Chương 6. Tiến trình

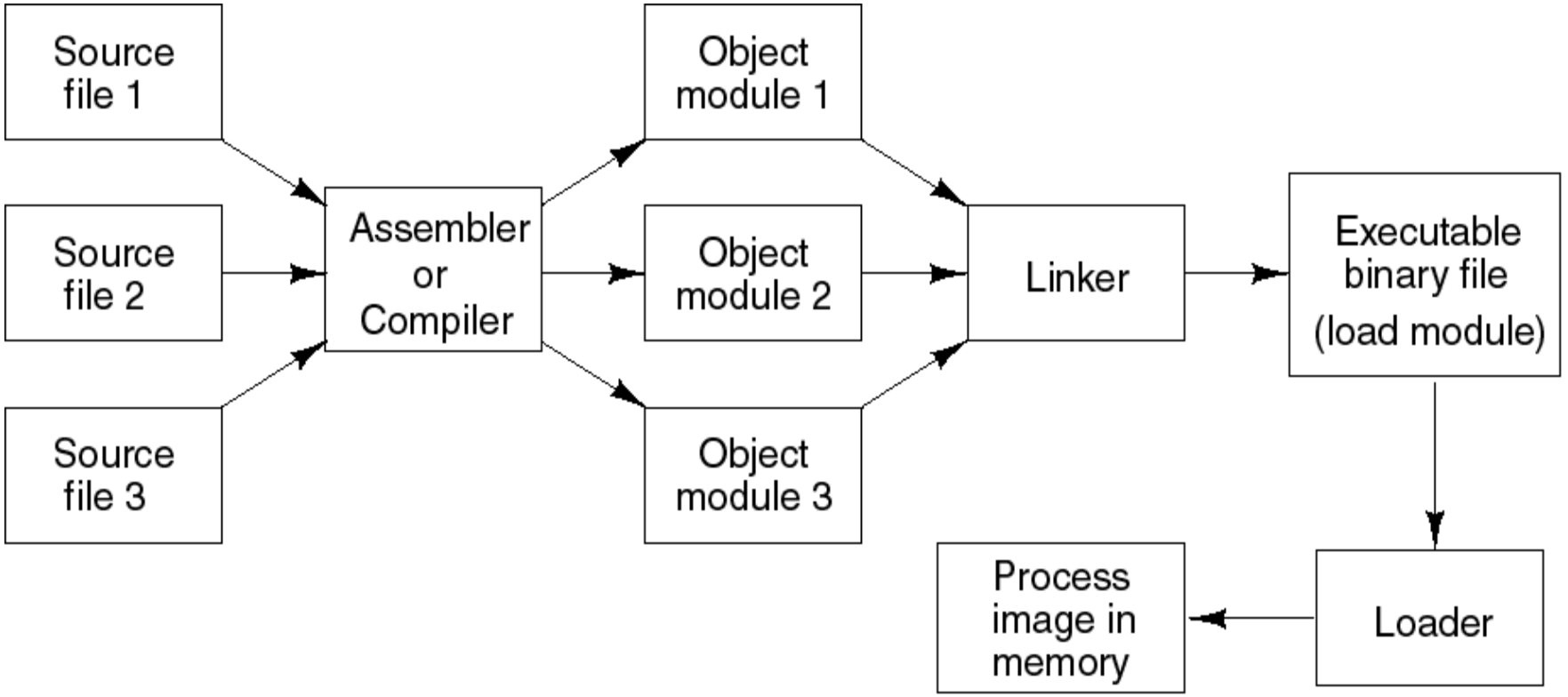
1. Khái niệm cơ bản
2. Định thời CPU
3. Các tác vụ cơ bản: T*ạ*o/Kết thúc tiến trình
4. Sự cộng tác giữa các tiến trình
5. Giao tiếp giữa các tiến trình

1. Khái niệm cơ bản

- Hệ thống máy tính thực thi nhiều chương trình khác nhau
  - Batch system: jobs
  - Time-shared systems: user programs, tasks
  - Job ≈ process
- *tiến trình* (process)
  - một chương trình đang thực thi

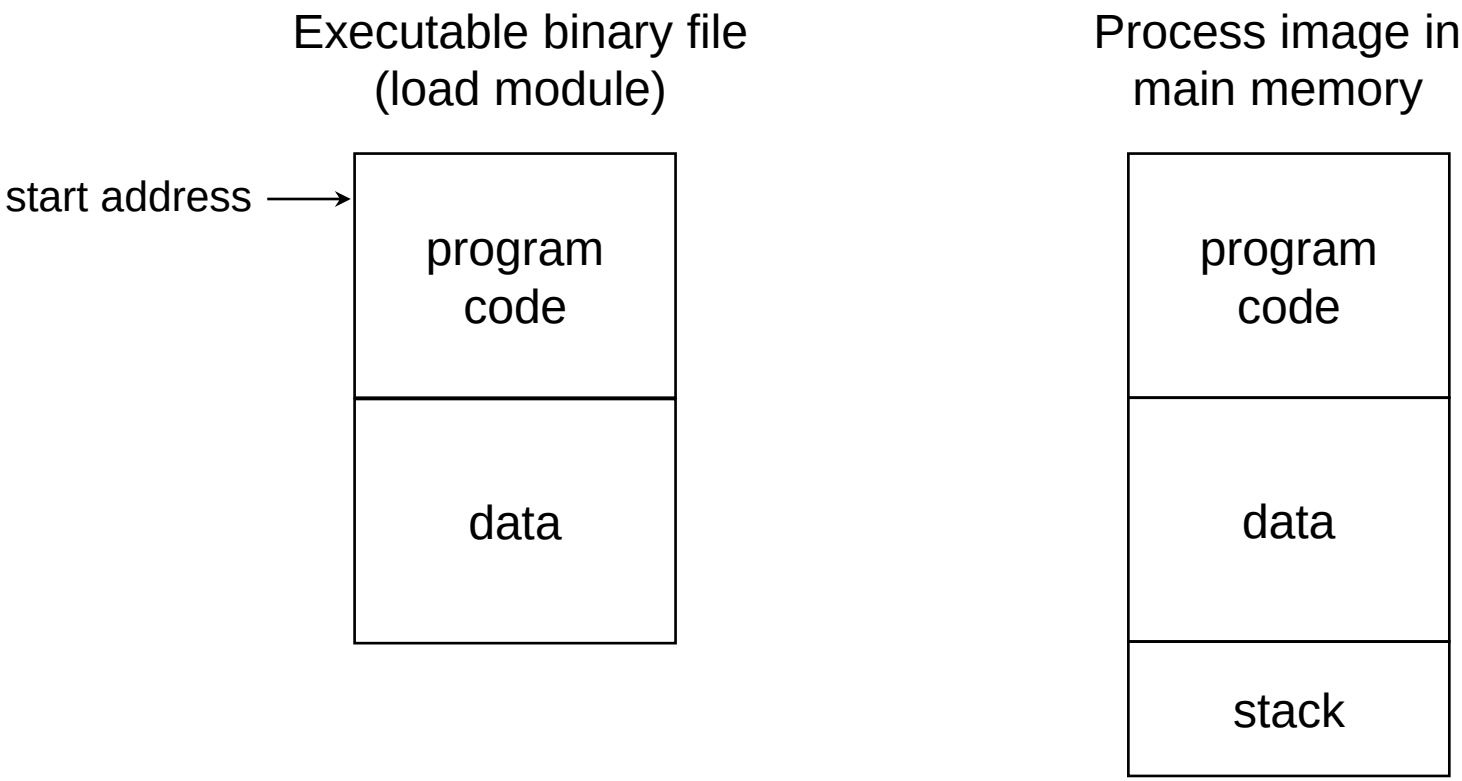
  Một tiến trình bao gồm
  - *Text section* (program code), *data section* (chứa global variables)
  - Hoạt động hiện thời: program counter (PC), process status word (PSW), stack pointer (SP), memory management registers

1.1. Các bước nạp chương trình vào bộ nhớ



1.2. Từ chương trình đến tiến trình

- Dùng *load module* để biểu diễn chương trình thực thi được
- Layout luận lý của *process image*



1.3. Khởi tạo tiến trình

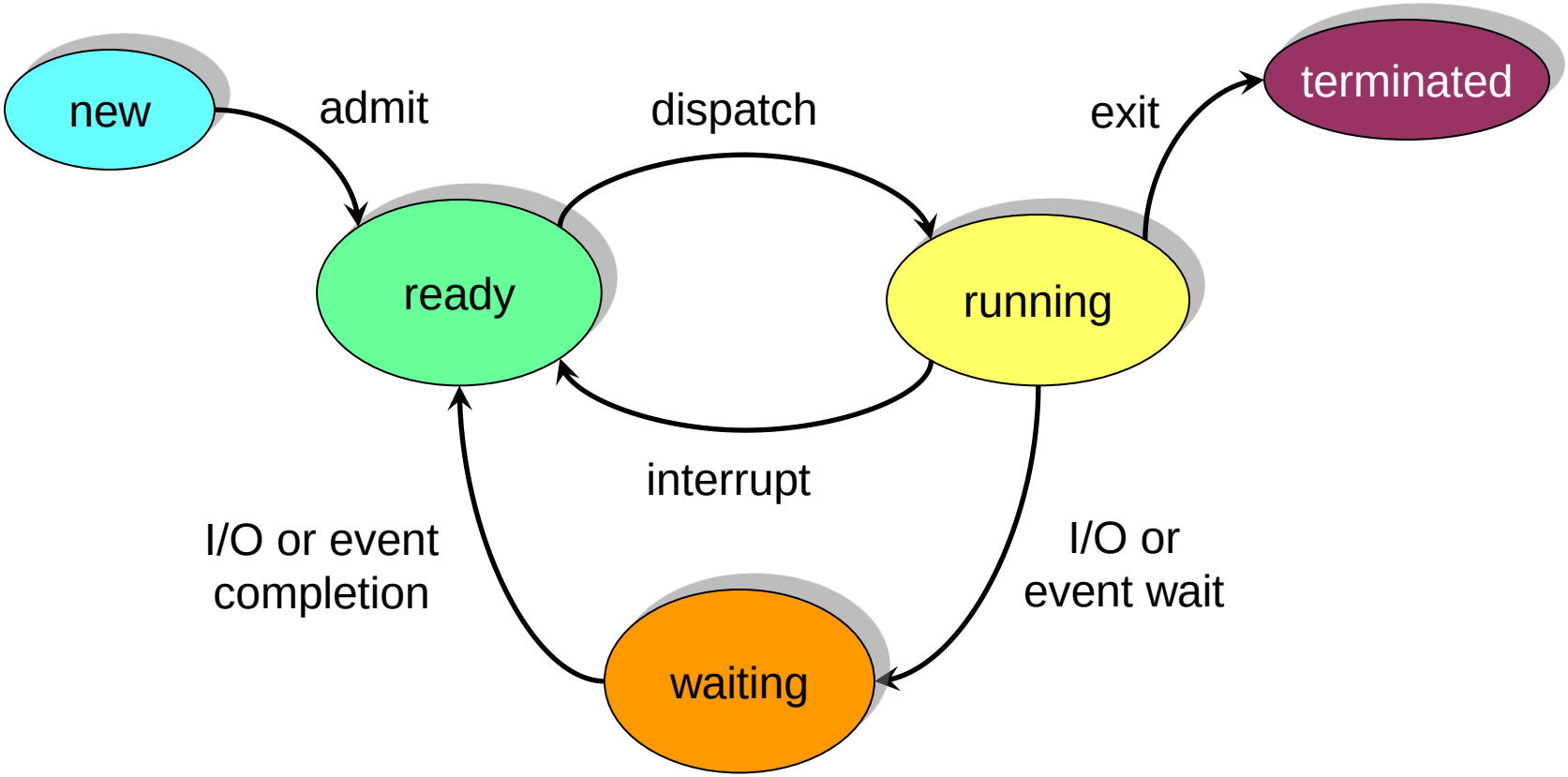
- Các bước hệ điều hành khởi tạo tiến trình
  - Cấp phát một *định danh* duy nhất (process number hay process identifier, pid) cho tiến trình
  - Cấp phát không gian nhớ để nạp tiến trình
  - Khởi tạo khối dữ liệu *Process Control Block* (PCB) cho tiến trình
    - PCB là nơi hệ điều hành lưu các thông tin về tiến trình
  - Thiết lập các mối liên hệ cần thiết (vd: sắp PCB vào hàng đợi định thời,...)

1.4. Các trạng thái của tiến trình

- Các *trạng thái của tiến trình* (process states):
  - *new*: tiến trình vừa được tạo
  - *ready*: tiến trình đã có đủ tài nguyên, chỉ còn cần CPU
  - *running*: các lệnh của tiến trình đang được thực thi
  - *waiting*: hay là *blocked*, tiến trình đợi I/O hoàn tất, tín hiệu.
  - *terminated*: tiến trình đã kết thúc.

1.5. Các trạng thái của tiến trình (tt)

- Chuyển đổi giữa các trạng thái của tiến trình



Ví dụ về trạng thái tiến trình

```
/* test.c */
int main(int argc, char** argv)
{
   printf("Hello world\n");
   exit(0);
}
```

Biên dịch chương trình trong Linux

gcc test.c –o test

Thực thi chương trình test

./test

Trong hệ thống sẽ có một tiến trình *test* được tạo ra, thực thi và kết thúc.

