận kết nối đang chờ xử lý trên socket lisnening. Khi ổ cắm máy chủ ở trạng thái lắng nghe (sau khi gọi listen()), nó có thể sử dụng accept()để tạo ổ cắm mới để xử lý việc trao đổi dữ liệu thực tế với máy khác. Hàm này trả về -1 nếu thất bại và trả về giá trị bộ mô tả tệp của máy khách nếu thành công

*#include <sys/types.h>*

*#include <sys/socket.h>*

*int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);*

* sockfd: Bộ mô tả tập tin của socket.
* addr: Một con trỏ tới cấu trúc ( struct sockaddr) sẽ chứa thông tin địa chỉ của máy khách kết nối.
* addrlen: Một con trỏ tới một socklen\_t biến xác định kích thước của cấu trúc địa chỉ. Điều này ban đầu phải được đặt thành kích thước của cấu trúc được truyền vào addrvà khi trả về, nó sẽ chứa kích thước thực của cấu trúc địa chỉ. Khi accept()được gọi và kết nối được chấp nhận, cấu addrtrúc sẽ chứa đầy thông tin địa chỉ của máy khách được kết nối và kích thước thực của cấu trúc này được ghi lại vào biến addrlen. Điều này quan trọng vì kích thước thực tế của cấu trúc địa chỉ có thể khác với kích thước bạn cung cấp ban đầu.

1. connect() : thiết lập kết nối với máy chủ từ xa. Nó thường được các ứng dụng khách sử dụng để kết nối với máy chủ đang lắng nghe các kết nối đến. Trả về 0 nếu connect OK , trả về -1 nếu không connect được.

*#include <sys/types.h>*

*#include <sys/socket.h>*

*int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);*

* sockfd: Bộ mô tả tập tin của ổ cắm ( là kết quả trả về của hàm socket() trong chương trình máy khách)
* addr: Một con trỏ tới a struct sockaddrchứa thông tin địa chỉ của máy chủ (địa chỉ IP và số cổng).
* addrlen: Kích thước của cấu trúc địa chỉ được trỏ bởi addr.

1. Read() : được sử dụng để đọc dữ liệu từ bộ mô tả tệp và trong trường hợp socket, bộ mô tả ổ cắm. Nếu giá trị trả về lớn hơn 0, nó cho biết số byte được đọc thành công. Nếu giá trị trả về là 0, điều đó có nghĩa là phần cuối của tệp (đối với ổ cắm, điều này có thể cho biết kết nối đã bị đóng ở đầu kia). Nếu giá trị trả về là -1 thì có nghĩa là có lỗi. Bạn có thể kiểm tra errnobiến để biết thêm thông tin về lỗi.

*ssize\_t read(int fd, void \*buf, size\_t count);*

* fd: Đây là bộ mô tả tệp của socket mà bạn muốn đọc dữ liệu từ đó
* buf: Đây là con trỏ tới bộ đệm nơi dữ liệu đọc sẽ được lưu trữ.
* count: Điều này thể hiện số byte bạn muốn đọc từ socket

1. write(): hàm này được sử dụng để ghi dữ liệu vào bộ mô tả tệp. Hàm write()trả về số byte được ghi, có thể nhỏ hơn số byte đã chỉ định count. Nó trả về -1 trong trường hợp có lỗi và bạn có thể kiểm tra errnobiến để biết thêm thông tin về lỗi.

*ssize\_t write(int fd, const void \*buf, size\_t count);*

* fd: Đây là phần mô tả tập tin của socket mà bạn muốn ghi dữ liệu vào đó.
* buf: Đây là con trỏ tới vùng đệm chứa dữ liệu mà bạn muốn ghi.
* count: Điều này thể hiện số byte bạn muốn ghi vào ổ cắm.

1. recvfrom(): Hàm recvfrom trong Linux thường được sử dụng trong lập trình socket, đặc biệt là với các socket UDP (Giao thức gói dữ liệu người dùng). Nó được sử dụng để nhận dữ liệu từ socket và thu thập thông tin về người gửi, chẳng hạn như địa chỉ của người gửi.

*ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags, struct sockaddr \*src\_addr, socklen\_t \*addrlen);*

* sockfd: Bộ mô tả tập tin đại diện cho socket.
* buf: Một con trỏ tới bộ đệm nơi dữ liệu nhận được sẽ được lưu trữ.
* len: Kích thước của bộ đệm tính bằng byte.
* flag: Cờ có thể được sử dụng để sửa đổi hành vi của hàm (thường được đặt thành 0).
* src\_addr: Một con trỏ tới struct sockaddr sẽ chứa thông tin địa chỉ của người gửi.
* addrlen: Một con trỏ tới biến socklen\_t chỉ định kích thước của cấu trúc src\_addr. Ở đầu vào, nó chứa kích thước của bộ đệm được trỏ bởi src\_addr; ở đầu ra, nó được đặt thành kích thước thực tế của địa chỉ người gửi.

1. sendto() : Chắc chắn! Hàm sendto trong Linux được sử dụng trong lập trình socket để gửi dữ liệu qua socket, đặc biệt với các socket UDP (Giao thức gói dữ liệu người dùng). Chức năng này cho phép bạn gửi dữ liệu đến một địa chỉ đích cụ thể.

*ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags, const struct sockaddr \*dest\_addr, socklen\_t addrlen);*

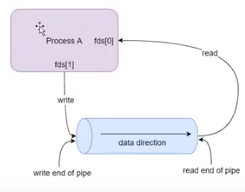
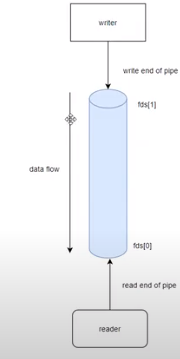
* sockfd: Bộ mô tả tập tin đại diện cho socket.
* buf: Con trỏ tới vùng đệm chứa dữ liệu cần gửi.
* len: Kích thước của dữ liệu tính bằng byte.
* flag: Cờ có thể được sử dụng để sửa đổi hành vi của hàm (thường được đặt thành 0).
* dest\_addr: Một con trỏ tới struct sockaddr chứa thông tin địa chỉ đích.
* addrlen: Kích thước của cấu trúc dest\_addr.

­

PIPE and FIFOs

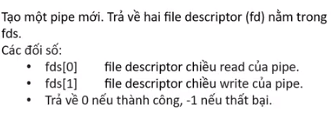
# Khái niệm

* Pipe là 1 trong số các IPC đc sử dụng để truyền thông liên tiến trình
* Pipe chỉ là giao tiếp 1 chiều , tức là ta chỉ sử dụng pipe sao cho 1 quá trình thực hiện ghi vào pipe và qía trình kia đọc từ pipe

* Khi tạo ra pipe, nó sẽ nằm trong RAM và 1 virtual pipe
* Pipe có thể sử dụng trong quá trình tạo process
* Khi v process ghi data vào “virtual file” thì 1 tiến trình liên quan khác có thể đọc data từ đó

1. Các hàm
2. Pipe() : Nếu ta tạo pipe trước hàm fork() thì parent và child process có thể giao tiếp thông qua pipe



Hàm read và write đối với pipe

Truyền vào các tham số như bình thường , các trường hợp đặc bt :

* Nếu ống mở ở đầu write mà hàm read đã đọc hết data trong pipe-> hàm read bị block
* Nếu hàm read đã đọc hết data trong pipe và đầu write bị close thì hàm read sẽ return ra giá trị 0
* Nếu muốn đóng write hoặc read thì phải đóng ở cả cha và con
* *Nếu giao tiếp 2 chiều giưuax 2 process thì đầu reader chỉ dùng đọc => đóng đầu write(fd[1]) còn đầu writer chỉ dùng để ghi => đóng đầu read(fd[0])*

