Thực hành Unix, Linux (2)

Bộ môn Hệ thống và Mạng máy tính
Khoa Khoa học và kỹ thuật máy tính

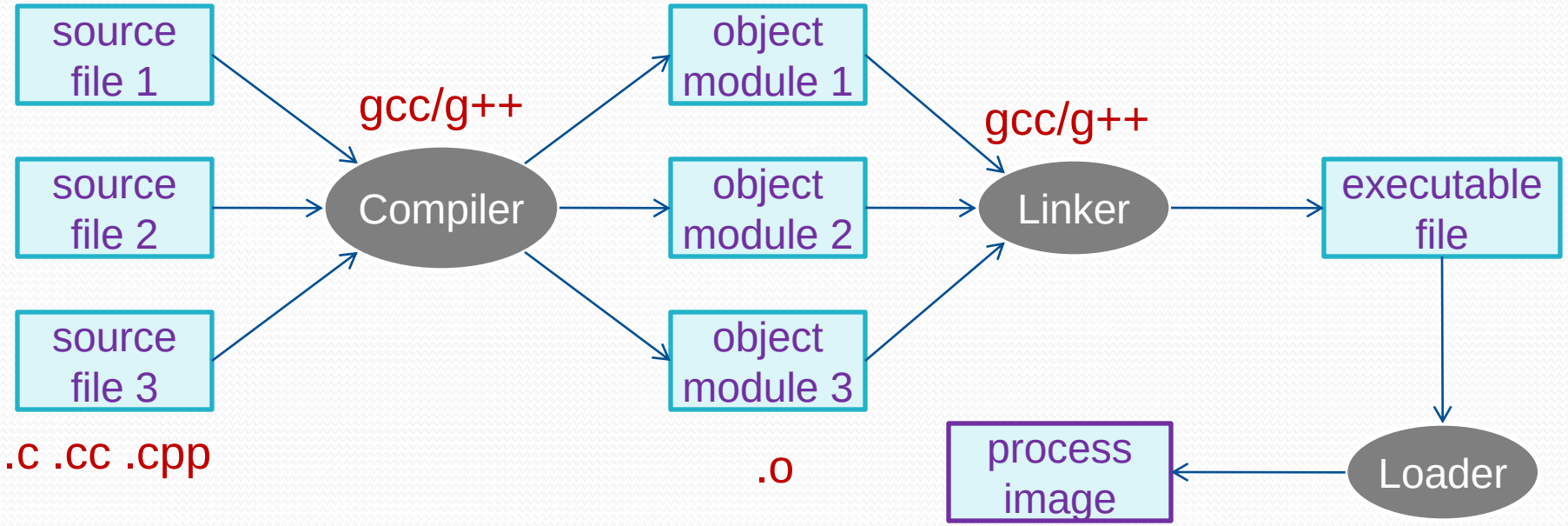
Nội dung

- Biên dịch và thực thi chương trình C/C++
- Cơ bản về process
  - Tổ chức của một process
  - Background và foreground process
  - Các lệnh thao tác với process
- Lập trình với process

Nội dung

- Biên dịch và thực thi chương trình C/C++
- Giới thiệu về process
  - Cơ bản về process
  - Background và foreground process
  - Các lệnh thao tác với process
- Lập trình với process

Quá trình tạo process



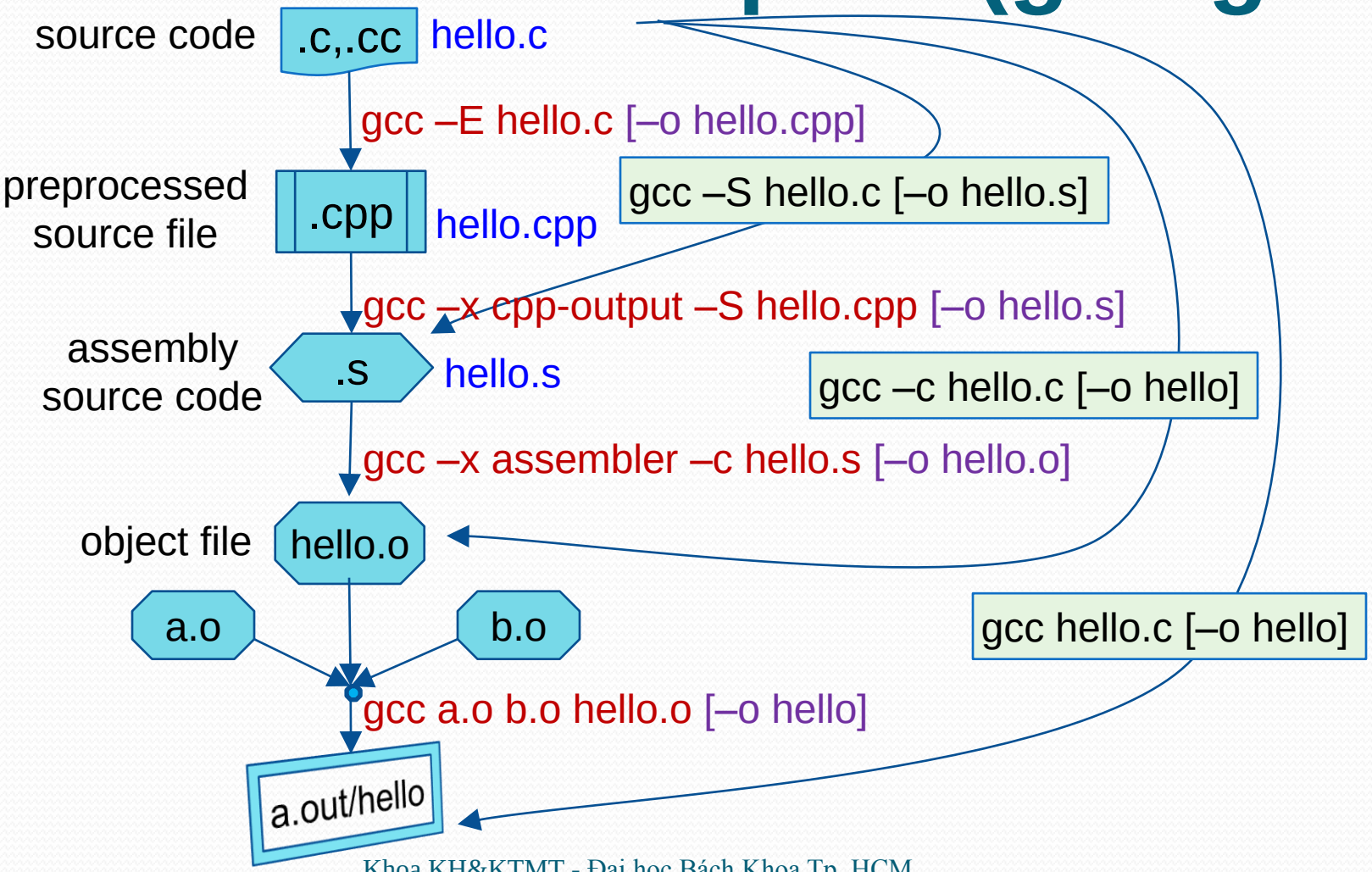
Bộ công cụ phát triển ứng dụng GNU

- GNU Compiler Collection (GCC)
  - Thư viện các hàm tiện ích: libc, libstdc++, …
  - Các trình biên dịch: gcc, g++, gcj, gas, …
  - Trình khử lỗi: gdb
  - Trình tiện ích khác trong binutils như nm, strip, ar, objdump, ranlib (dùng lệnh info binutils để xem thêm)
  - Tiện ích: gmake
  - …

Trình biên dịch GNU C/C++

- Công cụ dùng biên dịch các chương trình C/C++
- Quá trình biên dịch thành file thực thi gồm 4 giai đoạn theo thứ tự như sau:
  1. preprocessing (tiền xử lý)
  2. compilation (biên dịch)
  3. assembly (hợp dịch)
  4. linking (liên kết)
- Ba bước 1, 2, 3 chủ yếu làm việc với một file đầu vào
- Bước 4 có thể liên kết nhiều object module liên quan để tạo thành file thực thi nhị phân (executable binary)
- Lập trình viên có thể can thiệp vào từng bước ở trên

GNU C/C++ compiler (gcc/g++)



Tóm tắt một số tùy chọn của gcc

Tùy chọn	Công dụng
-o FILE	Chỉ định tên của file output (khi biên dịch thành file thực thi, nếu không có **-o filename** thì tên file mặc định sẽ là **a.out**)
-c	Chỉ biên dịch mà không linking (i.e. chỉ tạo ra object file *.o)
-I*DIRNAME*	Chỉ tên thư mục *DIRNAME* là nơi chứa các file header (.h) mà gcc sẽ tìm trong đó (mặc định gcc sẽ tự tìm ở các thư mục chuẩn /usr/include, ...)
-L*DIRNAME*	Chỉ tên thư mục *DIRNAME* là nơi chứa các thư viện (.a, .so) mà gcc sẽ tìm trong đó (mặc định gcc sẽ tự tìm ở các thư mục chuẩn /usr/lib, ...)
-O [n]	Tối ưu mã thực thi tạo ra (e.g. -O2, -O3, hoặc -O)
-g	Chèn thêm mã phục vụ công việc debug
-E	Chỉ thực hiện bước tiền xử lý (preprocessing) mà không biên dịch
-S	Chỉ dịch sang mã hợp ngữ chứ không linking (i.e. chỉ tạo ra file *.s)
-l**foo**	Link với file thư viện có tên là lib**foo** (e.g. -l**m**, -l**pthread**)
-ansi	Biên dịch theo chuẩn ANSI C/C++ (sẽ cảnh báo nếu code không chuẩn)