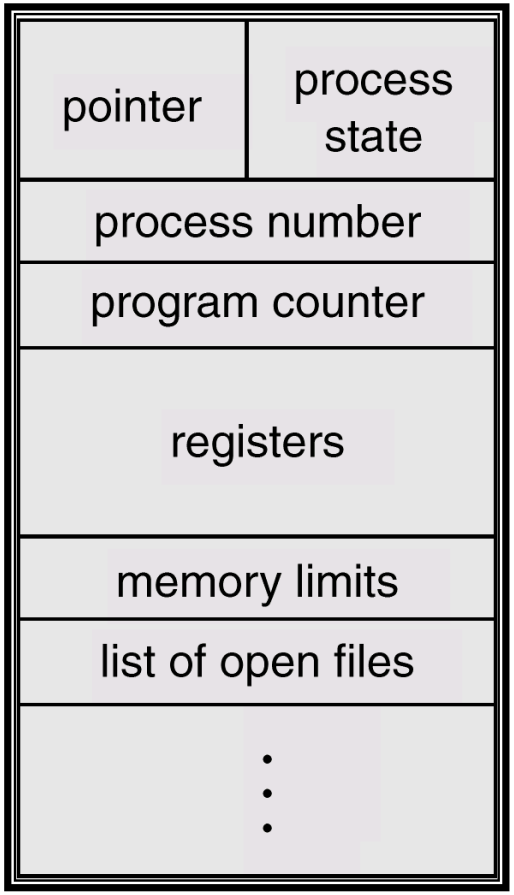
- Chuỗi trạng thái của tiến trình test như sau (trường hợp tốt nhất):
  - new
  - ready
  - running
  - waiting (do chờ I/O khi gọi printf)
  - ready
  - running
  - terminated

1.6. Process control block

- Đã thấy là mỗi tiến trình trong hệ thống đều được cấp phát một *Process Control Block* (PCB)
- PCB là một trong các cấu trúc dữ liệu

  quan trọng nhất của hệ điều hành

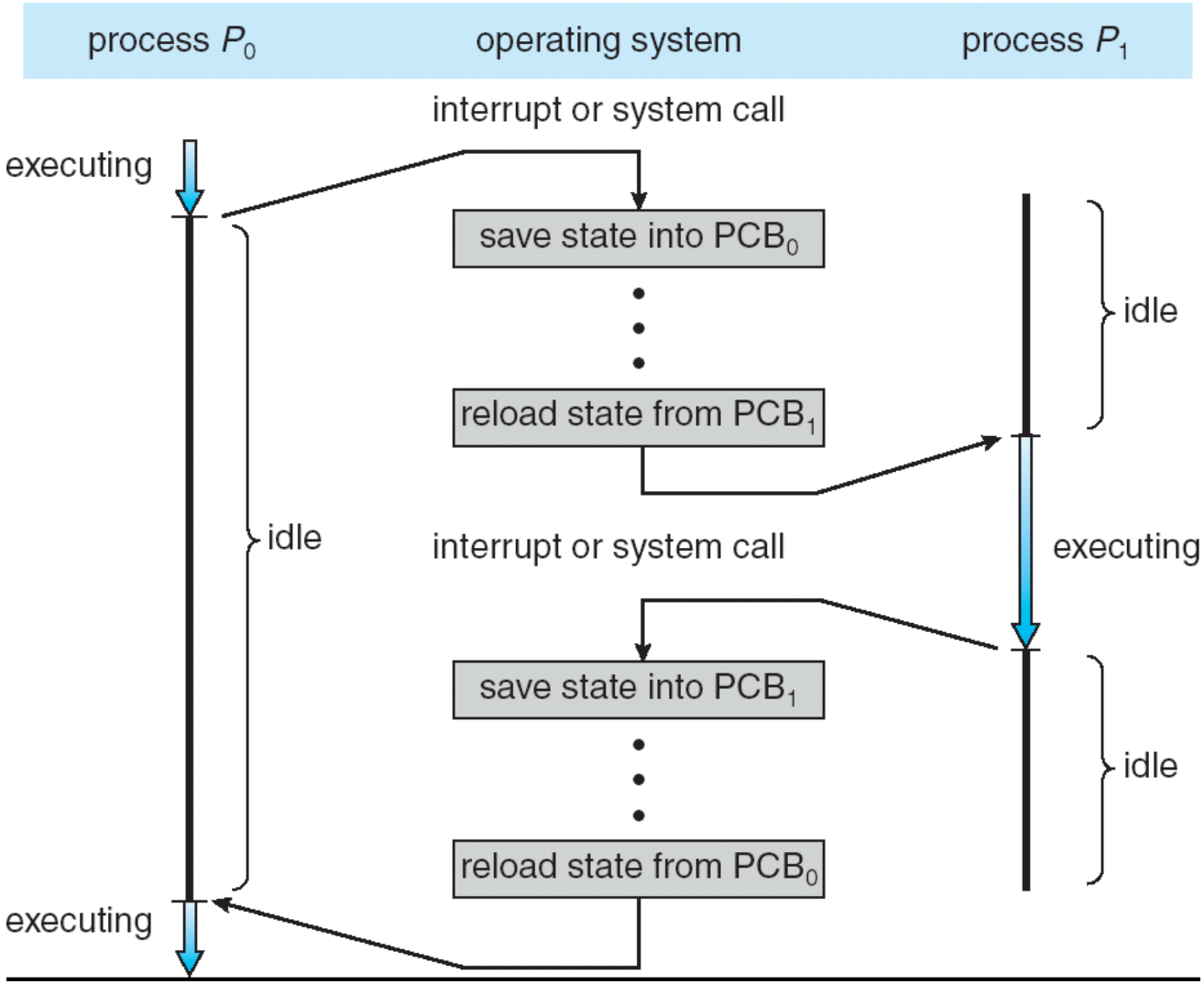
Ví dụ layout của một PCB:
(trường pointer dùng để liên kết các PCBs thành một linked list)



1.7. Chuyển ngữ cảnh (context switch)

- *Ngữ cảnh* (context) của một tiến trình là trạng thái của tiến trình
- Ngữ cảnh của tiến trình được biểu diễn trong PCB của nó
- *Chuyển ngữ cảnh* (context switch) là công việc giao CPU cho tiến trình khác. Khi đó cần:
  - lưu ngữ cảnh của tiến trình cũ vào PCB của nó
  - nạp ngữ cảnh từ PCB của tiến trình mới để tiến trình mới thực thi

Chuyển ngữ cảnh (tt)

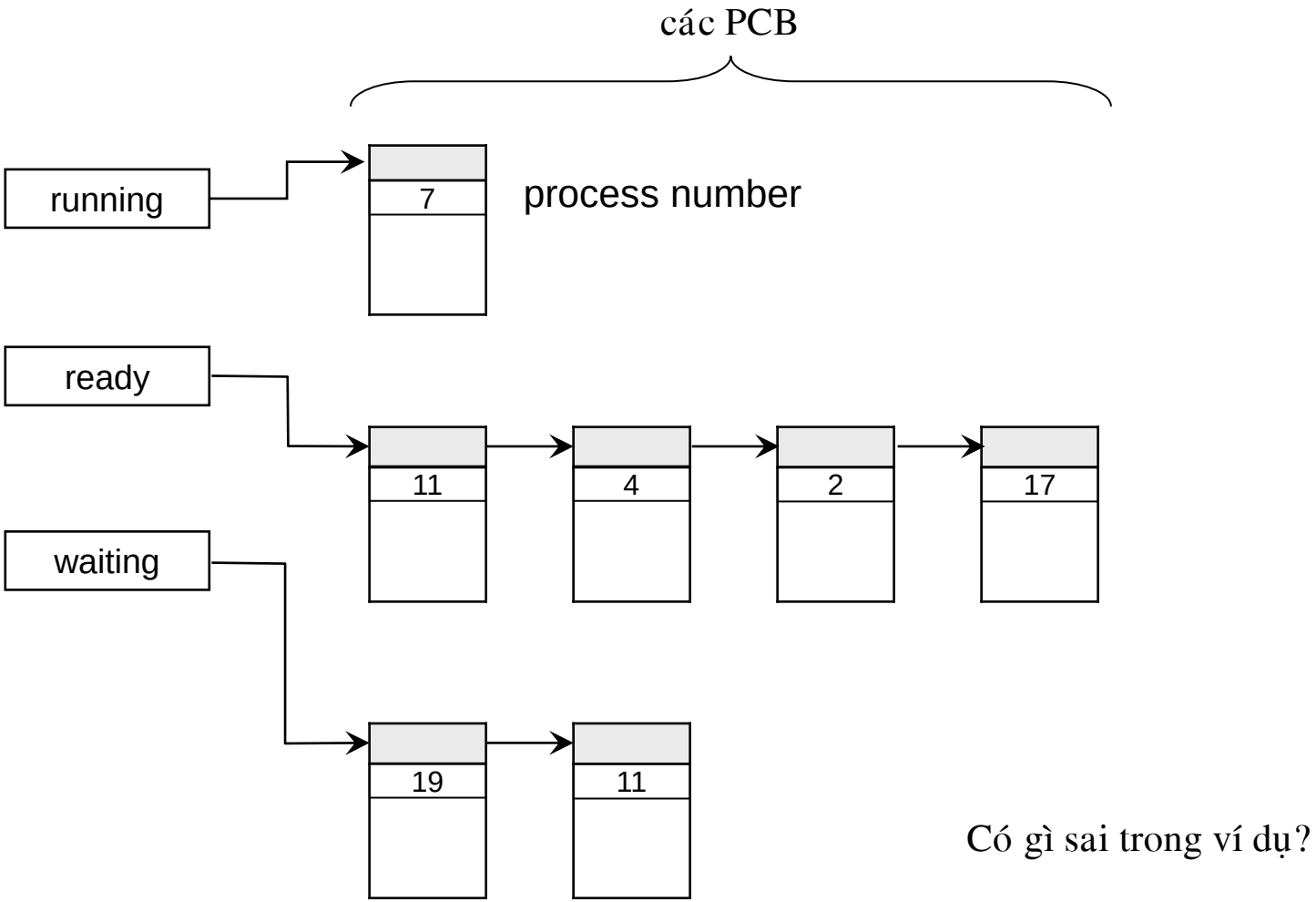


Yêu cầu đối với hệ điều hành về quản lý tiến trình

- Hỗ trợ sự thực thi luân phiên giữa nhiều tiến trình
  - Hiệu suất sử dụng CPU
  - Thời gian đáp ứng
- Phân phối tài nguyên hệ thống hợp lý
  - tránh deadlock, trì hoãn vô hạn định,...
- Cung cấp cơ chế giao tiếp và đồng bộ hoạt động các tiến trình
- Cung cấp cơ chế hỗ trợ user tạo/kết thúc tiến trình

