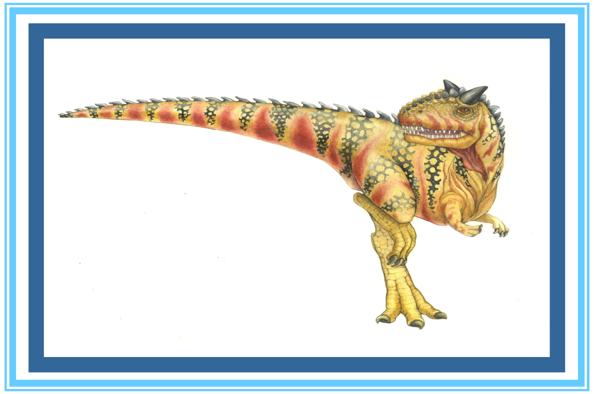
Chapter 3: Process Concept

School of Computing, Gachon Univ.
Jungchan Cho



Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing

Chapter 3: Process Concept

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

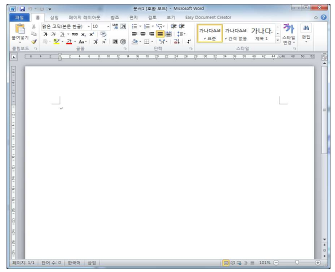
Process Concept

- **Process** (= **job/task, ≠program, ≠processor**)
  - a program *in execution*
    - Program is ***passive*** entity stored in disk (executable file), Process is ***active*** *loaded* into **main memory**
- One program can be several processes
  - e.g., same user executing multiple copies of a program
- These two terms are ***different***!!!!!!

  - **Process**: a program in execution

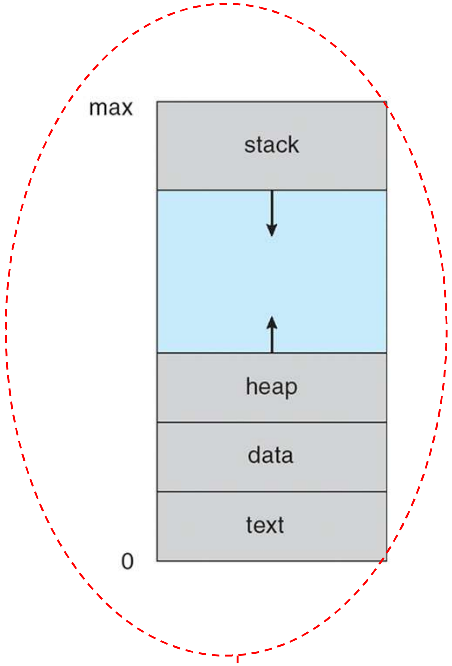
프로세스
  - **Processor**: processor chip (e.g., CPU)

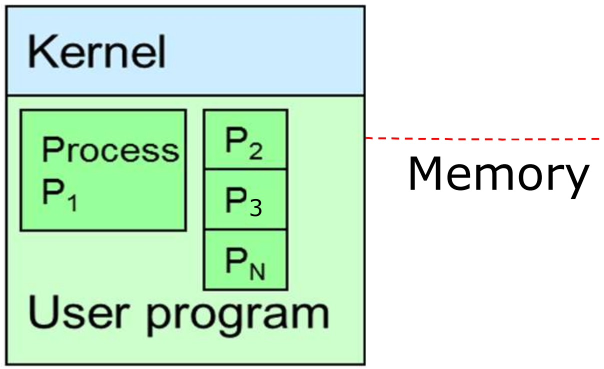
프로세서
  - Multiprocessor vs. Multiprocess(ing)



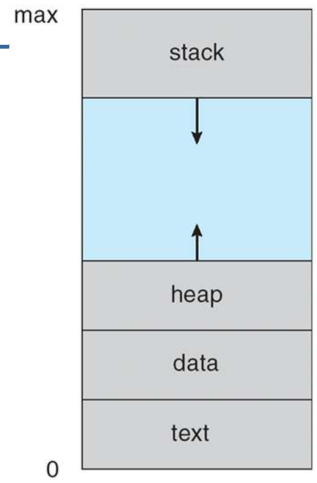
Process in Memory

- Process **in memory**
  - **Text**: The binary assembly **program code**
  - **Data**: global **variables**
  - **Stack**: temporary local **data**
    - Function parameters, return addresses, local variables
  - **Heap**: memory ***dynamically*** allocated **during run time** (e.g., C malloc(), Java objects)





Simple C-code example



Binary Assembly Code (text)

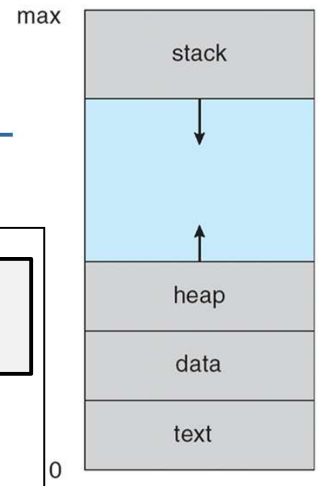
```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main()
{
        int *values;
        int i;

        values = (int *)malloc(sizeof(int)*5);
        for(i = 0; i < 5; i++)
            values[i] = I;

        return 0;
}
```

Global variables (data)

Local variables (stack)

C malloc (heap)

Simple C-code example



Binary Assembly Code (text)

```
#include <stdio.h>
FILE* fp;
int counter;

void main(int argc, char *argv[]) {
        char  c;
        counter=0;
        fp = fopen(argv[1], “w”);
        while(1) {
            c = char_read();
            if( c == ‘s’ ) break;
            counter++;
            printf(“%d th character is %c \n”, counter, c);
            fprintf(fp, “%c”, c);
        }
      fclose(fp);
}

char char_read()
{
      char  c;
      scanf(“%c”, &c);
      return c;
}
```

Global variables (data)

Return address (stack)

Program counter

Executing here

Local variables (stack)

Process in Memory: Stack

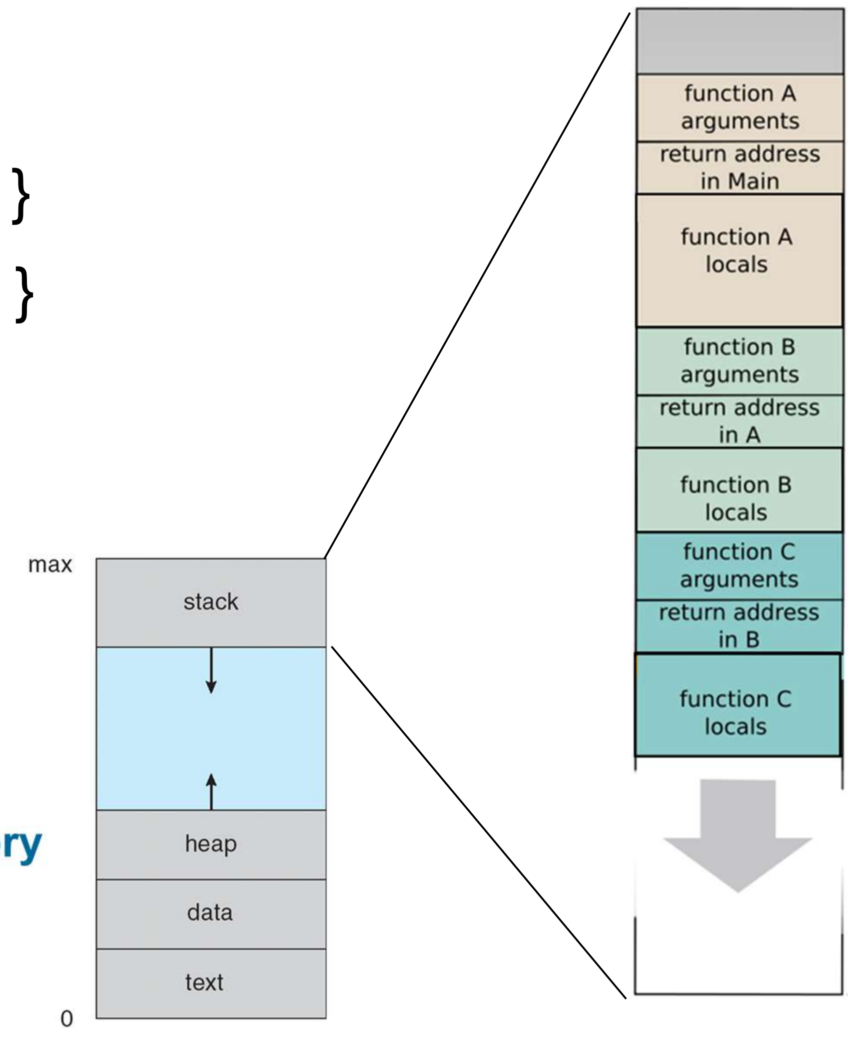
- Stack used for function calls

  main() { func_A()... }

  func_A() { func_B()... return }

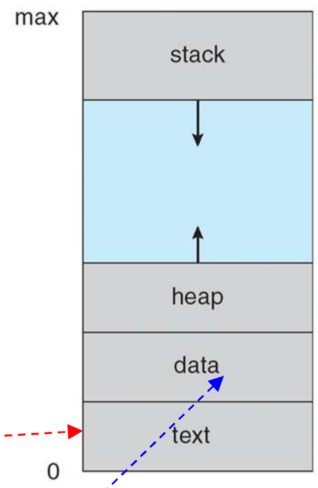
  func_B() { func_C()... return }

  func_C() { ... return }



Process in Memory

Process in Memory: Text & Data




```
        .text
main:   la $t0,str              # put starting address of "hello world" into t0
loop:   lb $t1,0($t0)           # load byte in phrase
        beqz $t1,over           # if character null, we are finished
        beq $t1,0x6c,cnt        # if the character is an l, go to count
incr:   addi $t0,$t0,1          # add 1 to current byte address
        j loop                  # get next byte to compare
cnt:    addi $t2,$t2,1          # add one to count of letter l's in phrase
        j incr                  # go back into loop
over:   la $a0,rept
        li $v0,4                # Ouput report phrase
        syscall
        move $a0,$t2            # move total l-count to $a0 for output
        li $v0,1
        syscall                 # output letter total
        li $v0,10
        syscall                 # end program
```