# FiFO

1. Khai niem

* 1 khái niệm mở rộng của pipe

1. Cac ham
2. mkfifo(pathname, quyen)

* pathname : đường dẫn đến file FIFO
* quyen : VD 0666

1. open(name , flag\_quyen) : return fd

name

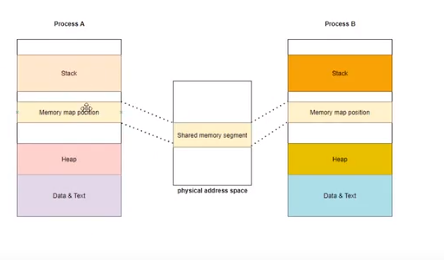
flag\_quyen :VD O\_RDONLY

1. write(fd, str, size \_ str)

Share memory

# Overview

* Share memmory la 1 vung nho dung chung ma nhieu tien trinh cung truy cap den
* Lam tang toc do xu ly cac tien trinh k can truyen nhan data
* 2 phuong phap tao ra share mem

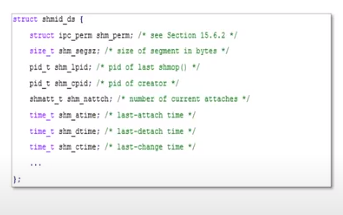


# Systam V Share mem

* Các bước triển khai
* Tạo key
* Cấp phát Share mem segment
* Mapa/ umap share mem segment
* Giải phóng share mem segment
* Chi tiết các buoc:

1. Tạo key

* Share mem đc quản lý thông qua IPC struct . Mỗi IPC struct đc xác định bởi 1 số nguyên k âm Identifier
* Để có thể đọc ghi data vào share mem thì cần bt đc Identifirer
* Để tạo Identifier => hàm ftok()



1. Tạo share mem segment :

* Mỗi segment đc liên kết vs 1 struct về permission cho phéo read/write của process
* Tròng đó, mode là 1 tổ hopwj phím bitwise
* Để tạo mem segment share => API : shmget(key, size, ) ( hàm này raturn shmid): Hàm shmget() là lệnh gọi hệ thống System V IPC (Giao tiếp giữa các quá trình) được sử dụng để phân bổ hoặc truy xuất phân đoạn bộ nhớ dùng chung của System V(share memory segment). Chức năng này cung cấp một cách để các tiến trình thiết lập các vùng bộ nhớ dùng chung để liên lạc giữa các tiến trình. Nếu thành công, hàm trả về mã định danh của phân đoạn bộ nhớ dùng chung(share memory ID) (số nguyên không âm). Khi thất bại, nó trả về -1 và errno được đặt để báo lỗi.

*#include <sys/ipc.h>*

*#include <sys/shm.h>*

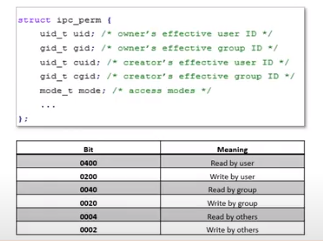
*int shmget(key\_t key, size\_t size, int shmflg);*

* key: Khóa xác định duy nhất phân đoạn bộ nhớ dùng chung. Khóa này có thể được tạo bằng hàm ftok() hoặc được đặt thành IPC\_PRIVATE cho các segment riêng tư.
* size: Kích thước, tính bằng byte, của phân đoạn bộ nhớ dùng chung sẽ được tạo hoặc truy xuất.
* shmflg: Cờ để chỉ định chế độ hoạt động, chẳng hạn như quyền và tùy chọn tạo. Cờ có thể được sử dụng để chỉ định chế độ tạo và quyền. Các cờ phổ biến bao gồm IPC\_CREAT để tạo phân khúc nếu nó không tồn tại và IPC\_EXCL để đảm bảo tạo độc quyền.
* ftok() function : Ham tao identifer key

*#include <sys/types.h>*

*#include <sys/ipc.h>*

*key\_t ftok(const char \*pathname, int proj\_id);*



1. Attach/detach vùng share mem

* API shmat() / shmdt()
* Shmat() : Khi một tiến trình gắn vào share mem segment c bằng shmat(), nó sẽ có quyền truy cập vào vùng bộ nhớ dùng chung đó. Nếu thành công, shmat() trả về địa chỉ bắt đầu của phân đoạn bộ nhớ dùng chung được đính kèm. Khi thất bại, nó trả về (void \*)-1 và errno được đặt để chỉ ra lỗi

*#include <sys/types.h>*

*#include <sys/types.h>*

*#include <sys/shm.h>*

*void \*shmat(int shmid, const void \*shmaddr, int shmflg);*

* shmid: Mã định danh của phân đoạn bộ nhớ dùng chung (share memory ID)thu được từ shmget().
* shmaddr: Địa chỉ mong muốn mà phân đoạn bộ nhớ dùng chung sẽ được đính kèm. Nếu shmaddr là NULL, hệ thống sẽ chọn địa chỉ phù hợp.
* shmflg: Cờ để kiểm soát hành vi đính kèm. Cờ thường được sử dụng là SHM\_RDONLY để đính kèm phân đoạn để truy cập chỉ đọc.
* shmdt() : Hàm shmdt() trong IPC (Giao tiếp giữa các quá trình) với bộ nhớ dùng chung được sử dụng để tách phân đoạn bộ nhớ dùng chung khỏi không gian địa chỉ của quá trình gọi

*#include <sys/types.h>*

*#include <sys/shm.h>*

*int shmdt(const void \*shmaddr);*

parameter là giá trị bắt đầu của địa chỉ share memory segment( giá trị trả về hàm shmat)

* Mở trong Go

1. Giải phóng share mem

* API shmctl() đi với flag IPC\_RMID

#include <sys/ipc.h>

#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

# POSIX share memory

1. **Tao share mem object**

* Int shm\_fd = Shm\_open(const char \*name, int oflag, mode\_t mode): On success, shm\_open() returns a file descriptor that can be used for subsequent operations on the shared memory object.
* *#include <sys/mman.h>*

*#include <fcntl.h>*

*int shmid = shm\_open(const char \*name, int oflag, mode\_t mode);*

* name: Một chuỗi biểu thị tên của đối tượng share memory. Tên này phải bắt đầu bằng dấu gạch chéo lên ("/") và là tên duy nhất trong hệ thống. Nếu bạn đang tạo một đối tượng bộ nhớ dùng chung mới thì đây là tên mà nó sẽ được xác định.
* oflag: Sự kết hợp của các cờ kiểm soát hành vi của hàm. Các cờ phổ biến bao gồm:

+ O\_CREAT: Tạo đối tượng bộ nhớ dùng chung nếu nó không tồn tại.

+ O\_RDWR: Mở đối tượng bộ nhớ dùng chung cho cả đọc và ghi.

+ O\_EXCL: Khi được sử dụng với O\_CREAT, chức năng sẽ thất bại nếu đối tượng bộ nhớ dùng chung đã tồn tại.

* mode: Các quyền được đặt khi tạo đối tượng bộ nhớ dùng chung mới. Tham số này chỉ liên quan khi sử dụng O\_CREAT. Nó được chỉ định dưới dạng giá trị bát phân, tương tự như quyền đối với tệp.