Biên dịch chương trình C/C++

File main.c

```
#include <stdio.h>
#include "reciprocal.h"
int main (int argc, char **argv)
{
   int i;
   i = atoi(argv[1]);
   printf("The reciprocal of %d is %g\n “,i,reciprocal(i));
   return 0;
}
```

Biên dịch chương trình C/C++

File reciprocal.h

```
extern double reciprocal(int i);
```

File reciprocal.c

```
#include <assert.h>   /* some debug routines here */
#include "reciprocal.h"
double reciprocal(int i){
   assert (i != 0);              /* used for debugging */
   return 1.0/i;
}
```

Biên dịch chương trình C/C++

- Biên dịch (không link) một file chương trình nguồn C đơn lẻ

```
gcc -c main.c
```

- Biên dịch (không link) có sử dụng các file *.*h* trong thư mục *include*

```
gcc -c -I ../include reciprocal.c
```

- Biên dịch (không link) có tối ưu mã

```
gcc -c -O2 main.c
```

- Biên dịch có kèm thông tin phục vụ debug => kích thước file output lớn

```
gcc -g -c reciprocal.c
```

Biên dịch chương trình C/C++

- Liên kết (link) nhiều file đối tượng (object files) đã có

```
gcc -o myapp main.o reciprocal.o
```

- Liên kết object files với các thư viện (libraries) khác

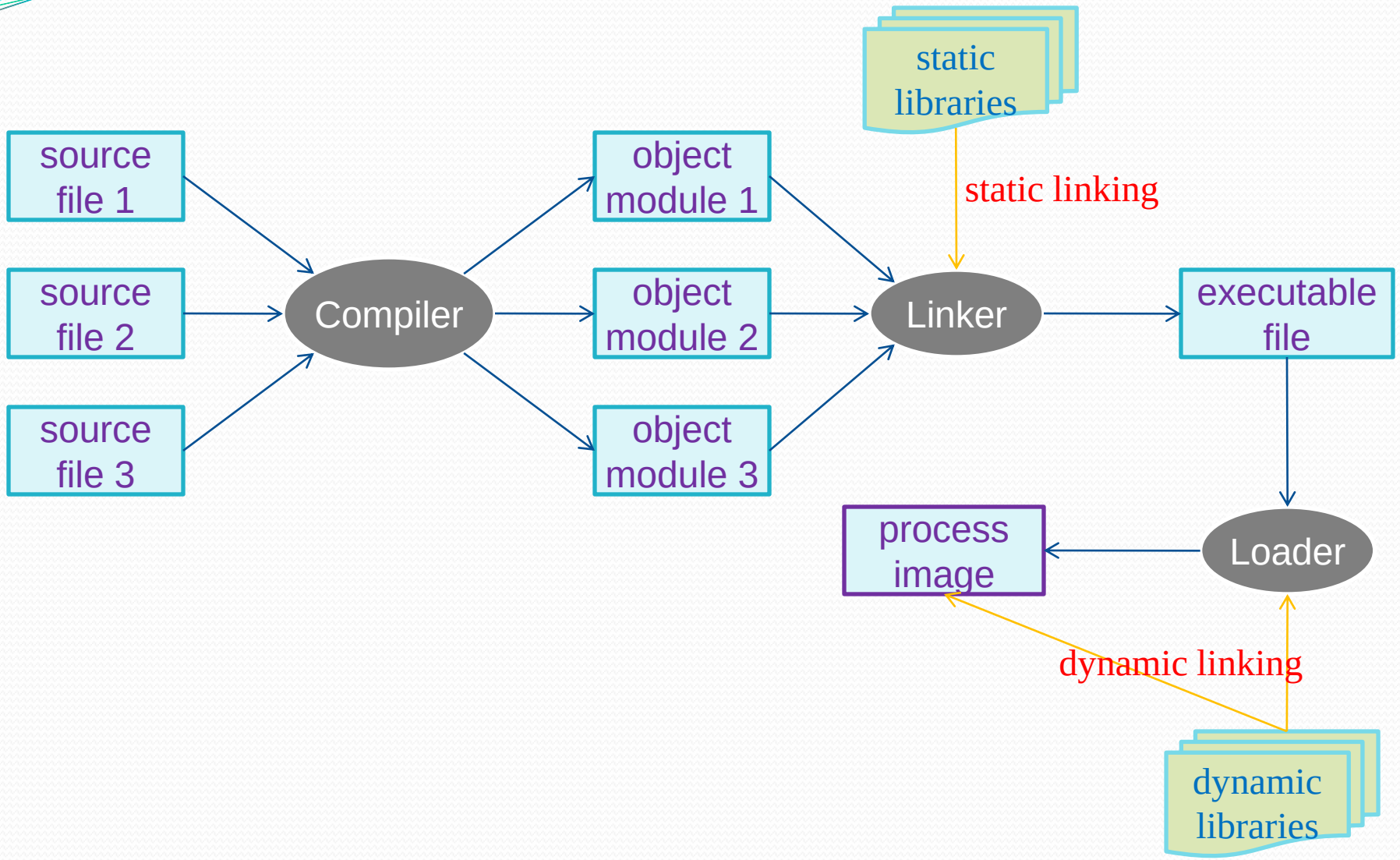
```
gcc -o myapp main.o -lpthread
gcc -o myapp main.o -L/usr/somelib -lutil
gcc -o myapp main.o -L. -ltest
```

Biên dịch chương trình C/C++

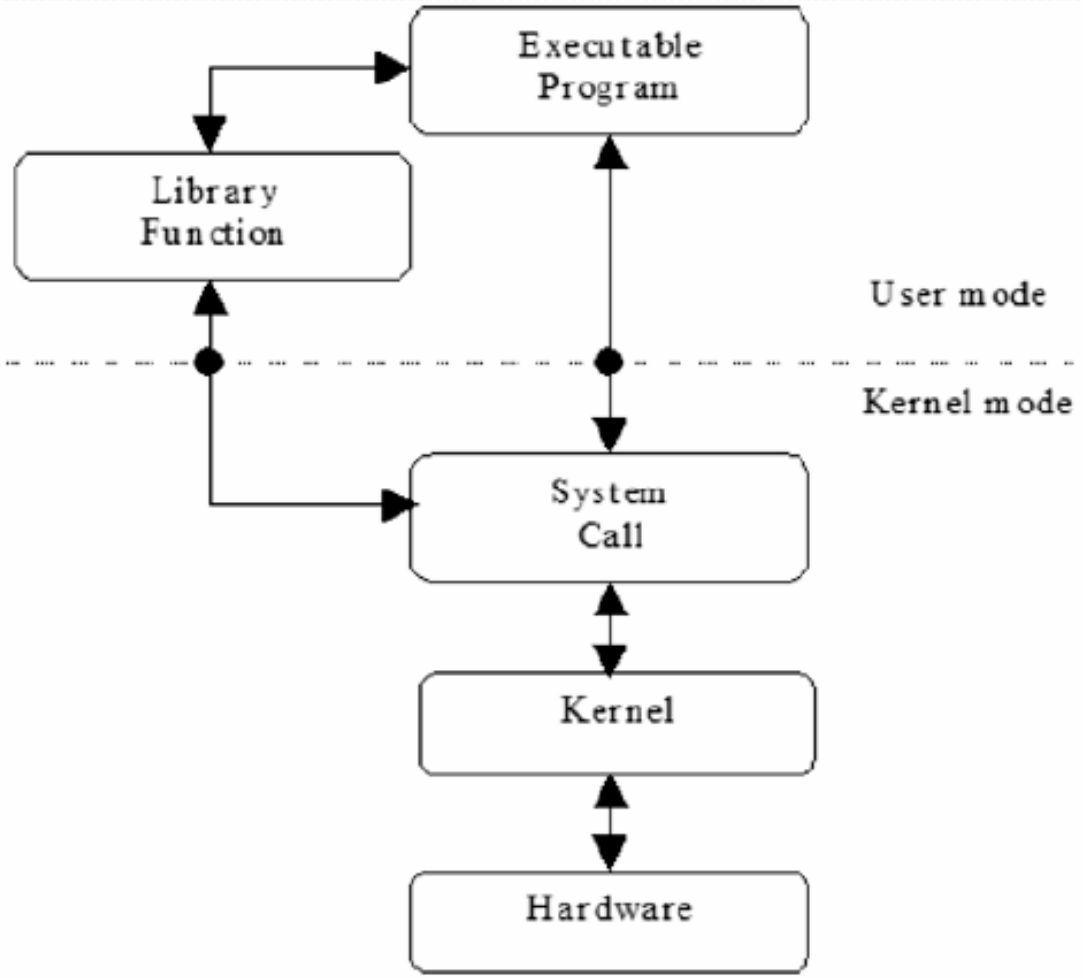
- Lưu ý khi biên dịch trong Linux
  - Dùng g++ nếu chương trình có chứa mã C lẫn C++
  - Dùng gcc nếu chương trình chỉ có mã C
  - File thực thi tạo ra không có đuôi .exe, .dll như môi trường Windows
- Giả sử ứng dụng của bạn gồm nhiều hơn một file source code, (e.g. *main.c* và *reciprocal.c*). Để tạo thành chương trình thực thi, bạn có thể biên dịch trực tiếp bằng một lệnh gcc như sau:

```
$ gcc -o myapp main.c reciprocal.c
```

