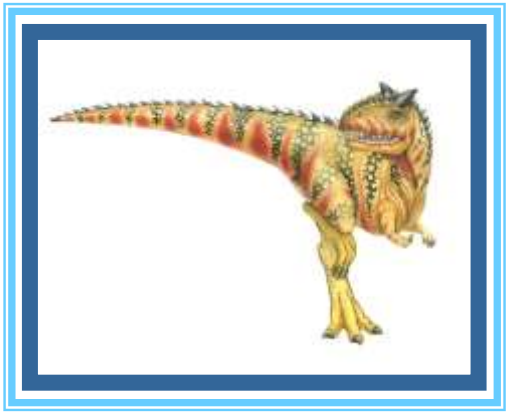
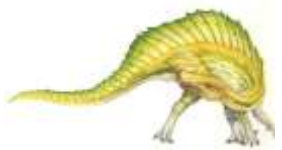
Chapter 3: Processes



Early computers allowed only one program to be executed at a time, this program had complete control of the system and had access to all the system's resources

In contrast, cotemporary computer systems allow multiple programs to be loaded into memory and executed concurrently

This evolution required firmer control and more fine division of the various programs; and these needs resulted in the notion of a process, which is **a program in execution**



Review

Main memory of the computer system is known to be

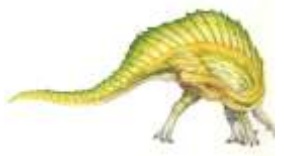
A. non volatile
B. volatile
C. reserved
D. Restricted

Controller of the computer system transfers data from device to

A. buffers
B. cache
C. registers
D. indexes

2. The return address from the interrupt-service routine is stored on the _____

a) System heap
b) Processor register
c) Processor stack
d) Memory



Review

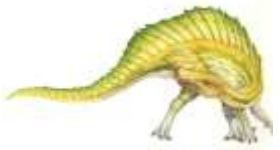
Table of pointers for interrupt to be executed contains the

A. interrupts
B. programs
C. addresses
D. compilers

More devices can be connected to computer system through

A. buffers
B. interrupt
C. registers
D. controllers

12. CPU as two modes privileged and non-privileged. In order to change the mode from privileged to non-privileged.
a) A hardware interrupt is needed
b) A software interrupt is needed
c) Either hardware or software interrupt is needed
d) A non-privileged instruction (which does not generate an interrupt)is needed



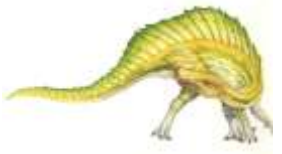
Review

Static programs of the computer system are stored in

A. RAM
B. ROM
C. hard disk
D. CD

2). DMA transfers data between——

- ○ Memory and processor.
- ○ Processor and I/O devices.
- ○ I/O devices and memory.
- ○ All of the above.



Review

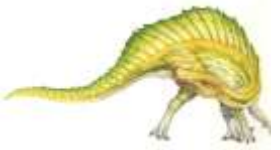
14. When the process requests for a DMA transfer?

a) Then the process is temporarily suspended

b) The process continues execution

c) Another process gets executed

d) process is temporarily suspended & Another process gets executed



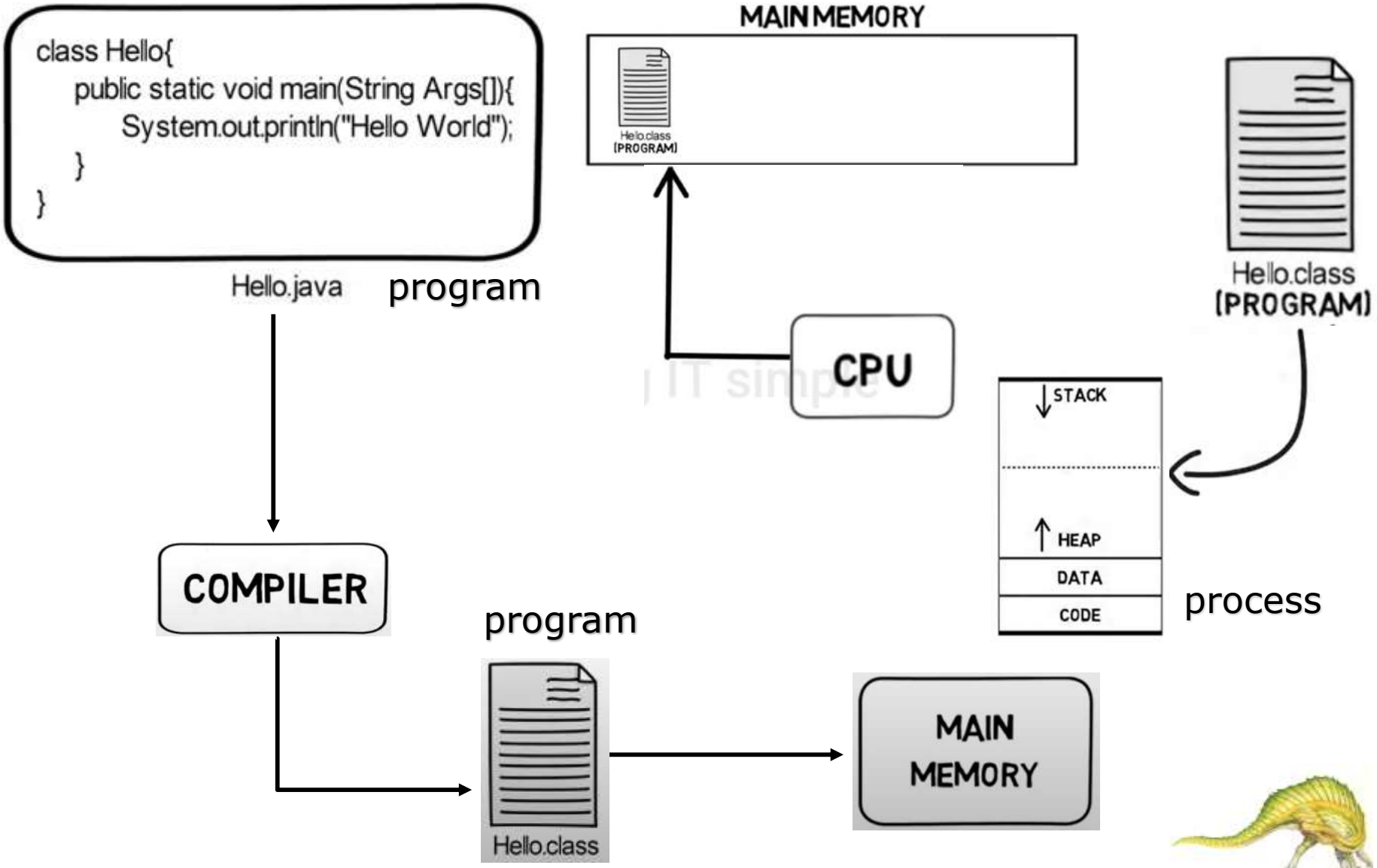
Process Concept

A process is the instance of computer program that is being executed.

OR

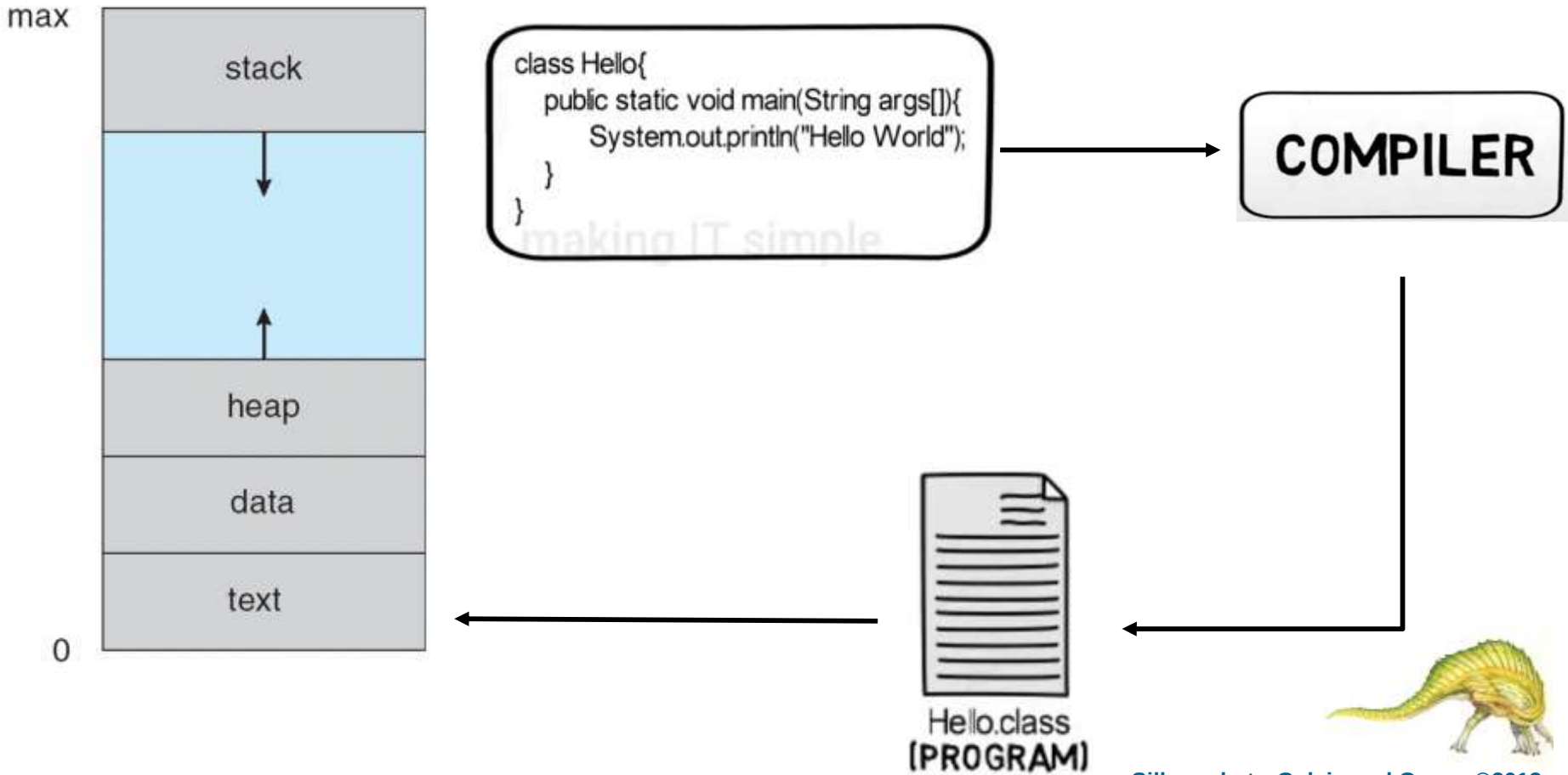
Process is basically a program in execution.

Process Concept



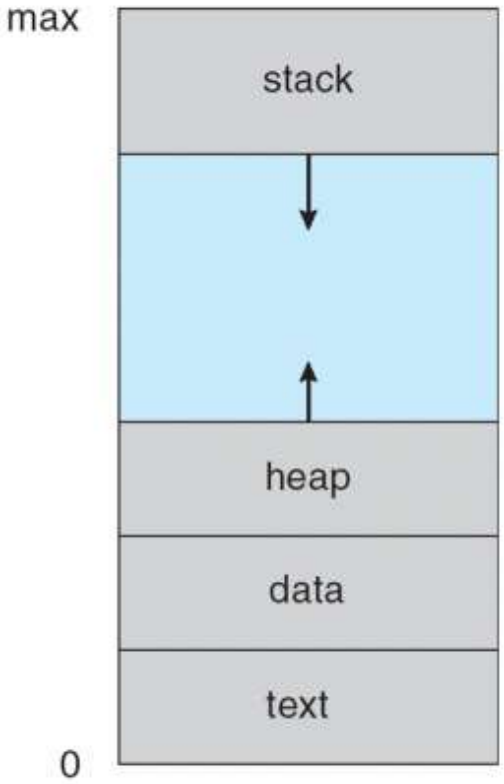
Process Concept

- **Process** – a program in execution
- Multiple parts
  - The program code, also called **text section**



Process Concept

- **Process** – a program in execution
- Multiple parts
  - **Data section** containing **global variables**, **static variables**



Static variables preserve their value even after they are out of their scope!

```c
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

prints 1 2

```c
#include<stdio.h>
int fun()
{
  int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

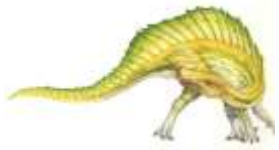
prints 1 1

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```

time

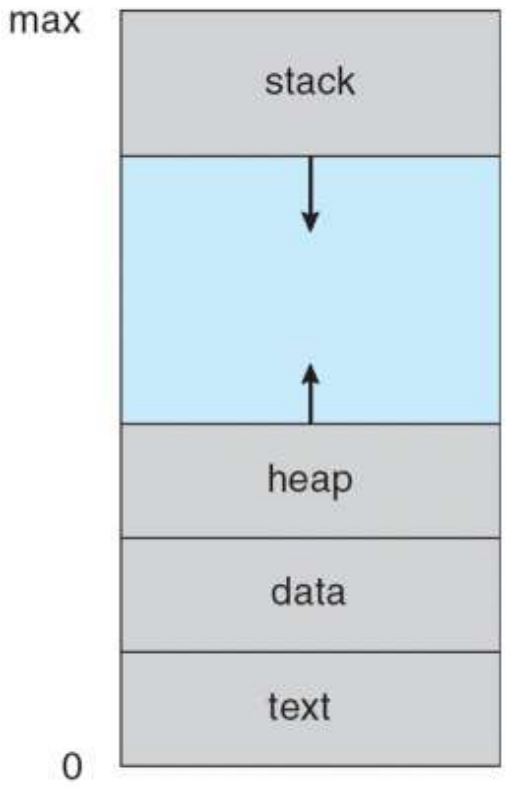
n

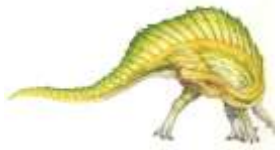
e



Process Concept

- **Process** – a program in execution
- Multiple parts
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables





Review: Function Call Stack

- What happens when a function call is made?
  - For each function, a STACK FRAME is created.

  - Stack Frame stores:
  - Return Address of caller
  - Input parameters
  - Local variables

Stake Frame of checkPositive()
return address of add
a=5 b=6

Stake Frame of add()
return address of main
— Line after return of add func
x=5 y=6 chk=TRUE

Stake Frame of main()
return address of C-Runtime
a=5 b=6 c=11

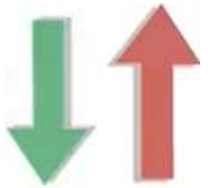
```
// C
bool checkPositive(int a,  int b){
   return  (a>=0  &&  b>=0);
}

int add(int x,  int  y){
   bool  chk = checkPositive(x, y);
   if (chk)
      return   x+y
   return -1;
}

int main() {
    int  a=5;  b=6;
    int  c = add(a,  b);
    printf("%d", c );
    return 0;
}
```

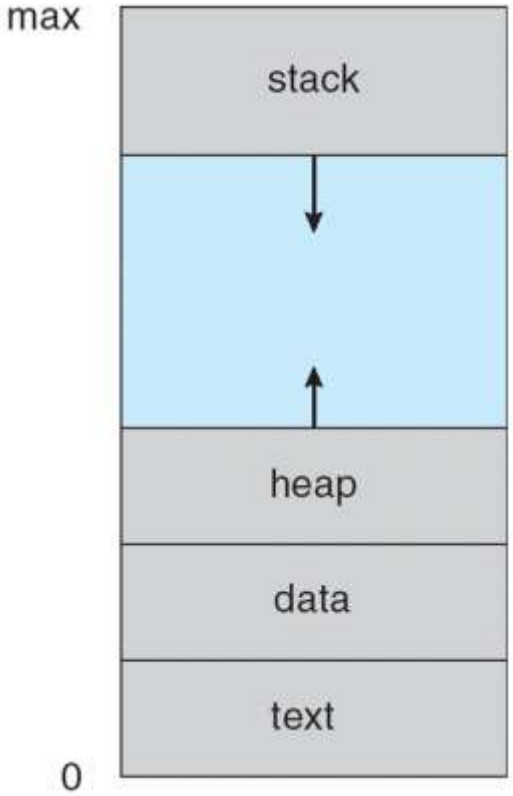
Process Concept

FIX DIVISION

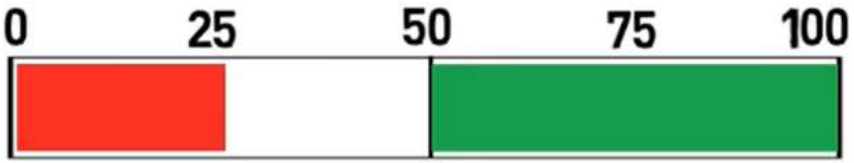
STACK

HEAP



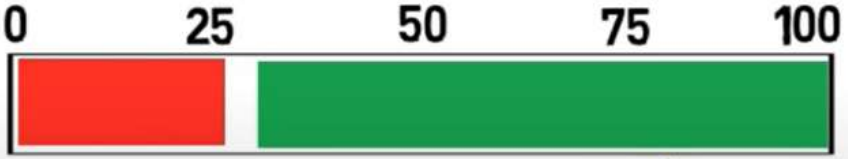


2 SUBJECTS



SCENARIO 1

20 MORE FOR SUBJECT 2



SCENARIO 2

Why there is no fixed space between stack and heap?

Why stack and heap propagate to reverse direction?