Quản lý các tiến trình: các hàng đợi

- Ví dụ

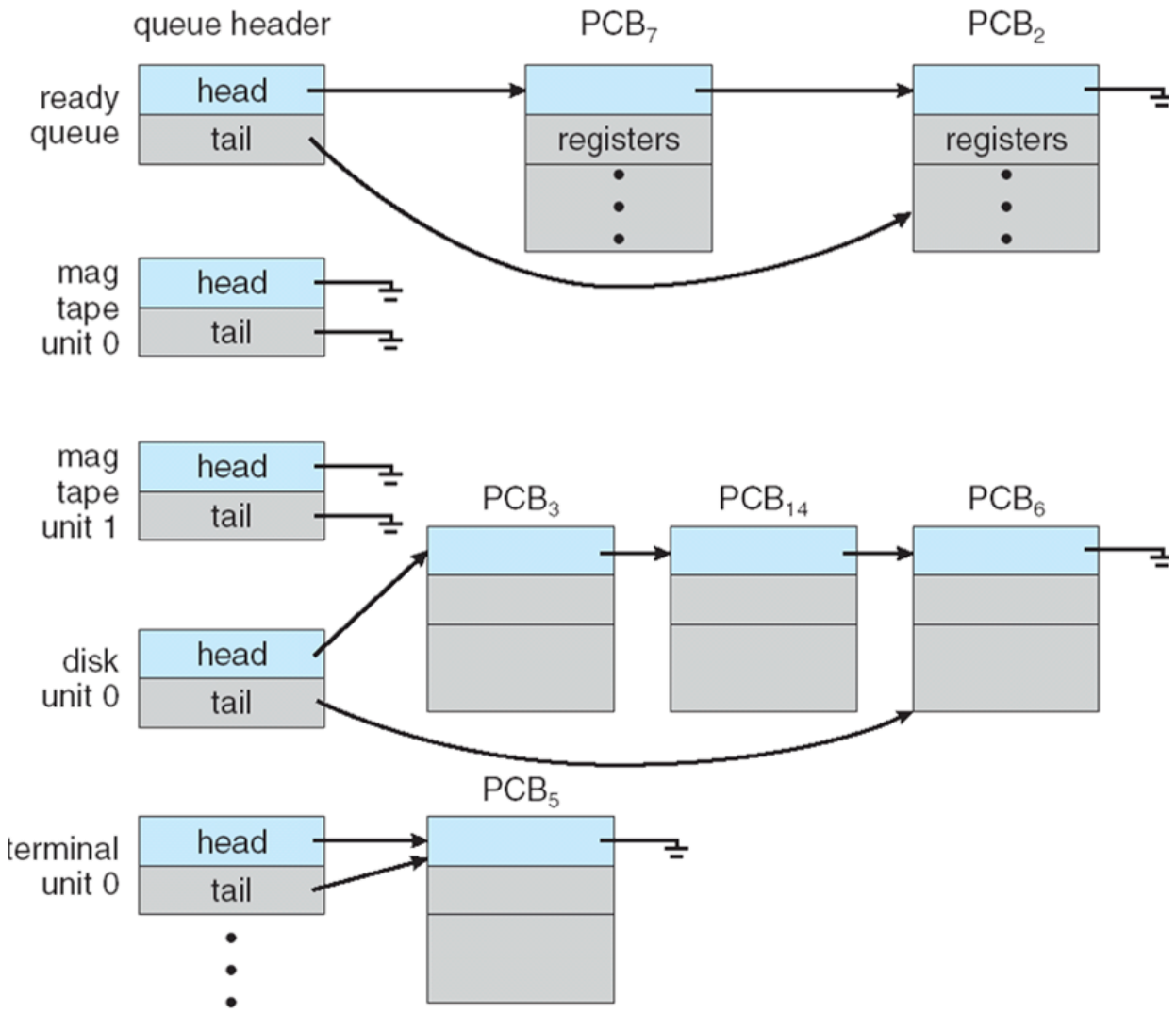


2. Định thời tiến trình

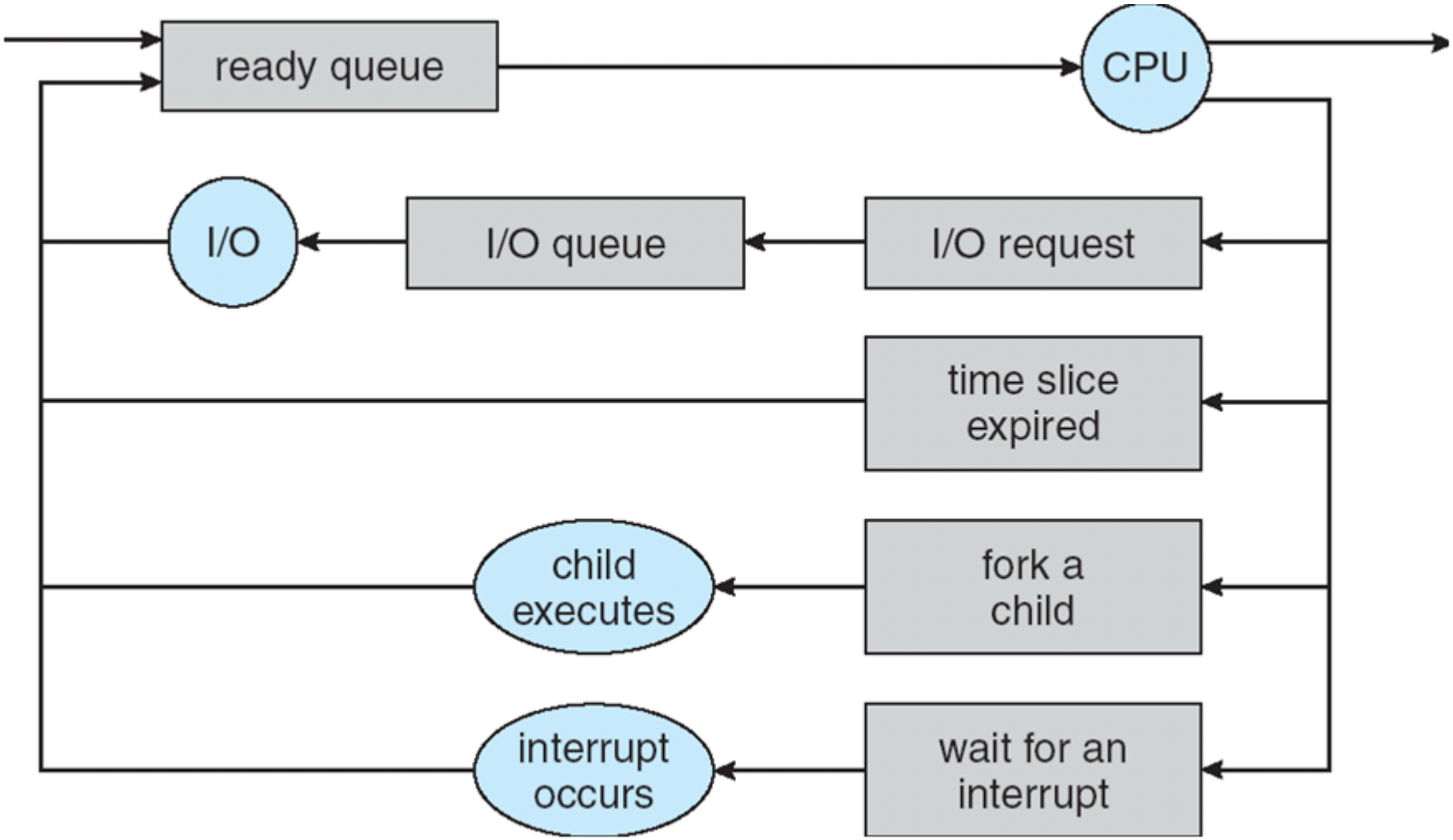
- Tại sao phải định thời?
  - Multiprogramming
    - Có nhiều tiến trình phải thực thi luân phiên nhau
    - Mục tiêu: cực đại hiệu suất sử dụng của CPU
  - Time-sharing
    - Cho phép users tương tác với tiến trình đang thực thi
    - Mục tiêu: tối thiểu thời gian đáp ứng
- Một số khái niệm cơ bản
  - Các *bộ định thời* (scheduler)
  - Các *hàng đợi định thời* (scheduling queue)

Các hàng đợi định thời

- Job queue
  - Set of all processes in the system
- Ready queue
  - Set of all processes residing in main memory, ready and waiting to execute
- Device queues
  - Set of processes waiting for an I/O device
- ...

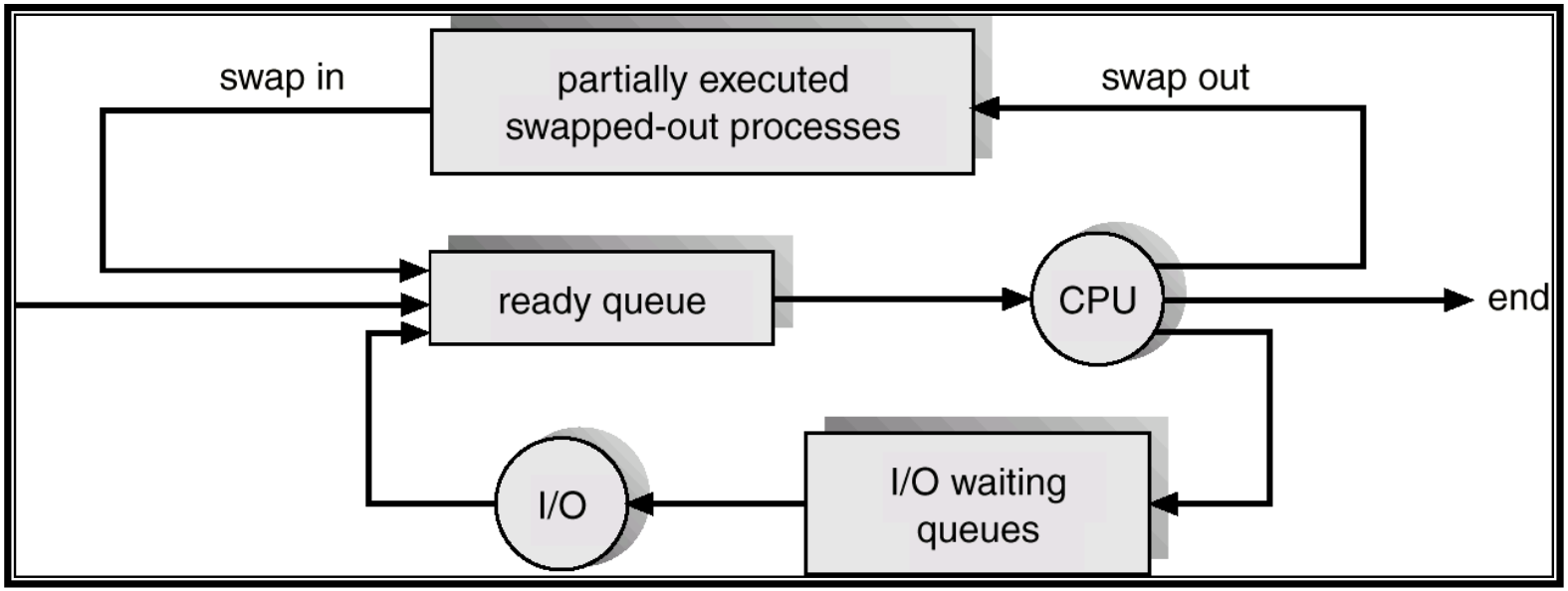


Representation of Process Scheduling



Thêm medium-term scheduling

- Đôi khi hệ điều hành (như time-sharing system) có thêm medium-term scheduling để điều chỉnh mức độ multiprogramming của hệ thống
- *Medium-term scheduler*
  - chuyển tiến trình từ bộ nhớ sang đĩa (swap out)
  - chuyển tiến trình từ đĩa vào bộ nhớ (swap in)



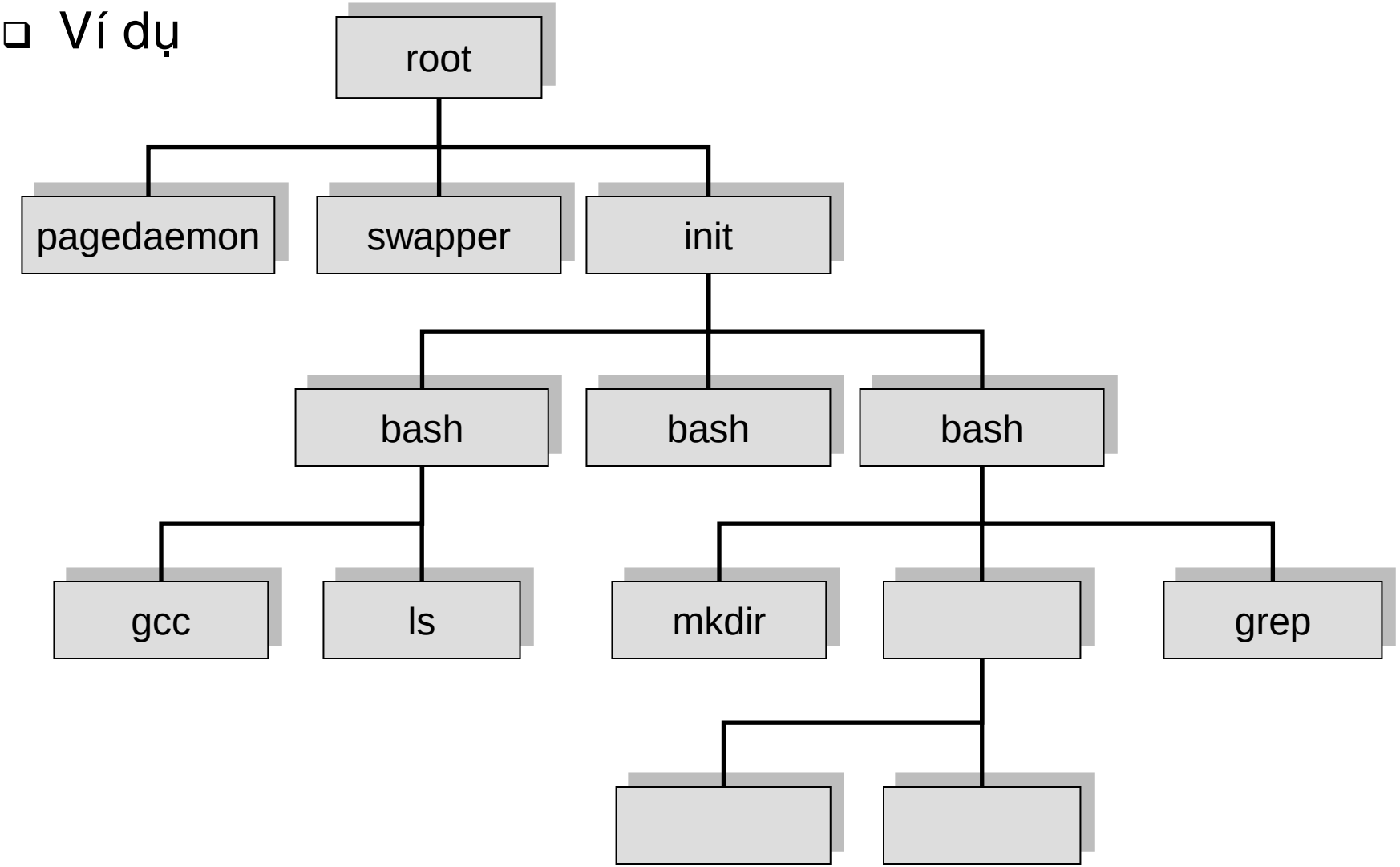
Các tác vụ đối với tiến trình

- Tạo tiến trình mới (process creation)
  - Một tiến trình có thể tạo tiến trình mới thông qua một system call (vd: hàm fork trong Unix)
    - Ví dụ: (Unix) Khi user đăng nhập hệ thống, một command interpreter (shell) sẽ được tạo ra cho user

  tiến trình được tạo là tiến trình *con* của tiến trình tạo (tiến trình *cha*). Quan hệ cha-con định nghĩa một *cây tiến trình.*

Cây tiến trình trong Linux/Unix

- Ví dụ

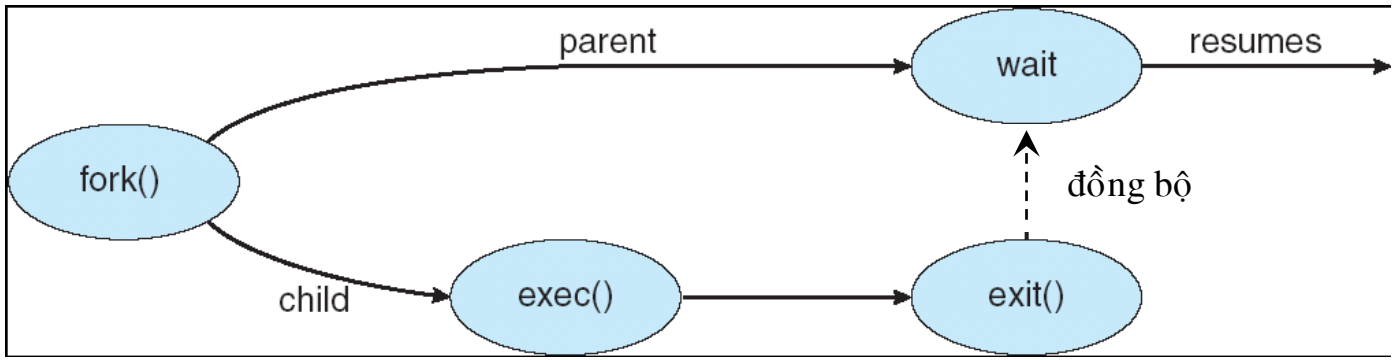


Các tác vụ đối với tiến trình

- Tạo tiến trình mới
  - Chia sẻ tài nguyên của tiến trình cha
    - tiến trình cha và con chia sẻ mọi tài nguyên
    - tiến trình con chia sẻ một phần tài nguyên của cha
  - Trình tự thực thi
    - tiến trình cha và con thực thi đồng thời (concurrently)
    - tiến trình cha đợi đến khi các tiến trình con kết thúc.

Về quan hệ cha/con

- Không gian địa chỉ (address space)
  - Không gian địa chỉ của tiến trình con được nhân bản từ cha
  - Không gian địa chỉ của tiến trình con được khởi tạo từ template.
- Ví dụ trong UNIX/Linux
  - System call fork() tạo một tiến trình mới
  - System call exec() dùng sau fork() để nạp một chương trình mới vào không gian nhớ của tiến trình mới



Ví dụ tạo process với fork()

