Bootstrapping

Introduction

Bootstrapping

Introduction:

Computers execute programs stored in main memory, and initially the operating system is on the hard disk.

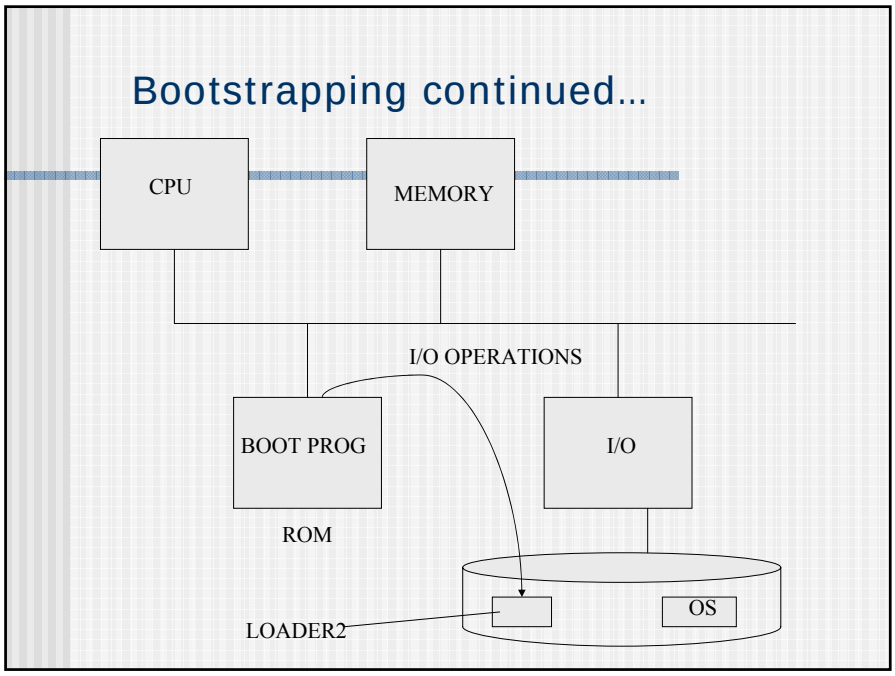
When the computer is turned on it does not have an operating system loaded in memory and the hardware alone cannot do the operations of an OS. To solve this paradox a special program called bootstrap loader is created.

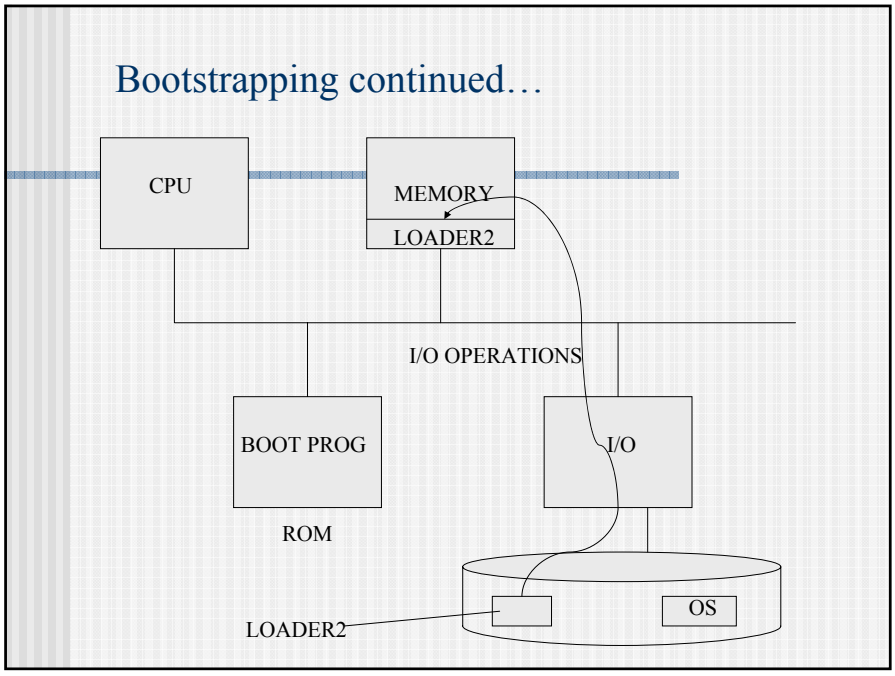
Bootstrapping continued…

- This program does not have the full functionality of an operating system, but it is capable of loading into memory a more elaborated software(i.e. loader2) which in its turn will load the operating system.
- Once the OS has been loaded the loader transfers the control of the system to the Operaing system.