Task Manager

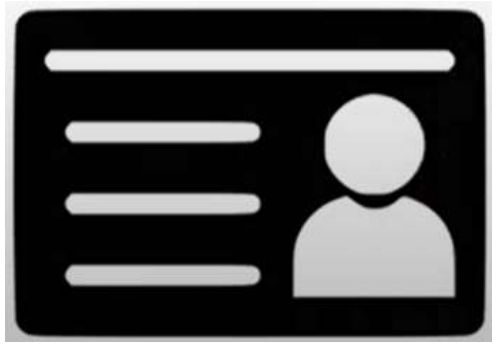
File Options View

Processes | Performance | App history | Startup | Users | Details | Services

Name	Sta...	12% CPU	72% Memory	1% Disk	0% Network	2% GPU	GPU eng
Google Chrome (21)		0.3%	32.5%	0 MB/s	0 Mbps	0.5%	GPU 0 -
Microsoft PowerPoint		0.4%	5.9%	0 MB/s	0 Mbps	0%	
Desktop Window Manager		0.6%	3.6%	0 MB/s	0 Mbps	0.8%	GPU 0 -
Antimalware Service Executable		0%	3.2%	0.1 MB/s	0 Mbps	0%	
Google Chrome		0%	2.9%	0 MB/s	0 Mbps	0%	
腾讯QQ (32 bit) (2)		0.2%	1.9%	0 MB/s	0 Mbps	0%	
Google Chrome		2.4%	1.4%	0 MB/s	0 Mbps	0%	
Windows Explorer		0.4%	1.2%	0 MB/s	0 Mbps	0%	
Google Chrome		0%	1.1%	0 MB/s	0 Mbps	0%	
Service Host: Diagnostic Policy Service		0%	0.9%	0 MB/s	0 Mbps	0%	
Google Chrome		0%	0.9%	0.1 MB/s	0 Mbps	0%	
Google Chrome		0%	0.8%	0 MB/s	0 Mbps	0%	
Task Manager		6.6%	0.8%	0 MB/s	0 Mbps	0%	
YoudaoDictHelper.exe (32 bit)		0%	0.8%	0 MB/s	0 Mbps	0%	
Google Chrome		0%	0.6%	0 MB/s	0 Mbps	0%	
Adobe Acrobat Reader DC (32 bit)		0%	0.5%	0 MB/s	0 Mbps	0%	
Google Chrome		0%	0.5%	0 MB/s	0 Mbps	0%	
Adobe RdrCEF (32 bit)		0%	0.4%	0 MB/s	0 Mbps	0%	
Google Chrome		0%	0.4%	0 MB/s	0 Mbps	0%	
网易有道词典 (32 bit) (2)		0.2%	0.4%	0 MB/s	0 Mbps	0%	

Process Control Block (PCB)

OPERATING SYSTEM --> MULTIPLE PROCESSES



process state
process number
program counter
registers
memory limits
list of open files
. . .

the repository

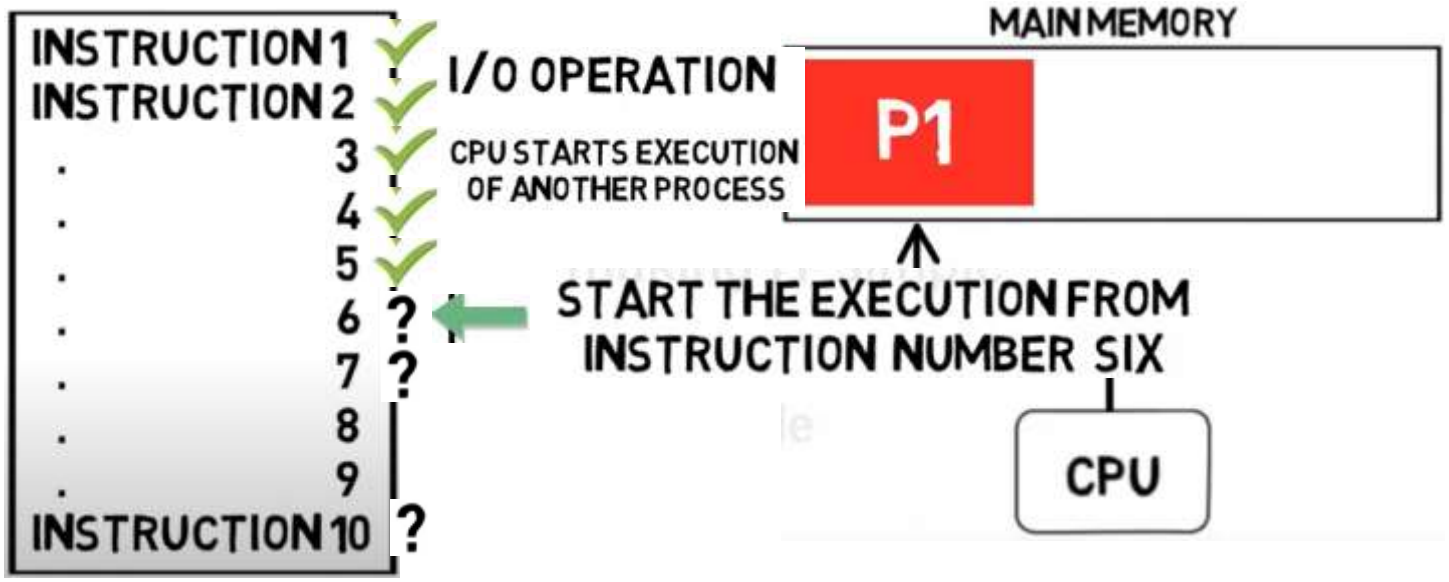
Each process is represented in the operating system by a **process control block**

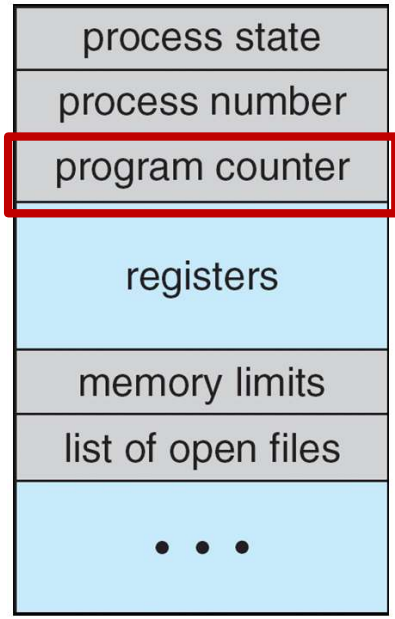
Process Control Block (PCB)

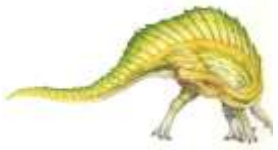
Information associated with each process

(also called **task control block**)

- Program counter – address of next instruction to be executed





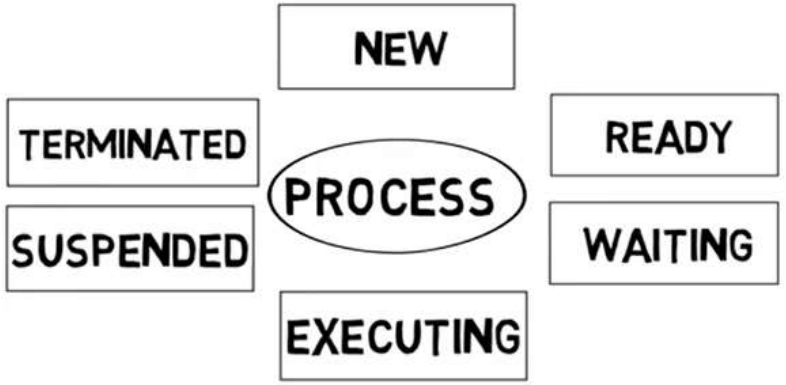


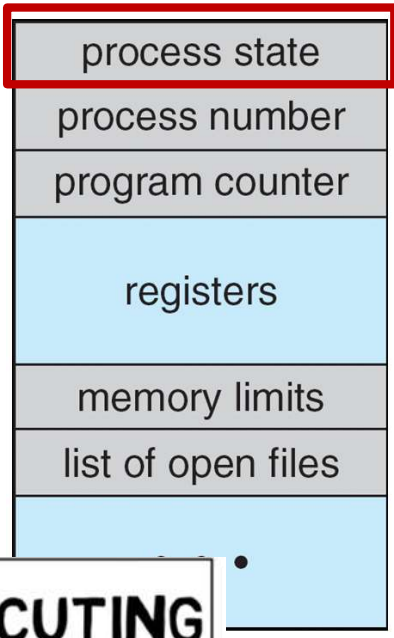
Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc





PROCESS BEING EXECUTED BY THE CPU — EXECUTING

PROCESS COMPLETES EXECUTION — TERMINATED

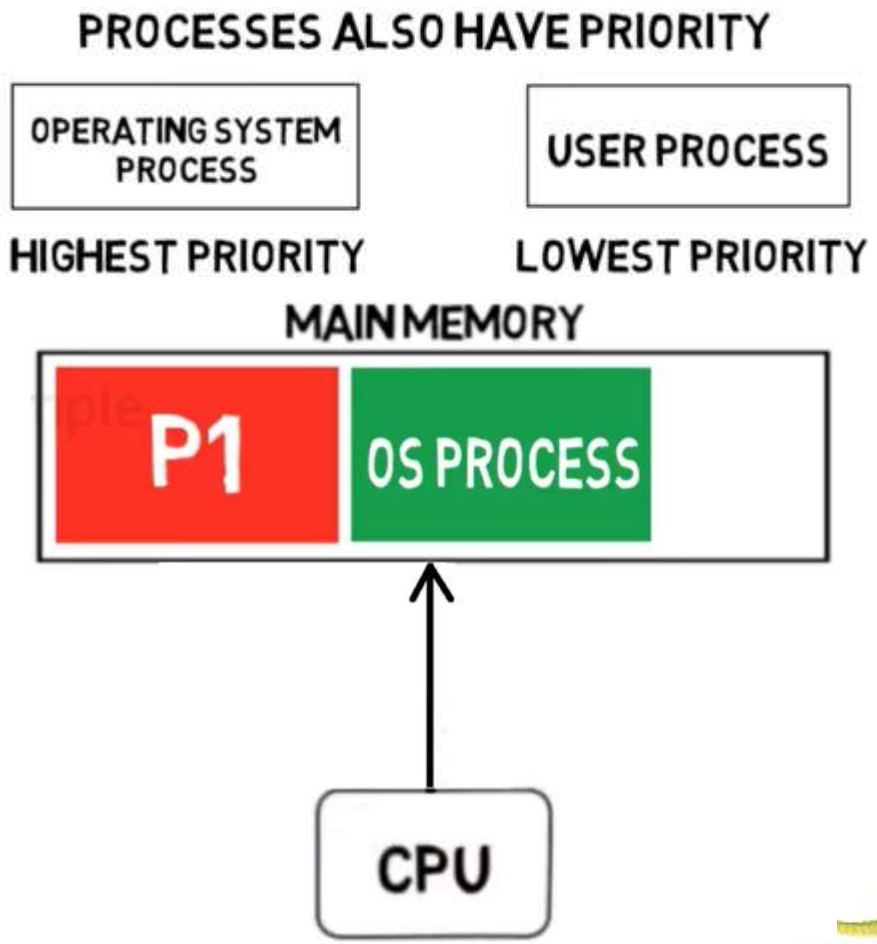
PROCESS IN RAM READY FOR EXECUTIUON — READY

Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process **Priority**

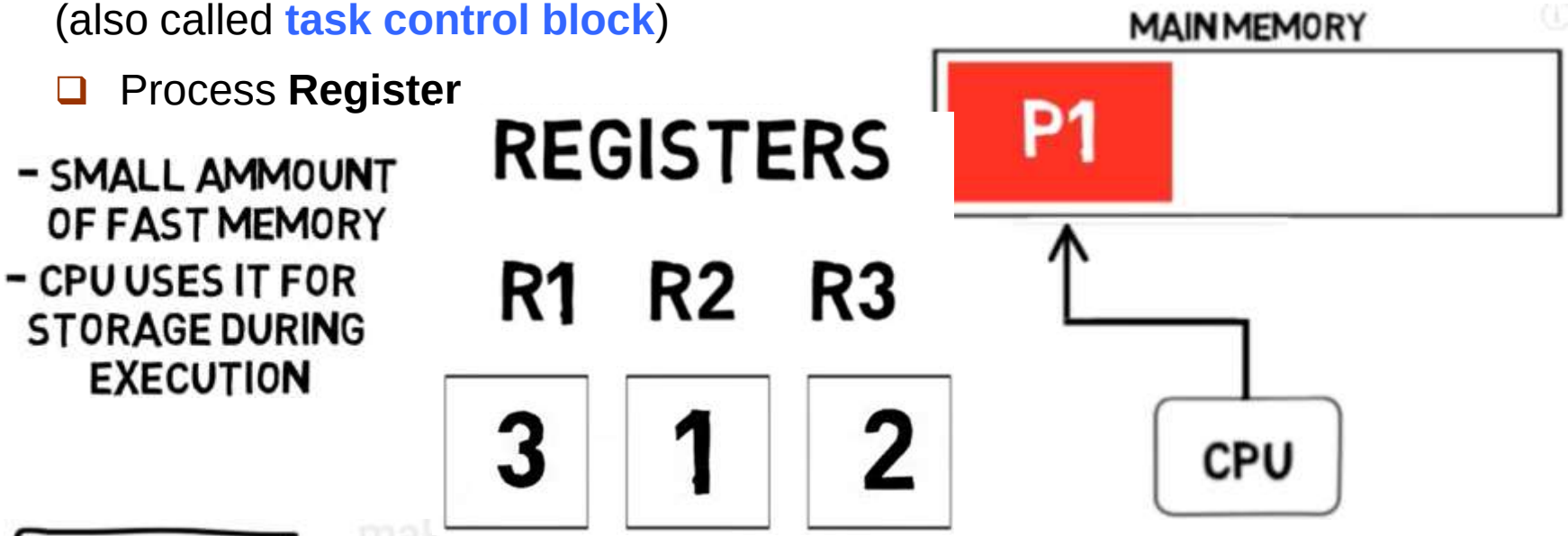


Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process **Register**



SP(stack pointer)

BP(base pointer)

IP(instruction pointer)

Process Control Block (PCB)

Information associated with each process

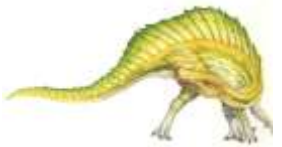
(also called **task control block**)

- Process **Register**

  SP(stack pointer)
  BP(base pointer)
  IP(instruction pointer)

- When a processes is running and it's time slice expires, the current value of process specific registers would be stored in the PCB and the process would be swapped out.

- When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the registers in the PCB.

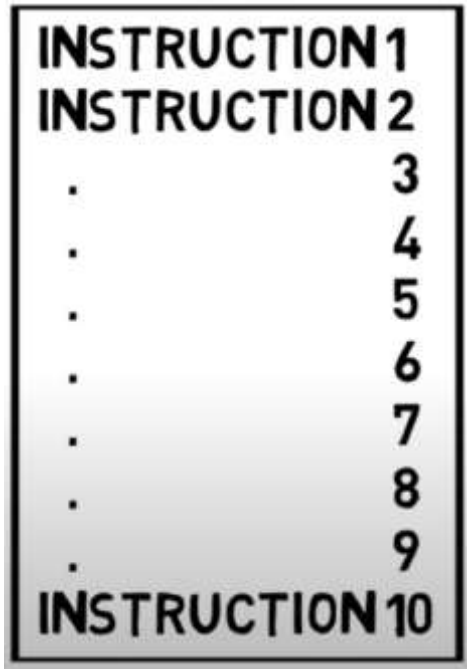


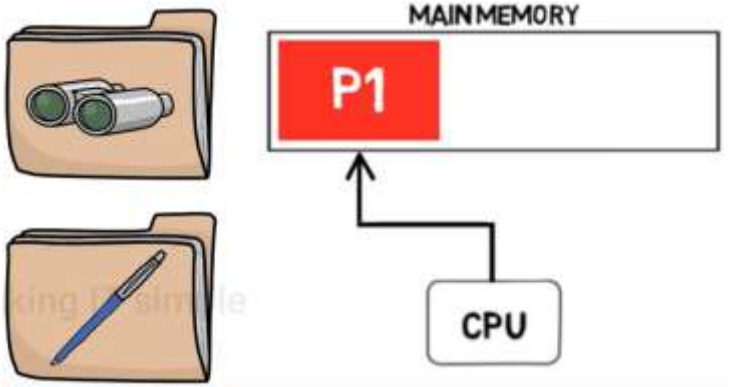
Process Control Block (PCB)

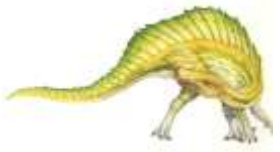
Information associated with each process

(also called **task control block**)

- List of Open Files







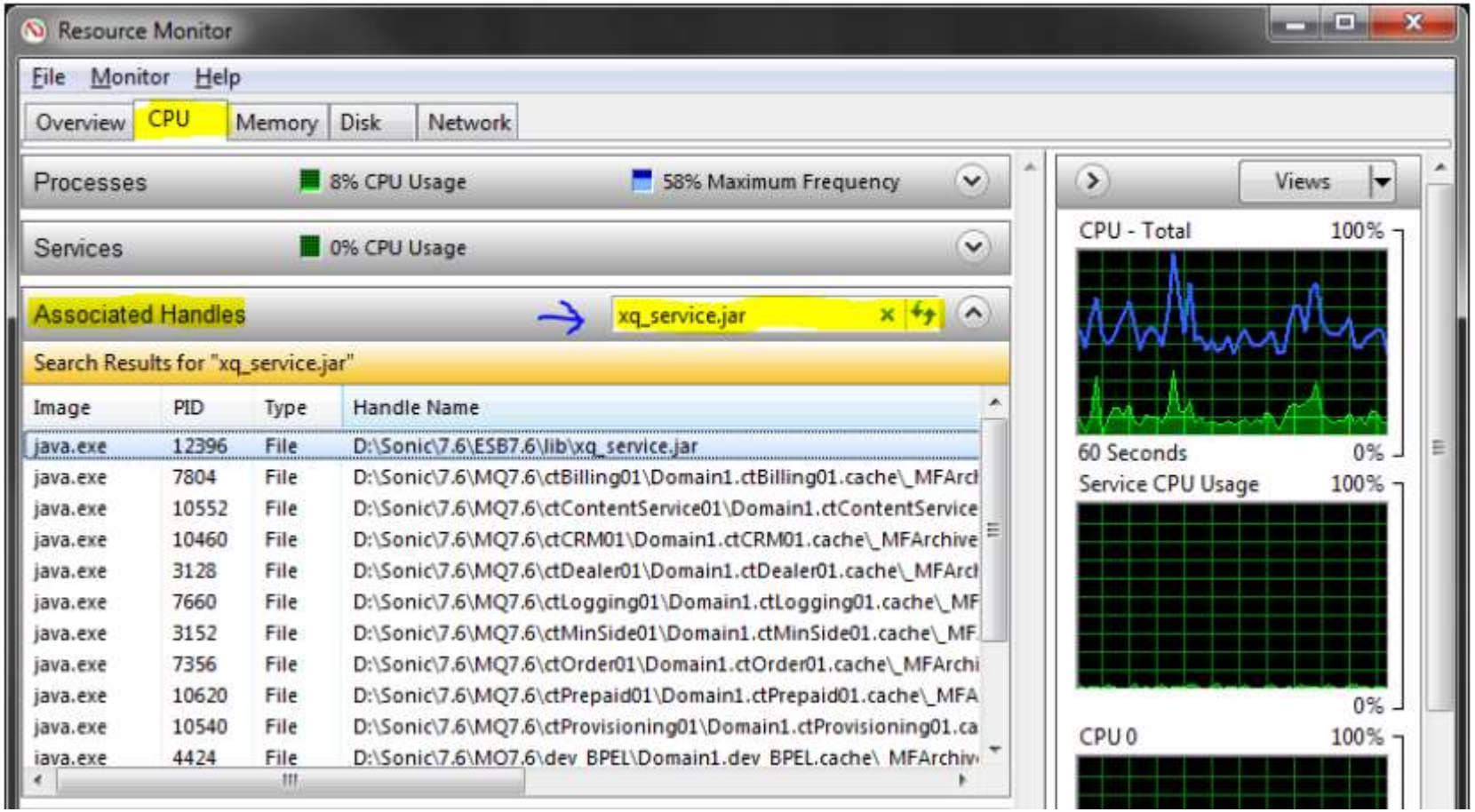
Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- List of Open Files

How do you find what process is holding a file open in Windows?