1. **Set size**

* Khi share mem dc tao ra thi size = 0 => Need to set size for it
* Ftruncate(shm\_fd, size): Nếu thành công, hàm này trả về 0, nếu k thành công thì trả về -1
* *#include <unistd.h>*

*int ftruncate(int fd, off\_t length);*

* fd (bộ mô tả tệp): Bộ mô tả tệp của tệp đang mở hoặc đối tượng bộ nhớ dùng chung. Bộ mô tả tệp này phải được lấy trước đó thông qua lệnh gọi open hoặc shm\_open thành công.
* length: Kích thước mong muốn để đặt cho tệp hoặc đối tượng bộ nhớ dùng chung. Nếu tệp hoặc đối tượng bộ nhớ dùng chung được mở rộng, phần mở rộng sẽ được lấp đầy bằng 0 byte. Nếu nó được rút ngắn, dữ liệu dư thừa sẽ bị loại bỏ.

1. **Map/unmap share memory object**

* Mmap()/umap()
* *#include <sys/mman.h>*

*void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);*

* addr: Địa chỉ bắt đầu ưa thích để ánh xạ. Nếu NULL, kernel sẽ chọn một địa chỉ phù hợp.
* length: Độ dài (tính bằng byte) của ánh xạ.
* prot: Mức bảo vệ bộ nhớ mong muốn của ánh xạ. Tham số này là sự kết hợp của các cờ sau:

+ PROT\_READ: Các trang có thể được đọc.

+ PROT\_WRITE: Các trang có thể được viết.

+ PROT\_EXEC: Các trang có thể được thực thi.

* flags: Cờ bổ sung kiểm soát ánh xạ:

+ MAP\_SHARED: Chia sẻ ánh xạ với các quy trình khác.

+ MAP\_PRIVATE: Tạo ánh xạ sao chép khi ghi riêng tư. (copy and write)

* fd: Bộ mô tả tệp của tệp đang mở hoặc đối tượng bộ nhớ dùng chung. Đối với bộ nhớ dùng chung, đây thường là bộ mô tả tệp thu được từ shm\_open().
* offset: Phần bù (tính bằng byte) trong tệp hoặc đối tượng bộ nhớ dùng chung mà ánh xạ sẽ bắt đầu từ đó.
* Nếu thành công, mmap() trả về địa chỉ bắt đầu của vùng được ánh xạ., nếu k thành công thì hàm return -1
* *#include <sys/mman.h>*

*int munmap(void \*addr, size\_t length);*

* addr: Địa chỉ bắt đầu của vùng được ánh xạ mà bạn muốn hủy bản đồ. Đây phải là địa chỉ đã được trả về bởi lệnh gọi mmap() trước đó.
* length: Độ dài (tính bằng byte) của vùng được ánh xạ mà bạn muốn hủy ánh xạ. Độ dài này phải khớp với độ dài được sử dụng trong lệnh gọi mmap() tương ứng.
* Nếu thành công, hàm return 0, k thành công thì return -1
* 4 kieu memory mapping

+ private anonymous mapping

+shared anonymous mapping

+private file mapping

***+shared file mapping***

1. **Giải phóng share memory**

Để giải phóng shared memory object được tạo ra trước đó chúng ta sử dụng API *shm\_unlink(char \*name).*

Semaphore

1. Semaphore la gi

* Cơ chế cho phép đồng bộ truy cập giữa các process và thread
* Block process và thread truy cập vào 1 vùng nhớ đg đc used bởi process/thread khác

1. POSIX semaphore

* Name semaphore
* Unname semaphore

1. Hoạt đoong

* Semaphoe như là 1 số nguyên dương đx duy trỳ bởi kernel

1. Các hàm

* Sem\_wait(): return ra giá trị semaphore sẽ đc giảm đi 1
* Giảm giá trị của semaphore
* Nếu semaphore hiện giá trị lớn hơn 0 => sem\_wait() return ngay
* …………………………… hiện là 0 -> hàm sem\_wait() bị block cho đến khi semaphore đc tăng trên 0

*#include <semaphore.h>*

*int sem\_wait(sem\_t \*sem);*

* Sem\_trywait() là 1 phiên bản non\_blocking của sem\_wait()
* Sem\_timedwait() chỉ thực hiện chờ trong 1 thời gian nhất định, nếu sau timeout gt semaphore vẫn = 0 thì lỗi sẽ trả về ETIMEDOUT
* Sem\_post(0 : tăng gt semaphore lên 1
* Nếu giá trị của semaphore đang = 0 thì thực hiện tăng gt semaphore lê 1, khi đó các process/thread đg chờ sem\_wait() sẽ đc đánh thức

*#include <semaphore.h>*

*int sem\_post(sem\_t \*sem);*

* Sem\_getvalue(): Trả về giá trị hiện tại của semaphore

*#include <semaphore.h>*

*int sem\_getvalue(sem\_t \*sem, int \*sval);*

1. Name semaphore

* Sem\_open() : mở 1 semaphore đg tồn tại / tạo mới semaphore

*#include <semaphore.h>*

*sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, unsigned int value);*

* name: Tên của semaphore. Tên này phải bắt đầu bằng dấu gạch chéo lên ("/") và là tên đường dẫn hợp lệ.
* oflag: Cờ điều khiển hoạt động của sem\_open(). Các cờ phổ biến bao gồm O\_CREAT (tạo semaphore nếu nó không tồn tại) và O\_EXCL (đảm bảo rằng O\_CREAT được sử dụng và semaphore được tạo nguyên tử, nếu cờ này đc dùng thì nếu sema đã tồn tại thì hàm tạo thất bại).
* mode : Các quyền dành cho semaphore khi nó được tạo (chỉ áp dụng nếu sử dụng O\_CREAT).
* value: Giá trị ban đầu của semaphore.
* Sem\_close() : Khi sem\_open() đc gọi ra thì semaphore sẽ đc liên kết vs process, hệ thống sẽ ghi lại liên kết này
* Sem\_close() sẽ hủy bỏ mối liên kết trên và giải phóng bất kỳ tài nguyên mà semaphore và process liên kết vs nhau-> Giảm số lượng process tham chiếu đến sema

*#include <semaphore.h>*

*int sem\_close(sem\_t \*sem);*

* Sem\_unlink() : Xóa hẳn sema ra khỏi hệ thống

*#include <semaphore.h>*

*int sem\_unlink(const char \*name);*

1. Unnmed semaphore

* Sem\_init(): Khởi tạo sema và thông báo cho kernel

*#include <semaphore.h>*

*int sem\_init(sem\_t \*sem, int pshared, unsigned int value);*

* pshared: Tham số này chỉ định liệu semaphore có nên được chia sẻ giữa các tiến trình hay luồng hay không. Nếu pshared là 0, semaphore được chia sẻ giữa các luồng của cùng một tiến trình. Nếu pshared khác 0 thì semaphore có thể được chia sẻ giữa các tiến trình khác nhau. Trong các ứng dụng hiện đại, người ta thường sử dụng 0 để chia sẻ luồng.
* value : gia tri khởi đầu của semaphore
* Sem\_destroy() : Hủy sema

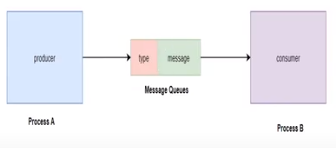
*#include <semaphore.h>*

*int sem\_destroy(sem\_t \*sem);”/*

Queue

# Overview

* 1 message laf 1 danh sachs message đc duy trỳ bởi kernel
* Tất cả process có thể trao đổi data thông qua việc truy cập vào cùng 1 queue
* 1 queue có thể đc đình kèm thông tin về type



# System V message Queue

1. Tạo key

* Ftok(): ạo khóa dựa trên tên tệp và mã định danh dự án. Khóa này thường được sử dụng làm đối số khi tạo hoặc truy cập tài nguyên IPC như hàng đợi tin nhắn, phân đoạn bộ nhớ dùng chung hoặc ngữ nghĩa.