Bootstrapping continued…

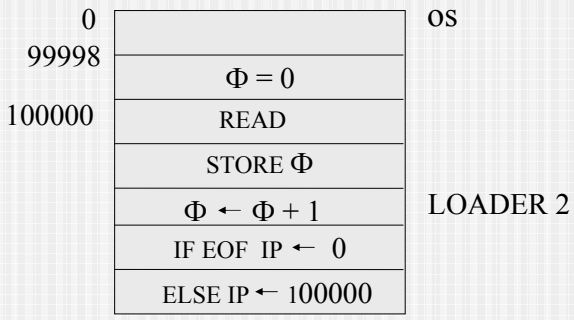
- Early programmable computers had toggle switches on the front panel to allow the operator to place the bootloader into the program store before starting the CPU.
- In modern computers the bootstrapping process begins with the CPU executing software contained in ROM at a predefined address whose elementary functionality is to search for devices eligible to participate in booting, and load a small program from a special section of a device.


Bootstrapping continued...
CPU
MEMORY
I/O OPERATIONS
BOOT PROG
I/O
ROM
OS
LOADER2

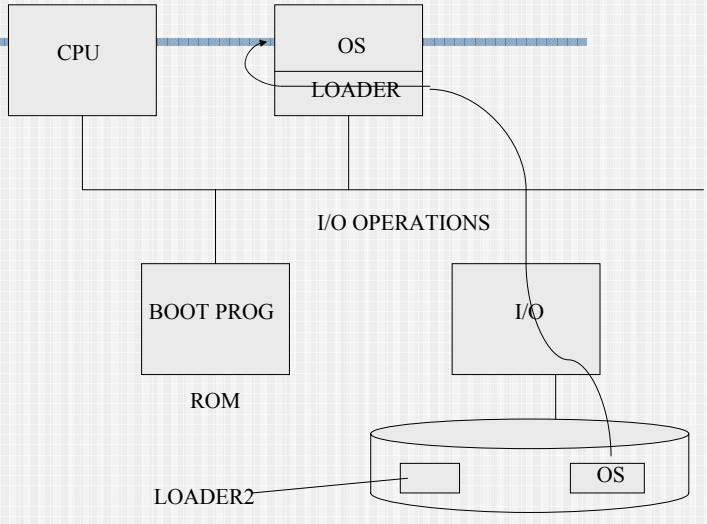

Bootstrapping continued…
CPU
MEMORY
LOADER2
I/O OPERATIONS
BOOT PROG
I/O
ROM
OS
LOADER2

Bootstrapping continued…

- In earlier computers data had to be hand loaded as specified before, but nowadays a small piece of software called loader helps us to avoid the manual loading.



Bootstrapping continued…



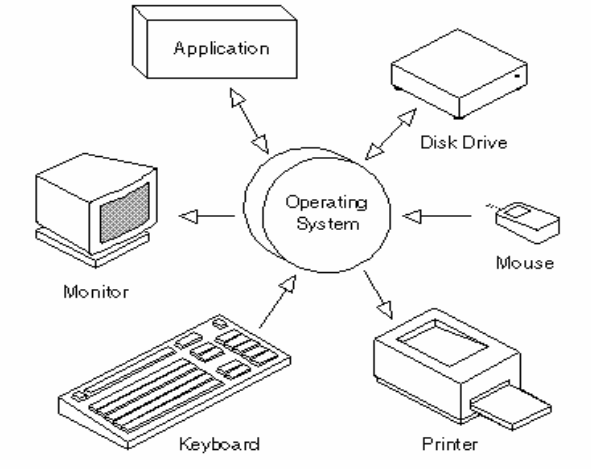
Bootstrapping continued…

- The above diagram can be explained in the following steps.
  1. Check hardware
  2. Initiate I/O to load the loader2 program into memory
  3. loader2 loads the OS and passes control to it

Conclusion

- We have seen that once the OS has control over the system , it can create an environment for programs to run.
- The operating system will load device drivers and other programs that are needed for the normal operation of the computer system.