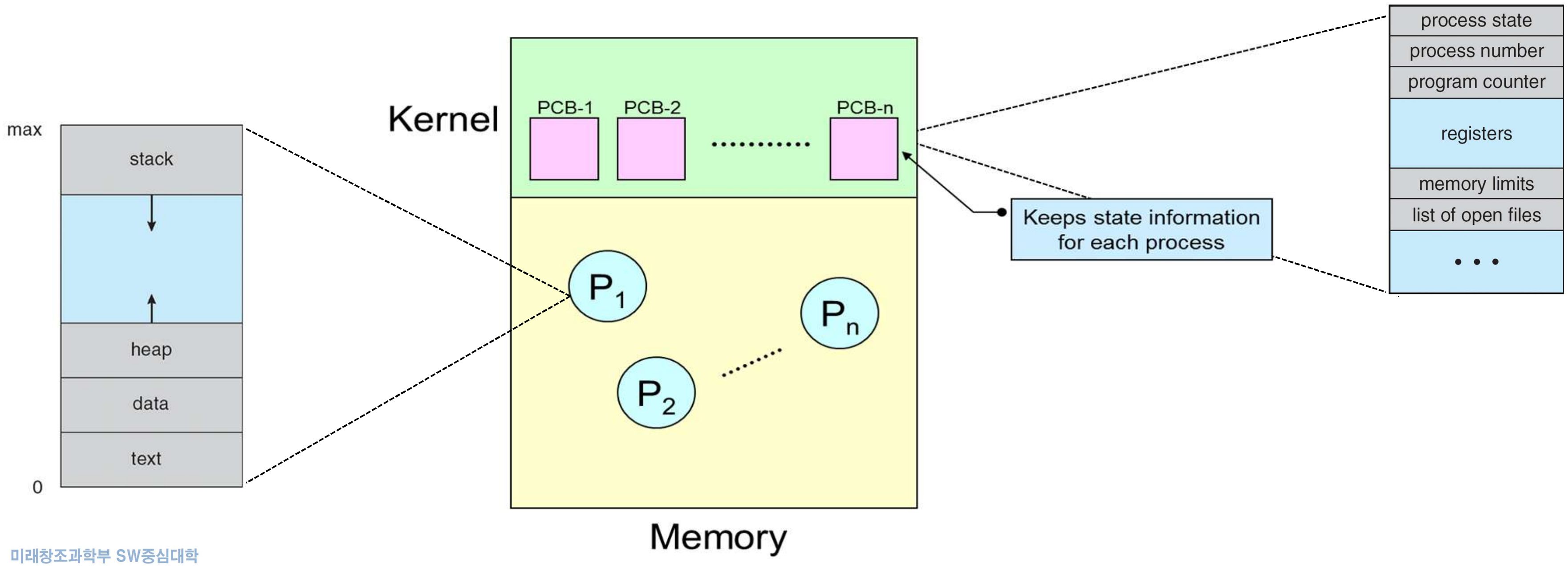
Binary Assembly Code (text)

```
        .data
str:    .asciiz "Hello, world!\n"
rept:   .asciiz "The total count of the letter l is "
```

Binary data (data)

Process Management

- Each process is registered to and managed by the OS
  - The OS manages and take care of all the processes.
  - Therefore, the OS needs to manage the current information (e.g., state) of each process – use a data structure called **PCB (Process Control Block)**



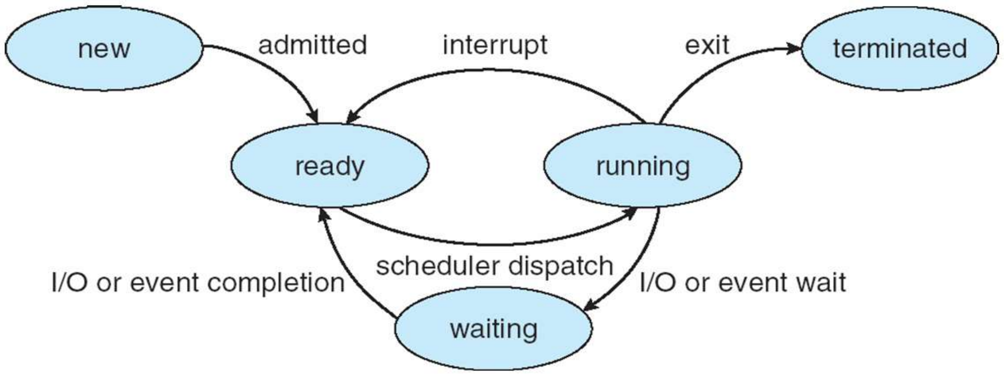
Process Control Block (PCB)

PCB: OS maintains the **information** for *each* process

- Process **state** – running, ready, waiting, etc (next slide)
- Process stops while running (e.g., interrupt) and now in ready state. Need to save current **running information** to resume running where we stopped...
  - Program counter (PC) – next instruction address,
    CPU registers – contents of all registers (in CPU)
  - CPU scheduling information
- Where is the process located in **memory**?
  - Memory-management information
    – memory allocated to the process
- Which **I/O devices/files** are the process using?
  - I/O status information
    – I/O devices allocated to process, list of open files

Process State

- As a process executes, it changes **state**
  - **new**: The process is *being* created
  - **ready**: The process (in memory) is ready to be assigned to CPU
  - **running**: Process instructions are being executed by CPU
  - **waiting**: The process is waiting for some event (e.g., I/O operation) to occur
  - **terminated**: The process has *finished* execution



Important!
- Only **one** process is **running** on any CPU core at any instant
- All the other processes are waiting in **ready** or **waiting** states

미래창조과학부 SW중심대학
가천대학교 소프트웨어학과
Gachon University Department of Software

Chapter 3: Process Concept

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

Questions

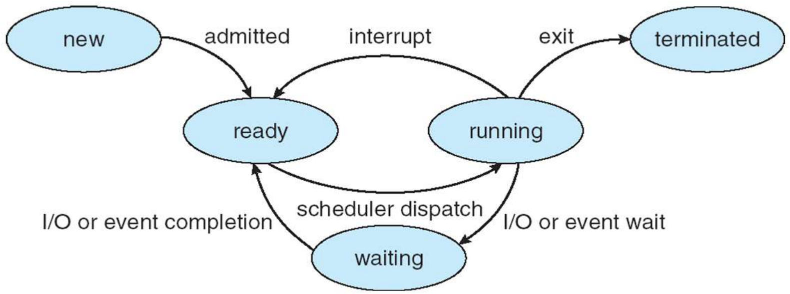
- How many processes can be executed ***concurrently*** in an **OS**?
  - hint: multi-tasking (programming, processing)
- How many processes can use a **CPU core** ***in parallel*** at the same time?
  - hint: CPU instruction execution

On a single CPU core…

- SUPPOSE: One process is currently using the CPU

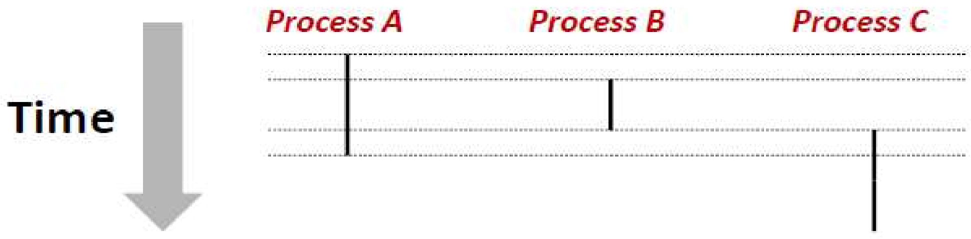
```
long count = 0;
while (count >= 0)
      count++;
```

- What are other processes doing?
  - Usually nothing! (Just waiting for CPU in ___ state or doing I/O in ____ state)

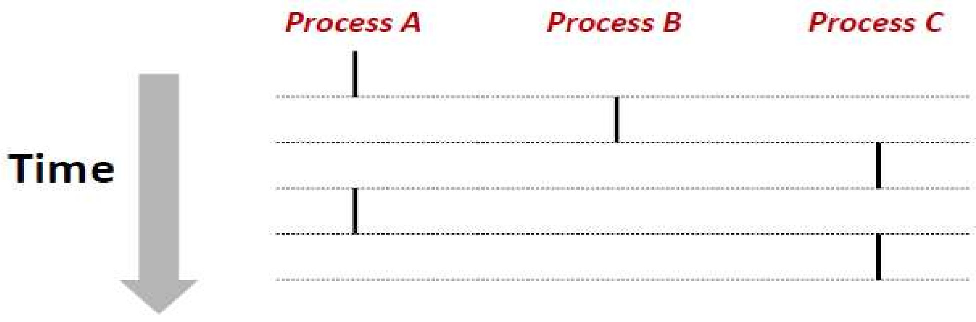


User View of Concurrent Processes

- *Intuitively*, we can think of **concurrent** processes as running **in parallel** with each other

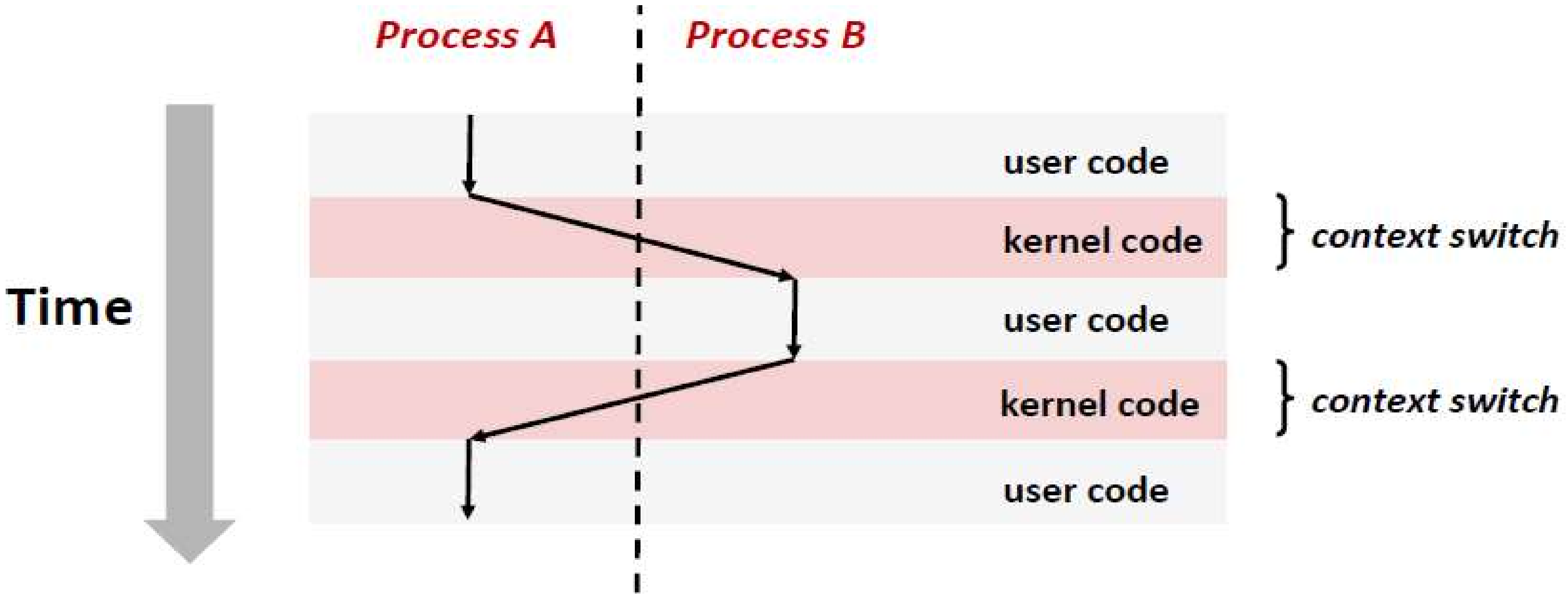


- However, in a single CPU core, only one process can run at a time



Concurrent Processes

- Process switching from one process to another by OS !
  - called **Context switch** (we will learn soon!)