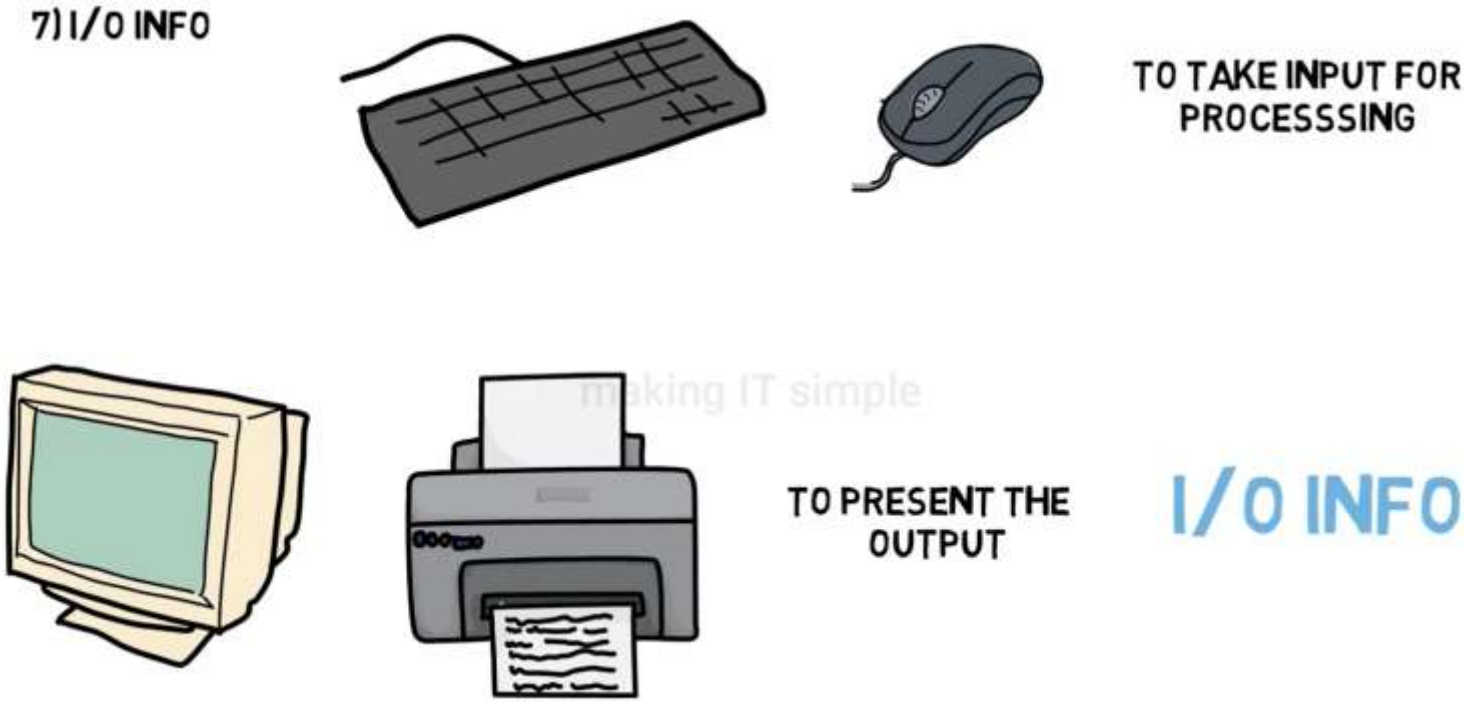


Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- I/O Info

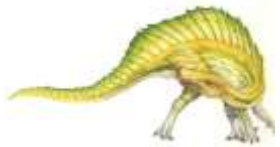


Process Control Block (PCB)

Information associated with each process

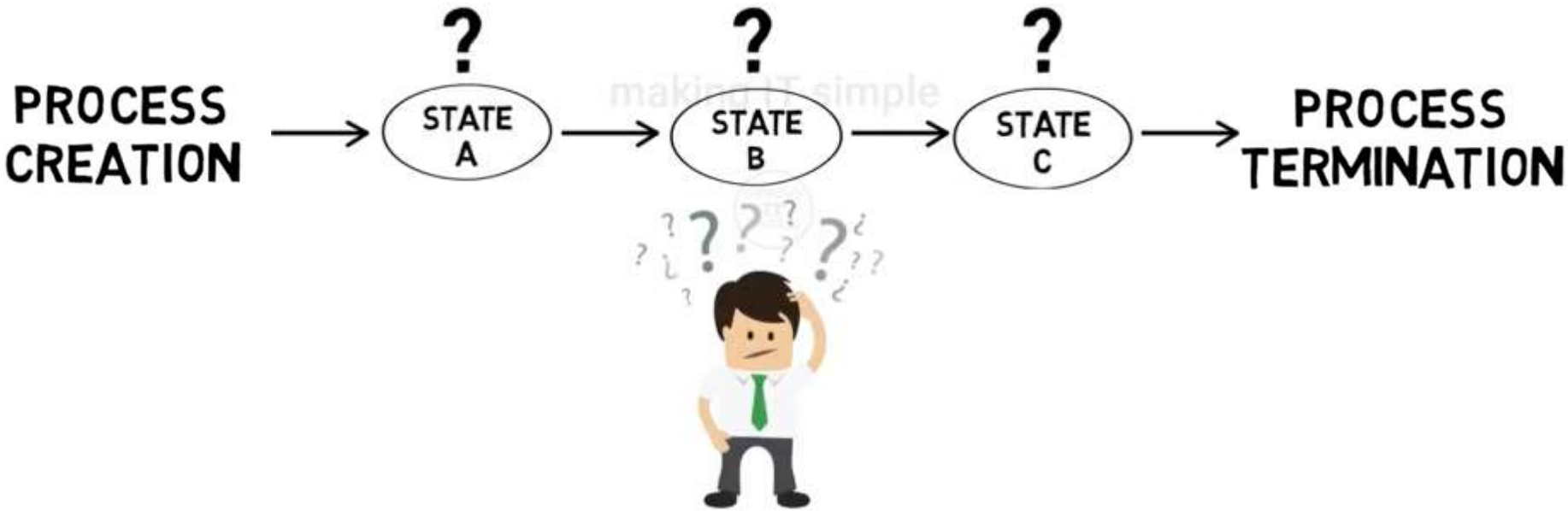
(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – address of next instruction to be executed
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Process State

The state of a process is defined in part by the current activity of that process



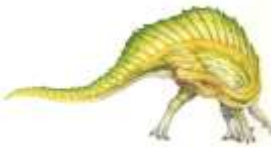
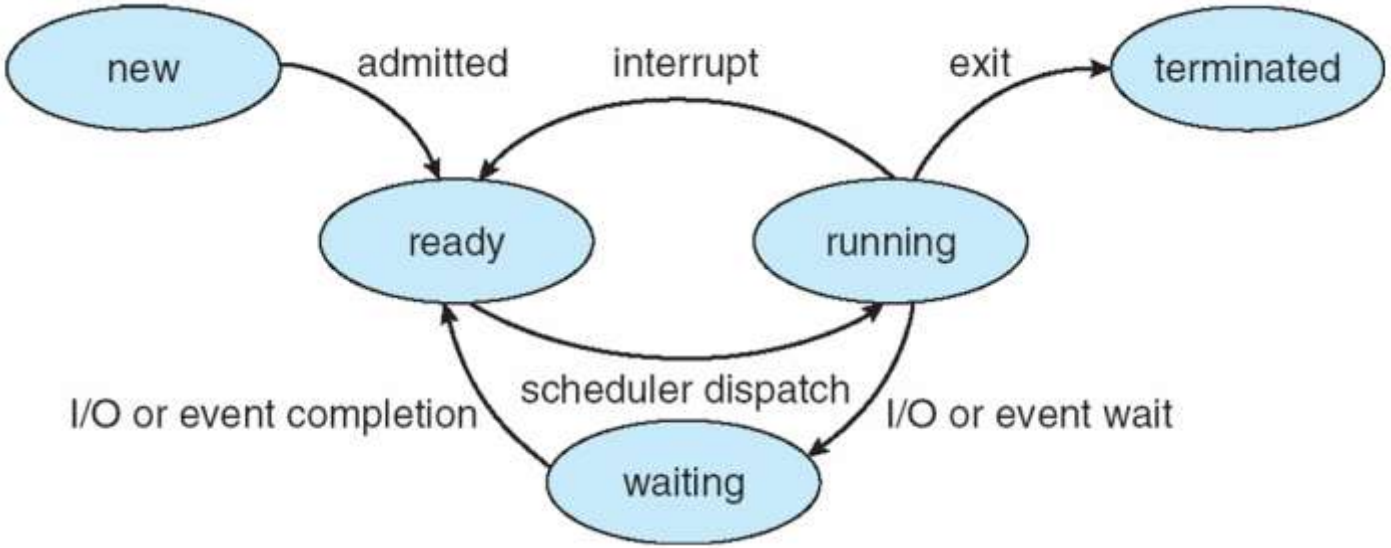
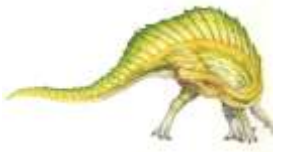


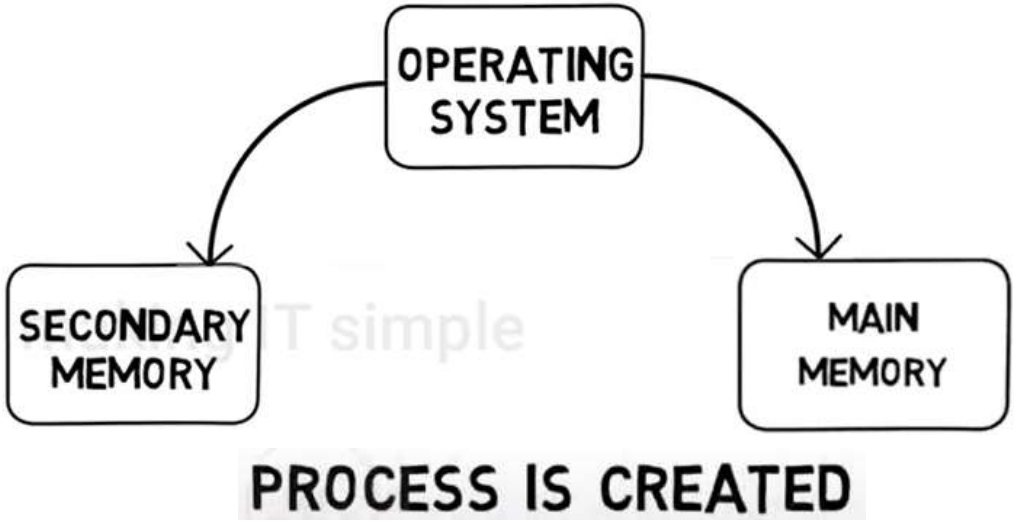
Diagram of Process State

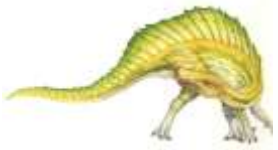




Process State

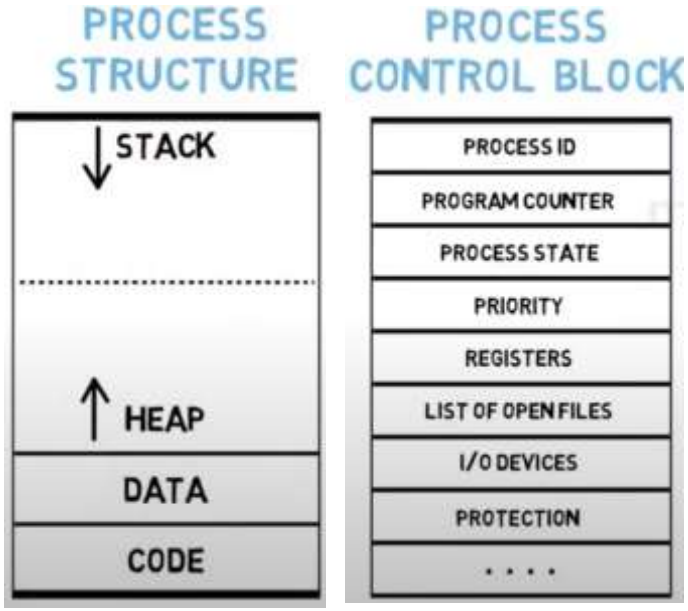
- As a process executes, it changes **state**
  - **new**: The process is being created

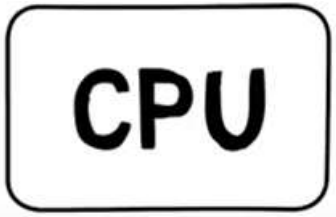


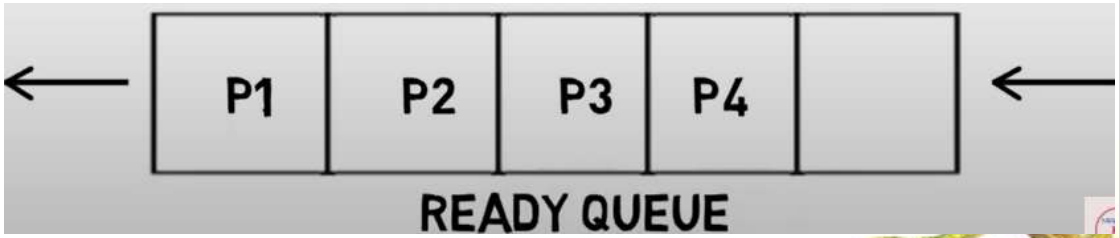


Process State

- As a process executes, it changes **state**
  - **ready:** The process is waiting to be assigned to a processor

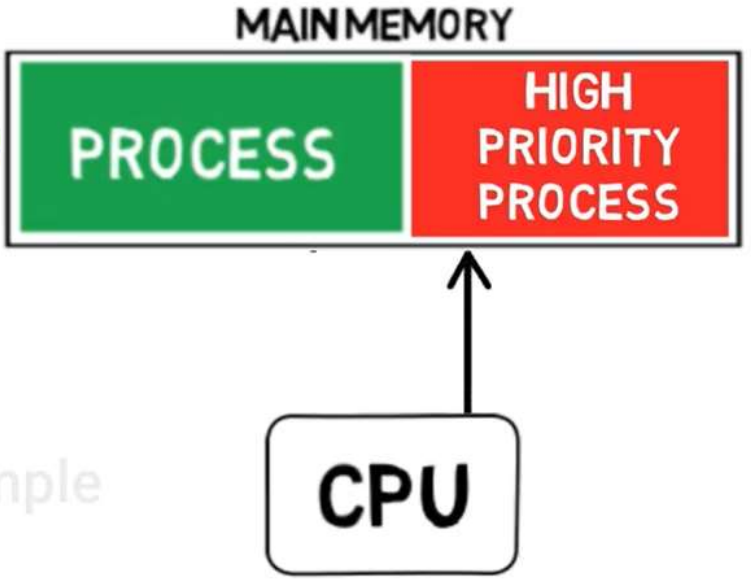


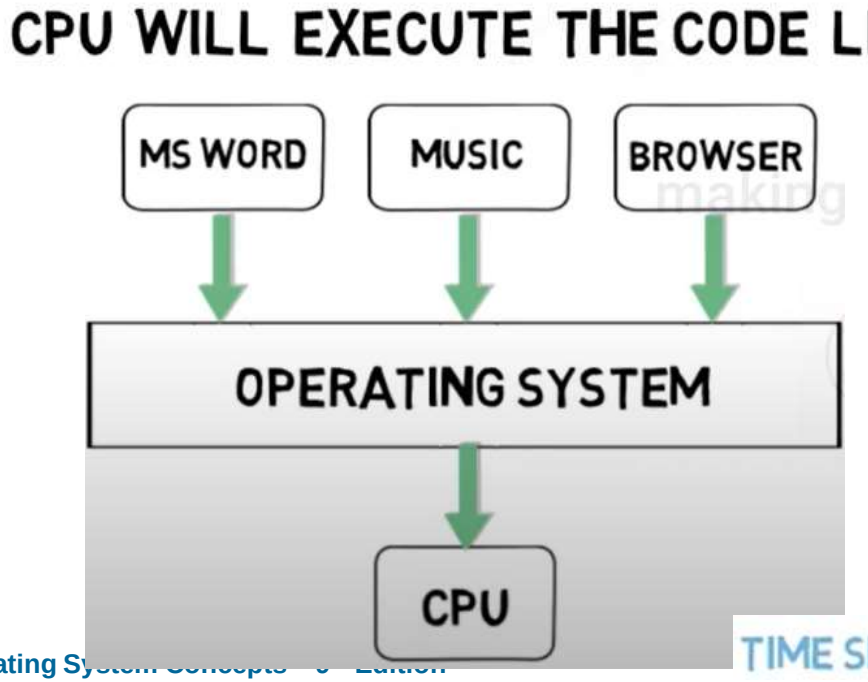




Process State

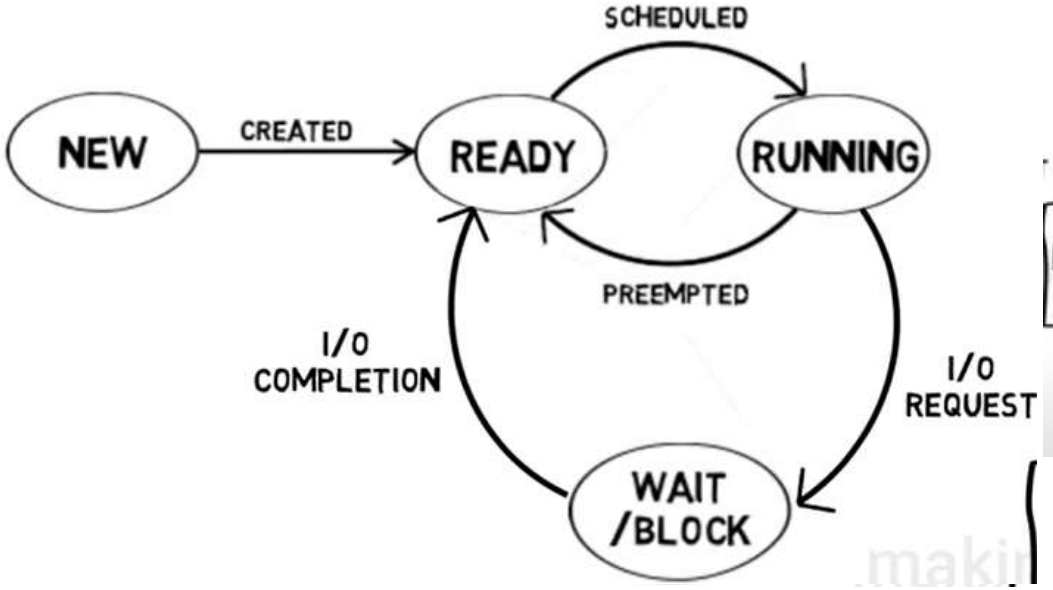
- As a process executes, it changes **state**
  - **running**: Instructions are being execut

