Hệ Điều Hành

(*Nguyên lý các hệ điều hành*)

Đỗ Quốc Huy
huydq@soict.hust.edu.vn
Bộ môn Khoa Học Máy Tính
Viện Công Nghệ Thông Tin và Truyền Thông

Chapter 2 Process Management

① Process's definition

Chapter 2 Process Management

Process (review)

- A running program
  - Provided resources (CPU, memory, I/O devices. . .) to complete its task
  - Resources are provided at:
    - The process creating time
    - While running
- The system includes many processes running at the same time
  - OS's process: Perform system instruction code
  - User's process: Perform user's code
- Process may contain one or more threads
- OS's roles:
  - Guarantee process and thread activities
  - Create/deltete process (user, system)
  - Process scheduling
  - Provide synchronization mechanism, communication and prevent deadlock among processes

① Process
② Thread
③ CPU scheduling
④ Critical resource and process synchronization
⑤ Deadlock and solutions

Chapter 2 Process Management

1. Process

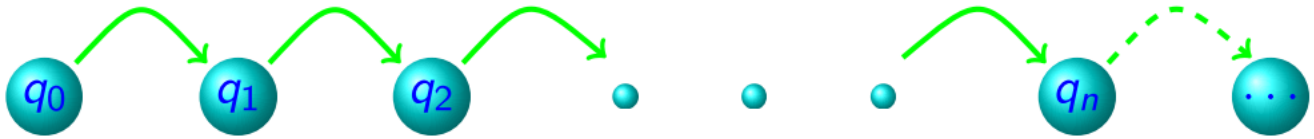
1.1. Notion of process

- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

System's state

- Processor: Registers' values
- Memory: Content of memory block
- Peripheral devices: Devices' status

- Program's executing ⇒ system's state changing
  - Change discretely, follow each executed instruction



- Process: a sequence of system's state changing
  - Begin with start state
  - Change from state to state is done according to requirement in user's program

Process is the execution of a program

Process >< program

- ***Program***: *passive object (content of a file on disk)*
  - *Program's code: machine instruction (CD2190EA...)*
  - *Data:*
    - *Variable stored and used in memory*
    - *Global variable*
    - *Dynamic allocation variable (malloc, new,..)*
    - *Stack variable (function's parameter, local variable)*
  - *Dynamic linked library (DLL)*
    - *Not compiled and lined with program*

When program is running, minimum resource requirement

- *Memory for program's code and data*
- *Processor's registers used for program execution*

- ***Process***: *active object (instruction pointer, set of resources)*

A program can be

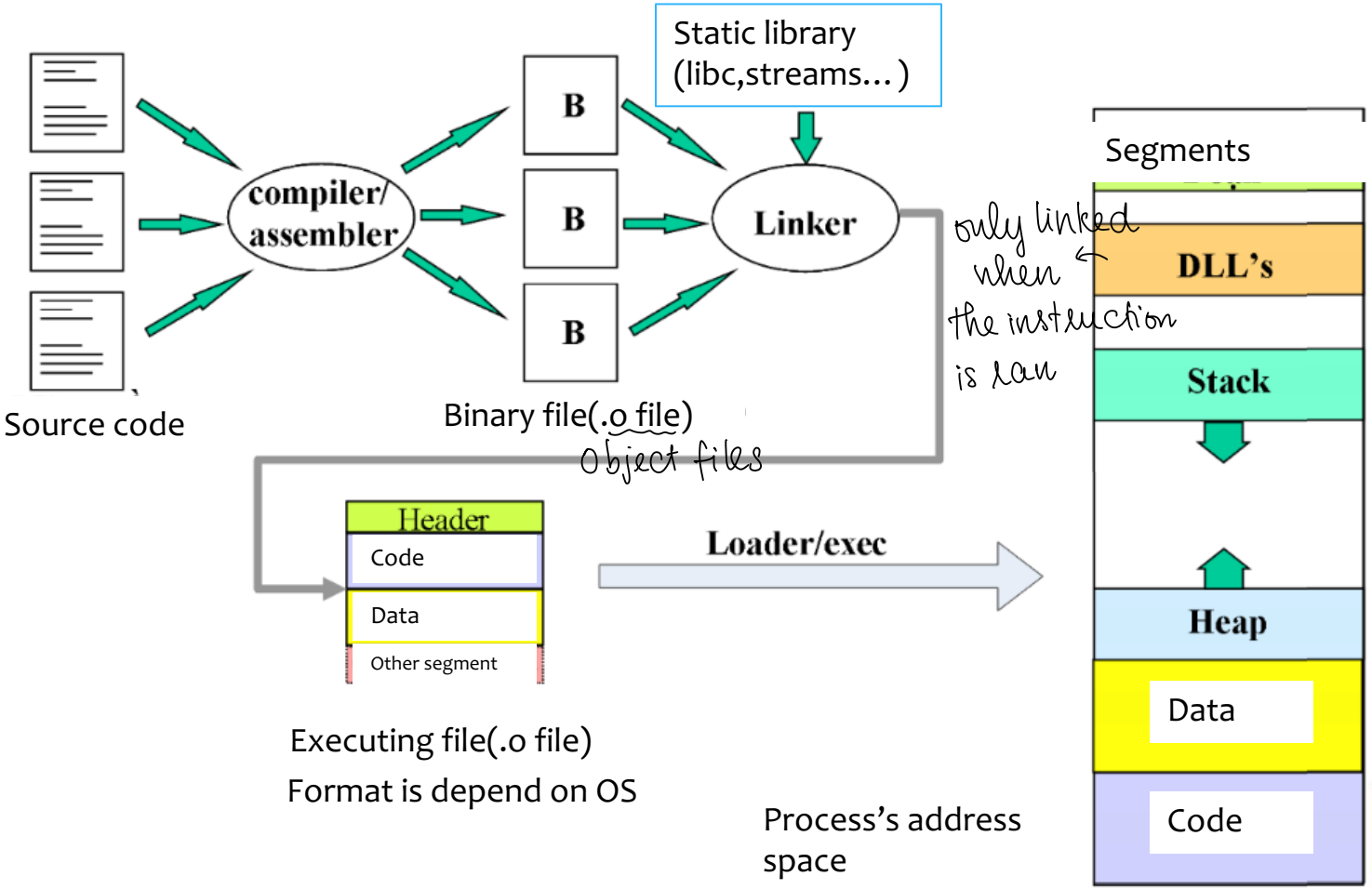
- *Part of process's state*
  - *One program, many process (different data set)*
    Example: gcc hello.c || gcc baitap.c input parameters
- *Call to many process*

(+) modify independently from program

(+) smaller program

Static (.lib)	Dynamic (.dll)
copied + complied w/ the program (copy entire lib to code; lấy lib to thì mã nguồn ra dài hơn)	NOT copied but only linked to the program (chạy đến instruction nào thì kiểm dll khớp để dùng)

Compile and run a program



Compile and run a program

includes a new data struct to manage the process

- *The OS creates a process and allocate a memory area for it*
- *System program loader/exec*
  - *Read and interprets the executive file (file's header)*
  - *Set up address space for process to store code and data from executive file*
  - *Put command's paramenters, environment variable (argc, argv, envp) into stack*
  - *Set proper values for processor's registerand call "_start()" (OS's fucntion)* = parse the 1st address of the program → IP register
- *Program begins to run at "_start()". This function call to main() (program's functions)*

⇒*"Process" is running, not mention "program" anymore*

- *When main() function end, OS call to "_exit()" to cancel the process and take back resources*

study how to use this

Process's state

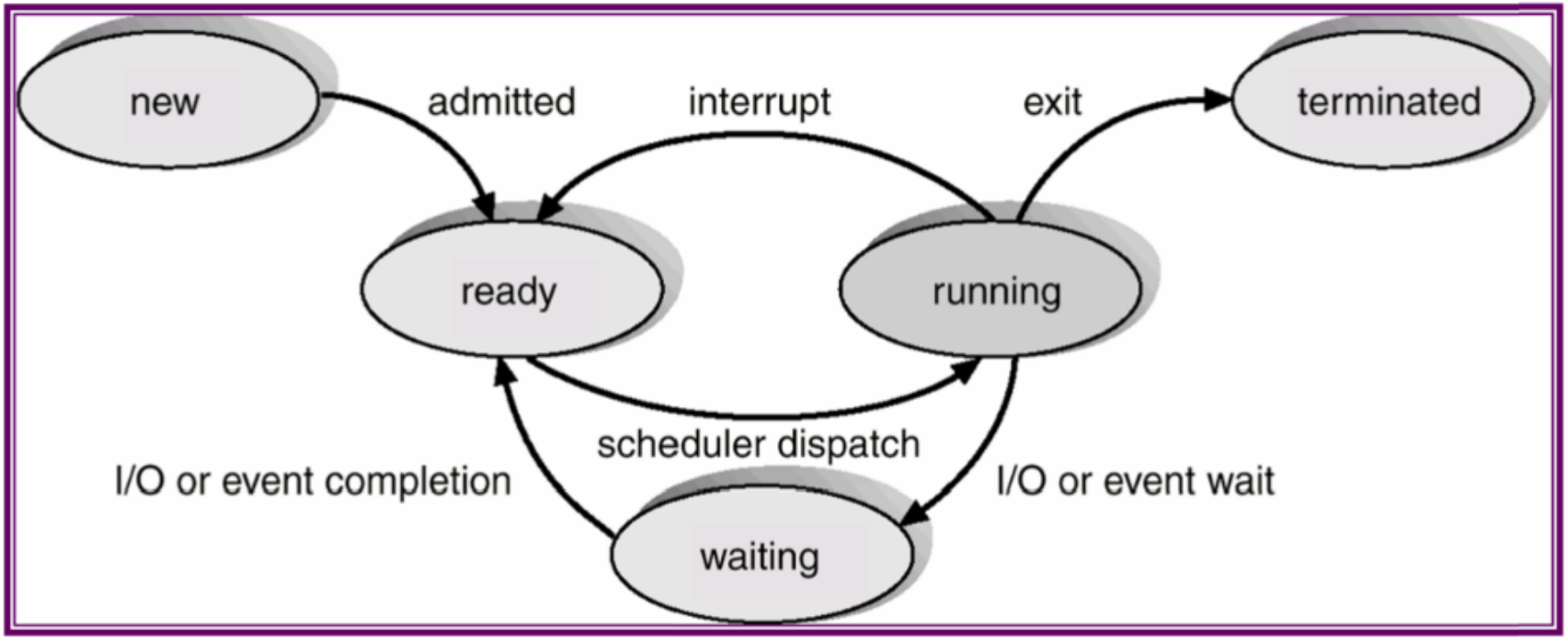
When executing, process change its state

- New (Created) Process is being initialized
- Ready Process is waiting for its turn to use physical processor (everything ready, except CPU)
- Running Process's instructions are being executed
- Waiting (Blocking) Process is waiting for an event to appear (I/O operation completion)
- Terminated (Finished) Process finish its job (delete any associated data structure/process)

Process's state is part of process current's activity

(there are more than 5 states *)

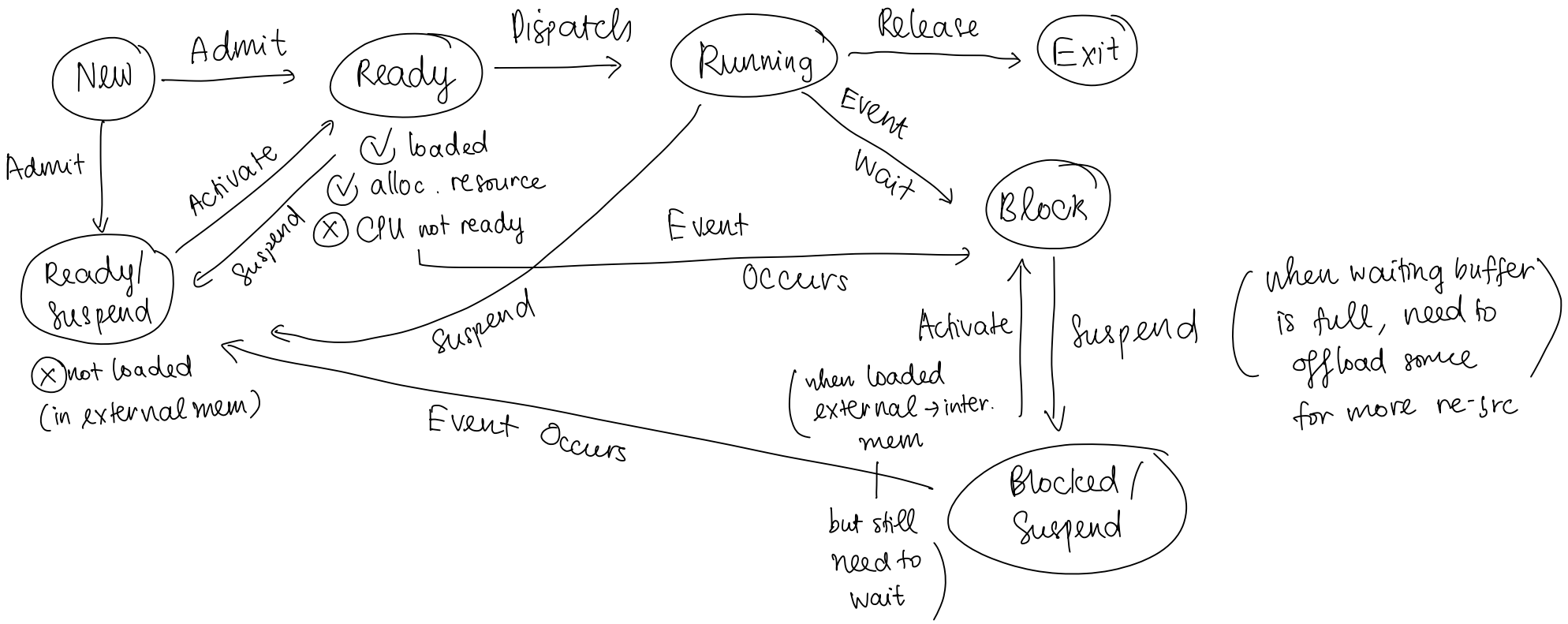
Process's states changing flowchart (Silberschatz 2002)



System that has only one processor

- Only one process in running state
- Many processes in ready or waiting state

7-State Model

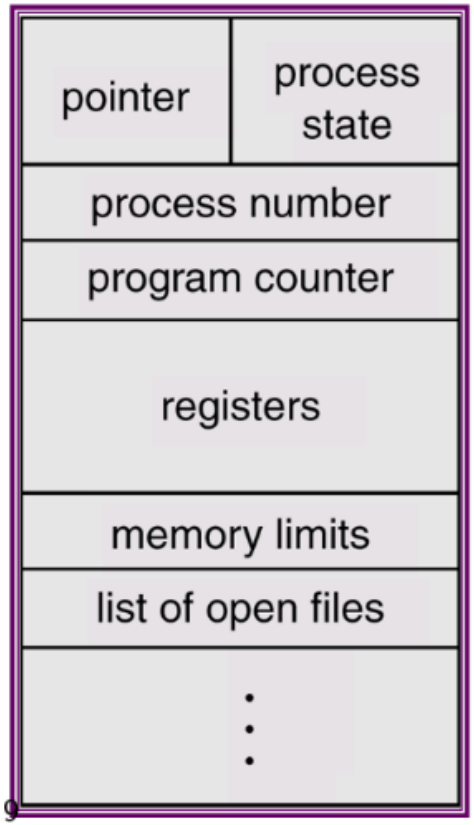


Process Info (Windows) ; Task Struct (Linux)

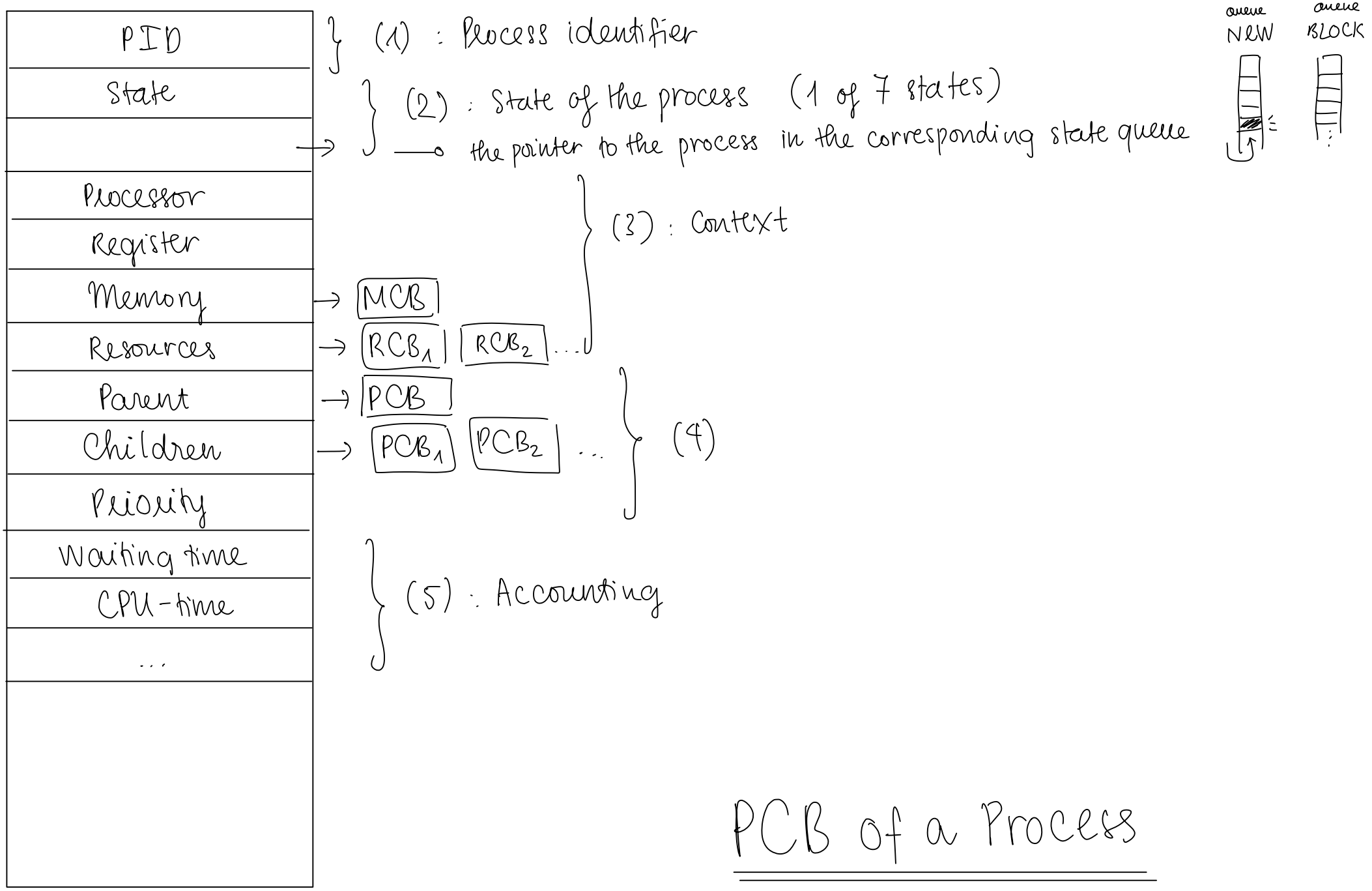
Process Control Block (PCB)

an abstract name ; diff. implementation in diff. OS

- Each process is presented in the system by a process control block
- PCB: an information structure allows identify only one process



- Process state
- Process counter
- CPU's registers
- Information for process scheduling
- Information for memory management
- Inforamtion of usable resources
- Statistic information
- Pointer to another PCB
- . . .

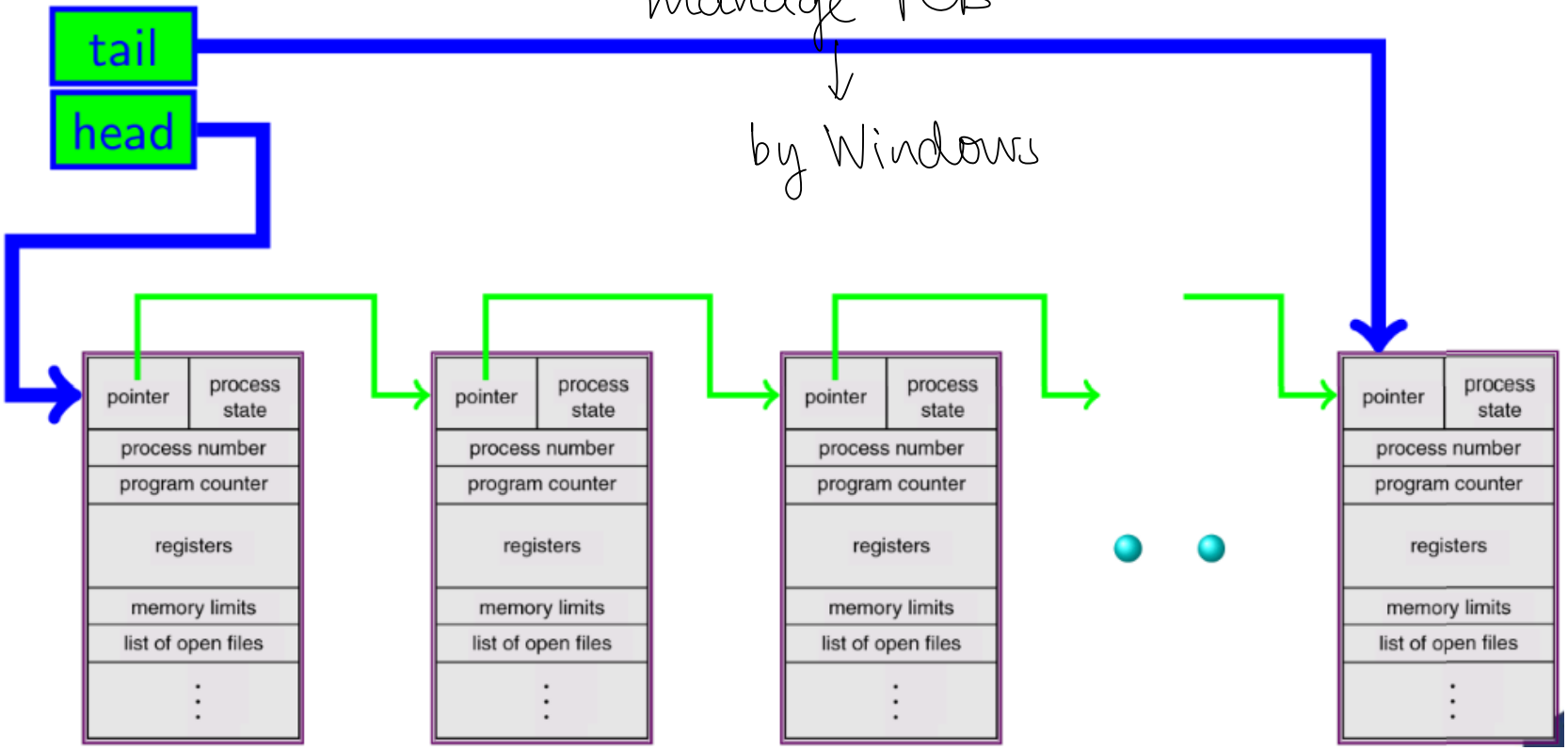


PCB of a Process

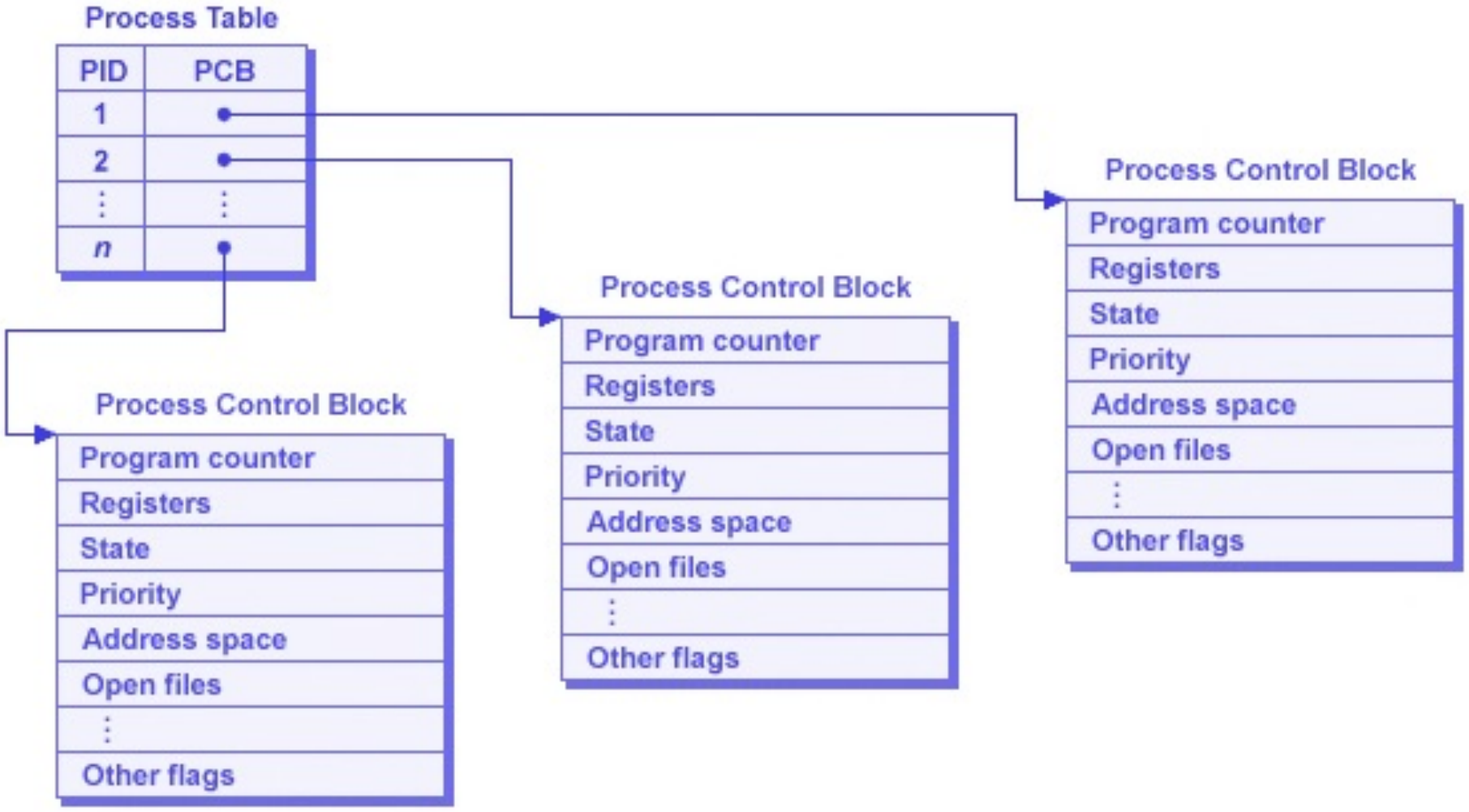
Process List

(Linked List is a data structure used to manage PCB)

by Windows



Linux Process Table



Single-thread process and Multi-thread process

- Single-thread process : A running process with only one executed thread
  Have one thread of executive instruction
  ⇒ Allow executing only one task at a moment

- Multi-thread process : Process with more than one executed thread
  ⇒ Allow more than one task to be done at a moment

- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

Introduction

Objective Maximize CPU's usage time

⇒ Need many processes in the system

Problem CPU switching between processes

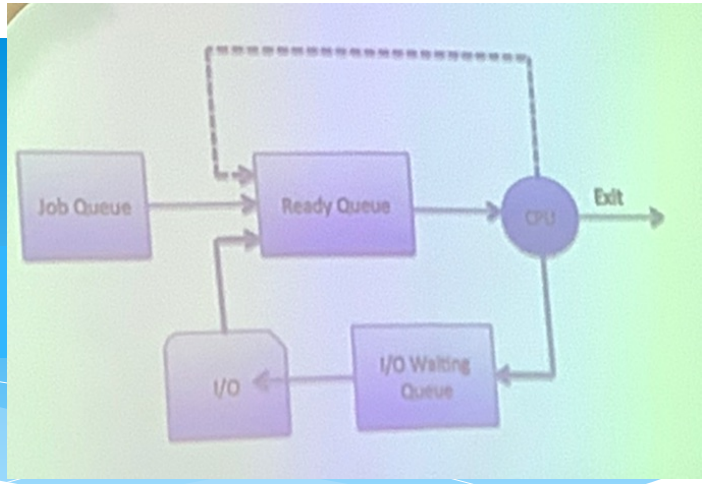
⇒ Need a queue for processes

Single processor system

⇒ One process running

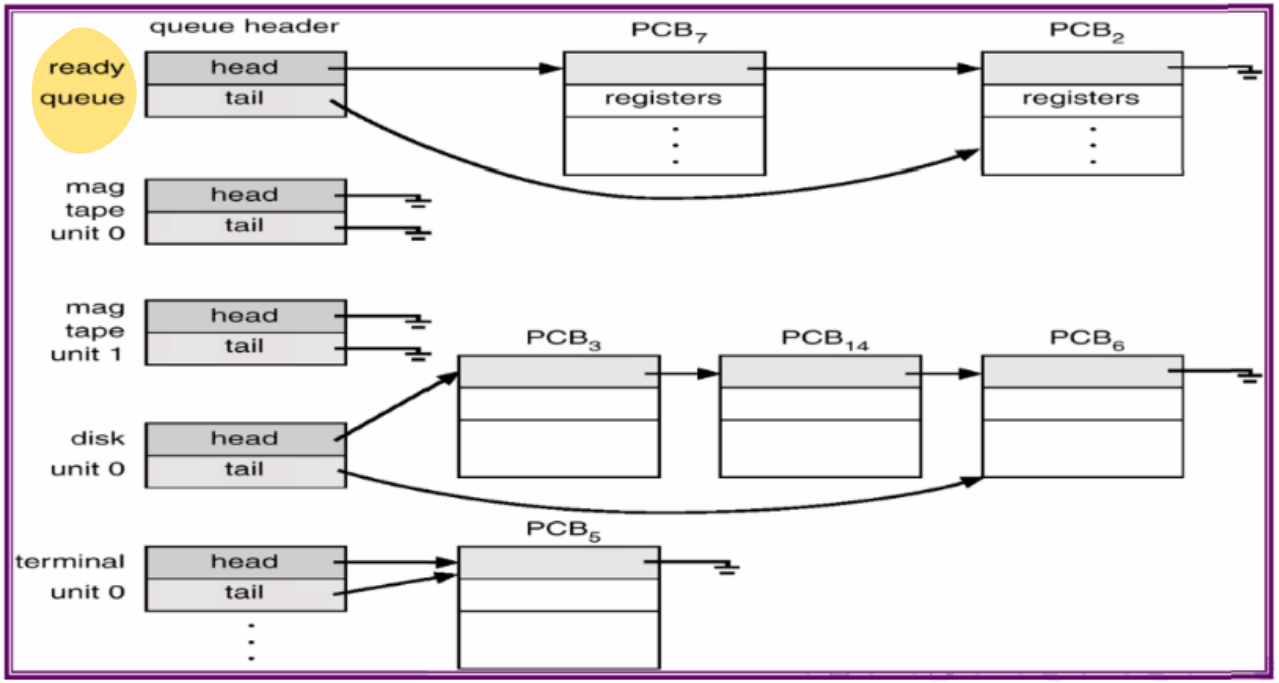
⇒ Other processes have to wait until processor is free

Process queue I



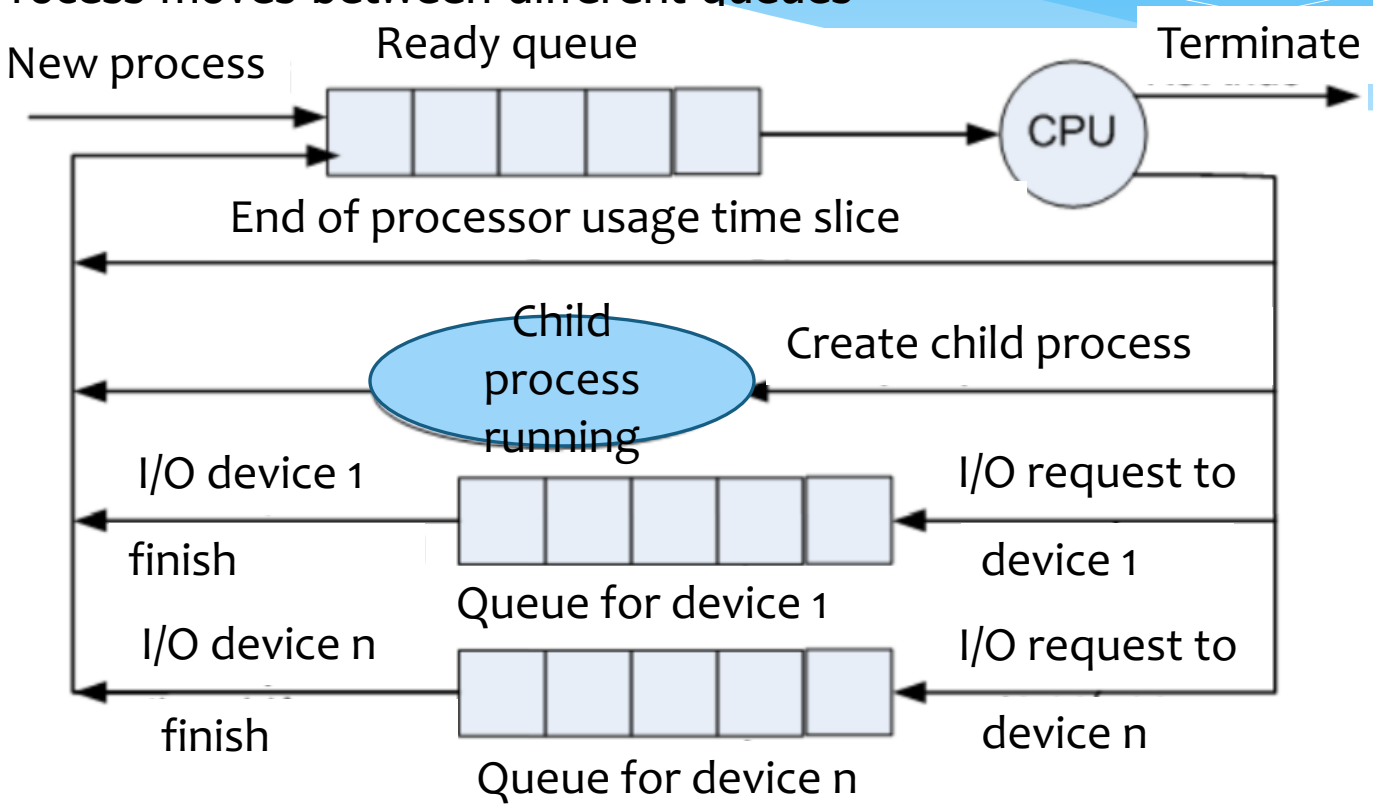
The system have many queues for processes

- **Job-queue Set of processes in the system**
- **Ready-Queue Set of processes exist in the memory, ready to run**
- **Device queues Set of processes waiting for an I/O device. Queues for each device are different**



Process queue II

- Process moves between different queues

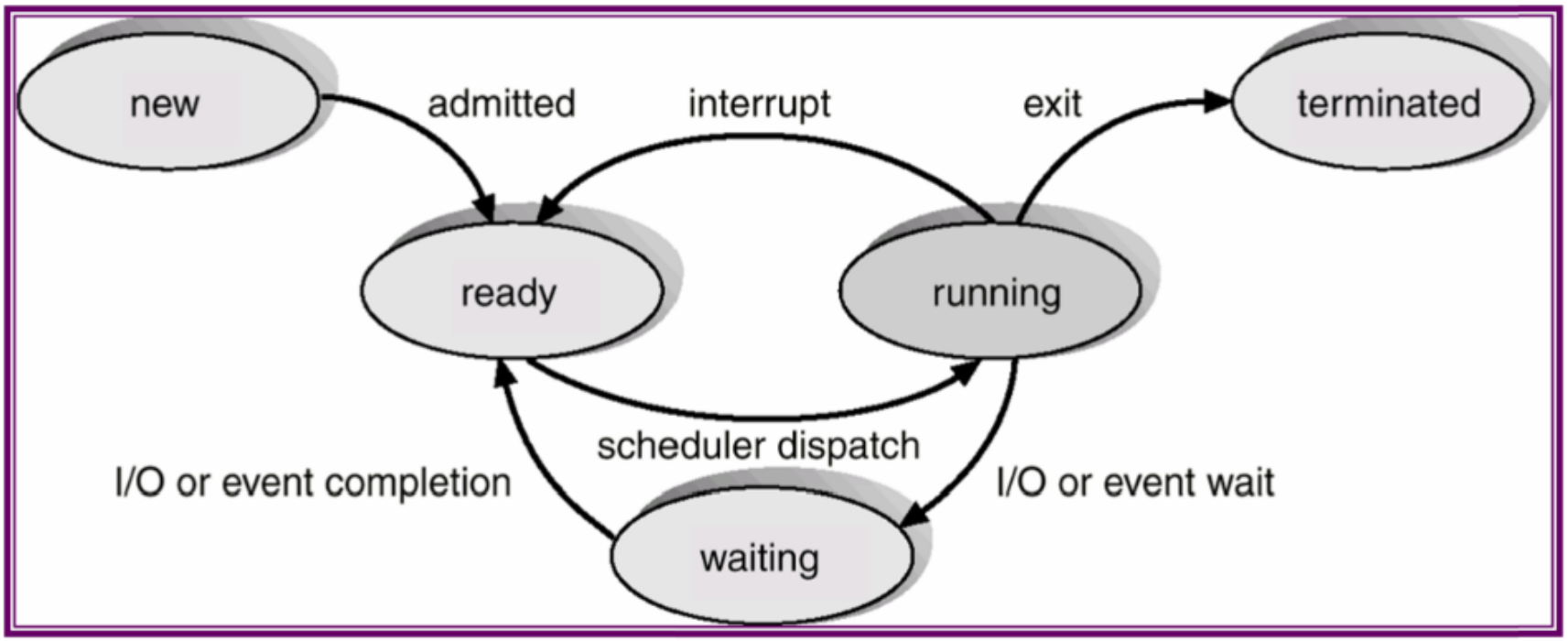


- Newly created process is putted in ready queue and wait until it's selected to execute

Process queue III

- Process selected and running
  - ① process issue an I / O request, and then be placed in an I / O queue
  - ② process create a new sub-process and wait for its termination
  - ③ process removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue
- In case (1&2) after the waiting event is finished,
  - Process eventually switches from waiting state to ready state
  - Process is then put back to ready queue
- Process continues this cycle (ready, running, waiting) until it terminates.
  - it is removed from all queues
  - its PCB and resources deallocated

Scheduler



* Selection process is carried out by the appropriate scheduler
  - Job scheduler; Long-term scheduler
  - CPU scheduler; Short-term scheduler

} 2 basic types of scheduler

Job Scheduler

(load : external → internal)

- Select processes from process queue stored in disk and put into memory to execute
- Not frequently (seconds/minutes)
- Control the degree of the multi-programming (number of processin memory)
- When the degree of multi-programming is stable, the scheduler may need to be invoked when a process leaves the system

(which process to bring into main memory? – from a queue of waiting proc.)