- Cách làm thủ công như trên sẽ bất tiện và không hiệu quả khi ứng dụng gồm quá nhiều file (khoảng>10 files ???)
  - Tham khảo thêm công cụ rất hữu ích là *GNU make*


static libraries
static linking
source file 1
source file 2
source file 3
Compiler
object module 1
object module 2
object module 3
Linker
executable file
process image
Loader
dynamic linking
dynamic libraries

Thư viện lập trình trong Linux



Các loại thư viện lập trình

- Thư viện liên kết tĩnh (static library)
  - Là tập hợp các file object tạo thành một file đơn nhất
  - Tương tự file .LIB trên Windows
  - Khi bạn chỉ định liên kết ứng dụng của mình với một static library thì linker sẽ tìm trong thư viện đó để trích xuất những file object mà bạn cần. Sau đó, linker sẽ tiến hành liên kết các file object đó vào chương trình của bạn.
- Thư viện liên kết động (dynamic, shared library)
  - Tương tự thư viện dạng .DLL của Windows
- Thư mục chứa thư viện chuẩn
  - `/usr/lib, /lib`

Tạo thư viện liên kết tĩnh

- Giả sử bạn có hai file mã nguồn chứa hàm là a.c và b.c

a.c

```
int func1(){
    return 7;
}
```

b.c

```
double  func2(){
    return 3.14159;
}
```

Tạo thư viện liên kết tĩnh

- Tạo thư viện tĩnh tên là ***libab.a***

1. Biên dịch tạo các file object

```
$ gcc -c a.c b.c
```

2. Dùng lệnh ***ar*** để tạo thành thư viện tĩnh tên là ***libab.a***

```
$ ar cr libab.a a.o b.o
```

3. Có thể dùng lệnh ***nm*** để xem lại kết quả

```
$ nm libab.a
```

4. Có thể dùng lệnh file để xem file libab.a là loại file gì

```
$ file libab.a
```

Dùng thư viện liên kết tĩnh

- Tạo ứng dụng có sử dụng hàm thư viện trong a.c

myapp.c

```
int main(){
  printf("Ket qua ham func1: %d\n" ,func1());
  exit(0);
}
```

Dùng thư viện liên kết tĩnh

- Biên dịch không link đến thư viện tĩnh *libab.a*

```
$ gcc -o myapp myapp.c
/tmp/cc2dMic1.o: In function `main':
/tmp/cc2dMic1.o(.text+0x7): undefined reference to
  `func1'
collect2: ld returned 1 exit status
```

- Biên dịch có link đến thư viện tĩnh *libab.a*

```
$ gcc -o myapp myapp.c -L. -lab  hoặc
$ gcc -o myapp myapp.c libab.a
$ ./myapp
Ket qua dung ham func1: 7
```

Tạo thư viện liên kết động

- Tạo thư viện liên kết động *libab.so* từ *a.c* và *b.c*
  1. Biên dịch tạo các file object có dùng tùy chọn *–fPIC*

     ```
     $ gcc -c -fPIC a.c b.c
     ```
  2. Tạo thư viện liên kết động tên là *libab.so*

     ```
     $ gcc -shared -fPIC -o libab.so a.o b.o
     ```
  3. Có thể dùng lệnh file để xem file *libab.so* là loại file gì

     ```
     $ file libab.so
     libab.so: ELF 32-bit LSB shared object, Intel 80386,
       version 1 (SYSV), not stripped
     ```

Dùng thư viện liên kết động

- Biên dịch không link đến thư viện động *libab.so*

```
$ gcc -o myapp myapp.c
/tmp/cc2dMic1.o: In function `main' :
/tmp/cc2dMic1.o(.text+0x7): undefined reference to  'func1'
collect2: ld returned 1 exit status
```

- Biên dịch có link đến thư viện động *libab.so*

```
$ gcc -o myapp myapp.c -L. -lab hoặc
$ gcc -o myapp myapp.c libab.so
$ ./myapp
./myapp: error while loading shared libraries: libab.so: cannot
  open shared object file: No such file or directory
```

Dùng thư viện liên kết động

- Nguyên nhân: do loader tìm trong thư mục thư viện chuẩn như */usr/lib*, */lib* không có *libab.so*
- Cách giải quyết

```
# cp libab.so /lib
$ ./myapp
Ket qua dung ham func1: 7
```

hoặc

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
$ ./myapp
Ket qua dung ham func1: 7
```

Dùng thư viện liên kết động

- Một số chú ý khi lập trình với thư viện liên kết động
  - Kiểm tra xem ứng dụng cuối cùng của mình tạo ra phụ thuộc vào các thư viện liên kết động nào bằng lệnh *ldd*. Nếu bị thiếu thư viện thì phải khắc phục theo 2 cách ở trên

```
$ ldd myapp
libab.so=>not found
libc.so.6=>/lib/i686/libc.so.6(0x42000000)
/lib/ld-linux.so.2 (0x40000000)
```

  - Trong thư mục hiện tại có 2 thư viện là *libab.a* và *libab.so*. Khi đó, linker sẽ ưu tiên liên kết thư viện *.so* trước. Muốn chỉ định buộc linker tiến hành liên kết tĩnh với thư viện *libab.a* thì thêm tùy chọn *–static*

Nội dung

- Biên dịch và thực thi chương trình C/C++
- Giới thiệu về process
  - Cơ bản về process
  - Background và foreground process
  - Các lệnh thao tác với process
- Lập trình với process

Cơ bản về process

- Process: chương trình đang thực thi.
- User có thể theo dõi trạng thái của process, tương tác với process
- Có hai loại user process trong hệ thống
  - Foreground process
  - Background process
- Các "process" thực hiện các công việc của hệ điều hành còn gọi là các kernel_thread, daemon

Theo dõi các process

- Xem trạng thái các process (process status)

  `ps [option]`

- Options
  - -e chọn tất các process
  - -f liệt kê tất cả (full) các thuộc tính
  - -A liệt kê tất cả processs
  - …

Background và foreground process

- Background process
  - Không có giao diện
  - Không thể tương tác trực tiếp với chương trình
  - Chạy nhiều process cùng lúc
- Foreground process
  - Có giao diện
  - Tương tác trực tiếp với chương trình
  - Chờ process kết thúc mới chạy được process khác

Thực thi foreground process

- Khi gõ lệnh tương ứng với tên chương trình theo cách thông thường
- Khi click vào icon trên giao diện đồ hoạ tương ứng với chương trình.
- Foreground process tương tác được với người dùng qua thiết bị nhập chuẩn là bàn phím.
- Kết xuất của chương trình chủ yếu là thiết bị xuất chuẩn là màn hình.
- Trình thông dịch lệnh sẽ bị blocked cho tới khi foreground process kếtthúc

Kết thúc thực thi foreground process

- Dùng tổ hợp phím *Ctrl-C*

- Ví dụ

```
$ find / -name "*.ps" -print
...
...
^C
```

Tạm hoãn thực thi foreground process

- Dùng tổ hợp phím *Ctrl-Z*

  ⇒Process tương ứng chuyển sang trạng thái suspended

- Ví dụ

```
$ find / “*.profile” -print
...
^Z
[1]+ Stopped     find / “*.profile” -print
$ ps
PID   TTY    TIME   CMD
2750 pts/1 00:00:00 bash
2881 pts/1 00:00:00 find
2883 pts/1 00:00:00 ps
```

Tạm hoãn thực thi foreground process (2)

- Nếu muốn cho process tiếp tục thực thi ở chế độ foreground, dùng lệnh *fg n* (trong đó n là chỉ số của job hiển thị trong ngoặc vuông, ví dụ [1], [4], …), còn muốn process thực thi ở chế độ background thì dùng lệnh *bg n*.
- Ví dụ

```
$ jobs
[1]+ Stopped    find / “*.profile” -print
$ fg 1 hoặc
$ bg 1
```

Thực thi process ở background

- Thêm dấu & (ampersand) vào cuối lệnh
- Ví dụ