Operating system



Process concept

Concept of Multiprogramming

- When there is a single program running in the CPU, it leads to the degradation of the CPU utilization.
- Example: When a running program initiates an I/O operation, the CPU remain idle until the I/O operation is completed.

Solution to this problem is provided by *Multiprogramming*.

Multiprogramming Continued..

Definition:

A mode of operation that provides for the interleaved execution of two or more programs by a single processor.

Multiprogramming Continued..

Improving CPU utilization

By allowing several programs to reside in main memory at the "**same time**" the CPU might be shared,

such that when one program initiates an I/O operation,

another program can be assigned to the CPU, thus the improvement of the CPU utilization.

Multiprogramming Continued..

Implementation

The idea of ***process*** need to be introduced in order to understand operating system in a multiprogramming computer system.

What is a process?

Process

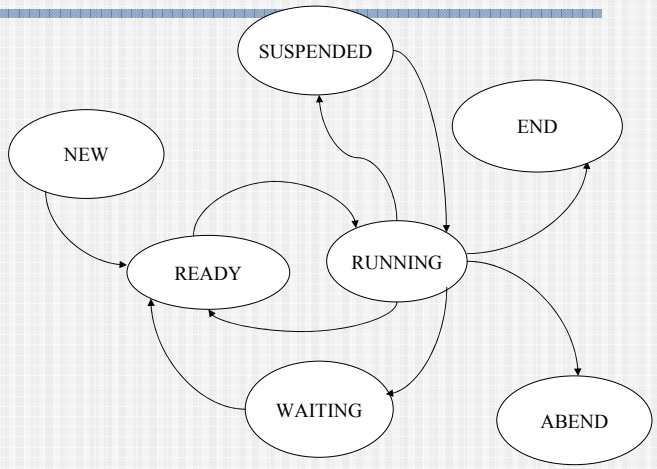
- Definition:
  - A program in execution
  - An asynchronous activity
  - The “locus of control” of a procedure in execution
  - It is manifested by the existence of a process control block in the operating system.

Process States

A state of a process describes the activity that process is doing at a certain moment in time.

New : A newly created process, not in the ready queue.
Ready : It can use the CPU , if available.
Running : If it is currently in the CPU.
Waiting : Waiting for some event ex: I/O
Abend : Stops executing due to an error.
End : Finished executing properly.

States of processes

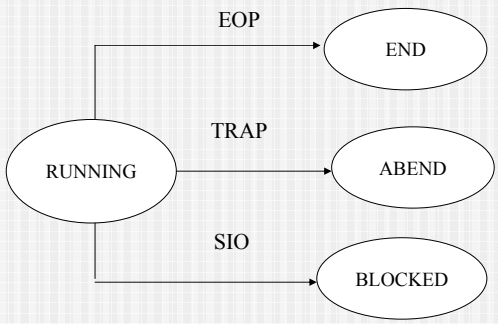


Causes of state change

When a process executes, it changes states and interrupts cause process to change states.

Current State	New state	Interrupt
Running (End of Program)	End	EOP
Running (Abnormal end)	ABEND	Trap
Running (Start I/O)	Blocked for I/O	System Call (SIO)

Depiction of state change



Process Continued…

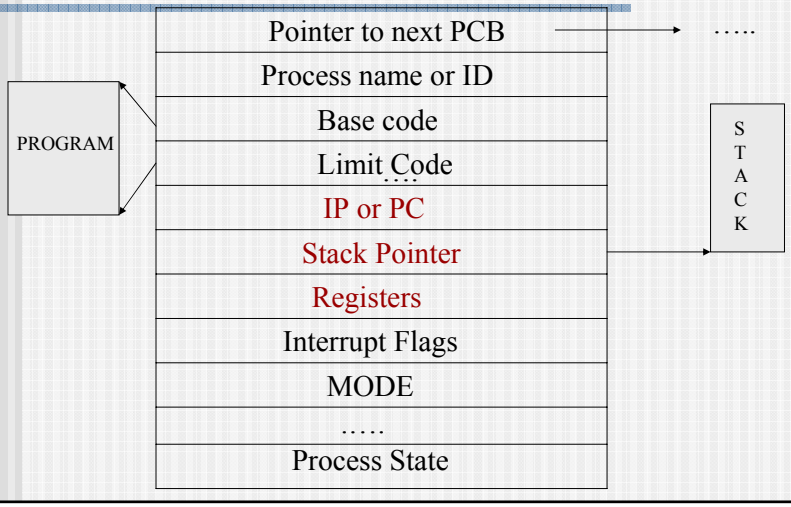
- The activity of a process is controlled by a data structure called **P**rocess **C**ontrol **B**lock(PCB).
- A PCB is created every time a program is loaded to be executed.
- So, Process is defined by PCB-Program couple.