- Job selection problem
  - I/O bound process: use less CPU time
  - CPU bound process: use more CPU time
  - Should select good process mix of both

  ⇒ I/O bound process: ready queue is empty, waste CPU

  ⇒ CPU bound: device queue is empty, device is wasted

1.2. Process Scheduling

Job Scheduler

- Job selection's problem
  - Consider the following program

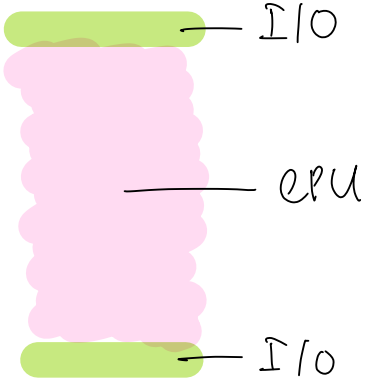
```
PROGRAM PrintValue:
BEGIN
  Input A;
  Input B;                                  IO
  C = A + B;
  D = A - B;                                CPU
  Print "The sum of inputs is: ", C;
  Print "The Difference of inputs is: ", D;  IO
END.
```

I/O Bound Process :

↳ Utilizes less CPU time

CPU Bound Process :

↳ Use more CPU time

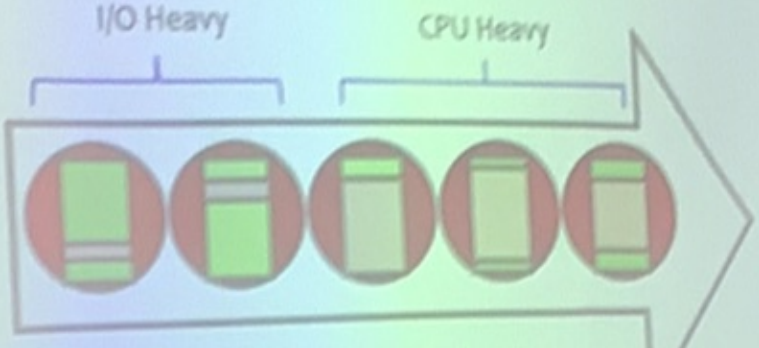


1.2. Process Scheduling

Job Scheduler

- Job selection's problem
  - Should select good process mix of both !!!
  - ⇒ I/O bound : ready queue is empty, waste CPU
  - ⇒ CPU bound: device queue is empty, device is wasted

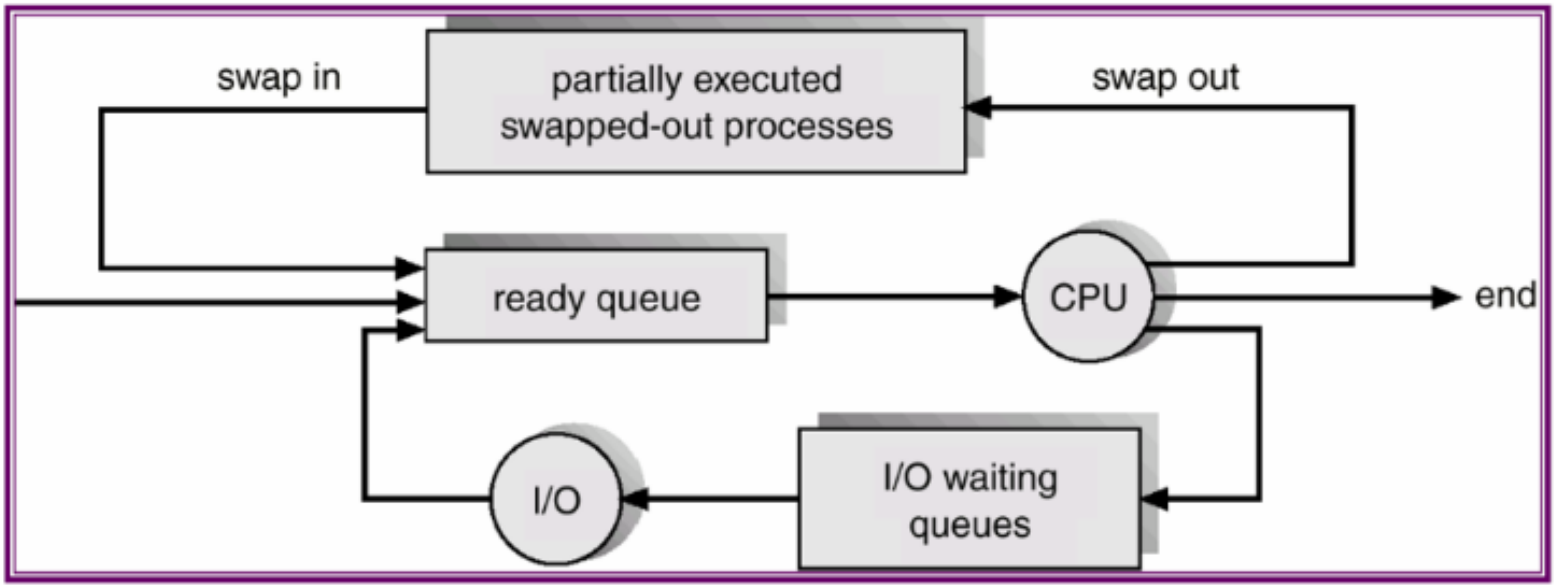
If input are the following jobs



CPU Scheduler

* Selects from processes that are ready to execute, and allocates the CPU to one of them
- Frequently work (example: at least 100ms/once)
  - A process may execute for only a few milliseconds before waiting for an I/O request
  - Select new process that is ready
- the short-term scheduler must be fast
  - 10ms to decide ⇒10/(110)=9% CPU time is wasted
- Process selection problem (CPU scheduler)

Process swapping (Medium-term scheduler)



- Task
  - Removes processes from memory, reduces the degree of multiprogramming
  - Process can be reintroduced into memory and its execution can be continued where it left off
- Objective: Free memory, make an wider unused memory area

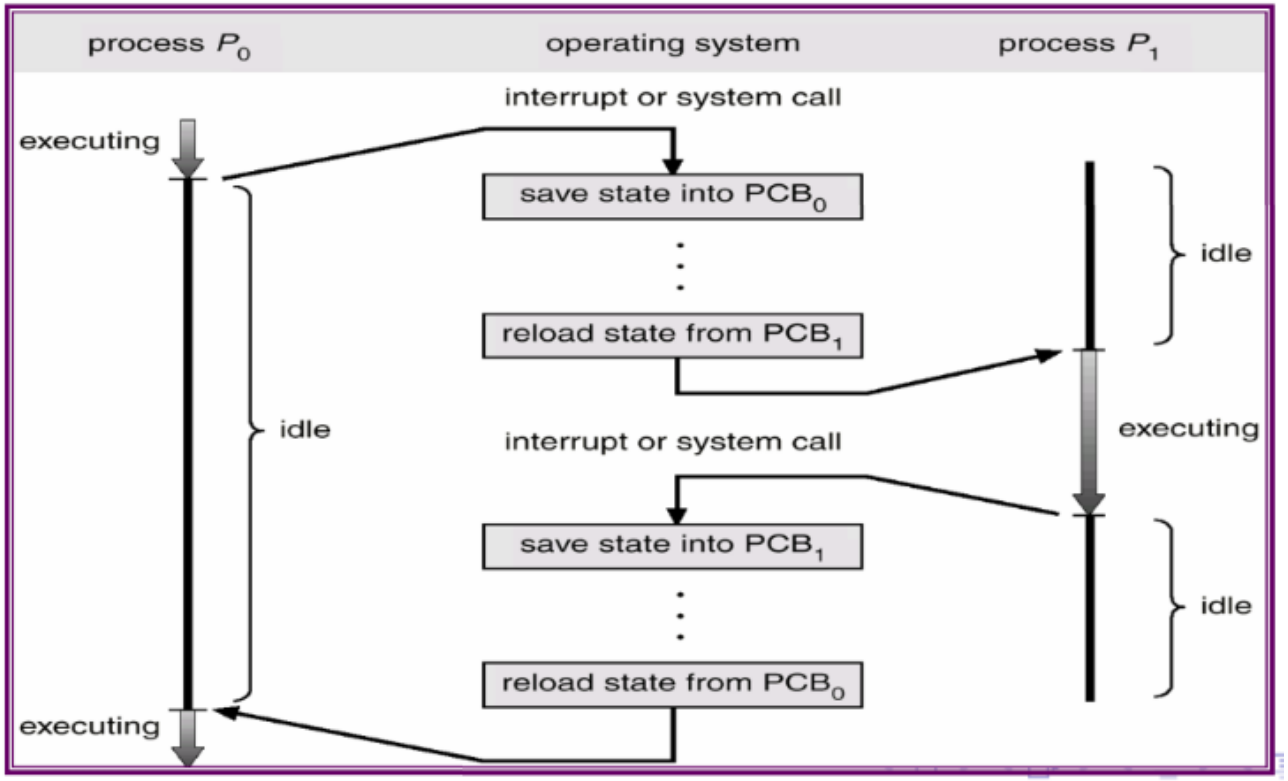
Context switch

- **Switch CPU from one process to another process**
  - Save the state of the old process and loading the saved state for the new process
    - includes the value of the CPU registers, the process state and memory-management information
  - Take time and CPU cannot do anything while switching
  - Depend on the support of the hardware
  - More complex the operating system, the more work must be done during a context switch

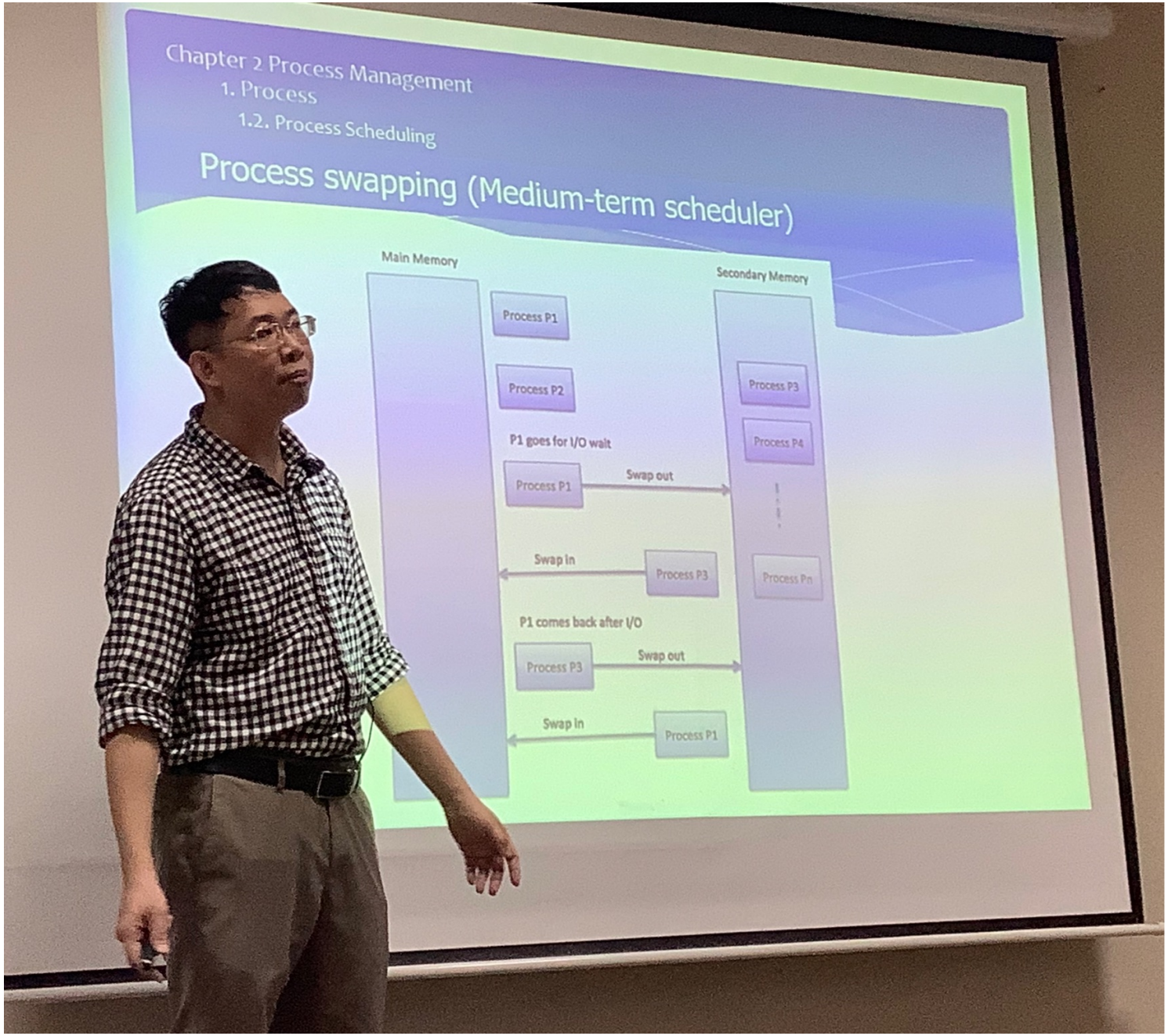
Bigger PCB ⇒ more computing effort

Context switch

- Occurs when an intterupt signal appear(timely interrupt) or when process issue a system call (to perform I/O)
- CPU switching diagram(Silberschatz 2002)



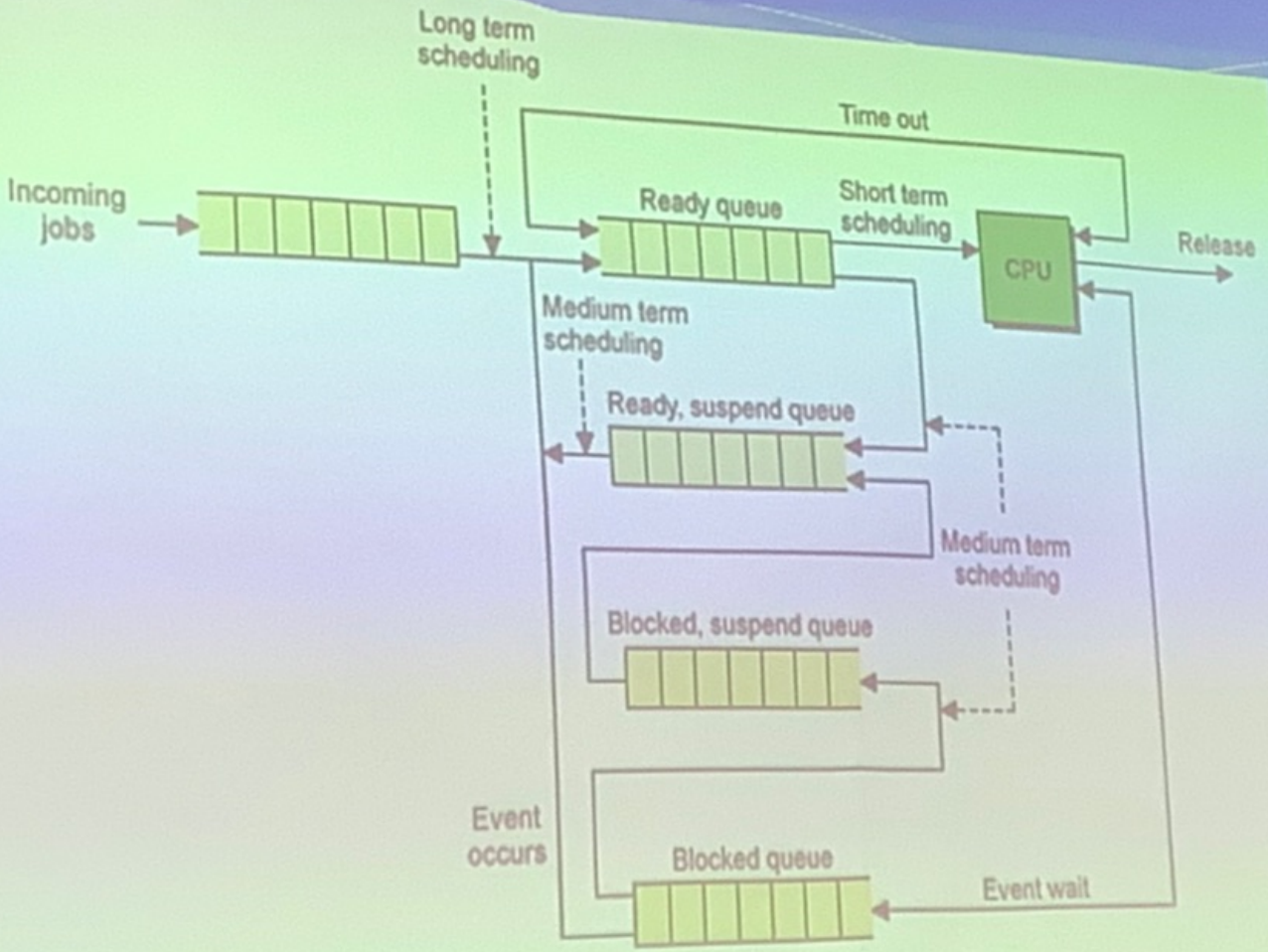
- CPU switch from this process to another process (switch the running process)


Chapter 2 Process Management
1. Process
1.2. Process Scheduling
Process swapping (Medium-term scheduler)
Main Memory
Secondary Memory
Process P1
Process P2
Process P3
Process P4
P1 goes for I/O wait
Process P1
Swap out
Swap in
Process P3
Process Pn
P1 comes back after I/O
Process P3
Swap out
Swap in
Process P1

1. Process

1.2. Process Scheduling

Medium-term scheduler



- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

Operations on process

- Process Creation
- Process Termination

Process Creation

- Process may create several new processes (CreateProcess(), fork())
  - The creating process: parent-process
  - The new process: child-process
- Child-process may create new process ⇒ Tree of process
- Resource allocation
  - Child process receive resource from the OS
  - Child process receive resource from parent-process
    - All of the resource
    - a subset of the resources of the parent process (to prevent process from creating too many child process)
- When a process creates a new process, two possibilities exist in terms of execution:
  - The parent continues to execute concurrently with its children
  - The parent waits until some or all of its children have terminated

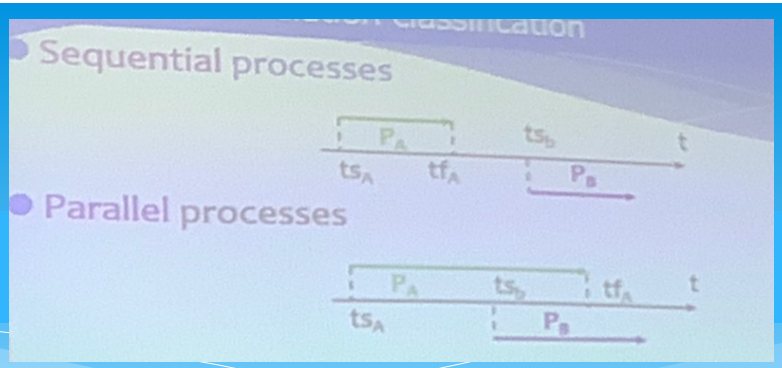
Process Termination

- Finishes executing its final statement and asks the OS to delete (exit)
  - Return data to parent process
  - Allocated resources are returned to the system
- Parent process may terminate the execution of child process
  - Parent must know child process's id ⇒ child process has to send its id to parent process when created
  - Use the system call (abort)
- Parent process terminate child process when
  - The child has exceeded its usage of some of the resources
  - The task assigned to the child is no longer required
  - The parent is exiting, and the operating system does not allow a child to continue
  - ⇒Cascading termination. Example: *turn off computer*

Some functions with process in WIN32 API

- CreateProcess(. . .)
  - LPCTSTR Name of the execution program
  - LPTSTR Command line paramenter
  - LPSECURITY_ATTRIBUTES A pointer to process **security attribute** structure
  - LPSECURITY_ATTRIBUTES A pointer to thread **security attribute** structure
  - BOOL allow process inherit devices (TRUE/FALSE)
  - DWORD process creation flag (Example: CREATE_NEW_CONSOLE)
  - LPVOID Pointer to environment block
  - LPCTSTR full path to program
  - LPSTARTUPINFO info structure for new process
  - LPPROCESS_INFORMATION Information of new process
- *TerminateProcess(HANDLE hProcess, UINT uExitCode)*
  - *hProcess* A handle to the process to be terminated
  - *uExitCode process termination code*
- *WaitForSingleObject(HANDLE hHandle, DWORD dwMs)*
  - *hHandle Object handle*
  - *dwMs Waiting time (INFINITE)*

- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

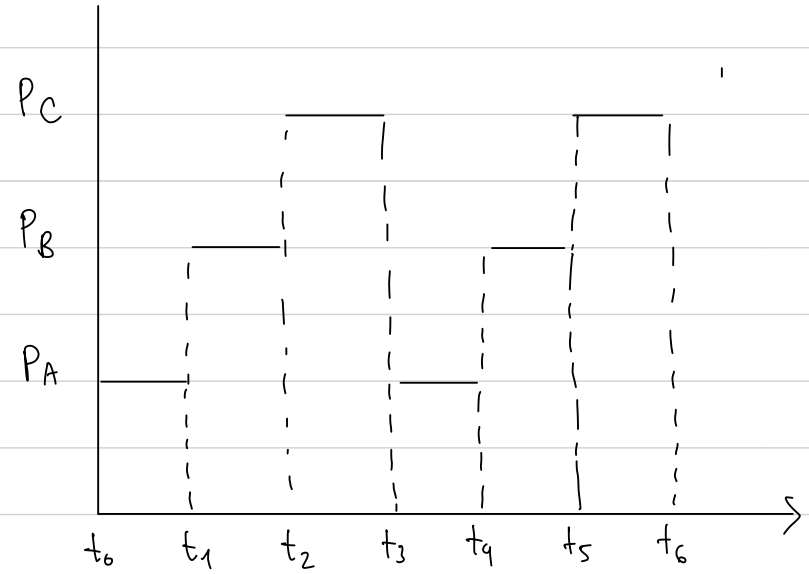


Processes' relation classification

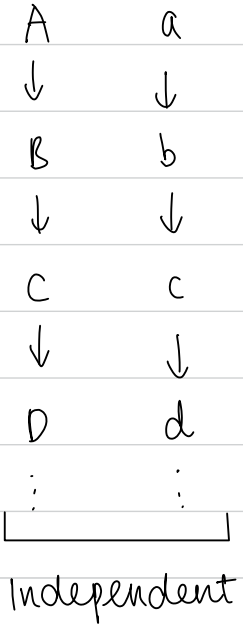
- Sequential processes
  - The start point of one process lie after the finish point of the other process
- Parallel processes
  - The start point of one process lie between the start and finish point of the other process
  - Independence: Not affect or affected by other running process in the system
  - Cooperation: affect or affected by other running process in the system
  - Cooperation in order to
    - Information sharing
    - Computation speedup
    - Modularity
    - Increase the convenience
  - Process collaborate require mechanism that allow process to
    - Communicate with one another
    - Synchronize their actions

Parallel vs. Concurrent ?
(song song) (đồng hành)

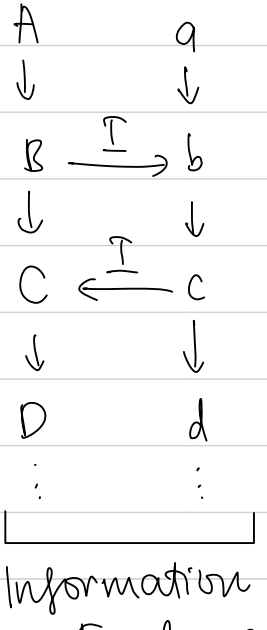
↳ If the process switching time is very small, it may makes users think as if multiple program are running simultaneously.
⇒ Concurrent



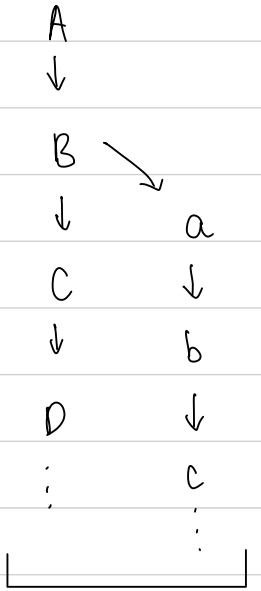
↳ At the same time, 1 process is served by 1 processor
⇒ Parallel



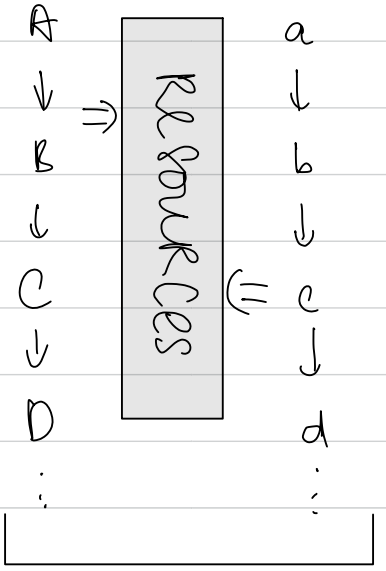
Independent



Information Exchange

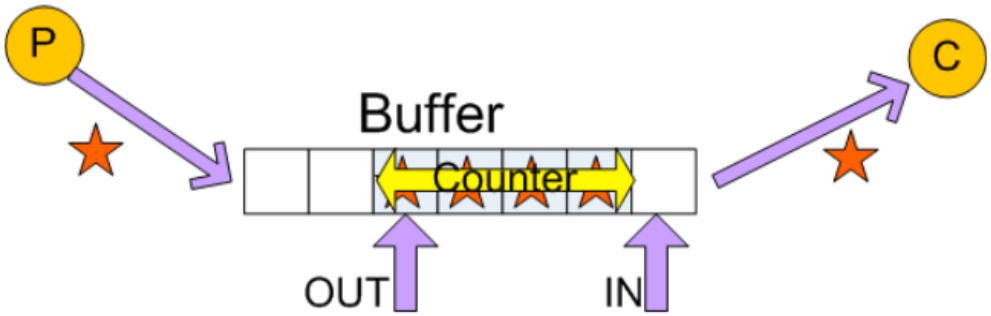


Parent - child



Equal
(shared resources)

Producer - Consumer Problem I



- The system includes 2 processes
  - Producer creates product
  - Consumer consumes created product
- Application
  - Print program (producer) creates characters that are consumed by printer driver (consumer)
  - Compiler (producer) creates assembly code, Assembler (consumer/producer) consumes assembly code and generate object module which is consumed by the exec/loader (consumer)

Producer - Consumer Problem II

- *Producer* and *Consumer* work simultaneously
  Use the sharing **Buffer** which store product filled in by producer and taken out by consumer
  - IN Next empty place in buffer
  - OUT First filled place in the buffer.
  - Counter Number of products in the buffer
- Producer and Consumer must be synchronized
  - Consumer does not try to consume a product that was not created
- Unlimited size buffer
  When buffer is empty, Consumer need to wait
  Producer can put product into buffer without waiting
- Limited size buffer
  - When Buffer is empty, *Consumer* need to wait
  - *Producer* need to wait if the Buffer is full

Producer - Consumer Problem II

Producer

```
while(1) {
    /*produce an item in nextProduced*/
    while (Counter == BUFFER_SIZE) ; /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}
```

only go to here if this while's condition is no longer satisfied (counter < buffer_size)

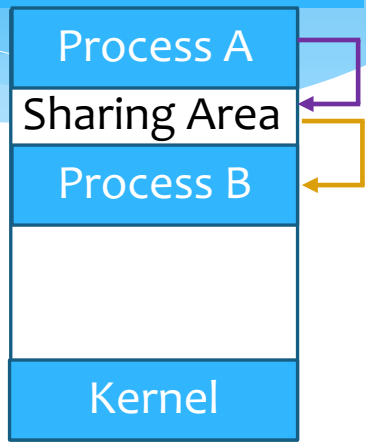
Consumer

```
while(1){
    while(Counter == 0) ; /*do nothing*/
    nextConsumed = Buffer[OUT];
    OUT =(OUT + 1) % BUFFER_SIZE;
    Counter--; /*consume the item in nextConsumed*/
}
```

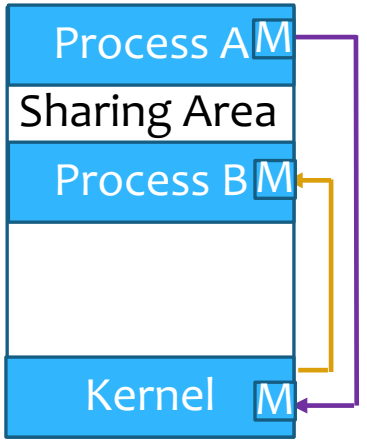
- Notion of process
- Process Scheduling
- Operations on process
- Process cooperation
- Inter-process communication

Communicate between process

- Memory sharing model
  - Process share a common memory area
  - Implementation code are written explicitly by application programmer
  - Example: Producer-Consumer problem



- Inter-process communication model
  - A mechanism allow processes to communicate and synchronize
  - Often used in distributed system where processes lie on different computers (chat)
  - Guaranteed by message passing system



Message passing system

- Allow processes to communicate without sharing variable
- Require two basic operations
  - Send (msg) msg has fixed or variable size
  - Receive (msg)
- If 2 processes P and Q want to communicate, they need to
  - Establish a communication link (physical/logical) between them
  - Exchange messages via operations: send/receive

Direct Communication

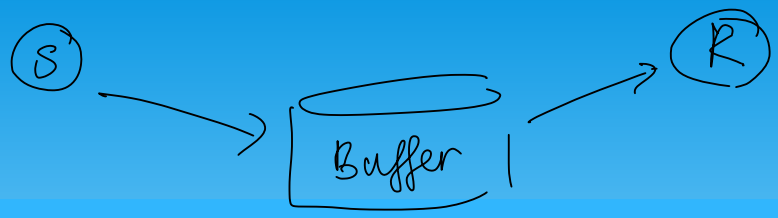
- Each process that wants to communicate must explicitly name the recipient or sender of the communication
  - send (P, message) – Send a message to process P
  - receive(Q, message) – Receive a message from process Q
- The properties of a communication link
  - A link is established automatically
  - A link is associated with exactly two processes
  - Exactly one link exists between each pair of processes
  - Link can be one direction but often two directions

Indirect Communication

- Messages are sent to and received from mailboxes, or ports
- Each mailbox has a unique identification
  - Two processes can communicate only if they share a mailbox
- Communication link's properties
  - Established only if both members of the pair have a shared mailbox
  - A link may be associated with more than two processes
  - Each pair of communicating processes may have many links
    - Each link corresponding to one mailbox
  - A link can be either one or two direction
- Operations:
  - Create a new mailbox
  - Send and receive messages through the mailbox
    - send(A, msg): send a msg to mailbox A
    - receive(A, msg): Receuve a msg from mailbox A
  - Delete a mailbox

Synchronization

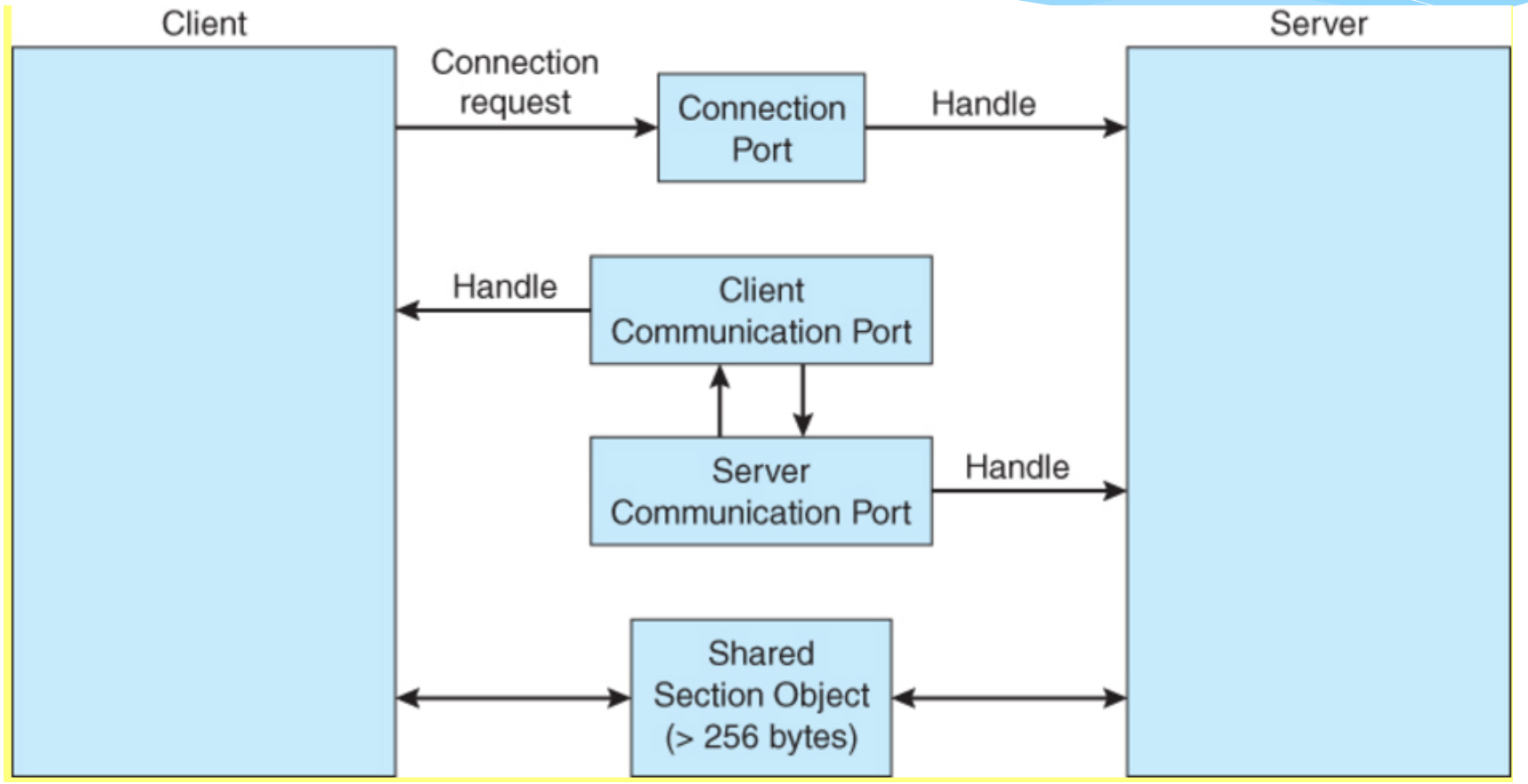
- Message passing may be either blocking or nonblocking-
  - Blocking synchronous communication
  - Non-blocking asynchronous communication
- send() and receive() can be blocking or non-blocking
  - Blocking send sending process is blocked until the message is received by the receiving process or by the mailbox
  - Non-blocking send The sending process sends the message and resumes operation
  - Blocking receive The receiver blocks until a message is available
  - Non-blocking receive The receiver retrieves either a valid message or a null



Buffering

- Messages exchanged by communicating processes reside in a temporary queue
- A queue can be implemented in three ways
  - Zero capacity: maximum length 0 => the link cannot have any messages waiting in it
    - sender must block until the recipient receives the message
  - Bounded capacity
    - Queue has finite length n ⇒ store at most n messages
    - If the queue is not full, message is placed in the queue and the sender can continue execution without waiting
    - If the link is full, the sender must block until space is available in the queue
  - Unbound capacity
    - The sender never blocks

Windows XP Message Passing



Communication in Client - Server Systems with Sockets

- Socket is defined as an endpoint for communication,
  - Each process has one socket
- Made up of an IP address and a port number. E.g.: 161.25.19.8:1625
  - IP Address: Computer address in the network
  - Port: identifies a specific process
- Types of sockets
  - Stream Socket: Based on TCP/IP protocol → Reliable data transfer
  - Datagram Socket: based on UDP/IP protocol → Unreliable data transfer
- Win32 API: Winsock
  - Windows Sockets Application Programming Interface

Winsock API 32 functions

socket()Tạo socket truyền dữ liệu
bind() Định danh cho socket vừa tạo (gán cho một cổng)
listen() Lắng nghe một kết nối
accept() Chấp nhận một kết nối
connect() kết nối với server.
send() Gửi dữ liệu với stream socket.
sendto() Gửi dữ liệu với datagram socket.
receive() Nhận dữ liệu với stream socket.
recvfrom() Nhận dữ liệu với datagram socket.
closesocket() Kết thúc một socket đã tồn tại.
.....

Homework

Use Winsock to make a Client-Server program

Chat program.

.....

① Process

② Thread

③ CPU scheduling

④ Critical resource and process synchronization

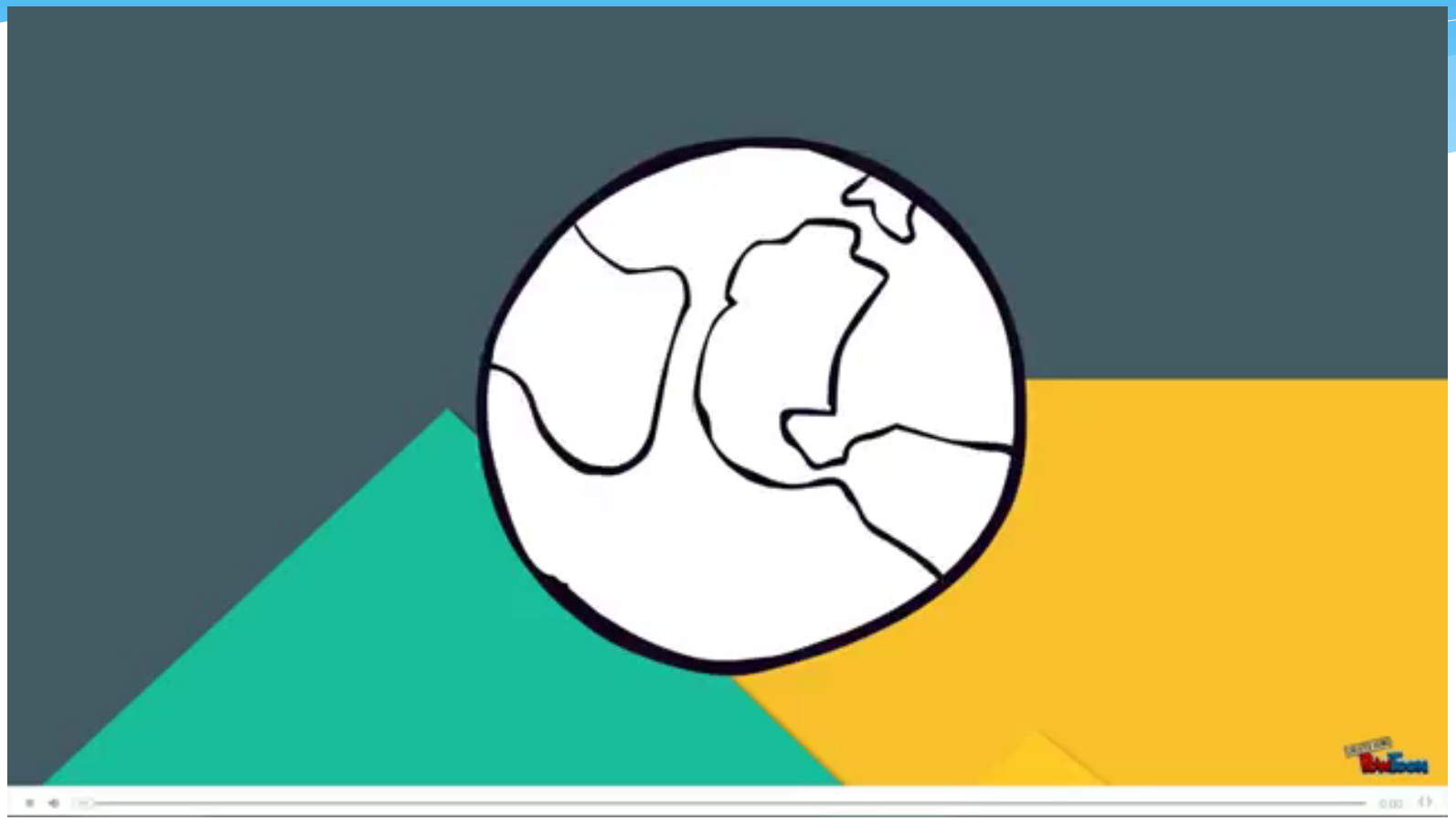
⑤ Deadlock and solutions

- Introduction
- Multithreading model
- Thread implementation with Windows
- Multithreading problem

Chapter 2 Process Management

2. Thread

2.1. Introduction

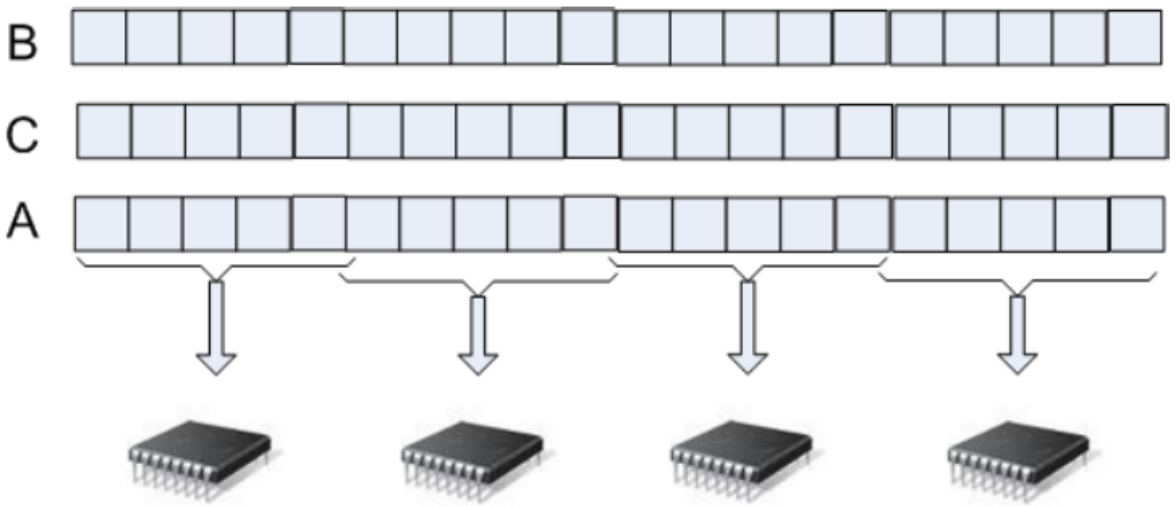


Example: Vector computation

- Large size vector computing

```
For (k = 0;k < n;k++) {
    a[k] = b[k]*c[k];
}
```

With multi processors system



Example: Chat

Process P

```
While (1) {
   ReadLine(Msg);
   Send(Q,Msg);
  Receive(Q,Msg);
   PrintLine(Msg);
}
```

Msg receiving problem

- Blocking Recieve
- Non-blocking Receive

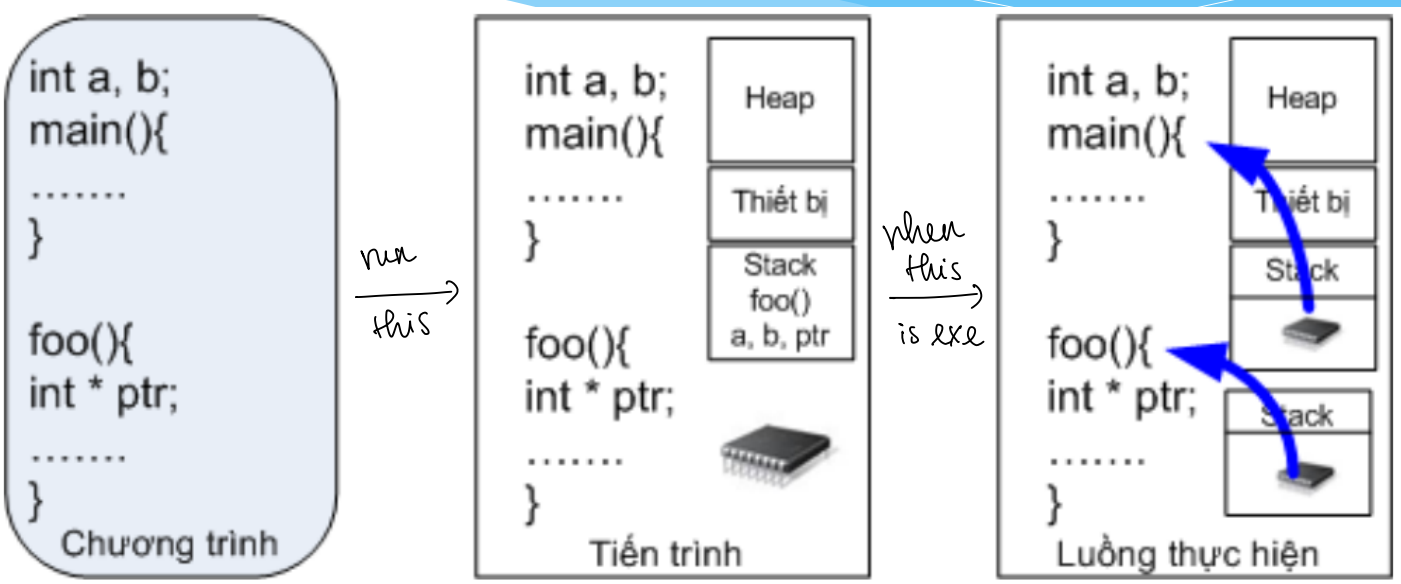
Solution

Concurrently perform Receive & Send

Process Q

```
While (1) {
   Receive(P,Msg);
   PrintLine(Msg);
   ReadLine(Msg);
  Send(P,Msg);
}
```