```
$ find / "*.profile" -print > kq&
[1] 2548
```

- Trình thông dịch lệnh tạo ra một process tương ứng với chương trình đó đồng thời in ra *job number* và *PID* của process được tạo ra.
- Ngay sau khi thực thi, trình thông dịch sẵn sàng nhận lệnh mới (không bị blocked như đối với *foreground process*)

Thực thi background process

- Background process vẫn xuất kết quả ra *standard output* là màn hình trong lúc thực thi

  => cần tái định hướng *standard output* để tránh mất dữ liệu xuất.

- Người dùng không thể tương tác với chương trình qua *standard input* là bàn phím

  => cần phải tái định hướng *standard input* thông qua file nếu process đó cần nhập dữ liệu.

Quản lý background process

- Liệt kê các job đang hoạt động – dùng lệnh *jobs*

```
$ jobs -l
[1]+ 3584 Running                 xterm-g 90x55
[2]- 3587 Running                 xterm-g 90x55
```

- Đối với các quá trình, có thể:
  - Chuyển process từ thực thi *background* sang *foreground* và ngược lại dùng lệnh *fg* hoặc *bg*
  - Kết thúc một quá trình

Chuyển foreground thành background process

1. Trì hoãn quá trình đó (bằng *Ctrl + Z*)
2. Dùng lệnh *bg* (background) để chuyển process sang chế độ thực thi background.

- Ví dụ

```
$ ls -R / > kq
...
^Z
[1]+ Stopped                 ls -R / > kq
$ bg        (vì chỉ có một jobs nên bg không cần tham số)
[1]+ ls -R / > kq&
```

Chuyển background thành foreground process

- Dùng lệnh *fg* (foreground)
- Ví dụ