*key\_t ftok(const char \*pathname, int proj\_id);*

VD : key\_t key = ftok("/path/to/file", 'A'); // 'A' is the project identifier

1. Tạo ms / mở queue có sẵn : tạo hàng đợi tin nhắn mới hoặc để truy cập hàng đợi tin nhắn hiện có. Chức năng này thường được sử dụng cùng với các chức năng xếp hàng tin nhắn khác của Hệ thống V để cho phép liên lạc giữa các quá trình (IPC) giữa các quá trình khác nhau. Hàm msgget() trả về mã định danh hàng đợi tin nhắn (một số nguyên không âm) nếu thành công hoặc -1 nếu thất bại. Mã định danh được sử dụng trong các hoạt động xếp hàng tin nhắn tiếp theo.

* *#include <sys/types.h>*

*#include <sys/ipc.h>*

*#include <sys/msg.h>*

*int msgget(key\_t key, int msgflg);*

* key: Tham số khóa thường được tạo bằng hàm ftok(), như đã thảo luận trong phản hồi trước đó. Nó đại diện cho một khóa duy nhất được liên kết với hàng đợi tin nhắn.
* msgflg: Tham số msgflg chỉ định các cờ để tạo và truy cập hàng đợi tin nhắn. Nó có thể là sự kết hợp của các cờ ORed với nhau, chẳng hạn như:

+ IPC\_CREAT: Tạo hàng đợi tin nhắn nếu nó chưa tồn tại.

+ IPC\_EXCL: Trả về lỗi nếu hàng đợi tin nhắn đã tồn tại (được sử dụng với IPC\_CREAT). (ví dụ: IPC\_PRIVATE, IPC\_CREAT | 0666).

1. Write into queue

Msgsnd( ): msgsnd() là chức năng System V IPC trong Linux được sử dụng để gửi tin nhắn đến hàng đợi tin nhắn. Hàm msgsnd() trả về 0 nếu thành công và -1 nếu thất bại. Nếu IPC\_NOWAIT được chỉ định trong msgflg và hàng đợi tin nhắn đã đầy, hàm sẽ trả về ngay lập tức với lỗi

* *#include <sys/types.h>*

*#include <sys/ipc.h>*

*#include <sys/msg.h>*

*int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);*

* msqid: Mã định danh của hàng đợi tin nhắn, thu được từ lệnh gọi trước tới msgget().
* msgp: Một con trỏ tới cấu trúc chứa dữ liệu tin nhắn. Cấu trúc thường có dạng sau:

*struct msgbuf {*

*long mtype; // Message type*

*char mtext[1]; // Message data (can be larger than 1)*

*};*

* Trường mtype đại diện cho loại tin nhắn và mtext là một mảng để chứa dữ liệu tin nhắn thực tế. Kích thước của dữ liệu tin nhắn (mtext) có thể lớn hơn 1 và được chỉ định bởi tham số msgsz.
* msgsz: Kích thước của dữ liệu tin nhắn được trỏ tới bởi msgp.
* msgflg: Cờ kiểm soát hành vi của hàm msgsnd(). Các cờ phổ biến bao gồm:

+ IPC\_NOWAIT: Quay lại ngay nếu hàng đợi tin nhắn đầy.

+ 0: Hành vi mặc định (đợi nếu hàng đợi tin nhắn đầy).

1. Read from queue

Msgrcv(): nhận tin nhắn từ hàng đợi tin nhắn

* *#include <sys/types.h>*

*#include <sys/ipc.h>*

*#include <sys/msg.h>*

*ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg);*

* msqid: Mã định danh của hàng đợi tin nhắn, thu được từ lệnh gọi trước tới msgget().
* msgp: Một con trỏ tới cấu trúc nơi lưu trữ dữ liệu tin nhắn đã nhận. Cấu trúc thường có dạng sau:

*struct msgbuf {*

*long mtype; // Message type*

*char mtext[1]; // Message data (can be larger than 1)*

*};*

* msgsz: Kích thước của bộ đệm được trỏ tới bởi msgp.
* msgtyp: Loại tin nhắn sẽ nhận được. Nếu msgtyp là 0, tin nhắn đầu tiên trong hàng đợi sẽ được truy xuất bất kể loại của nó. Ngược lại, msgrcv() trả về tin nhắn đầu tiên trong hàng đợi có kiểu thông điệp được chỉ định.
* msgflg: Cờ kiểm soát hành vi của hàm msgrcv(). Các cờ phổ biến bao gồm:

IPC\_NOWAIT: Trả về ngay lập tức nếu không có tin nhắn thuộc loại được chỉ định trong hàng đợi.

MSG\_NOERROR: Cắt bớt tin nhắn nếu nó lớn hơn msgsz.

0: Hành vi mặc định (chặn nếu không có tin nhắn thuộc loại được chỉ định trong hàng đợi).

1. Delete queue

Msgctl() : Có, msgctl() là hàm System V IPC trong Linux được sử dụng để kiểm soát các hoạt động trên hàng đợi tin nhắn. Nó cho phép bạn thực hiện nhiều thao tác điều khiển khác nhau như truy xuất thông tin về hàng đợi tin nhắn, xóa hàng đợi tin nhắn hoặc sửa đổi các quyền liên quan đến hàng đợi tin nhắn.

* *#include <sys/types.h>*

*#include <sys/ipc.h>*

*#include <sys/msg.h>*

*int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);*

* msqid: message id thu được từ msgget().
* cmd:

IPC\_RMID: Xóa message queue ngay lập tức.

IPC\_STAT

IPC\_SET

* Để xóa một message queue thông thường cmd chúng ta dùng là IPC\_RMID và buf để thành giá trị NULL.

# POSIX message queue

1. Opening

Mq\_open( ) : Có, mq\_open() là một hàm được sử dụng trong hàng đợi tin nhắn POSIX trên Linux để tạo hoặc mở hàng đợi tin nhắn. Hàm trả về bộ mô tả hàng đợi tin nhắn (mqd\_t) nếu thành công hoặc -1 nếu thất bại.

* *#include <mqueue.h>*

*mqd\_t mq\_open(const char \*name, int oflag, mode\_t mode, struct mq\_attr \*attr);*

* name: Tên của hàng đợi tin nhắn. Tên này phải bắt đầu bằng dấu gạch chéo lên ("/") và là tên duy nhất trong hệ thống.
* oflag: Cờ biểu thị hành vi của hàng đợi tin nhắn. Những lá cờ này bao gồm:

+ O\_RDONLY: Mở hàng đợi tin nhắn chỉ để nhận tin nhắn.

+ O\_WRONLY: Mở hàng đợi tin nhắn chỉ để gửi tin nhắn.

+ O\_RDWR: Mở hàng đợi tin nhắn cho cả gửi và nhận.

+ O\_CREAT: Tạo hàng đợi tin nhắn nếu nó chưa tồn tại.

+ O\_EXCL: Trả về lỗi nếu O\_CREAT cũng được chỉ định và hàng đợi tin nhắn đã tồn tại.

* mode: Các quyền sẽ được đặt nếu O\_CREAT được chỉ định. Nó là sự kết hợp của các bit cấp phép (ví dụ: S\_IRUSR | S\_IWUSR).
* Struct mq\_attr

*struct mq\_attr {*

*long mq\_flags; // Flags (ignored for mq\_open)*

*long mq\_maxmsg; // Maximum number of messages in the queue*

*long mq\_msgsize; // Maximum size of a message (in bytes)*

*long mq\_curmsgs; // Current number of messages in the queue*

*long \_\_pad[4]; // Padding to ensure sizeof(struct mq\_attr) is a multiple of long*

*};*

* mq\_flags: Trường này hiện không được sử dụng và bị bỏ qua trong mq\_open(). Nó được dành riêng để sử dụng trong tương lai.
* mq\_maxmsg: Trường này chỉ định số lượng tin nhắn tối đa mà hàng đợi có thể chứa.