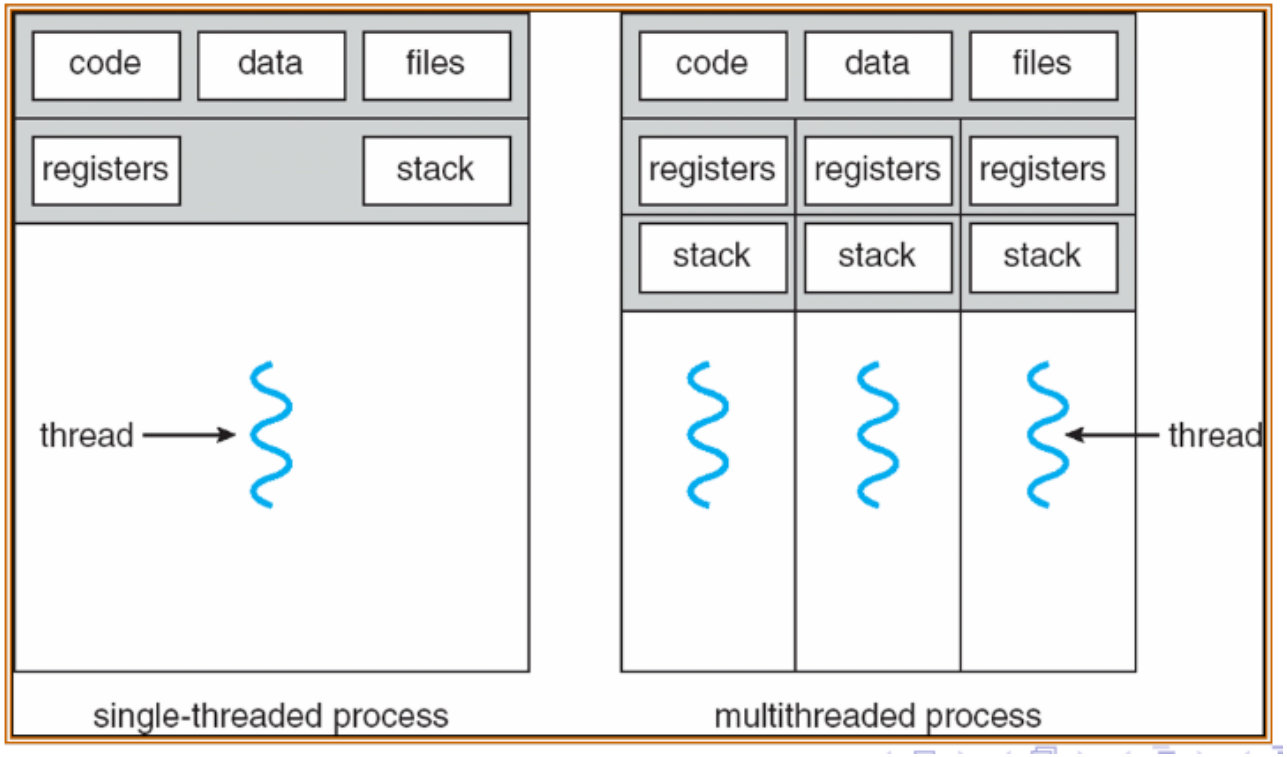
Program - Process - Thread



- Program: Sequence of instructions, variables,..
- Process: Running program: Stack, devices, processor,..
- Thread: A running program in process context (tiến trình con)
  - Multi-processor → Multi threads, each thread runs on one processor
    - Different in term of registers' values, stack's content

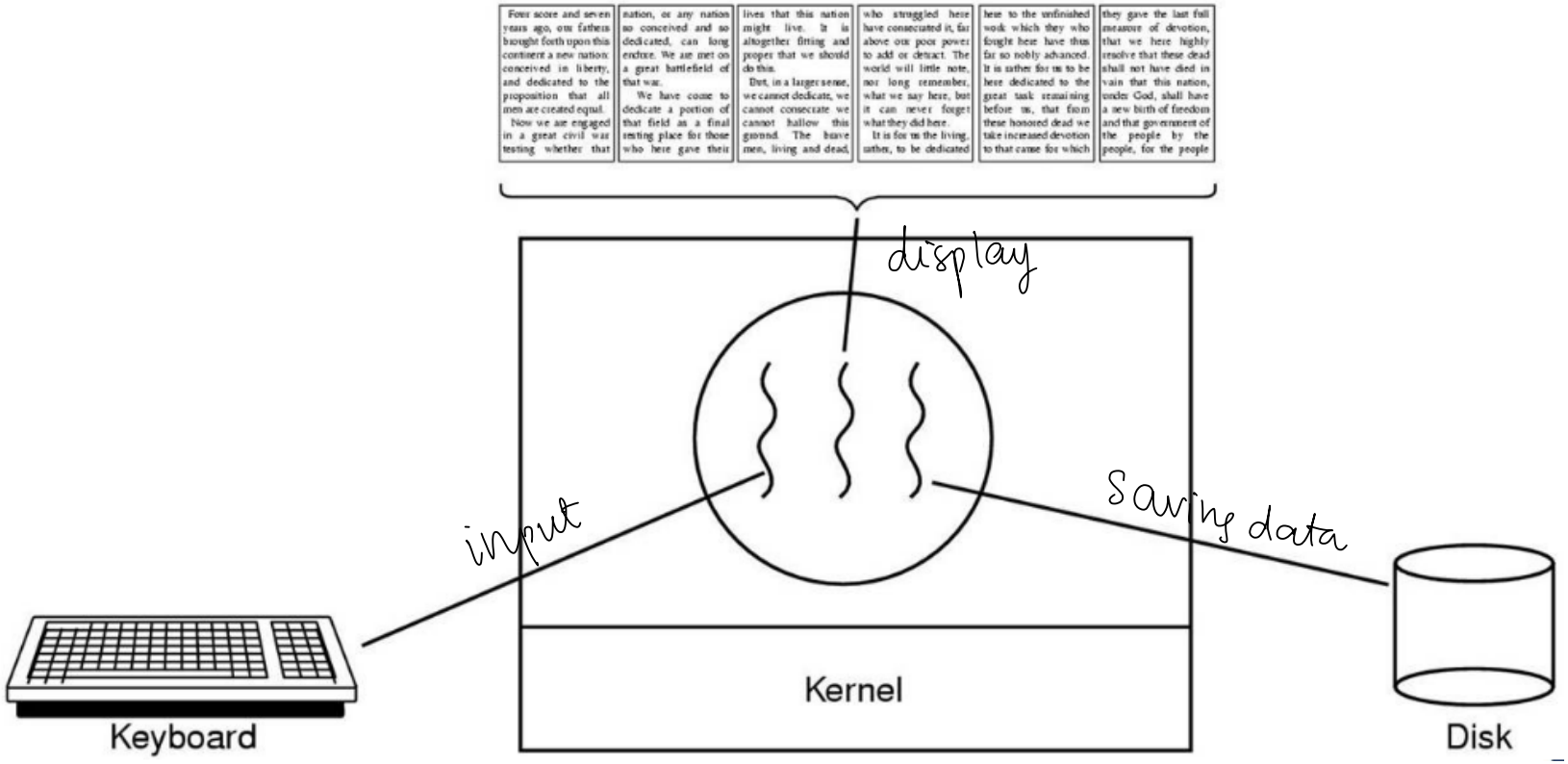
Single-threaded and multi-threaded process

- Traditional operating system(MS-DOS, UNIX)
  - Process has one controlling thread (heavyweight process)
- Modern operating system (Windows, Linux)
  - Process may have many threads
  - Perform many task at a single time

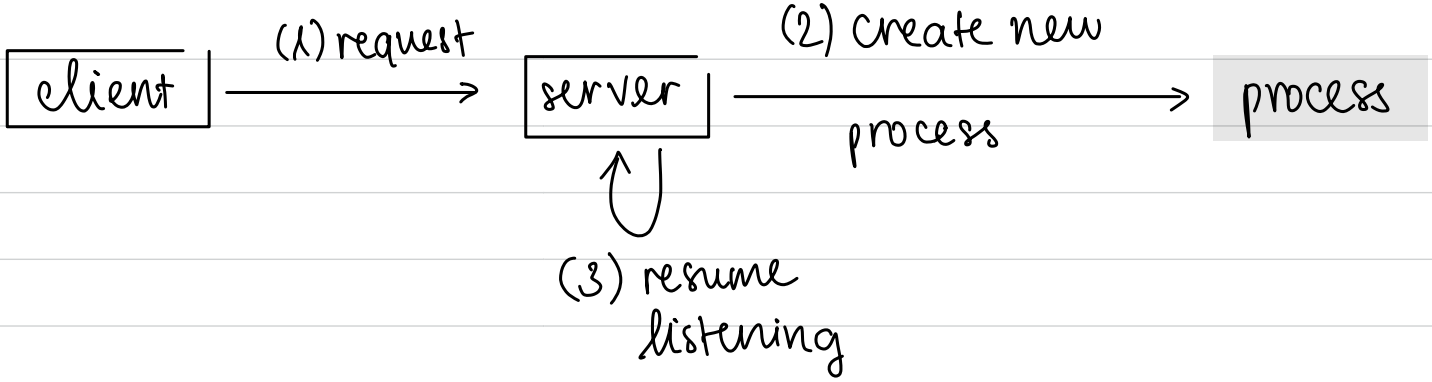


Example: Word processor (Tanenbaum 2001)

If your computer is suddenly powered off, rest assured that your doc is saved!



Initially we have:



there is no need to create a new process for each new request

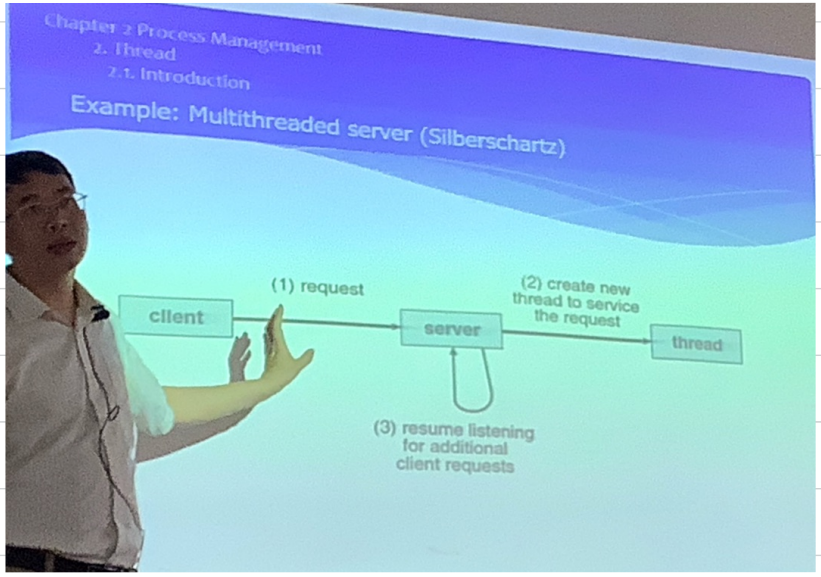
↓

there are requests that are identical

↓

✓ Create a new thread instead → faster!

in the same process



Notion of thread

- Basic CPU using unit, consists of
  - Thread ID
  - Program Counter
  - Registers
  - Stack space
- Sharing between threads in the same process
  - Code segment
  - Data segment (global objects)
  - Other OS's resources (opening file…)
- Thread can run same code segment with different context (*Register set, program counter, stack*)
- LWP: Lightweight Process
- A process has at least one thread

Distinguishing between process and thread

Process	Thread
Has code/data/heap and other segments	No separating code or heap segment
Has at least one thread	Not stand alone but must be inside a process
Threads in the same process share code/data/heap, devices but have separated stack and registers	Many threads can exist at the same time in each process. First thread is the main thread and own process's stack
Create and switch process operations are expansive	Create and switch thread operations are inexpensive
Good protection due to own address space	Common address space, need to protect
When process terminated, resources are returned and threads have to terminated	Thread terminated, its stack is returned

Why thread creation is cheaper than process creation?
↳ a thread control block has less field than PCB

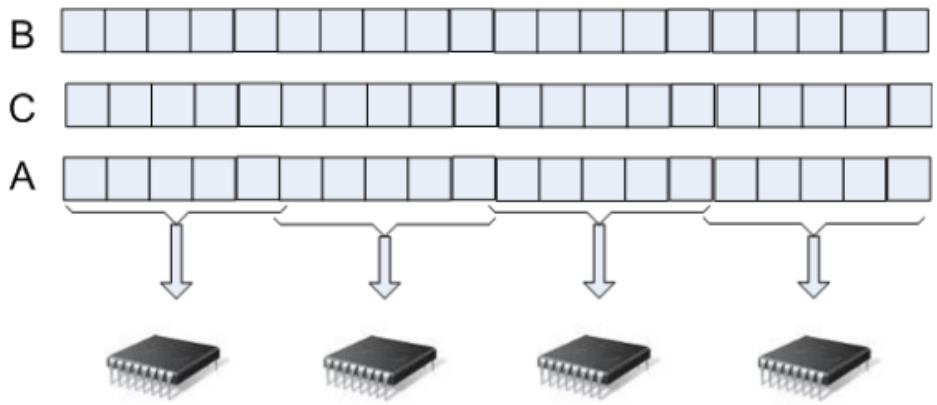
Benefits

- Responsiveness
  - allow a program to continue running even if part of it is blocked or is performing a long operation
  - Example: A multi-threaded Web browser
    - One thread interactives with user
    - One thread downloads data
- Resource sharing
  - threads share the memory and the resources of the process
    - Good for parallel algorithms (share data structures)
    - Threads communicate via sharing memory
  - Allow application to have many threads act in the same address space
- Economy
  - Create, switch, terminate threads is less expensive than process
- Utilization of multiprocessor architectures
  - each thread may be running in parallel on a different processor.

Benefit of multithreading-> example

Vector computing

```
for (k = 0;k < n;k++) {
    a[k] = b[k]*c[k];
}
```

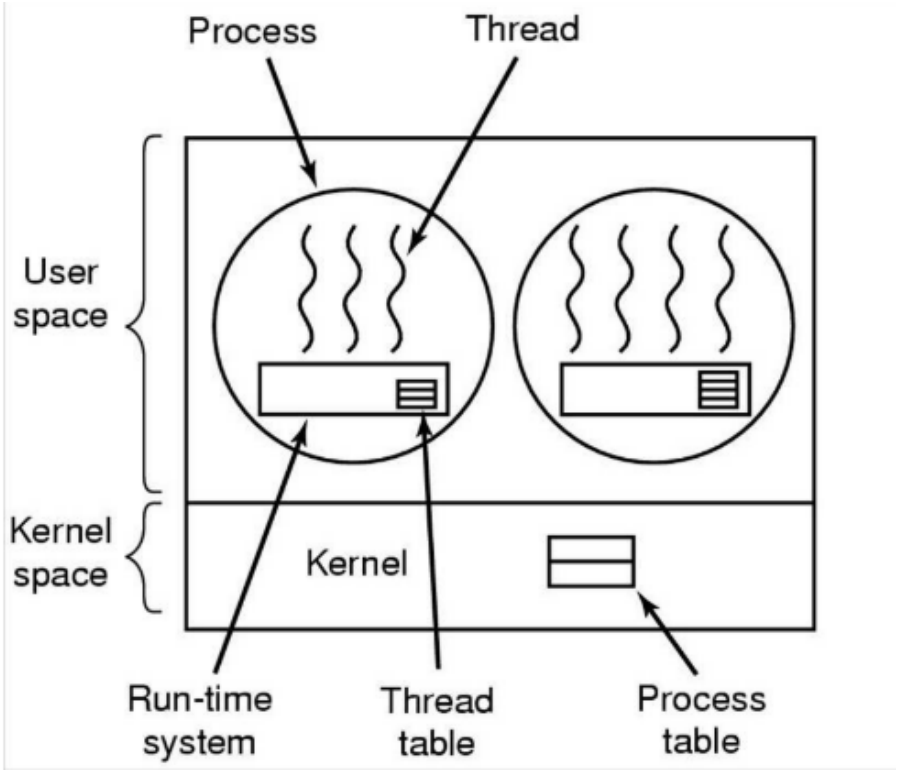


Multi-threading model

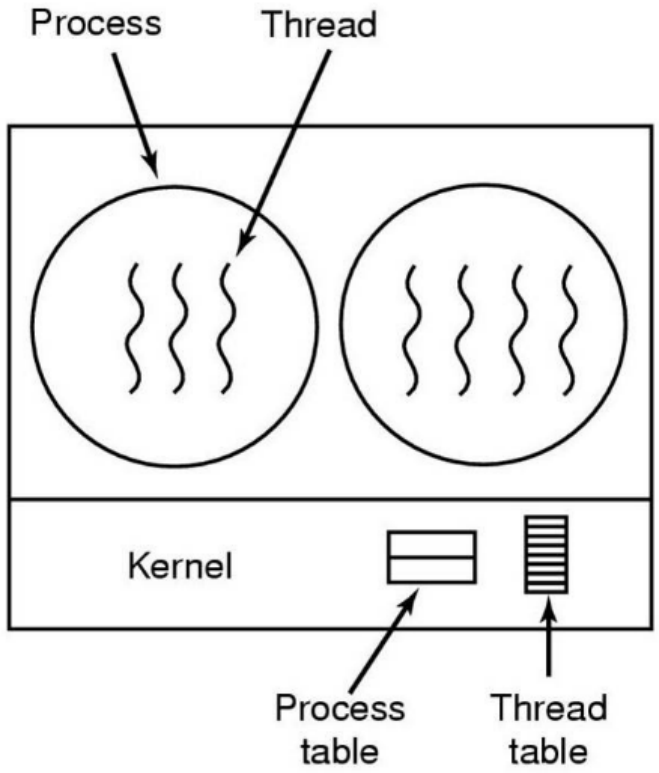
```
void fn(a,b)
for(k = a; k < b; k ++){
    a[k] = b[k] * c[k];
}
void main(){
   CreateThread(fn(0, n/4));
   CreateThread(fn(n/4, n/2));
   CreateThread(fn(n/2, 3n/4));
   CreateThread(fn(3n/4, n));
}
```

Thread implementation

User space



Kernel space



User -Level Threads

- Thread management is done by application
- Kernel does not know about thread existence
  - Process scheduling like a single unit
  - Each process is assigned with a single state
    - Ready, waiting, running,..
- User threads are supported above the kernel and are implemented by a thread library
  - Library support creation, scheduling and management..
- Advantage
  - Fast to create and manage
- Disadvantage
  - if a thread perform blocking system call , the entire process will be blocked ⇒ Cannot make use of multi-thread.

Kernel - Level threads

POSIX PThread
Win32 Thread

- Kernel keeps information of process and threads
- Threads management is performed by kernel
  - No thread management code inside application
  - Thread scheduling is done by kernel
- Disadvantage:
  - Slow in thread creation and management
- Advantage:
  - One thread perform system call (e.g. I/O request), other threads are not affected
  - In a multiprocessor environment, the kernel can schedule threads on different processors
- Operating system support kernel thread: Windows NT/2000/XP, Linux, OS/2,..

↳ A special class of thread that are NOT < user / kernel

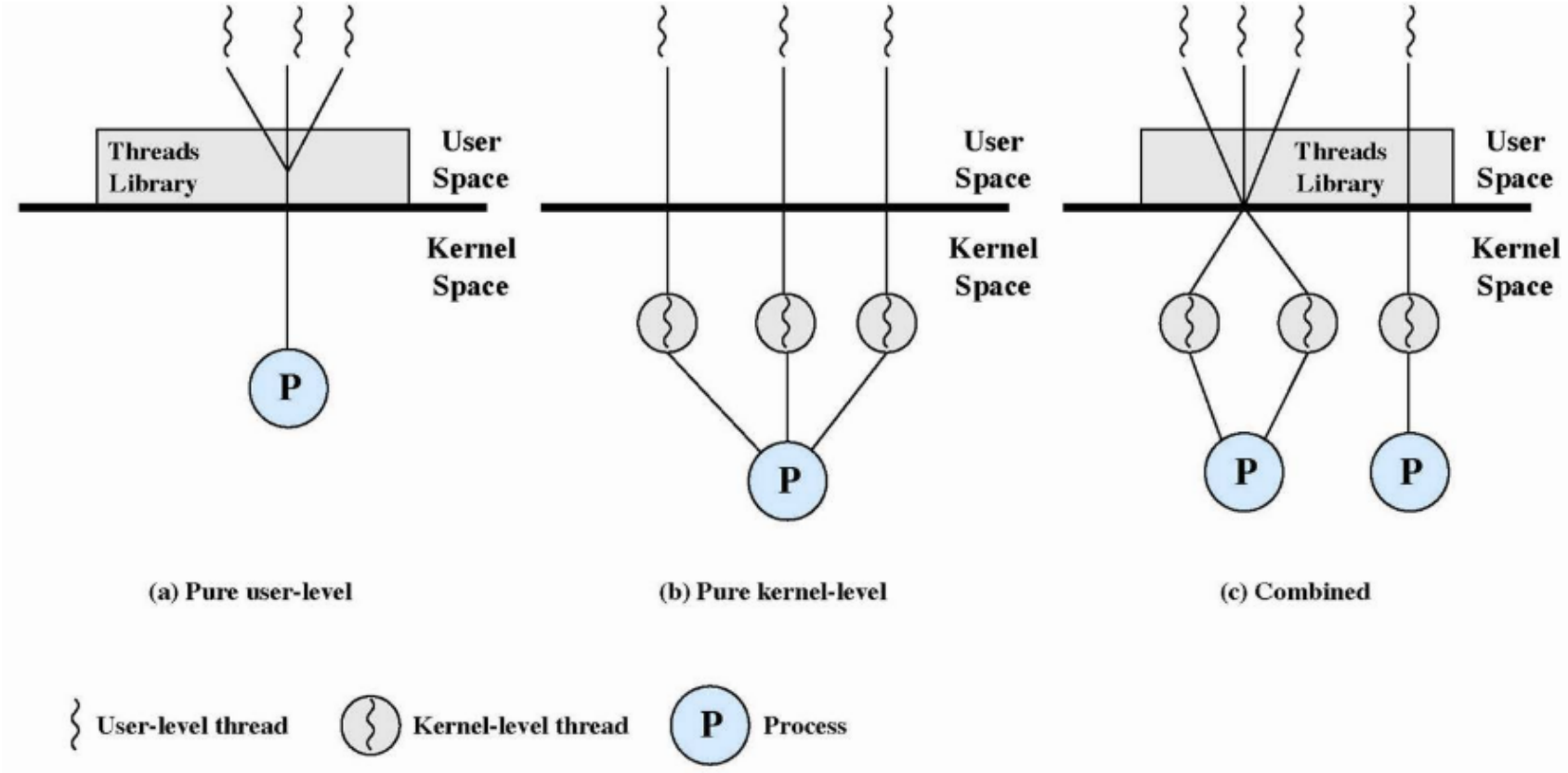
Java thread

- Introduction
- Multithreading model
- Thread implementation with Windows
- Multithreading problem

Introduction

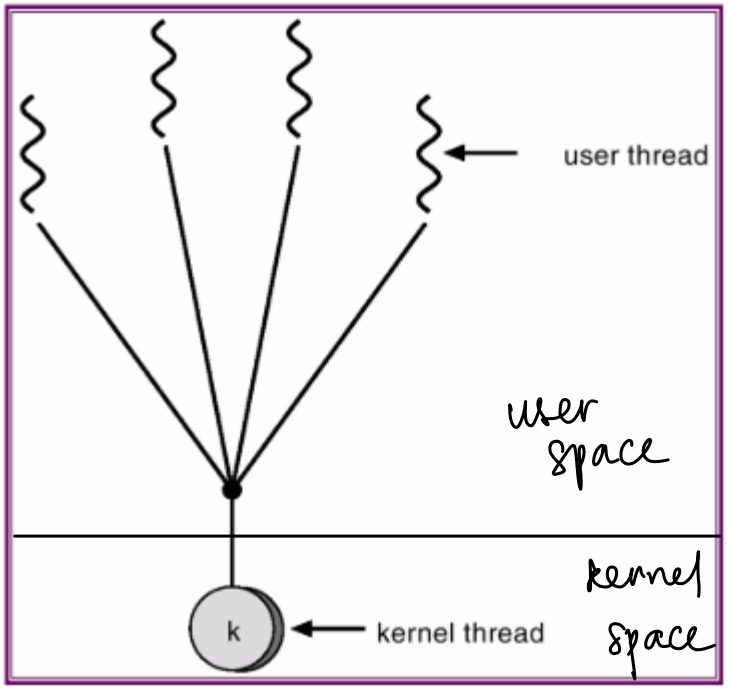
4 models total

- Many systems provide support for both user and kernel threads -> different multithreading models



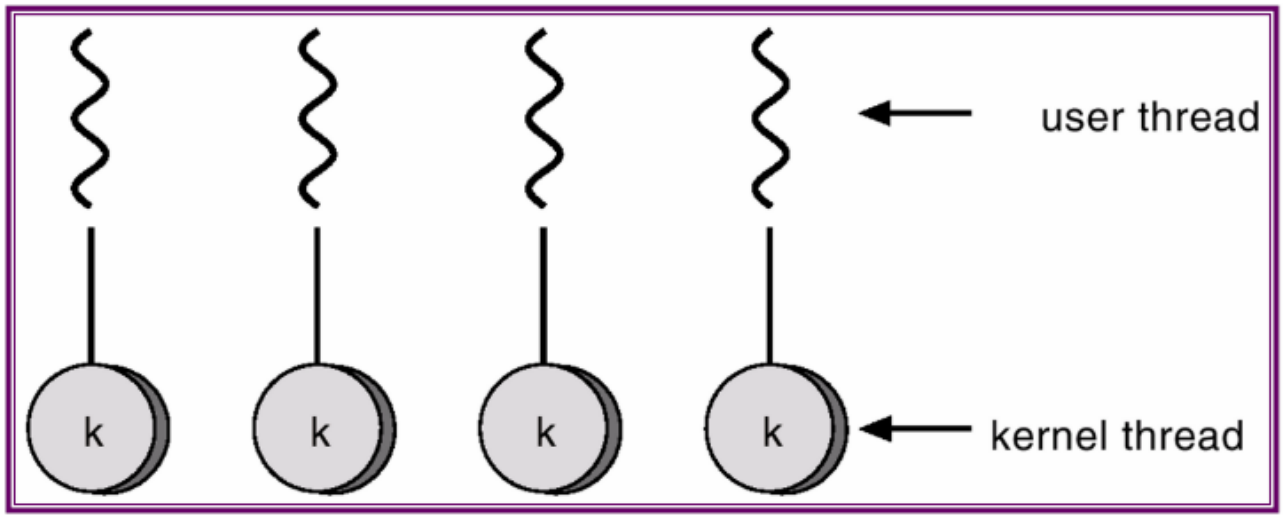
Many-to-One Model (OS ngày xưa)

- Maps many user-level threads to one kernel thread.
- Thread management is done in user space
  - Efficient
  - the entire process will block if a thread makes a blocking system call
  - multiple threads are unable to run in parallel on multi processors
- implemented on operating systems that do not support kernel threads use the many-to-one model



⊖ when 1 thread is I/O, the kernel thinks that the whole bundle is I/O and block all of them

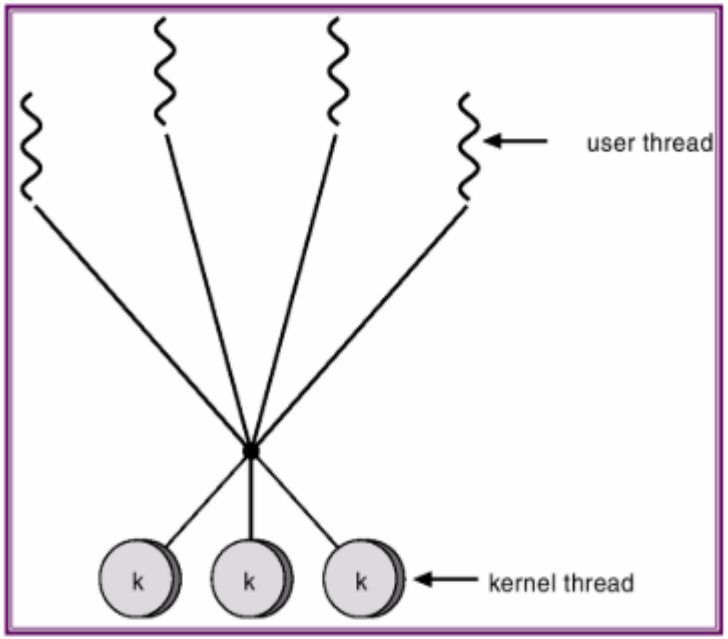
One-to-one Model (Windows)



- Maps each user thread to a kernel thread
- Allow another thread to run when a thread makes a blocking system call
- Creating a user thread requires creating the corresponding kernel thread
  - the overhead of creating kernel threads can burden the performance of an application
  - ⇒ restrict the number of threads supported by the system
- Implemented in Window NT/2000/XP

Many-to-Many Model (Linux)

- Multiplexes many user-level threads to a smaller or equal number of kernel threads
- Number of kernel threads: specific to either a particular application or a particular machine
  - E.g.: on multiprocessor -> more kernel thread than on uniprocessor
- Combine advantages of previous models
  - Developers can create as many user threads as necessary
  - kernel threads can run in parallel on a multiprocessor
  - when a thread performs a blocking system call, the kernel can schedule another thread for execution
- Supported in: UNIX



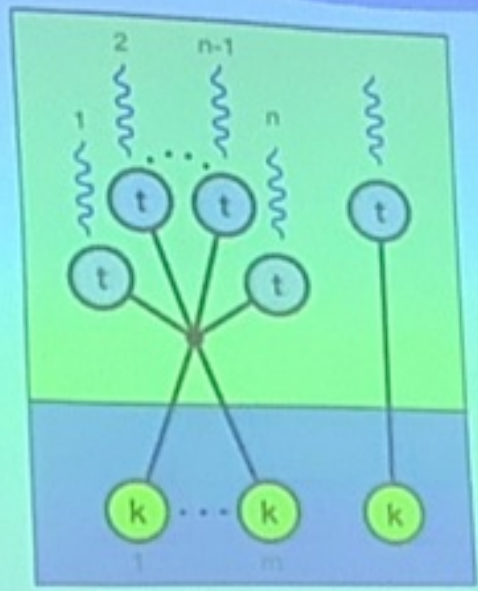
Chapter 2 Process Management
2. Thread
2.2.Multi-threading model

Two-level model

(trên OS thay đổi quản lý chuyển đổi)

- A variation of many to many
- Allow favor process with higher priority

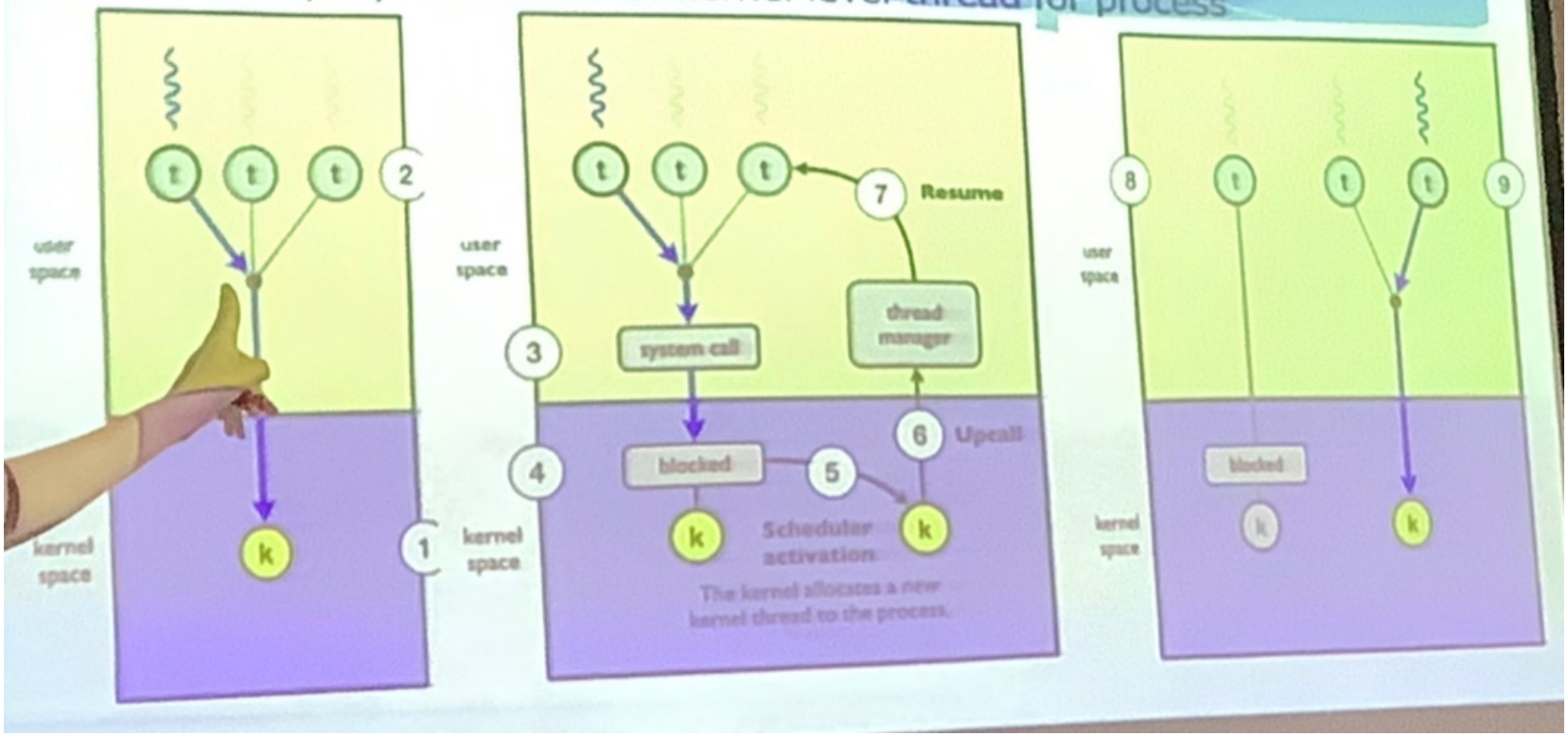
(1-1 → many-many)



will be converted onto 1-level if needed

scheduler activation

- A way for kernel to communicate with user level thread manager to maintain a proper number of kernel level thread for process



- Introduction
- Multithreading model
- Thread implementation with Windows
- Multithreading problem

Functions with thread in WIN32 API

- HANDLE CreateThread(. . .);
  - LPSECURITY_ATTRIBUTESlpThreadAttributes,
    ⇒A pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes
  - DWORD dwStackSize,
    ⇒The initial size of the stack, in bytes
  - LPTHREAD_START_ROUTINE lpStartAddress,
    ⇒ pointer to the application-defined function to be executed by the thread
  - LPVOID lpParameter,
    ⇒A pointer to a variable to be passed to the thread
  - DWORD dwCreationFlags,
    ⇒The flags that control the creation of the thread
    - CREATE_SUSPENDED : thread is created in a suspended state
    - 0: thread runs immediately after creation
  - LPDWORD lpThreadId
    ⇒pointer to a variable that receives the thread identifier
- Return value: handle to the new thread or NULL if the function fails

Example

```
#include <windows.h>
#include <stdio.h>
void Routine(int *n){
printf("My argument is %d\n", &n);
}
int main(){
      int i, P[5]; DWORD Id;
       HANDLE hHandles[5];
       for (i=0;i < 5;i++) {
           P[i] = i;
           hHandles[i] = CreateThread(NULL,0,
                          (LPTHREAD_START_ROUTINE)Routine,&P[i],0,&Id);
           printf("Thread %d was created\n",Id);
       }
       for (i=0;i < 5;i++) WaitForSingleObject(hHandles[i],INFINITE);
    return 0;
}
```

① How many threads are there?

→ main() itself is a process

↳ requires at least 1 thread inside it

↳ ⑥ total

② where will the thread be saved?

↳ stack

③ where is the location of variable in process memory?

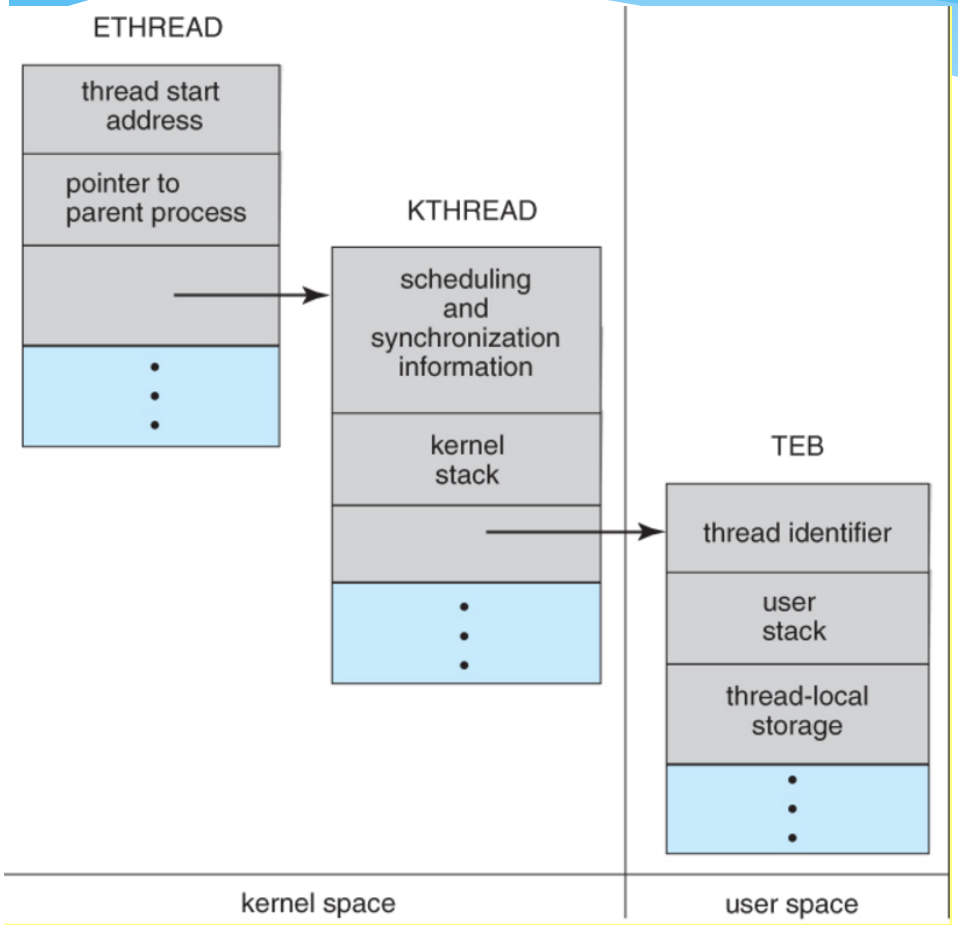
Stack segment

④ What is the output printed?

