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

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**

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

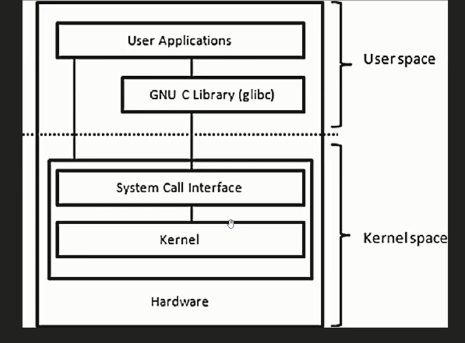
Chmod u+w : thêm quyền write cho User

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

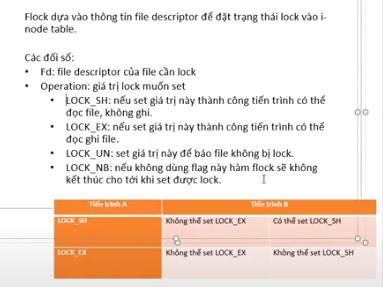
* Kernel điều khiển việc tương tác giữa các tiến trìn và file thông qua 3 bảng:

+ 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

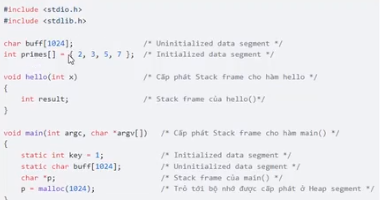
* Có quyền read-write

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

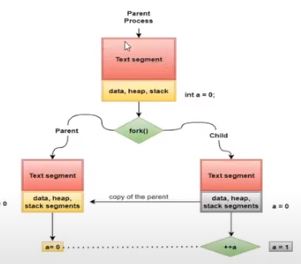
VD :



# Thao tác với process

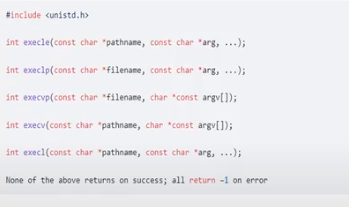
1. Tạo 1 tiến trình mới

* 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*.*

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

execl(path,"hello",NULL);

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

execl(path,"check",NULL);

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)

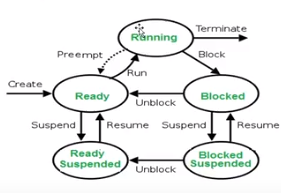
* Thread ID : định danh thread trong process . Là 1 số nguyên hoặc 1 struct( thường sử dụng struct)

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)