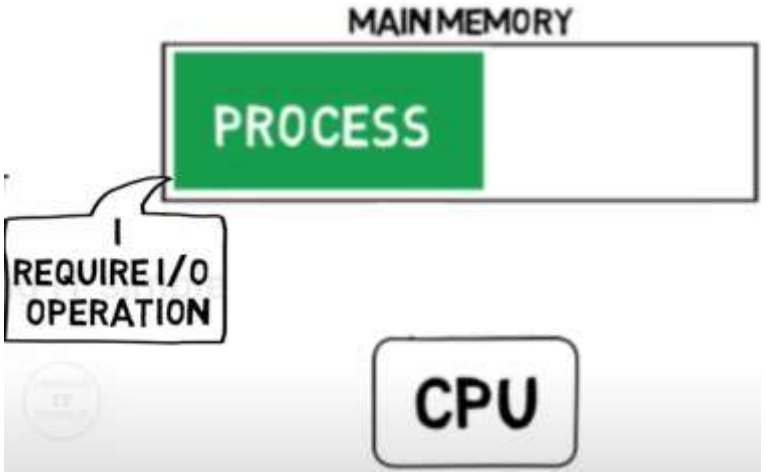


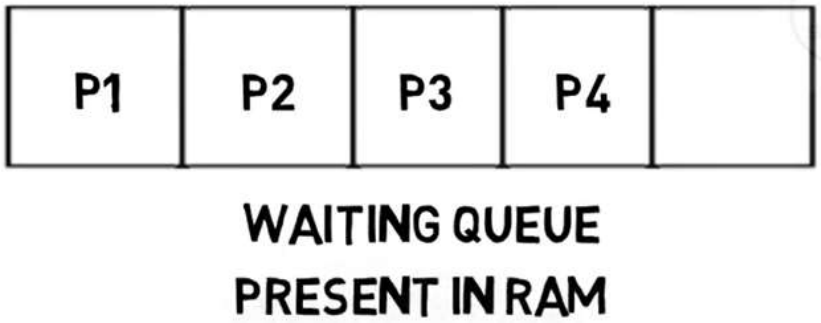


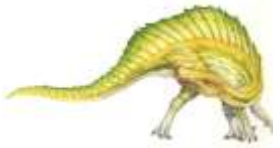
Process State

- As a process executes, it changes **state**
  - **waiting**: The process is waiting for some event to occur



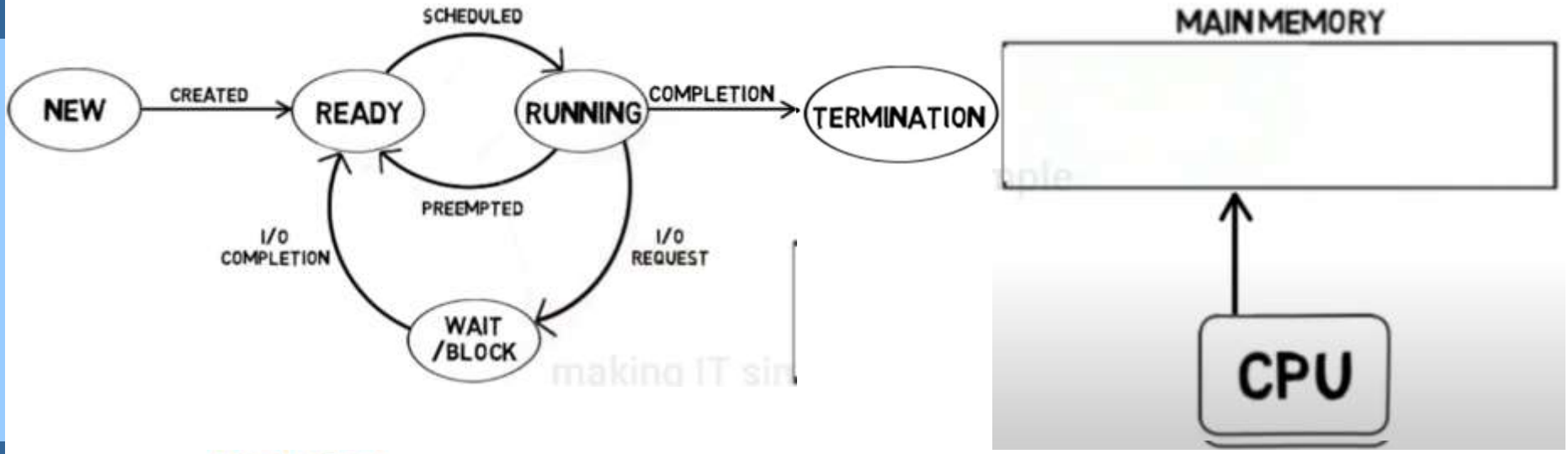






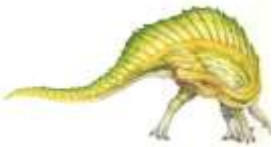
Process State

- As a process executes, it changes **state**
  - **terminated**: The process has finished execution

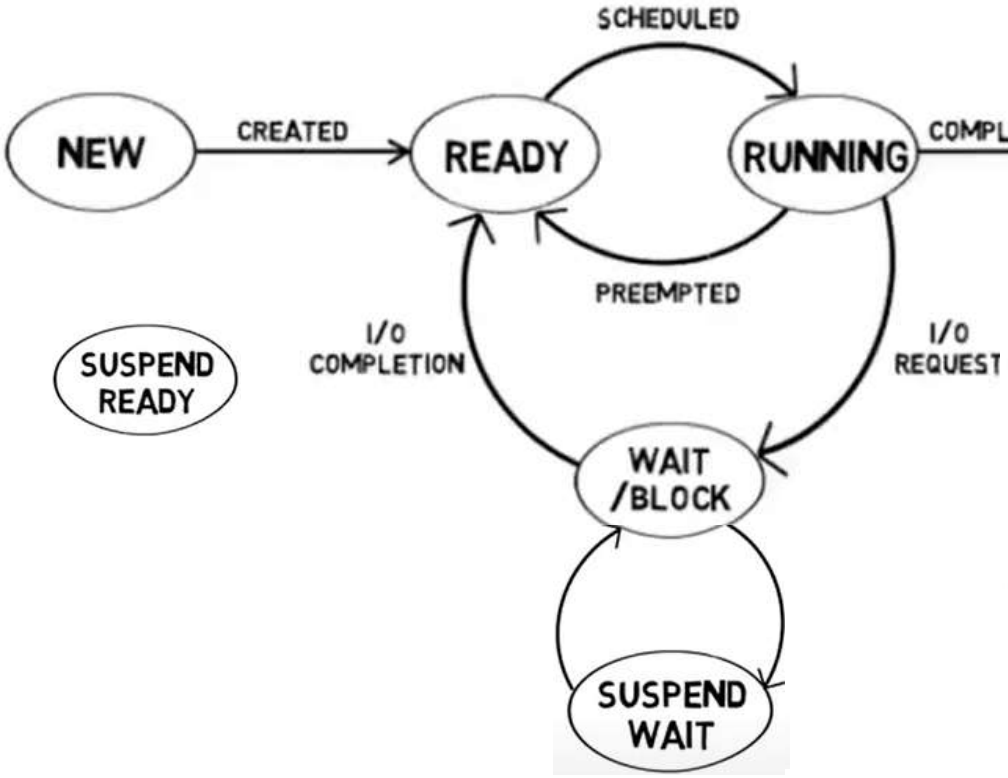


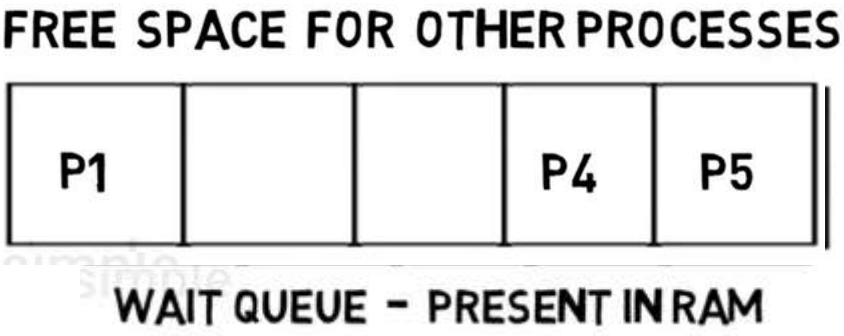
PROCESS STRUCTURE DELETED

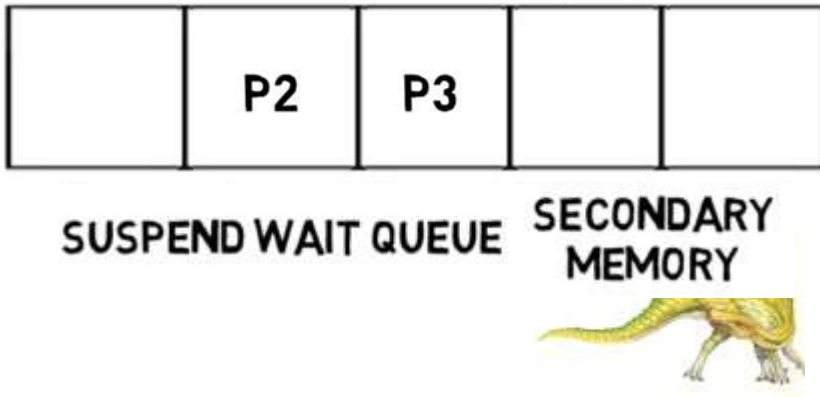
PROCESS CONTROL BLOCK DELETED



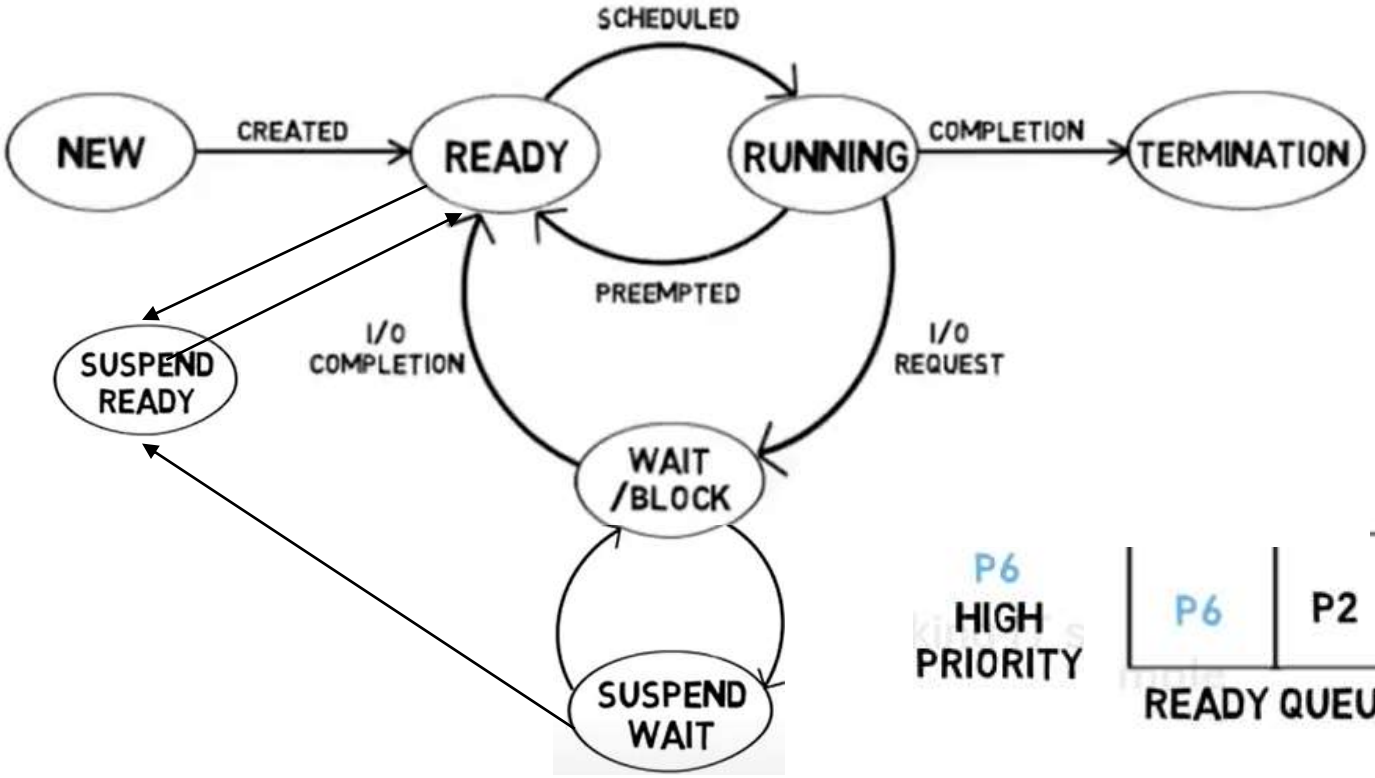
Process State







Process State



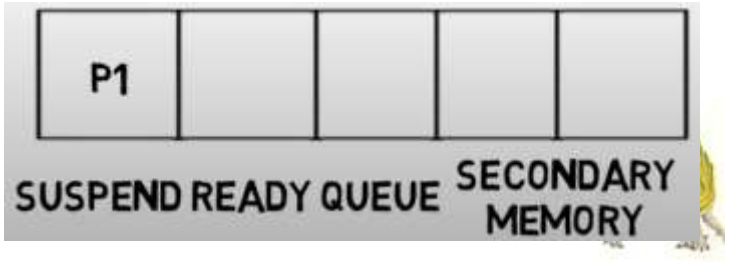
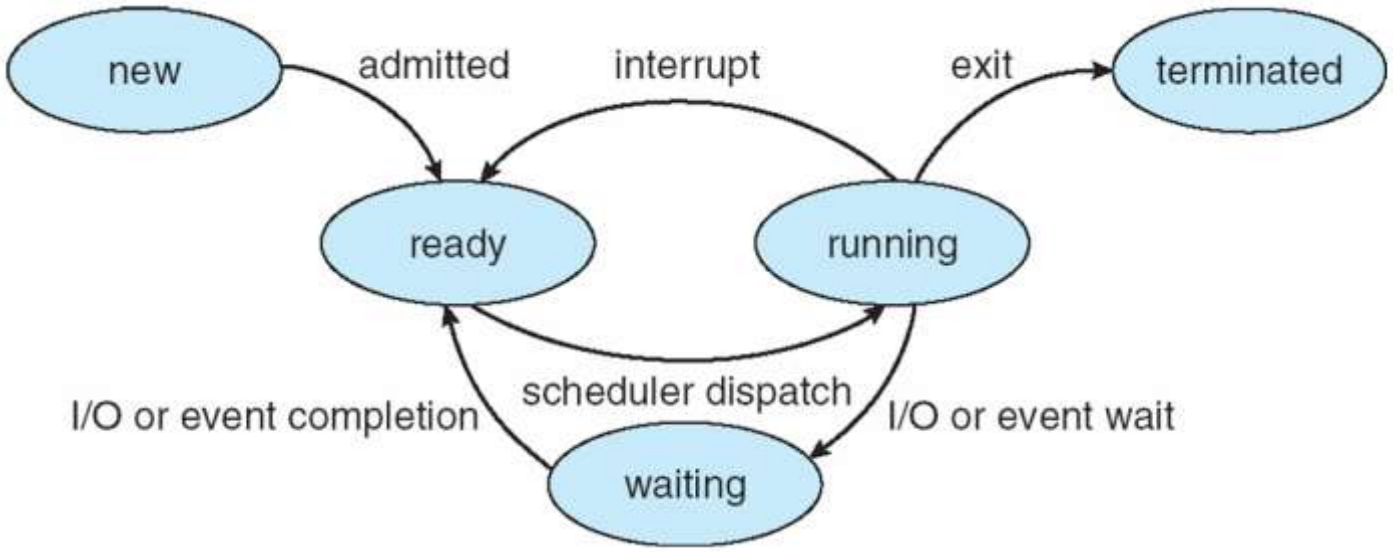
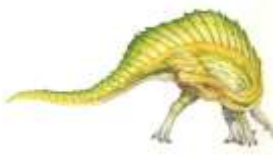