[compile & run multiple times to see the result. explain why]

Thread in Windows XP

kernel thread block > user thread blk
→ more explensive



Thread includes

- Thread ID
- Registers
- user stack used in user mode, kernel stack used in kernel mode.
- Separated memory area used by runtime and dynamic linked library

make call → API
to access this data structure

Executive thread block
Kernel thread block
Thread environment block

- Introduction
- Multithreading model
- Thread implementation with Windows
- Multithreading problem

Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
void T1(){
    while(1){ x = y + 1; printf("%4d", x); }
}
void T2(){
    while(1){ y = 2; y = y * 2; }
}
int main(){
    HANDLE h1, h2; DWORD Id;
    h1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T1,NULL,0,&Id);
    h2=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T2,NULL,0,&Id);
    WaitForSingleObject(h1,INFINITE);
    WaitForSingleObject(h2,INFINITE);
    return 0;
}
```

① How many threads are there?

3 threads (main() h1 h2)

② Where are x, y saved?

Global variable:

Data segment of program

③ Where are h1, h2 saved?

Stack of the main() thread

④ What'll be printed?

2 3 5

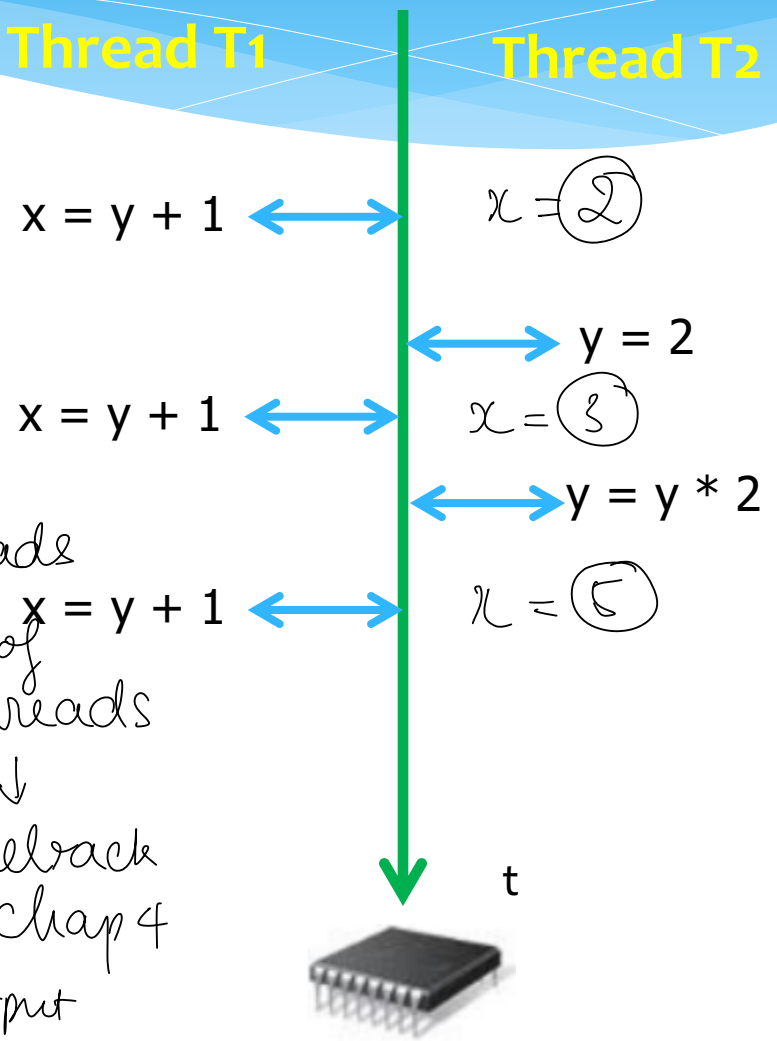
Why? explain below ↓

Limited processor

Decision of the scheduler are different on different time, different architecture

Explanation

Shared int y = 1	
Thread T_1	**Thread T_2**
x ←y+ 1	y ←2 y ← y * 2
x = ?	



Solution:

① Synchronize data for 2 threads

② " order of statements of 2 threads

The result of parallel running threads depends on the order of accessing the sharing variable

↓

Comeback in Chap 4

Goal: to obtain the consistent output

① Process
② Thread
③ CPU scheduling
④ Critical resource and process synchronization
⑤ Deadlock and solutions

- Basic Concepts
- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling

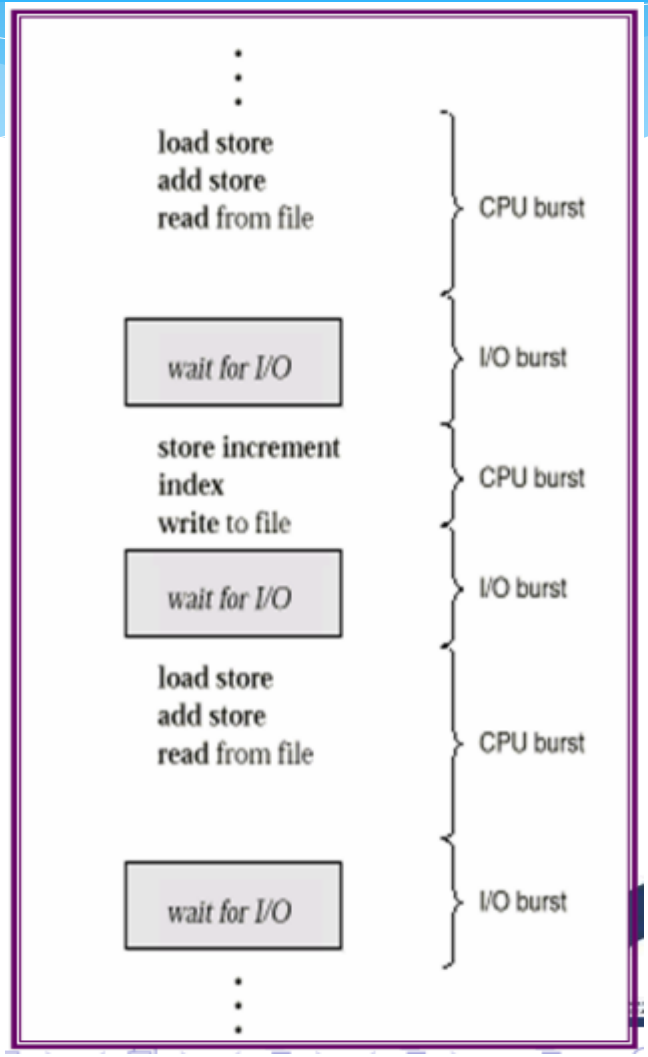
Introduction

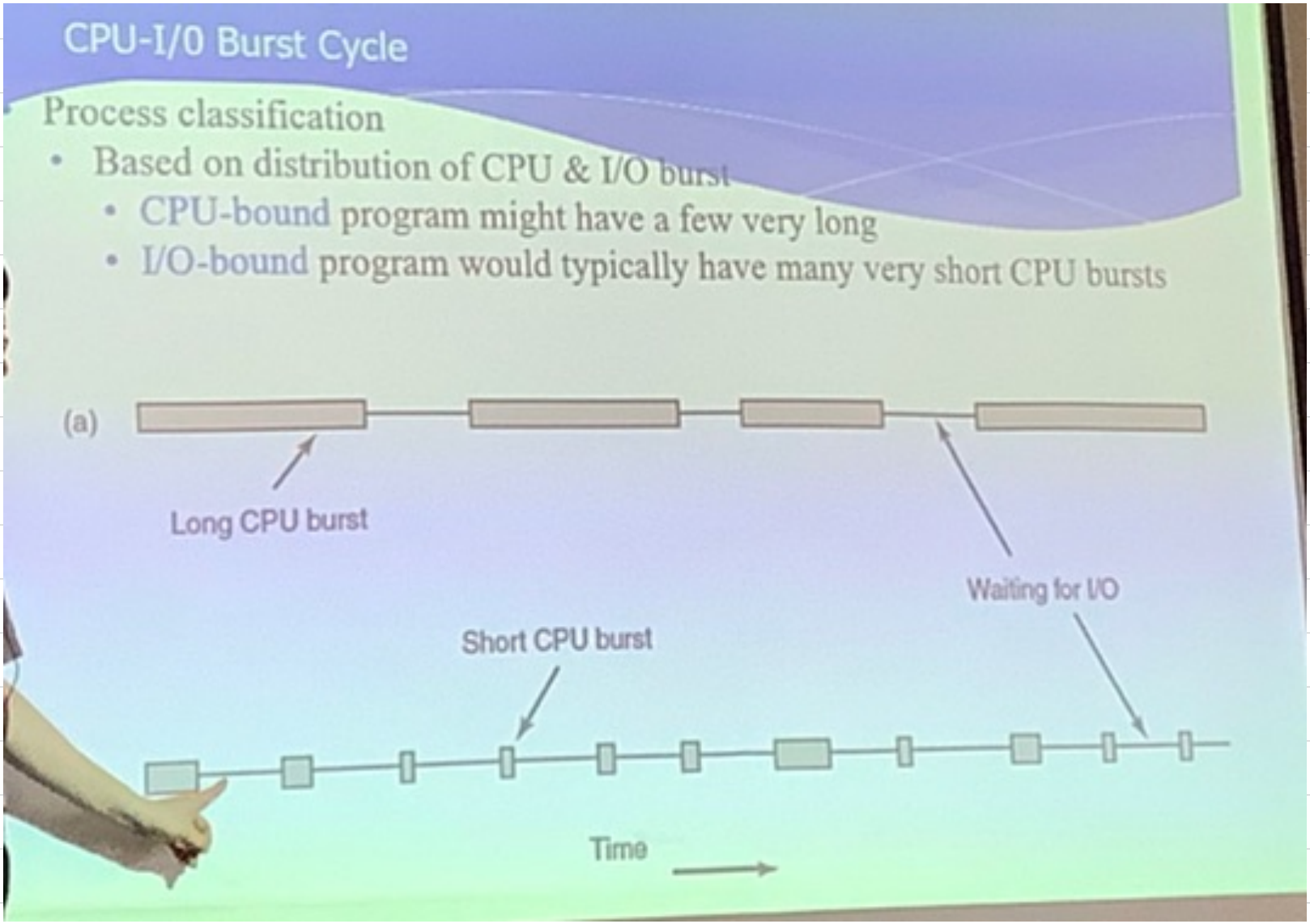
- System has one *processor* → Only one process is running at a time
- Process is executed until it must wait, typically for the completion of some I/O request
  - Simple system: CPU is not be used ⇒ Waste CPU time
  - Multiprogramming system: try to use this time productively, give CPU for another process
    - Several ready processes are kept inside memory at one time
    - When one process has to wait, OS takes the CPU away and gives the CPU to another process
- Scheduling is a fundamental operating-system function
  - Switch CPU between process → exploit the system more efficiently

(Better algorithm)

CPU-I/0 Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait
  - begins with a CPU burst
  - followed by an I/O burst
  - CPU burst → I/O burst → CPU burst → I/O burst → . . .
  - End: the last CPU burst will end with a system request to terminate execution
- Process classification
  - Based on distribution of CPU & I/O burst
    - CPU-bound program might have a few very long
    - I/O-bound program would typically have many very short CPU bursts
  - help us select an appropriate CPU-scheduling algorithm




CPU-I/0 Burst Cycle
Process classification
• Based on distribution of CPU & I/O burst
• CPU-bound program might have a few very long
• I/O-bound program would typically have many very short CPU bursts
(a)
Long CPU burst
Waiting for I/O
Short CPU burst
Time

CPU Scheduler

- CPU idle -> select one process from ready queue to be executed
  - Process in ready queue
    - FIFO queue, Priority queue, Simple linked list . . .
- CPU scheduling decisions may take place when
  1) Process switches from the running state to the waiting state (I/O request)
  2) Process switches from the running state to the ready state (out of CPU usage time→ time interrupt)
  3) Process switches from the waiting state to the ready state (e.g. completion of I/O)
  4) Process terminates
- Note
  - Case 1&4 ⇒ non-preemptive scheduling scheme
  - Other cases ⇒ preemptive scheduling scheme

trưng dụng (i.e. cảnh sát lấy xe dân để bắt trộm)

Preemptive and non-preemptive Scheduling

- Non-preemptive scheduling
  - Process keeps the CPU until it releases the CPU either by
    - terminating
    - switching to the waiting state
    - does not require the special hardware (timer)
  - Example: DOS, Win 3.1, Macintosh
- Non-preemptive scheduling
  - Process only allowed to run in specified period
    - End of period, time interrupt appear, dispatcher is invoked to decide to resume process or select another process
  - Protect CPU from “CPU hungry" processes
  - Sharing data problem
    - Process 1 updating data and the CPU is taken
    - Process 2 executed and read data which is not completely update
  - Example: Multiprogramming OS WinNT, UNIX

- Basic Concepts
- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling

Scheduling Criteria I

- CPU utilization
  - keep the CPU as busy as possible
  - should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system)
- Throughput
  - number of processes completed per time unit
    - Long process: 1 process/hour
    - Short processes: 10 processes/second
- Turnaround time
  - Interval from the time of submission of a process to the time of completion
  - Can be the sum of
    - periods spent waiting to get into memory
    - waiting in the ready queue
    - executing on the CPU
    - doing I/O

Scheduling Criteria II

- Waiting time
  - sum of the periods spent waiting in the ready queue
  - CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue
- Response time
  - time from the submission of a request until the first response is produced
    - process can produce some output fairly early
    - continue computing new results while previous results are being output to the user

- Assumption: Consider one CPU burst (ms) per process
- Measure: Average waiting time

interactive sys

real-time sys

- Basic Concepts
- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling

To calculate average waiting time, it's best to draw Gantt chart

FCFS: First Come, First Served

- Rule: process that requests the CPU first is allocated the CPU first

  Process own CPU until it terminates or blocks for I/O request

- Example

Process	Burst Time
P_1	24
P_2	3
P_3	3

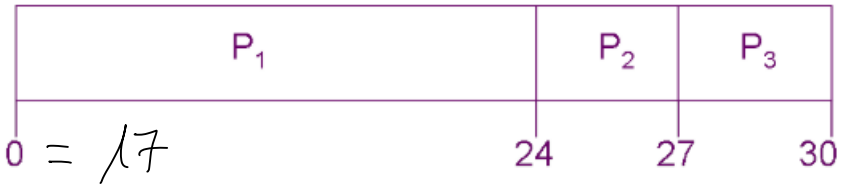
Average waiting time:

P_1 : 0

P_2 : 24

P_3 : 27

$\Rightarrow \quad \bar{T}_{wait} = \frac{0+24+27}{3} = 17$



- Pros and cons
  - Simple, easy to implement
  - Short process has to wait like short process
    - If P1 executed last?

SJF: Shortest Job First

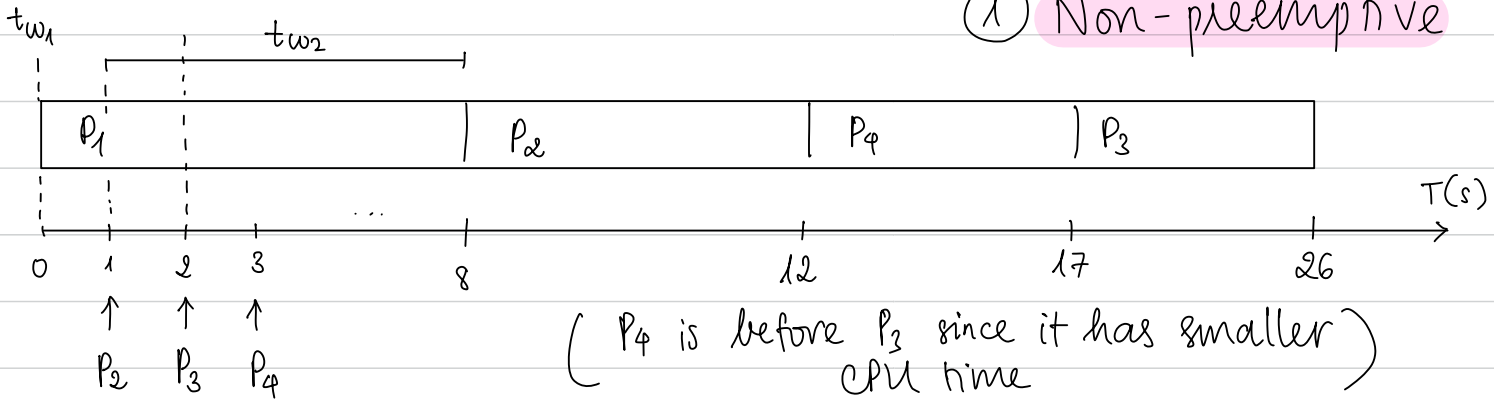
- Rule: associates with each process the length of the latter's next CPU burst
  - process that has the smallest next CPU burst
  - Two methods
    - Non-preemptive
    - preemptive (SRTF: Shortest Remaining Time First)
- Example

Average wait time?

Process	Burst Time	Arrival Time
P_1	8	0.0
P_2	4	1.0
P_3	9	2.0
P_4	5	3.0

- Pros and Cons
  - SJF (SRTF) is optimal: Average waiting time is minimum
  - It's not possible to predict the length of next CPU burst
    - Predict based on previous one

(1) Non-preemptive



(P_4 is before P_3 since it has smaller CPU time)

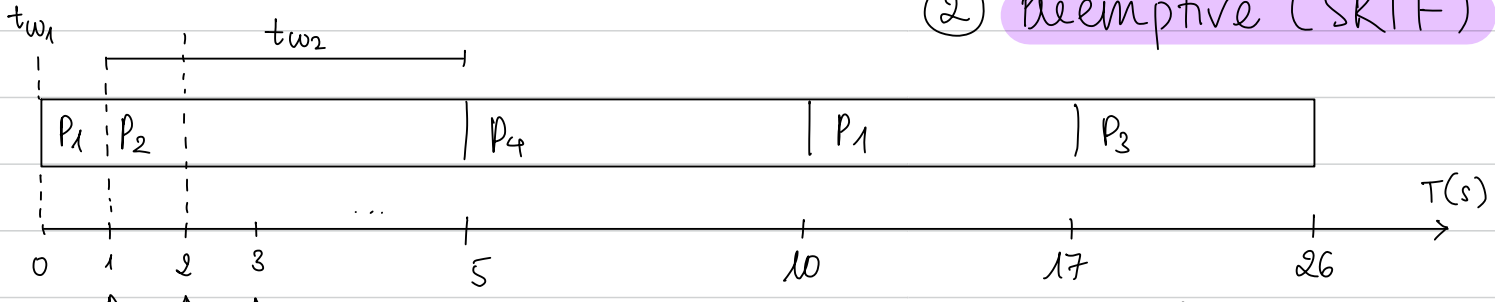
The waiting time:

$$\left.\begin{aligned} t_{w_1} &= 0 \\ t_{w_2} &= 7 \\ t_{w_3} &= 15 \\ t_{w_4} &= 9 \end{aligned}\right\} \rightarrow \bar{t}_w = \frac{31}{4}$$

※ Most common mistakes: Students dismiss arrival time and tread all process the same!

(optimal avg wait time)

② Preemptive (SRTF)



P_2 P_3 P_4

remain time

$P_1 = 8-1=7$ → run P_2

$P_2 = 4$

$t_{w_1} = 17-0-8 = 9$

$t_{w_2} = 0$

$t_{w_3} = 15$

$t_{w_4} = 2$

$\rightarrow \bar{T} = \frac{26}{4}$

※ When wait time is splitted over different intervals, we should use:

$$t_{turn\text{-}around} = t_{finish} - t_{arrival}$$

$$t_{waiting} = t_{turn\_around} - t_{compute}$$

Priority Scheduling

- Each process is attached with a priority value (a number)
  - CPU is allocated to the process with the highest priority
  - SJF: priority (p) is the inverse of the (predicted) next CPU burst
  - Two method
    - Non-preemptive
    - Preemptive
- Example

Priority

Process	Burst Time	~~Arrival Time~~
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

$(P_2 > P_5 > P_1 > P_3 > P_4)$

← smaller the number, higher priority

- "Starvation" problem: low-priority process has to wait indefinitely (even forever)
- Solution: increase priority by the time process wait in the system

Round Robin Scheduling

- Rule
  - Each process is given a time quantum (time slice) τ to be executed
  - When time's up processor is preemptive and process is placed in the last position of ready queue
  - If there are n process, longest waiting time is $(n-1)\tau$
- Example

Process	Burst Time
P_1	24
P_2	3
P_3	3

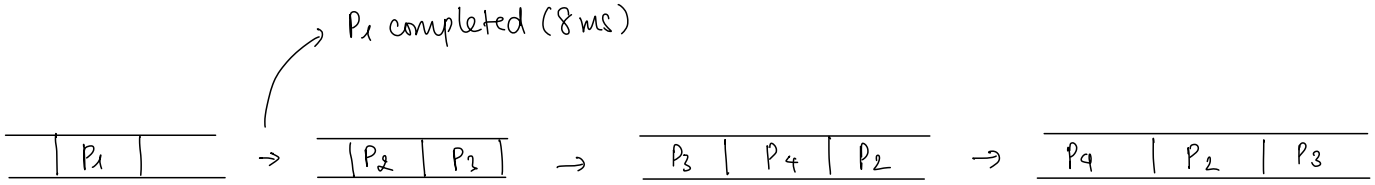
quantum τ = 4

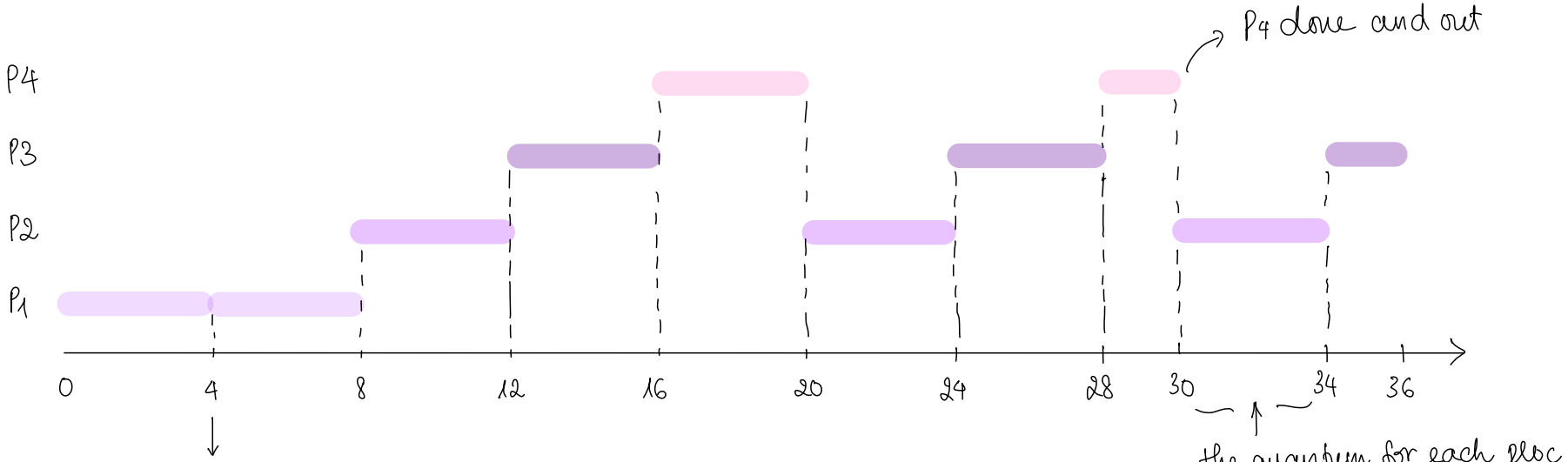
- Problem: select τ
  - τ large: FCFS
  - τ small: CPU is switched frequently
  - Commonly τ = 10-100ms

	CPU Burst	Arrival time
P_1	8	0
P_2	12	5
P_3	10	8
P_4	6	10

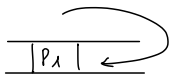
Round robin w/ quantum = 4 ms

↳ Average waiting time ?





(1) P_1 is pop out of stack
(2) Stack only have P_1
(3) P_1 is served again

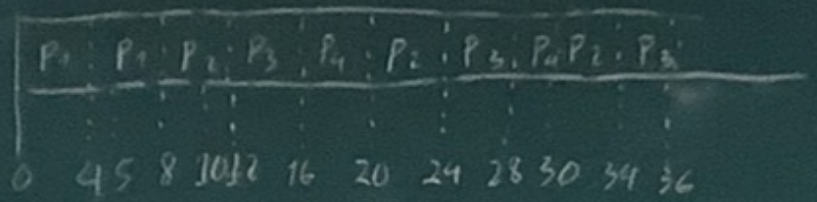


the quantum for each proc must still be $\tau = 4$ ms

cess Management

ng

RR week quantum = 4 ms



t = 34

Ready Queue

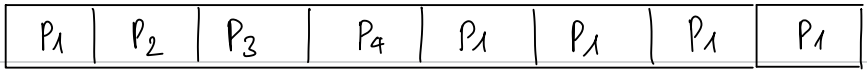
$t_{w_1} = 8 - 0 - 8 = 0$

$t_{w_2} = 34 - 5 - 12 = 17$

$t_{w_3} = 36 - 8 - 10 = 18$

$t_{w_4} = 30 - 10 - 6 = 14$

$\bar{t}_w = \frac{0 + 17 + 18 + 14}{4} = \frac{49}{4}$



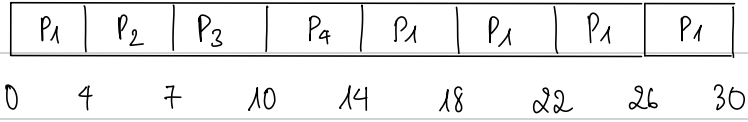
$t_{w_1} = 30 - 0 - 24 = 6$

$t_{w_2} = 4$

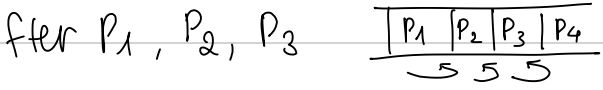
$t_{w_3} = 7$

※ Major mistake:

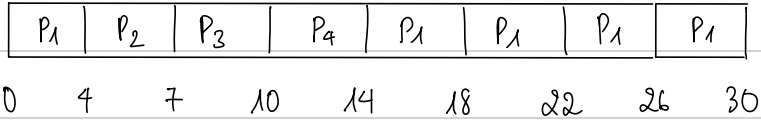
a) Consider this



Where will P_4 be in the queue? After P_1, P_2, P_3



b) Trickier



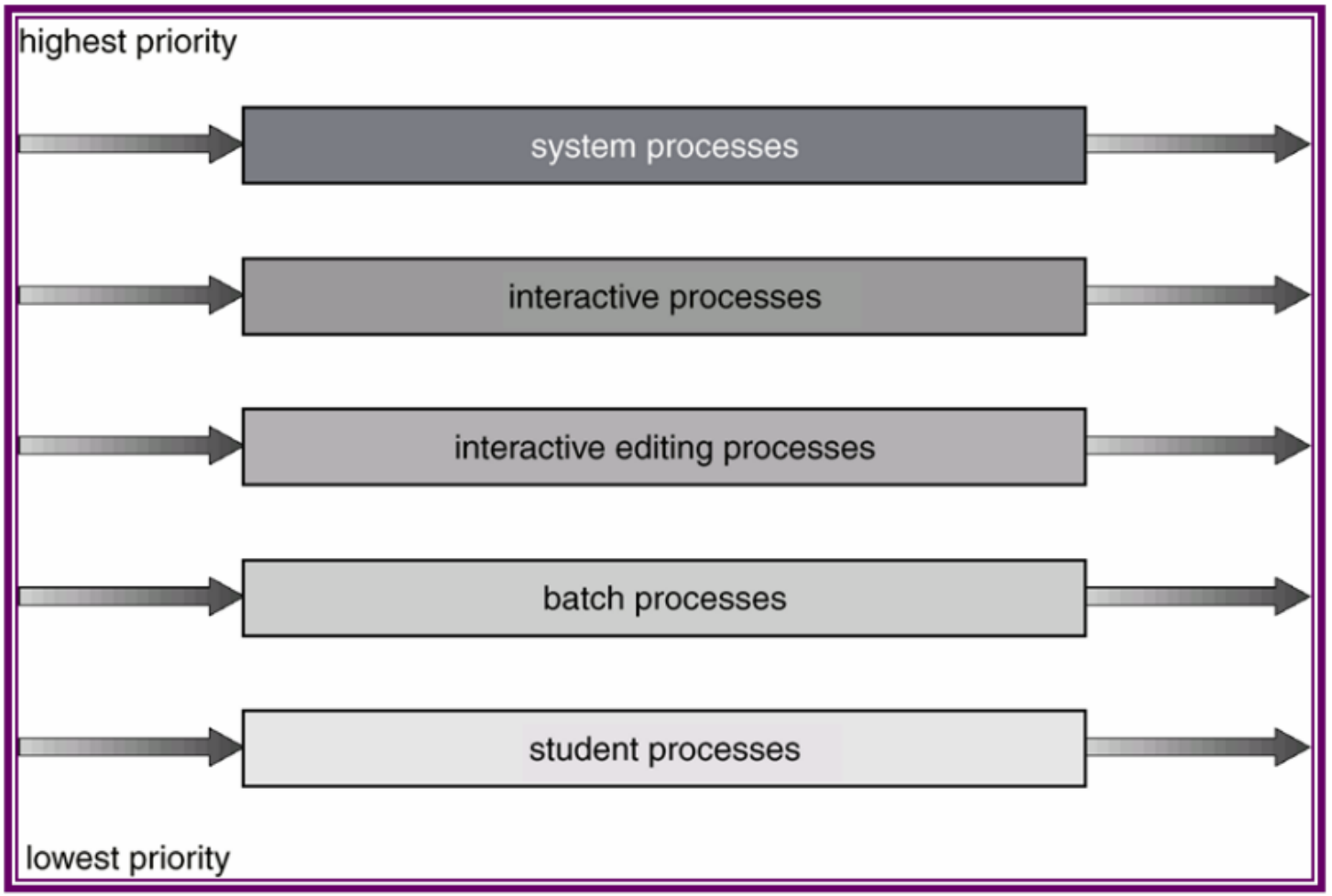
Which will come to queue first? P_1 done / (P_5 new)

※ What does OS do when the process is done before time quantum expired? wait / (run next process immediately)

Multilevel Queue Scheduling

- Partitions the ready queue into several separate queue
- Processes are permanently assigned to one queue
  - Based on some property of the process, such as memory size, process priority, or process type..
- Each queue has its own scheduling algorithm
- There must be scheduling among the queues
  - Commonly implemented as fixed-priority preemptive scheduling
    - Processes in lower priority queue only executed if higher priority queues are empty
    - High priority process preemptive CPU from lower priority process
    - *Starvation* is possible
  - Time slice between the queues
    - foreground process, queue 80% CPU time for RR
    - background process queue, 20% CPU time for FCFS

Multilevel Queue Scheduling (Example)



Multilevel Queue Scheduling (Example)

Ex: Queue 1 utilize RR with quantum =2
Queue 2 utilize FCFS

Process	Queue No	Arrival Time	Burst Time
P1	1	0	4
P2	1	0	3
P3	2	0	8
P4	1	10	5

P3 wait time = ?

Q1 (RR)

P1	P2	P4

Q2 (FCFS)

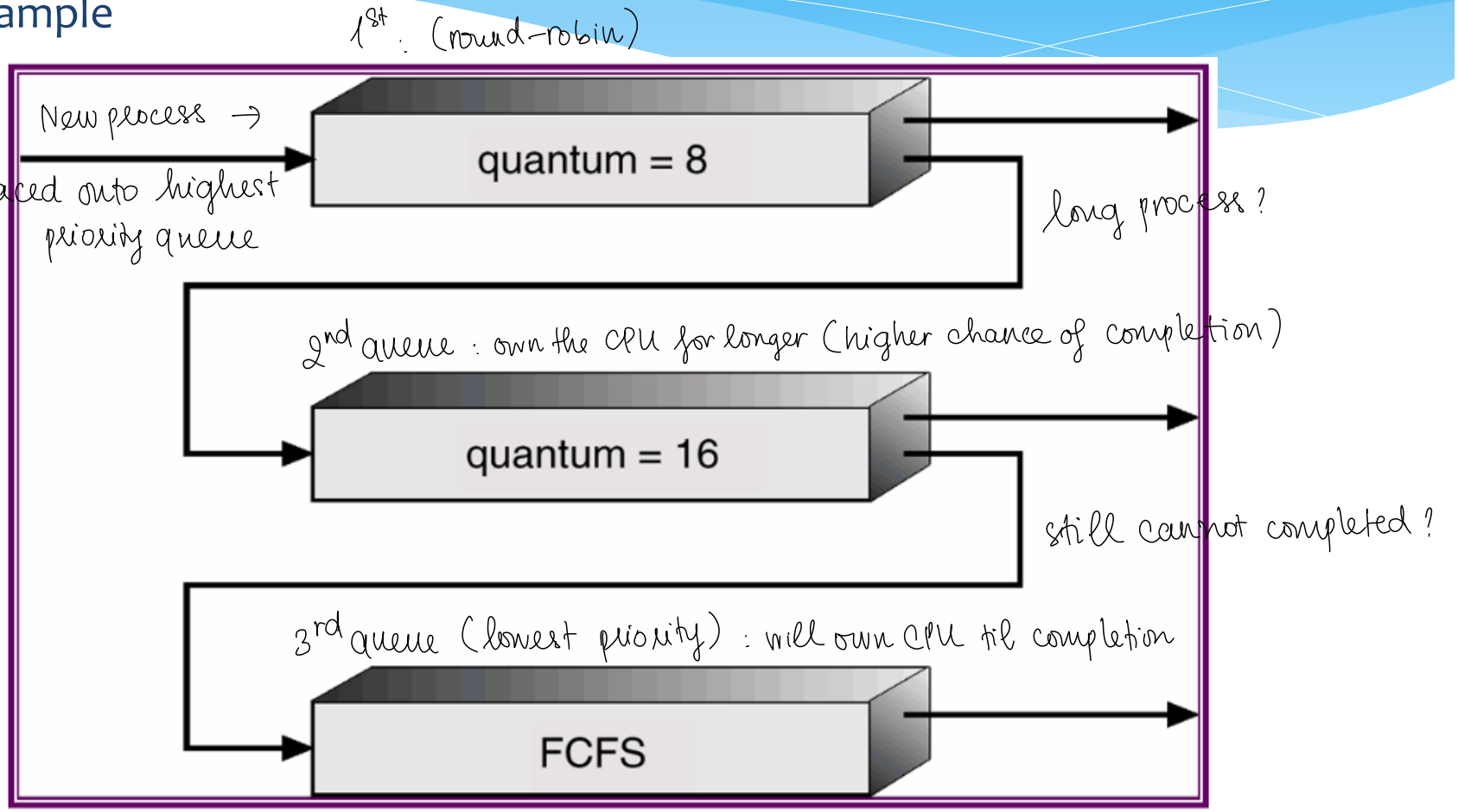
P3

Multilevel Feedback Queue

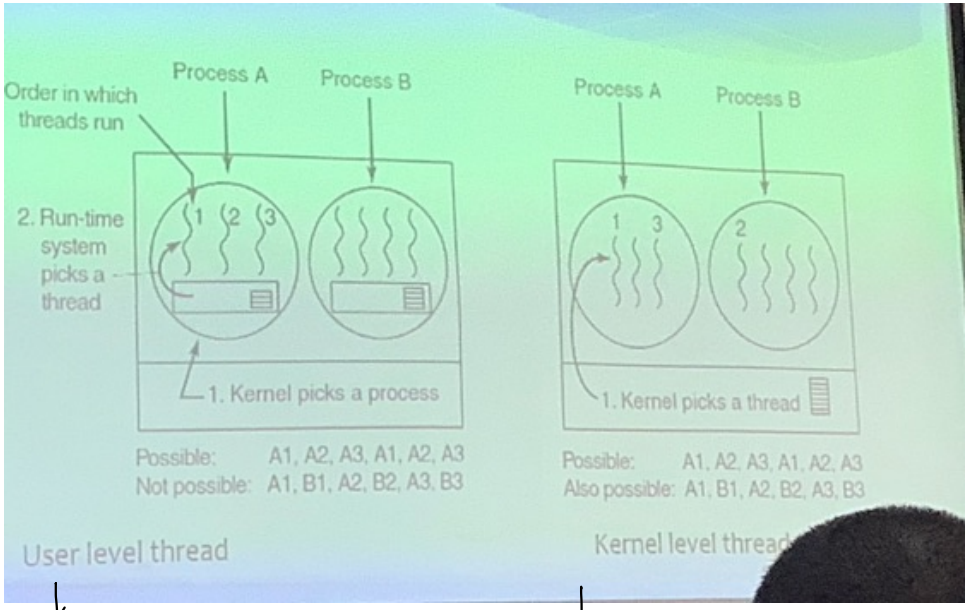
- Allows a process to move between queues
- separate processes with different CPU-burst characteristics
  - If a process uses too much CPU time ->moved to a lower-priority queue
  - I/O-bound and interactive processes in the higher-priority queues
  - process that waits too long in a lower- priority queue may be moved to a higher-priority queue
    - Prevent “starvation”
- multilevel feedback queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithm for each queue
  - method used to determine when to upgrade/demote a process to a higher/lower priority queue
  - method used to determine which queue a process will enter when that process needs service (based on some characteristics: CPU/IO-bound; interactive/non-~; ...)

Multilevel Feedback Queue

- Example



Thread scheduling



↓ not possible to switch between different processes' threads

↓ possible to switch between A & B (system knows about existence of all these threads)

- Windows thread library actually implements both user-level & kernel-level threads

2 parallel levels: Process & Thread

↳ Round Robin is commonly used

Suppose process A has been done, before the time & resources allocated for A expired (there is still much of it remains). However, instead of making the switch $A_1 \rightarrow B_1$; the scheduler choose $A_1 \rightarrow A_2$. Why?

↳ Since resources is allocated process-wise; it is actually more context-switching (thus, more work) to do $A_1 \rightarrow B_1$. Thus the scheduler would prefer $A_1 \rightarrow A_2$.

- Basic Concepts
- Scheduling Criteria
- Scheduling algorithms
- Multi-processor scheduling

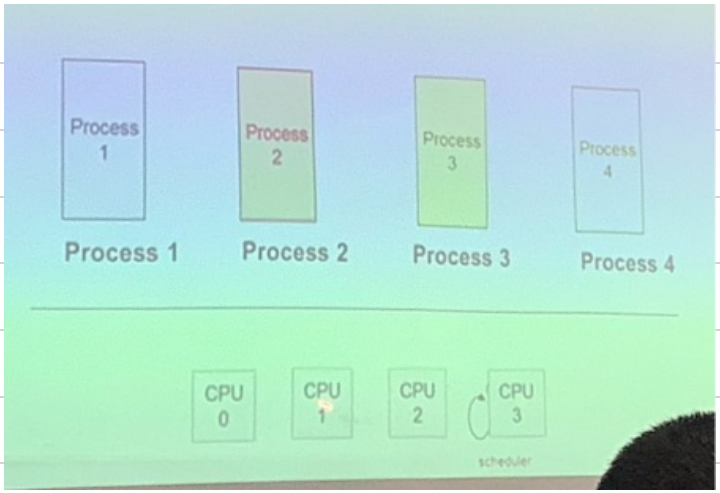
Problem

- the scheduling problem is correspondingly more complex
- Load sharing
  - separate queue for each processor
    - one processor could be idle, with an empty queue, while another processor was very busy
    - use a common ready queue
  - common ready queue's problems :

    →two processors do not choose the same process or

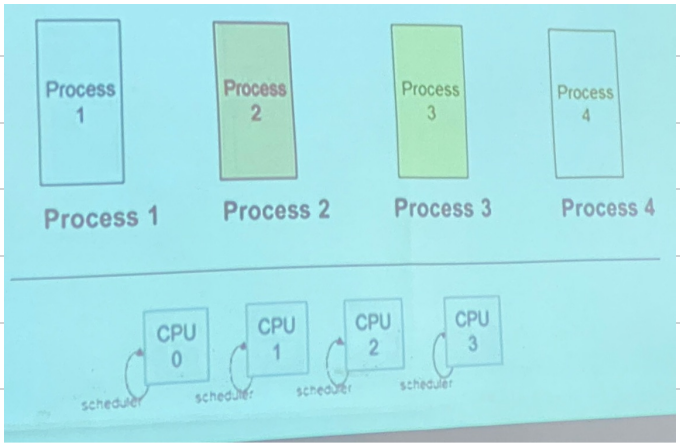
    →processes are lost from the queue (1 process may be execute by processor A, then served by processor B → data may be lost during the processor switch)
- asymmetric multiprocessing
- only one processor accesses the system's queue -> no sharing problem
  - I/O-bound processes may bottleneck on the one CPU that is performing all of the operations

Asymmetric scheduling



- Only 1 processor can access the queue
- May have bottleneck at 1 processor
- Only 1 processor schedules the processes

Symmetric scheduling



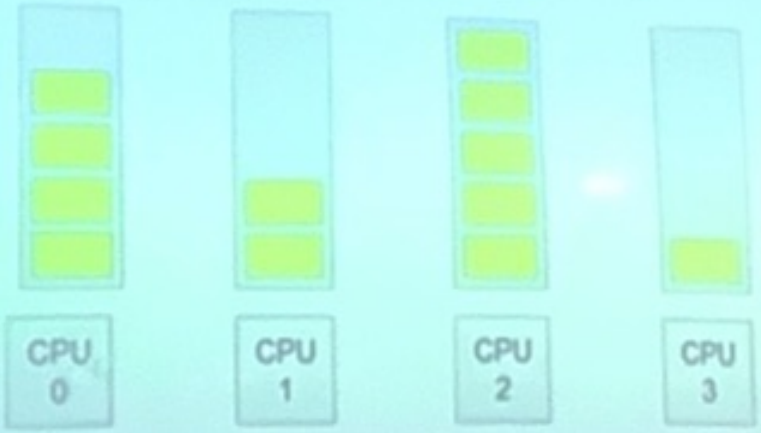
- Each processor runs it own Scheduler
- Independently selects available processes in the queue

⊖ Unbalanced load between processors

3-4. Multi-processor scheduling

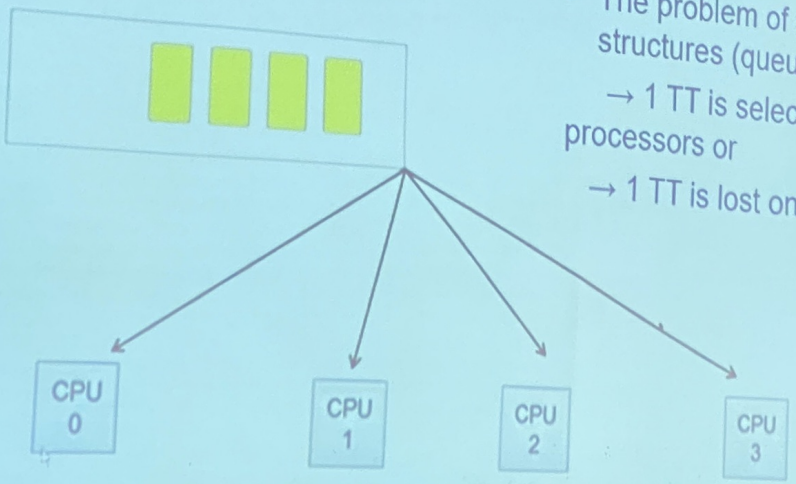
Symmetric multiprocessor

- Each processor has its own ready queue
- divide proceses into fixed queues
- Pros: Easy to organize
- Cons: Exist idle processor with empty queue while other processor has to do a lot of computation



Symmetric multiprocessor

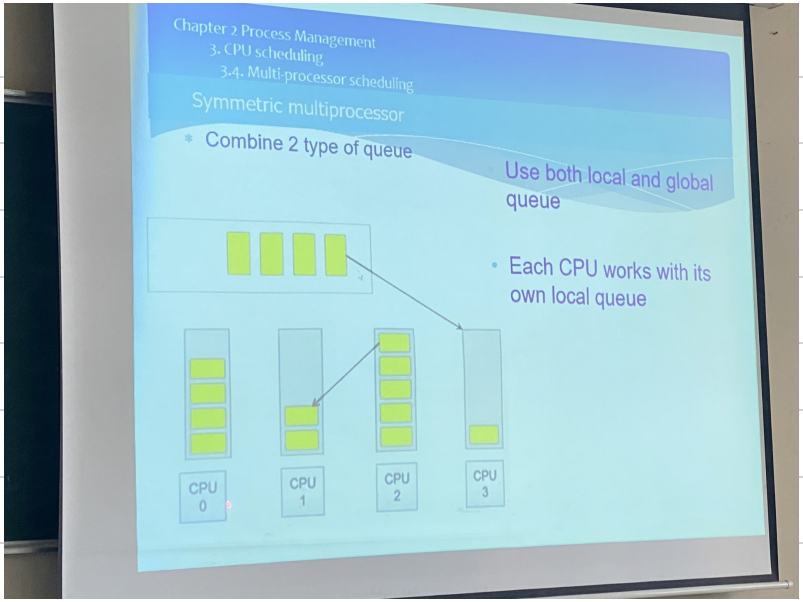
* Share ready queue (Global queue)
  * Pros:
    * Efficient use of CPU
    * Fair to the Process
  * Cons:
  * The problem of sharing data structures (queues):
    → 1 TT is selected by 2 processors or
    → 1 TT is lost on the queue



Symmetric multiprocessor

① 1st have a global queue

② Processes migrate to local queues 1 by 1
(if local queue ! full i.e. busy CPU)



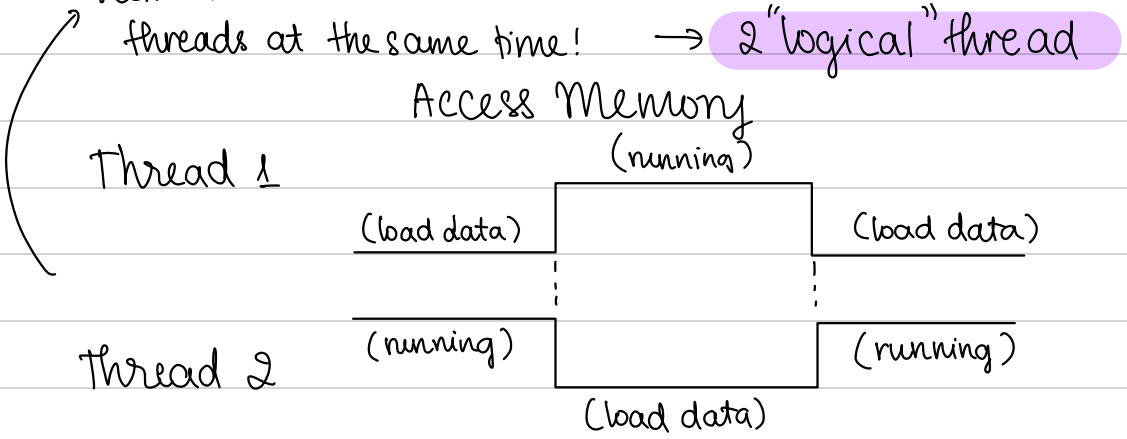
Transfer speed can not keep up w/ computation speed
↳ Processor can run another thread while waiting for data transfer of current thread

Hyper thread

∩

Processor listing
i.e. 2 physical
4 logical

seems to be able to run 2 threads at the same time! → 2 "logical" thread



CPU scheduling in Windows

Preemptive priority-based scheduling

32 priority values

- (0 memory management thread)
- 1-31 divided into 6 class
- Each value corresponding to 1 queue -> search for ready thread from higher priority to lower priority queue
- Find no ready thread -> run idle thread

- New thread → normal priority queue
- Preemptive

Homework

- Write program to simulate multilevel feedback queue

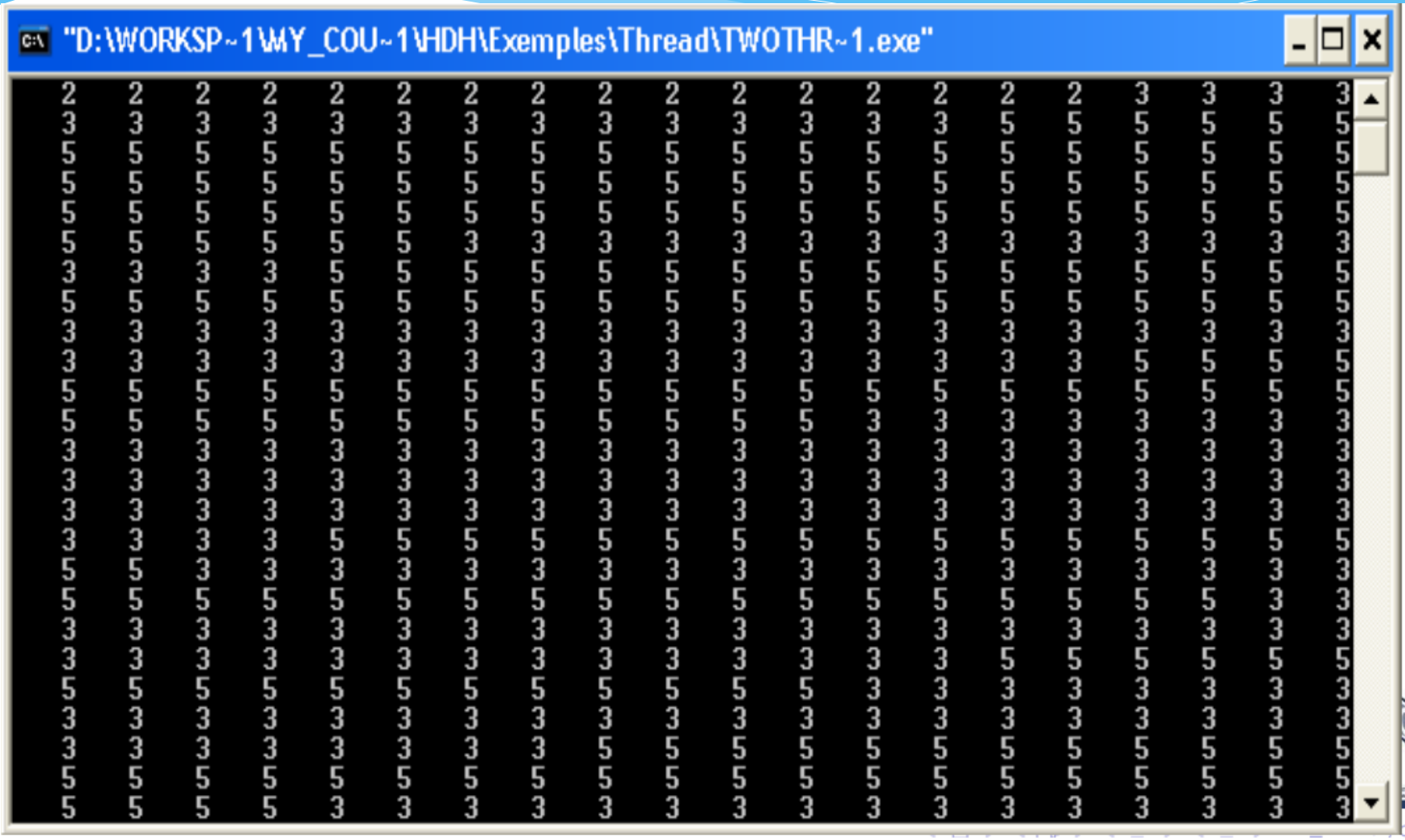
① Process
② Thread
③ CPU scheduling
④ Critical resource and process synchronization
⑤ Deadlock and solutions

- Critical resource
- Internal lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization example
- Monitor

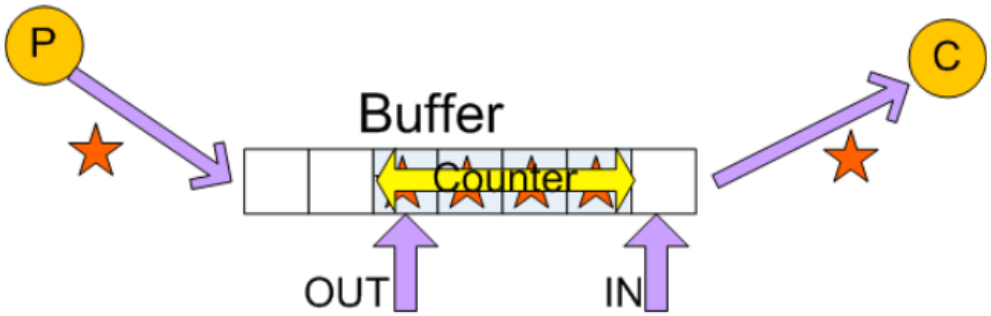
Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
void T1(){
	while(1){ x = y + 1; printf("%4d", x); }
}
void T2(){
	while(1){ y = 2; y = y * 2; }
}
int main(){
	HANDLE h1, h2; DWORD Id;
	h1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T1,NULL,0,&Id);
	h2=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)T2,NULL,0,&Id);
	WaitForSingleObject(h1,INFINITE);
	WaitForSingleObject(h2,INFINITE);
	return 0;
}
```