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[]){
   int  pid;
   /* create a new process */
   pid = fork();

   if (pid > 0){
      printf("This is parent process");
      wait(NULL);
      exit(0);
   }
   else if (pid == 0)
   {
      printf("This is child process");
      execlp("/bin/ls", "ls", NULL);
      exit(0);
   }
   else {
        printf("Fork error\n");
        exit(-1);
   }
}
```

Các tác vụ đối với tiến trình (tt)

- Tạo tiến trình mới ✓
- Kết thúc tiến trình
  - tiến trình tự kết thúc
    - tiến trình kết thúc khi thực thi lệnh cuối và gọi system routine exit
  - tiến trình kết thúc do tiến trình khác (có đủ quyền, vd: tiến trình cha của nó)
    - Gọi system routine abort với tham số là pid (process identifier) của tiến trình cần được kết thúc
  - Hệ điều hành thu hồi tất cả các tài nguyên của tiến trình kết thúc (vùng nhớ, I/O buffer,...)

Cộng tác giữa các tiến trình

- Trong tiến trình thực thi, các tiến trình có thể *cộng tác* (cooperate) để hoàn thành công việc

- Các tiến trình cộng tác để
  - Chia sẻ dữ liệu (information sharing)
  - Tăng tốc tính toán (computational speedup)
    - Nếu hệ thống có nhiều CPU, chia công việc tính toán thành nhiều công việc tính toán nhỏ chạy song song
  - Thực hiện một công việc chung
    - Xây dựng một phần mềm phức tạp bằng cách chia thành các module/process hợp tác nhau

- Sự cộng tác giữa các tiến trình yêu cầu hệ điều hành hỗ trợ cơ chế giao tiếp và cơ chế đồng bộ hoạt động của các tiến trình

Bài toán producer-consumer

- Ví dụ cộng tác giữa các tiến trình: *bài toán producer-consumer*
  - *Producer* tạo ra các dữ liệu và *consumer* tiêu thụ, sử dụng các dữ liệu đó. Sự trao đổi thông tin thực hiện qua buffer
    - *unbounded buffer:* kích thước buffer vô hạn (không thực tế).
    - *bounded buffer:* kích thước buffer có hạn.
  - Producer và consumer phải hoạt động đồng bộ vì
    - Consumer không được tiêu thụ khi producer chưa sản xuất
    - Producer không được tạo thêm sản phẩm khi buffer đầy.

Interprocess communication (IPC)

- *IPC* là cơ chế cung cấp bởi hệ điều hành nhằm giúp các tiến trình
  - giao tiếp với nhau
  - và đồng bộ hoạt động

  mà không cần chia sẻ không gian địa chỉ

- IPC có thể được cung cấp bởi message passing system

Message passing system

- Làm thế nào để các tiến trình giao tiếp nhau? Các vấn đề:
  - *Naming*
    - Giao tiếp trực tiếp
      - send(P, msg): gửi thông điệp đến tiến trình P
      - receive(Q, msg): nhận thông điệp đến từ tiến trình Q