- Determining when to switch, to which process is very important to the system performance – Enter **process scheduling**!

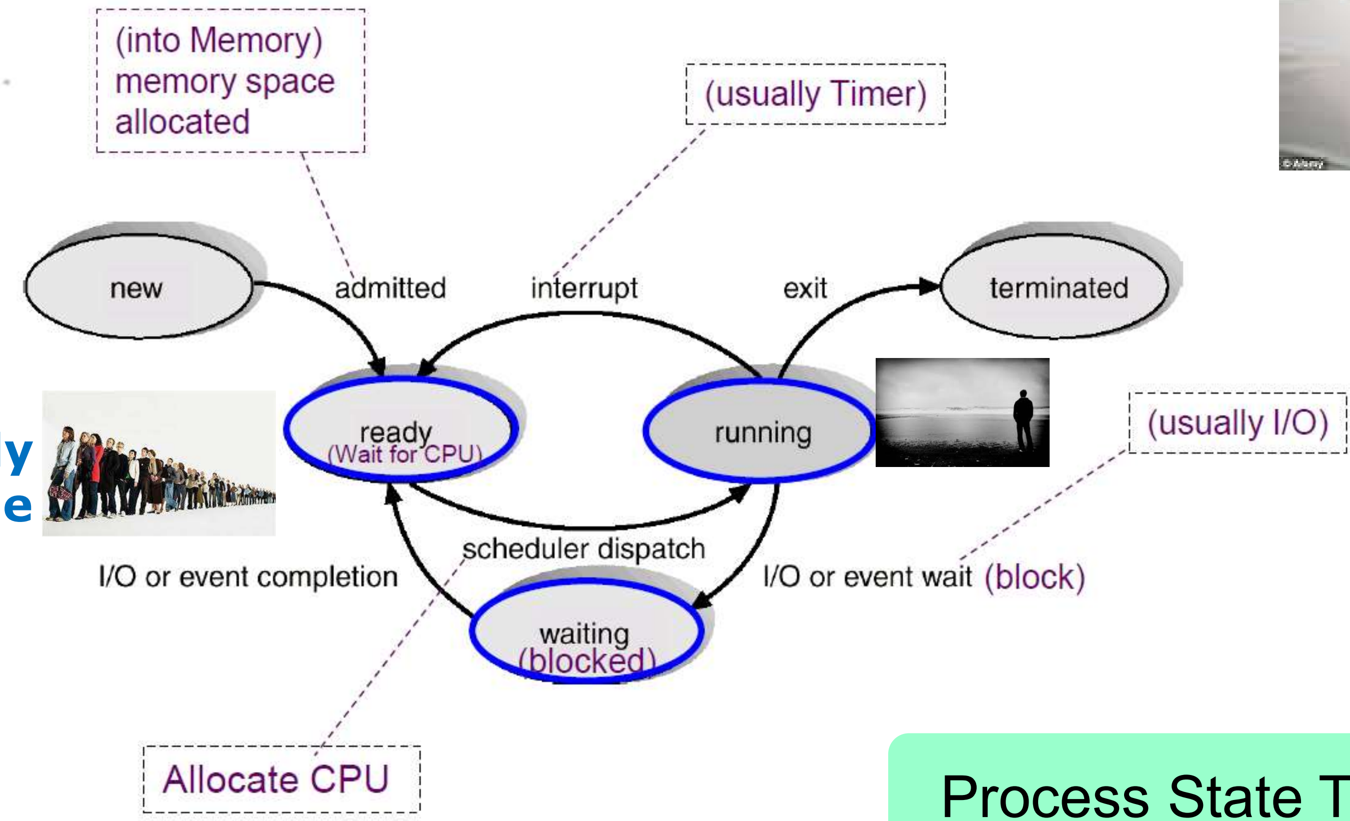
Process Scheduling

- Objective of **Multiprogramming**
  - Maximize **CPU utilization**: Switch the CPU to another process when I/O happens
  - **Time-sharing:** Switch the CPU among processes so frequently that users can interact with each program while it is running
- How do we implement multi-programming?
  - **Process scheduler** selects among available processes (in Ready state) for next execution on CPU

Representation of Process State

- **How to describe the state of a process ?**
- **A process's point of view**

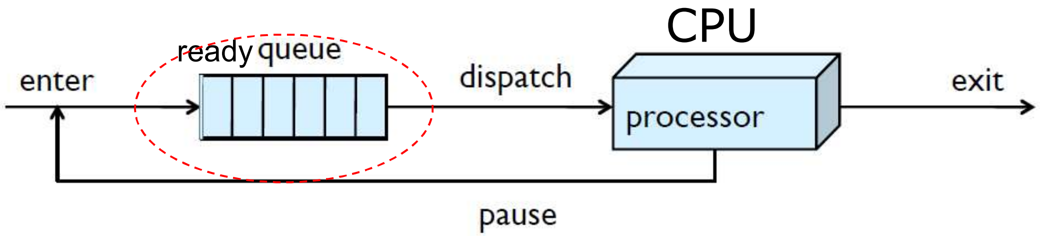
Ready queue



Process State Transition Diagram

Process Scheduling Queues

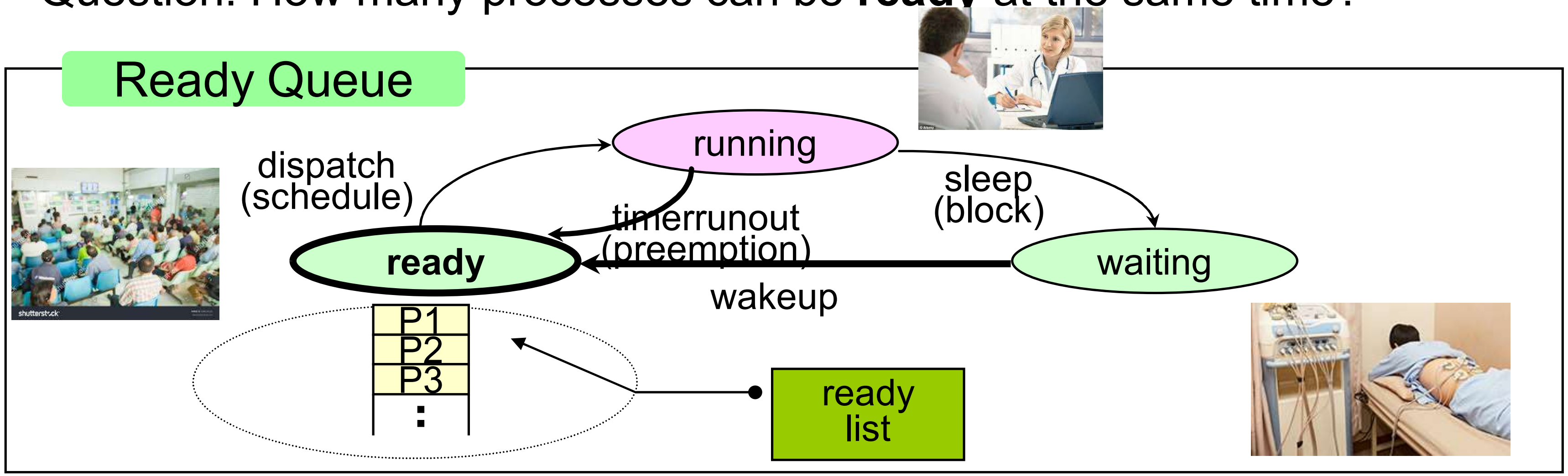
- **Ready queue**
  - The processes that are in the **main memory** and are ready to execute are kept on a list called **ready queue** (in Ready state; waiting for CPU)



- **I/O (Device) queues**
  - The list of processes waiting for a particular **I/O** operation is called a device queue (waiting for I/O device)
    - Each device has its own device queue

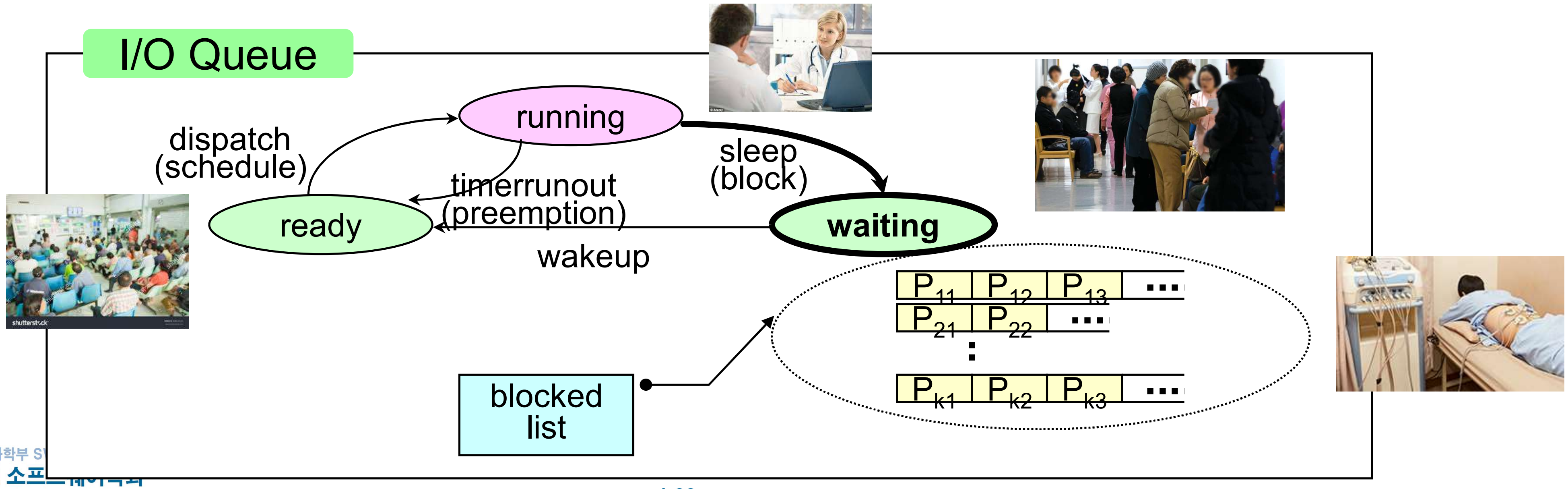
Scheduling Queues

- **Ready queue**
  - Processes in _____ state
    - Awaiting for the processor (i.e. CPU)
  - **Process scheduling**
    - **Selecting** a process from the **ready queue** and **dispatch** it when the **processor** is available
  - Question: How many processes can be **running** at the same time?
  - Question: How many processes can be **ready** at the same time?