Results



producer-consumer I



- Two processes in the system
  - Producer creates products
  - Consumer consumes created product
- Producer and Consumer must be synchronized.
  - Do not create product when there is no space left
  - Do not consume product when there is none

Producer-Consumer Problem II

Producer

```
while(1) {
    /*produce an item in
nextProduced*/
    while (Counter == BUFFER_SIZE) ;
                /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}
```

Consumer

```
while(1){
    while(Counter == 0) ;
            /*do nothing*/
     nextConsumed = Buffer[OUT];
     OUT =(OUT + 1) % BUFFER_SIZE;
    Counter--; /*consume the item in
                nextConsumed*/
}
```

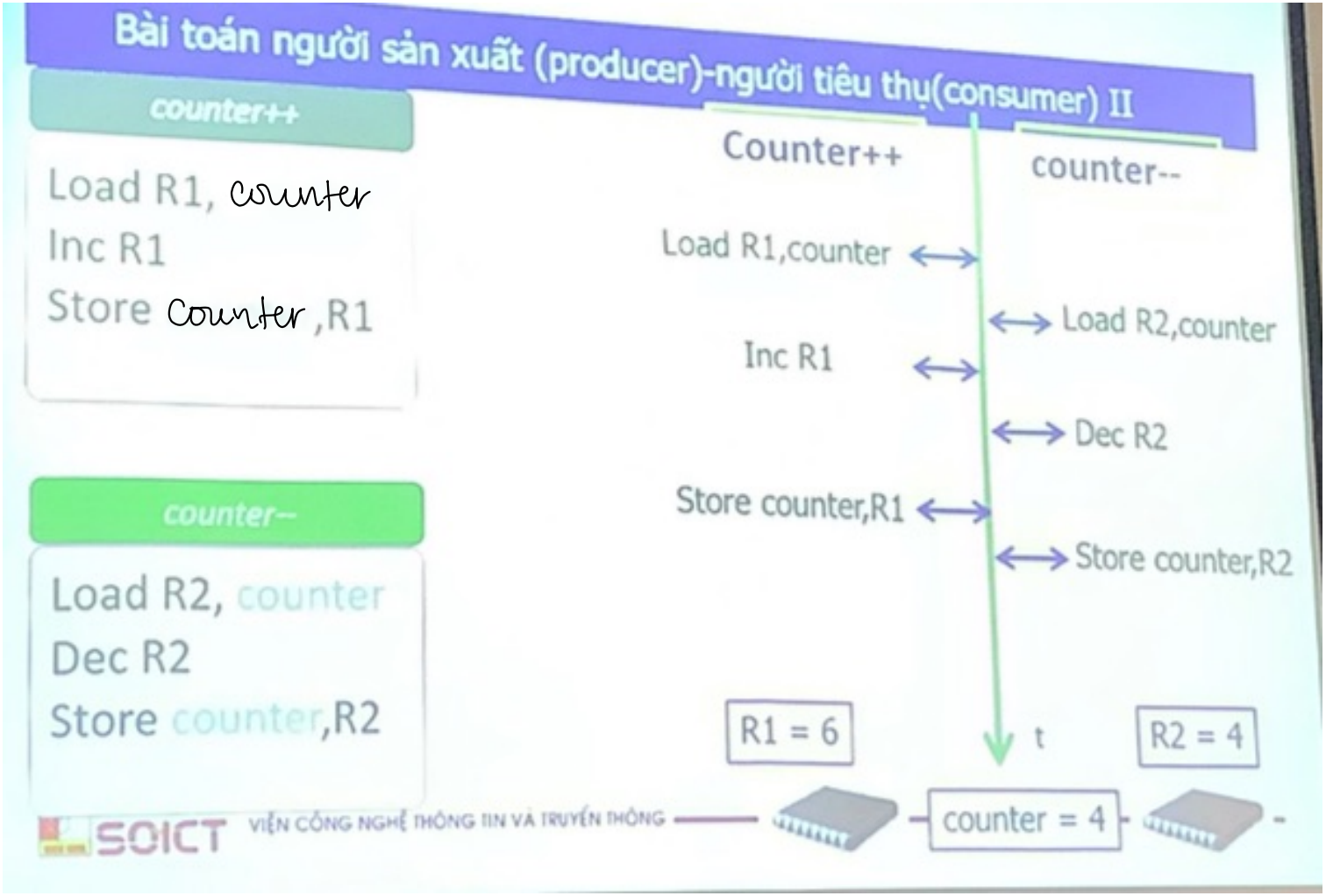
Nhận xét

- Producer creates one product
- Consumer consumes one product

⇒Number of product inside Buffer does not change

Is there any chance that the counter is updated wrongly?

↓



the counter here is the critical resources.

Definition

Resource

Everything that is required for process's execution

Critical resource

(nb of process can access is limited)

- Resource that is limited of sharing capability
- Required concurrently by processes
- Can be either physical devices or sharing data

Problem

Sharing critical resource may not guarantee data completeness
⇒ Require process synchronization mechanism

Monitor / printer / keyboard / mouse

Which one is critical resource ?

→ None of them

A resource will only become critical when the nb of requesting processes EXCEEDED the nb of processes that the resources can serve at 1 time

Race condition

- Situation where the results of many processes access the sharing data depend of the order of these actions
  - Make program's result undefinable
- Prevent race condition by synchronize concurrent running processes
  - Only one process can access sharing data at a time
    - Variable counter in Producer-Consumer problem
  - The code segment that access sharing data in the process have to be executed in a defined order
    - E.g.: $x \leftarrow y+1$ instruction in Thread T_1 only both two instruction of Thread T_2 are done

Critical section

- The part of the program where the shared memory is accessed is called the critical region or critical section
- When there are more than one process use critical resource then we have to synchronize them
  - Object: guarantee that no more than one process can stay inside critical section

Conditions to have a good solution

- Mutual Exclusion: Critical resource does not have to serve the number of process more than its capability at any time
  - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
- Progress: If critical resource still able to serve and there are process want to be executed in critical section then this process can use critical resource
- Bounded Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

↳ (i.e. somebody really want to use the restroom while you're inside - you'd better go faster!)

Rule

- There are two processes *P1&P2* concurrently running
- Sharing the same critical resource
- Each process put the critical section at begin and the remainder section is next
  - Process must ask before enter the critical section *{entry section}*
  - Process perform *{exit section}* after exiting from critical section
- Program structure

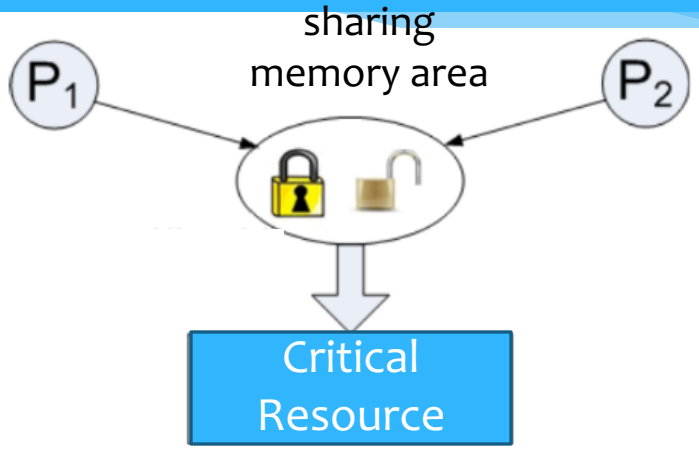
```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

Methods' classification

- Low level method
  - Variable lock
  - Test and set
  - Semaphore : all sync instructions are written by dev
- High level method
  - Monitor

- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization example
- Monitor

Principle



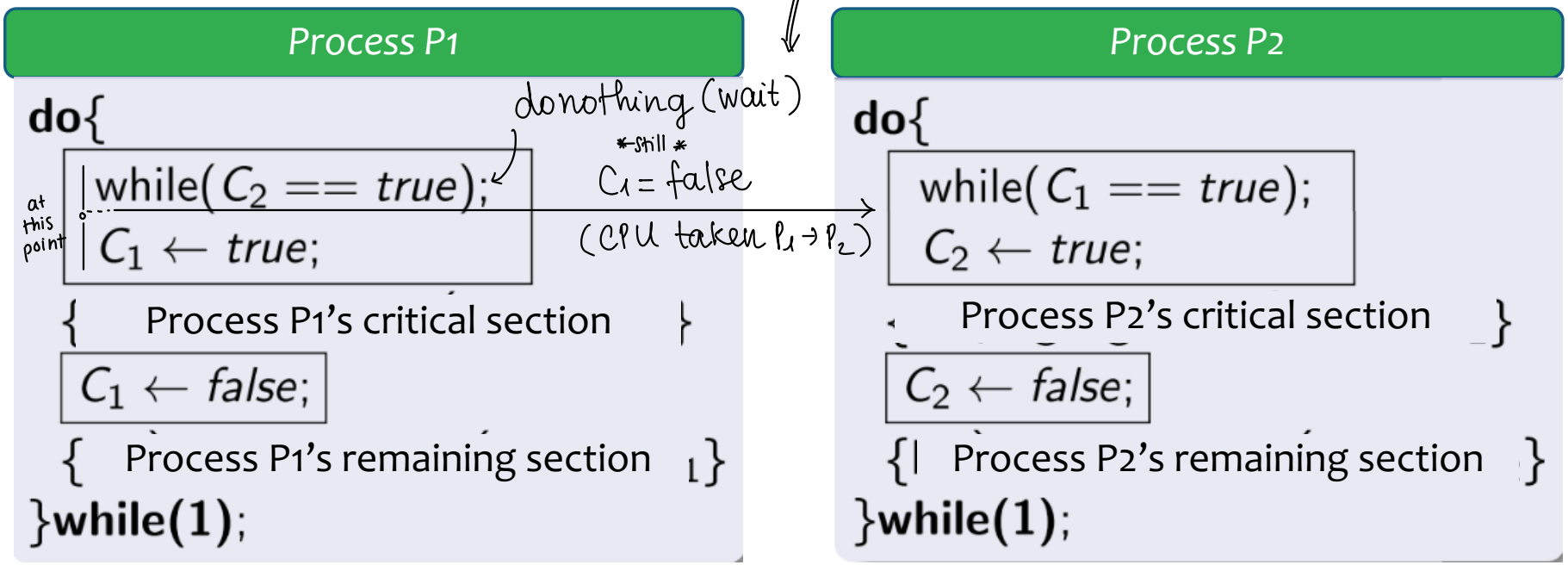
- Each process use one byte in the sharing memory area as a lock
  - Process enter critical section, lock (byte lock = true)
  - Process exit from critical section, unlock (byte lock= false)
- Process want to enter critical section: check other process's lock byte 's status
  - Locking ⇒ Wait
  - Not lock ⇒ Have the right to enter critical section

Algorithm

* At a single moment, only 1 process is in the critical section

- Share var C1,C2 Boolean // sharing variable used as lock
- Initialization C1 = C2 = false // Critical resource is free

Is there any chance that both processes acknowledge that they have the right to run?



Algorithm

⇒ No more violation to MUTUAL EXCLUSION rule

- Share var C1,C2 Boolean // sharing variable used as lock
- Initialization C1 = C2 = false // Critical resource is free

But can not fulfill PROGRESSIVE rule

Process P1

```
do{
    C1 ← true;
    while(C2 == true);
    { Process P1's critical section }
    C1 ← false;
    { Process P1's remaining section }
}while(1);
```

choose lock from the beginning

Process P2

```
do{
    C2 ← true;
    while(C1 == true);
    { Process P2's critical section }
    C2 ← false;
    { Process P2's remaining section }
}while(1);
```

Remark

- Not properly synchronize
  - Two processes request resource at the same time
    - Mutual exclusion problem (Case 1)
    - Progressive problem (Case 2)
- Reason: The following actions are done separately
  - Check the right to enter critical section
  - Set the right to enter critical section

MUTUAL EXCLUSIVE + PROGRESSIVE ⇒ problem resolved!

Dekker's algorithm

- Utilize ***turn*** variable to show process with priority

Process P1

```
do{
  C1 ← true;
  while(C2==true){
    if(turn == 2){
      C1 ← false;
      while(turn ==2);
      C1 ← true;
    }
  }
    { Process P1's critical section }
  turn = 2;
  C1 ← false;
    { P1's remaining section }
}while(1);
```

Process P2

```
do{
  C2 ← true;
  while(C1==true){
    if(turn == 1){
      C2 ← false;
      while(turn ==1);
      C2 ← true;
    }
  }
    { P2's critical section }
  turn = 1;
  C2 ← false;
    { P2s remaining section }
}while(1);
```

Remark

- Synchronize properly for all cases
- No hardware support requirement -> implement in any languages
- Complex when the number of processes and resources increase
- "busy waiting" before enter critical section
  - When waiting, process has to check the right to enter the critical section => Waste processor's time

- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization example
- Monitor

Principle

- Utilize hardware support
- Hardware provides uninterruptible instructions
- Test and change the content of aword

```
boolean TestAndSet(VAR boolean target) {
    boolean rv = target;
    target = true;
    return rv;
}
```

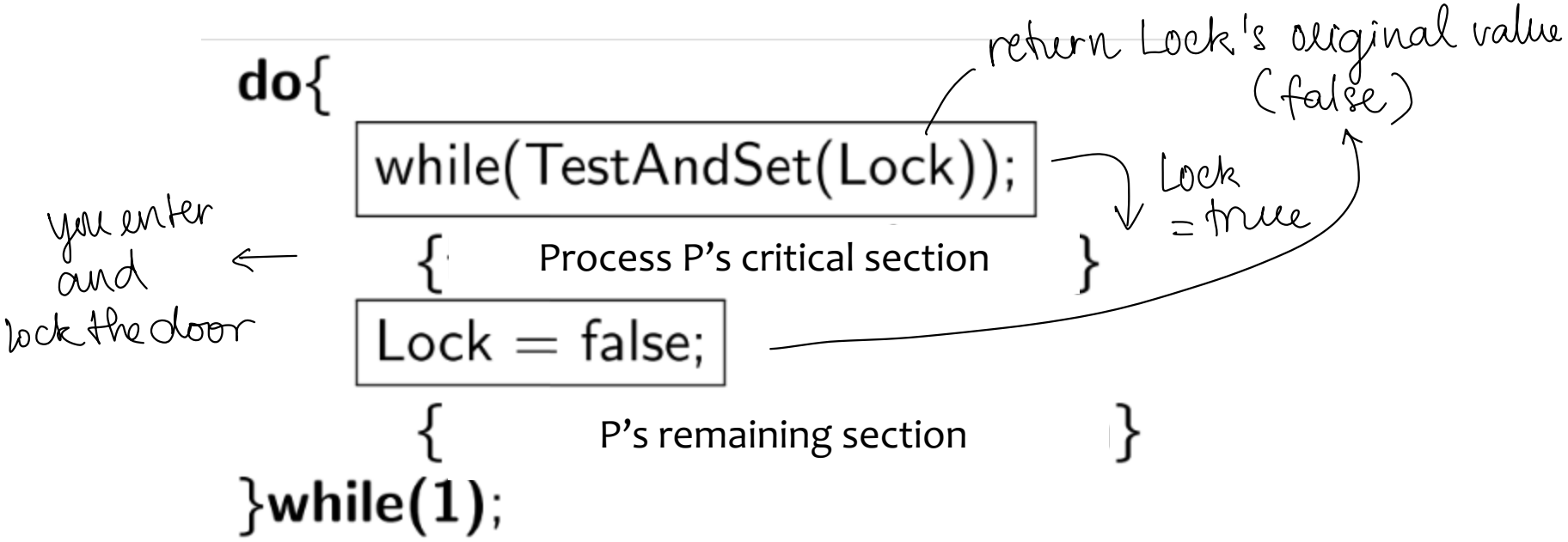
  - Swap the content of two different words

```
void Swap(VAR boolean , VAR boolean b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

- instruction is executed atomically
- Code block is uninterruptible when executing
  - When called at the same time, done in any order

Algorithm with TestAndSet instruction

- Sharing variable **Boolean: Lock**: resource's status:
  - Locked (*Lock=true*)
  - Free (*Lock=false*)
- Initialization: *Lock = false* ⇒ Resource is free
- Algorithm for process *Pi*



Algorithm with Swap instruction

- Sharing variable **Lock** shows the resource's status
- Local variable for each process: **Key**: Boolean
- Initialization: *Lock = false* ⇒ Resource is free
- Algorithm for process *Pi*

```
do{
    key = true;
    while(key == true)
        swap(Lock, Key);
    { Process P's critical section }
    Lock = false;
    { P's remaining section }
}while(1);
```

Remark

- Simple, complexity is not increase when number of processes and critical resource increase
- "busy waiting" before enter critical section
- When waiting, process has to check if sharing resource is free or not => Waste processor's time
- No bounded waiting guarantee
  - The next process entering critical section is depend on the resource release time of currently resource-using process

    ⇒ Need to be solved

P1 and P2 call at the same time ⇒ Order is abitrary
(1 random chosen can run, others need to wait)

- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization example
- Monitor

Semaphore

- An integer variable, initialize by resource sharing capability
  - Number of available resources (e.g. 3 printers)
  - Number of resource's unit (10 empty slots in buffer)
- Can only be changed by 2 operation P and V
- Operation P(S) (wait(S))

```
wait(S) {
    while(S ≤ 0) no-op;
    S − −;
}
```

- Operation V(S) (signal(S))

```
signal(S) {
    S + +;
}
```

- P and V are uninterruptible instructions

Semaphore usage I

- n-process critical-section problem
  - processes share a semaphore, mutex
  - initialized to 1.
  - Each process Pi is organized as

Sẽ thi tự luận

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

for 1st process, mutex will be set 1 → 0 ⇒ next process have to wait when it also do wait(mutex)

Semaphore usage II

- The order of execution inside processes:
  - PI with a statement S1 , P2 with a statement S2.
  - require that S2 be executed only after S1 has completed

P1 and P2 share a common semaphore synch, initialized to 0,
Code for each process

Process 1
S1; signal (synch) ; { Remainder code}

Process 2
wait (synch) ; S2; { Remainder code}

To overcome the need for busy waiting

Use 2 operations

- Block() Temporarily suspend running process
- Wakeup(P) Resume process P suspended by block() operation

- When a process executes the wait operation and semaphore value is not positive
  - it must wait. (block itself -> not busy waiting)
  - block operation places a process into a waiting queue associated with the semaphore,
  - Process's state is switched to the waiting
  - Control is transferred to the CPU scheduler,
  - process that is blocked, waiting on a semaphore S, restarted when some other process executes a signal operation.
- The process is restarted by a wakeup operation
  - changes the process from the waiting state to the ready state.
  - process is then placed in the ready queue.

Semaphore implementation

Semaphore S

```
typedef struct{
   int value;
   struct process * Ptr;
}Semaphore;
```

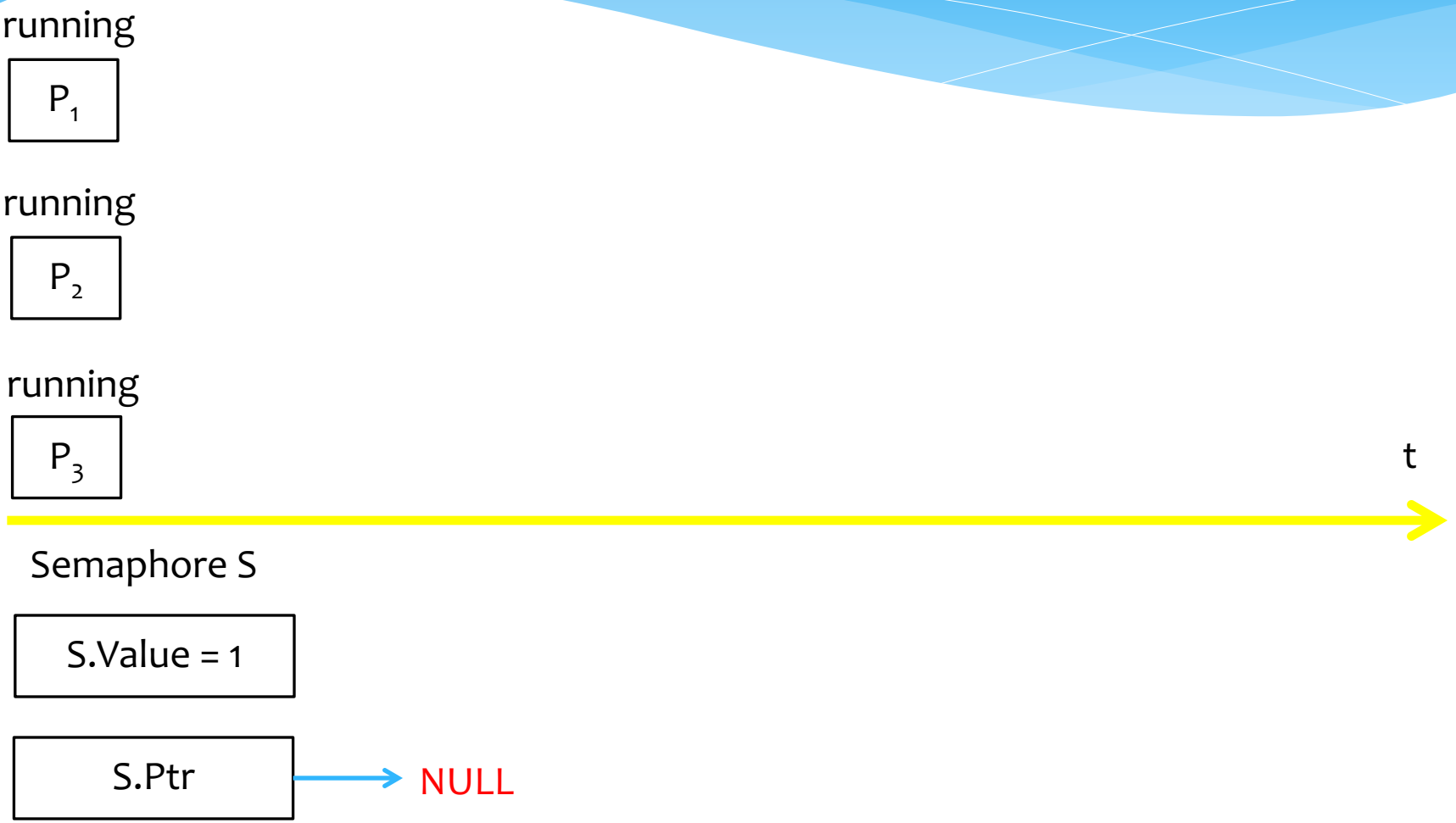
wait(S)/P(S)

```
void wait(Semaphore S) {
   S.value−−;
   if(S.value < 0) {
        Insert process to    S.Ptr
      block();
    }
}
```

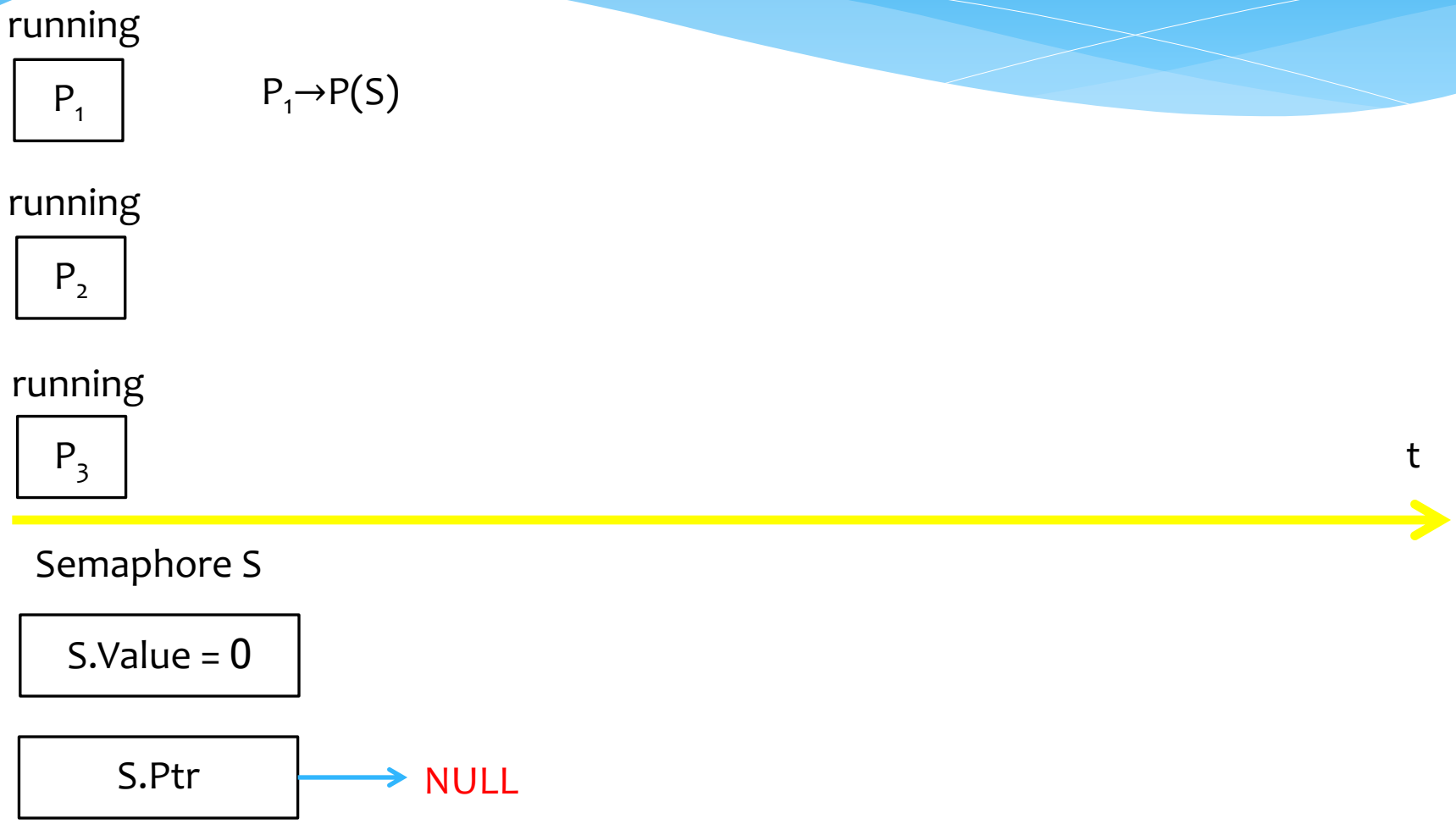
signal(S)/V(S)

```
void signal(Semaphore S) {
 S.value++;
 if(S.value ≤ 0) {
      Get process from    S.Ptr
   wakeup(P);
 }
}
```