```
$ ls -R / > kq &
[1] 2959
$ jobs
[1]+ Running    ls -R / > kq &
$ fg 1
```

Kết thúc quá trình

- Dùng lệnh kill

  `kill [-signal] process_identifier`

- Cần dùng lệnh *ps* trước để biết PID của quá trình.
- Có thể dùng lệnh đơn giản như sau:

  `kill process_identifier` *hoặc*

  `kill -9 process_identifier`

Nội dung

- Biên dịch và thực thi chương trình C/C++
- Giới thiệu về process
  - Cơ bản về process
  - Background và foreground process
  - Các lệnh thao tác với process
- Lập trình với process

Lập trình với process

- Nội dung
  - Kiến trúc hệ thống *NIX
  - Tổ chức của process
  - Xử lý tham số dòng lệnh (command line arguments)
  - Tạo mới và kết thúc process
  - Gọi thực thi lệnh/chương trình khác bằng system(), exec…()

Lập trình với process

- Nội dung
  - Kiến trúc hệ thống *NIX
  - Tổ chức của process
  - Xử lý tham số dòng lệnh (command line arguments)
  - Tạo mới và kết thúc process
  - Gọi thực thi lệnh/chương trình khác bằng system(), exec…()

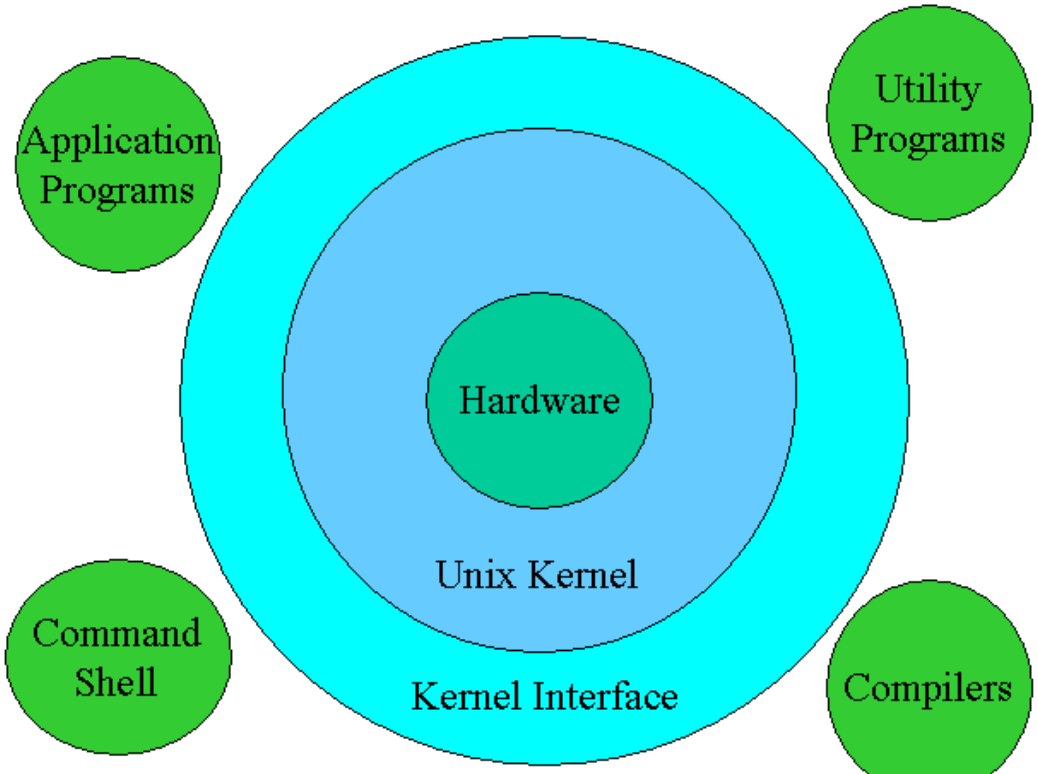
Sơ lược về *NIX

- Đơn khối (monolithic)
- Đa nhiệm (multitasking)
- Nhiều người dùng đồng thời (multiuser)
- Đa dụng (general purpose)
- Chia sẻ thời gian (time-sharing)
- Bảo mật

Một số nhánh phát triển

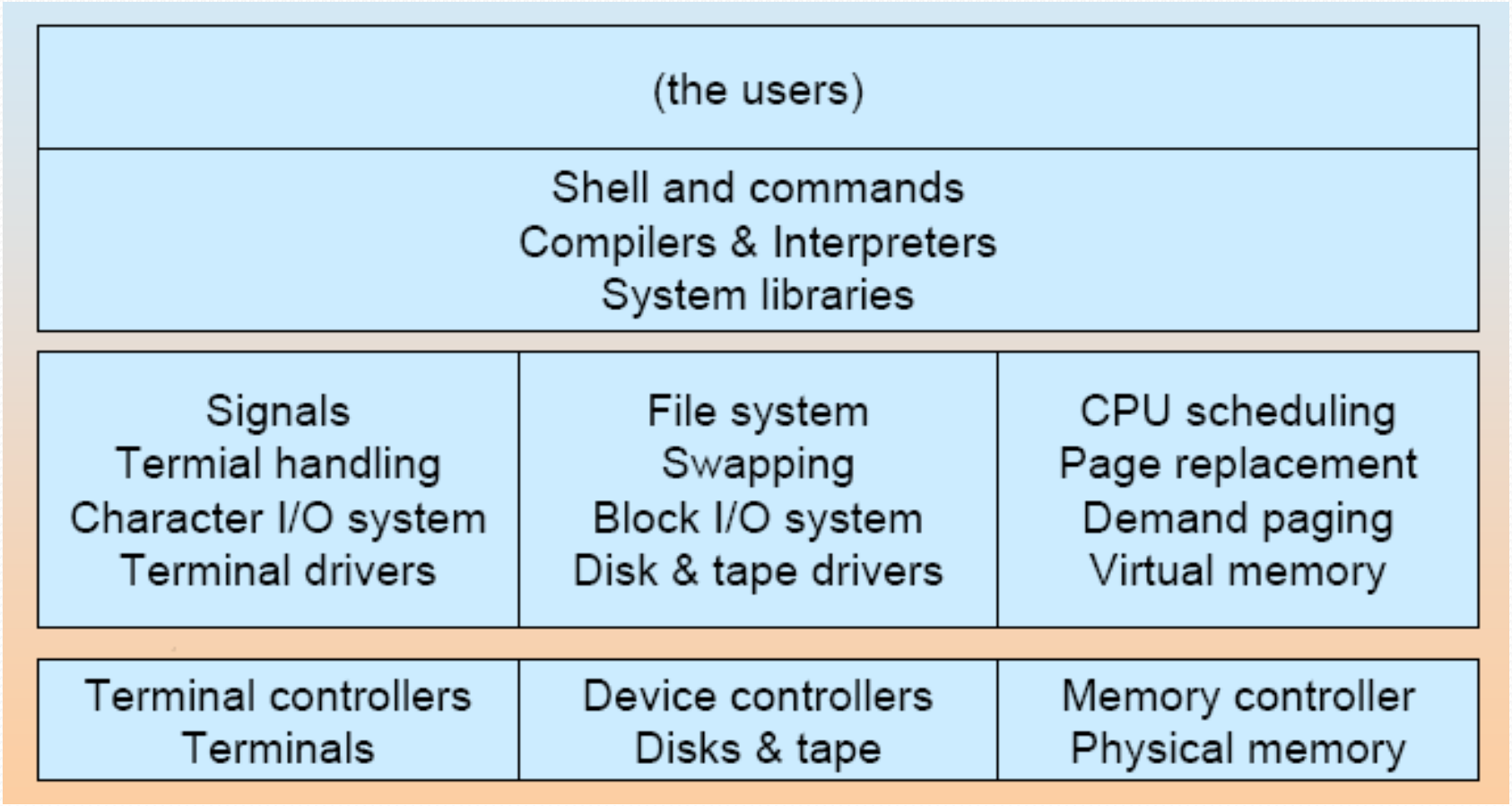
- BSD UNIX: California Univ. of Berkeley
- System V: AT&T
- SunOS/Solaris: Sun Microsystem
- AIX: IBM Corp.
- HP-UX: Hewlett-Packard
- Linux: Linus Torvalds

Kiến trúc tổng quan

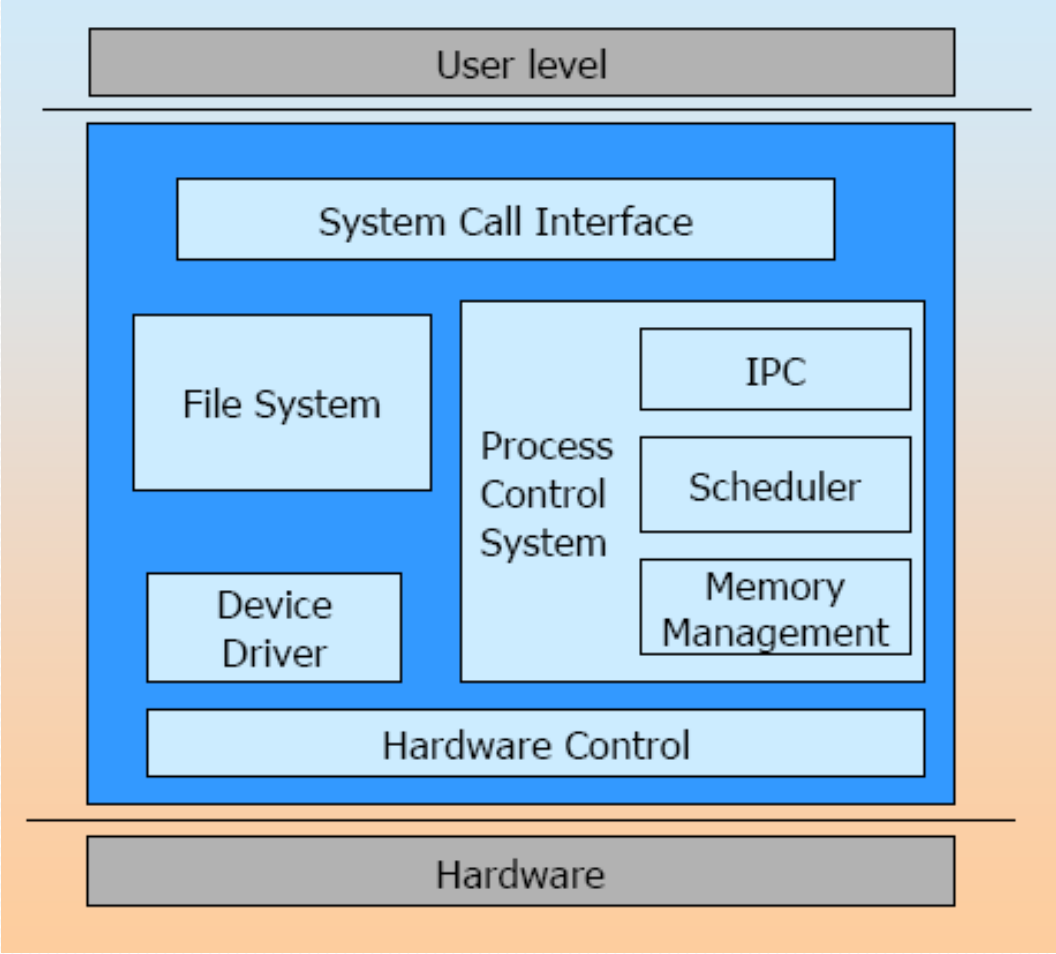


Unix Architecture

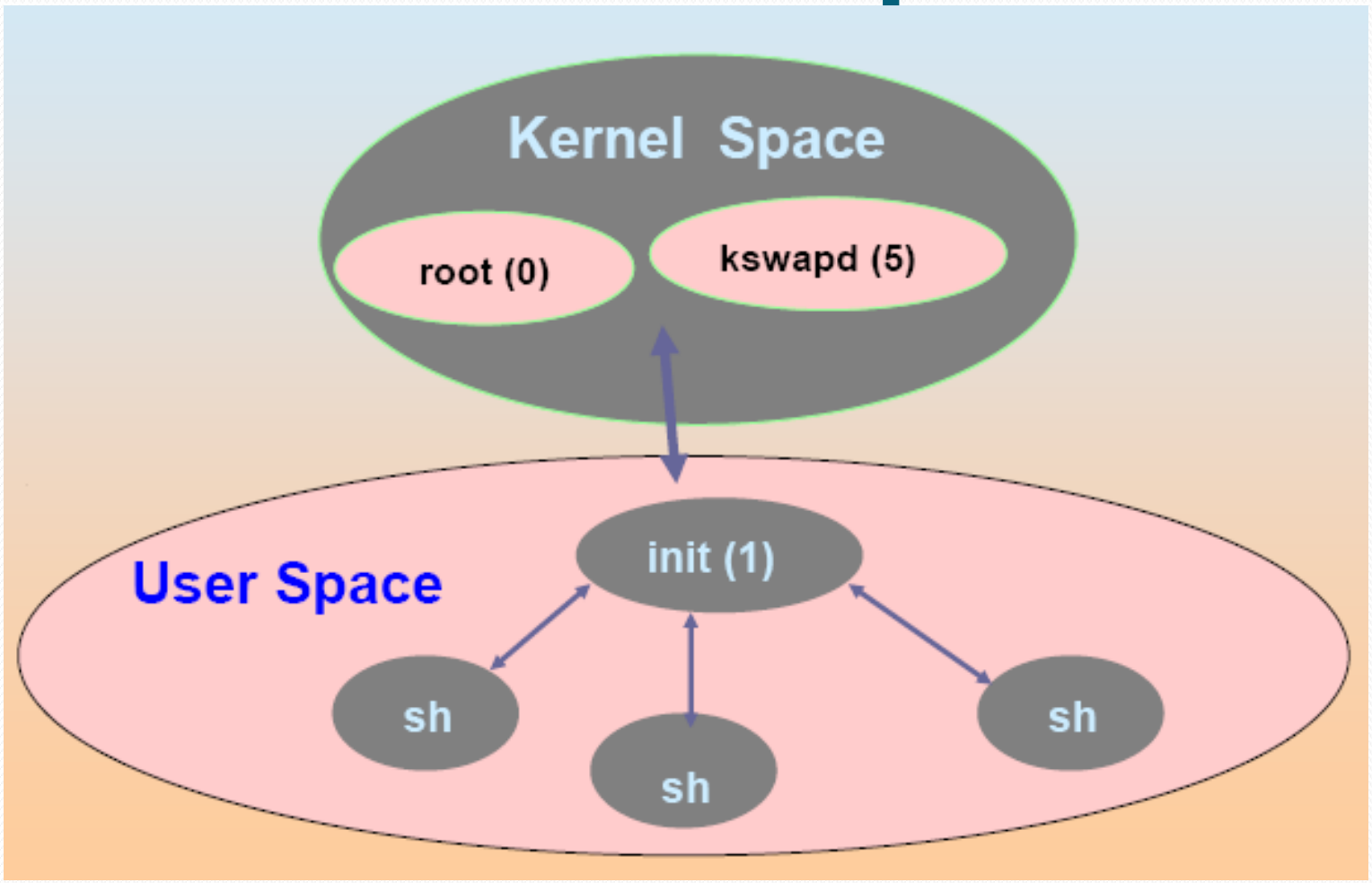
Kiến trúc luận lý



*NIX Kernel


User level
System Call Interface
File System
IPC
Process Control System
Scheduler
Memory Management
Device Driver
Hardware Control
Hardware

Kernel vs. user space



Lập trình với process

- Nội dung
  - Kiến trúc hệ thống *NIX
  - Tổ chức của process
  - Xử lý tham số dòng lệnh (command line arguments)
  - Tạo mới và kết thúc process
  - Gọi thực thi lệnh/chương trình khác bằng system(), exec…()

Quản lý các quá trình

- Đa nhiệm
- Tác vụ -> process.
- Mỗi process có:
  - Không gian địa chỉ (address space)
  - Code thực thi
  - Các vùng chứa dữ liệu
  - Stack
- Trạng thái process:
  - registers, program counter, stack pointer, ….

Danh định của process

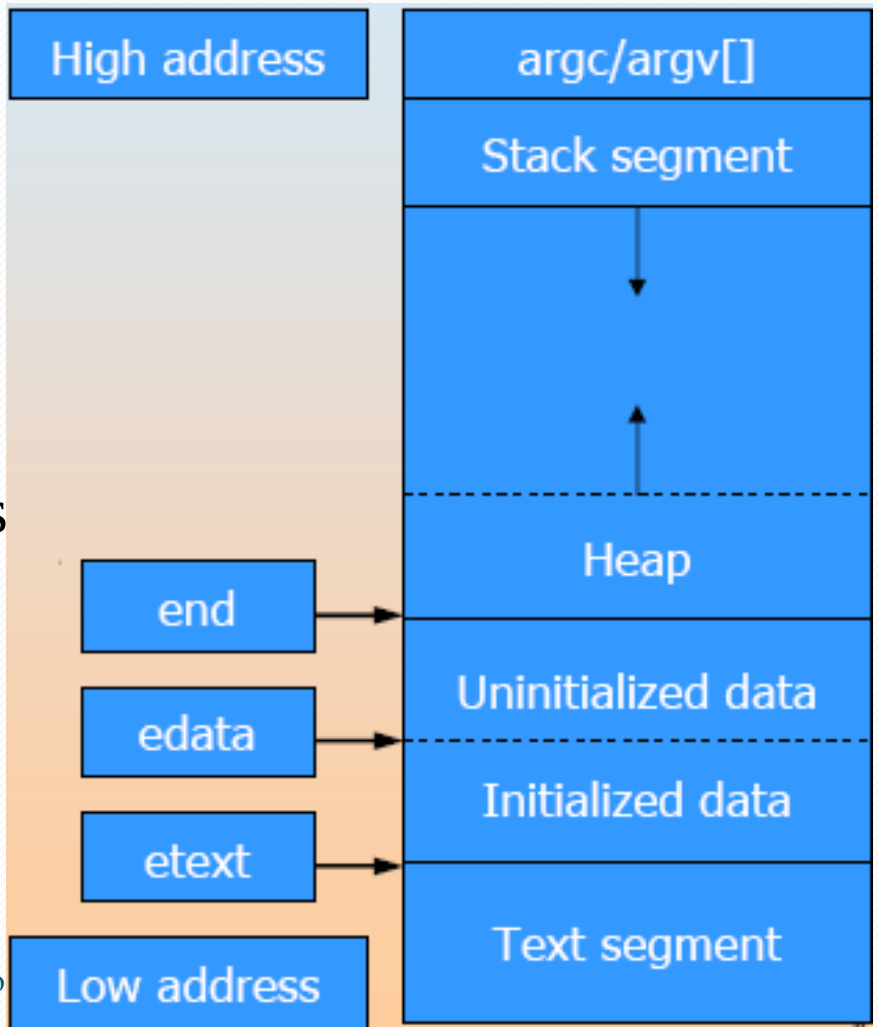
- Process identifier (PID) duy nhất, tăng dần từ 0
- Một số PID đặc biệt:
  - 0: root
  - 1: init
  - …

Bộ nhớ của process

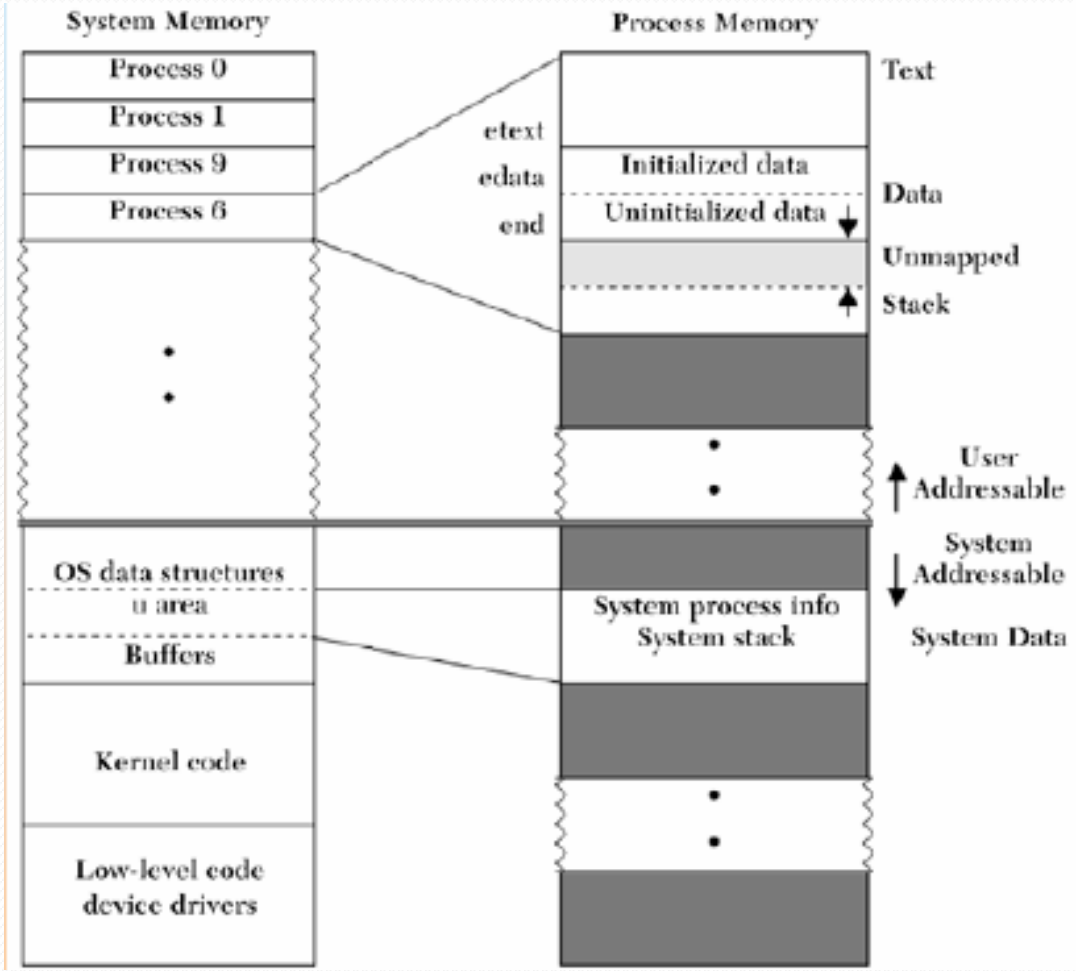
- **Text**: chứa chương trình – code thực thi - chứa các instruction dành cho CPU thực hiện - read only.
- **Data:** vùng dữ liệu - chứa các biến được khai báo tĩnh hoặc động – xin cấp phát trong lúc thực thi.
- **Stack:** chứa trạng thái và các thông tin liên quan đến việc gọi hàm.

Cấu trúc bộ nhớ

- Process memory layout:
  - Text segment (code)
  - Data segment
  - Stack
  - Heap
  - Command-line arguments
  - Environment variables



Cấu trúc bộ nhớ



Địa chỉ bộ nhớ

- Địa chỉ do process tham khảo chỉ là địa chỉ ảo
- Có thể truy xuất thông tin bộ nhớ qua các biến toàn cục:
  - **etext:** địa chỉ sau vùng text
  - **edata:** địa chỉ kết thúc vùng initialized data
  - **end:** địa chỉ bắt đầu vùng heap
- Định nghĩa macro để in địa chỉ một biến