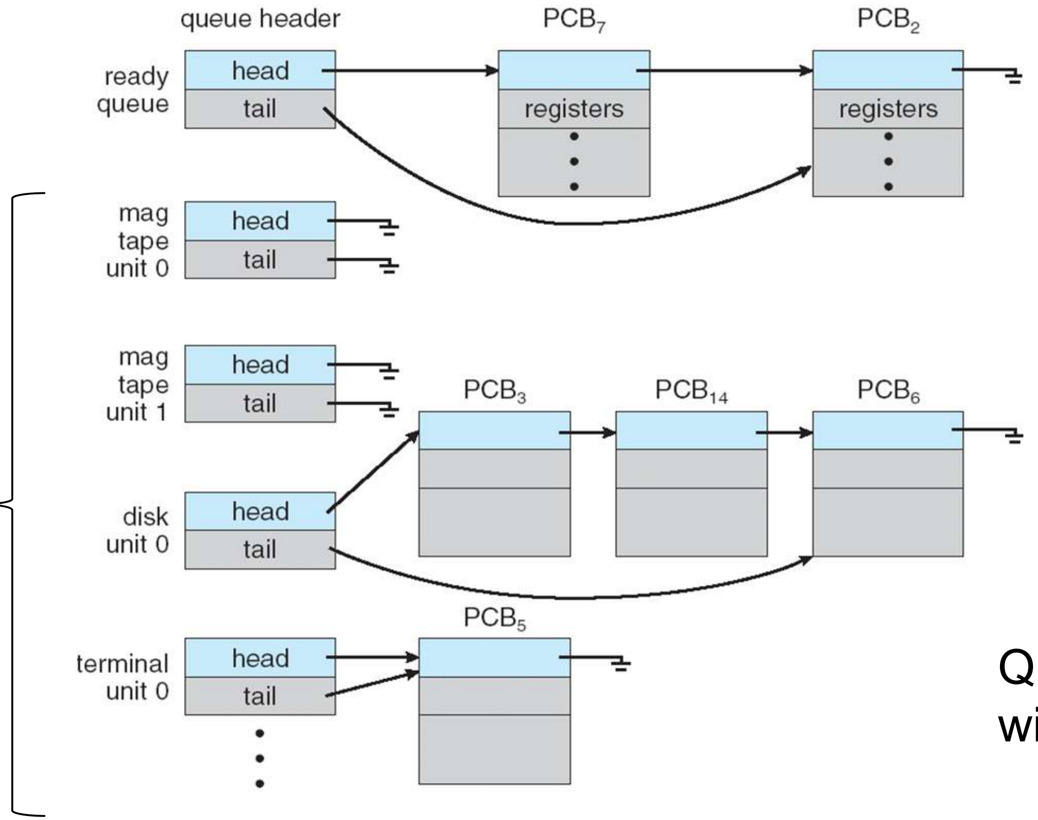
Scheduling Queues

- **I/O (Device) queue**
  - When a process is allocated the CPU, it **executes** for a while and eventually **quits**, is **interrupted**
  - or **waits for an event** (e.g., I/O completion)
    - The process has to be **waiting** (or **blocked**) in the **I/O queue**
    - Each device as a I/O (device) queue – if I/O device is busy, multiple processes may be waiting in I/O queue
    - Also called Blocked list (block queue, I/O queue)



Ready Queue And Various I/O Device Queues

I/O (device) queues



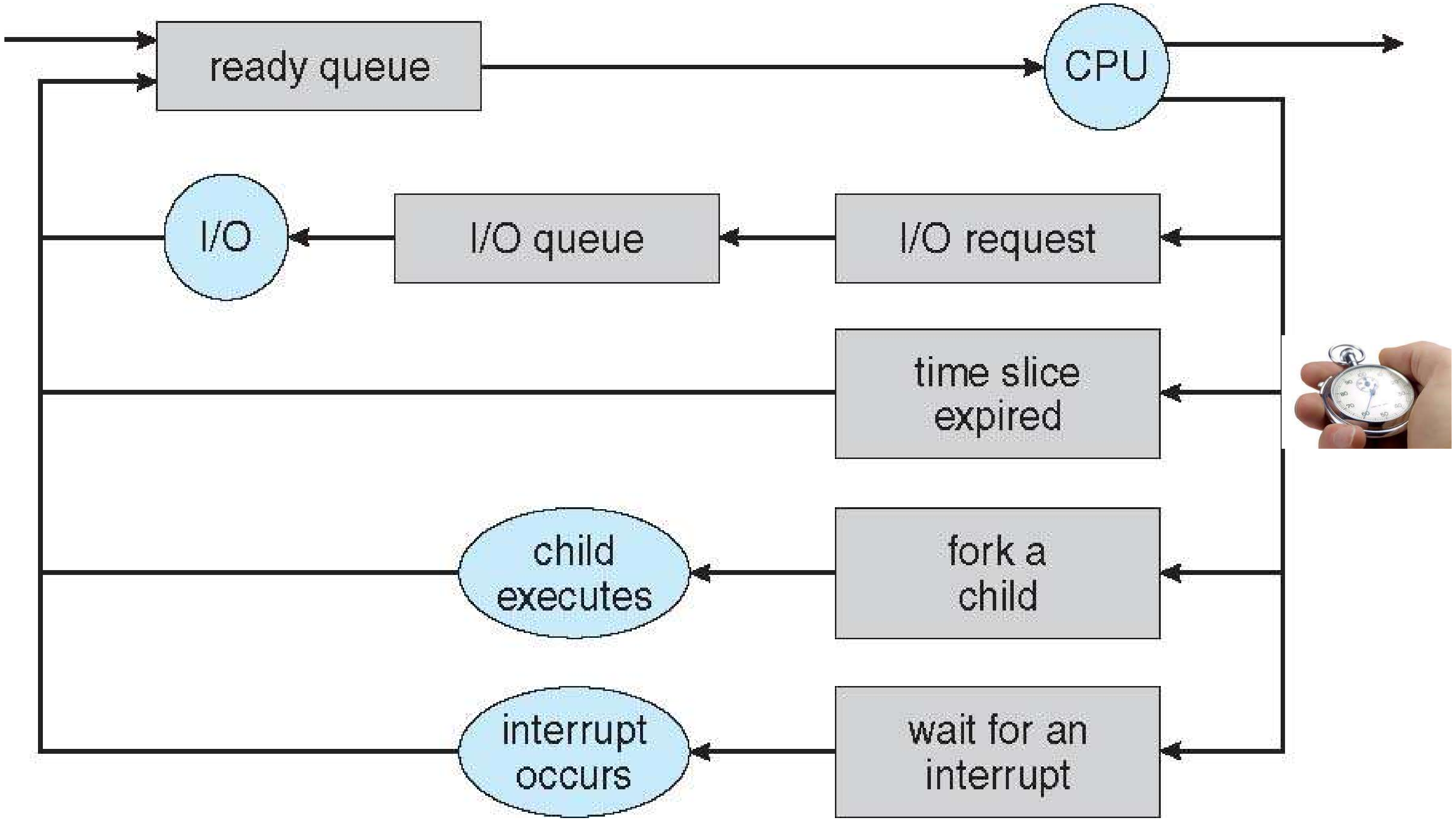
Queues are usually implemented with linked lists

Queue header → pointing the first and last PCB structures

Each PCB structure has the address of its next PCB structure

Representation of Process Scheduling

Queuing diagram



Process Scheduler

- Process Scheduler
  - an **operating system module** that selects the next jobs to be admitted into the system and the next process to run

- Types
  - Long-term scheduler (=job scheduler)
  - Short-term scheduler (= CPU scheduler)

Schedulers

- Long-term scheduler (or job scheduler)
  - Decides which jobs or processes (in disk) are to be admitted into the ready queue *(i.e.* Main memory)
  - Giving **Memory**
  - invoked infrequently (seconds, minutes) ⇒ (may be slow)

- Short-term scheduler (or **CPU scheduler**)
  - Among processes in ready queue, selects which process (memory resident) should be executed next and allocated CPU
  - Giving **CPU**
  - invoked very frequently (milliseconds) ⇒ (must be fast)

Long-term Schedulers

- The long-term scheduler controls the **degree of multiprogramming**
  - Degree of multi-programming : the number of processes in memory (determining the degree of contention for resources)
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Q: What happens if most processes in ready queue are I/O bound? CPU bound?
  - Long-term scheduler strives for good ***process mix***

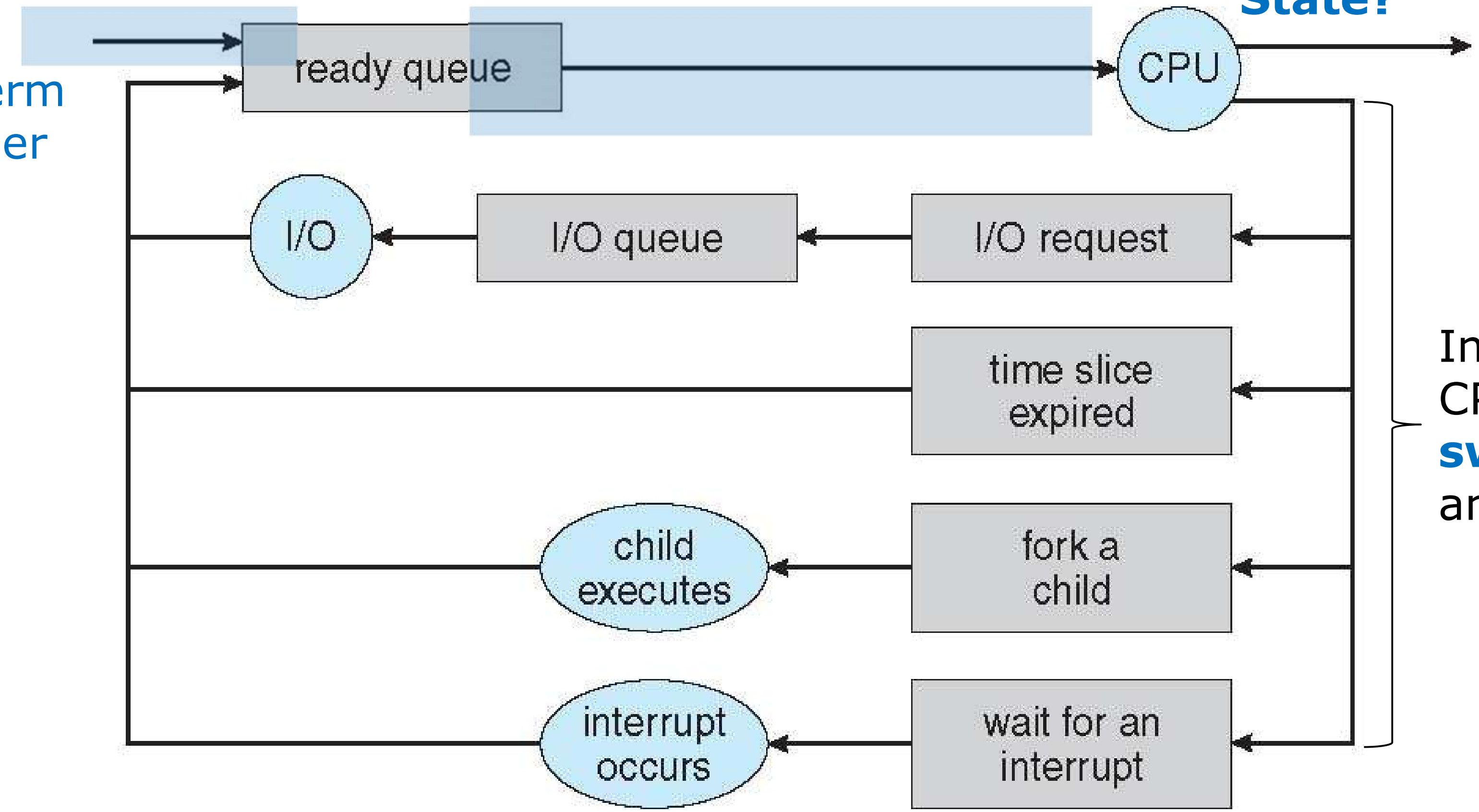
Representation of Process Scheduling

Queuing diagram

Short-term scheduler

Process State?