Khi tạo hoặc mở hàng đợi tin nhắn bằng mq\_open(), bạn có thể đặt trường này trong struct mq\_attr để chỉ định số lượng tin nhắn tối đa trong hàng đợi.

* mq\_msgsize: Trường này chỉ định kích thước tối đa của tin nhắn trong hàng đợi, tính bằng byte. Khi tạo hoặc mở hàng đợi tin nhắn bằng mq\_open(), bạn có thể đặt trường này trong struct mq\_attr để chỉ định kích thước tối đa của tin nhắn.
* mq\_curmsgs: Trường này thể hiện số lượng tin nhắn hiện tại trong hàng đợi.
* Khi gọi mq\_getattr(), trường này được cập nhật để phản ánh số lượng tin nhắn hiện tại trong hàng đợi.

\_\_pad[4]:

Trường này là phần đệm để đảm bảo rằng kích thước của struct mq\_attr là bội số của độ dài. Nó được đưa vào nhằm mục đích căn chỉnh và không mang theo bất kỳ dữ liệu cụ thể nào.

1. Sending message

Mq\_send(): được sử dụng để gửi tin nhắn đến hàng đợi tin nhắn.

* *#include <mqueue.h>*

*int mq\_send(mqd\_t mqdes, const char \*msg\_ptr, size\_t msg\_len, unsigned int msg\_prio);*

* mqdes: Bộ mô tả hàng đợi tin nhắn thu được từ lệnh gọi trước tới mq\_open().
* msg\_ptr: Một con trỏ tới dữ liệu tin nhắn sẽ được gửi.
* msg\_len: Độ dài của dữ liệu tin nhắn được trỏ tới bởi msg\_ptr.
* msg\_prio: Mức độ ưu tiên của tin nhắn. Các tin nhắn trong hàng đợi được sắp xếp dựa trên mức độ ưu tiên của chúng. Giá trị ưu tiên cao hơn có nghĩa là thông báo có mức độ ưu tiên cao hơn.

1. Receive message

Mq\_receive(): nhận được tin nhắn từ hàng đợi tin nhắn.

* *#include <mqueue.h>*

*ssize\_t mq\_receive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned int \*msg\_prio);*

* mqdes: Bộ mô tả hàng đợi tin nhắn thu được từ lệnh gọi trước tới mq\_open().
* msg\_ptr: Một con trỏ tới bộ đệm nơi dữ liệu tin nhắn nhận được sẽ được lưu trữ.
* msg\_len: Kích thước của bộ đệm được trỏ tới bởi msg\_ptr. Nó chỉ định số byte tối đa được nhận.
* msg\_prio: Một con trỏ tới một số nguyên không dấu nơi lưu trữ mức độ ưu tiên của tin nhắn nhận được. Nếu msg\_prio là NULL thì thông tin ưu tiên sẽ không được trả về.

1. Closing a queue

Mq\_close(): được sử dụng để đóng bộ mô tả hàng đợi tin nhắn. Khi bạn mở hàng đợi tin nhắn bằng mq\_open(), hệ điều hành sẽ trả về bộ mô tả hàng đợi tin nhắn (mqd\_t) và mq\_close() được sử dụng để đóng bộ mô tả này.

* *#include <mqueue.h>*

*int mq\_close(mqd\_t mqdes);*

1. Huy queue

Mq\_unlink() : Hàm mq\_unlink() trong Linux được sử dụng để xóa (hủy liên kết) hàng đợi tin nhắn khỏi hệ thống. Khi bạn tạo hàng đợi tin nhắn POSIX bằng mq\_open(), hàng đợi tin nhắn sẽ được tạo với tên được chỉ định. Hàm mq\_unlink() được sử dụng để xóa hàng đợi tin nhắn được đặt tên này khỏi hệ thống, khiến nó không thể truy cập được bằng tên của nó nữa.

Kernel Module Basic

# Overview

# Các hàm

1. **Tạo device number**

* Kiểu dev\_t:
* dev\_t là một kiểu dữ liệu trong hệ điều hành Linux (cũng như trong các hệ điều hành UNIX khác) được sử dụng để đại diện cho một số thiết bị. Trong Linux, mỗi thiết bị như ổ đĩa, socket, hoặc pipe đều được định danh bằng một số không dấu kiểu dev\_t.
* Hàm cấp phát động

*int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, const char \*name);*

* Con trỏ tới device number sẽ được gán cho phạm vi được phân bổ. Nó là một tham số đầu ra.
* firstminor: Số thứ bắt đầu của dải ô. ( thường là 0)
* count: Số lượng thiết bị liền kề cần phân bổ.(thường là 1)
* tên: Một chuỗi xác định thiết bị trong /proc/devices. Nó là tùy chọn và có thể được đặt thành NULL.

*VD : alloc\_chrdev\_region(&mdev.dev\_num, 0, 1, "m-cdev")*

* Hàm trả về 0 nếu thành công

Để lấy ra giá trị major hoặc minor của 1 device number, ta dùng macro MAJOR và MINOR

VD :

*pr\_info("Allocated character device region, major=%d, minor=%d\n", MAJOR(first\_dev), MINOR(first\_dev));*

* Hàm hủy bộ nhớ cấp phát động

*void unregister\_chrdev\_region(dev\_t from, unsigned int count);*

* from: Device bắt đầu của dãy cần hủy đăng ký.
* count: Số lượng thiết bị cần hủy đăng ký.

VD : *unregister\_chrdev\_region(mdev.dev\_num, 1);*

1. **Tạo device file**

b.1 Tạo Class device

* Struct class :

struct class được sử dụng để đại diện cho một nhóm các thiết bị có liên quan logic hoặc chức năng chung. Cụ thể, struct class thường được sử dụng để tạo ra một lớp thiết bị (device class), và các thiết bị trong cùng một lớp thường chia sẻ một số thuộc tính và hành vi chung.

* Class\_create()

*struct class \*class\_create(struct module \*owner, const char \*name);*

* owner: Một con trỏ tới mô-đun "sở hữu" lớp này. Nó thường được đặt thành THIS\_MODULE, đại diện cho mô-đun chứa mã gọi hàm class\_create().
* name: Một chuỗi cung cấp tên cho lớp. Tên này được sử dụng để xác định lớp trong không gian người dùng. Bạn nên chọn một tên duy nhất và mang tính mô tả.
* Hàm trả về một con trỏ tới class cấu trúc nếu thành công hoặc NULL nếu thất bại.
* Khi một lớp được tạo, nó sẽ hiển thị trong thư mục /sys/class trong hệ thống tệp sysfs, cho phép không gian người dùng tương tác với các thiết bị được sắp xếp theo class

VD : mdev.m\_class = class\_create(THIS\_MODULE, "m\_class");

*if (mdev.m\_class == NULL) {*

*pr\_err("Cannot create the struct class for my device\n");*

*goto rm\_device\_numb;*

*}*

* class\_destroy();

*void class\_destroy(struct class \*cls);*

Hàm class\_destroy() chịu trách nhiệm giải phóng các tài nguyên được liên kết với lớp được chỉ định. Sau khi gọi hàm này, lớp và các thiết bị liên quan không còn hiển thị trong thư mục /sys/class trong hệ thống tệp sysfs.