Diagram of Process State



Consider how the process changes states in the following scenarios:

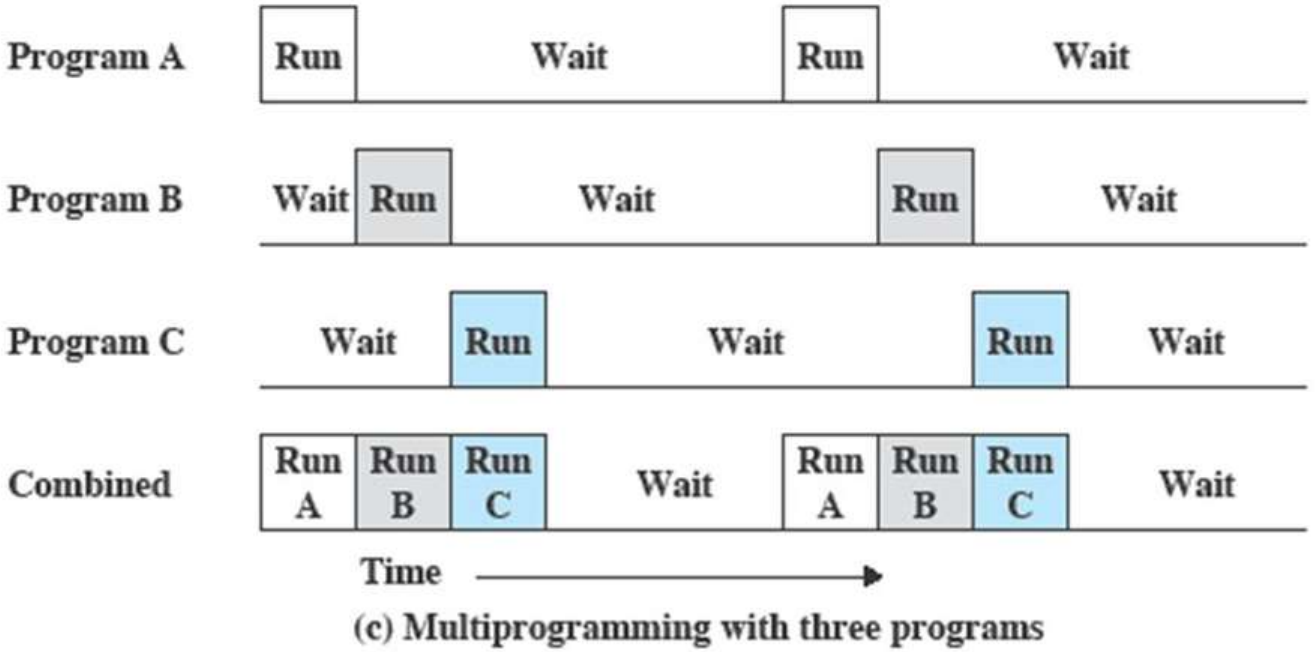
1. A process that writes data to a busy hard disk
2. A process that does extensive scientific computation



Process Scheduling

- The objective of **multiprogramming** is to have some processes running at all times, to maximize CPU utilization.

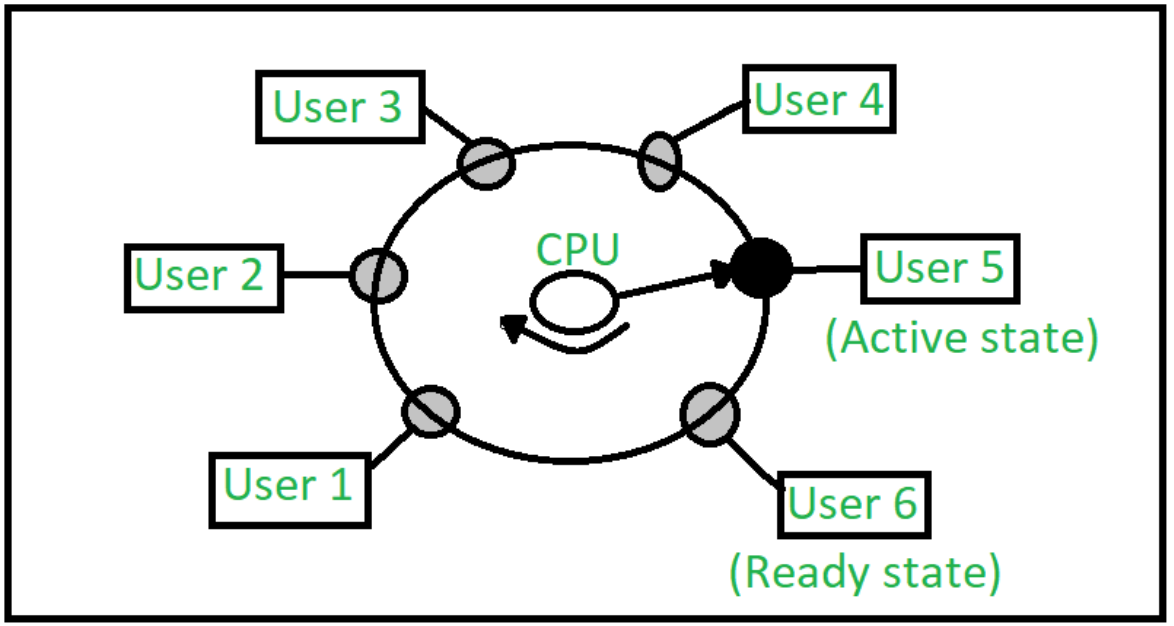
Multiprogramming



(c) Multiprogramming with three programs

Process Scheduling

- The objective **of time sharing** is to switch the CPU among processes **so frequently** that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process for program execution on the CPU



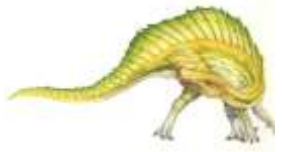
Process Scheduling

Time Sharing Systems:

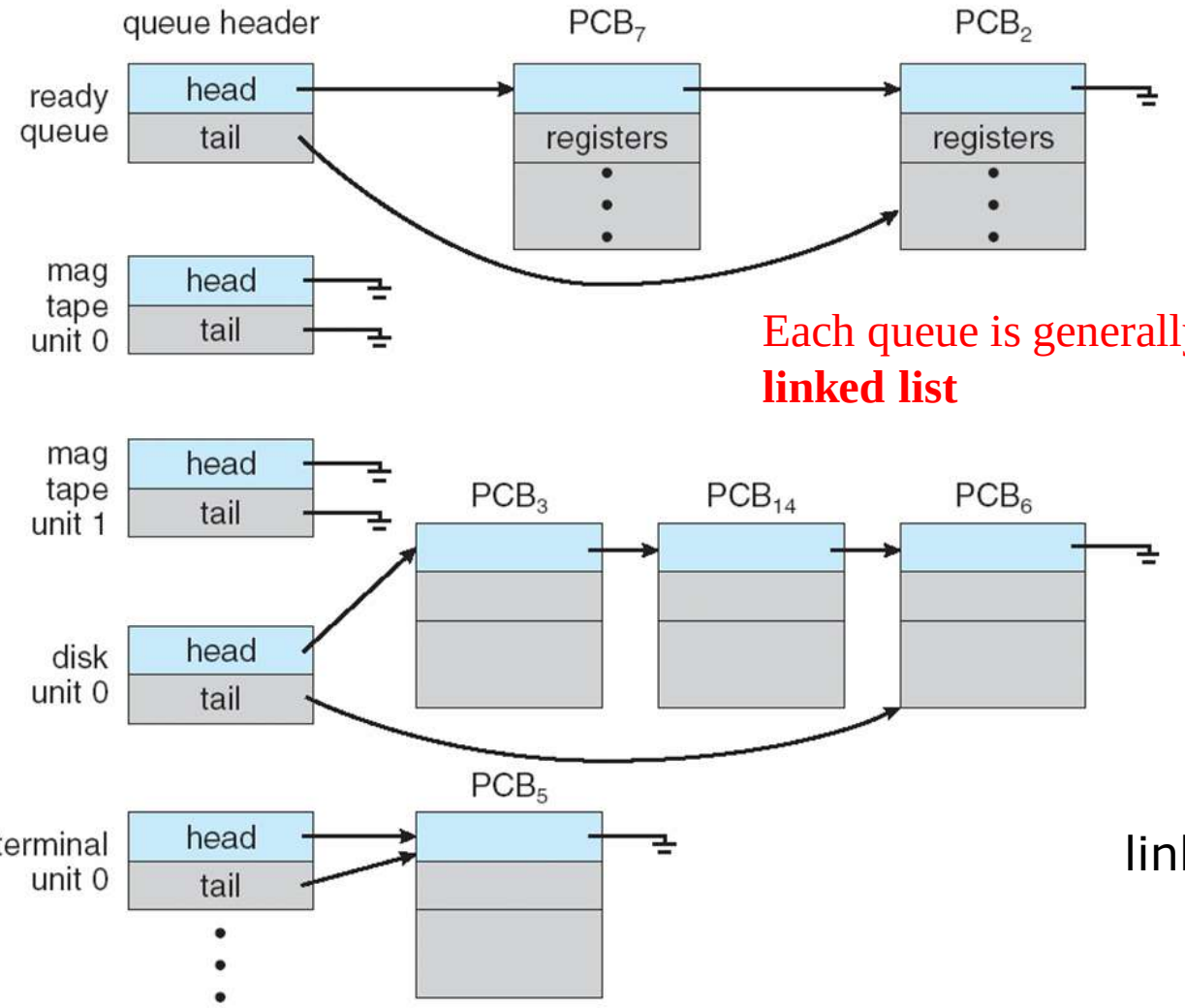
- Time sharing, or multitasking, is a logical extension of multiprogramming.
- Multiple jobs are executed by switching the CPU between them.
- In this, the CPU time is shared by different processes, so it is called as "Time sharing Systems".
- Time slice is defined by the OS, for sharing CPU time between processes.
- Examples: Multics, Unix, etc.,

Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues



Ready Queue And Various I/O Device Queues



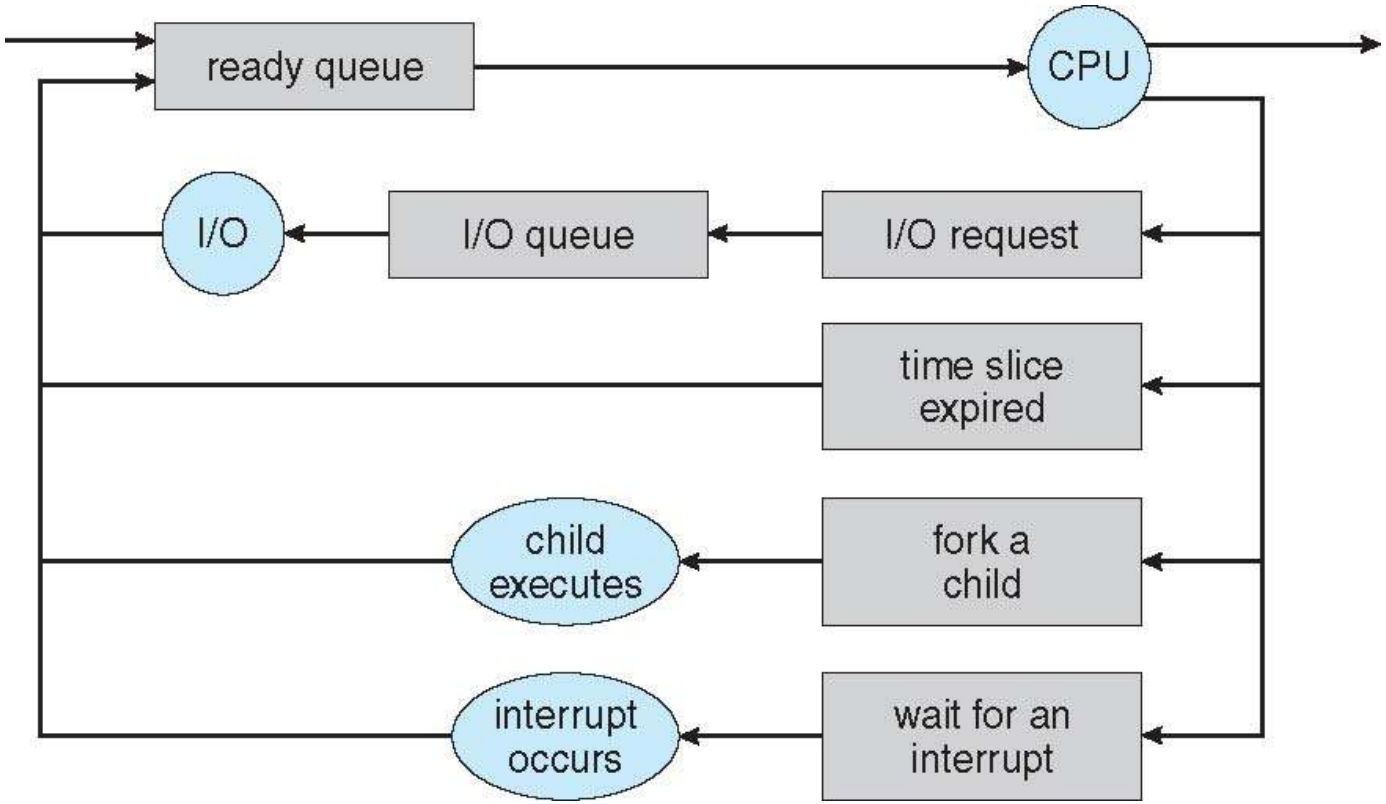
Each queue is generally stored as a **linked list**

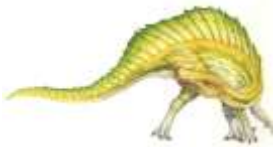
linked list

Each device has its device queue

Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

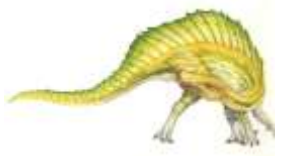




Schedulers

- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked **infrequently** (seconds, minutes)
  - The long-term scheduler controls the **degree of multiprogramming**
  - May be slow

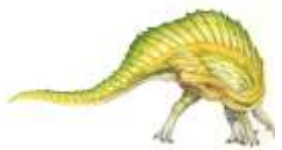
Degree of multiprogramming: the number of processes in memory.



Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Short-term scheduler is invoked **frequently** (milliseconds)
  - It must be fast, because of the short time between executions

For example, it the short-term scheduler takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100+10)=9 percent of the CPU is being wasted simply for scheduling the work

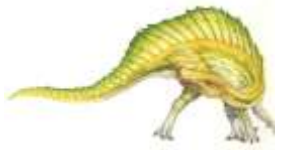


Schedulers (Cont.)

- 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; very long CPU bursts

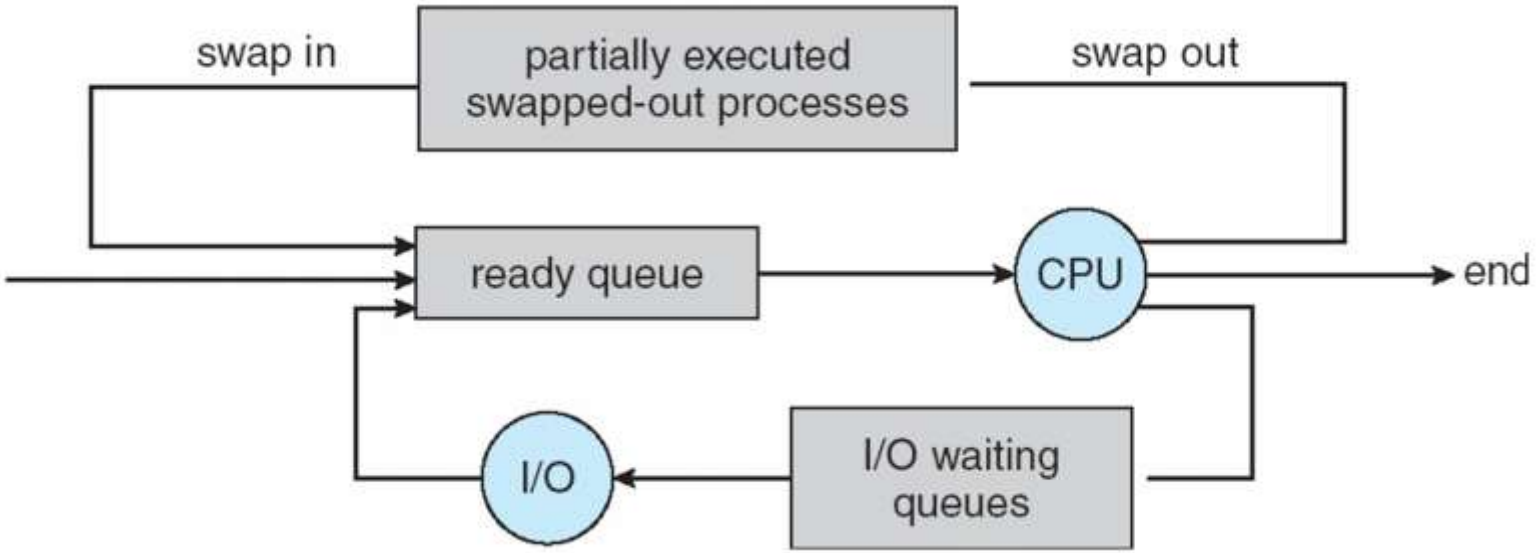
- Long-term scheduler strives for good ***process mix***
  - What happens if all processes are I/O bound?
  - What happens if all processes are CPU bound?

1. If all processes are **I/O bound**, the **ready queue** will almost always **be empty**, and **the short-term scheduler will have little to do**;
2. If all processes are **CPU bound**, the **I/O waiting queue** will almost always be empty, **devices will go unused**, and again the system will be **unbalanced**

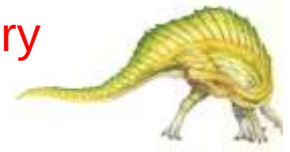


Addition of Medium-Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to **decrease**
  - Remove process from memory, store on disk, bring back from disk to continue execution: **swapping**
  - Why medium-term scheduler?