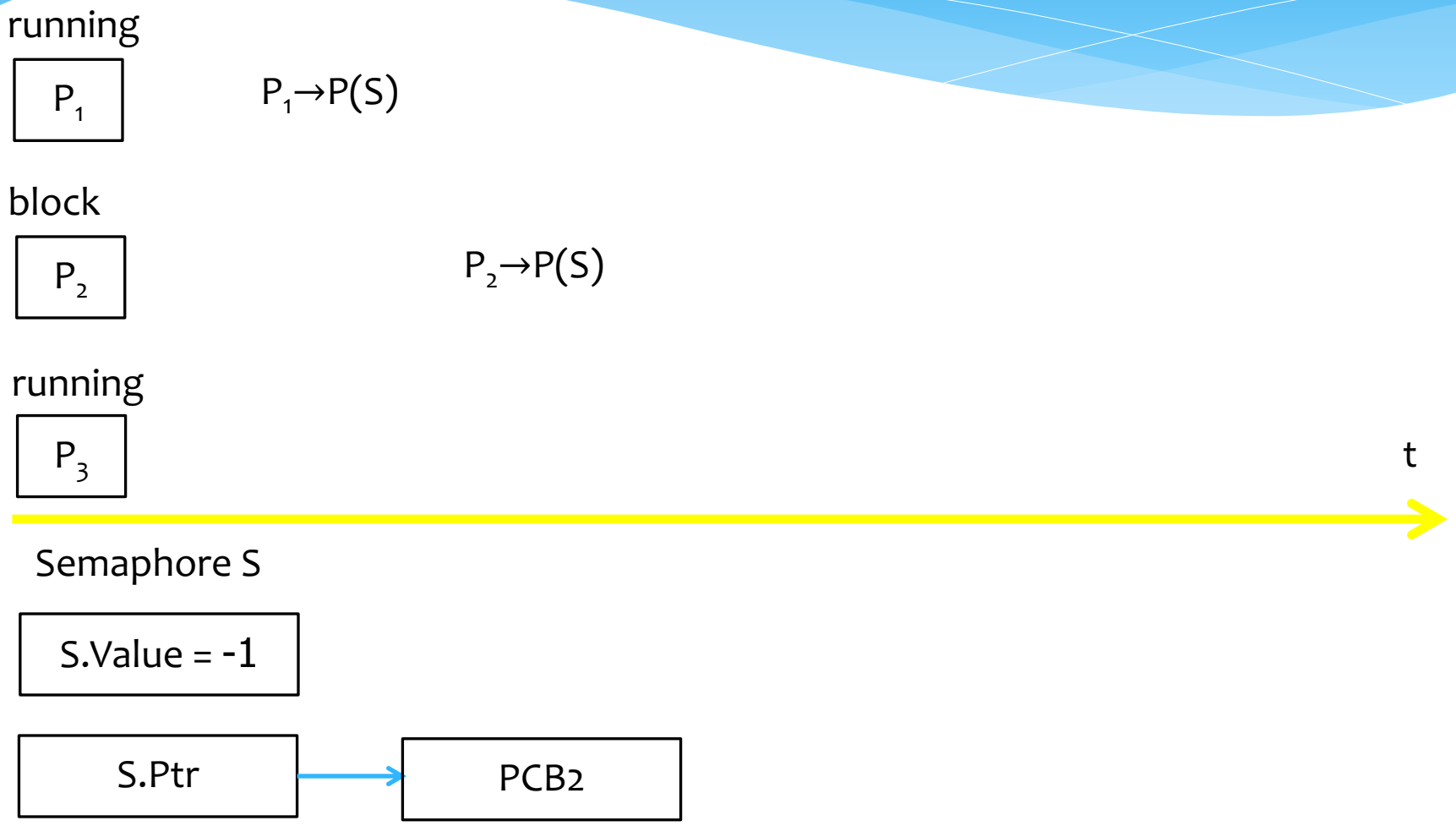
Synchronization example



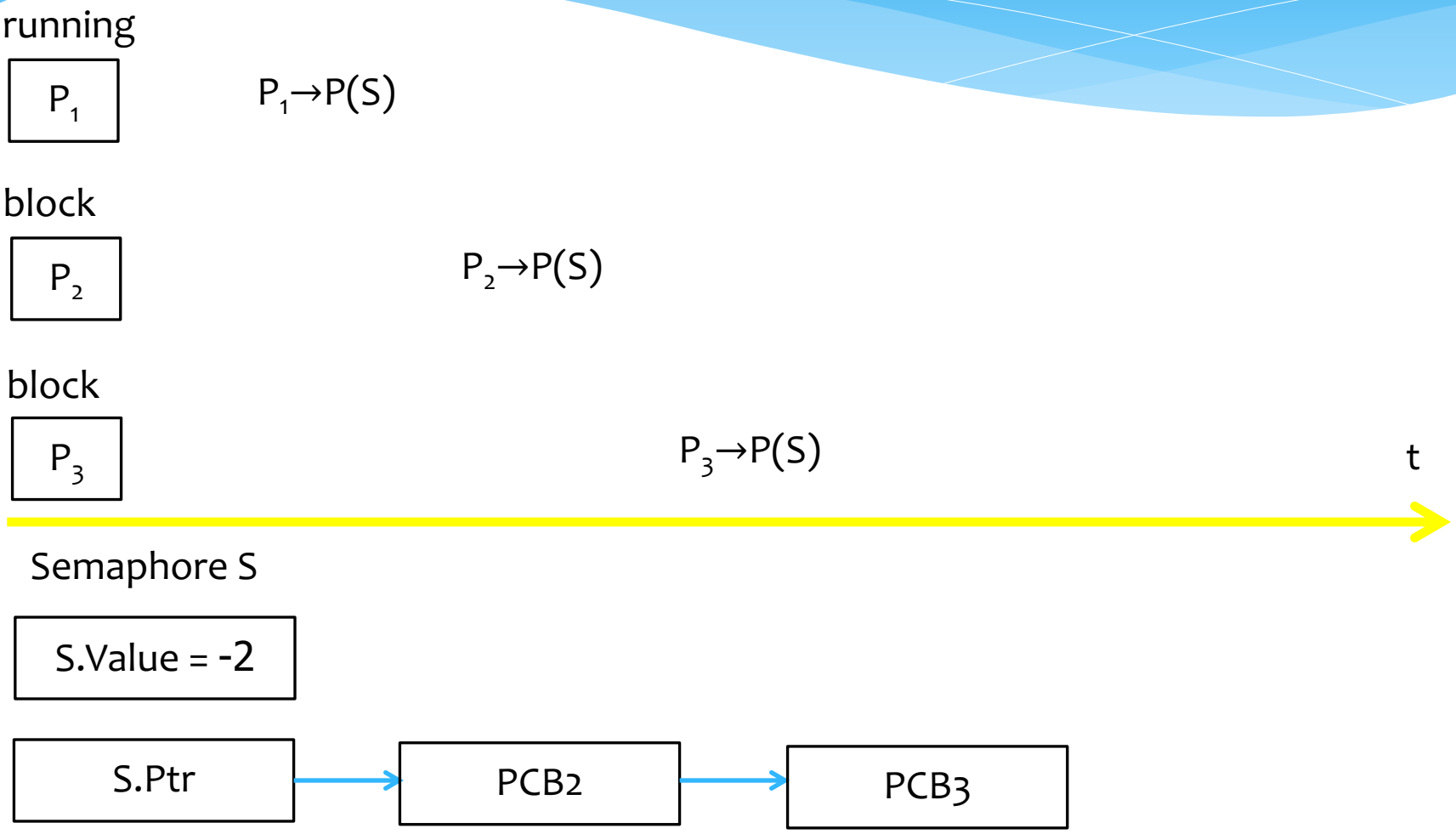
Synchronization example



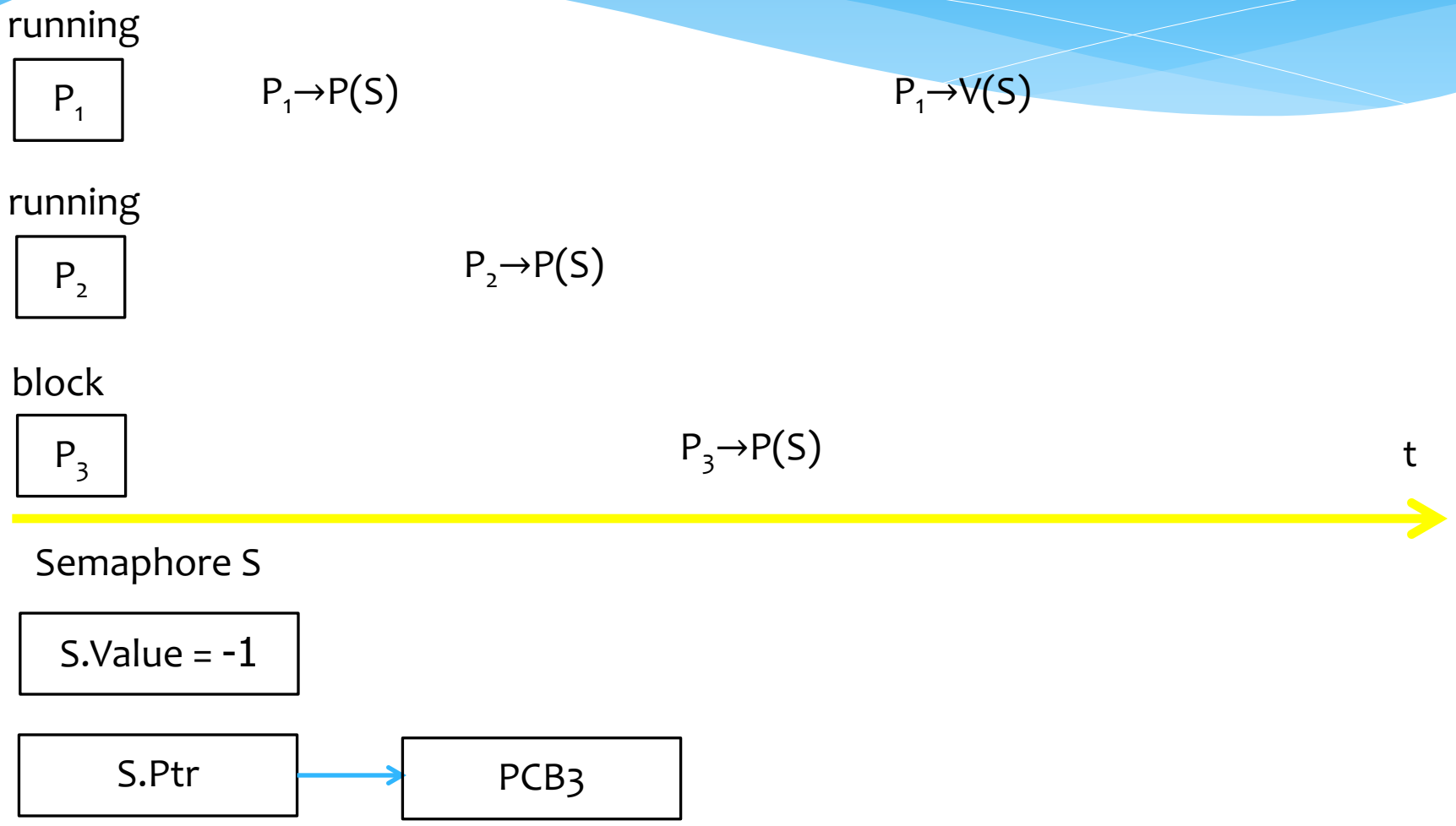
Synchronization example



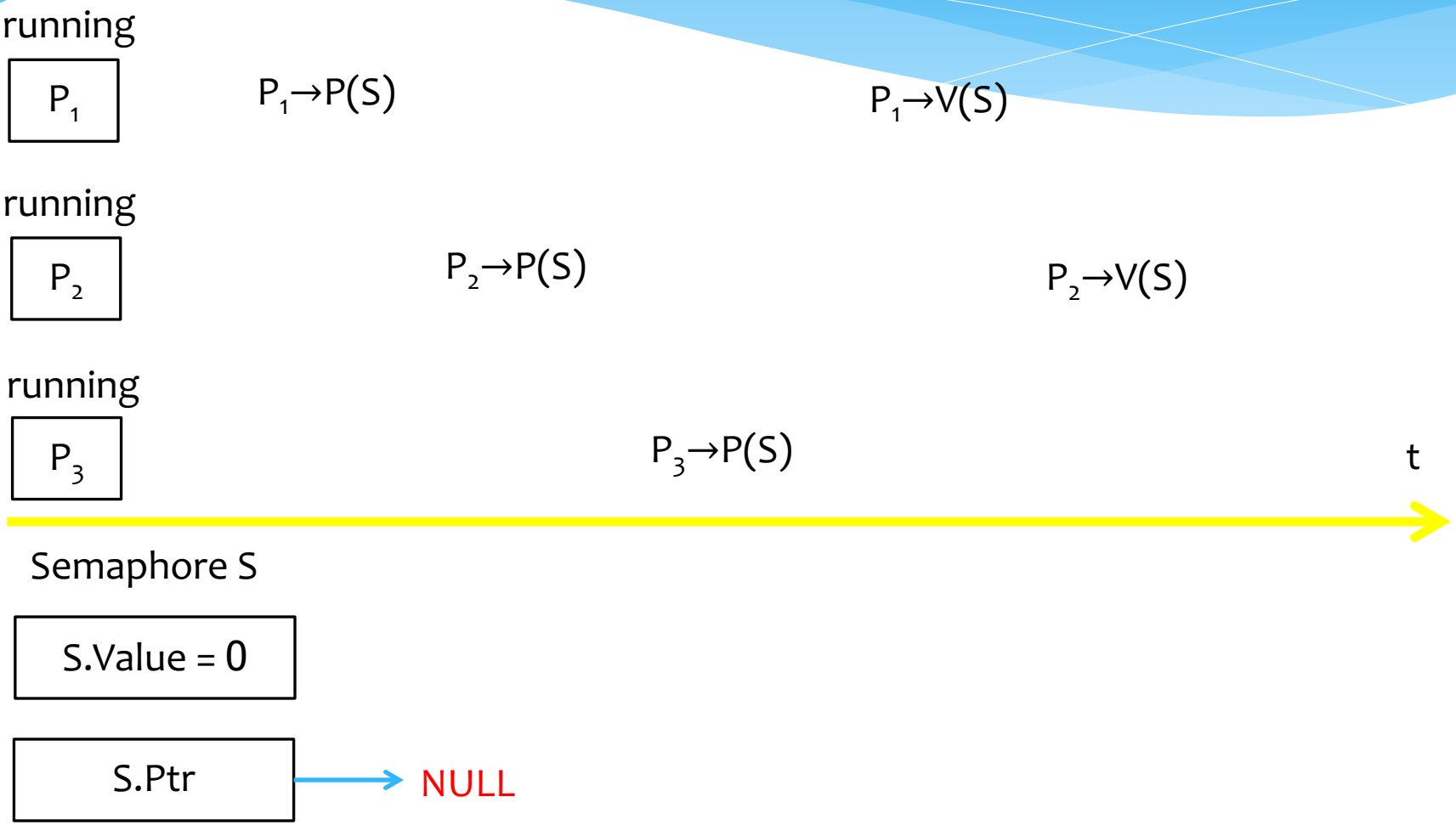
Synchronization example



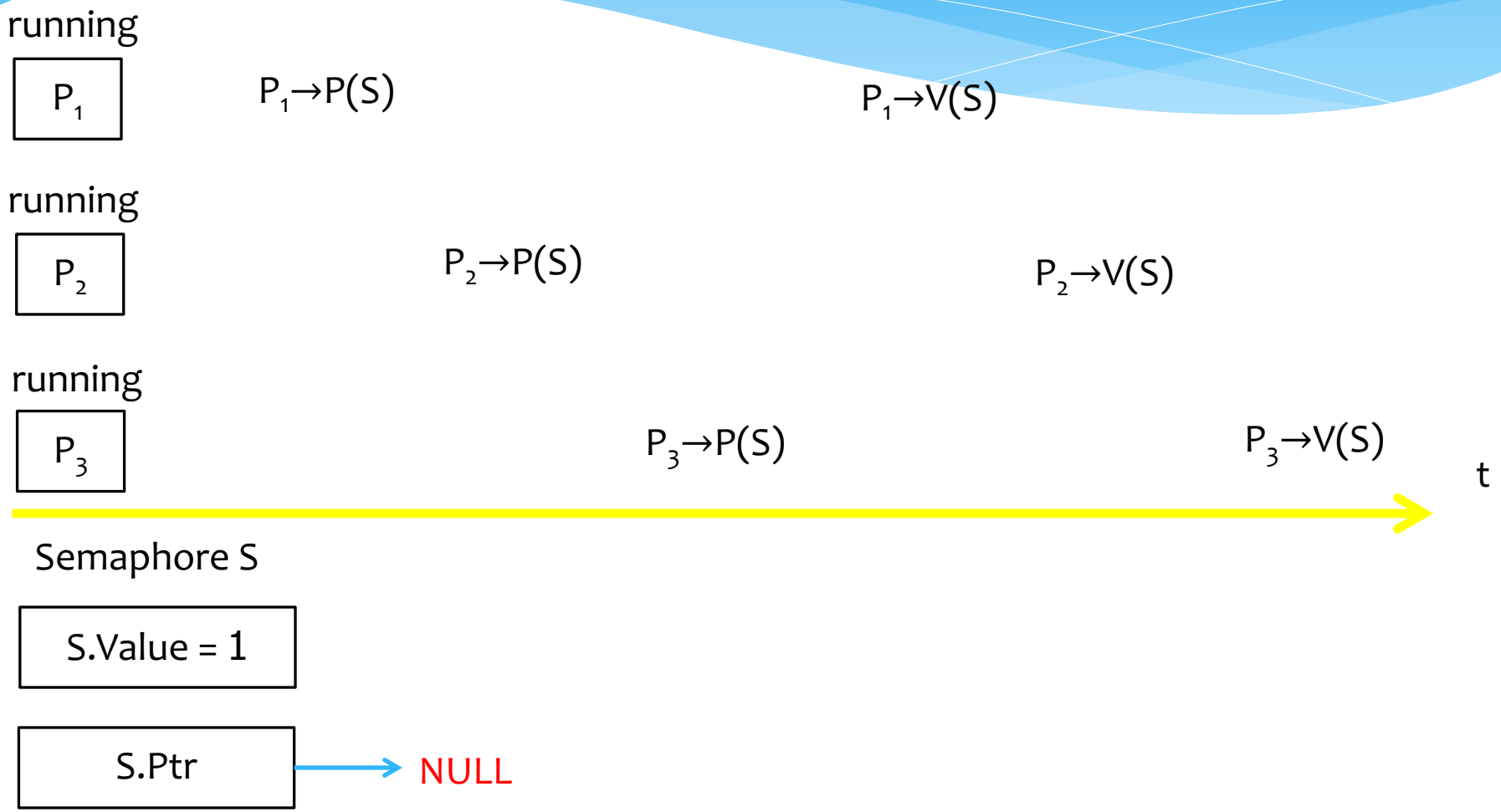
Synchronization example



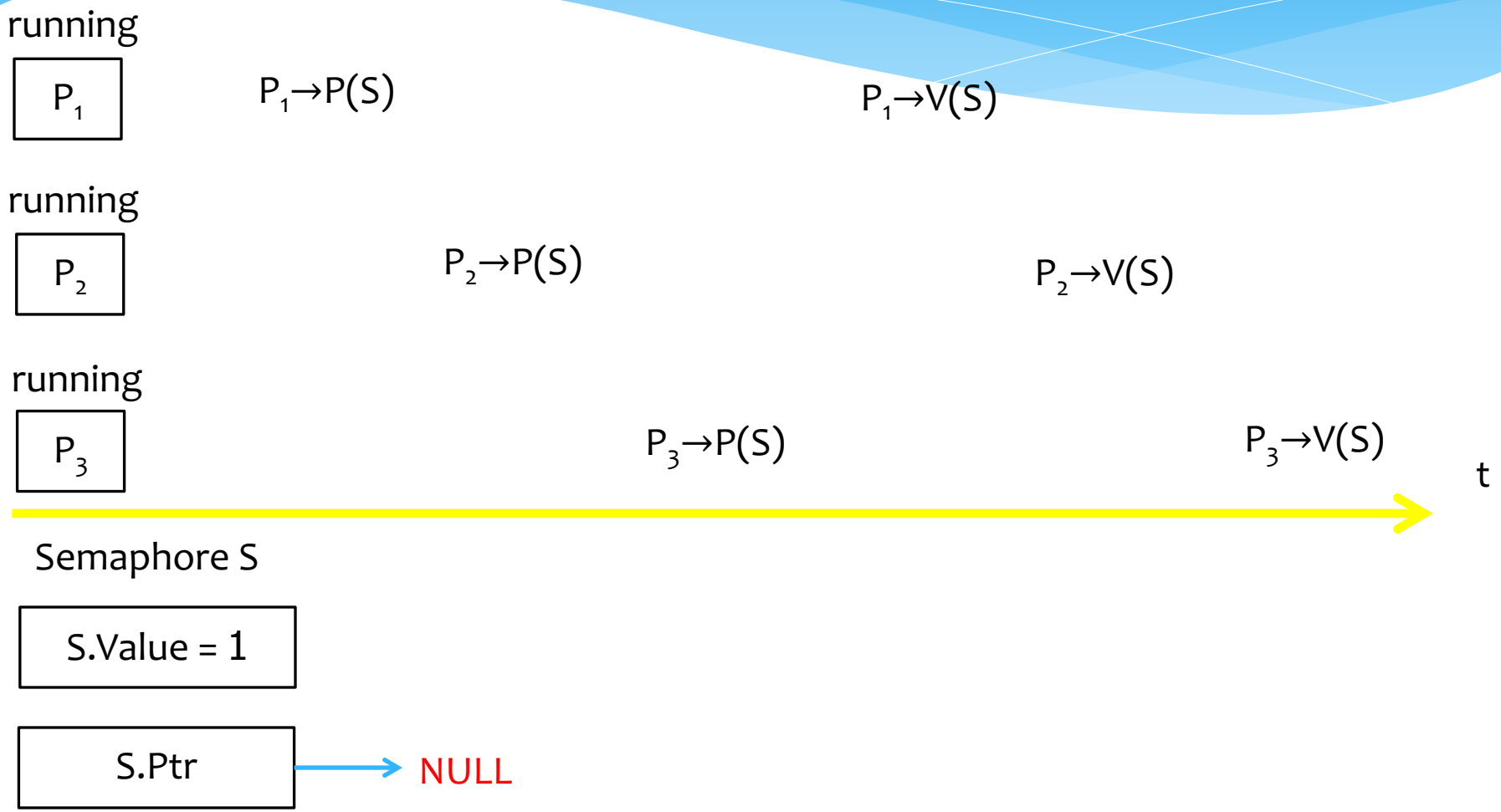
Synchronization example



Synchronization example

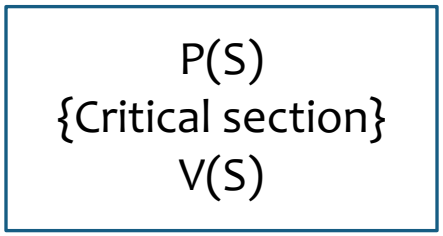


Synchronization example

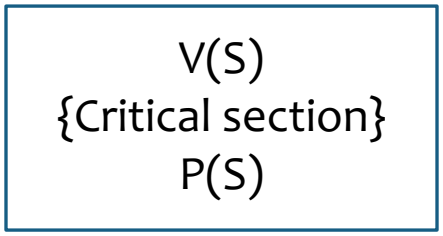


Remark

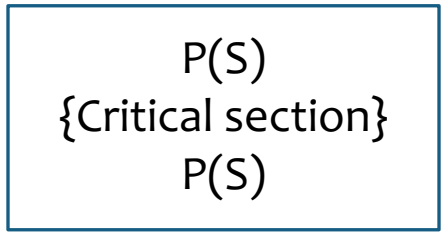
- Easy to apply for complex system
- No busy waiting
- The effectiveness is depend on user



Correct synchronize



Wrong order



Wrong command

Remark

- P(S) and V(S) is nonshareable

⇒P(S) and V(S) are 2 critical resource

⇒Need synchronization

- Uniprocessor system: Forbid interrupt when perform wait(), signal()
- Multiprocessor system
  - Not possible to forbid interrupt on other processors
  - Use variable lock method ⇒ busy waiting, however waiting time is short (10 commands)

Semaphore object in WIN32 API

- CreateSemaphore(. . .) : Create a Semaphore
  - LPSECURITY_ATTRIBUTES lpSemaphoreAttributes
    ⇒ pointer to a SECURITY_ATTRIBUTES structure, handle can be inherited?
  - LONG InitialCount, ⇒ initial count for Semaphore object
  - LONG MaximumCount, ⇒ maximum count for Semaphore object
  - LPCTSTR lpName ⇒ Name of Semaphore object

  Example: CreateSemaphore(NULL,0,1,NULL);
  - Return HANDLE of Semaphore object or NULL
- WaitForSingleObject(HANDLE h, DWORD time)
- ReleaseSemaphore (. . .)
  - HANDLE hSemaphore, ⇐ handle for a Semaphore object
  - LONG lReleaseCount, ⇐ increase semaphore object's current count
  - LPLONG lpPreviousCount ⇐ pointer to a variable to receive the previous count
- Example: ReleaseSemaphore(S, 1, NULL);

Example

```
#include <windows.h>
#include <stdio.h>
int x = 0, y = 1;
HANDLE S1, S2;
void T1();
void T2();
int main(){
    HANDLE h1, h2;
    DWORD ThreadId;
    S1 = CreateSemaphore( NULL,0, 1,NULL);
    S2 = CreateSemaphore( NULL,0, 1,NULL);
    h1 = CreateThread(NULL,0,T1, NULL,0,&ThreadId);
    h2 = CreateThread(NULL,0,T2, NULL,0,&ThreadId);
    getch();
    return 0;
}
```

Example

```
void T1(){
    while(1){
        WaitForSingleObject(S1,INFINITE);
        x = y + 1;
        ReleaseSemaphore(S2,1,NULL);
        printf("%4d",x);
    }
}
void T2(){
    while(1){
        y = 2;
        ReleaseSemaphore(S1,1,NULL);
        WaitForSingleObject(S2,INFINITE);
        y = 2 * y;
    }
}
```

wait till this smphr is released

Given space _____ inside codes.
Where to put the wait/signal s.t.
a) Output value is 5 ? (General)
b) ... satisfies some conditions

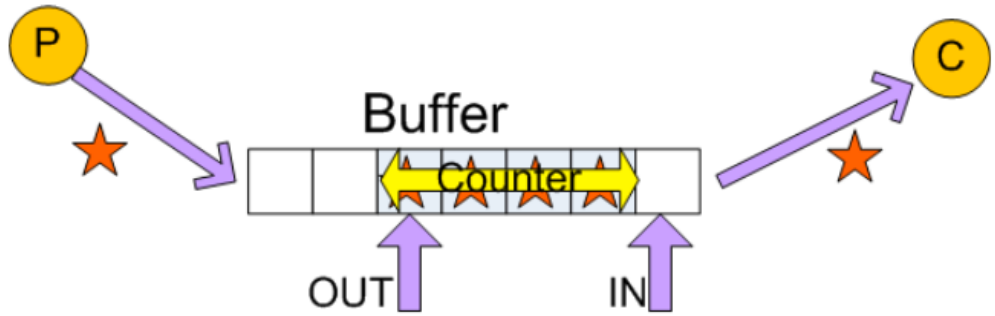
swap here

Release Semaphore

- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization examples
- Monitor

- Producer-Consumer problem
- Dining Philosophers problem
- Readers-Writers
- Sleeping Barber
- Bathroom Problem

Producer-consumer problem



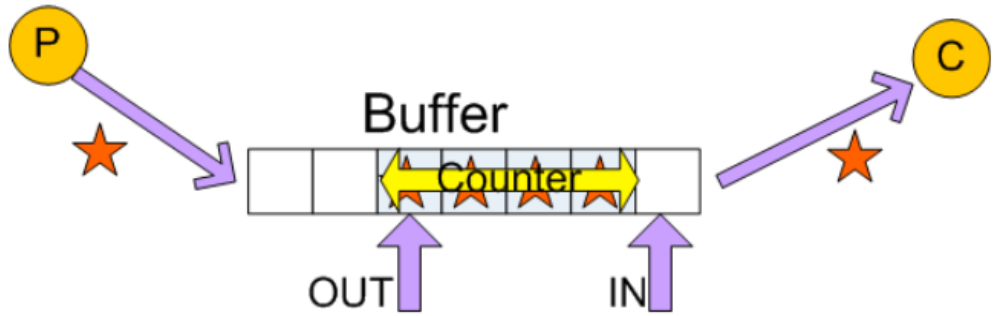

```
while(1) {
    /*produce an item in  Buffer*/
     while (Counter == BUFFER_SIZE) ;
                /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
}
```

Producer

```
while(1){
    while(Counter == 0) ;
            /*do nothing*/
     nextConsumed = Buffer[OUT];
     OUT =(OUT + 1) % BUFFER_SIZE;
    Counter--; /*consume the item in
                    nextConsumed*/
}
```

Consumer

Producer-Consumer problem




```
while(1) {
    /*produce an in  Buffer */
   if(Counter==SIZE) block();
                /*do nothing*/
    Buffer[IN] = nextProduced;
    IN = (IN + 1) % BUFFER_SIZE;
    Counter++;
if(Counter==1) wakeup(Consumer);
}
```

∃ chance consumer is put in waiting queue

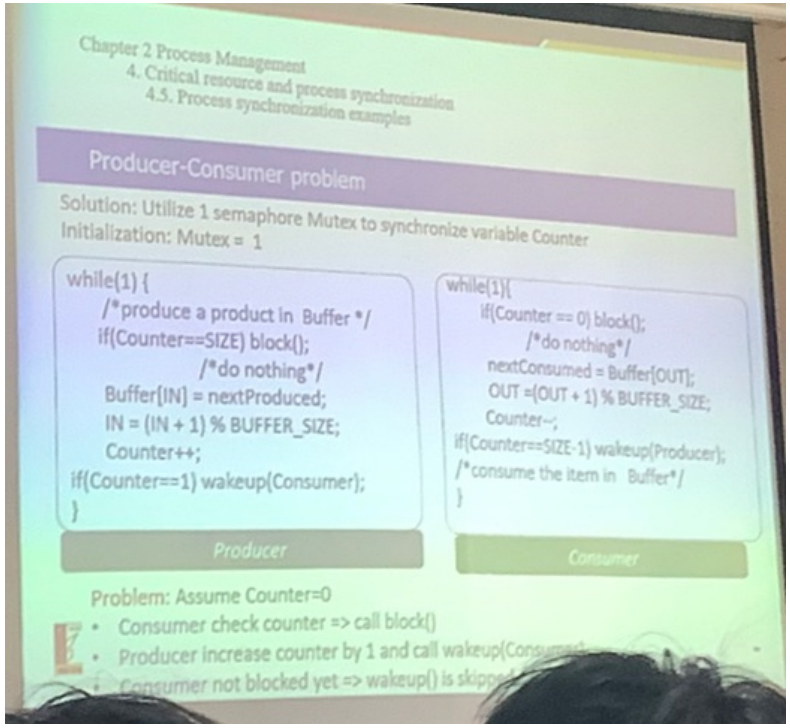
Producer

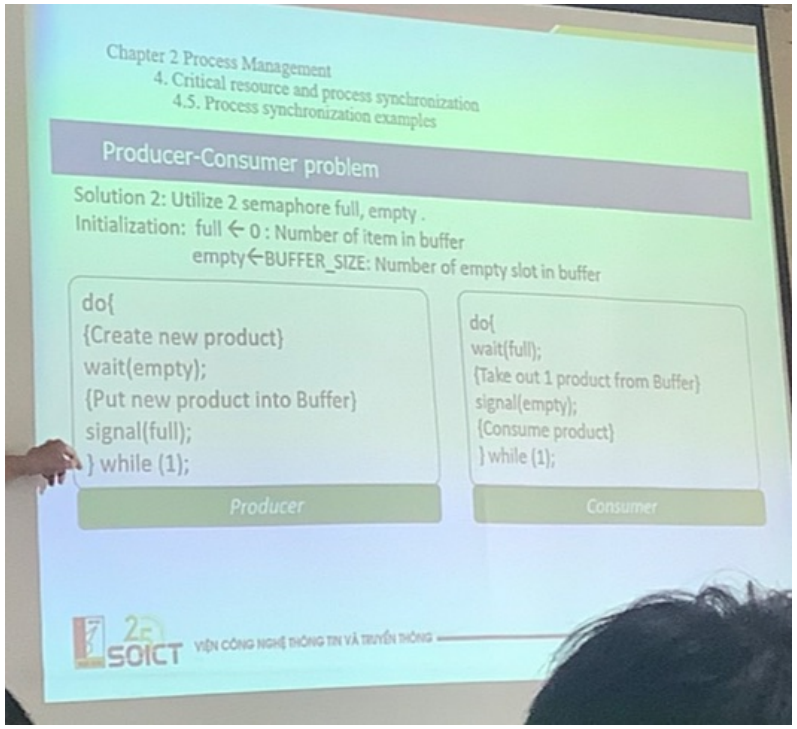
```
while(1){
    if(Counter == 0) block();
          /*do nothing*/
     nextConsumed = Buffer[OUT];
     OUT =(OUT + 1) % BUFFER_SIZE;
    Counter--;
if(Counter==SIZE-1) wakeup(Producer);
/*consume the item in   Buffer*/
}
```

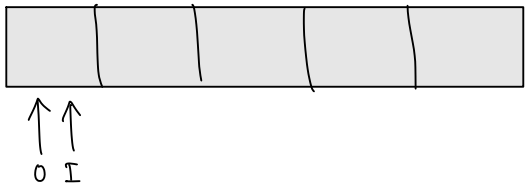
when buffer is empty

when the buffer empty

Consumer







Consumer
BLOCKED

Producer
RUNNING

empty = 5

full = -1

→ wait(empty)

Producer do NOT have to wait

↓

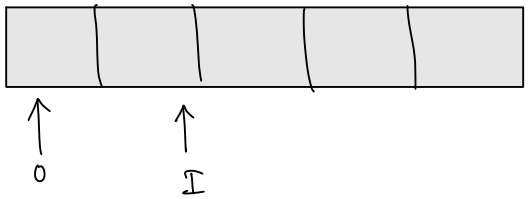
put product into buffer

↓

decrease: empty = 4

↓

signal(full)

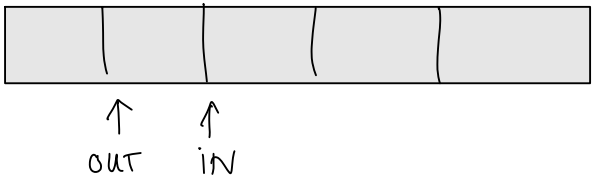


empty = 4

full = 1

Consumer
RUNNING

Producer
RUNNING



empty = -1

full = 5

Consumer
running

PRODUCER
blocked

Expand the problem

What if there are several consumers & producers?

↳ the pointers IN / OUT are now critical resources

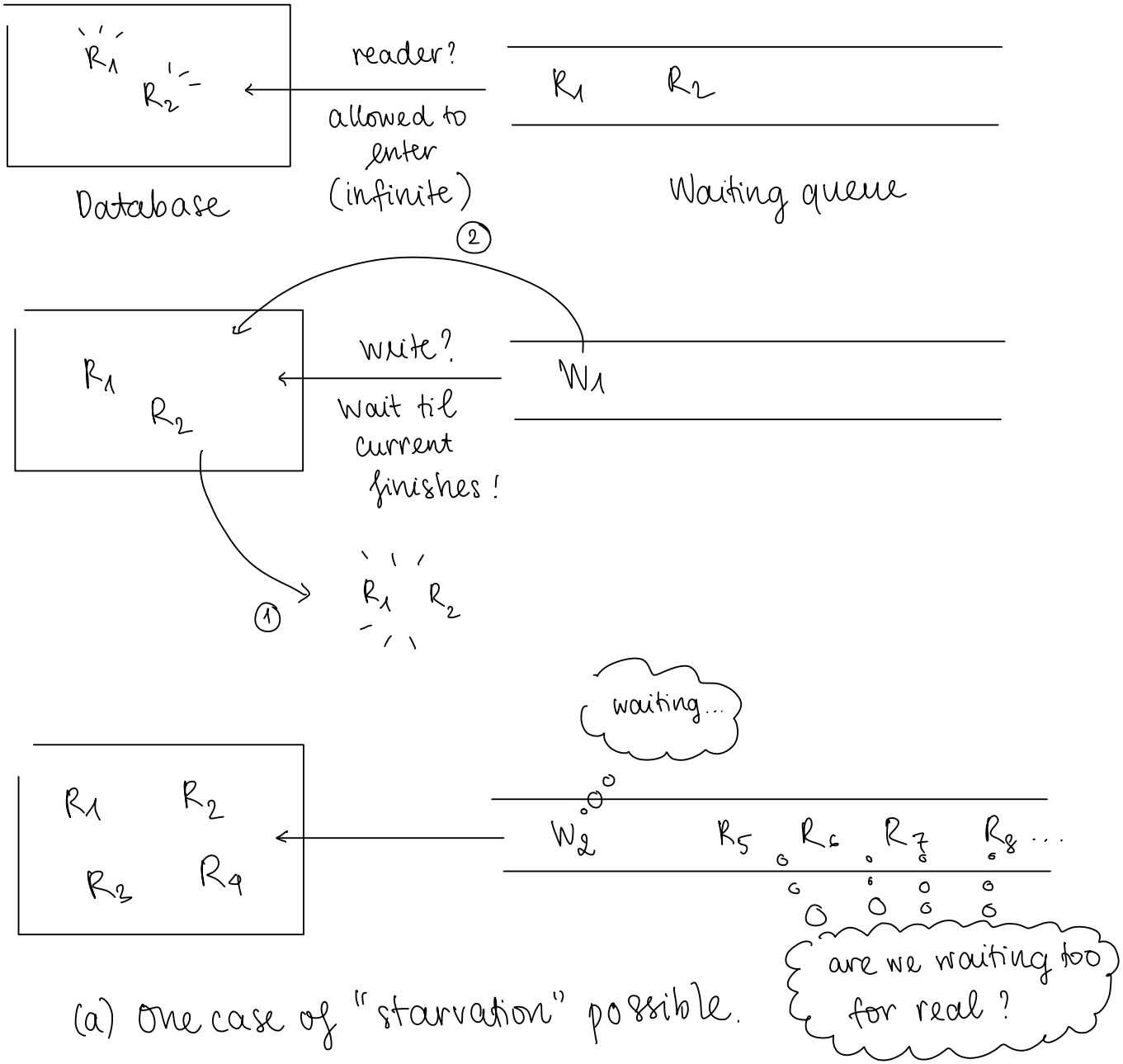
How to sync them?

↳ Can use semaphore i.e. Only 1 process can input / consume products at a time

Readers and Writers problem

do not modify the database

- Many Readers processes access the database at the same time
- Several Writers processes update the database
- Allow unlimited Readers to access the database
  - (1) One Reader process is accessing the database, new Reader process can access the database
  - (2) (Writers processes has to stay in waiting queue)
- Allow only one Writer process to update the database at a time
- Non-preemptive problem. Process stays inside critical section without being interrupted



(a) One case of "starvation" possible.

there are ways to solve this problem

↳ This is one of the project's topics!

↳ Demonstrate w/ different types of processes

↳ Implement the solution!

Sleeping barber

- N waiting chair for client (limited)
- Barber can cut for one client at a time (limited)
- No client, barber go to sleep
- When client comes
  - If barber is sleeping ⇒ wake him up
  - If barber is working
    - Empty chair exists ⇒ sit and wait
    - No empty chair left ⇒ Go away

How to solve this problem? One suggestion:

Test x Set?

① Check the right to get haircut

② Check the right to have a waiting seat

↳ Project's topics

Bathroom Problem

- A bathroom is to be used by both men and women, but not at the same time
- If the bathroom is empty, then anyone can enter
- If the bathroom is occupied, then only a person of the same sex as the occupant(s) may enter
- The number of people that may be in the bathroom at the same time is limited (if bathroom is full, same gender ✓ still have to wait)

- Problem implementation require to satisfy constraints
  - 2 types of process: male() và female()
  - Each process enter the Bathroom in a random period of time

Solution suggestion

These processes may share the same memory

→ Variable lock :

Different gender, check the lock of the other gender

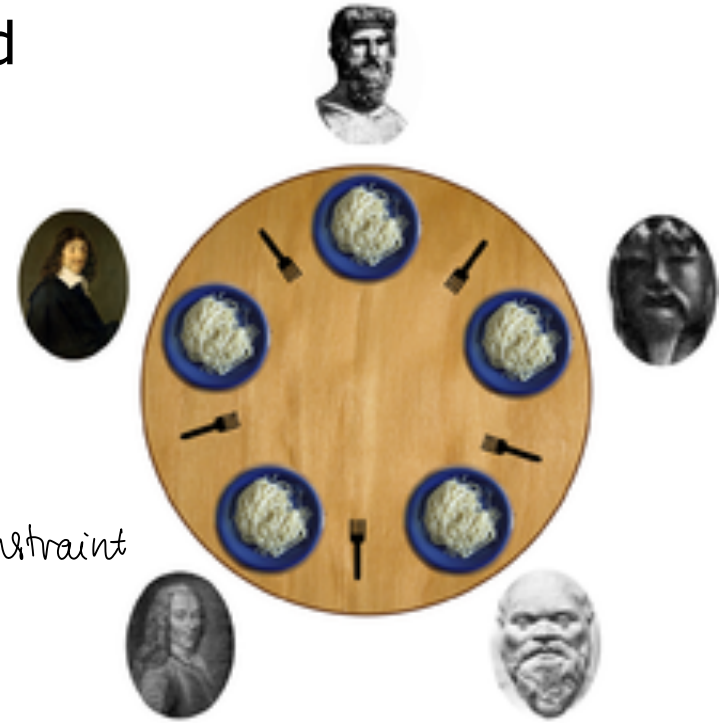
→ Semaphore

Same gender, check for right to enter

Dining philosopher problem

Classical synchronization problem, show the situation where many processes share resources

- 5 philosophers having dinner at a round table
- In front each person is a disk of spaghetti
- Between two disk is a fork
- Philosopher do 2 things :Eat and Think
- Each person need two forks to eat
- Take only one fork at a time
- Take the left fork then the right fork
- Finish eating, return the fork to original place



Dining philosopher problem: Simple method

- Each fork is a critical resource, synchronized by a semaphore fork[i]
- Semaphore fork[5] = {1, 1, 1, 1, 1};
- Algorithm for philosopher Pi

```
do{
 wait(fork[i])
     wait(fork[(i+1)% 5]);
          {Eat}
     signal(fork[(i+1)% 5]);
 signal(fork[i]);
       {Thinks}
} while (1);
```

- If all the philosophers want to eat
  - Take the left fork (call to: wait(fork[i]))
  - Wait for the right fork (call to: wait(fork[(i+1)%5]))

⇒ deadlock

waiting forever
(all have left fork, none have right fork)

Dining philosopher problem – Solution 1

- Allow only one philosopher to take the fork at a time
- Semaphore mutex ← 1;
- Algorithm for philosopher Pi

```
do{
   wait(mutex)
      wait(fork[i])
      wait(fork[(i+1)% 5]);
   signal(mutex)
         {Eat}
      signal(fork[(i+1)% 5]);
      signal(i);
         {Thinks}
} while (1);
```

- It's possible to allow 2 non-close philosopher to eat at a time (P1: eats, P2: owns mutex⇒ P3 waits)

Dining philosopher problem – Solution 1

- Philosopher take the forks with different order
- Even id philosopher take the even id fork first
- Odd id philosopher take the odd id fork first

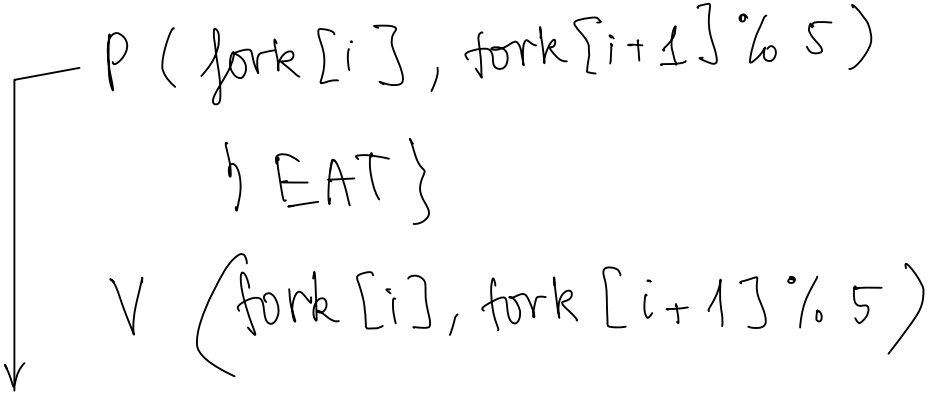
→ does not satisfy constraint
left fork first, then right fork

```
do{
   j = i%2
   wait(fork[(i + j)%5])
      wait(fork[(i+1 - j)% 5]);
         {Eat}
      signal(fork[(i+1 - j)% 5]);
   signal((i + i)%5);
         {Thinks}
} while (1);
```

- Solve the deadlock problem

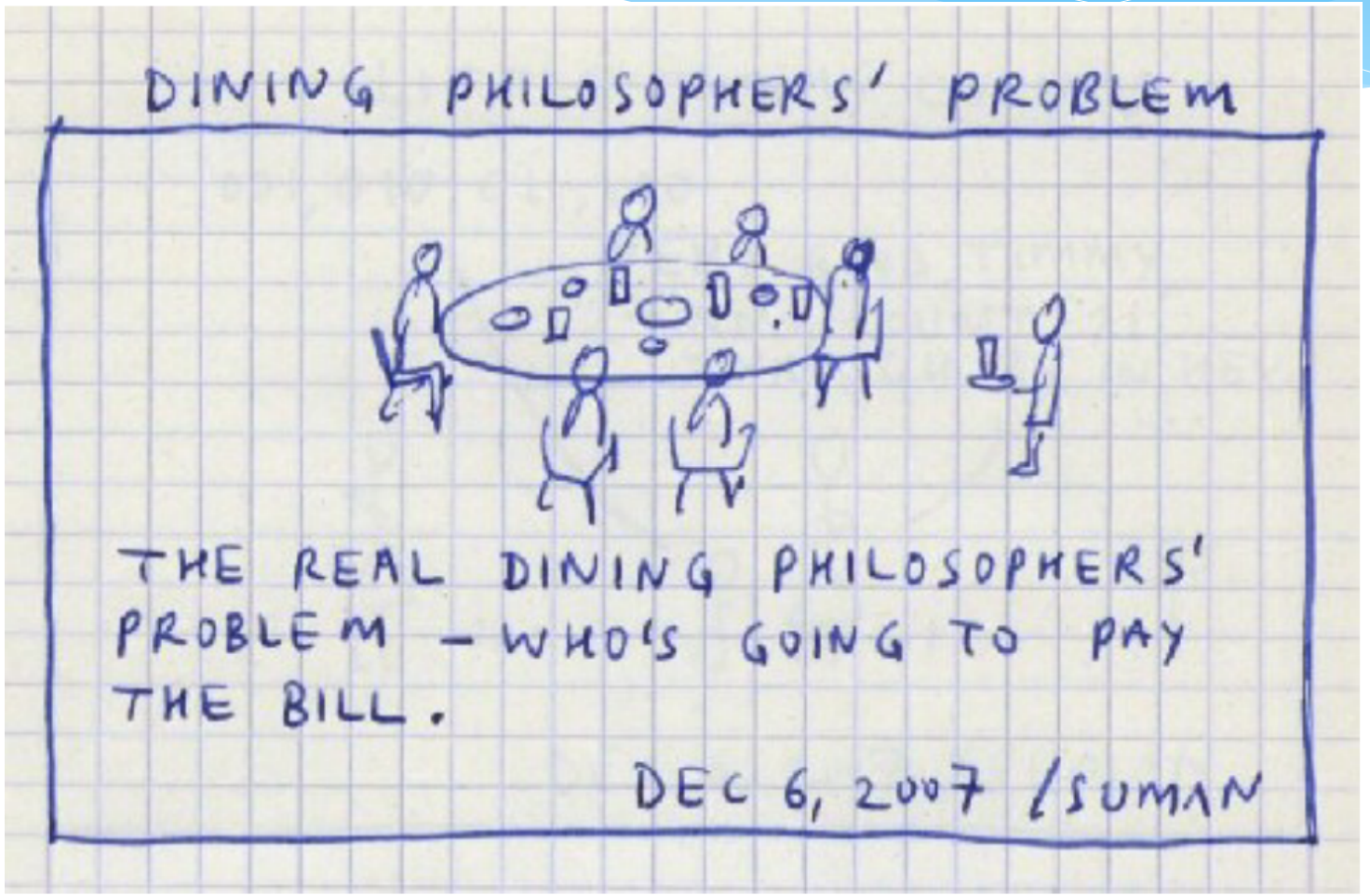
mutex = mutual exclusion

Another solution (NOT possible to implement IRL)



↳ 1 person trying to pick up BOTH folks at the same time

True problem ?



IRL Problem

① Withdraw money from bank.

↳ If the remainder is not sync ; I can withdraw more than possible

i.e. 1000 $ from 3 devices at the same time → 3000 $

(!) remain: 1000 $

② Ticket purchasing for concert

↳ If the ws of available tickets is not sync ...