Vd : *class\_destroy(mdev.m\_class);*

* Khi viết hàm class\_create() trong init() , để tránh trường hợp xấu xảy ra, ta thường sử dụng từ khóa “goto” nếu class create trả về NULL. “goto” đến hàm hủy device\_number : unregister\_chrdev\_region() được đặt cuối hàm init()

b.2 Tạo device file

* device\_create()

*struct device \*device\_create(struct class \*class, struct device \*parent,*

*dev\_t devt, void \*drvdata, const char \*fmt, ...);*

* class: Một con trỏ tới lớp cấu trúc đại diện cho lớp thiết bị mà thiết bị mới thuộc về.
* parent: Một con trỏ tới thiết bị cha mẹ. Nó thường được đặt thành NULL trừ khi thiết bị được tạo là con của thiết bị khác.
* devt: Số thiết bị (major vs minor) được liên kết với thiết bị. Nó thường là số thiết bị thu được thông qua alloc\_chrdev\_zone().
* drvdata: Con trỏ tới dữ liệu dành riêng cho trình điều khiển có thể được liên kết với thiết bị, thuong dc dat la NULL
* fmt: Chuỗi định dạng kiểu printf được sử dụng để tạo tên thiết bị. Nó là tùy chọn và có thể được đặt thành NULL.
* void device\_destroy();  
  *void device\_destroy(struct class \*class, dev\_t devt);*

VD : *device\_destroy(mdev.m\_class, mdev.dev\_num);*

* Khi viết hàm device\_create() trong init() , để tránh trường hợp xấu xảy ra, ta thường sử dụng từ khóa “goto” nếu device create trả về NULL. “goto” đến hàm hủy class: class\_destroy() được đặt cuối hàm init()
* Note :
* Các hàm tạo class và device file thường đc đặt trong constructor , tuy nhiên cần phải có cả hàm hủy class và device number trong đó để đề phòng sai lầm ( sử dụng từ khóa goto)
* Trong hàm destructor , thứ tự hủy là device, class và device number

1. **Đăng ký file operation**

* Struct cdev : Trong Linux, struct cdev là cấu trúc dữ liệu đại diện cho một character device. Nó là một phần của giao diện character device của kernel, cung cấp cách thức cho các chương trình trong không gian người dùng giao tiếp với trình điều khiển thiết bị. Thông thường, một struct cdev được nhúng trong một cấu trúc lớn hơn đại diện cho toàn bộ thiết bị.

Vd :

*#include <linux/cdev.h>*

*struct my\_device {*

*struct cdev cdev;*

*// Other device-specific fields*

*};*

*struct file\_operations my\_fops = {*

*// Set up file operations (open, read, write, etc.)*

*};*

*struct my\_device my\_dev;*

*// Initialization*

*cdev\_init(&my\_dev.cdev, &my\_fops);*

*my\_dev.cdev.owner = THIS\_MODULE;*

*// Registration*

*cdev\_add(&my\_dev.cdev, dev\_num, 1); // dev\_num represents the major and minor numbers*

* Cdev\_nit() : được sử dụng để khởi tạo một biến kiểu struct cdev.

*void cdev\_init(struct cdev \*cdev, const struct file\_operations \*fops);*

* cdev: Con trỏ tới biến kiểu struct cdev mà bạn muốn khởi tạo.
* fops: Con trỏ tới biến kiểu struct file\_operations chứa các con trỏ đến các hàm thực hiện các thao tác trên thiết bị, như mở (open), đọc (read), ghi (write), đóng (release), và các hàm khác.

Vd : *struct my\_device {*

*struct cdev cdev;*

*// Other device-specific fields*

*};*

*struct file\_operations my\_fops = {*

*// Set up file operations (open, read, write, etc.)*

*};*

*struct my\_device my\_dev;*

*// Initialization*

*cdev\_init(&my\_dev.cdev, &my\_fops);*

* cdev\_add() : được sử dụng để đăng ký một struct cdev với hệ thống và liên kết nó với một số chính và số lượng chính/minor đã cấp phát.

*int cdev\_add(struct cdev \*p, dev\_t dev, unsigned count);*

* p: Con trỏ tới biến kiểu struct cdev bạn muốn đăng ký.
* dev: Số chính và số lượng chính/minor đã cấp phát cho thiết bị.
* count: Số lượng thiết bị con (minor) bạn muốn đăng ký. Thông thường, bạn sẽ đăng ký một thiết bị, nên giá trị này thường là 1.
* Return -1 if fail

VD : *#include <linux/cdev.h>*

*struct my\_device {*

*struct cdev cdev;*

*// Other device-specific fields*

*};*

*struct file\_operations my\_fops = {*

*// Set up file operations (open, read, write, etc.)*

*};*

*struct my\_device my\_dev;*

*// Initialization*

*cdev\_init(&my\_dev.cdev, &my\_fops);*

*my\_dev.cdev.owner = THIS\_MODULE;*

*// Registration*

*cdev\_add(&my\_dev.cdev, dev\_num, 1); // dev\_num represents the major and minor numbers*

* Khi viết hàm cdev\_add() trong init() , để tránh trường hợp xấu xảy ra, ta thường sử dụng từ khóa “goto” nếu cdev\_add trả về -1. “goto” đến hàm hủy device number : unregister\_chrdev\_region () được đặt cuối hàm init()
* cdev\_del() : dev\_del được sử dụng để loại bỏ một struct cdev khỏi hệ thống.

*void cdev\_del(struct cdev \*p);*

* p: Con trỏ tới biến kiểu struct cdev mà bạn muốn loại bỏ

VD *: #include <linux/cdev.h>*

*// Assuming 'my\_cdev' is a struct cdev representing your device*

*// Module cleanup function*

*static void \_\_exit my\_module\_exit(void) {*

*// Remove the cdev from the system*

*cdev\_del(&my\_cdev);*

*pr\_info("Character device unregistered\n");*

*}*

*module\_exit(my\_module\_exit);*

# Cấu trúc tổng quan của 1 chương trình Character Device Driver

1. **Khai báo thư viện**

*#include <linux/module.h> /\* Thu vien nay dinh nghia cac macro nhu module\_init/module\_exit \*/*

*#include <linux/fs.h> /\* Thu vien nay dinh nghia cac ham allocate major/minor number \*/*

*#include <linux/device.h> /\* Thu vien nay dinh nghia cac ham class\_create/device\_create \*/*

*#include <linux/cdev.h> /\* Thu vien nay dinh nghia cac ham cdev\_init/cdev\_add \*/*

1. **Định nghĩa các modul information**

*#define DRIVER\_AUTHOR "TuanTNT xxxxxxxx@gmail.com"*

*#define DRIVER\_DESC "Hello world kernel module"*

1. **Khai báo và định nghĩa struct device**

*typedef struct {*

*dev\_t dev\_num;*

*struct class \*m\_class;*

*struct cdev m\_cdev;*

*} m\_foo\_dev;*

*M\_foo\_dev mdev;*

1. **Khai báo các nguyên mẫu hàm**

static int m\_open(struct inode \*inode, struct file \*file);

static int m\_release(struct inode \*inode, struct file \*file);

static ssize\_t m\_read(struct file \*filp, char \_\_user \*user\_buf, size\_t size,loff\_t \* offset);

static ssize\_t m\_write(struct file \*filp, const char \*user\_buf, size\_t size, loff\_t \* offset);

1. **Khai báo struct hàm ( file operation)**

*static struct file\_operations fops =*

*{*

*.owner = THIS\_MODULE,*

*.read = m\_read,*

*.write = m\_write,*

*.open = m\_open,*

*.release = m\_release,*

*};*

1. **Định nghĩa các hàm đã khai báo ở bước c ( thuộc file operation)**

*static int m\_open(struct inode \*inode, struct file \*file)*

*{*

*pr\_info("System call open() called...!!!\n");*

*return 0;*

*}*

*/\* This function will be called when we close the Device file \*/*

*static int m\_release(struct inode \*inode, struct file \*file)*

*{*

*pr\_info("System call close() called...!!!\n");*

*return 0;*

*}*

*/\* This function will be called when we read the Device file \*/*

*static ssize\_t m\_read(struct file \*filp, char \_\_user \*user\_buf, size\_t size, loff\_t \*offset)*