- May be necessary to improve the process mix or to free memory

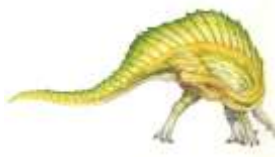


Quiz

After an interrupt occurs, **hardware** needs to save its current state (content of registers etc.) before starting the interrupt service routine. One issue is where to save this information. Here is one method:

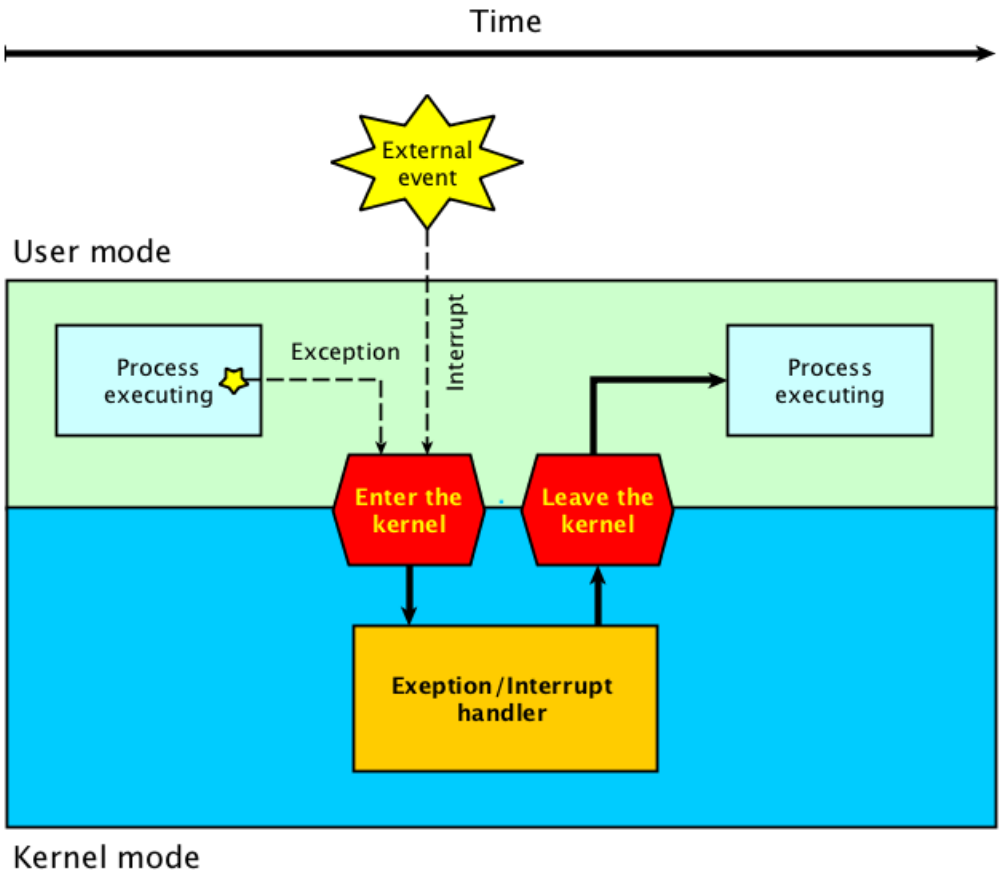
Put them in some special purpose internal registers which are exclusively used by interrupt service routine.

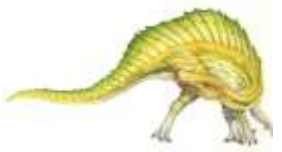
What's the problems with the above approach?



Context Switch

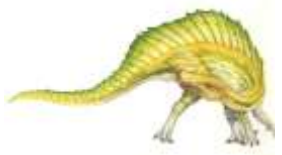
- **Interrupts** cause the OS to change a CPU from its current task and to run a kernel routine. Such operation happen frequently on general-purpose systems



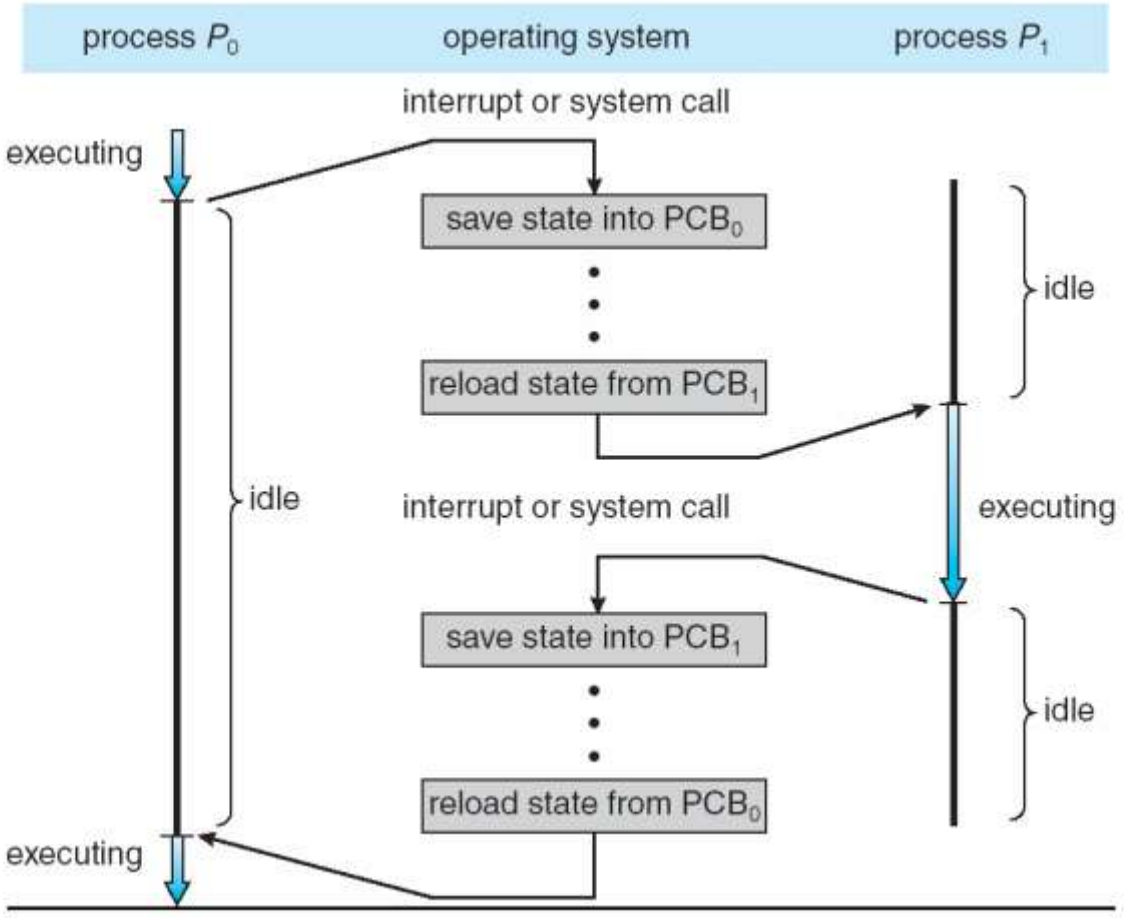


Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and **load** the **saved state** for the new process scheduled to run via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is **overhead**; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch

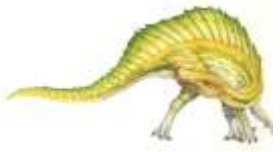


CPU Switch From Process to Process



Time-sharing systems switch the CPU among processes so frequently

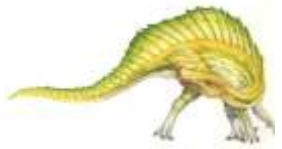
Context switch



Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,

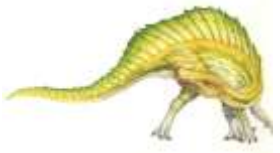
The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination



Process Creation

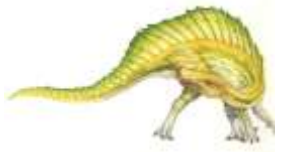
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

- Generally, process identified and managed via a **process identifier** (**pid**)
  - A unique value for each process
  - An index to access various attributes of a process



Process Creation

- **Fork()**
  - Returns a value of type **pid_t** (essentially, an integer), and
  - Does not take any input parameters, what is indicated by the formal parameter void.
  - The return code for fork is **0** for the child process and **the process identifier** of child is returned to the parent process.
  - **On success, both processes continue execution at the instruction after the fork call.**
  - On failure, **-1** is returned to the parent process.
- Execution options
  - **Parent and children execute concurrently**



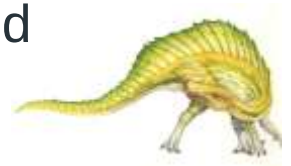
Process Creation

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

int main(){
    pid_t pid;
    pid = fork();

    if(pid==0){
        //Child process
        printf("As a Test in ChildProcess: pid=%d\n", pid);
        printf("Current process is CHILD process with pid=%d, my PARENT process's pid is %d!\n",
        getpid(),getppid());
    }else{
        //Parent process
        printf("As a Test in ParentProcess: pid=%d\n", pid);
        printf("current process is PARENT process with pid=%d, my CHILD process's pid is %d!\n",
        getpid(),pid);
    }
    return 0;
}
```

```
zaobohe@ubuntu:/mnt/hgfs/Desktop$ gcc fork1.c -o fork1
zaobohe@ubuntu:/mnt/hgfs/Desktop$ ./fork1
As a Test in ParentProcess: pid=23894
current process is PARENT process with pid=23893, my CHILD process's pid is 23894!
As a Test in ChildProcess: pid=0
Current process is CHILD process with pid=23894, my PARENT process's pid is 23893!
```

Explanation: All the statements after the **fork()** are executed twice:

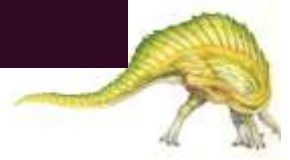
1.Once by the **parent process.**



Process Creation

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

int main(void)
{
    printf("Hello \n");
    fork();
    printf("bye\n");
    return 0;
}
```

```
zaobohe@ubuntu:/mnt/hgfs/Desktop$ gcc fork2.c -o fork2
zaobohe@ubuntu:/mnt/hgfs/Desktop$ ./fork2
Hello
bye
bye
```



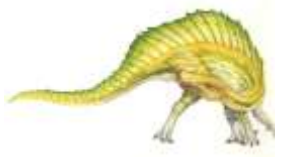
Exercises: fork()

Predict the Output of the following program:

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output:

```
Hello world!
Hello world!
```



Exercises: fork()

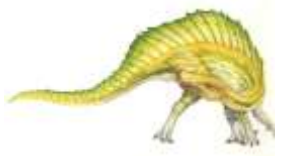
Calculate number of times hello is printed:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Output:

```
hello
hello
hello
hello
hello
hello
hello
hello
```

The number of times 'hello' is printed is equal to number of process created. Total Number of Processes = 2^n, where n is number of fork system calls. So here n = 3, $2^3 = 8$



Exercises: fork()

Calculate number of times hello is printed:

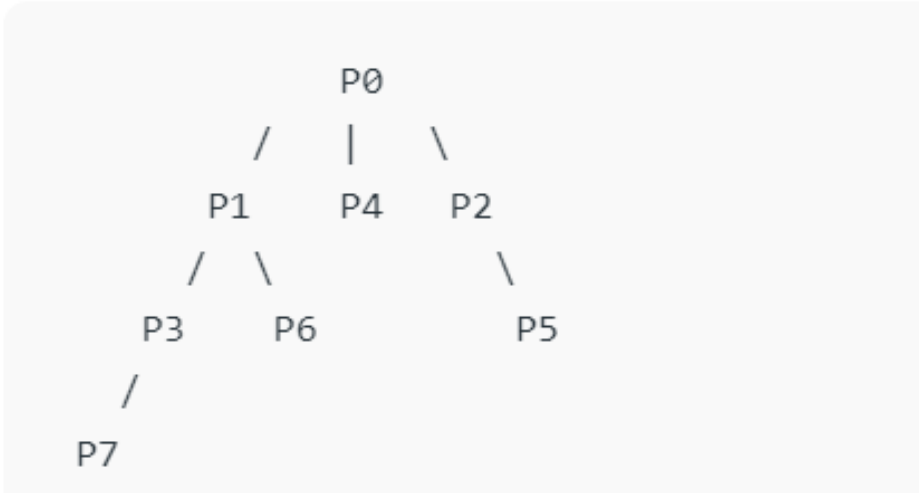
```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

The main process: P0

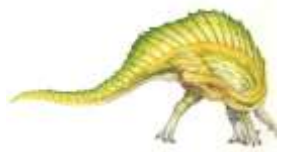
Processes created by the 1st fork: P1

Processes created by the 2nd fork: P2, P3

Processes created by the 3rd fork: P4, P5, P6, P7



The number of times ‘hello’ is printed is equal to number of process created. Total Number of Processes = 2^n, where n is number of fork system calls. So here n = 3, $2^3 = 8$



Exercises: fork()

Predict the Output of the following p

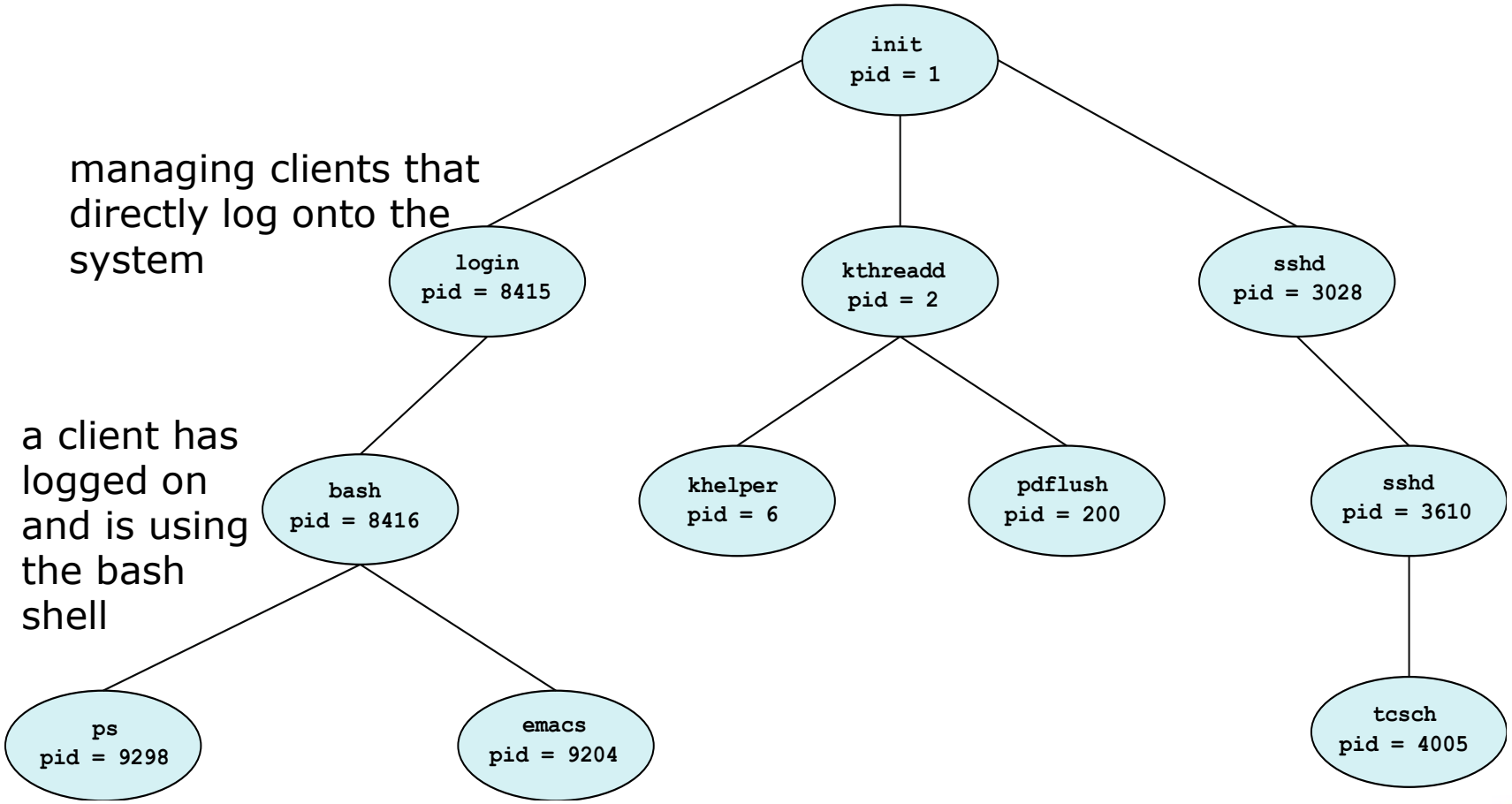
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

Output:

```
1.
Hello from Child!
Hello from Parent!
     (or)
2.
Hello from Parent!
Hello from Child!
```

- In the above code, a child process is created. fork() returns 0 in the child process and positive integer in the parent process.
- Here, two outputs are possible because the parent process and child process are running concurrently.
- So we don't know whether the OS will first give control to the parent process or the child process.

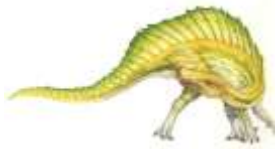
A Tree of Processes in Linux



managing clients that directly log onto the system

a client has logged on and is using the bash shell

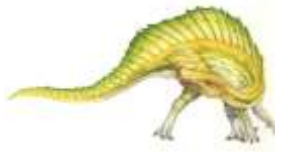
has created two processes



Process Creation (Cont.)

- Resource sharing options
  - A child process may obtain its resources directly from the OS
  - A child process may be constrained to a subset of the resources of the parent process
    - Prevent any process from overloading the system by creating too many child processes

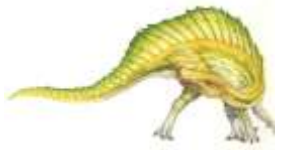
- There are also two address-space possibilities for the new process:
  - The child process is a duplicate of the parent process (it has the same program and data as the parent)
  - Child process has a program loaded into it



Process Termination

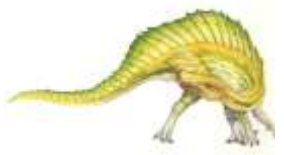
- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system

- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates



Process Termination

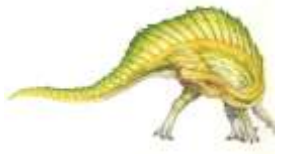
- When a process creates a child process, sometimes it becomes necessary that the parent process should execute only after the child has finished.
- **wait() system call does exactly this. It makes the parent process wait for child process to finish and then the parent continues its working from the statement after the wait().**
- On success, wait returns the PID of the terminated child process while on failure it returns -1.



Process Termination

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

int main()
{
    pid_t cpid;
    if (fork()== 0)
        exit(0);
    else
        cpid = wait(NULL);

    printf("Parent pid = %d\n", getpid());
    printf("Child pid = %d\n", cpid);
    return 0;
}
```