```
#define PADDR(x) printf(#x " at %u\n", &x)
```

```
#include <stdio.h>
#include <stdlib.h>
#define PADDR(x) printf(#x " at %u\n", &x);
extern unsigned etext, edata, end;
int a = 0, b;
int main(int argc, char *argv[]) {
   printf("End of text seqment at %u\n", &etext);
   printf("End of initialized statics and externals at
     %u\n", &edata);
   printf("End of uninitialized statics and externals at
     %u\n", &end);
   PADDR(a);
   PADDR(b);
   return 0;
}
```

Lập trình với process

- Nội dung
  - Kiến trúc hệ thống *NIX
  - Tổ chức của process
  - Xử lý tham số dòng lệnh (command line arguments)
  - Tạo mới và kết thúc process
  - Gọi thực thi lệnh/chương trình khác bằng system(), exec…()

Lấy đối số và biến môi trường

- Chương trình C

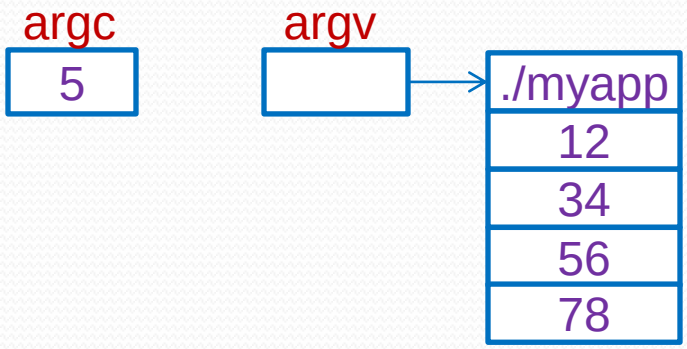
```
int main(int argc, char *argv[]) {
}
```

- Trong đó
  - int argc: số tham số của chương trình khi chạy
  - char *argv[]: danh sách các tham số
- Ngoài ra, còn có các biến ngoại (external variable)
  - extern char **environ: danh sách biến môi trường

Lấy đối số và biến môi trường

- Gọi thực thi chương trình myapp:

```
$ ./myapp 12 34 56 78
The biggest integer is 78
```



```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
int main(int argc, char *argv[]) {
   int i;
   printf( “Number of arguments is %d\n",argc);
   printf( “Arguments:\n");
   for(i=0; i<argc; i++)
     printf("argv[%d]=%s\n",i,argv[i]);
   getchar();
   for (i=0; environ[i]!=(char *)0; i++)
     printf("%s\n",environ[i]);
   return 0;
}
```

Lấy PID của process

```
#include <unistd.h>

pid_t getpid(void);
```

Lấy PID của process hiện hành

```
pid_t getppid(void);
```

Lấy PID của process cha

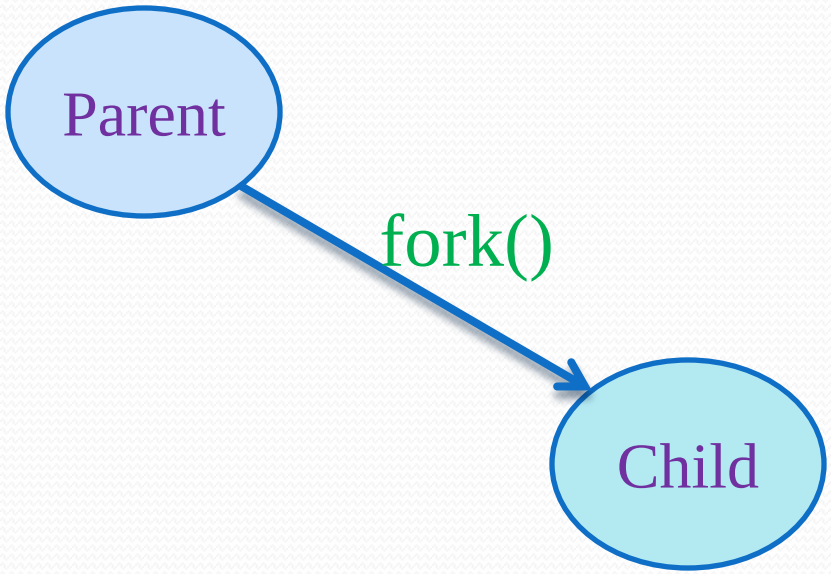
Ví dụ

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("Processid: %d\n", getpid());
   printf("Parent process id: %d\n", getppid());
   return 0;
}
```

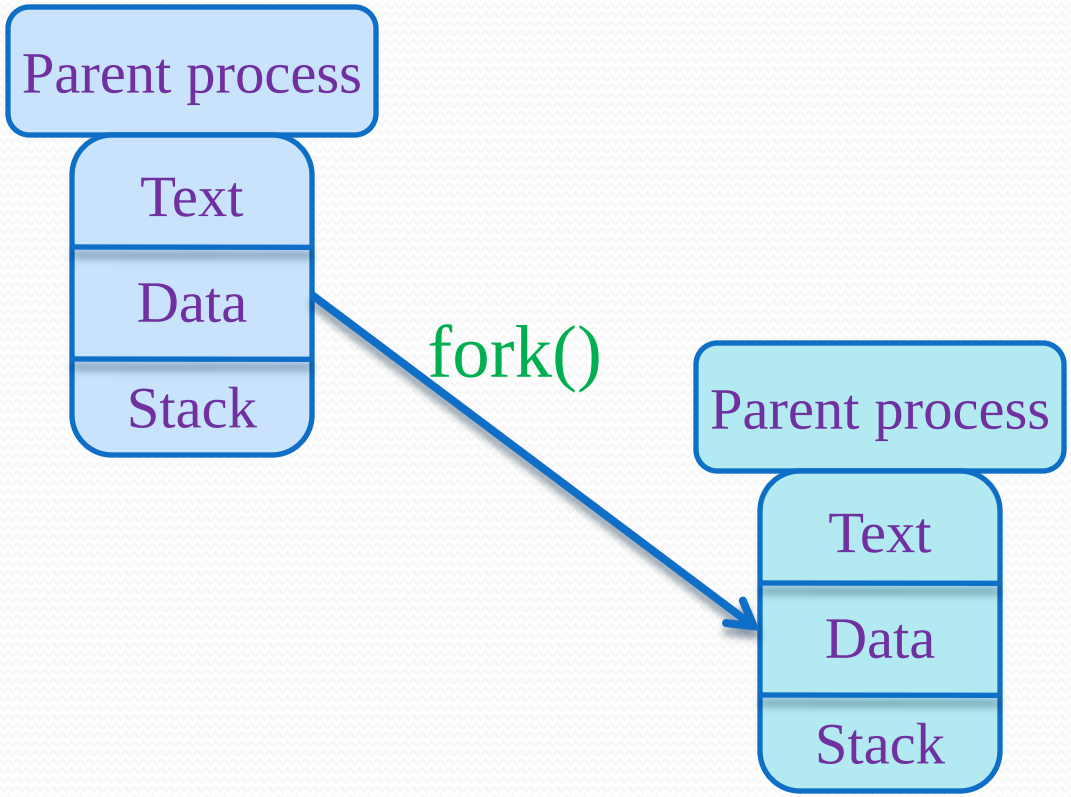
Lập trình với process

- Nội dung
  - Kiến trúc hệ thống *NIX
  - Tổ chức của process
  - Xử lý tham số dòng lệnh (command line arguments)
  - Tạo mới và kết thúc process
  - Gọi thực thi lệnh/chương trình khác bằng system(), exec…()

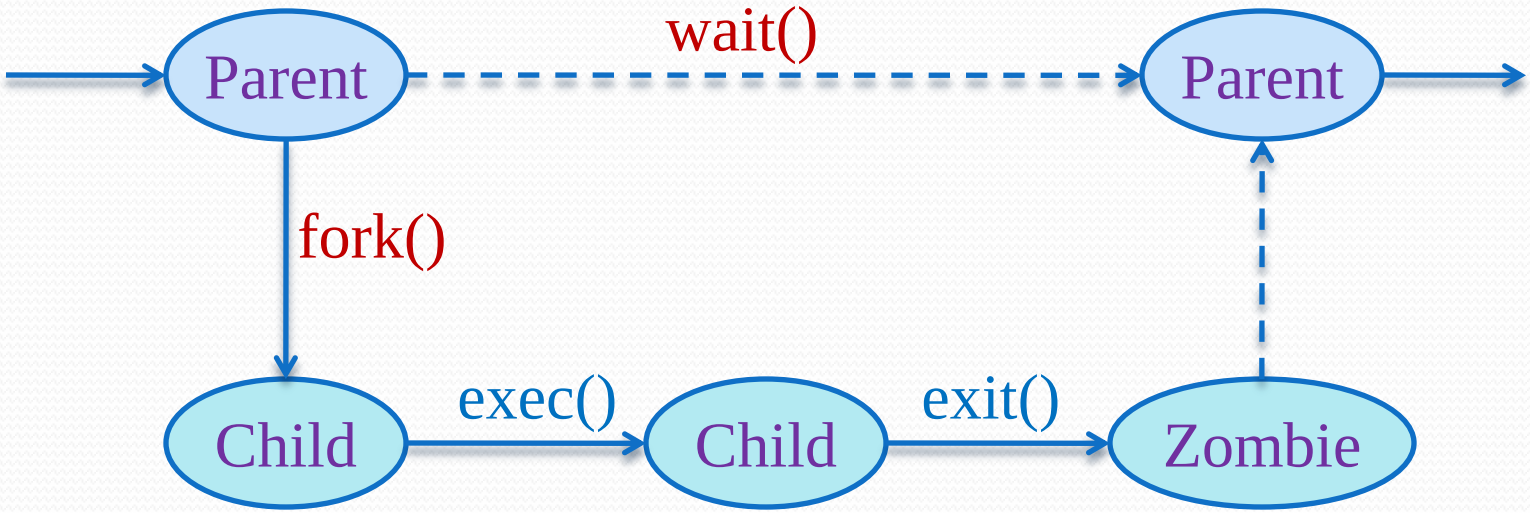
Tạo process



Tạo process


Parent process
Text
Data
Stack
fork()
Parent process
Text
Data
Stack

Chu kỳ sống của process



Tạo process

- `pid_t fork(void);`

- Nếu thành công:
  - trả về 0 trong thân process con
  - PID của con (>0) trong thân process cha.

- Nếu thất bại, trả về -1 và lý do kèm theo:
  - **ENOMEM:** không đủ bộ nhớ
  - **EAGAIN:** số process vượt quá giới hạn cho phép

Ví dụ

- Dạng mẫu chương trình

```
pid_t pid;
pid = fork();
if (pid==0) {
    // child code here
} else if (pid>0) {
    // parent code here
} else {
    // error warning
}
```

```
#include <stdio.h>
#include <unistd.h>
int main() {
   int pid;
   if ((pid=fork())==0) {
     printf("Child process output: PID=%d\n",getpid());
     printf("My parent PID is %d\n\n",getppid());
   }else if(pid>0) {
     printf("Parent process output: PID=%d\n “,getpid());
     printf("Child PID=%d\n “,pid);
   }else {
     printf("Fork error!\n");
     exit(1);
   }
   return 0;
}
```

Kết thúc process

- Dùng system call `exit()`

- *Orphaned process*: process cha kết thúctrước -> process con sau đó sẽ có cha là *init* (PID=1)
- *Zombied process*
  - Process kết thúc nhưng chưa báo trạng thái cho process cha biết.
  - Dùng hàm *wait()* hay *waitpid()* ở process cha để lấy trạng thái trả về từ process con.