Structure of PCB

PCB contains information about processes

- the current state of a process
- Unique identification of process
- Process priority
- CPU registers
- Instruction Pointer (IP), also known as PC
- Base and limit registers
- Time limits and I/O status information

Structure of PCB Contd…



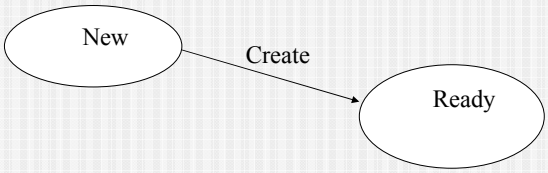
Process Continued…

We can now observe how each stage of a process takes place by the aid of state diagrams.

Process creation :

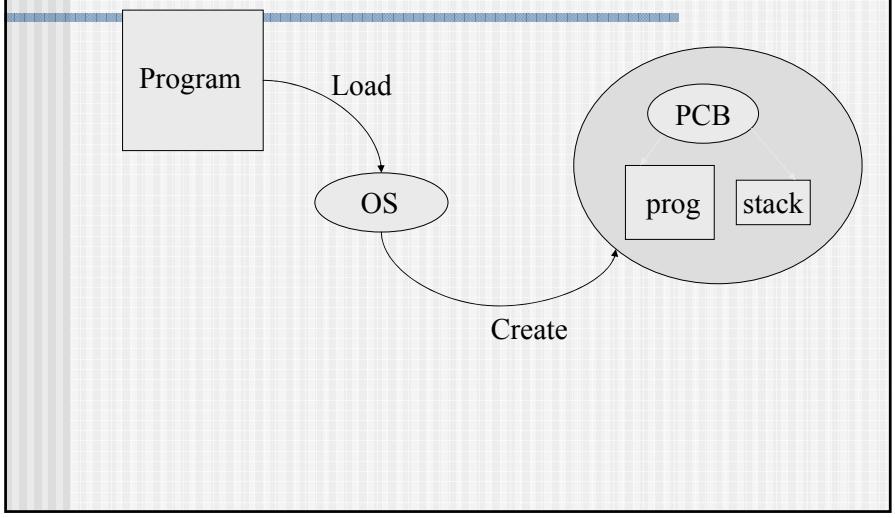
- An OS can create one or more processes, via a create-process system call.
- During the course of execution an user process may create processes as well. In this case, the creating process is called the parent and the created (new) process is named the child.

Process Creation

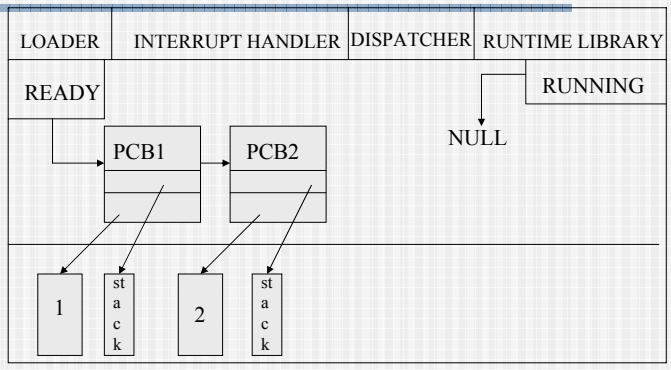


The process is created and then inserted at the back of the ready queue, it moves to the head of the queue according the CPU availability.