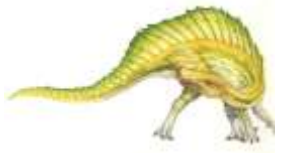
```
zaobohe@ubuntu:/mnt/hgfs/Desktop$ gcc fork3.c -o fork3
zaobohe@ubuntu:/mnt/hgfs/Desktop$ ./fork3
Parent pid = 24082
Child pid = 24083
zaobohe@ubuntu:/mnt/hgfs/Desktop$
```



Process Termination

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

int main()
{
    if (fork()== 0)
        printf("HC: hello from child\n");
    else {
        printf("HP: hello from parent\n");
        wait(NULL);
        printf("CT: child has terminated\n");
    }

    printf("Bye\n");
    return 0;
}
```

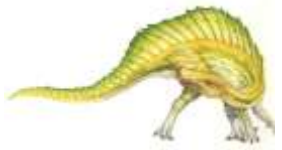
```
zaobohe@ubuntu:/mnt/hgfs/Desktop$ gcc fork4.c -o fork4
zaobohe@ubuntu:/mnt/hgfs/Desktop$ ./fork4
HP: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye
```



Process Termination

- If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of child process by using the **wait()** system call. The call returns status information and the pid of the terminated process

  pid = wait (&status);
- If no parent waiting (did not invoke **wait ()** yet), terminated process is a **zombie**
- If parent terminated without invoking **wait()**, process is an **orphan**

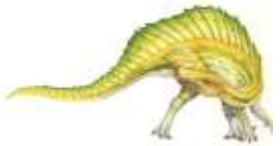


Review

2. In operating system, each process has its own _____
a) address space and global variables
b) open files
c) pending alarms, signals and signal handlers
d) all of the mentioned

5. What is the ready state of a process?
a) when process is scheduled to run after some execution
b) when process is unable to run until some task has been completed
c) when process is using the CPU
d) none of the mentioned

8. A process stack does not contain _____
a) Function parameters
b) Local variables
c) Return addresses
d) PID of child process

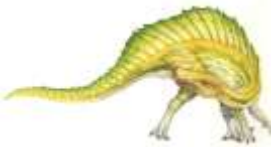


Review

2. The number of processes completed per unit time is known as _____
a) Output
b) Throughput
c) Efficiency
d) Capacity

2. When the process issues an I/O request _____
a) It is placed in an I/O queue
b) It is placed in a waiting queue
c) It is placed in the ready queue
d) It is placed in the Job queue

3. What will happen when a process terminates?
a) It is removed from all queues
b) It is removed from all, but the job queue
c) Its process control block is de-allocated
d) Its process control block is never de-allocated



Review

4. What is a long-term scheduler?
a) It selects processes which have to be brought into the ready queue
b) It selects processes which have to be executed next and allocates CPU
c) It selects processes which heave to remove from memory by swapping
d) None of the mentioned

5. If all processes I/O bound, the ready queue will almost always be ____ and the Short term Scheduler will have a ____ to do.
a) full, little
b) full, lot
c) empty, little
d) empty, lot

6. What is a medium-term scheduler?
a) It selects which process has to be brought into the ready queue
b) It selects which process has to be executed next and allocates CPU
c) It selects which process to remove from memory by swapping
d) None of the mentioned

Review

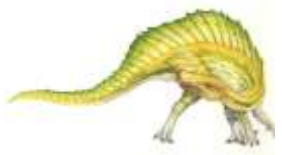
8. The primary distinction between the short term scheduler and the long term scheduler is
a) The length of their queues
b) The type of processes they schedule
c) The frequency of their execution
d) None of the mentioned

10. In a time-sharing operating system, when the time slot given to a process is completed, the process goes from the running state to the
a) Blocked state
b) Ready state
c) Suspended state
d) Terminated state

12. Suppose that a process is in "Blocked" state waiting for some I/O service. When the service is completed, it goes to the
a) Running state
b) Ready state
c) Suspended state
d) Terminated state

Interprocess Communication

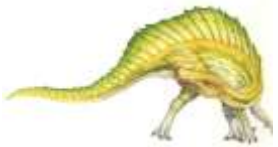
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **inter-process communication** (**IPC**)
  - To exchange data and information
- Two fundamental models of IPC
  - **Shared memory**
  - **Message passing**



Communications Models

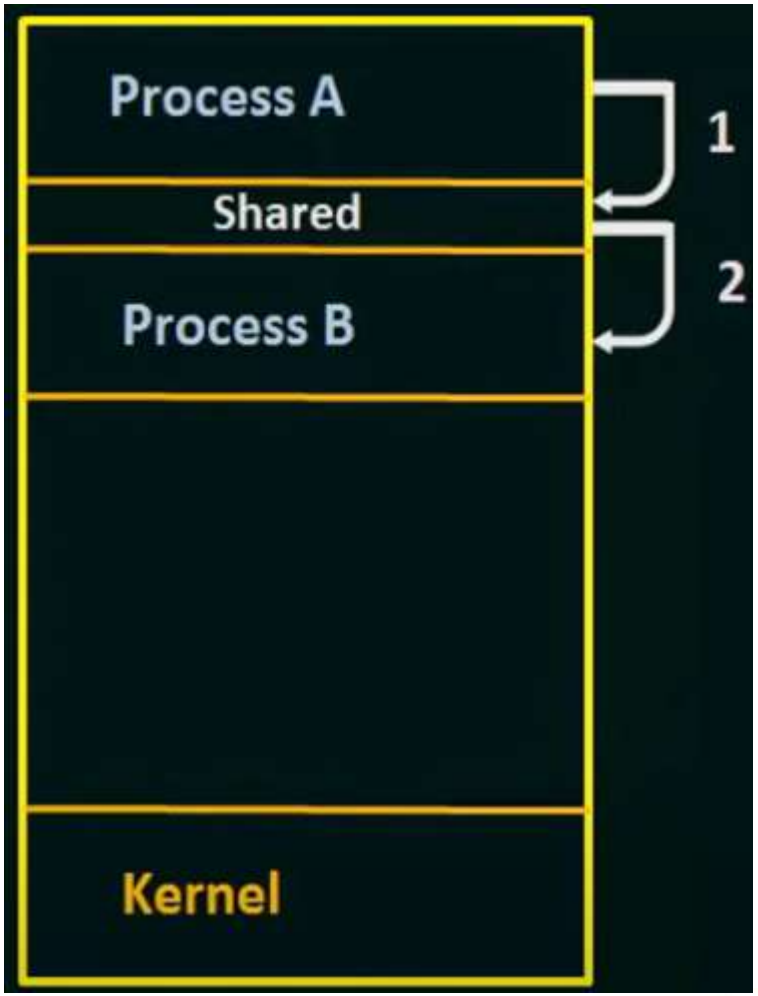
Shared memory: reading or writing something from/to the document on the table

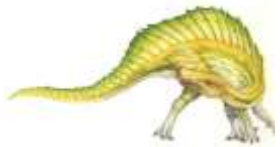
Message passing: reading or writing something through individual message, not a public document on the table



Shared Memory

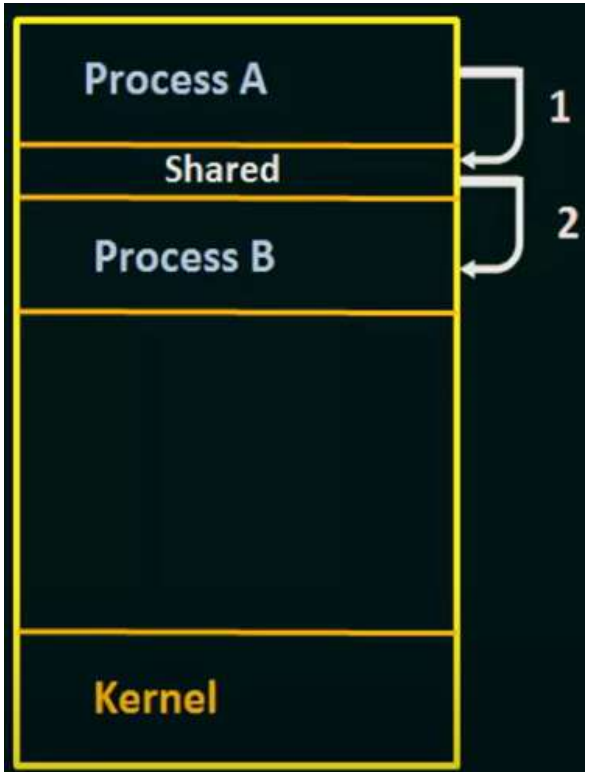
- **Shared memory**
  - A region of memory shared by cooperating processes is established
  - Processes can then exchange information by reading and writing data to the shared region



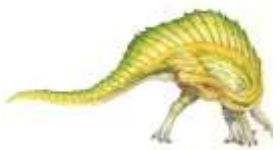


Shared Memory

Where does this shared memory region reside?



- The processes initially set up a region of their virtual memory to use for the IPC
- Once the region is established within the process, the process issues a system call to request that the kernel make the region shared.
- After the initial system call to set up the shared memory, the processes can read from and write to the region just as it would access non-shared data on its own heap.
- This data then appears within the context of the other process automatically. There is no explicit system call required to read the new data.



Shared Memory

Let's say P1 needs to send some data to P2. The following steps describe how this will happen.

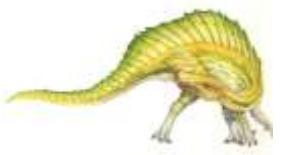
Step 1: First P1 establish a shared memory region in P1's address space.

Step 2: Now P2 should attach that shared memory region to its address space. Then P1 writes data that needs to share with P2 in the shared memory region.

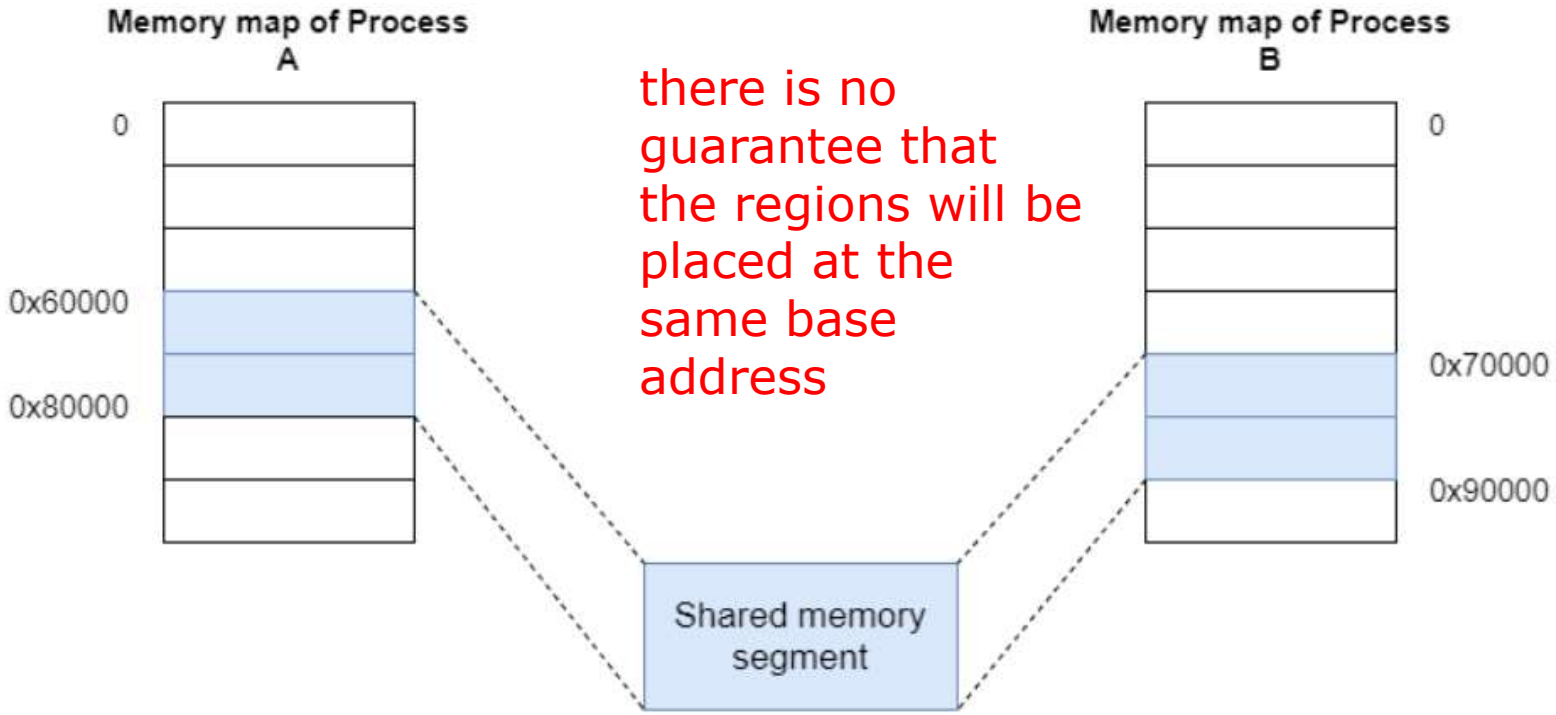
Step 3: P2 reads data from the shared memory region.

Operating system has to remove that restriction

However, the operating system does not allow to access the address space of a process by another process

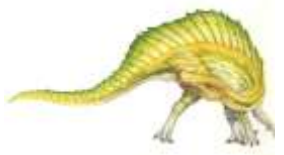


More about Shared Memory



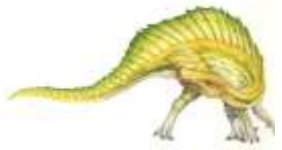
there is no guarantee that the regions will be placed at the same base address

So storing the number 1 in the first process's address 0x60000 means the second process has the value of 1 at 0x70000. The two (different) addresses refer to the exact same location.



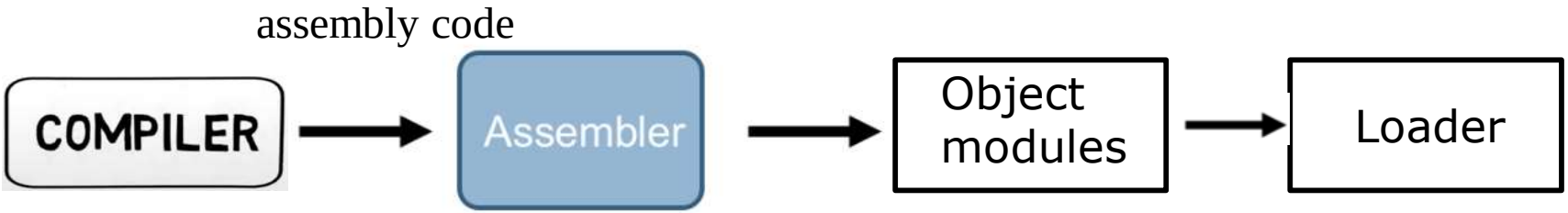
Shared Memory

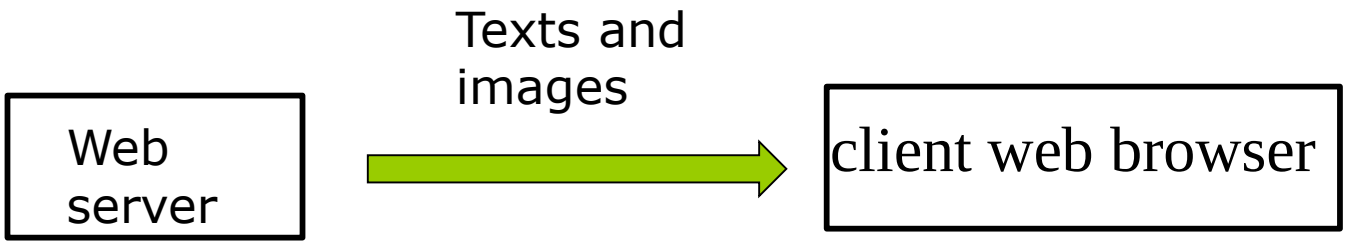
- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

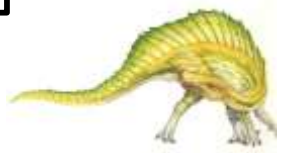


Bounded-Buffer – Shared-Memory Solution

- **Cooperating processes**
- Paradigm for cooperating processes, ***producer*** process produces information that is consumed by a ***consumer*** process



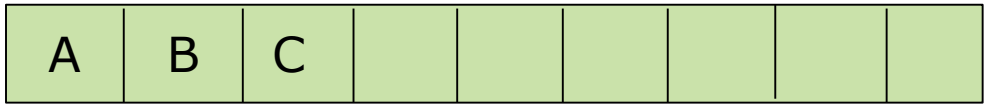




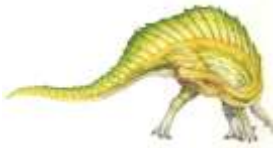
Bounded-Buffer – Shared-Memory Solution

- One solution to the producer-consumer problem uses **shared memory**
  - A buffer of items that can be filled by the producer and emptied by the consumer
  - Reside in memory, and shared by producer and consumer process
  - Producer and consumer must be **synchronized, why?**

D



TRY TO RESD???



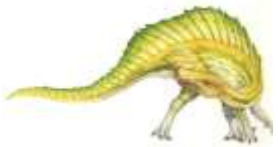
Bounded-Buffer – Shared-Memory Solution

- One solution to the producer-consumer problem uses **shared memory**
  - A buffer of items that can be filled by the producer and emptied by the consumer
  - Reside in memory, and shared by producer and consumer process
  - Producer and consumer must be **synchronized, why?**

J

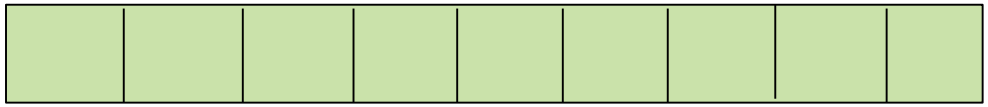
A	B	C	D	E	F	G	H	I

Synchronization Problem: case 1: the producer tries to produce (write) an item when the buffer is full

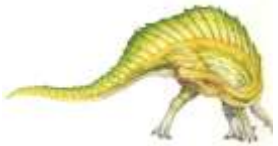


Bounded-Buffer – Shared-Memory Solution

- One solution to the producer-consumer problem uses **shared memory**
  - A buffer of items that can be filled by the producer and emptied by the consumer
  - Reside in memory, and shared by producer and consumer process
  - Producer and consumer must be **synchronized, why?**



Synchronization Problem: case 2: the consumer tries to consume an item that has not yet been produced (read a data from an empty buffer)

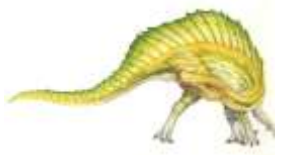


Bounded-Buffer – Shared-Memory Solution

- Two types of buffers can be used
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Can unbounded-buffer produce an unsynchronized problem?

Yes! the processes are also responsible for ensuring that they are not writing to the same location simultaneously

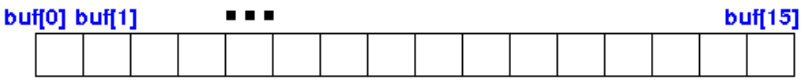


Bounded-Buffer – Shared-Memory Solution

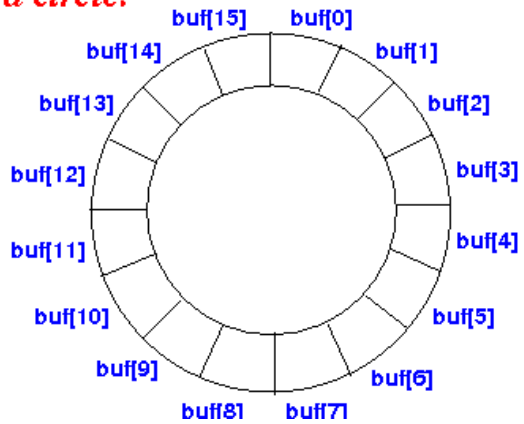
- Shared data