    - Giao tiếp gián tiếp: thông qua *mailbox* hay *port*
      - send(A, msg): gửi thông điệp đến mailbox A
      - receive(Q, msg): nhận thông điệp từ mailbox B
  - *Synchronization*: blocking send, nonblocking send, blocking receive, nonblocking receive
  - *Buffering*: dùng queue để tạm chứa các message
    - Zero capacity (no buffering)
    - Bounded capacity: độ dài của queue là giới hạn
    - Unbounded capacity: độ dài của queue là không giới hạn

Mô hình giao tiếp client-server

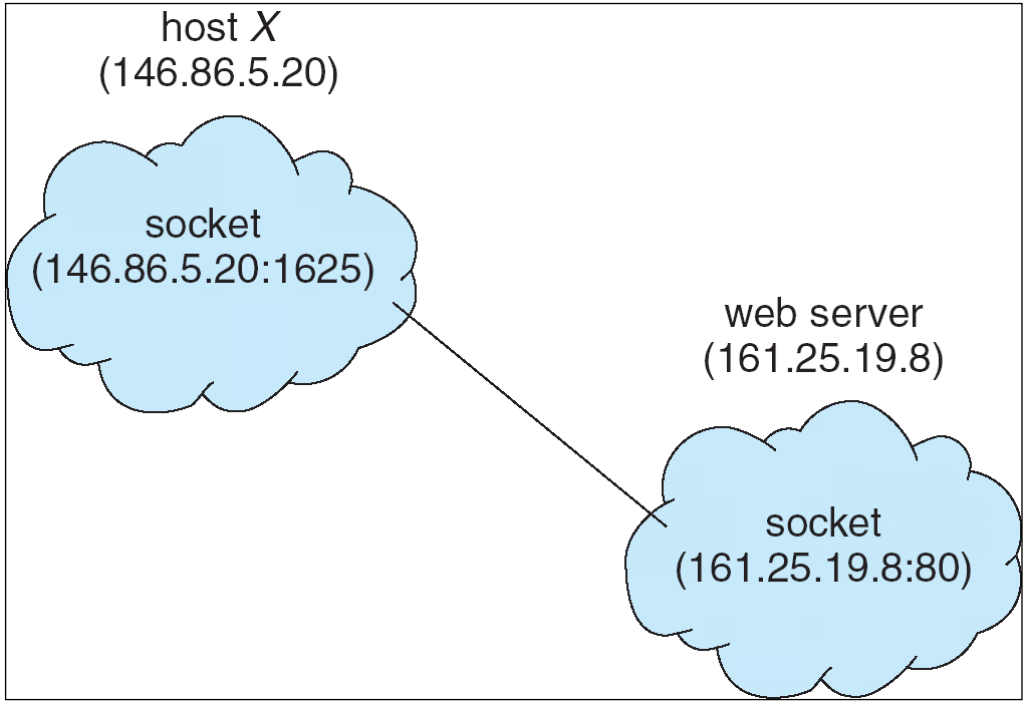
1. Sockets
2. Remote Procedure Calls (RPC)
3. Remote Method Invocation (RMI)

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

Socket

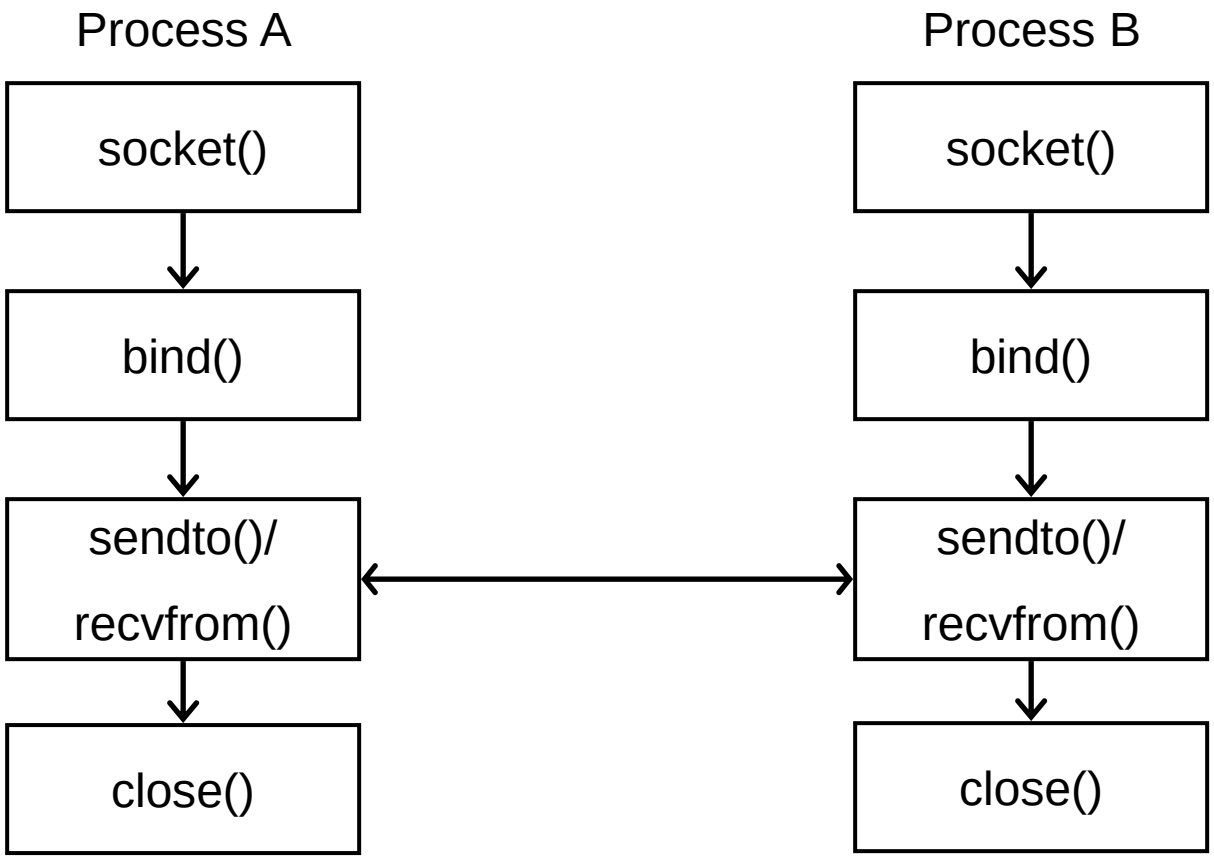
- Cơ chế giao tiếp mức thấp (low-level), gửi nhận một chuỗi byte dữ liệu không cấu trúc (unstructured stream of bytes)
- Giao tiếp qua socket: connectionless và connection-oriented
- Lập trình socket
  - Berkeley socket (BSD socket), WinSock



Cơ chế gửi/nhận qua socket

Hàm thư viện	*Diễn giải*
socket()	Tạo một socket
bind()	Gắn một địa chỉ cục bộ vào một socket
listen()	Xác định độ lớn/kích thước hàng đợi
accept()	(server) chờ kết nối đến từ client
connect()	(client) kết nối đến một server
send()/sendto()	Gửi dữ liệu qua kênh giao tiếp đã thiết lập
recv()/recvfrom()	Nhận dữ liệu qua kênh giao tiếp
close()	Đóng kết nối

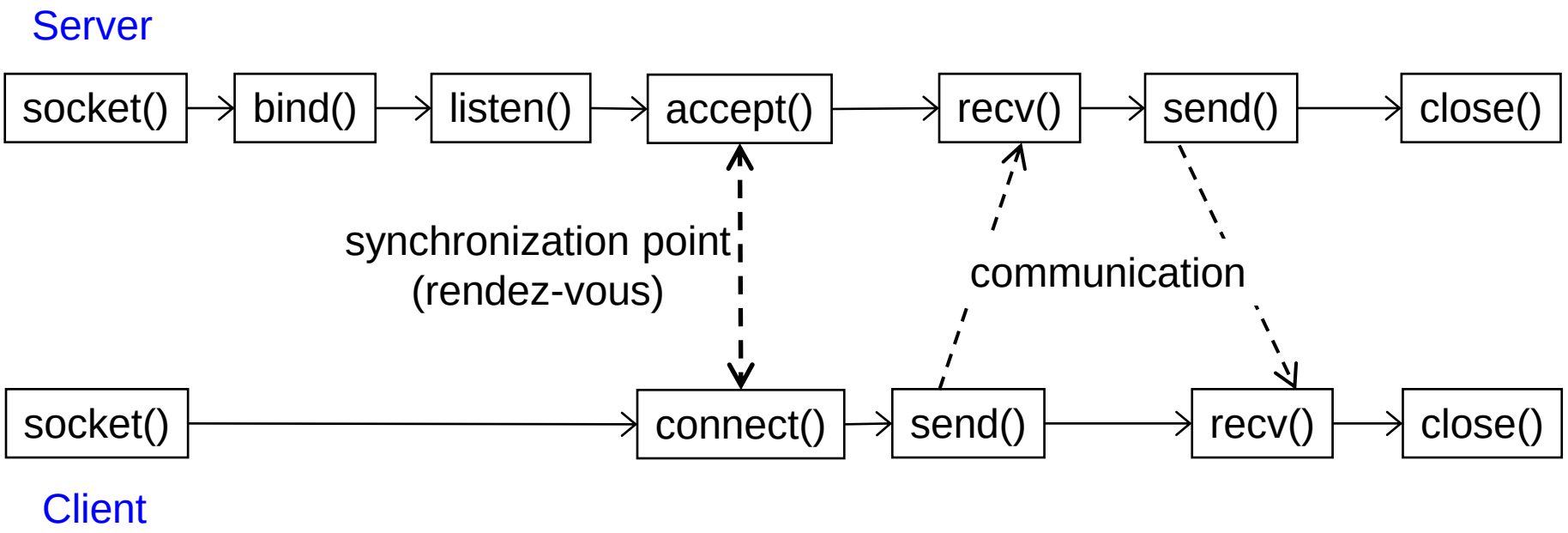
Connectionless socket



» sendto(socket, buffer, buffer_length, flags, destination_address, addr_len)

» recvfrom(socket, buffer, buffer_length, flags, from_address, addr_len)

Connection-oriented socket



» send(socket, buffer, buffer_length, flags)

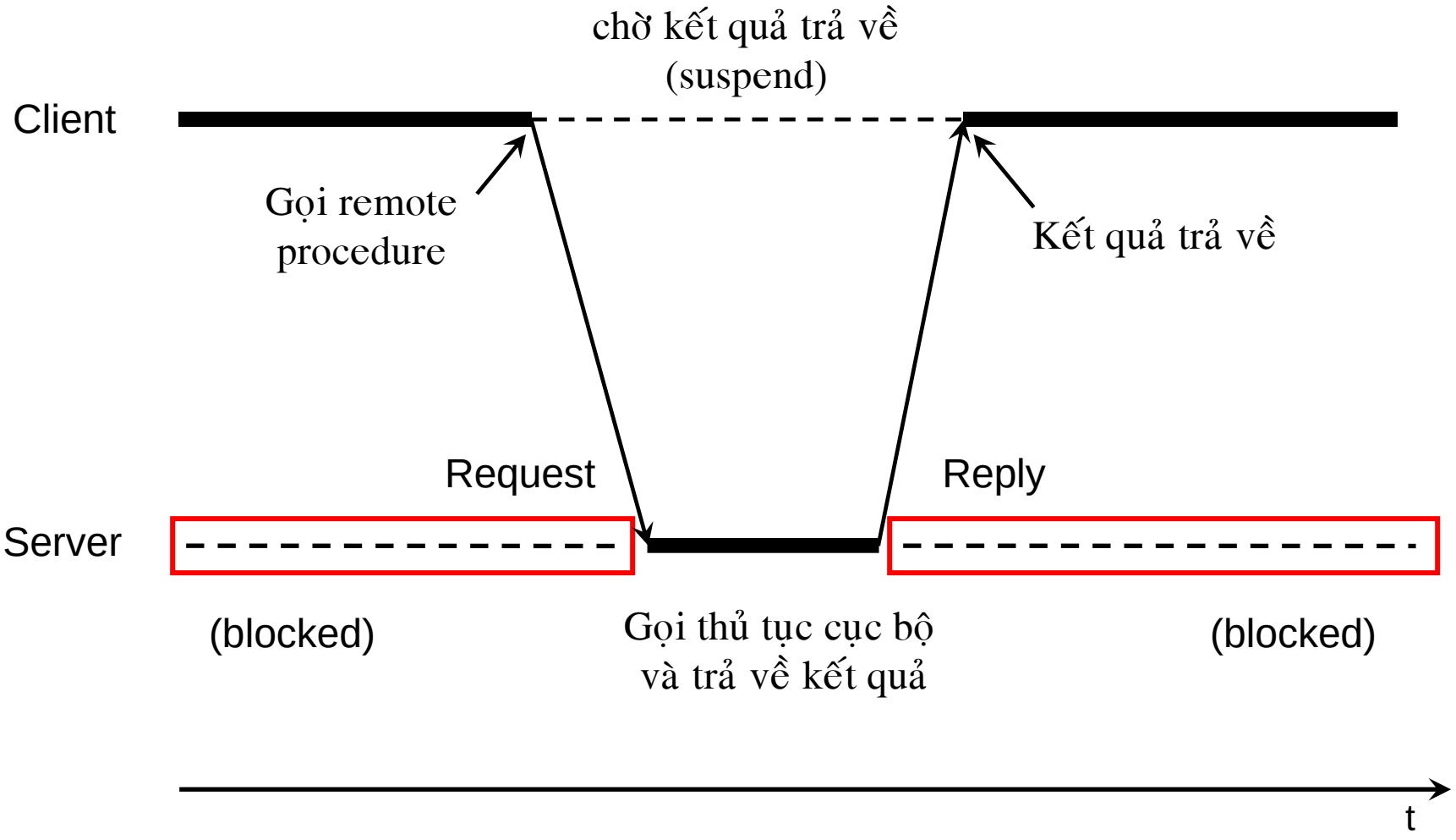
» recv(socket, buffer, buffer_length, flags)

Remote procedure call

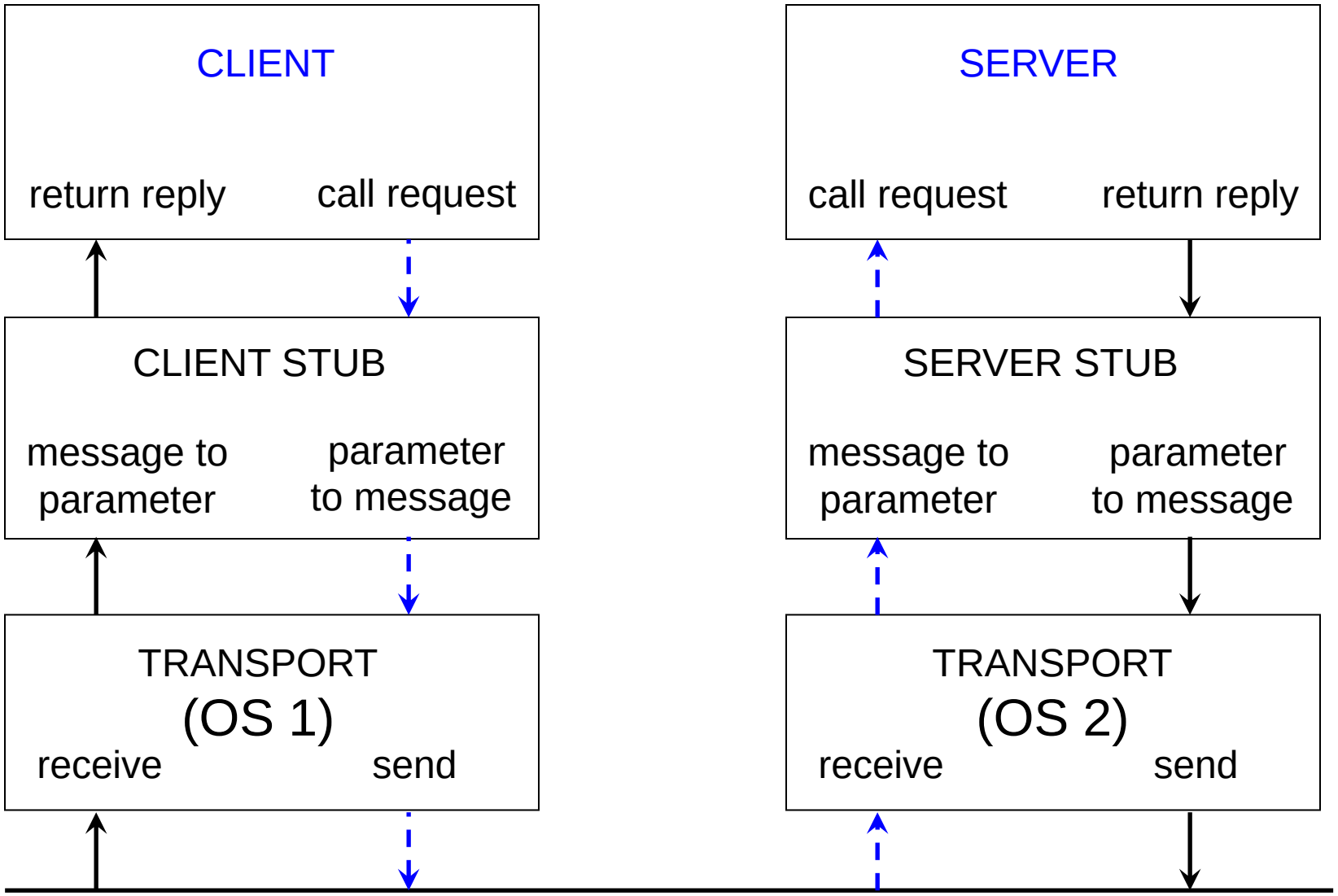
- *Remote procedure call* (RPC)
  - Cho phép một chương trình gọi một thủ tục nằm trên máy tính ở xa qua mạng.

- Các vấn đề khi thực hiện RPC
  - Truyền tham số và kết quả trả về của lời gọi thủ tục
  - Chuyển đổi dữ liệu khi truyền trên mạng (data conversion)
  - Kết nối client đến server
  - Biên dịch chương trình
  - Kiểm soát lỗi
  - Security

Sơ đồ hoạt động của RPC



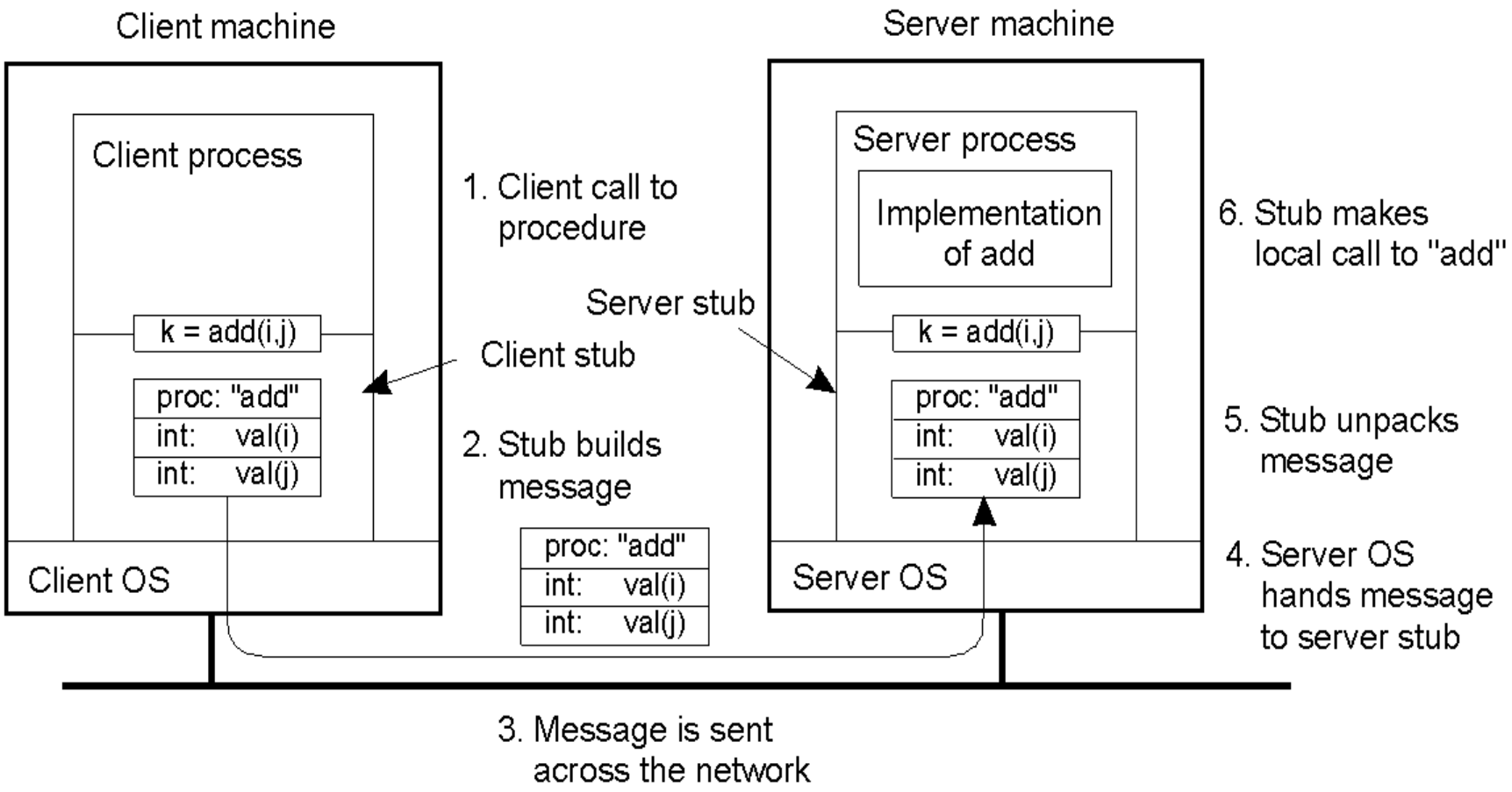
Lưu đồ thực hiện RPC



Truyền tham số trong RPC

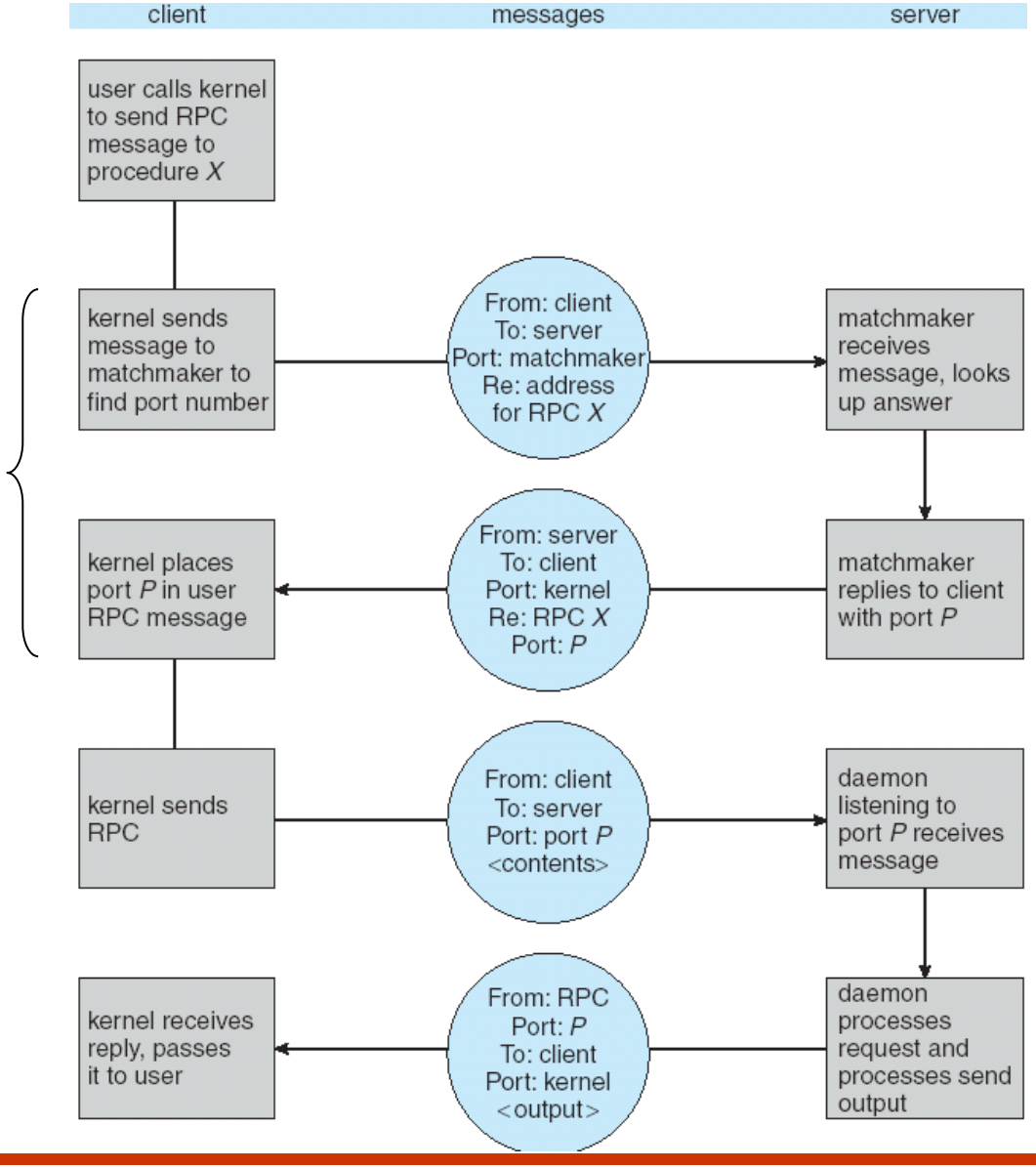
- *Marshalling*
  - qui tắc truyền tham số và chuyển đổi dữ liệu trong RPC bao gồm cả đóng gói dữ liệu thành dạng thức có thể truyền qua mạng máy tính.
- *Biểu diễn dữ liệu và kiểm tra kiểu dữ liệu*