Process Creation Contd…

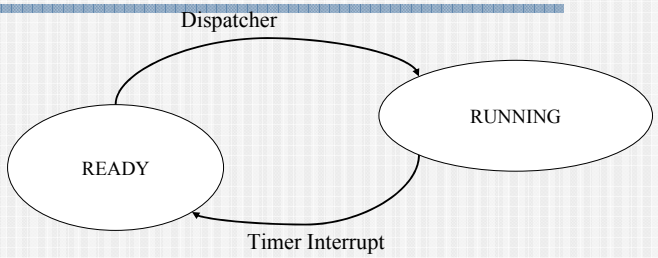


Memory snap shot of two processes in Ready State



The PCB is stored in the OS memory area in a linked list.

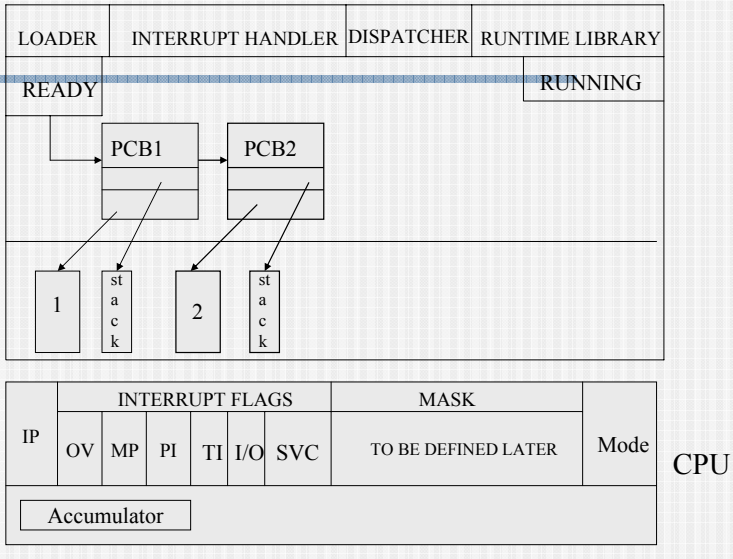
Ready to Running

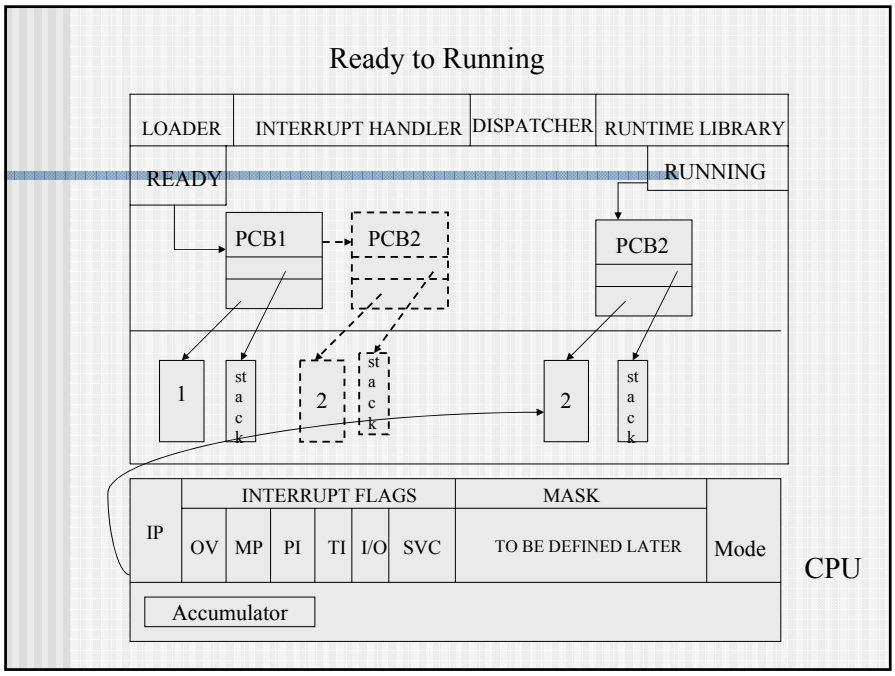


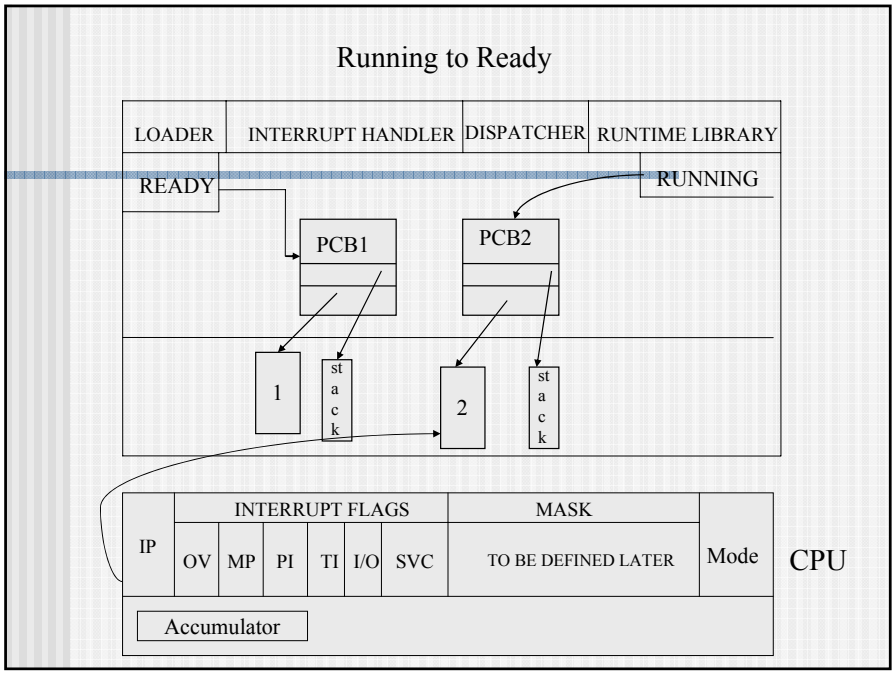
When a process reaches the head of the queue and the CPU is available, the process is dispatched which means that the CPU is assigned to the process. This cause a transition from the ready state to the running state.