*{*

*pr\_info("System call read() called...!!!\n");*

*return 0;*

*}*

*/\* This function will be called when we write the Device file \*/*

*static ssize\_t m\_write(struct file \*filp, const char \_\_user \*user\_buf, size\_t size, loff\_t \*offset)*

*{*

*pr\_info("System call write() called...!!!\n");*

*return size;*

*}*

1. Khai báo và định nghĩa hàm init()

* Trình tự trong hàm init()
* Khởi tạo device number ( hàm *alloc\_chrdev\_region(*))
* Khởi tạo device file ( class\_create() và device\_create())

+ Trong bước check lỗi khi khởi tạo *class\_create()* cần “goto” đến *unregister\_chrdev\_region()*

+ Trong bước check lỗi của *device\_create()* cần goto đến *class\_destroy()*

* Đăng ký file operation ( *cdev\_ini()* và *cdev\_add())*

+ Trong bước check lỗi khi khởi tạo *cdev\_add()* cần “goto” đến *unregister\_chrdev\_region()ư*

*static ssize\_t m\_write(struct file \*filp, const char \_\_user \*user\_buf, size\_t size, loff\_t \*offset)*

*{*

*pr\_info("System call write() called...!!!\n");*

*return size;*

*}*

*m\_foo\_dev mdev;*

*static int \_\_init Khoitao(void)*

*{*

*if (alloc\_chrdev\_region(&mdev.dev\_num, 0, 1, "my-cdev"))*

*{*

*pr\_err("ERROR: Can not make number device\n");*

*return -1;*

*}*

*pr\_info("Hello from main\n");*

*pr\_info("Major : %d Minor : %d\n", MAJOR(mdev.dev\_num), MINOR(mdev.dev\_num));*

*if ((mdev.m\_class = class\_create(THIS\_MODULE, "my\_class")) == NULL){*

*pr\_err("ERROR: Can not create class\n");*

*goto rm\_dev\_num;*

*}*

*if (device\_create(mdev.m\_class, NULL, mdev.dev\_num, NULL, "my\_device") == NULL)*

*{*

*pr\_err("ERROR: Can not create device\n");*

*goto rm\_class;*

*}*

*cdev\_init(&mdev.m\_cdev, &fops);*

*if (cdev\_add(&mdev.m\_cdev, mdev.dev\_num, 1) < 0)*

*{*

*pr\_err("ERROR: Can not add device\n");*

*goto rm\_dev\_num;*

*}*

*return 0;*

*rm\_dev\_num:*

*unregister\_chrdev\_region(mdev.dev\_num,1);*

*rm\_class:*

*class\_destroy(mdev.m\_class);*

*return -1;*

*}*

1. **Khai báo và định nghĩa hàm exit()**

* Thứ tự

*static void \_\_exit Huy(void)*

*{*

*device\_destroy(mdev.m\_class, mdev.dev\_num);*

*class\_destroy(mdev.m\_class);*

*cdev\_del(&mdev.m\_cdev);*

*unregister\_chrdev\_region(mdev.dev\_num,1);*

*pr\_info("GOODBYE from main\n");*

*}*

1. **Kết luận**

*module\_init(Khoitao);*

*module\_exit(Huy);*

*MODULE\_LICENSE("GPL");*

*MODULE\_AUTHOR(DRIVER\_AUTHOR);*

*MODULE\_DESCRIPTION(DRIVER\_DESC);*

# Một vài hàm khác trên kernel

1. Hàm cấp phát bộ nhớ động kmalloc()

* để cấp phát bộ nhớ động từ kernel's heap

*#include <linux/slab.h>*

*void\* ptr = kmalloc(size, GFP\_KERNEL);*

* size: Kích thước của bộ nhớ cần cấp phát.
* GFP\_KERNEL: Cờ (flags) xác định cách kernel cần cấp phát bộ nhớ. Trong trường hợp này, GFP\_KERNEL thường được sử dụng để yêu cầu cấp phát bộ nhớ trong không gian kernel và có thể kiểm soát được.
* Phải sử dụng kfree() để giải phóng bộ nhớ khi đã sử dụng xong: *kfree(ptr);*

1. Copy\_from\_user() : sao chép dữ liệu từ không gian người dùng (user space) vào không gian kernel (kernel space). Điều này thường xuyên cần thiết khi một module kernel cần truy xuất dữ liệu từ các thành phần của user space như buffers được cung cấp bởi ứng dụng người dùng.

*unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n);*

* to: Con trỏ đến vùng nhớ trong kernel space nơi dữ liệu sẽ được sao chép đến.
* from: Con trỏ đến vùng nhớ trong user space mà dữ liệu sẽ được sao chép từ đó.
* n: Số lượng byte cần sao chép.
* Hàm trả về số lượng byte không thể sao chép thành công. Nếu giá trị trả về khác 0, có nghĩa là có lỗi xảy ra trong quá trình sao chép. ( tức là nếu đúng thì trả về 0)

1. Copy\_to\_user(): sao chép dữ liệu từ không gian kernel (kernel space) vào không gian người dùng (user space)

*unsigned long copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long n);*

* to: Con trỏ đến vùng nhớ trong không gian người dùng mà dữ liệu sẽ được sao chép đến.
* from: Con trỏ đến vùng nhớ trong không gian kernel mà dữ liệu sẽ được sao chép từ đó.
* n: Số lượng byte cần sao chép.
* Hàm trả về số lượng byte không thể sao chép thành công. Nếu giá trị trả về khác 0, có nghĩa là có lỗi xảy ra trong quá trình sao chép.

# Ioctl

ioctl là một hàm hệ thống được sử dụng để thực hiện các thao tác đặc biệt trên các thiết bị hệ thống. Tên của hàm ioctl là viết tắt của "input/output control". Hàm này cho phép giao tiếp với các thiết bị hệ thống và thực hiện các thao tác như đặt cấu hình, điều khiển và nhận thông tin từ chúng.

* Để sử dụng ioctl , ngoài việc định nghĩa ở user\_space, thì trong kernel space, ta cũng phải định nghĩa nó thông qua struct file\_operation

1. **Trên user space**

*int ioctl(int file\_descriptor, unsigned long request, ...);*

* file\_descriptor: là số mô tả tệp (file descriptor) của thiết bị mà bạn muốn thực hiện thao tác.
* request: là mã yêu cầu cụ thể, chỉ định loại thao tác bạn muốn thực hiện trên thiết bị.
* ...: là các tham số tùy chọn phụ thuộc vào loại yêu cầu. ( thường là NULL)
* Ta có thể định nghĩa request thông qua: ( ioctl name is request)

*#define "ioctl name" \_IOX("magic number","command number","argument type")*

ở đâu IOX có thể là :

“ IO“: một ioctl không có tham số

“ IOW“: một ioctl có tham số ghi (copy\_from\_user)

“ IOR“: một ioctl có tham số đọc (copy\_to\_user)

“ IOWR“: một ioctl có cả tham số ghi và đọc

* Đây Magic Numberlà một số hoặc ký tự duy nhất sẽ phân biệt tập hợp các cuộc gọi ioctl của chúng tôi với các cuộc gọi ioctl khác. đôi khi số chính của thiết bị được sử dụng ở đây.
* Số Lệnh là số được gán cho ioctl. Điều này được sử dụng để phân biệt các lệnh với nhau.
* Cuối cùng là loại dữ liệu.