⇒ Ordered tickets may far exceeded the possible slots

- Critical resource
- Variable lock method
- Test and Set method
- Semaphore mechanism
- Process synchronization examples
- Monitor

Introduction

- Special data type, proposed by HOARE 1974
- Combines of procedures, local data, initialization code
- Process can only access variables via procedures of Monitor
- Only one process can work with Monitor at a time
  - Other processes have to wait
- Allow process to wait inside Monitor
  - Utilize condition variable

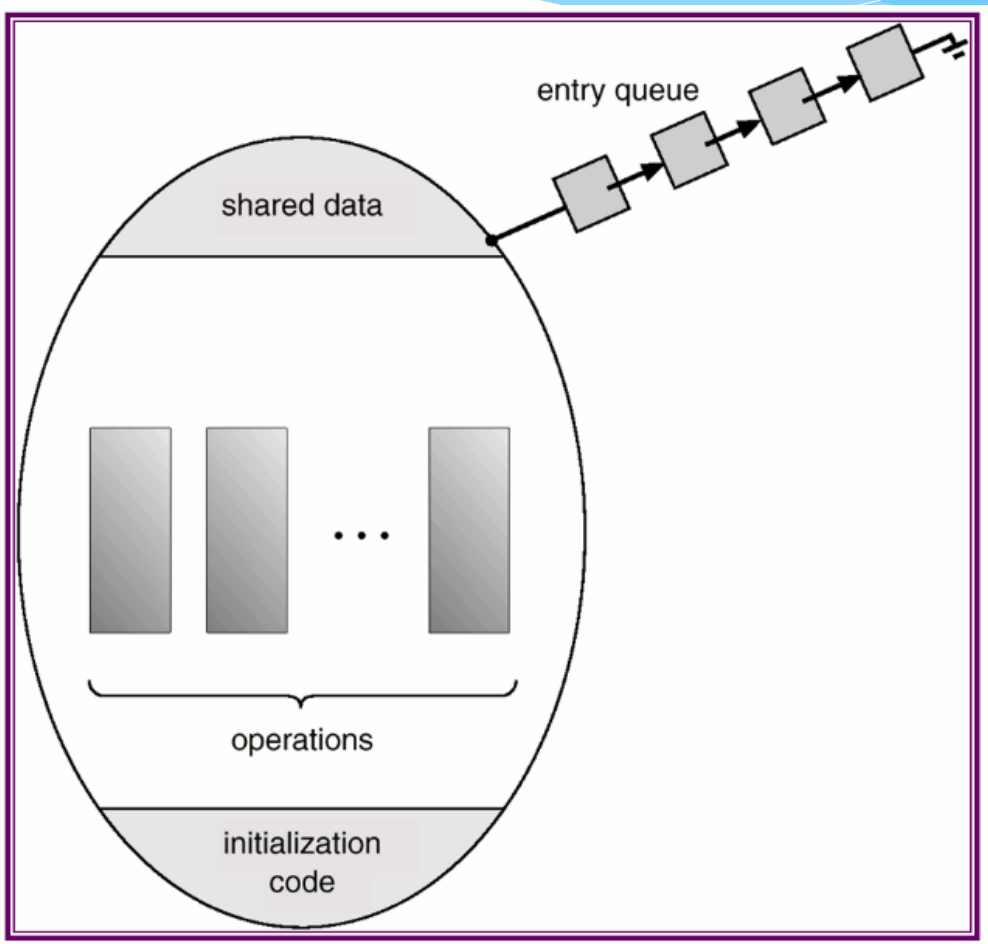
```
monitor monitorName{
    Sharing variables declarations ;
    procedure P1(...){
        ...
    }
    ...
    procedure Pn(...){
        ...
    }

    {
        Initialization code
    }
};
```

much similar to OOP structure (though around that time, OOP haven't existed)

Monitor's syntax

Model



Condition Variable

Actually name of a queue

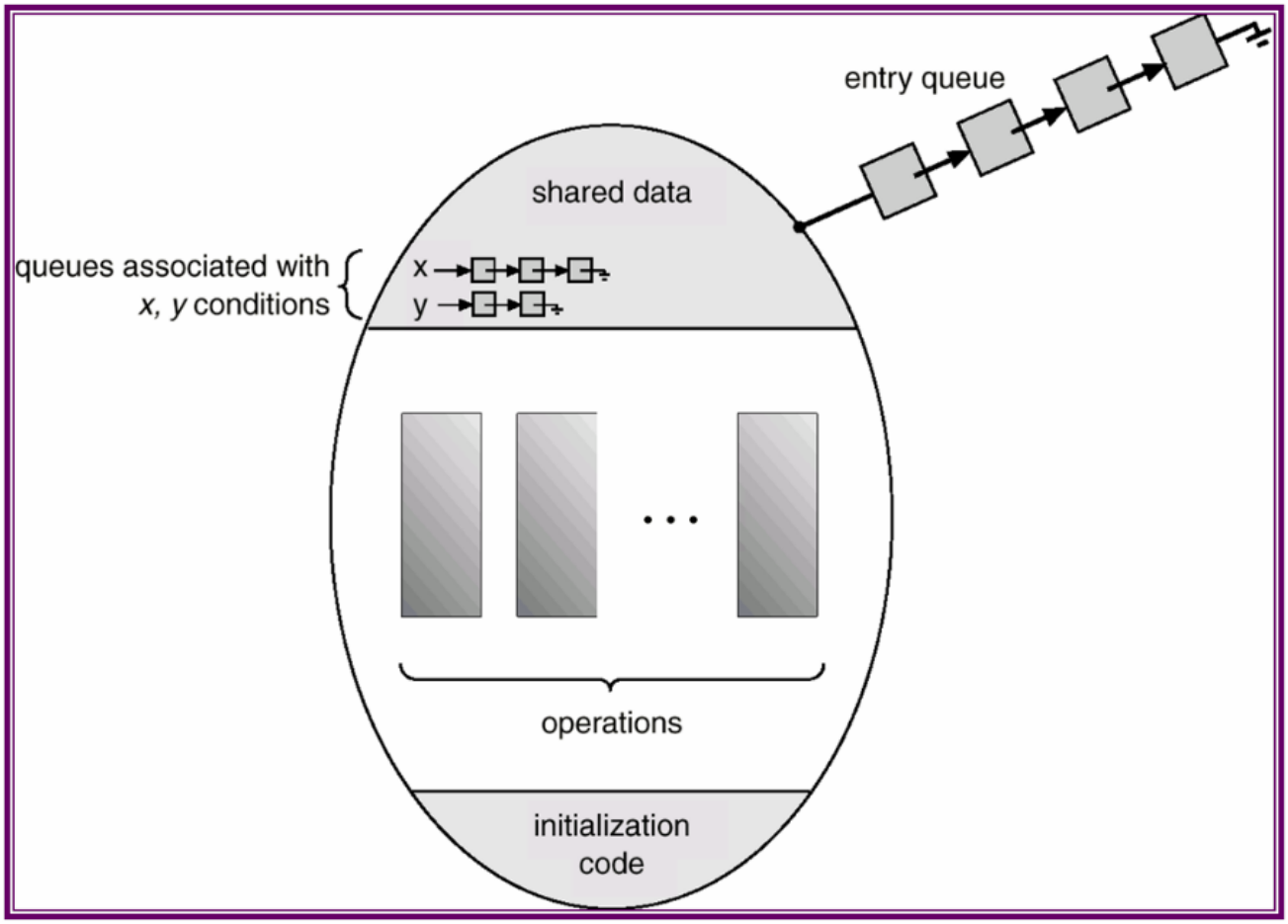
Declare: **condition x,y**;

Only used by 2 operations

wait() Called by Monitor's procedures ***(syntax x.wait() or wait(x))***. Allow process to be blocked until activated by other process via **signal()** procedure

signal() Called by Monitor's procedures **(syntax x.signal() or signal(x))**. Activate a process waiting at *x* variable queue. If no waiting process then the operation is skipped

Model


entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

Monitor's usage: sharing a resource

```
Monitor Resource{
  Condition Nonbusy;
  Boolean Busy
//-- 1      User's part      ; --
  void Acquire(){
    if(busy) Nonbusy.wait();
      busy=true;
  }
  void Release(){
    busy=false
    signal(Nonbusy)
  }
//--  Initialization part  ----
  busy= false;
  Nonbusy = Empty;
}
```

Process's structure

```
while(1){
  ...
  Resource.Acquire()
  { Using resource }
  Resource.Release()
  ...
}
```

try to use the resource

Producer – Consumer problem

```
Monitor ProducerConsumer{
  Condition Full, Empty;
  int Counter ; (shared variable)
    void Put(Item){
      if(Counter=N) Full.wait();
      { Put Item into Buffer };
      Counter++;
      if(Counter=1)Empty.signal()
  }
  void Get(Item){
    if(Counter=0) Empty.wait()
        {Take Item from Buffer}
    Counter--;
    if(Counter=N-1)Full.signal()
  }
  Counter=0;
  Full, Empty = Empty;
}
```

ProducerConsumer M;

Producer

```
while(1){
    Item = New product
    M.Put(Item)
    ...
}
```

Consumer

```
while(1){
    M.Get(&Item)
     {Use Item}
    ...
}
```

① Process
② Thread
③ CPU scheduling
④ Critical resource and process synchronization
⑤ Deadlock and solutions

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

Deadlock conception

- System combines of concurrently running processes, sharing resources
  - Resources have different types (e.g.: CPU, memory,..).
  - Each type of resource may has many unit (e.g.: 2 CPUs, 5 printers..)
- Each process is combines of sequences of continuous operations
  - Require resource: if resource is not available (being used by other processes) $\Rightarrow$ process has to wait
  - Utilize resource as required (printing, input data...)
  - Release allocated resources
- When processes share at least 2 resources, system may "in danger"

Deadlock conception

- Example: Two processes in the system P_1 & P_2
  - P_1 & P_2 share 2 resources R_1 & R_2
  - R_1 is synchronized by semaphore S_1 ($S_1 \leftarrow 1$)
  - R_2 is synchronized by semaphore S_2 ($S_2 \leftarrow 1$)
  - Code for P_1 and P_2

initial values

```
P(S1)
P(S2)
 { Use R1&R2 }
V(S1)
V(S2)
```

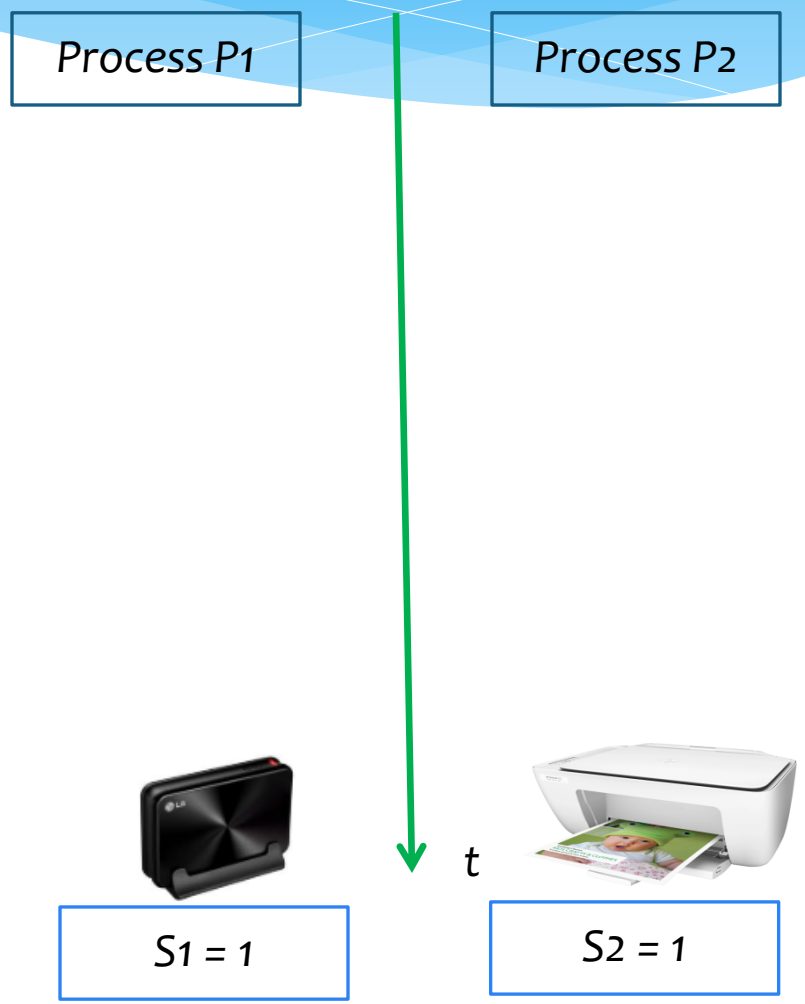
Example

Process P1

P(S_1)
P(S_2)
{ Use R_1&R_2 }
V(S_1)
V(S_2)

Process P2

P(S_1)
P(S_2)
{ Use R_1&R_2 }
V(S_1)
V(S_2)



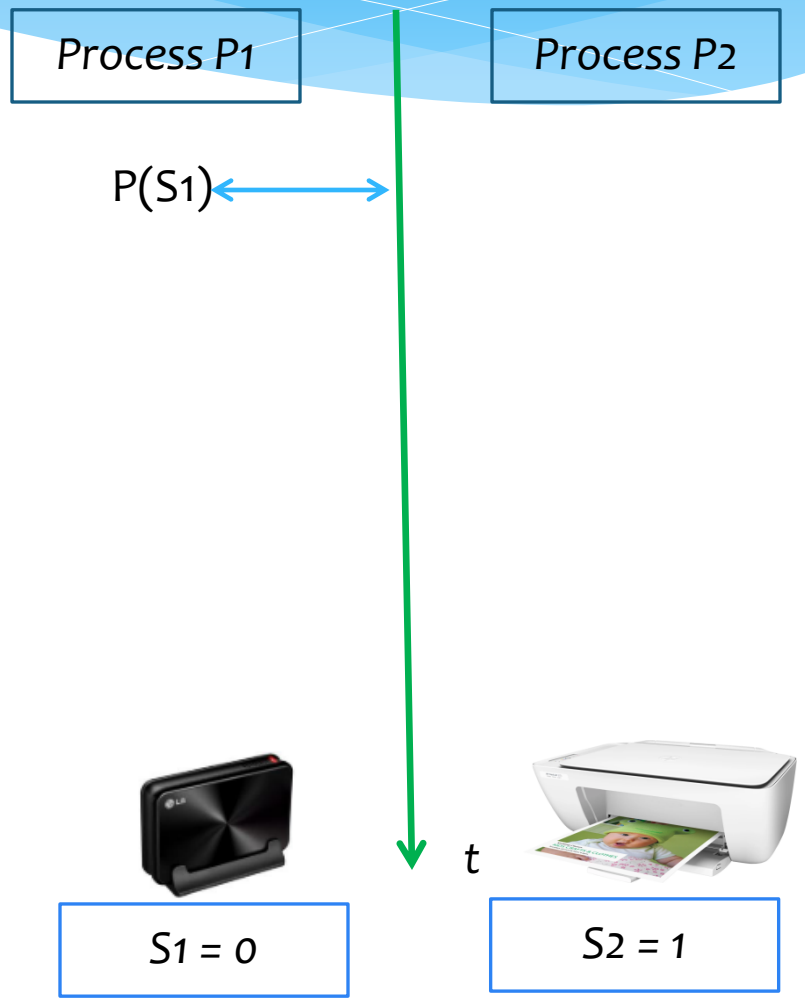
Example

Process P1

P(S_1)
P(S_2)
{ Use R_1&R_2 }
V(S_1)
V(S_2)

Process P2

P(S_1)
P(S_2)
{ Use R_1&R_2 }
V(S_1)
V(S_2)



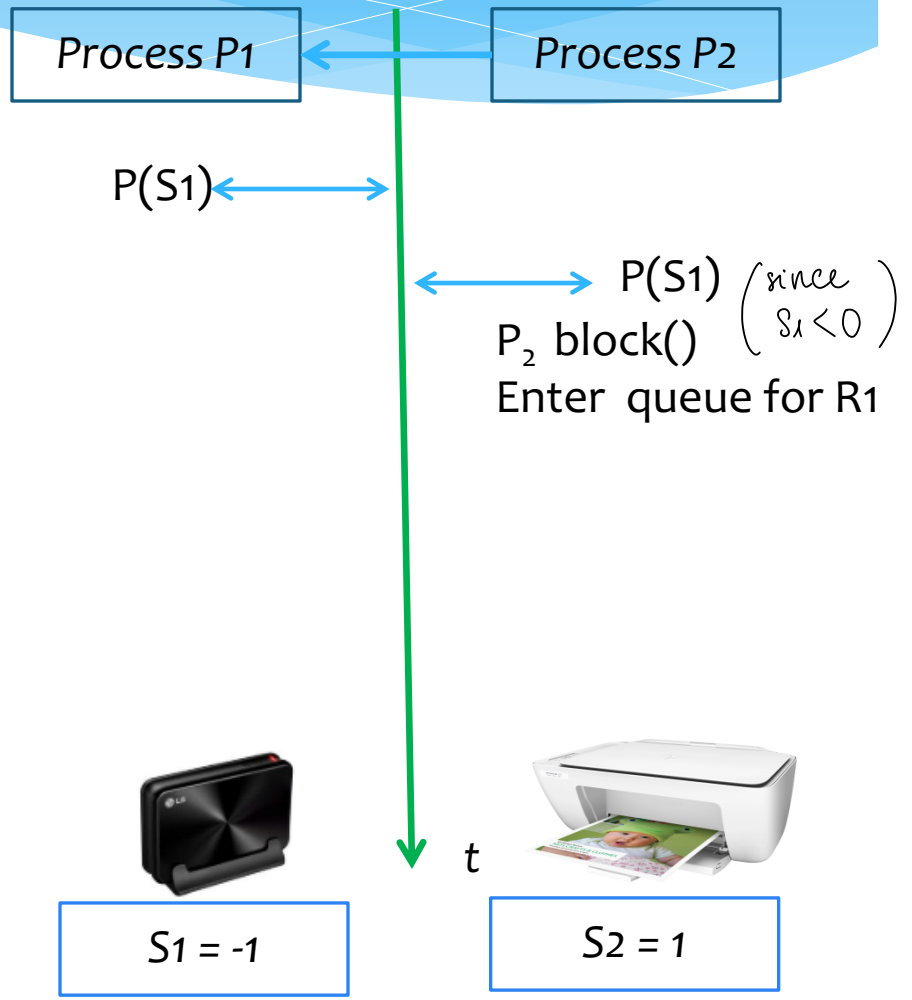
Example

Process P1

```
P(S1)
P(S2)
 { Use R1&R2 }
V(S1)
V(S2)
```

Process P2

```
P(S1)
P(S2)
 { Use R1&R2 }
V(S1)
V(S2)
```



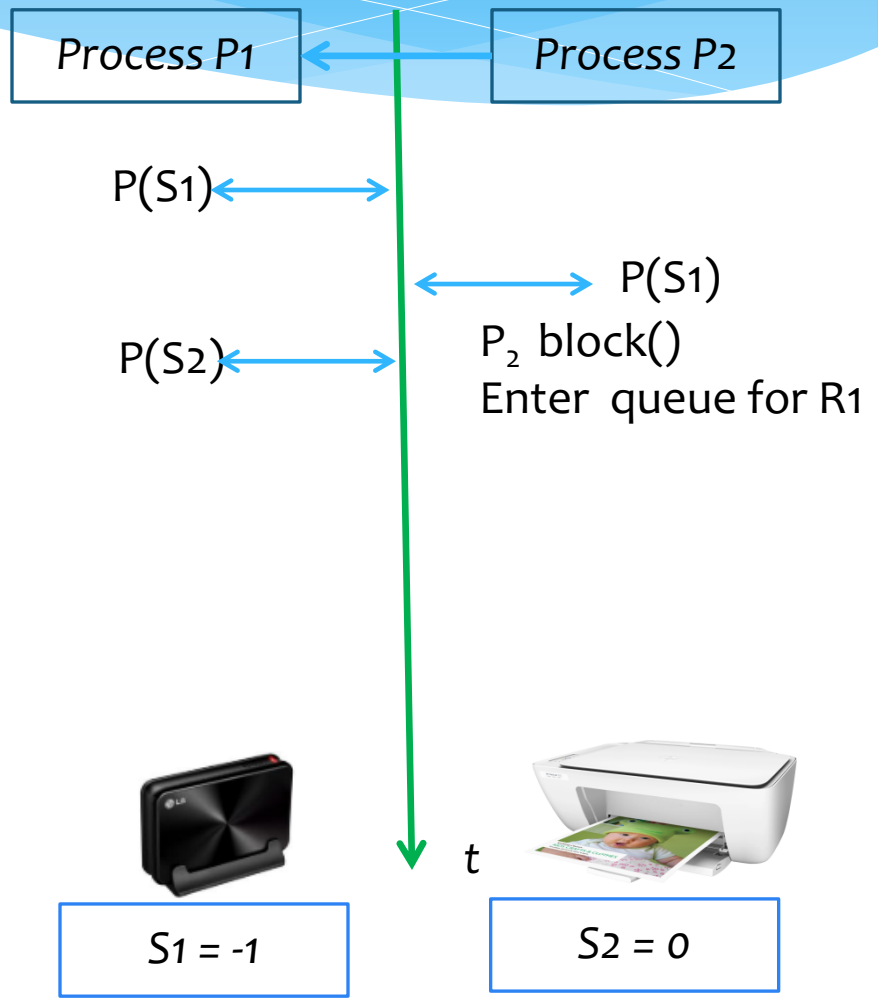
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



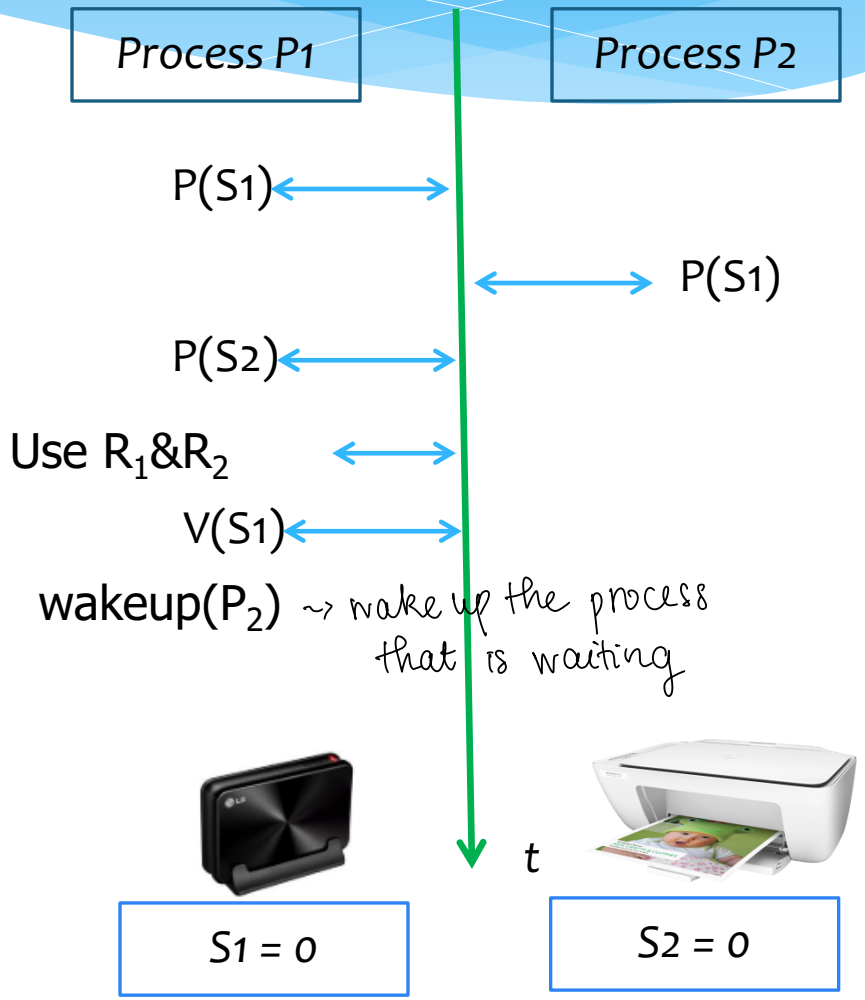
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



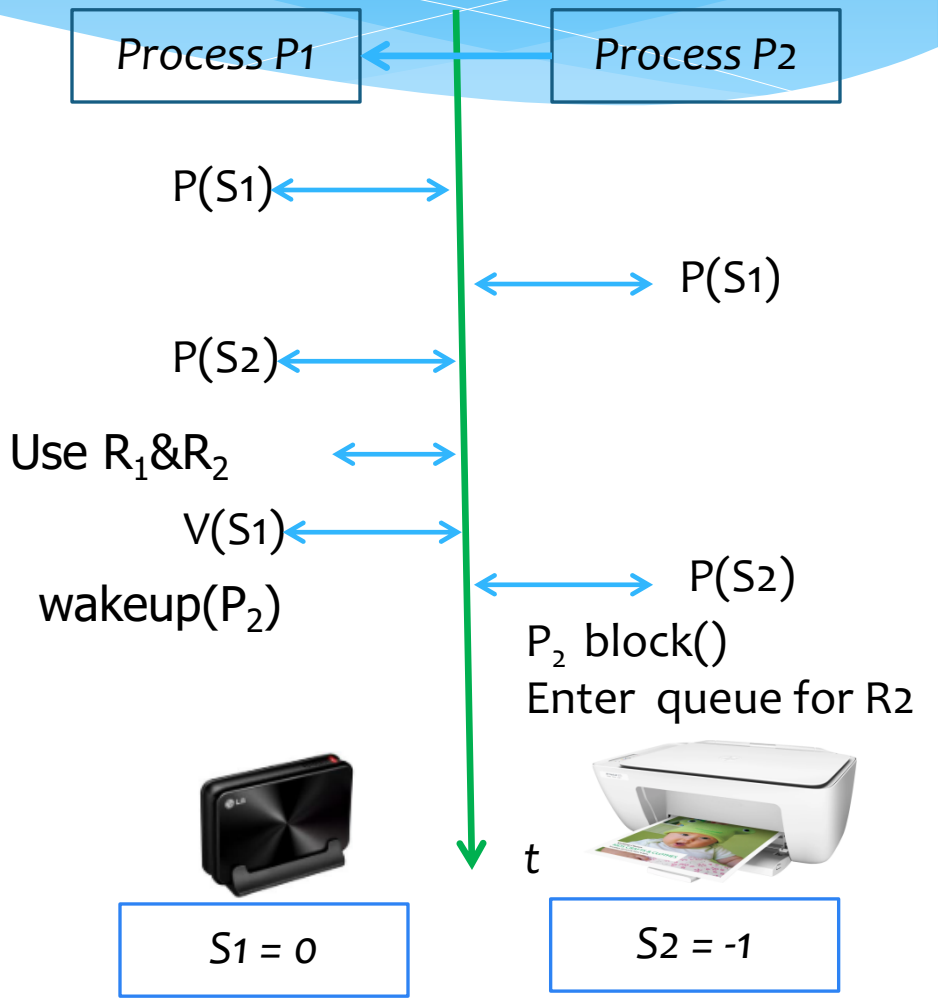
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



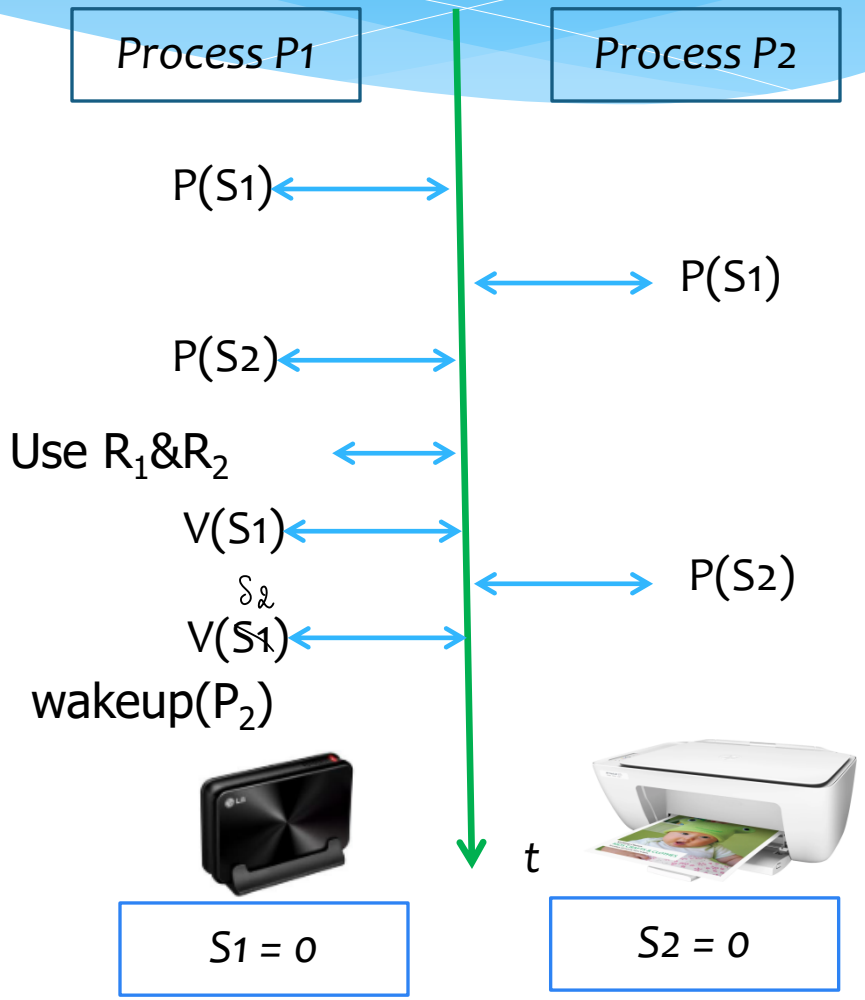
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



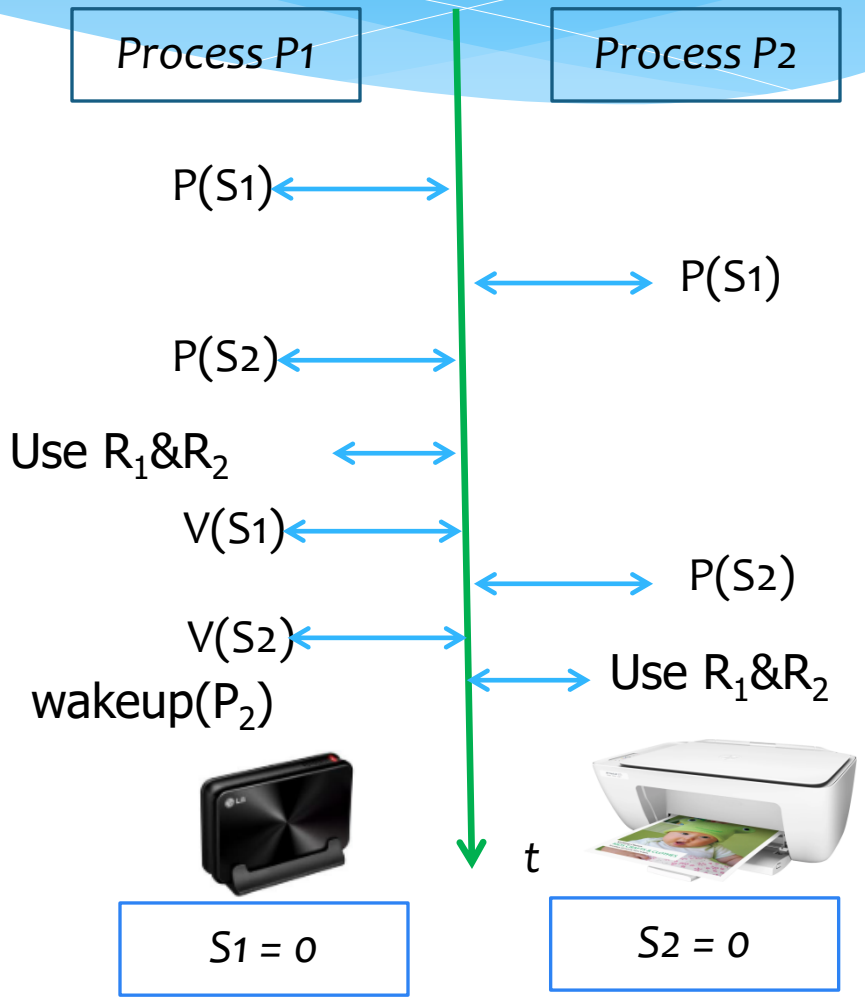
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



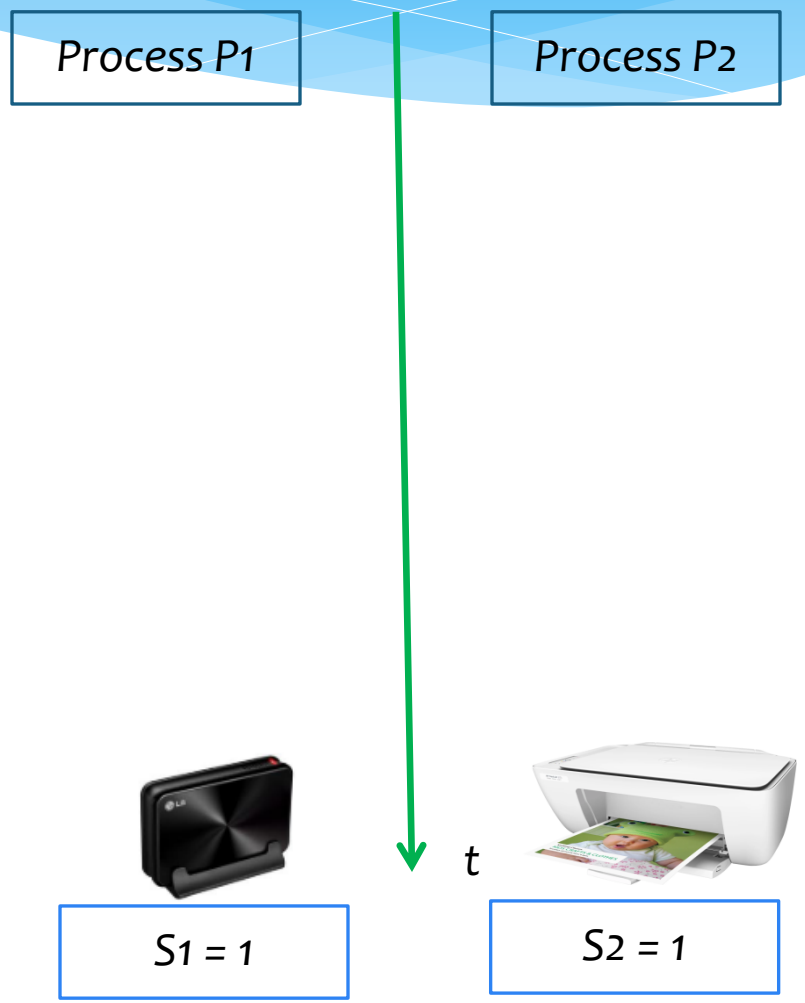
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



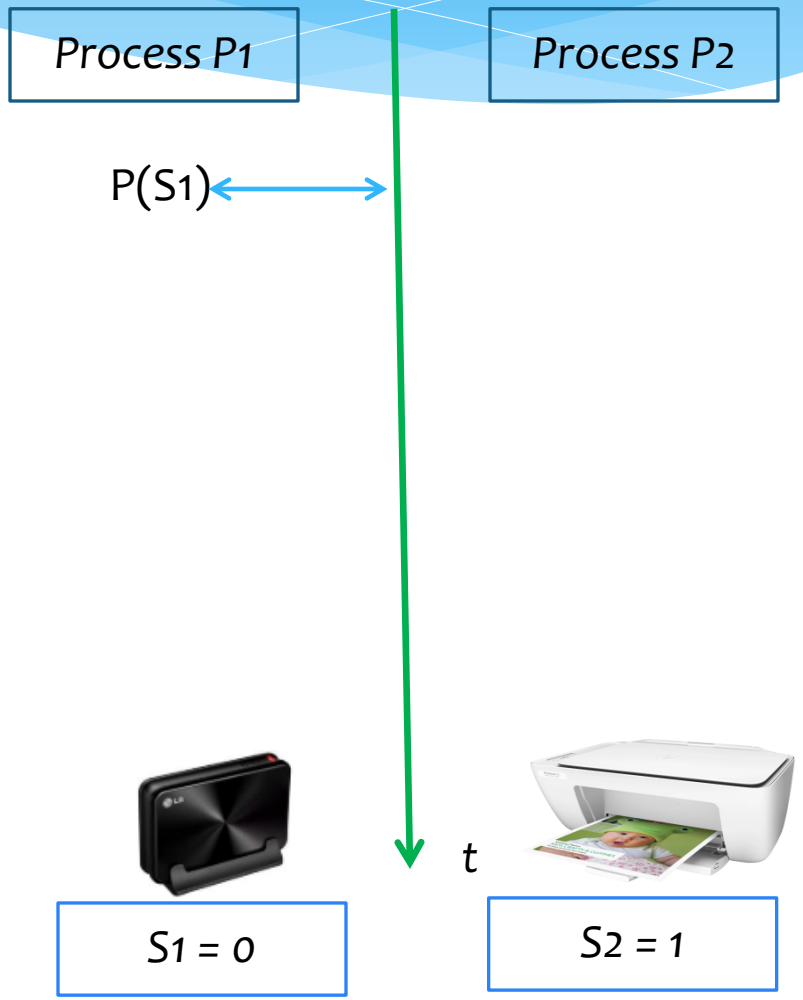
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



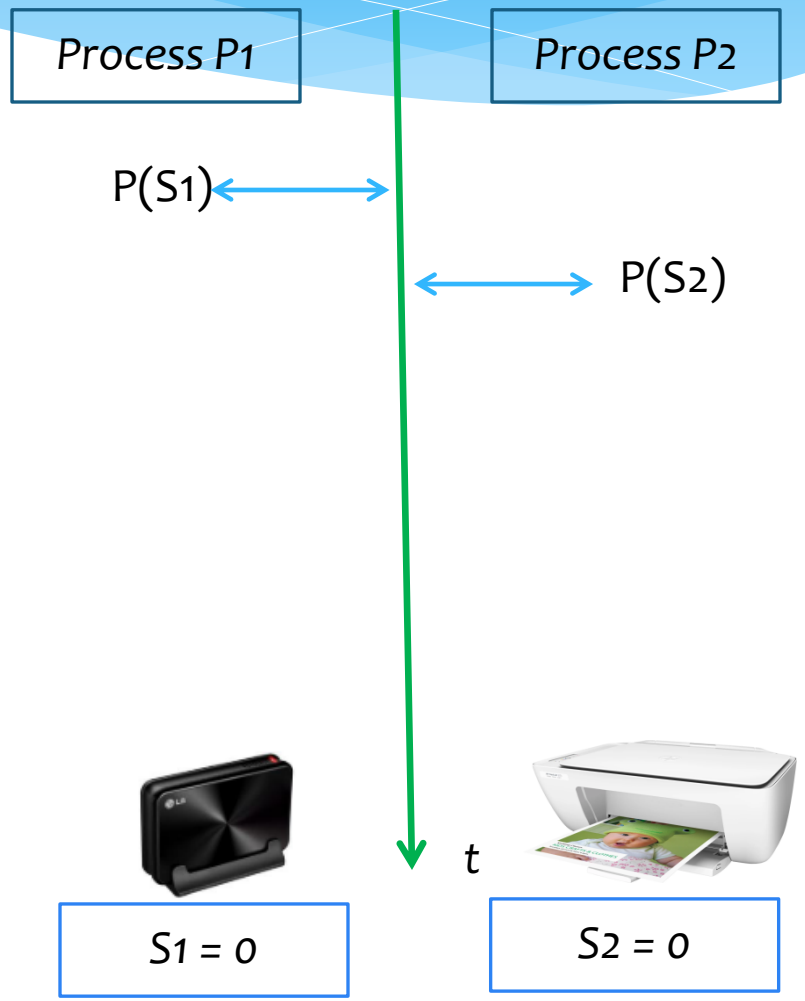
Example

Process P1

P(S_1)
P(S_2)
{ Use R_1&R_2 }
V(S_1)
V(S_2)

Process P2

P(S_2)
P(S_1)
{ Use R_1&R_2 }
V(S_1)
V(S_2)



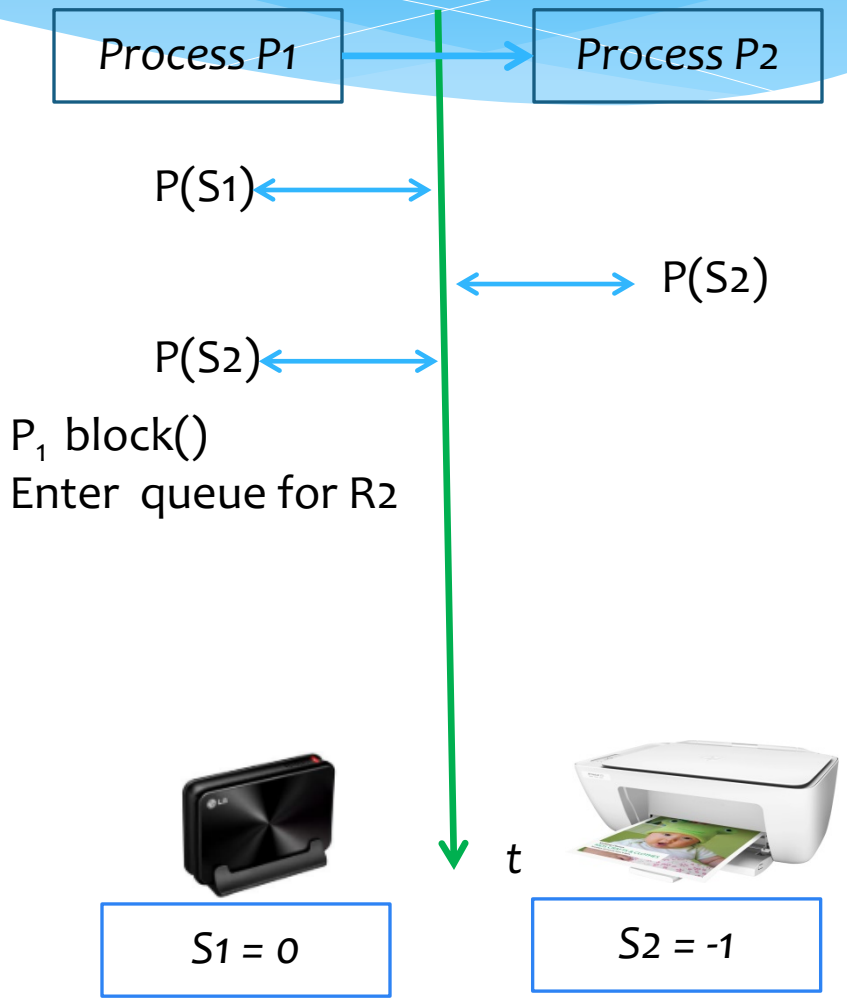
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



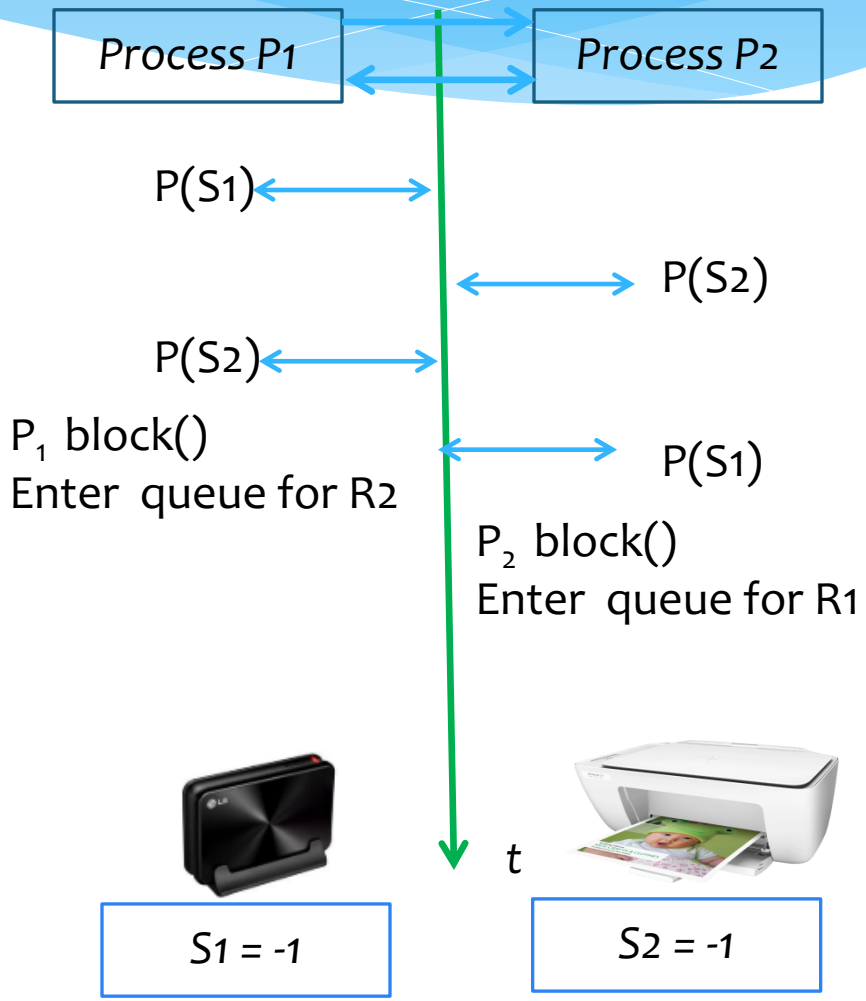
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



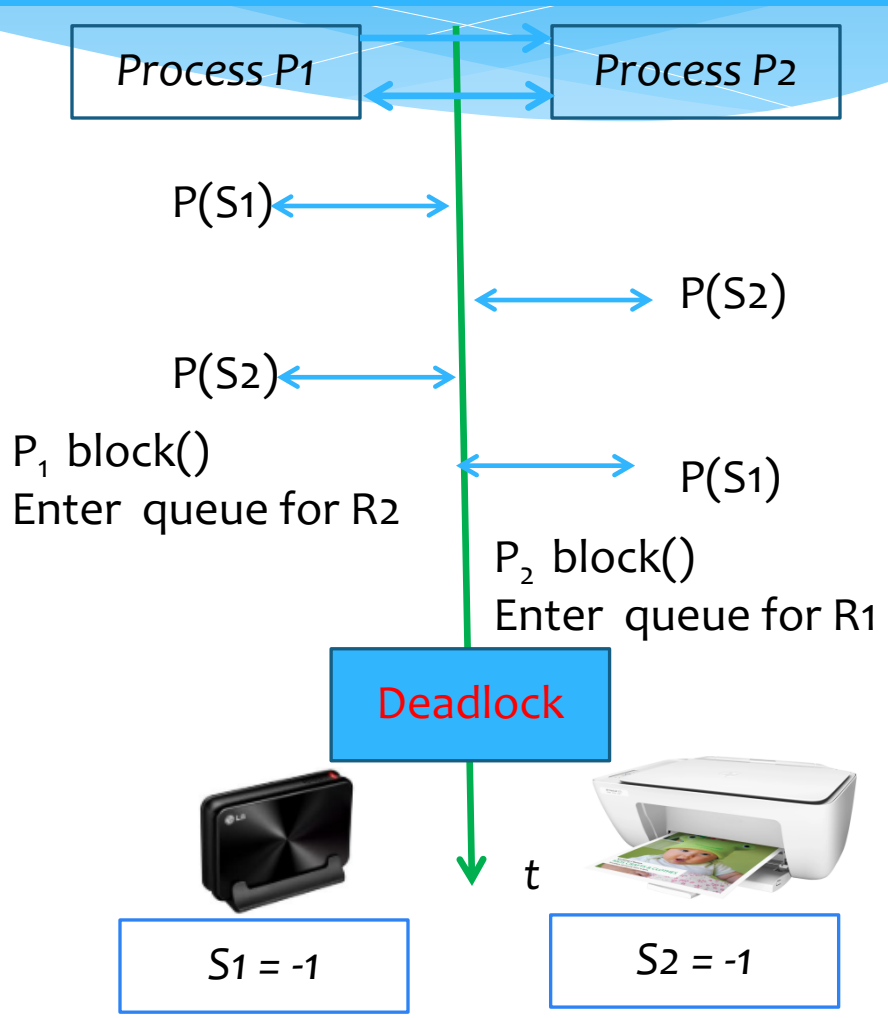
Example

Process P1

$P(S_1)$
$P(S_2)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$

Process P2

$P(S_2)$
$P(S_1)$
{ Use R_1&R_2 }
$V(S_1)$
$V(S_2)$



Definition

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

same

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

Conditions

4 conditions, must occur at the same time

- Critical resource
  - Resource is used in a non-shareable model
    - Only one process can use resource at a time
    - Other process request to use resource ⇒ request must be postponed until resource is released
- Wait before enter the critical section
  - Process can not enter critical section has to wait in queue
  - Still own resources while waiting
- No resource reallocation system
  - Resource is non-preemptive
  - Resource is released only by currently using process after this process finished its task
- Circular waiting
  - Set of processes $\{P_0, P_2, \ldots, P_n\}$ waiting in a order: $P_0 \rightarrow R_1 \rightarrow P_1$; $P_1 \rightarrow R_2 \rightarrow P_2$; $\ldots P_{n-1} \rightarrow R_n \rightarrow P_n$; $P_n \rightarrow R_0 \rightarrow P_0$
  - Circular waiting create nonstop loop