When the time slice of the running process expires it goes back to the ready state.

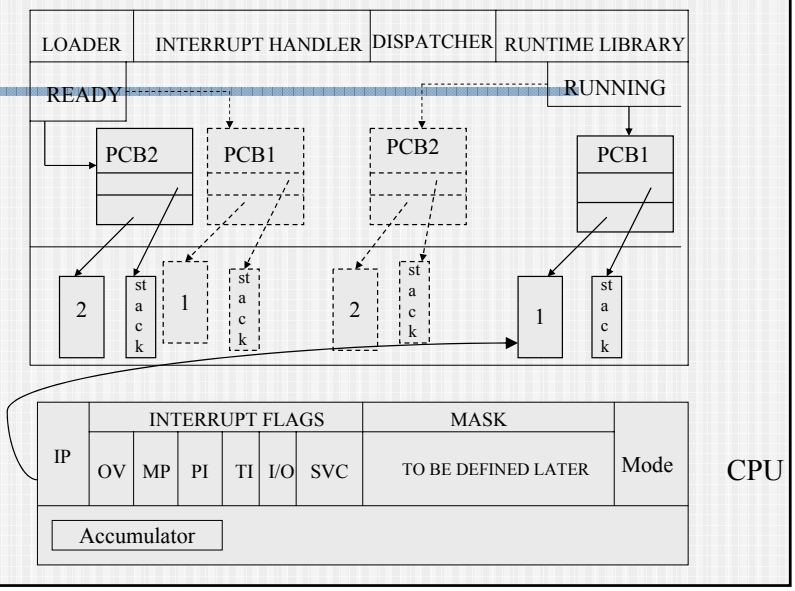
Ready to Running




Ready to Running
LOADER
INTERRUPT HANDLER
DISPATCHER
RUNTIME LIBRARY
READY
RUNNING
PCB1
PCB2
PCB2
1
stack
2
stack
2
stack
IP
INTERRUPT FLAGS
MASK
OV
MP
PI
TI
I/O
SVC
TO BE DEFINED LATER
Mode
Accumulator
CPU


Running to Ready
LOADER
INTERRUPT HANDLER
DISPATCHER
RUNTIME LIBRARY
READY
RUNNING
PCB1
PCB2
1
stack
2
stack
IP
INTERRUPT FLAGS
MASK
OV
MP
PI
TI
I/O
SVC
TO BE DEFINED LATER
Mode
Accumulator
CPU