  - Dữ liệu được biểu diễn khác nhau trên các hệ thống khác nhau
    - ASCII, EBCDIC
    - Ví dụ biểu diễn 32-bit integer trong máy:
      - *big-endian* → most significant byte tại high memory address (Motorola)
      - *little-endian* → least significant byte tại high memory address (Intel x86)
    - Dạng biểu diễn XDR (External Data Representation): biểu diễn dữ liệu machine-independent

Truyền tham số trong RPC (tt)



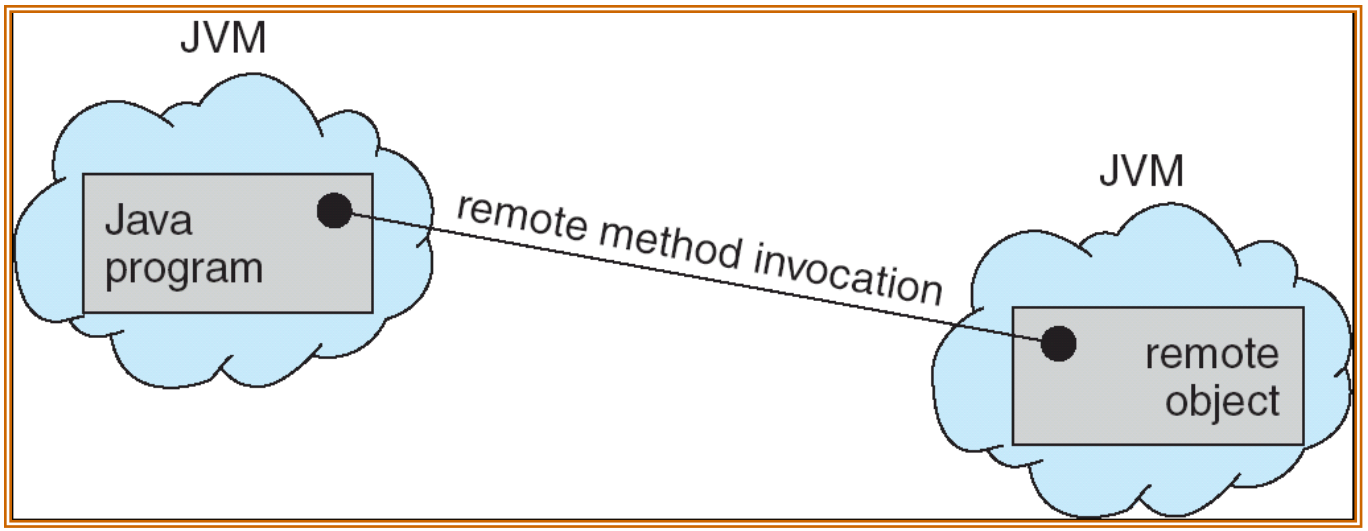
tiến trình thực hiện RPC

Dùng *dynamic binding* để xác định port number của RPC X

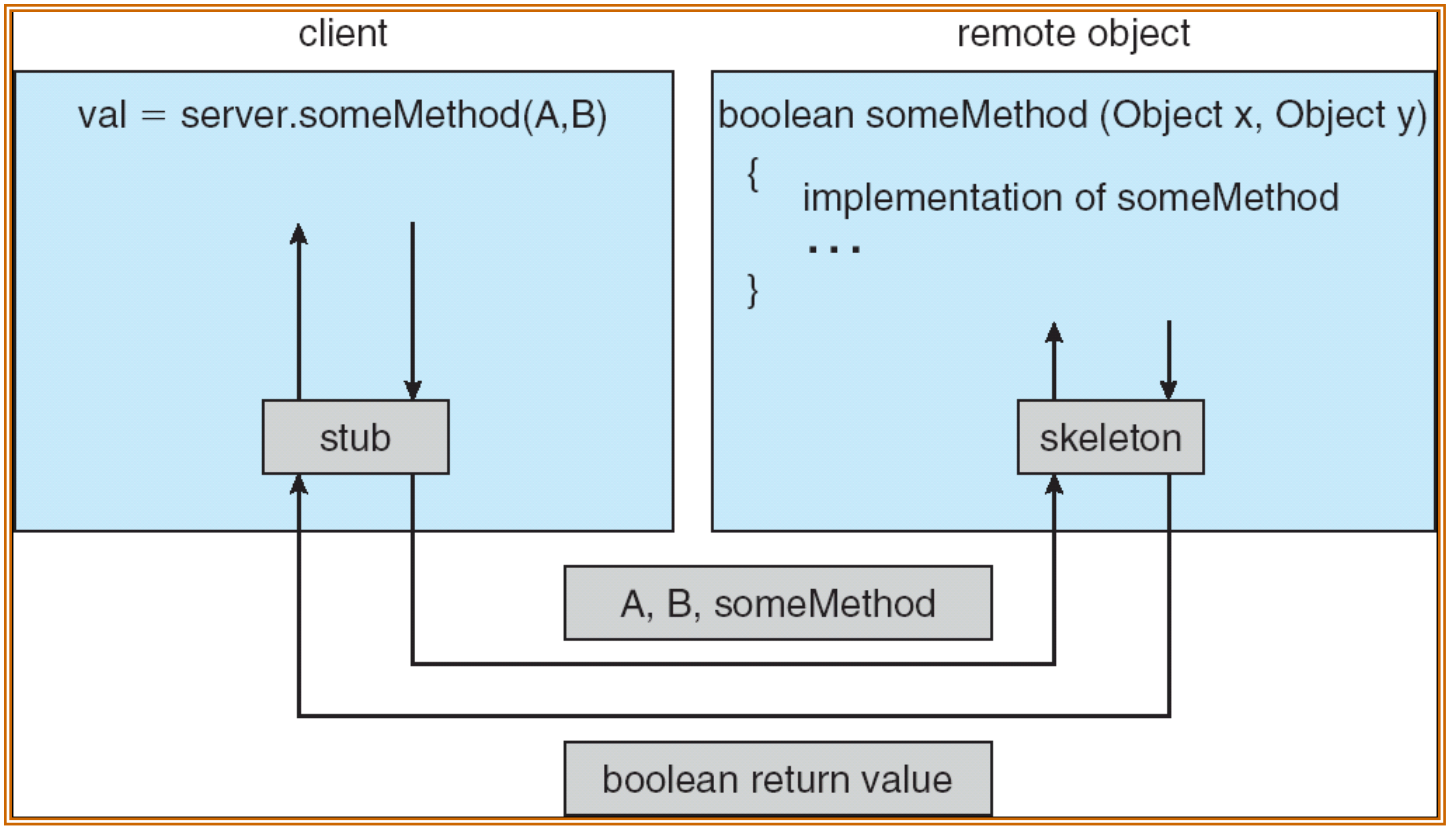


Remote method invocation

- *Remote Method Invocation* (RMI)
  - Cho phép một chương trình Java có thể gọi một phương thức (method) của một *đối tượng ở xa*, nghĩa là một đối tượng ở tại một máy ảo Java khác



Cơ chế marshalling trong RMI



Phương thức được triệu gọi có dạng sau:

boolean someMethod(Object x, Object y)

B. Thread

1. Khái niệm tổng quan
2. Các mô hình multithread
3. Pthreads (POSIX thread)
4. Multithreading trong Solaris 2
5. Multithreading với Java

Xem xét lại khái niệm tiến trình

- Khái niệm tiến trình truyền thống: tiến trình gồm
  - Không gian địa chỉ (text section, data section)
  - Một luồng thực thi duy nhất (single thread of execution)
    - program counter
    - các register
    - stack
  - Các tài nguyên khác (các open file, các tiến trình con,...)

Mở rộng khái niệm tiến trình

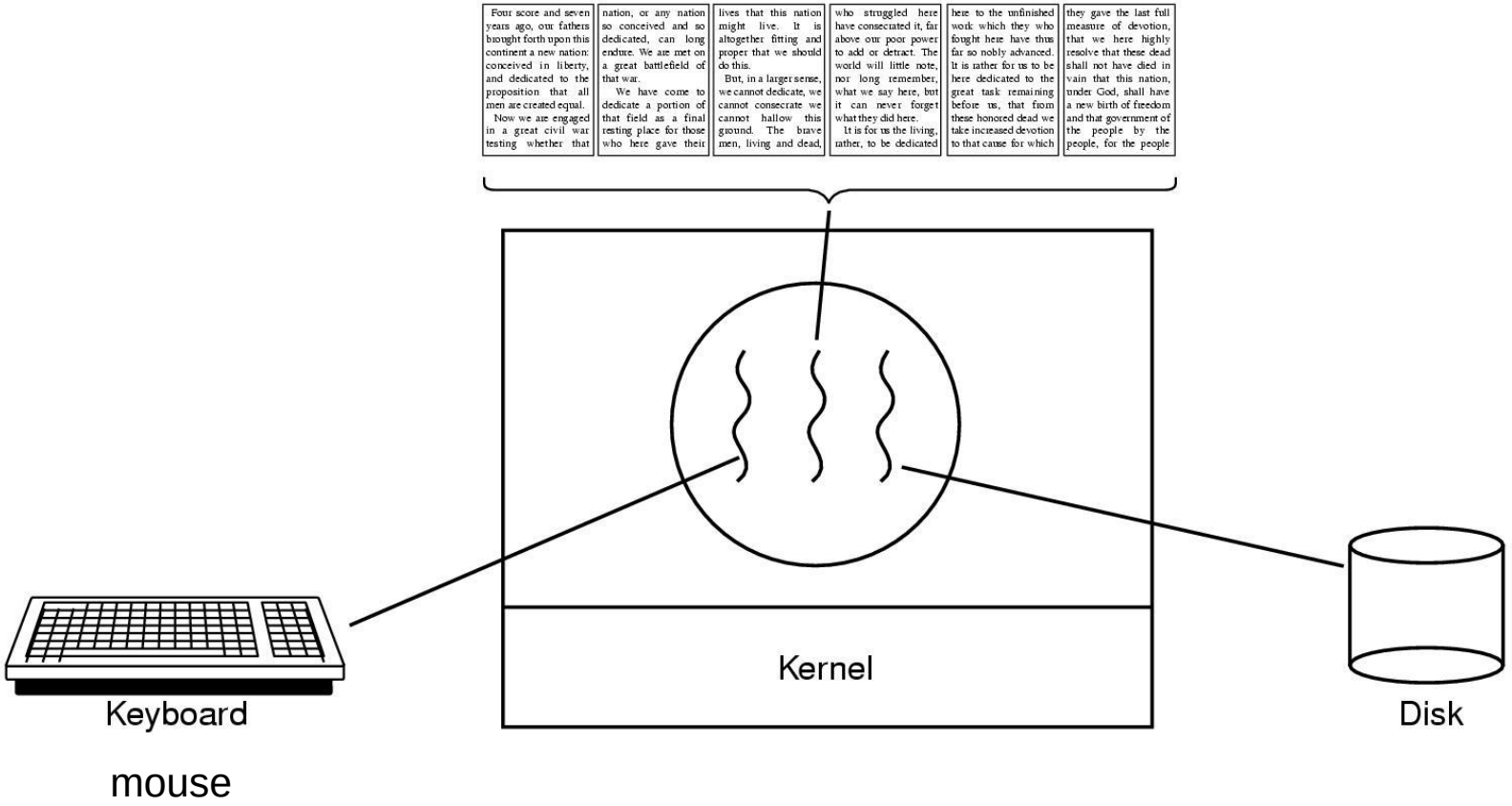
- Mở rộng khái niệm tiến trình truyền thống bằng cách thực hiện nhiều luồng thực thi trong cùng một môi trường của tiến trình.

- tiến trình gồm
  - Không gian địa chỉ (text section, data section)
  - Một hay nhiều luồng thực thi (thread of execution), mỗi luồng thực thi (thread) có riêng
    - program counter
    - các register
    - stack
  - Các tài nguyên khác (các open file, các tiến trình con,...)

tiến trình multithreaded

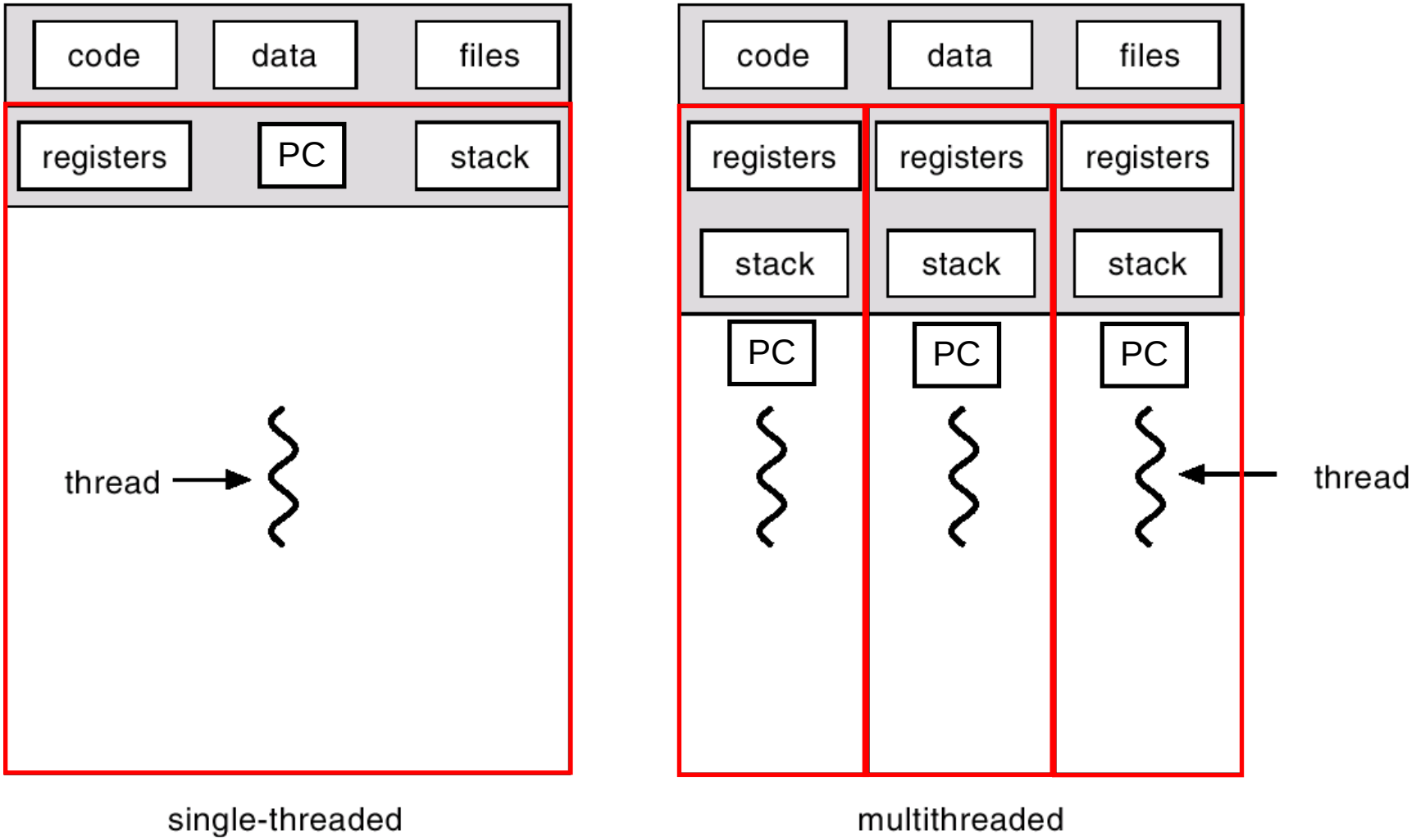
- Các thread trong cùng một process chia sẻ code section, data section và tài nguyên khác (các file đang mở,...) của process.
- tiến trình *đa luồng* (*multithreaded* process) là tiến trình có nhiều luồng.

Sử dụng thread

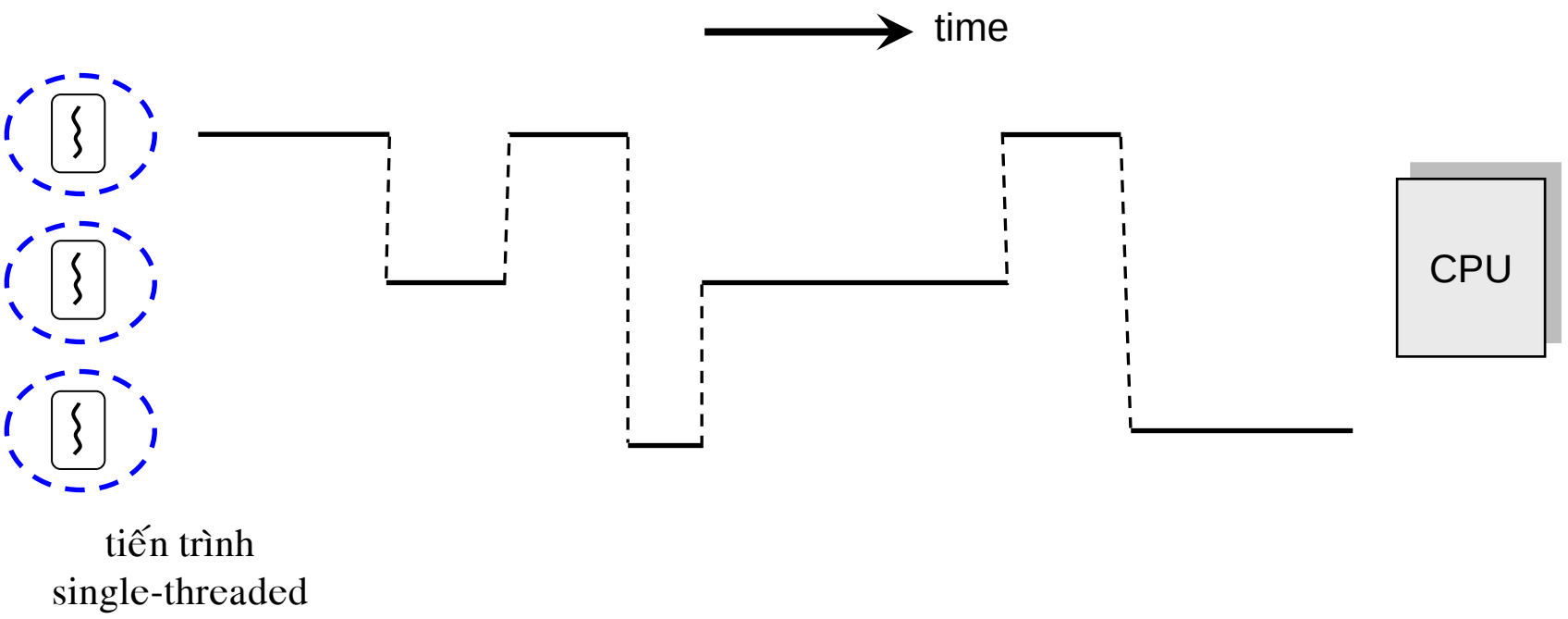


A word processor with three threads

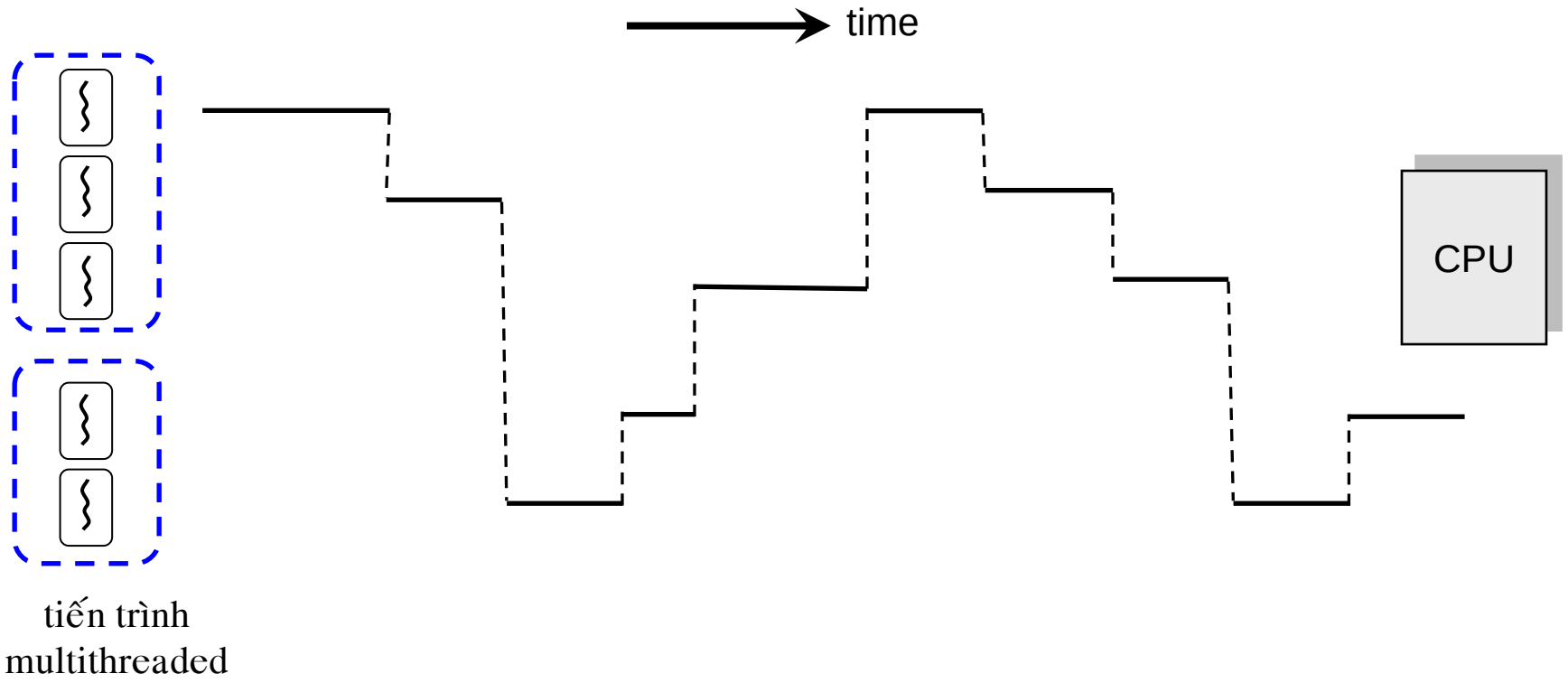
Single và multithreaded process



Multiplexing CPU giữa các thread

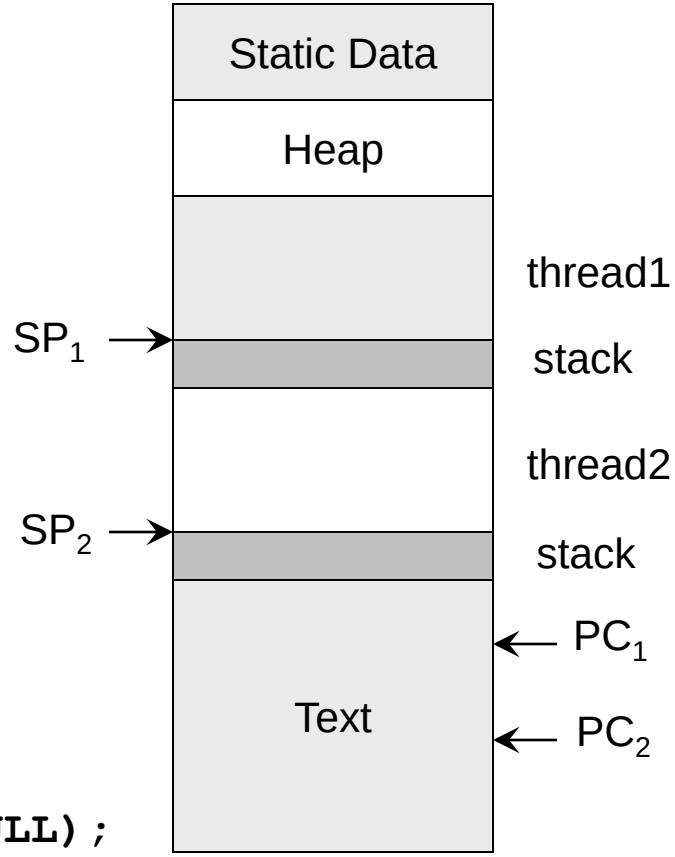


Multiplexing CPU giữa các thread (tt)



Ví dụ Pthread program (Linux)