Example: Dining philosopher problem

- Critical resource
- Wait before enter critical section
- Non-preemptive resource
- Circular waiting

- Deadlock conception
- Conditions for resource deadlocks
- **Solutions for deadlock**
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

4th method → rare, low-cost deadlock

Methods

- Prevention
  - Apply methods to guarantee that the system never has deadlock
  - Expensive
  - Apply for system that deadlock happens frequently and once it happen the cost is high
- Avoidance
  - Check each process's resource request and reject request if this request may lead to deadlock
  - Require extra information
  - Apply for system that deadlock happens least frequently and once it happen the cost is high
- Deadlock detection and recovery
  - Allow the system work normally $\Rightarrow$ deadlock may happen
  - Periodically check if deadlock is happening
  - If deadlock apply methods to remove deadlock
  - Apply for system that deadlock happens least frequently and once it happen the cost is low

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- **Deadlock prevention**
- Deadlock avoidance
- Deadlock detection and recovery

Rule

Attack 1 of 4 required conditions for deadlock to appear

Critical resource

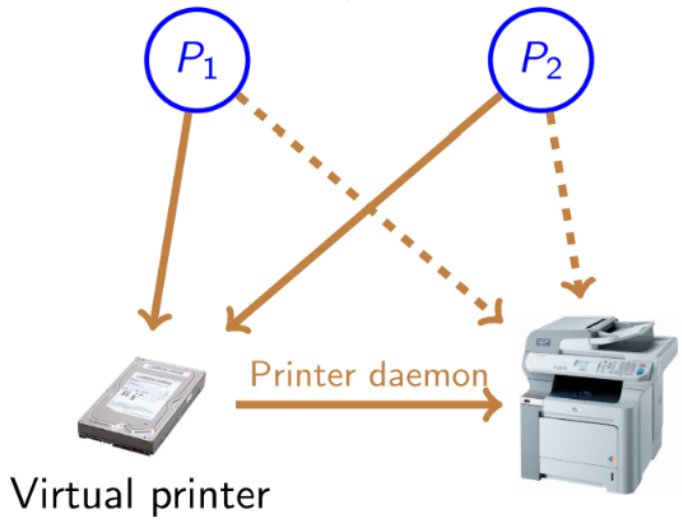
Wait before entering critical section

Non-preemptive resource

Circular wait

Critical resource condition

- Reduce the system's critical degree
  - Shareable resource(read-only ffile): accessed simultaneously
  - Non-shareable resource: Cannot be accessed simultaneously
- SPOOL mechanism(*Simultaneous peripheral operation on-line*)
  - Do not allocate resource when it's not necessary
  - A limited number of processes can request resourcen



- Only process *printer daemon* work with printer ⇒ Deadlock for resource printer is canceled
- Not any resource can be used with SPOOL

Wait before entering critical section condition

Rule: Make sure one process request resource only when it doesn't own any other resources

- Prior allocate
  - processes request all their resources before starting execution and only run when required resources are allocated
  - Effectiveness of resource utilization is low
    - Process only use resource at the last cycle?
    - Total requested resource higher than the system's capability?
- Resource release
  - Process releases all resource before apply(re-apply) new resource
    - Process execution's speed is low
    - Must guarantee that data kept in temporary release resource won't be lost

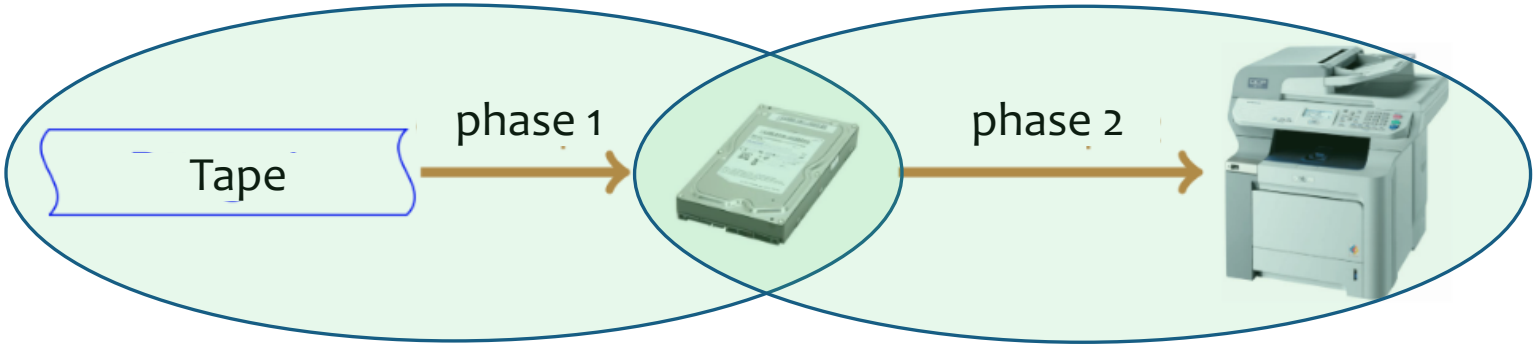
Wait before entering critical section condition - Example

- Process combines of 2 phases
  - Copy data from tape to a file in the disk
  - Arrange the data in file and bring to printer

- Prior allocation method
  - Request both tape, file and printer
  - Waste printer in first phase, tape in the second phase

Wait before entering critical section condition - Example

- Process combines of 2 phases
  - Copy data from tape to a file in the disk
  - Arrange the data in file and bring to printer



- Prior allocation method
  - Request both tape, file and printer
  - Waste printer in first phase, tape in the second phase
- Resource release method
  - Request tape and file for phase 1
  - Release tape and file
  - Request file and printer for phase 2

No preemption resource condition

Rule: allow process to preempt resource when it's necessary

- Process P_i apply for resource R_j
  - R_j is available: allocate R_j to P_i
  - R_j not available: (R_j is being used by process P_k)
    - P_k is waiting for another resource
      - Preempt R_j from P_k and allocate to P_i as requested
      - Add R_j into the list of needing resource of P_k
      - P_k is execution again when
        - Receive the needing resource
        - Take back resource R_j
    - P_k is running
      - P_i has to wait (*no resource release*)
      - Allow resource preempt only when it's necessary

No preemption resource condition

Rule: allow preemptive when it's necessary

- Only applied for resources that can be store and recover easily
  - (*CPU, memory space*)
  - Difficult to apply for resource like printer

- One process is preempted many time ?

Circular wait condition

- provide a global numbering of all type of resources
  - R = {R_1, R_2, . . . R_n} Set of resources
  - Construct an ordering function f : R → N
    - Function f is constructed based on the order of resource utilization
      f(Tape) = 1
      f(Disk) = 5
      f(Printer) = 12
- Process can only request resource in an increasing order
  - Process holding resource type R_k can only request resource type R_j satisfy $f(R_j) > f(R_k)$
  - Process requests resource R_k has to release all resource R_i satisfy condition $f(R_i) \geq f(R_k)$

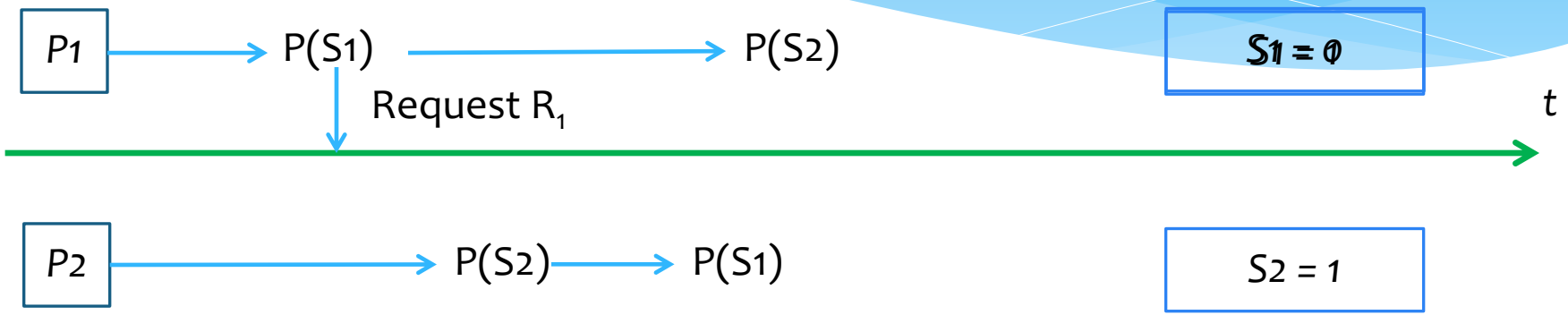
Circular wait condition

- Process can only request resource in an increasing order
  - Process holding resource type R_k can only request resource type R_j satisfy $f(R_j) > f(R_k)$
  - Process requests resource R_k has to release all resource R_i satisfy condition $f(R_i) \geq f(R_k)$

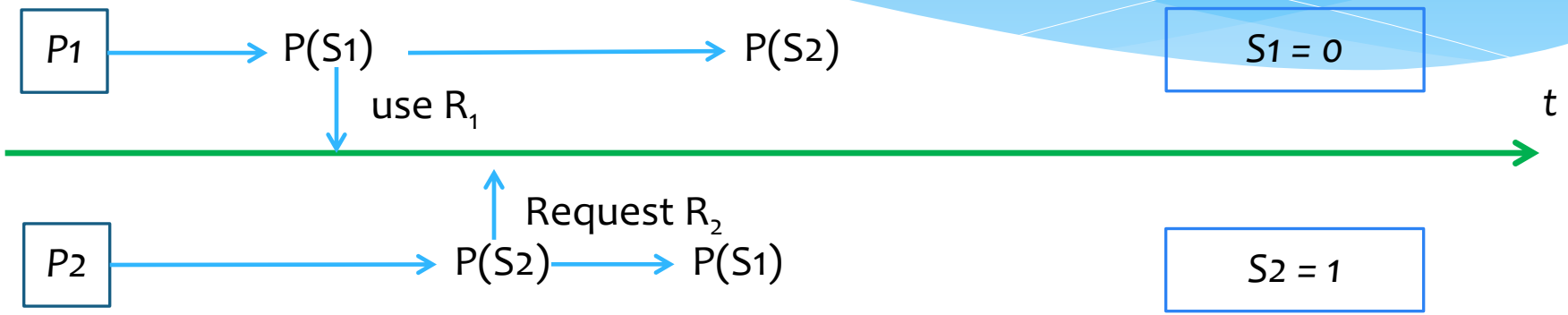
- Prove
  - Suppose deadlock happen between processes $\{P_1, P_2, \ldots P_m\}$
    - $R_1 \rightarrow P_1 \rightarrow R_2 \rightarrow P_2 \Rightarrow f(R_1) < f(R_2)$
    - $R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \Rightarrow f(R_2) < f(R_3) \ldots$
    - $R_m \rightarrow P_m \rightarrow R_1 \rightarrow P_1 \Rightarrow f(R_m) < f(R_1)$
  - $f(R_1) < f(R_2) < \ldots < f(R_m) < f(R_1) \Rightarrow$invalid

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- **Deadlock avoidance**
- Deadlock detection and recovery

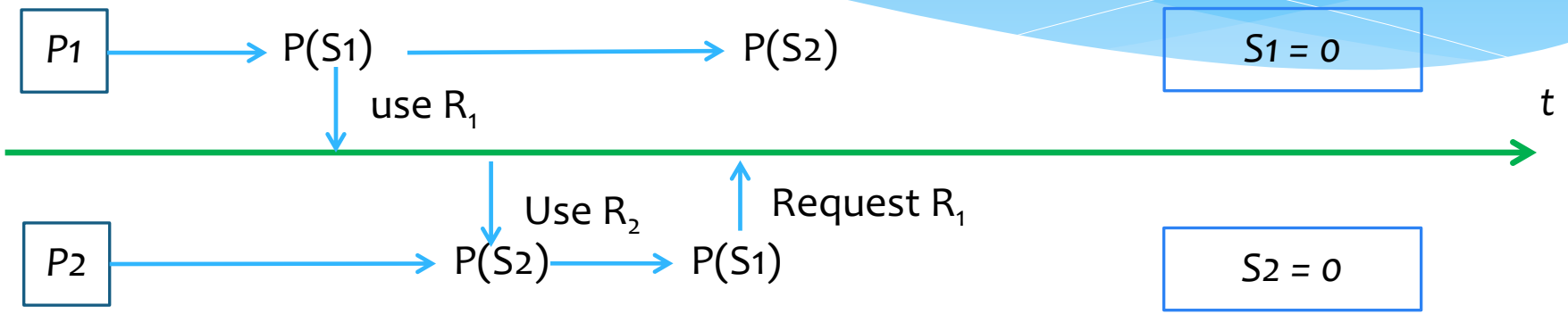
Example



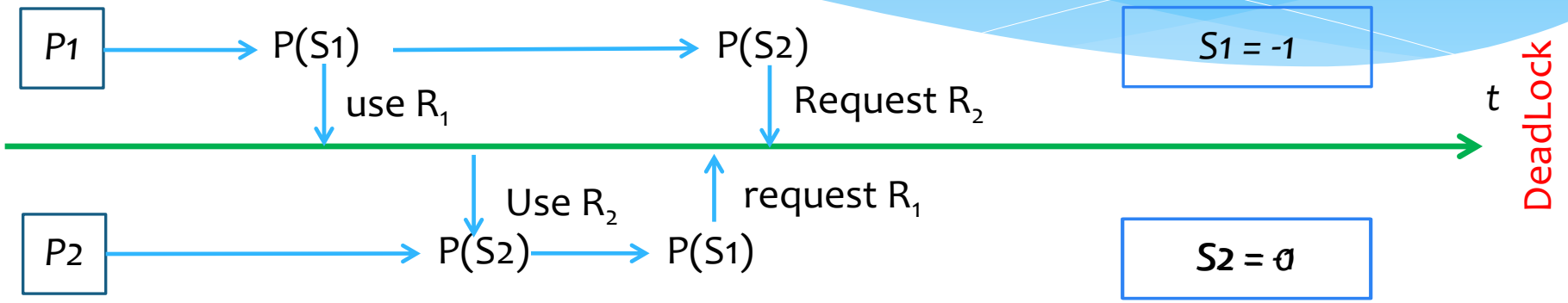
Example



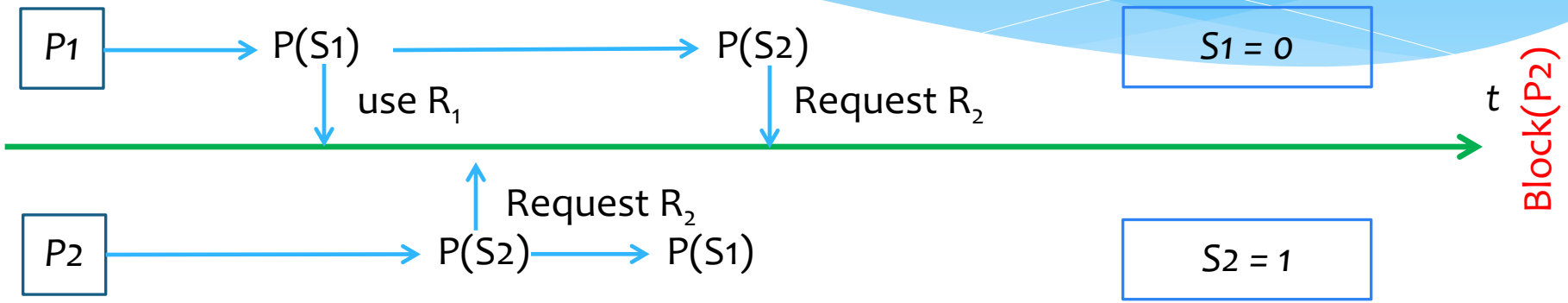
Example



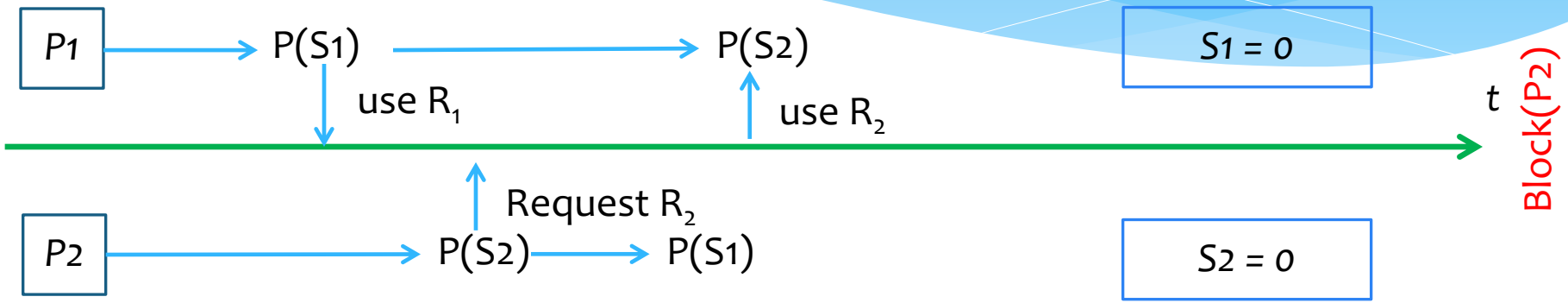
Example


P1
P(S1)
P(S2)
S1 = -1
use R1
Request R2
t
DeadLock
Use R2
request R1
P2
P(S2)
P(S1)
S2 = 0

Example


P1
P(S1)
P(S2)
S1 = 0
use R1
Request R2
t
Block(P2)
Request R2
P2
P(S2)
P(S1)
S2 = 1

Example



Conclude:

If processes' orders of request/release resources are known in advance, the system could make a resource allocation decision (accept or block) for all request to let deadlock not occur.

Rule

- Must know in advance information of processes and resources
  - Process has to declare the maximum amount of each resources type that is required for execution
- Decision is made based on the result of Resource-Allocation State check
  - Resource allocation state is defined by following parameters
    - Number of resource unit available in the system
    - Number of resource unit allocated for each process
    - Number of maximum resource unit each process may request
  - If system is safe, request is accepted
- Perform checking every time a resource request is received
  - Objective: Guarantee the system's always in safe state
    - At the beginning (resource is not allocated), system is safe
    - Only allocate resource when safety is guaranteed

  $\Rightarrow$ System changes from current safety state to another safety state

Safe state

The system's state is safe when

- It's possible to provide resource for each process (to maximum requirement) follow an order such that deadlock will not occur
- A safe sequence of all processes is existed

Safe sequence

Set of process P={P1, P2, . . . , Pn} is safe if

- For each process Pi, each resource request in future is accepted due to
  - The amount of available resource in the system
  - Resource is currently occupied by process Pj(j < i)

In safe sequence, when Pi request for resource

- If not accept immediately, Pi wait until Pj terminates (j < i)
- When Pj terminate and release resource, Pi receives required resource, executives, releases allocated resource and terminate
- When Pj terminate and release resource ⇒ Pi+1 will receive resource and able to finish. . .
- All the processes in the safe sequence is able to finish

Example

Consider a system includes

- 3 processes P_1, P_2, P_3 and one resource R has 12 units
- (P_1, P_2, P_3) may request maximum (10, 4, 9) unit of resource R
- At time t_0, (P_1, P_2, P_3) allocated (5, 2, 2) unit of resource R

- At current time (t_0), is the system safe?
  - System allocate (5 + 2 + 2) units => 3 units remain
  - (P_1, P_2, P_3) may request (5, 2, 7) units (*)
  - With current 3 units, all request of P_2 is acceptable ⇒ P_2 guaranteed to finish and will release 2 unit of R after finished
  - With 3 + 2 units, P_1 guaranteed to finish and will release 5 units
  - With 3 + 2 + 5 unit P_3 guaranteed to finish
- At time t_0, P_1, P_2, P_3 are guaranteed to finish ⇒ system is safe with safe sequence (P_2, P_1, P_3)

P_2 finished → release 4 (+1 remained after *) (= 1+4)
(= 3+2)

Example

Consider a system includes

- 3 processes P_1, P_2, P_3 and one resource R has 12 units
- (P_1, P_2, P_3) may request maximum (10, 4, 9) unit of resource R
- At time t_0, (P_1, P_2, P_3) allocated (5, 2, 2) unit of resource R

- A time t_1, P_3 request and allocated 1 resource unit. Is the system safe?
  - With current 2 units, all request of P2 is acceptable $\Rightarrow$ P2 guaranteed to finish and will release 2 unit of R after finished
  - When P_2 finished, the amount of available resource in the system is 4
  - With 4 resource unit, P_1 and P_3 both have to wait when apply for 5 more resource unit
  - Hence, system is not safe with (P_1, P_3)
- Remark: At time t_1, if P_3 has to wait when request for 1 more resource unit, deadlock will be removed

Example

- Conclude
  - System is safe ⇒ Processes are able to finish
    ⇒ no deadlock
  - System is not safe ⇒ Deadlock may occur

- Method
  - Verify all resource request
    - If the system is still safe after allocate resource ⇒ allocate
    - If not ⇒ process has to wait
  - The banker algorithm

Resource allocation graph

- Use when each type of resource has only 1 unit
- If there is a cycle, there will be a deadlock

- Add to the graph a new arc type: demand arc Pi → Rj
  - In the same direction as the request arc, shown in the graph -->
  - Indicates that Pi can request Rj in the future
- When joining the system, process must add all corresponding demand arcs to the graph
  - When Pi requests Rj, demand arc Pi → Rj transforms into the request arc Pi → Rj
  - When Pi releases Rj, using arc Rj → Pi changes to the demand arc Pi → Rj

* 3 types of arc

① $P_1 \leftarrow R_1$

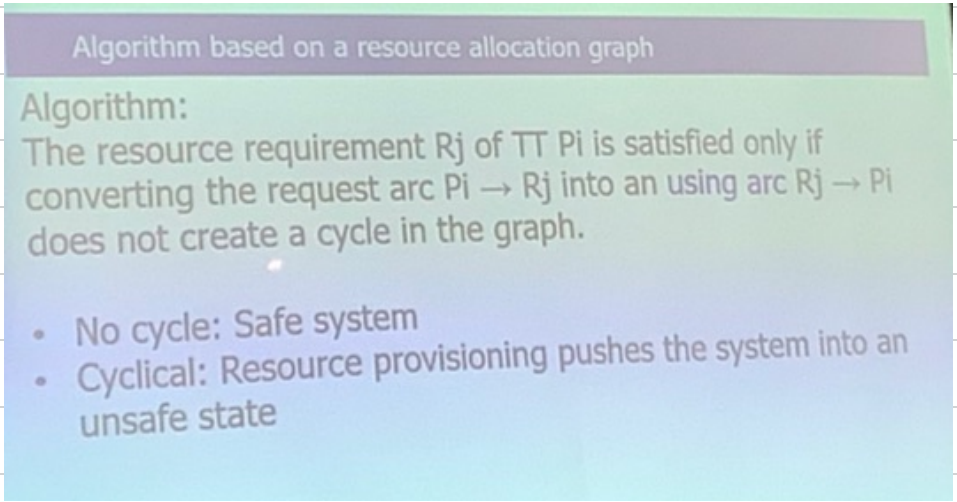
Process P_1 is served resource R_1

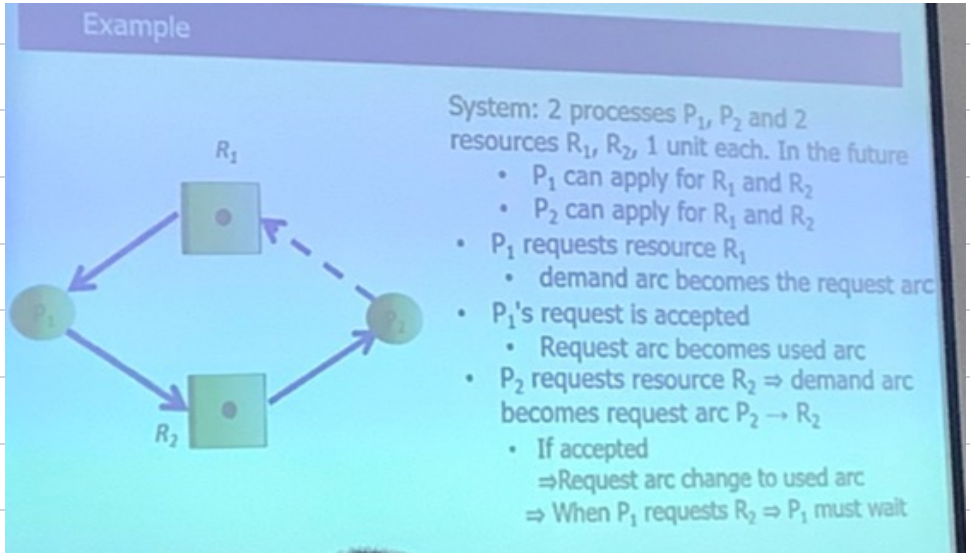
② $P_2 \rightarrow R_2$

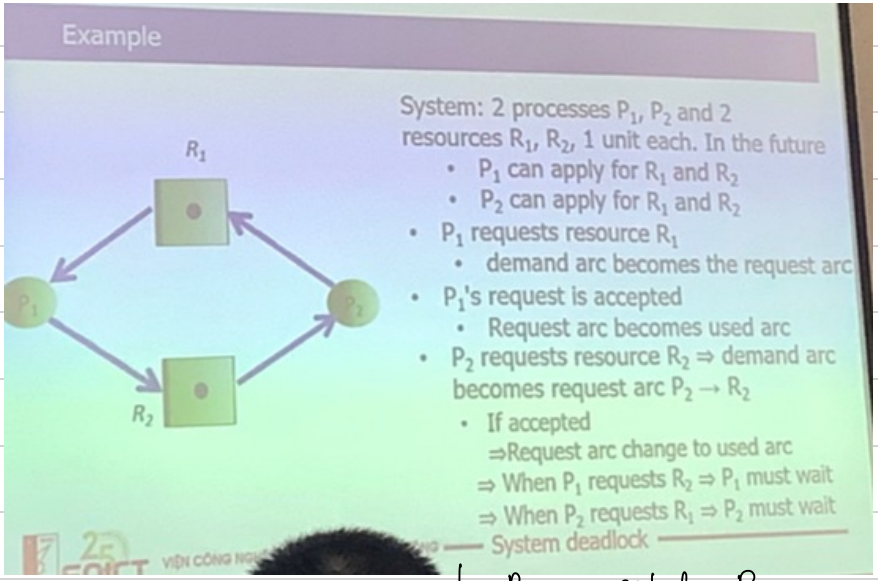
Process P_2 is requesting resource R_2

③ $P_2 \dashrightarrow R_3$

Demand arc (P_2 may need R_3 in the future)

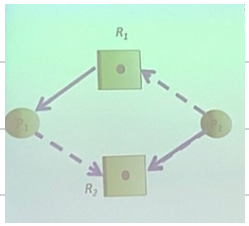




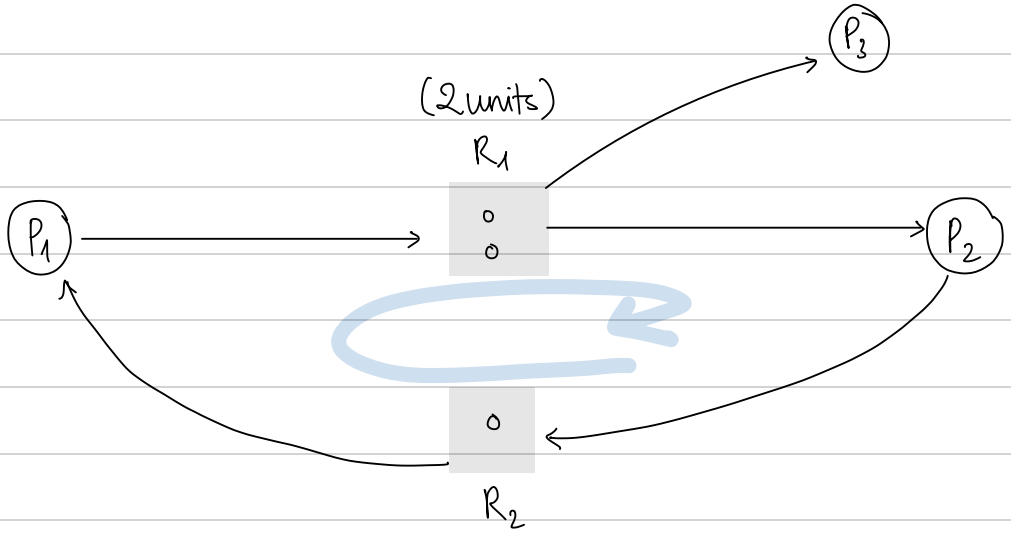


↳ P_2 request for R_2 is not accepted

↓



* How to detect cycle(s) in graph?
  Use either BFS or DFS



Though a cycle presents, there shall be NO deadlock

- P_3 finished ⇒ release R_1 ; R_1 alloc. to P_1
- R_1 served P_1 and P_2
- P_1 served by both R_1 and R_2 ⇒ done → release R_2
- P_2 now ——————— ⇒ completed

If all resources in the system only have 1 unit : ∃ cycle ⇒ ∃ deadlock!

——————— > 1 unit : ∃ cycle → deadlock may not happen

If ∄ cycle ⇒ ∄ deadlock

The banker algorithm: Introduction

- Good for systems have resources with many units
- New appearing process has to declare the maximum unit of each resource type
  - Not larger than the total amount of the system
- When one process request for resource, system verify if it's safe to accept this requirement
  - If system still safe ⇒ Allocate resource for process
  - Not safe ⇒ wait

Algorithm's data I

System

n number of process in the system

m number of resource type in the system

Data structures

Available Vector with length m represents the number of available resource in the system. (Available[3] = 8 ⇒?)

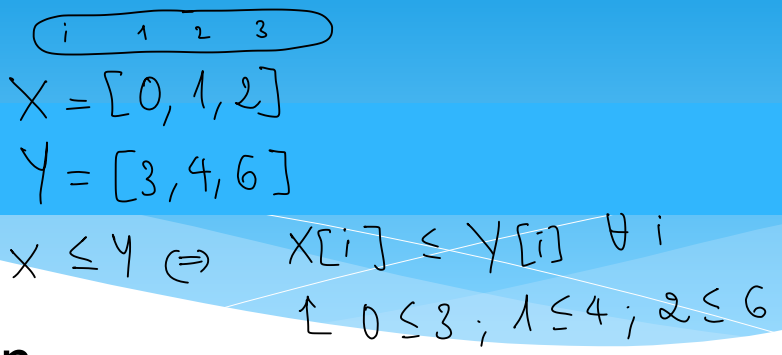
Max Matrix n ∗m represents each process maximums request for each type of resource. (Max[2,3] = 5 ⇒?) Process 2 requests max 5 units of resource type3

Allocation Matrix n ∗m represents amount of resource allocated for processes (Allocation[2,3] = 2 ⇒?) 2 units of resrc type 3 already allocated to P2

Need Matrix n ∗m represents amount of resource is needed for each process Need[2,3] = 3 ⇒?)

Need[i][j] = Max[i][j] - Allocation[i][j]

Algorithm 's data I



Rule

- X, Y are vectors with length n
  - $X \leq Y \Leftrightarrow X[i] \leq Y[i] \ \forall i = 1, 2, \ldots, n$
- Each row of *Max,Need,Allocation* is processed like vector
- Algorithm calculated on vectors

- Local structures

  Work vector with length m represents how much each resource still available

  Finish vector with Boolean type, length m represents if a process is guaranteed to finish or not

Algorithm for safety checking

```
BOOL Safe(Current Resource-Allocation State){
    Work←Available
    for (i : 1 → n) Finish[i]←false
    flag← true
    While(flag){
        flag←false
        for (i : 1 → n)
            if(Finish[i]=false AND Need[i] ≤Work){
                Finish[i]← true
                Work ← Work+Allocation[i]
                flag← true
            }//endif
    }//endwhile
    for (i : 1 → n) if (Finish[i]=false)return false
    return true;
}//End function
```

resrc đang có

stop when: all process done / not enough resource to alloc.

Can resrc be allocated to any process to finish?

If exist, and any process not done → Deadlock must have occur

Example

- Consider system with 5 processes P_0, P_1, P_2, P_3, P_4 and 3 resources R_0, R_1, R_2
  - R_0 has 10 units, R_1 has 5 units, R_2 has 7 units
- Maximum resource requirement and allocated resource for each process

	R0	R1	R2
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Max

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Allocation

- Is the system safe?
- P_1 requests 1 unit of R_0 and 2 unit of R_2?
- P_4 requests 3 unit of R_0 and 3 unit of R_1?
- P_0 requests 2 unit of R_1. allocate?

Example: Check for safety

- Number of available resource in the system *Available*(R_0, R_1, R_2) =(3, 3, 2)
- Remaining request of each process (*Need = Max - Allocation*)

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3
Max			

	R0	**R1**	**R2**
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	**R1**	**R2**
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Example: Check for safety

- Number of available resource in the system *Available*(R_0, R_1, R_2) =(3, 3, 2)

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3
Max			

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	T	F	F
Work	(5,3,2)				

check WORK vs. NEED
↓
if satisfies : T
else : F

update work :
WORK += alloc
(khi process đó xong)
initially = available

Example: Check for safety

- Number of available resource in the system Available(R0, R1, R2) =(3, 3, 2)

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3
Max			

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	T	T	T
Work	(7 4 3)				

System is safe
(P1, P3, P4, P0, P2)

Algorithm for resource request

- Request[i] Resource requesting vector of process *Pi*
  - Request[3,2] = 2: P_3 requests 2 units of resource R_2
- When Pi make a resource request, the system checks

① if(Request[i]>Need[i])
  Error(Request higher than declared number)

② if(Request[i]>Available)
  Block(Not enough resource, process has to wait)

③ Set the new resource allocation for the system
  - Available = Available - Request[i]
  - Allocation[i] = Allocation[i] + Request[i]
  - Need[i] = Need[i] - Request[i]

④ Allocate resource based on the result of new system safety check
  if(Safe(New Resource Allocation State))
    Allocate resource for Pi as requested
  else
    Pi has to wait
    Recover former state (Available, Allocation,Need)

1, 2, 2 (need)
↓
Prev. (P_1) 1, 0, 2 (request)
↓
0, 2, 0 (need)

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	F	F	F	F
Work	(2,3,0)				

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	F	F	F	F
Work	(2,3,0)				

+ alloc

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	F	F	F
Work	(5,3,2)				

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	F	F	F
Work	(5,3,2)				

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	F	T	F
Work	(7,4,3)				

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	T	F	T	T
Work	(7,4,5)				

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	F	T	T
Work	(7,5,5)				

Example: P_1 request (1, 0, 2)

- Request[1]≤Available ((1, 0, 2) ≤ (3, 3, 2)) ⇒ It's possible to allocate
- If allocate resource: Available = (2 , 3, 0)

	R0	R1	R2
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1
Need			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	T	T	T
Work	(10,5,7)				

Request is accepted

Example: (continue)

- P_4 request 3 units of R_0 and 3 units of R_2
  - Request[4] = (3, 0, 3) ①
  - Available = (2, 3, 0)

  ⇒ Resource is not enough, P_4 has to wait

COMMON MISTAKE
Check ① before ②

- P_0 request 2 units of R_1
  - Request[0]≤Available ((0, 2, 0) ≤ (2, 3, 0)) ⇒ It's possible to allocate
  - If allocate: Available = (2 , 1, 0)
  - Perform Safe algorithm ②

  ⇒ All processes may not finish

  ⇒ if accepted, system may be unsafe

⇒ Resource is sufficient but do not allocate, P_0 has to wait

- Deadlock conception
- Conditions for resource deadlocks
- Solutions for deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

Introduction

- Rule
  - Do not apply deadlock prevention or avoidance method, allow deadlock to occur
  - Timely check if the system has deadlock or not if yes then find a solution
  - To function properly, system has to provide
    - Algorithm to check if the system is deadlock or not
    - Algorithm to recorver from deadlock
- Deadlock detection
  - Algorithm for showing the deadlock
- Deadlock recovery
  - Terminate process
  - Resource preemptive

Algorithm to point out deadlock: Introduction

- Apply for system that has resource types with many units
- Similar to banker algorithms
- Data structures
  - Available Vector with length m: Available resource in the system
  - Allocation Matrix n ∗ m: Resources allocated to processes
  - Request Matrix n ∗ m: Resources requested by processes
- Local structures
  - Work Vector with length m: available resource
  - Finish Vector with length n: process is able to finish or not
- Rule
  - ≤ relations between Vectors
  - Rows in matrix n ∗ m are processed similar to vectors

Algorithm to point out deadlock

```
BOOL Deadlock(Current Resource-Allocation State){
    Work←Available
    for (i : 1 → n)
    if(Allocation[i]≠0) Finish[i]←false
    else Finish[i]←true;              //Allocation = 0 not in waiting circular
   flag← true
   While(flag){
      flag←false
      for (i : 1 → n)
            if (Finish[i] = false AND Request[i] ≤ Work){
               Finish[i]← true
              Work ← Work+Allocation[i]
             flag← true
            }//endif
    }//endwhile
   for (i : 1 → n) if (Finish[i] = false) return true;
   return false;            //Finish[i] = false, process Pi is in deadlock
}//End function
```

Example

- 5 processes P_0, P_1, P_2, P_3, P_4; 3 resources R_0, R_1, R_2
  - Resource R_0 has 7 units, R_1 has 2 units, R_2 has 6 units
- Resource allocation status at time t_0

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Example

① P_0 finishes ⇒ release (0,1,0)
work = (0,1,0)

② P_1 → skip

③ P_2 finishes ⇒ release (3,0,3)
⇒ work = (3,1,3)

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

④ P_3 → alloc (1,0,0)
→ work = (2,1,3)

↓

P_2 done.
→ work = (5,3,4) (+alloc)

⑤ P_4 → skip

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	F	F	F	F
Work	(0,0,0)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	F	F	F
Work	(0,1,0)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	T	F	F
Work	(3,1,3)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	T	T	F
Work	(5,2,4)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	T	T	F
Work	(5,2,4)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	T	T	F
Work	(7,2,4)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	T	T	T	T
Work	(7,2,6)				

System has no deadlock
(P_0, P_2, P_3, P_1, P_4)

Example

- At time t_1: P_2 request 1 more resource unit of R_2
- Resource allocation status

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	6	0	2
Request			

- Available resource (R_0, R_1, R_2) =(0, 0, 0)

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	6	0	2
Request			

Process	P_0	P_1	P_2	P_3	P_4
Finish	F	F	F	F	F
Work	(0,0,0)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	6	0	2
Request			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	F	F	F
Work	(0,1,0)				

Example

	R0	R1	R2
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Allocation			

	R0	R1	R2
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	6	0	2
Request			

Process	P_0	P_1	P_2	P_3	P_4
Finish	T	F	F	F	F
Work	(0,1,0)				

P_0 may finish but the system is deadlock.
Processes are waiting for each other(P_1, P_2, P_3, P_4)

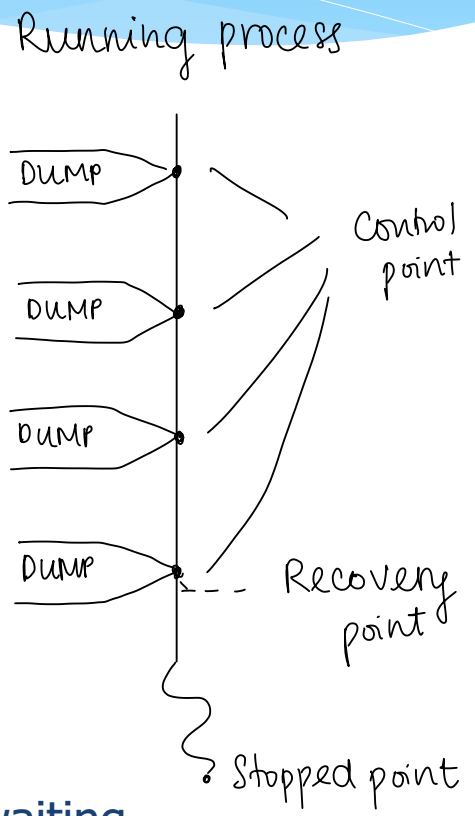
Deadlock recorvery: Process termination method

Rule: Terminate processes in deadlock and take back allocated resource

- Terminate all processes
  - Quick to eliminate deadlock
  - Too expensive
    - Killed processes may be almost finished
- Terminate processes consequently until deadlock is removed
  - After process is terminated, check if deadlock is still exist or not
    - Deadlock checking algorithm complexity is $m * n^2$
  - Need to point out the order of process to be terminated
    - Process's priority
    - Process's turn around time, how long until process finish
    - Resources that process is holding, need to finish
    - . . .
- Process termination's problem
  - Process is updating file $\Rightarrow$ File is not complete
  - Process is using printer $\Rightarrow$ Reset printer's status

Deadlock recorvery: Resource preemption method

Preempt continuously several resources from a deadlocked processes and give to other processes until deadlock is removed



Need to consider:

① Victim's selection
- Which resource and which process is selectedd?
- Preemptation 's order for smallest cost
- Amount of holding resource, usage time. . .

② Rollback
- Rollback to a safe state before and restart
- Require to store state of running process

③ Starvation
- One process is preempted many times ⇒ infinite waiting
- Solution: record the number of times that process is preempted

Another solution ?

Ostrich's reasoning :

I don't see you

=

You don't see me

↳ Modern OS uses this : Deadlock Ignorance