Running to Ready

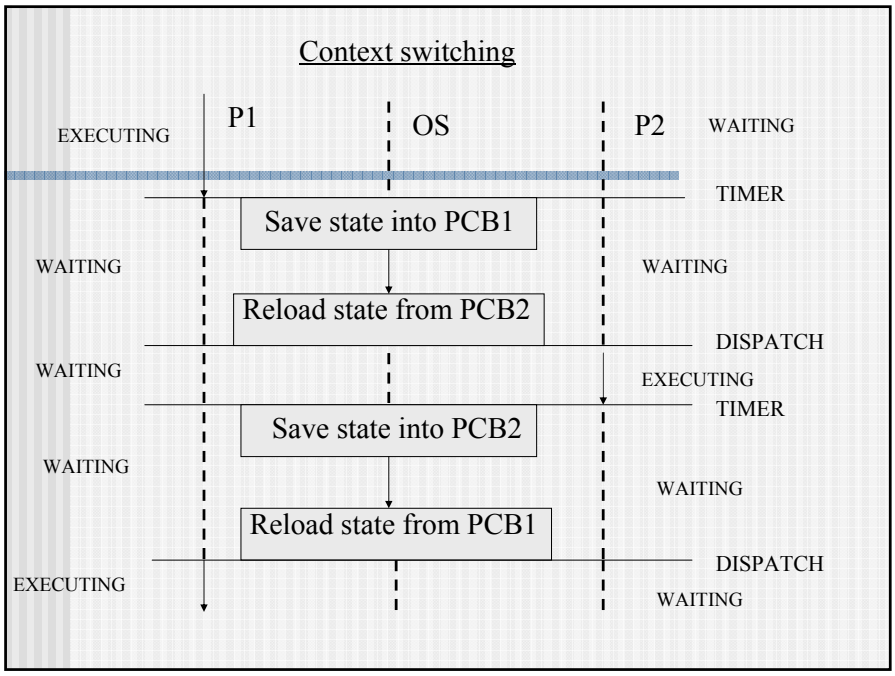


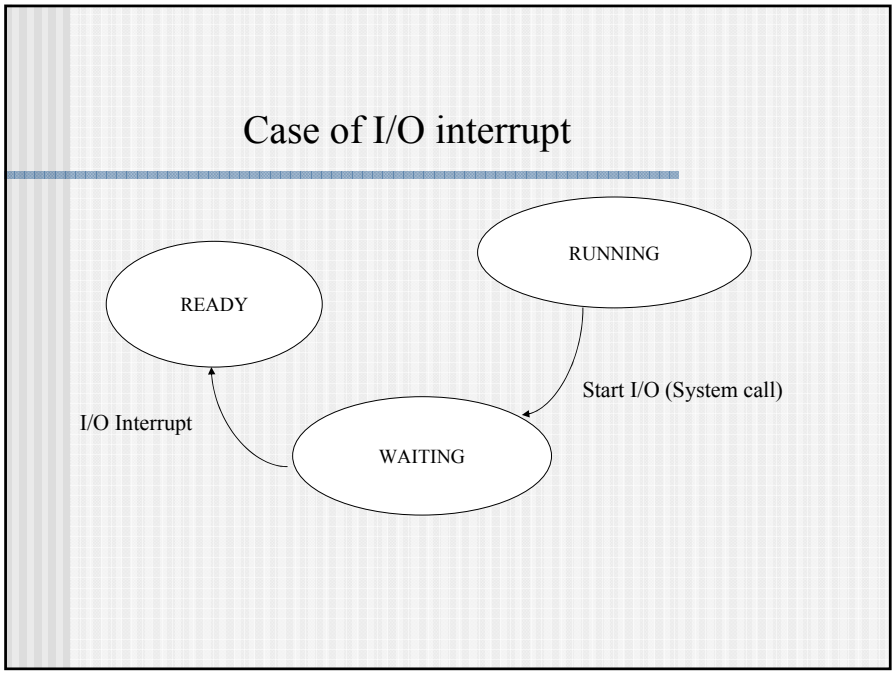
Process Continued…

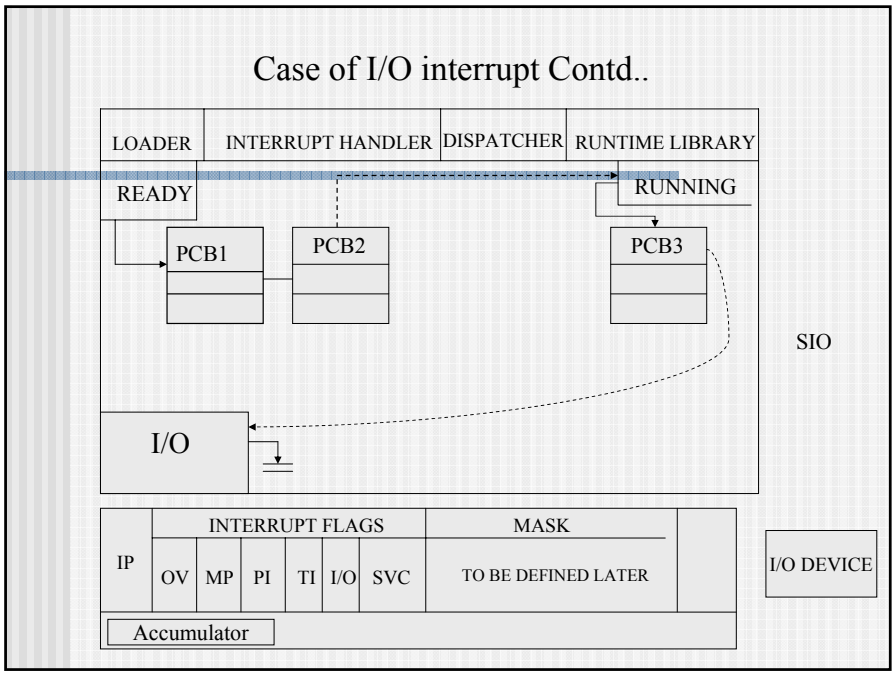
As the OS switches the allocation of CPU among processes it uses the PCB to store the CPU information or context, which represents the state of the process.