Đợi process con

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Trả về PID của process con thay đổi trạng thái, hoặc -1 nếu thất bại

Đối số *pid*:

< -1: Đợi bất kỳ process con nào có process group ID bằng với |pid|

= -1: Đợi bất kỳ process con nào

= 0: Đợi bất kỳ process con nào có process group ID bằng với process group ID của process gọi.

> 0: Đợi pocess có PID bằng với giá trị này

Đợi process con

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

Tham khảo *manpage* của system call `waitpid()`, `wait()`

Ví dụ

```
#include <stdio.h>
#include <unistd.h>
#define PROCESSNUM 3 /* Number of processes */
int main(int argc, char *argv[]) {
   int count, retval, child_no;
   /* Creating processes */
   retval=1;
   for (count=0; count<PROCESSNUM; count++) {
     if (retval!=0) retval=fork();
     else break;
   }
```

```
    /* Execution of child processes */
    if (retval==0) {
      child_no=count;
      printf("Child #%d has PID: #%d\n" , child_no,getpid());
      sleep(rand()%5);
    } else {
      /* Waiting for children termination */
      for (count=0; count<PROCESSNUM; count++)
        wait(NULL);
      printf( "Parent has PID (process-id): #%d\n" , getpid());
    }
    return 0;
}
```

Lập trình với process

- Nội dung
  - Kiến trúc hệ thống *NIX
  - Tổ chức của process
  - Xử lý tham số dòng lệnh (command line arguments)
  - Tạo mới và kết thúc process
  - Gọi thực thi lệnh/chương trình khác bằng system(), exec…()

Các hàm gọi thực thi chương trình

- `int system(const char *string);`

- `int execl(const char *path, const char *arg, ...)`
- `int execv(const char *path, const char *argv[ ])`

- `int execlp(const char *lename, const char *arg, ...)`
- `int execvp(const char *lename, const char *argv[ ])`

- `int execle(const char *path, const char *arg, ..., const char **env)`
- `int execve(const char *path, const char *argv[ ], const char **env)`

Hàm system()

- Xử lý một lệnh được chỉ ra trong thông số *string* và trả về sau khi lệnh được thực thi xong
- Thực chất thì lời gọi hàm *system(string)* sẽ thực hiện lệnh *sh -c string*
- Giá trị trả về:
  - 0: thành công
  - -127: mã lỗi không khởi động shell để thực hiện lệnh
  - -1: mã lỗi khác
  - <> -1: mã trả về khi thực hiện lệnh *string*

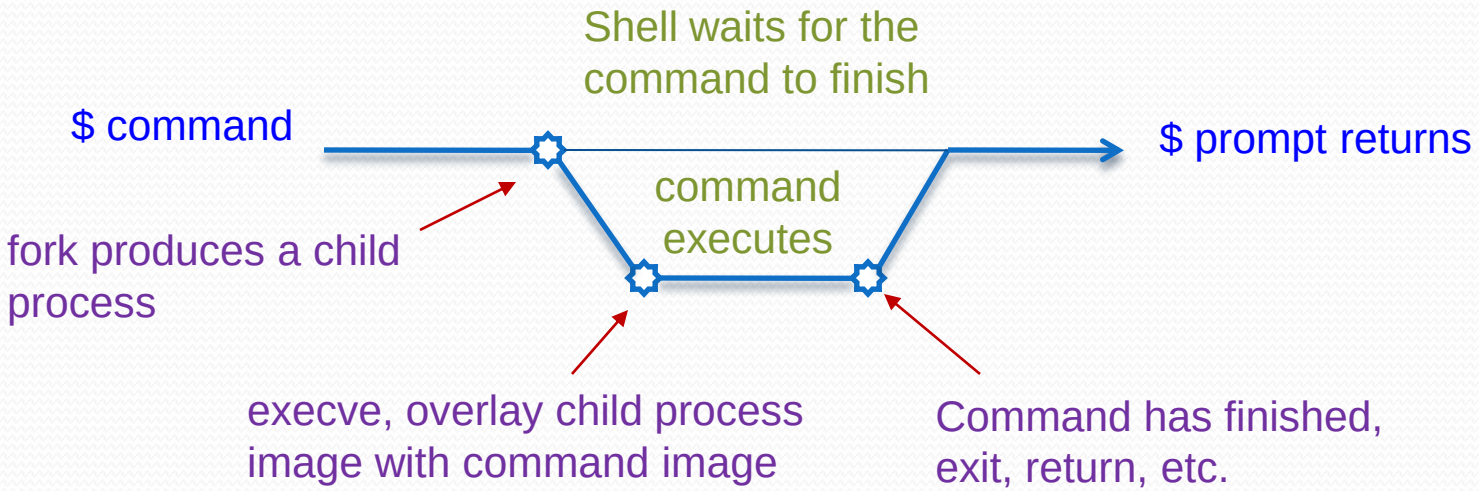
Hàm system()

- Ví dụ

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv) {
   if(!system("ls -l /"))
      printf("Call system() OK!\n");
   return 0;
}
```

Các hàm exec…()

- Các hàm exec…() thực hiện như sau:



Các hàm exec…()

- Chú ý
  - Các hàm exec sẽ thay thế process gọi hàm bằng chương trình tương ứng trong tham số nhập của hàm. Vùng text, data, stack bị thay thế, vùng user (user area) không bị thay thế.
  - Chương trình được gọi bắt đầu thực thi ở hàm main() (entry point), có thể nhận tham số nhập thông qua các tham số truyền vào các hàm exec.
- Ví dụ
  - Gọi thực thi lệnh ở dấu nhắc

    ```
    $ ls -l /
    ```

  - Chương trình dùng exec…()

    ```
    execl( “/bin/ls”, “/bin/ls”, “-l”, “/”, (char*)0);
    ```

Phân tích tên gọi các hàm exec...()

- int execl(const char *path, const char *arg, ...)
- int execv(const char *path, const char *argv[ ])

- int execlp(const char *lename, const char *arg, ...)
- int execvp(const char *lename, const char *argv[ ])

- int execle(const char *path, const char *arg, ..., const char **env)
- int execve(const char *path, const char *argv[ ], const char **env)

Phân tích tên gọi các hàm exec…()

- Hàm có chứa kí tự **l** (execl, execlp, execle) dùng danh sách tham số là các pointer đến string, kết thúc danh sách này là một NULL pointer.

- Hàm có chứa kí tự **v** (execv, execvp, execve) dùng danh sách tham số là một array các pointer đến string, kết thúc là một NULL pointer.

Phân tích tên gọi các hàm exec…()

- Hàm có chứa kí tự **p** (execlp, execvp) chấp nhận tên của chương trình cần chạy, khi chạy chương trình đó, nó sẽ tìm trong execution path hiện tại; các hàm không có kí tự p phải cung cấp đầy đủ đường dẫn đến chương trình cần chạy.

- Hàm có chứa kí tự **e** (execle, execve) có thêm một thông số khác là biến môi trường. Thông số này phải là một pointer trỏ đến các string, và string cuối cùng của pointer này là NULL pointer. Mỗi string mà pointer này trỏ đến có dạng "VARIABLE=value".

Tóm tắt tên các hàm exec…()

Tên hàm	Tham số	Tự động tìm biến môi trường	Tự động tìm đường dẫn
execl(…)	list	yes	no
execv(…)	array	yes	no
execle(…)	list	no	no
execve(…)	array	no	no
execlp(…)	list	yes	yes
execvp(…)	array	yes	yes

Ví dụ 1

```
#include <stdio.h>
#include <unistd.h>
void main() {
  execlp("/sbin/ifconfig","/sbin/ifconfig","-a",0);
}
```

- Dịch và chạy

```
$ gcc -o ex1 ex1.c
$ ./ex1
eth0 Link encap:Ethernet Hwaddr 00:60:8C:84:E6:0E
......
```

Ví dụ 2

```
#include <stdio.h>
#include <unistd.h>
void main(int argc, char *argv[]){
  execv("/sbin/ifconfig",argv);
}
```

- Dịch và chạy

```
$gcc -o ex2 ex2.c
$./ex2 -a
eth0 Link encap:Ethernet Hwaddr 00:60:8C:84:E6:0E
...
```

Ví dụ 3

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
extern char **environ;
int main(int argc, char **argv) {
   int pid, status;
   char *child_argv[4];
   if (argc==1)
     return 1;
   pid=fork();
```

Ví dụ 3 (tt)

```
if (pid==-1)
  return -1;
if (pid==0) {
  child_argv[0]= “sh” ;
  child_argv[1]= “-c” ;
  child_argv[2]=argv[1];
  child_argv[3]=0;
  execve( “/bin/sh” , child_argv, environ);
  exit(127);
}
```

Ví dụ 3 (tt)

```
    do {
        if (waitpid(pid, &status, 0)==-1) {
            if (errno!=EINTR) return -1;
        } else
            return status;
    } while(1);
    return 0;
}
```

Questions???