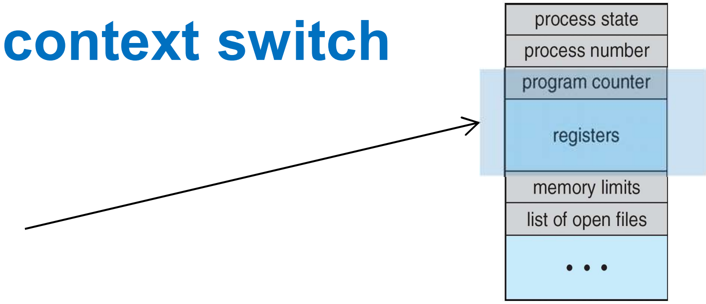
Long-term scheduler



In these cases, CPU **context switches** to another process

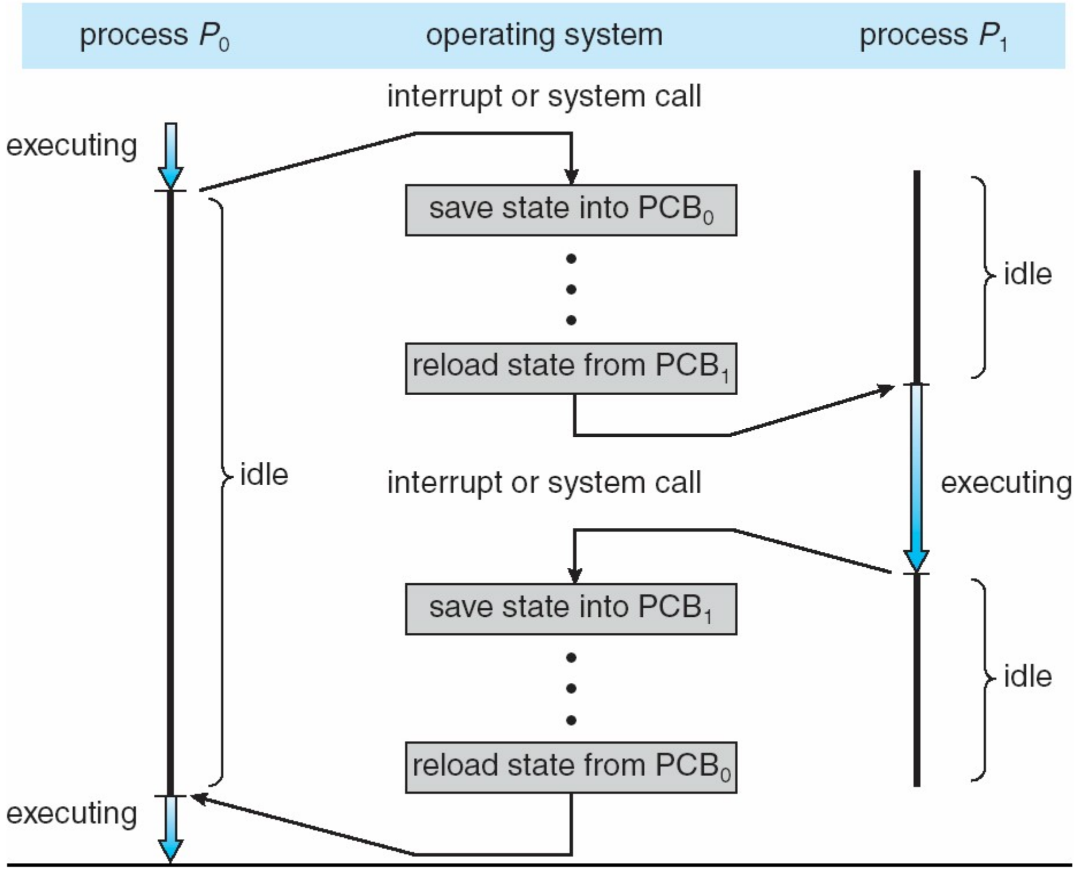
Context Switch

- When CPU switches to ***run*** another process
  - the system must **save the state** of the old process (to resume where we stopped) and
  - load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB

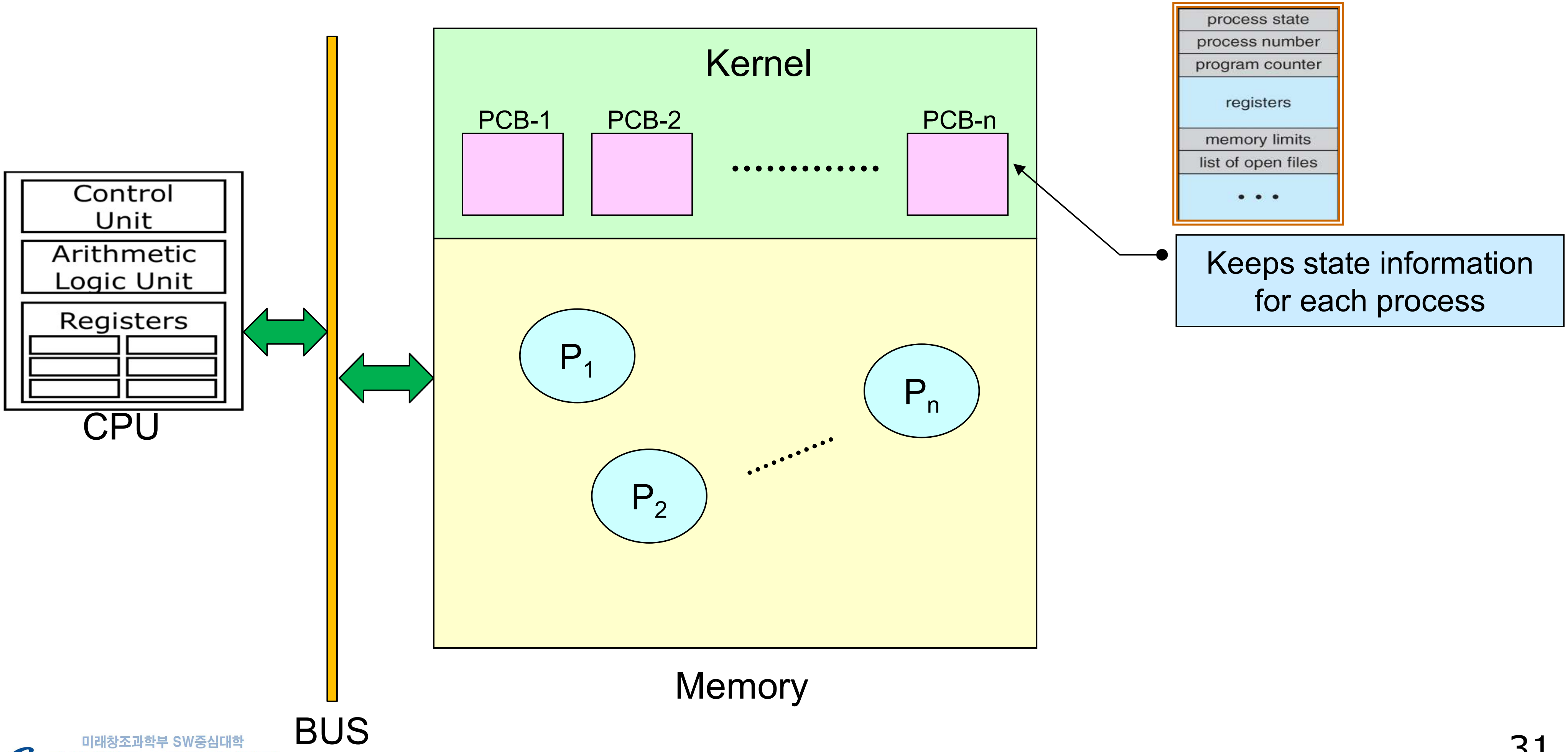


- Context-switch time is **overhead**; the system (or CPU) cannot do any other work while switching
  - The more complex the PCB -> longer the context switch

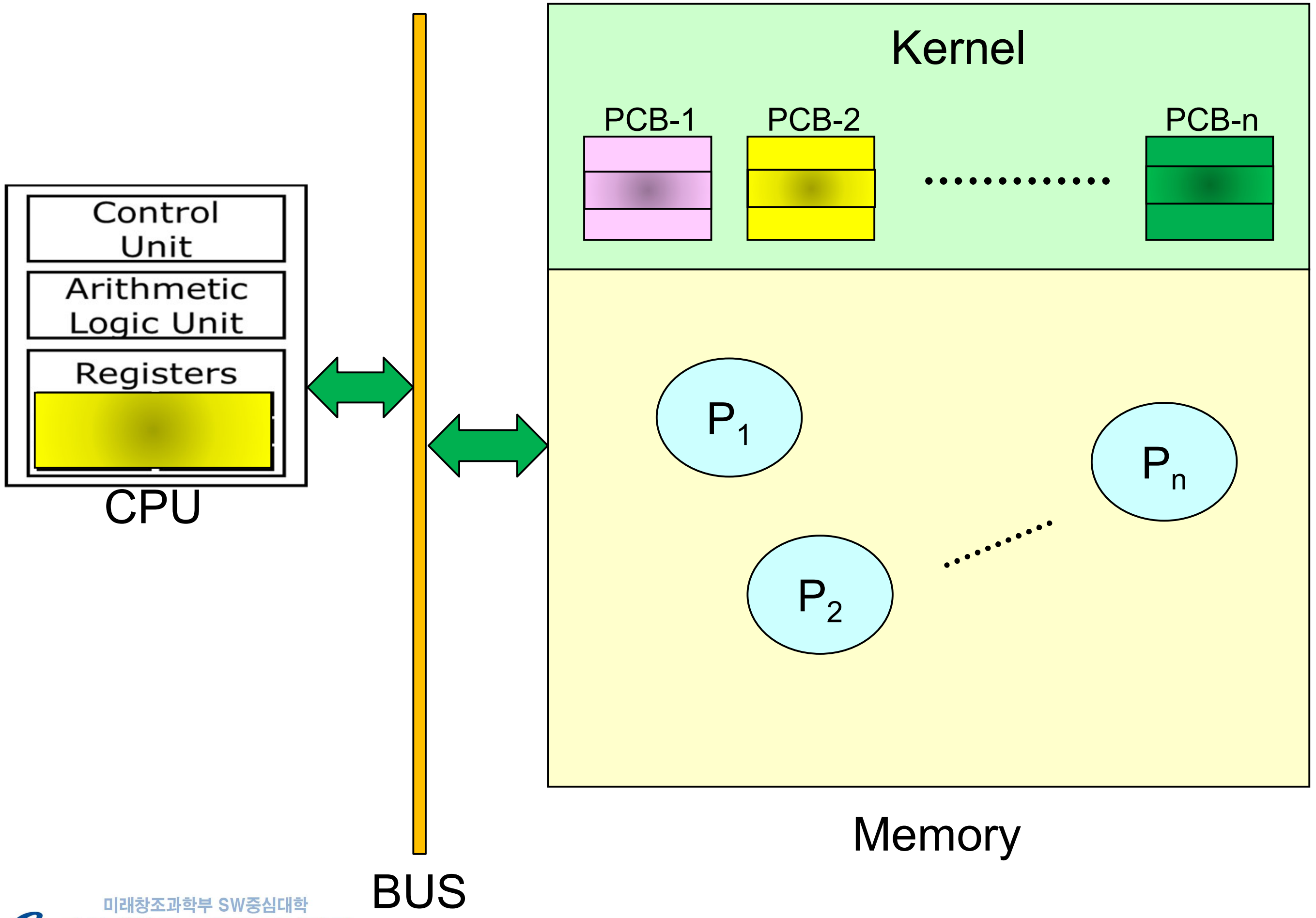
CPU Switch From Process to Process



(Cont.)