```
#include <stdio.h>
void* thread1(){
   int i;
   for (i = 0; i < 10; i++){
        printf("Thread 1\n"); sleep(1);
   }
}
void* thread2(){
   int i;
   for (i = 0; i < 10; i++){
        printf("Thread 2\n"); sleep(1);
}
int main(){
   pthread_t th1, th2;
   pthread_create(&th1, NULL, thread1, NULL);
   pthread_create(&th2, NULL, thread2, NULL);
   sleep(20);
   return 0;
}
```



Sơ đồ bộ nhớ

Ưu điểm của thread

- Tính đáp ứng (responsiveness) cao cho các ứng dụng tương tác multithreaded
- Chia sẻ tài nguyên (resource sharing): vd memory
- Tiết kiệm chi phí hệ thống (economy)
  - Chi phí tạo/quản lý thread nhỏ hơn so với tiến trình
  - Chi phí chuyển ngữ cảnh giữa các thread nhỏ hơn so với tiến trình
- Tận dụng kiến trúc đa xử lý (multiprocessor)
  - Mỗi thread chạy trên một processor riêng, do đó tăng mức độ song song của chương trình.

User thread

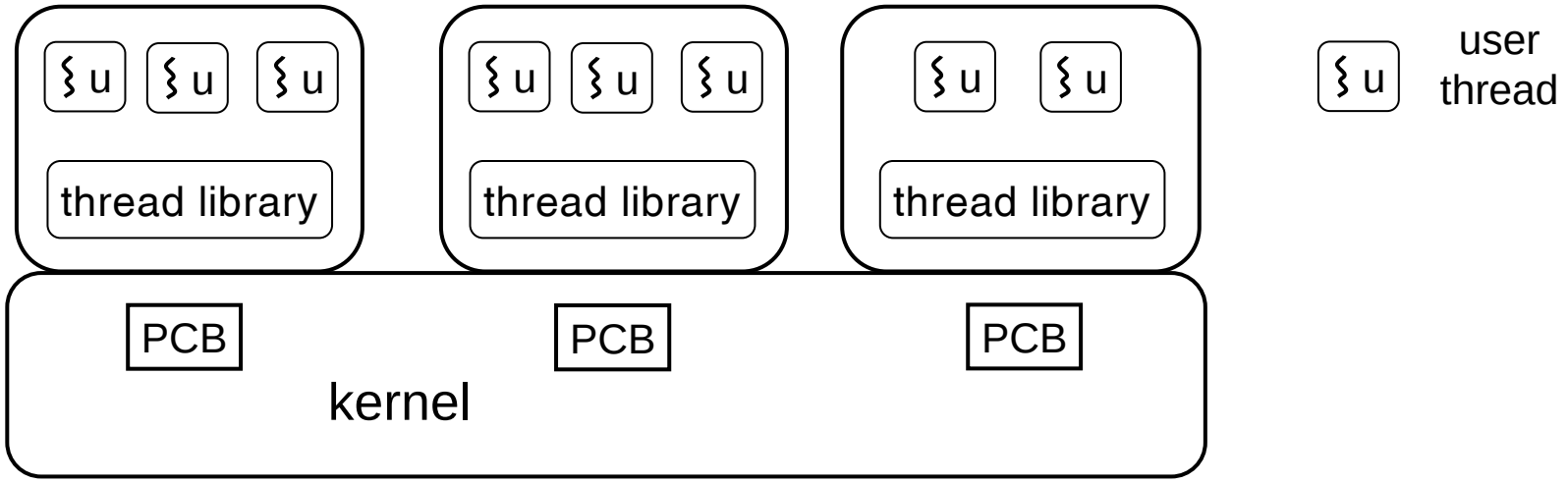
- Một *thư viện thread* (thread library, run-time system) được thực hiện trong user space để hổ trợ các tác vụ lên thread
  - Thư viện thread cung cấp các hàm khởi tạo, định thời và quản lý thread như
    - thread_create
    - thread_exit
    - thread_wait
    - thread_yield
  - Thư viện thread dùng *Thread Control Block* (TCB) để lưu trạng thái của user thread (program counter, các register, stack)

User thread (tt)

- Kernel không biết sự có mặt của user thread
- Ví dụ thư viện user-thread
  - POSIX Pthreads

User thread (tt)

- Ví dụ: hệ điều hành truyền thống chỉ cung cấp một "kernel thread" duy nhất (biểu diễn bởi một PCB) cho mỗi process.
  - *Blocking problem:* Khi một thread trở nên blocked thì kernel thread cũng trở nên blocked, do đó mọi thread khác của process cũng sẽ trở nên blocked.



Kernel thread

- Cơ chế multithreading được hệ điều hành trực tiếp hỗ trợ
  - Kernel quản lý cả process và các thread
  - Việc định thời CPU được kernel thực hiện trên thread

Kernel thread (tt)

- Cơ chế multithreading được hỗ trợ bởi kernel
  - Khởi tạo và quản lý các thread chậm hơn
  - Tận dụng được lợi thế của kiến trúc multiprocessor
  - Thread bị blocked không kéo theo các thread khác bị blocked.

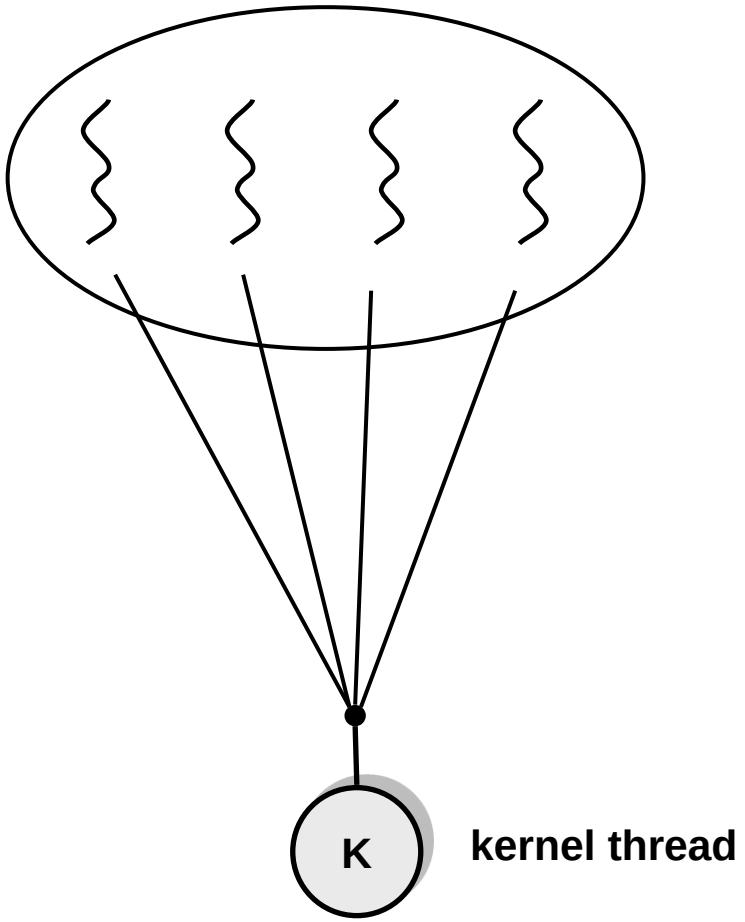
- Một số hệ thống multithreading (multitasking)
  - Windows 9x/NT/200x
  - Solaris
  - Linux

thực hiện thread

- Thread có thể thực hiện theo một trong các mô hình sau
  - Mô hình *many-to-one*
  - Mô hình *one-to-one*
  - Mô hình *many-to-many*

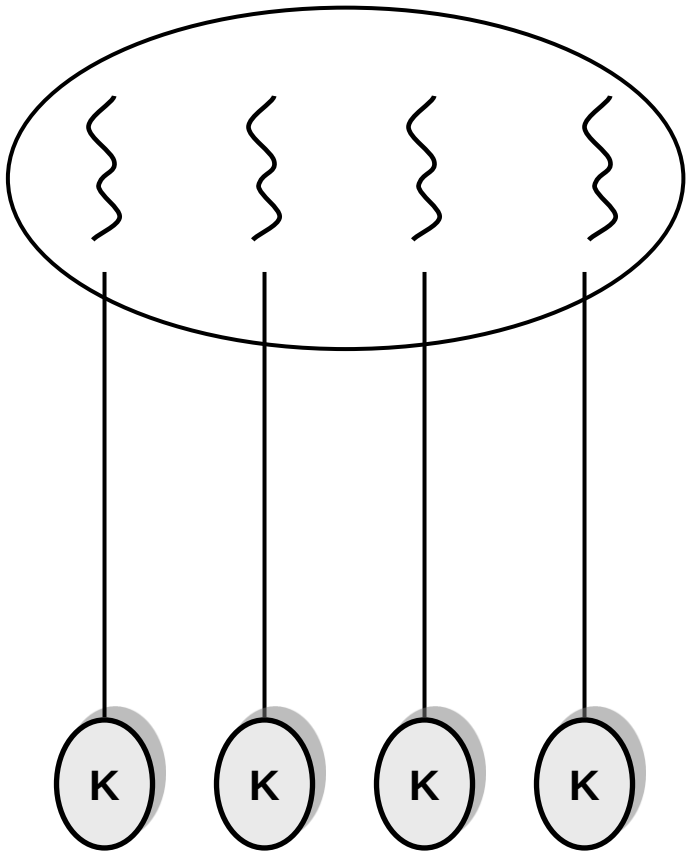
Mô hình many-to-one

- Nhiều user-level thread "chia sẻ" một kernel thread để thực thi
  - Việc quản lý thread được thực hiện thông qua các hàm của một thread library được gọi ở user level.
  - Blocking problem: Khi một thread trở nên blocked thì kernel thread cũng trở nên blocked, do đó mọi thread khác của process cũng sẽ trở nên blocked.
- Có thể được thực hiện đối với hầu hết các hệ điều hành.



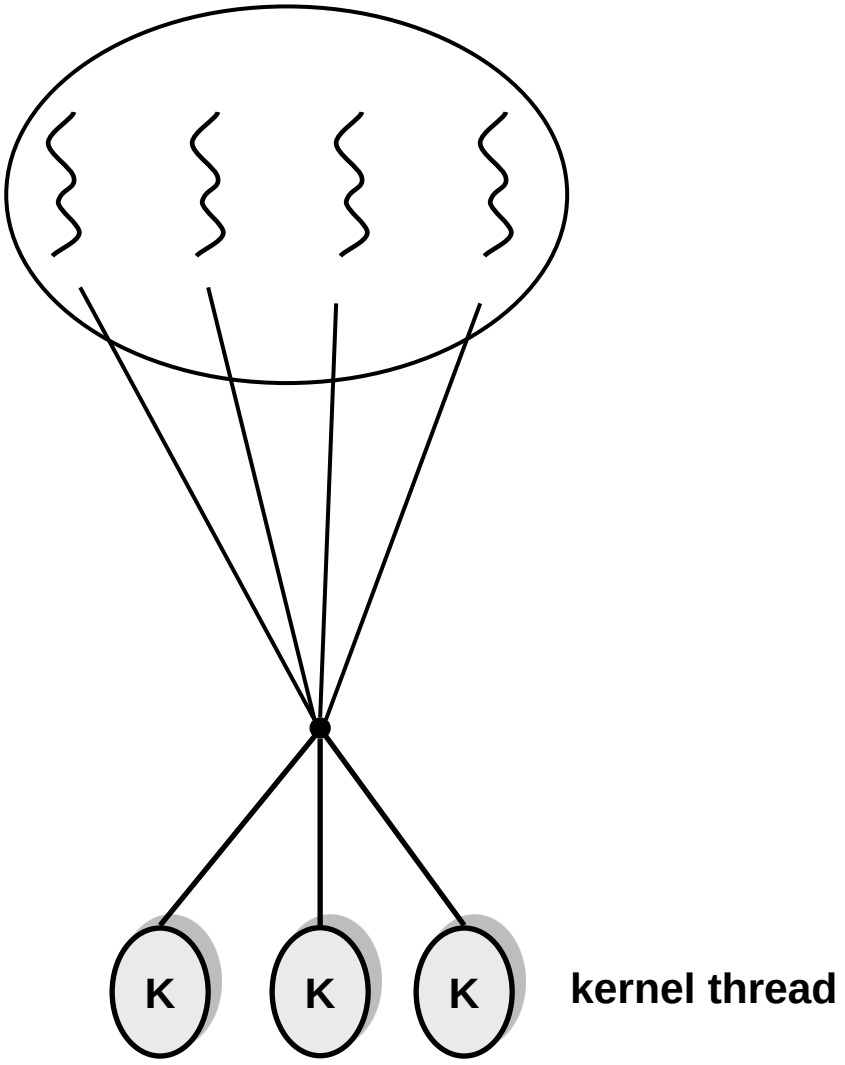
Mô hình one-to-one

- Mỗi user-level thread thực thi thông qua một kernel thread riêng của nó
  - Mỗi khi một user thread được tạo ra thì cũng cần tạo một kernel thread tương ứng
- Hệ điều hành phải có cơ chế cung cấp được nhiều kernel thread cho một tiến trình
- Ví dụ: Windows NT/2000



Mô hình many-to-many

- Nhiều user-level thread được phân chia thực thi (multiplexed) trên một số kernel thread.
  - Tránh được một số khuyết điểm của hai mô hình many-to-one và one-to-one
- Ví dụ
  - Solaris 2
  - Windows NT/2000 với package ThreadFiber



Pthreads

- Chuẩn POSIX (IEEE 1003.1c) cung cấp các API hỗ trợ tạo thread và đồng bộ thread (synchronization)
- Phổ biến trong các hệ thống UNIX/Linux
- Là một thư viện hỗ trợ user-level thread
  - Tham khảo thêm ví dụ về lập trình thư viện Pthread với ngôn ngữ C trong hệ thống Unix-like, trang 140, “Operating System Concepts”, Silberschatz et al, 6th Ed, 2003.

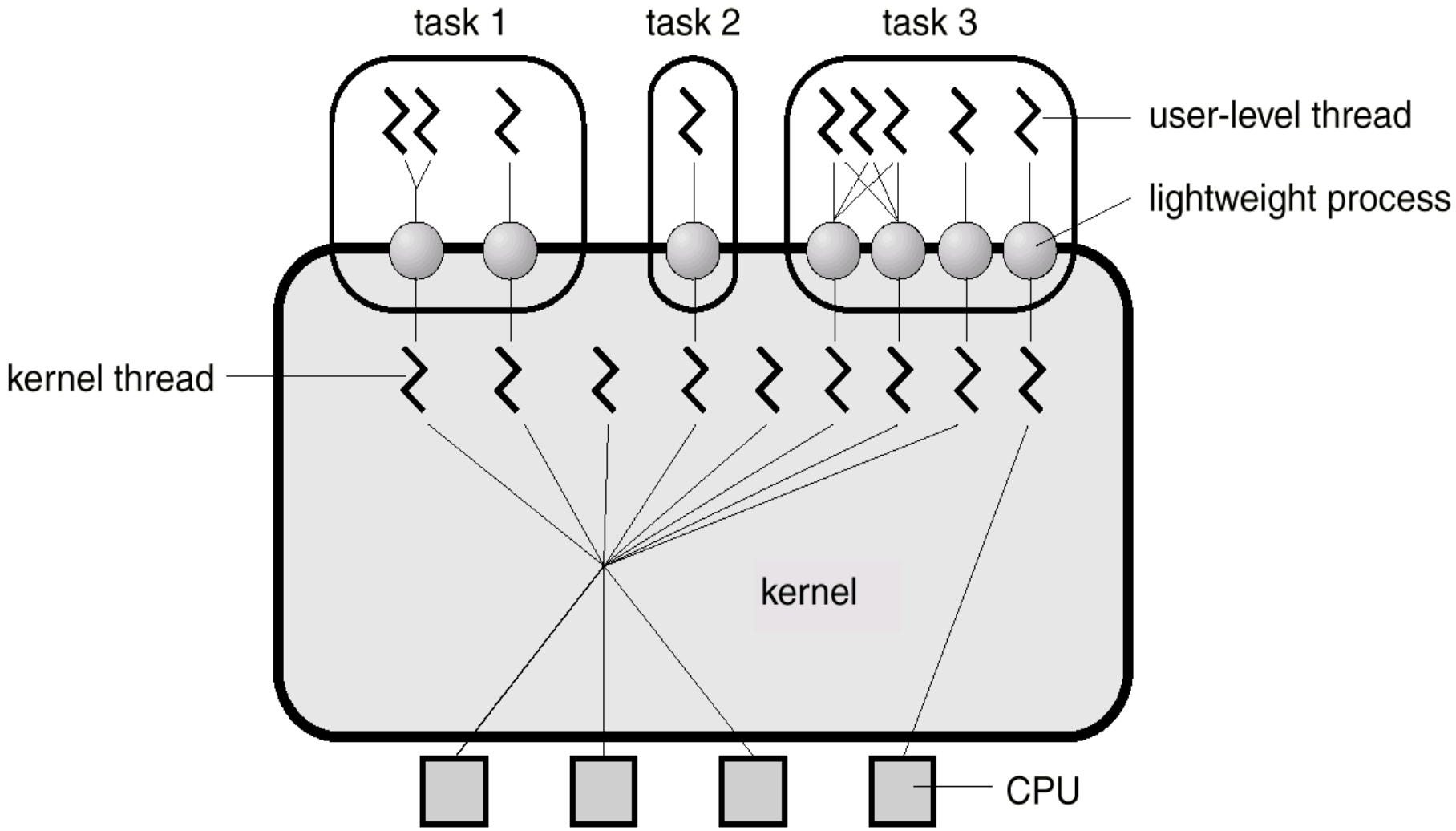
- Biên dịch và thực thi chương trình multithreaded C trong Linux