1. **Trong kernel space**

* Định nghĩa các request giống như user space
* Trong hàm ioctl ( được chỉ định trong struct operation) , định nghĩa các hành động, nhiệm vụ cần thực hiện thông qua swich – case

VD : static long etx\_ioctl**(**struct file \*file, unsigned int cmd, unsigned long arg**)**

**{**

switch**(**cmd**)** **{**

case WR\_VALUE:

if**(** copy\_from\_user**(**&value ,**(**int32\_t\***)** arg, sizeof**(**value**))** **)**

**{**

pr\_err**(**"Data Write : Err!\n"**)**;

**}**

pr\_info**(**"Value = %d\n", value**)**;

break;

case RD\_VALUE:

if**(** copy\_to\_user**((**int32\_t\***)** arg, &value, sizeof**(**value**))** **)**

**{**

pr\_err**(**"Data Read : Err!\n"**)**;

**}**

break;

default:

pr\_info**(**"Default\n"**)**;

break;

**}**

return 0;

**}**

Linux bring-up Device

# GPIO Modul

1. **Các bước viết driver cho 1 ngoại vi cơ bản**

các bước tuân theo

* B1: Xác định bank - chân

* B2: Xác định start\_addr, end\_addr, size cho ngoại vi ( Table 2.1)

* B3 : Xác định offset cho các thanh ghi quan trọng
* B4 : ioremap - ánh xạ dải địa chỉ
* B5 : Viết chương trình tác động vào các thanh ghi quan trọng

cú pháp \*(base\_add + offset/4)

* B5 : Trong hàm exit , ioumap

Các hàm, kiểu cần biết

* \_\_***iomem***: Đây là một vòng loại loại được sử dụng trong nhân Linux để chỉ ra rằng biến nhằm mục đích biểu thị vùng I/O (Đầu vào/Đầu ra) được ánh xạ bộ nhớ. I/O được ánh xạ bộ nhớ là một kỹ thuật trong đó các thanh ghi phần cứng được xử lý như thể chúng là các vị trí bộ nhớ.
* *Ioremap()* : Hàm ioremap() là một hàm kernel trong hệ điều hành Linux, được sử dụng để ánh xạ địa chỉ vật lý thành địa chỉ ảo cho các vùng bộ nhớ I/O. Nó thường được sử dụng trong các mô-đun hạt nhân để truy cập và điều khiển các thanh ghi phần cứng và các thiết bị I/O được ánh xạ bộ nhớ.

*void \_\_iomem \*ioremap(resource\_size\_t offset, unsigned long size);*

+ offset: Đây là địa chỉ vật lý (hoặc offset) của vùng bộ nhớ mà bạn muốn ánh xạ. Nó chỉ ra điểm bắt đầu của khu vực.

+ size: Đây là kích thước của vùng bộ nhớ được ánh xạ, bắt đầu từ offset đã chỉ định.

* Hàm trả về một con trỏ kiểu void \_\_iomem \*, biểu thị địa chỉ ảo trong không gian kernel nơi bắt đầu vùng bộ nhớ được ánh xạ.
* *Iounmapp*() : Hàm iounmap() được sử dụng trong nhân Linux để hủy ánh xạ một vùng bộ nhớ I/O được ánh xạ trước đó bằng hàm ioremap(). Khi một mô-đun hạt nhân ánh xạ một dải địa chỉ vật lý tới một không gian địa chỉ ảo bằng cách sử dụng ioremap(), thì sau này nó sẽ sử dụng iounmap() để giải phóng bộ nhớ được ánh xạ và các tài nguyên liên quan khi chúng không còn cần thiết nữa.

*void iounmap(volatile void \_\_iomem \*addr);*

# GPIO interger base

1. Overview
2. Functions

* **int gpio\_request(unsigned int gpio, const char \*label);:** Chức năng này thường được sử dụng khi một mô-đun muốn điều khiển một chân GPIO cụ thể và cần thông báo cho hệ thống con GPIO về ý định sử dụng chân cụ thể đó.
* gpio: Số chân GPIO mà mô-đun muốn yêu cầu.
* label Một chuỗi đóng vai trò là nhãn hoặc tên cho chân GPIO, giúp xác định nó trong hệ thống.
* **int gpio\_direction\_output(unsigned int gpio, int value):** Hàm được sử dụng trong nhân Linux để đặt hướng của chân GPIO làm đầu ra. Chức năng này là một phần của API GPIO do kernel cung cấp để tương tác với các chân GPIO trên các nền tảng phần cứng khác nhau.
* gpio: The GPIO pin number for which the direction is being set.
* value: The initial value for the GPIO pin, typically either 0 (LOW) or 1 (HIGH).( Việc lựa chọn giá trị ban đầu phụ thuộc vào yêu cầu ứng dụng cụ thể của bạn. Một số thiết bị hoặc mạch có thể cần một trạng thái ban đầu cụ thể để hoạt động bình thường, vì vậy bạn nên tham khảo tài liệu hoặc thông số kỹ thuật của phần cứng mà bạn đang tương tác.)
* **void gpio\_set\_value(unsigned int gpio, int value):** Nó được sử dụng để đặt giá trị logic của chân GPIO, có nghĩa là nó kiểm soát xem chân đó nên ở trạng thái cao (1) hay trạng thái thấp (0). Chức năng này cho phép điều khiển động trạng thái của chân GPIO trong thời gian chạy.
* Hãy nhớ rằng trước khi sử dụng gpio\_set\_value(), bạn thường cần yêu cầu kiểm soát chân GPIO bằng gpio\_request() và đặt hướng của nó làm đầu ra bằng gpio\_direction\_output(). Sau khi bạn sử dụng xong chân GPIO, nó sẽ được giải phóng bằng gpio\_free().

# Sysfs

1. Overview : một hệ thống tệp ảo được cung cấp bởi nhân Linux để hiển thị các đối tượng hạt nhân và thông tin về phần cứng hệ thống cho các ứng dụng trong không gian người dùng. Nó được gắn tại /sys. Mục đích chính của sysfs là cung cấp một cách có tổ chức và nhất quán để các chương trình trong không gian người dùng tương tác và truy xuất thông tin về kernel và các thiết bị có trong hệ thống.
2. Feature

* Thông tin thiết bị: Sysfs cung cấp thông tin về thiết bị và thuộc tính của chúng. Mỗi thiết bị được đại diện bởi một thư mục và thuộc tính của nó là các tệp trong thư mục đó.
* Tham số hạt nhân: Các tham số và cấu hình hạt nhân khác nhau có thể được truy cập thông qua sysfs.
* GPIO (Đầu vào/Đầu ra mục đích chung): Sysfs thường được sử dụng để quản lý các chân GPIO. Người dùng có thể xuất các chân GPIO, đặt hướng và đọc hoặc ghi các giá trị thông qua sysfs.
* Thông tin buýt: Thông tin về xe buýt và thiết bị được kết nối với hệ thống, chẳng hạn như PCI, USB, I2C và SPI, có sẵn trong sysfs.
* Quản lý nguồn: Sysfs cung cấp thông tin về các cài đặt và khả năng liên quan đến nguồn của thiết bị.

1. Điều khiển GPIO thông qua sysfs

**Các bước tuân theo : export pin -> set direction -> set value-> unexport  
VD** : Blink led gpio 31

* Echo 31 /sys/class/gpio/export
* Echo out /sys/class/gpio/gpio31/direction
* Echo 1 /sys/class/gpio/gpio31/value ( turn on led)
* Echo 0 /sys/class/gpio/gpio31/value ( turn off led)
* Echo 31 /sys/class/gpio/unexport