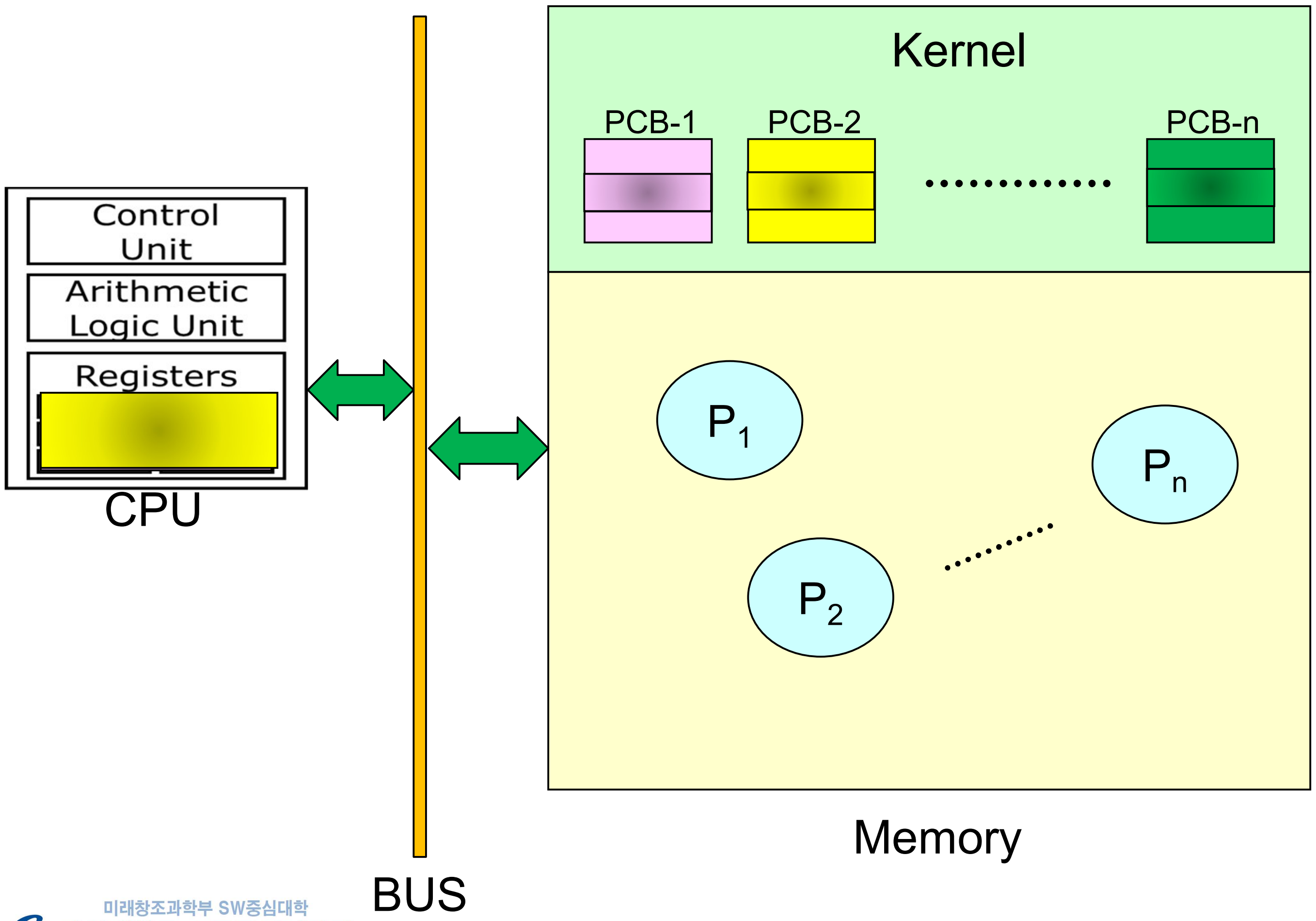

Control
Unit
Arithmetic
Logic Unit
Registers
CPU
BUS
Kernel
PCB-1
PCB-2
PCB-n
P1
P2
Pn
Memory
process state
process number
program counter
registers
memory limits
list of open files
Keeps state information for each process

(Cont.)

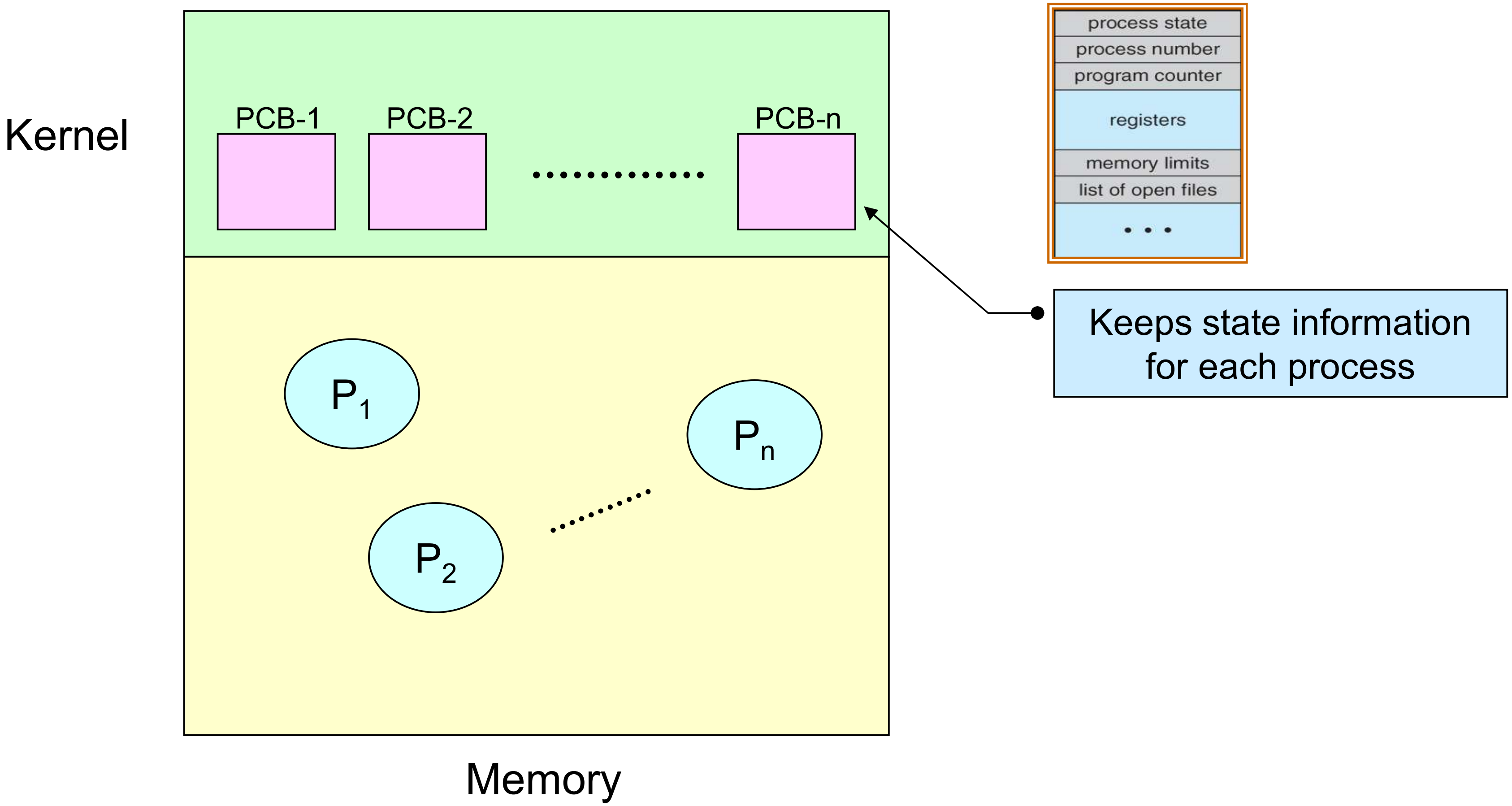


Kernel
PCB-1
PCB-2
PCB-n
Control Unit
Arithmetic Logic Unit
Registers
CPU
P_1
P_2
P_n
Memory
BUS

(Cont.)


Kernel
PCB-1
PCB-2
PCB-n
Control Unit
Arithmetic Logic Unit
Registers
CPU
P1
P2
Pn
Memory
BUS

(Cont.)