```
$ gcc source_file.c -lpthread –o output_file
$ ./output_file
```

Thread trong Solaris

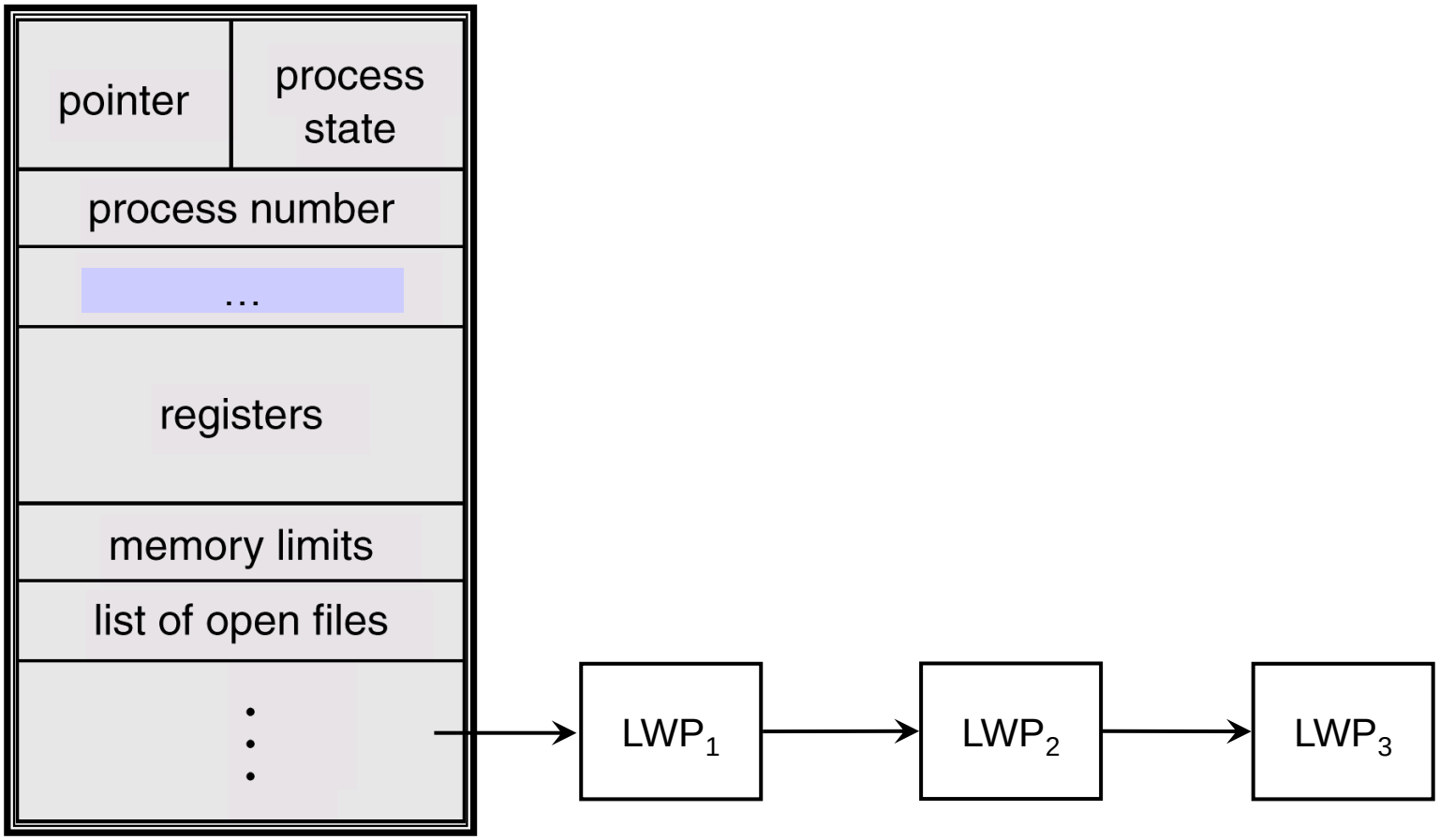
- User-level threads
  - Pthread và UI-thread
- *Lightweight process* (LWP)
  - Mỗi process chứa ít nhất một LWP
  - Thư viện thread có nhiệm vụ phân định user thread vào các LWP
    - User-level thread được gắn với LWP thì mới được thực thi.
  - Thư viện thread chịu trách nhiệm điều chỉnh số lượng LWP
- Kernel-level thread
  - Mỗi LWP tương ứng với một kernel-level thread
  - Ngoài ra, hệ thống còn có một số kernel threads dành cho một số công việc ở kernel (các thread này không có LWP tương ứng)
  - Đối tượng được định thời trong hệ thống là các kernel thread

Thread trong Solaris (tt)



many-to-many

Thread trong Solaris (tt)



Quá trình trong Solaris

Thread trong Java

- Hỗ trợ tạo và quản lý thread ở mức ngôn ngữ lập trình (language-level)
- Tất cả chương trình Java chứa ít nhất là một thread
- Các thread của Java được quản lý bởi JVM
- Hai phương pháp tạo Java Threads
  1. extend Thread class và override method run()
  2. Thực thi (implementing) Runnable interface