```
#define BUFFER_SIZE 15
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Array:



Pretend array is a circle:



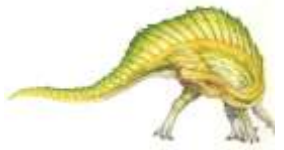
- The buffer is empty when in==out
- The buffer is full when ((in + 1) % BUFFER_SIZE) = = out

In------points to the next free position in the buffer;
Out------points to the first full position in the buffer

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

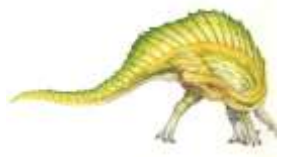
The producer process has a local variable **next-produced** in which the new item to be produced is stored



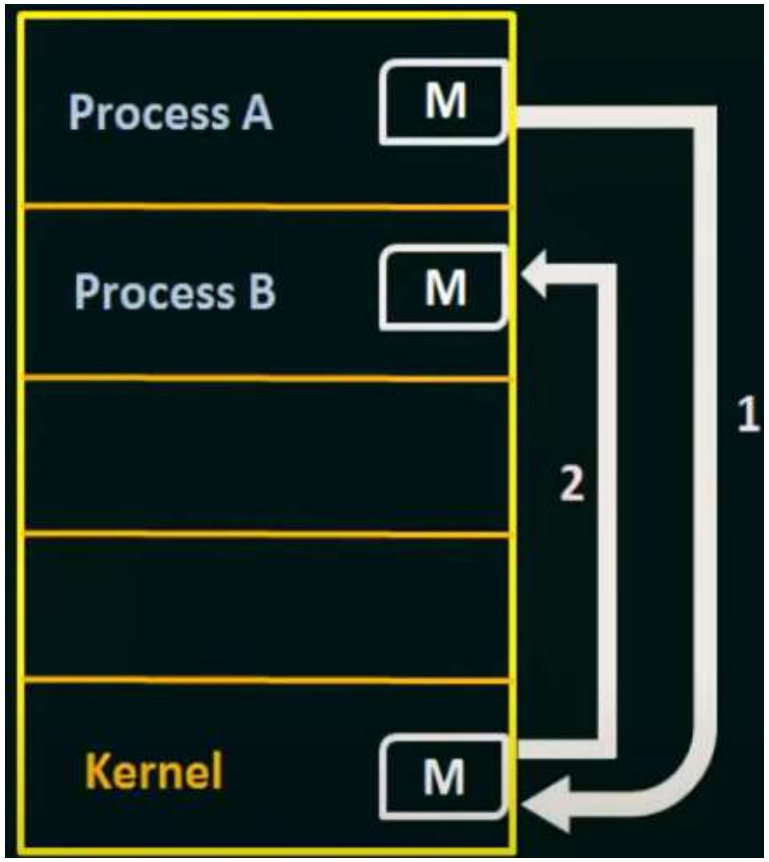
Bounded Buffer – Consumer

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];

        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */

}
```

- What about multiple producers and multiple consumers---Synchronizing not only the buffer but also the update of the *in* and *out* variables

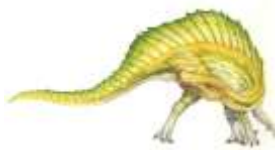


Message Passing

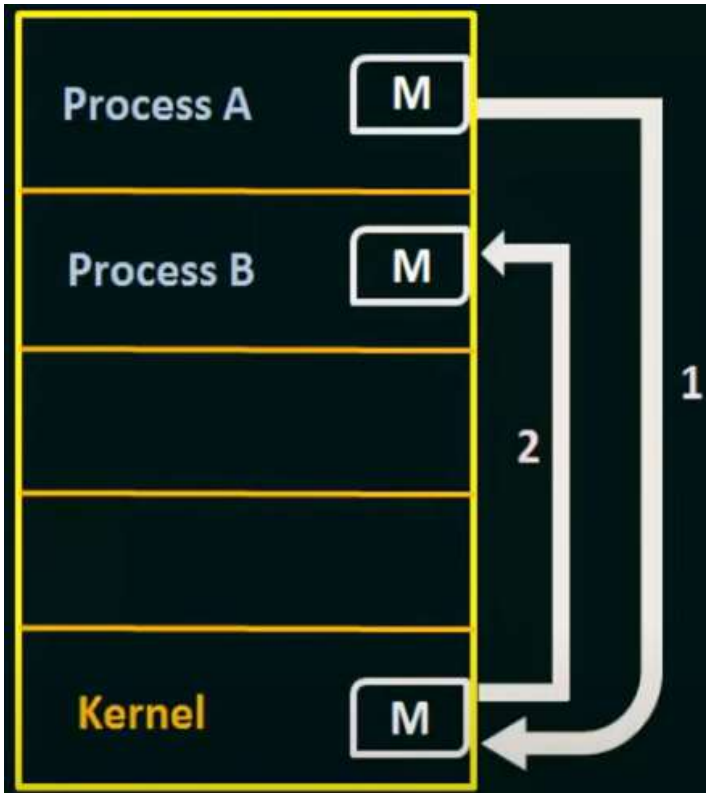


Communication takes place by means of messages exchanged between the cooperating processes

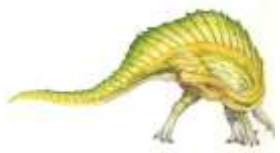
Particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network



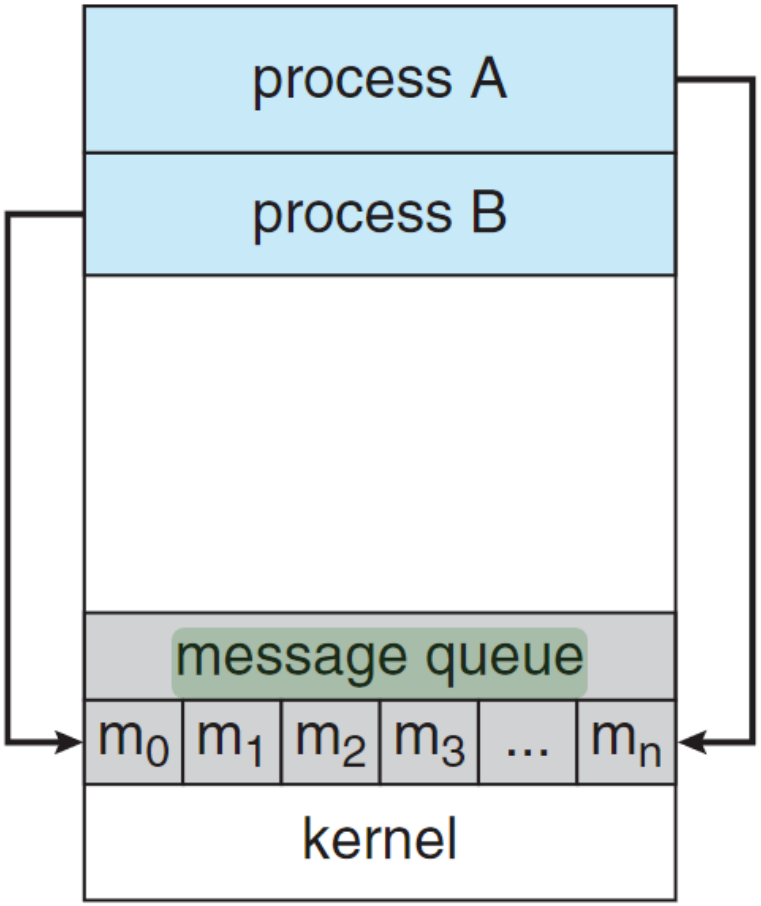
Message Passing



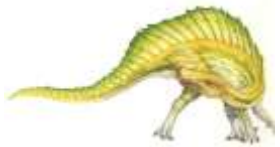
- The user-mode process will **copy** the data into a buffer, then **issue a system call** to request the data be transferred.
- Once the kernel is invoked, it will **copy** the transferred data first into **its own memory**.
- The target process will also **issue a system call** to retrieve the data.



Message passing

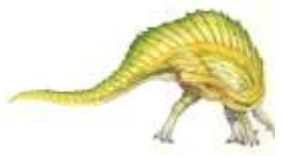


- In message passing, every piece of data exchanged requires **two system calls**: one to read and one to write.
- In addition, the transferred data must **be copied twice**: once into the kernel memory and once into the receiving process.



Interprocess Communication – Message Passing

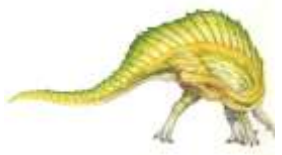
- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable



Message Passing (Cont.)

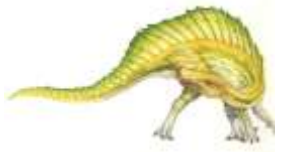
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering



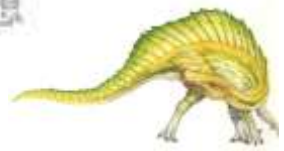
Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional



```
#define N 100                                   /* 缓冲区中的槽数目 */

void producer(void)
{
     int item;
     message m;                                 /* 消息缓冲区 */

     while (TRUE) {
          item = produce_item( );               /* 产生放入缓冲区的一些数据 */
          receive(consumer, &m);                /* 等待消费者发送空缓冲区 */
          build_message(&m, item);              /* 建立一个待发送的消息 */
          send(consumer, &m);                   /* 发送数据项给消费者 */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* 发送N个空缓冲区 */
     while (TRUE) {
          receive(producer, &m);                /* 接收包含数据项的消息 */
          item = extract_item(&m);              /* 将数据项从消息中提取出来 */
          send(producer, &m);                   /* 将空缓冲区发送回生产者 */
          consume_item(item);                   /* 处理数据项 */
     }
}
```


Direct Communication– Shared-Memory Solution

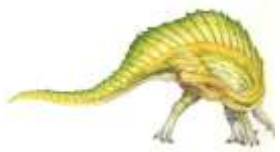


Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Primitives are defined as:

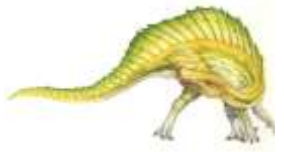
  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A



Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



Indirect Communication – Shared-Memory Solution

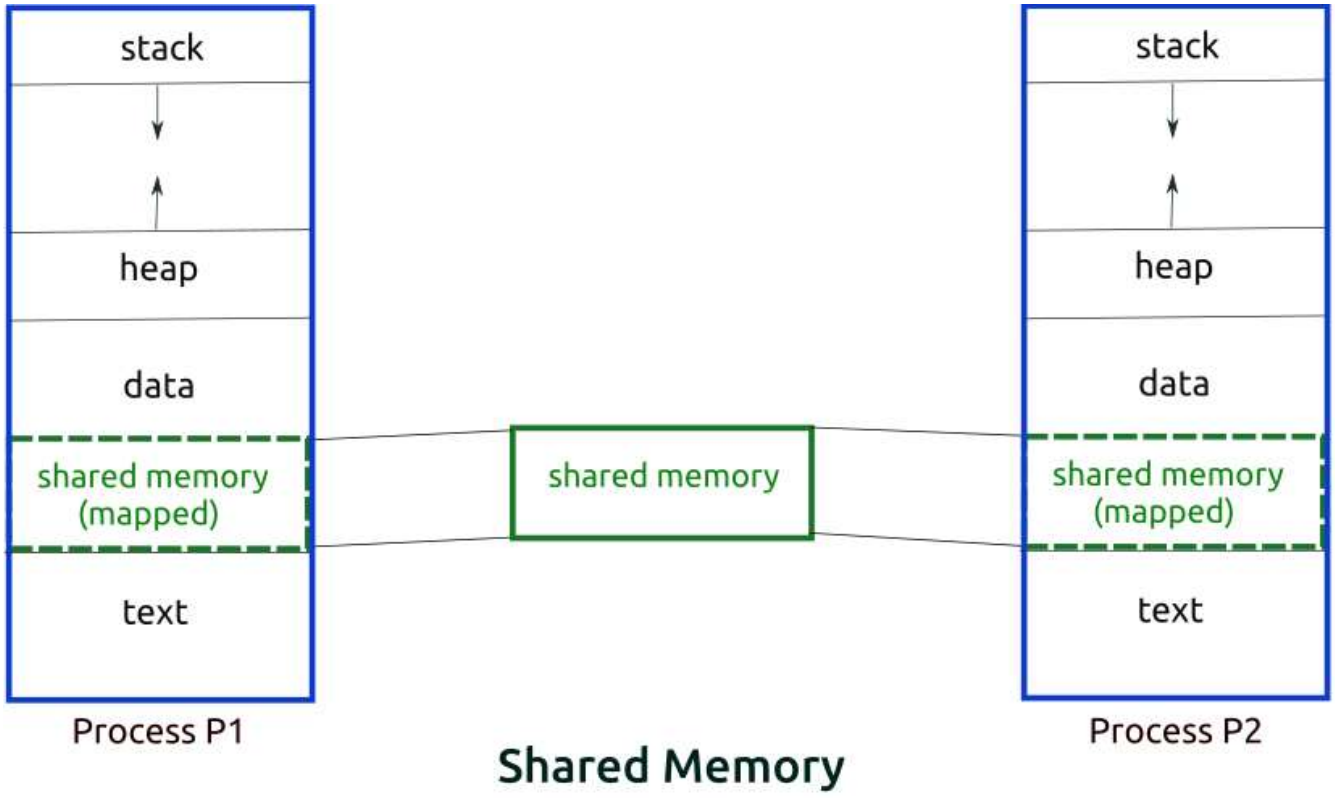
```
typedef message {
    ...
}
const capacity = ... ;
message dummy = {};

int main()
{
    int i;

    create_mailbox( mayconsume );
    create_mailbox( mayproduce );
    for ( i = 0; i < capacity; i++ )
      send( mayproduce, dummy );
    producer();
    consumer();
}
```

```
producer()
{
  message pmsg;

  while ( true ) {
    receive( mayproduce, pmsg );

    < produce >

    send( mayconsume, pmsg );
  }
}
```

```
consumer()
{
  message cmsg;

  while ( true ) {
    receive( mayconsume, cmsg );

    < consume >

    send( mayproduce, cmsg );
  }
}
```

Note: In this example, both send() and receive() are blocking operations.

Message passing v.s. Shared memory

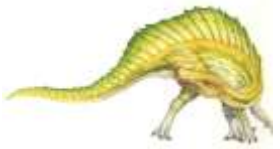


The shared memory techniques only require a **one-time** performance penalty during the set-up phase. Once the memory has been shared, there is no additional penalty, regardless of the amount of data transferred.

Message passing v.s. Shared memory

- Overall, if the two processes will be exchanging **a lot of data** back and forth repeatedly, **shared memory performs very well**. While the work to set up the shared memory is expensive.

- However, if processes only need to exchange a **single message of a few bytes**, shared memory will perform very poorly. Message passing techniques impose significantly **smaller overhead** to set up a one-time data exchange.



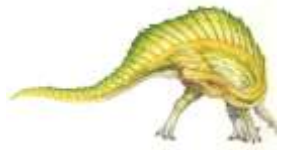
Message passing v.s. Shared memory

- Advantage of message passing
  - Useful for exchanging smaller amount of data, why? no conflicts need be avoided
  - Easier to implement in distributed system, why?

- Advantages of shared memory

Here needs kernel intervention for system call

  - Can be faster than message passing, since once the shared memory is established, all accesses are treated as routine memory accesses
  - In contrast, message passing are typically implemented using system calls. Time consuming! Why?
  - Disadvantage: synchronization
  - Shared memory suffers from *cache coherency issues*



End of Chapter 3