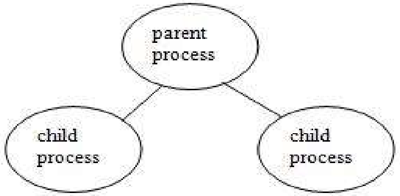

Kernel
PCB-1
PCB-2
PCB-n
P_1
P_2
P_n
process state
process number
program counter
registers
memory limits
list of open files
• • •
Keeps state information for each process
Memory

Chapter 3: Operations on Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

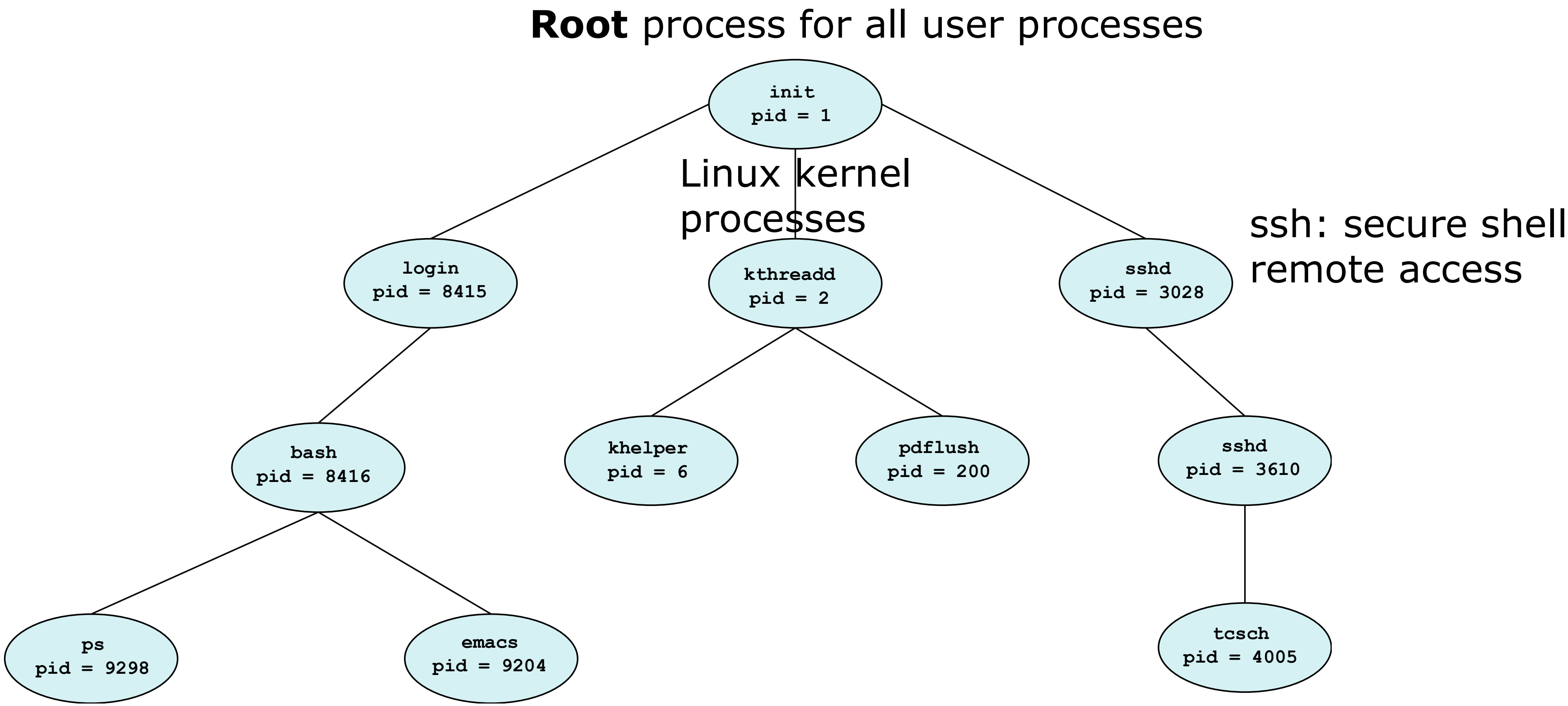
Operations on Processes

- Processes can execute concurrently, thus they may be **created** and **deleted** dynamically.
  - Operating System must provide mechanisms for process creation, termination, and so on.
  - Generally, process identified and managed via a unique **process identifier** (**pid**)
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes



  - So who created this universe (i.e., the first process)?

A Tree of Processes in Linux



type ps –el in Linux shell

Process Creation

- Child process needs memory address space
  - Child duplicate of parent: **fork()** system call
  - Child has a program loaded into it: fork() → **exec()** system call

UNIX examples: Process Creation

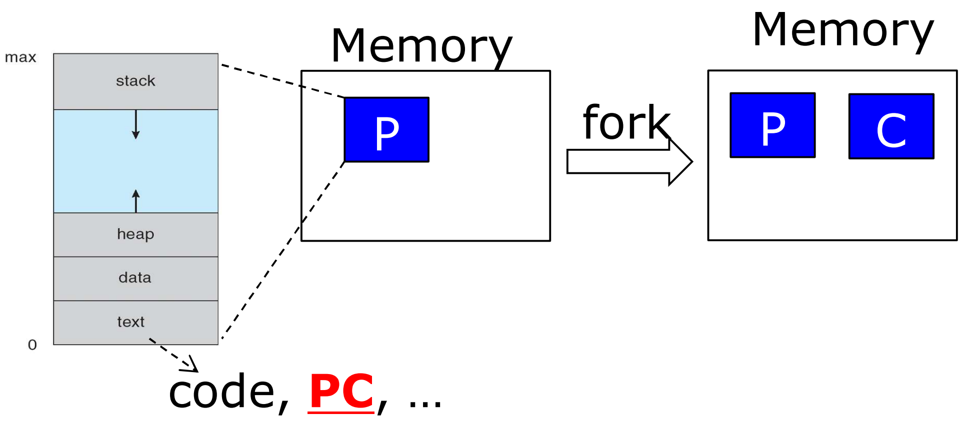
- UNIX examples
  - **`fork()`** system call creates a new process
- Both processes (parent and child)
  - continue execution after **`fork()`**,
  - with one difference: **<u>`fork()` return value</u>**:
    - Child process: 0
    - Parent process: **`pid`** of child process (>0)

      process identifier (<u>pid</u>)

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

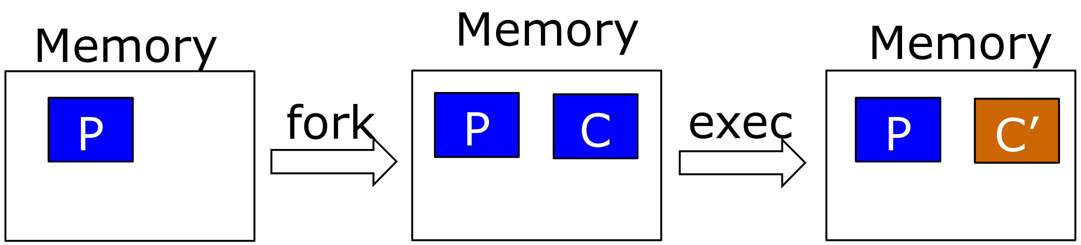
Parent

Child



Process Creation

- Memory address space
  - Child duplicate of parent: fork()
  - Child has a program loaded into it: exec() system call
    - **exec()** system call used after a **fork()** to *replace* the process' memory space with a new program
      - loads a new binary file into memory (deletes original memory – copy of parent)




```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Parent

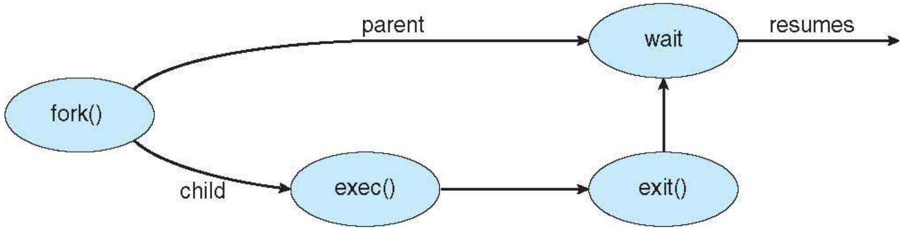
Child

Process Creation

- Execution options
  - Parent and children: Which executes first?
  - Parent can
    - execute concurrently with child or
    - wait until children terminate: `wait()` system call returning the pid:

```
pid_t pid; int status;
pid = wait(&status);
```

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```



Process Termination

- Process executes last statement and asks the operating system to delete it: **`exit()`** system call or just return()
  - Return status value (integer) to parent process
  - Process' resources (memory, open files, I/O buffers) are deallocated by operating system
- Parent may **terminate** execution of children processes: **`kill(pid,SIGKILL)`**
  - E.g., child has exceeded allocated resources, task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
    - All children terminated - **cascading termination**

Question 1

- How many processes are created ? (including the parent process)

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   /* fork a child process */
   fork();

   /* fork another child process */
   fork();

   /* and fork another */
   fork();
   printf("process forked\n");
   return 0;
}
```

Question 2

- Using the program in Figure, identify the values of **`pid`** at lines A, B, C, and D. (Assume that the actual **`pids`** of the parent and child are **`2600`** and **`2603`**, respectively.)

- recall
  - **`fork()`** return value:
    - Child process: 0
    - Parent process: **`pid`** of child process (>0)

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
      pid1 = getpid();
      printf("parent: pid = %d",pid); /* C */
      printf("parent: pid1 = %d",pid1); /* D */
      wait(NULL);
    }

    return 0;
}
```

Figure 3.29 What are the pid values?

Question 3

- What is the output of line A?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

Figure 3.30 What output will be at Line A?

`fork()` return value:

- Child process: 0
- Parent process: **`pid`** of child process (>0)

Chapter 3: Process Concept

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

Interprocess Communication

- In many cases, processes can ***cooperate*** with each others
  - Reasons for cooperating processes:
    - Information sharing – e.g., shared file
    - Computation speedup – e.g., parallel computing in multi-core environment
  - Example: PPT (100page) → Print
    - The **PPT process** needs to communicate to a **printer-process**, so that it sends data to the printer-process

Interprocess Communication

- Why can't processes just communicate with each other?
  - A process cannot directly access other process's **memory** – why not?
    - A: For system protection

- Cooperating processes need **interprocess communication (IPC)** provided by OS via system call

Example: Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble (e.g., JavaScript, Flash, HTML5), entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 categories
  - **Browser** process manages user interface, disk and network I/O (1 process created)
  - **Renderer** process renders web pages (deals with HTML, Javascript), new process for each website opened in a new tab
  - **Plug-in** process for each type of plug-in (e.g. Flash, QuickTime)

Communications Models

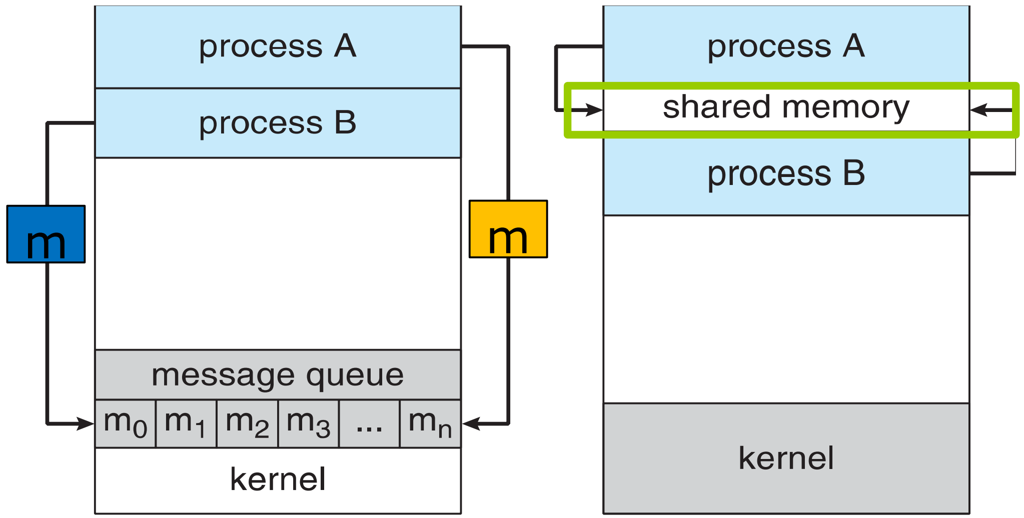
Mechanism

for processes to **communicate** & to **synchronize** their actions

Two fundamental models of IPC

Shared memory

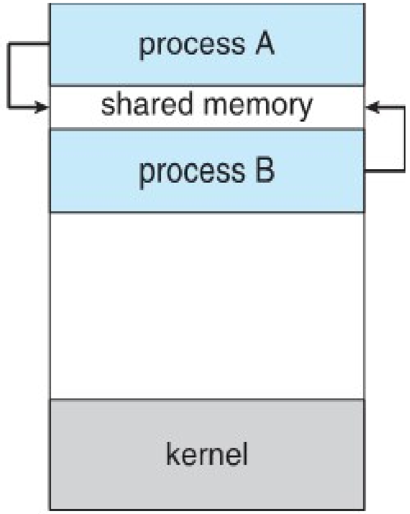
Message passing



(a) Message passing (b): Shared memory

Shared-Memory Systems

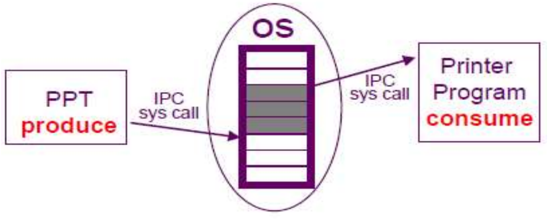
- OS prevents one process from accessing another process's memory – Protection
  - **Shared memory**: Two or more processes agree to remove this restriction



- A region of memory that is shared is created by system call
  - Processes exchange information by reading and writing data on shared area
  - All access to shared memory are treated as routine memory access
- The shared-memory can be modeled as **Producer-Consumer Problem**

Shared-Memory Systems

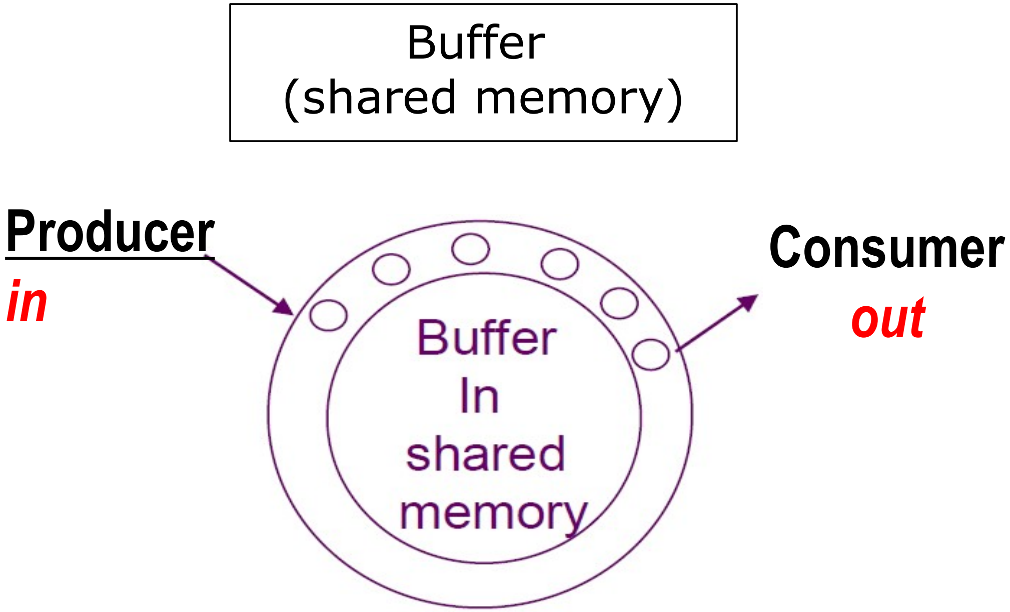
- Producer-Consumer Problem
  - A **producer process** produces information
  - that is consumed by **consumer process**.
    - e.g. producer : PPT – send to printer; consumer : printer program
    - e.g., producer: a web server produces -- HTML files and images ; consumer -- a client web browser



- The producer and consumer must be ***synchronized*** so that
  - **Producer** does not try to produce an item when **buffer** (shared memory) is *full*
  - **Consumer** does not try to consume an item that has not yet been produced (i.e., buffer is *empty*)

Shared Memory

```
#define BUFFER_SIZE 6

typedef struct
{
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;  //next free position in buffer
int out = 0; // first full position in buffer
```

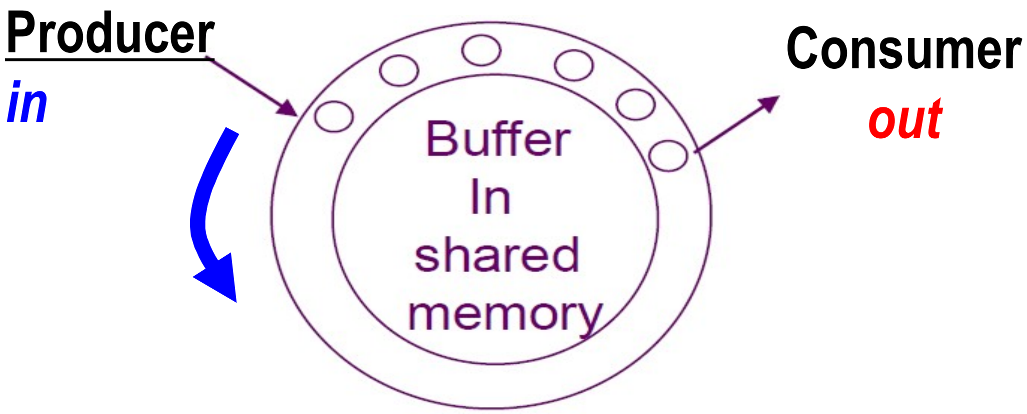


Circular Queue

Shared Memory (producer view)

```
while (true)
{
   /* Produce an item */
   while (((in + 1) % BUFFER SIZE) == out)
            ; /* do nothing --no free buffers */

   buffer[in] = next_produced;
   in = (in + 1) % BUFFER SIZE;
}
```

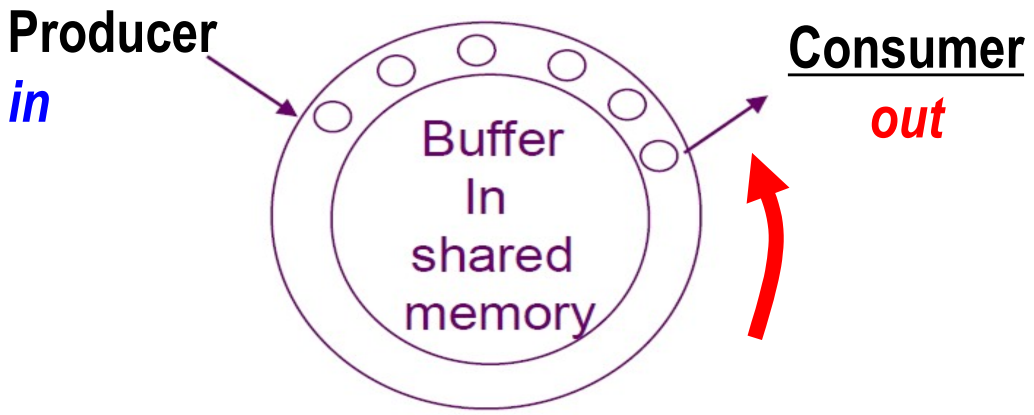
Make sure buffer is not full



Shared Memory (Consumer view)

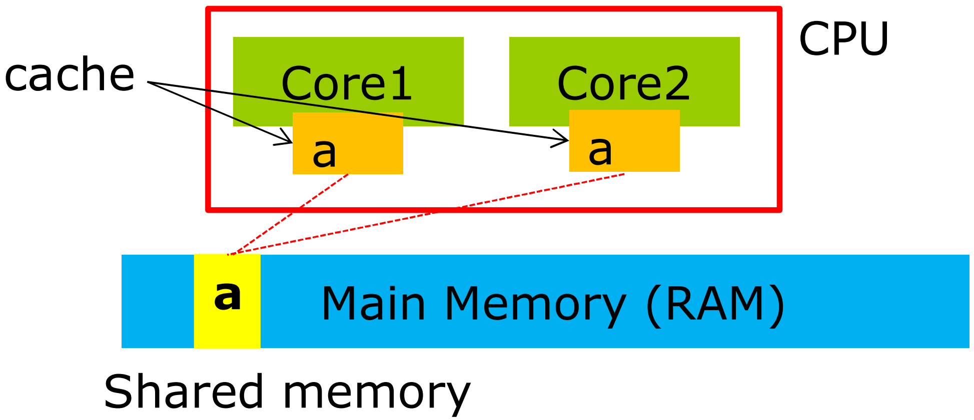
```
item nextConsumed;
while (true)
{
   while (in == out)
          ; // do nothing --nothing to consume
            // until remove an item from the buffer

   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER SIZE;
}
```

Make sure buffer is not empty



Shared Memory Systems

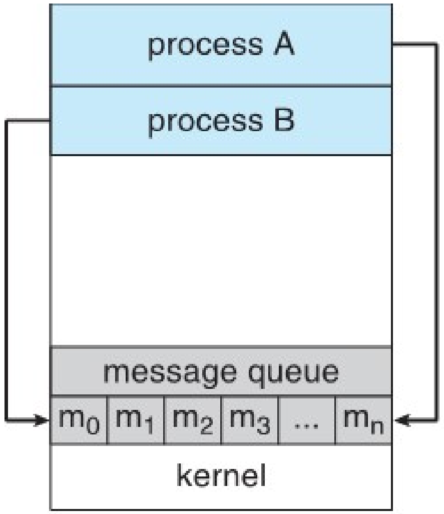
- **Bigger problem:** What happens if both producer process and the consumer process attempt to access the shared buffer concurrently?
  - Process synchronization
- **Another problem:** Multicore processors
  - Each core have separate cache – cache coherence problem



Interprocess Communication – Message Passing

- Message system – processes communicate with each other by **messages** without resorting to shared variables

- IPC facility provides two operations:
  - **`send`**(*message*) **`receive`**(*message*)



- If *P* and *Q* wish to communicate, they need to:
  - establish a ***communication link*** between them
  - exchange messages via send/receive
    - need ***system call***
    - More time-consuming compared to shared memory
  - e.g., **Microkernel structure**
  - e.g., **Sockets** (networking), RPC (distributed systems)

Message Passing Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous** and **Non-blocking** is considered **asynchronous**
- **Send**
  - **Blocking** send has the sender block until the message is received
  - **Non-blocking** send has the sender send the message and continue
- **Receive**
  - **Blocking** receive has the receiver block until a message is available
  - **Non-blocking** receive has the receiver receive a valid message or null