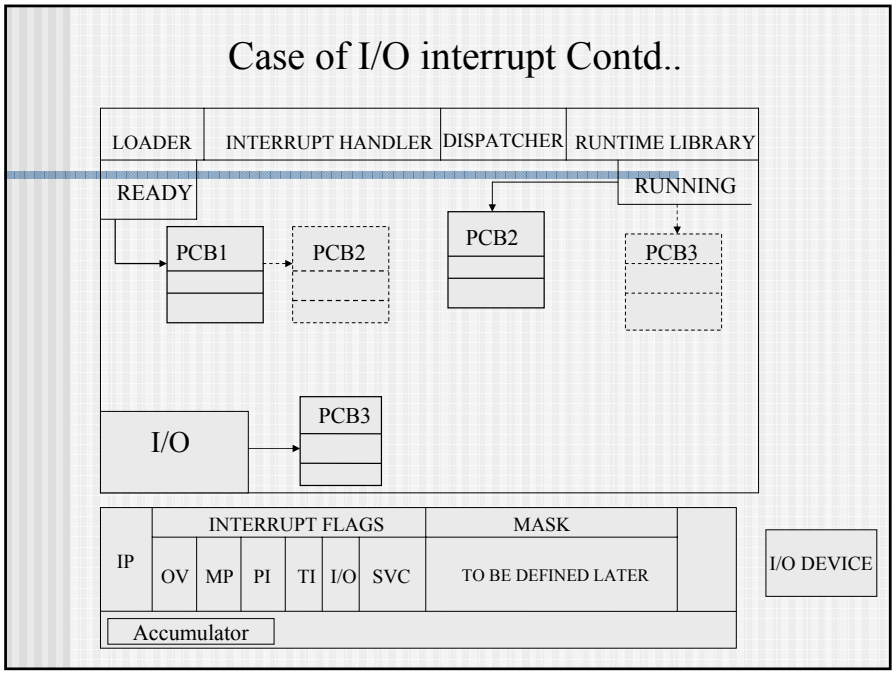
In the previous example we have seen how the OS performed a context switch between processes P2(from Running to Ready) and P1(from Ready to Running).

When a context switch occurs we need to save the state of the running process in its PCB and load the state of the new process in the CPU.

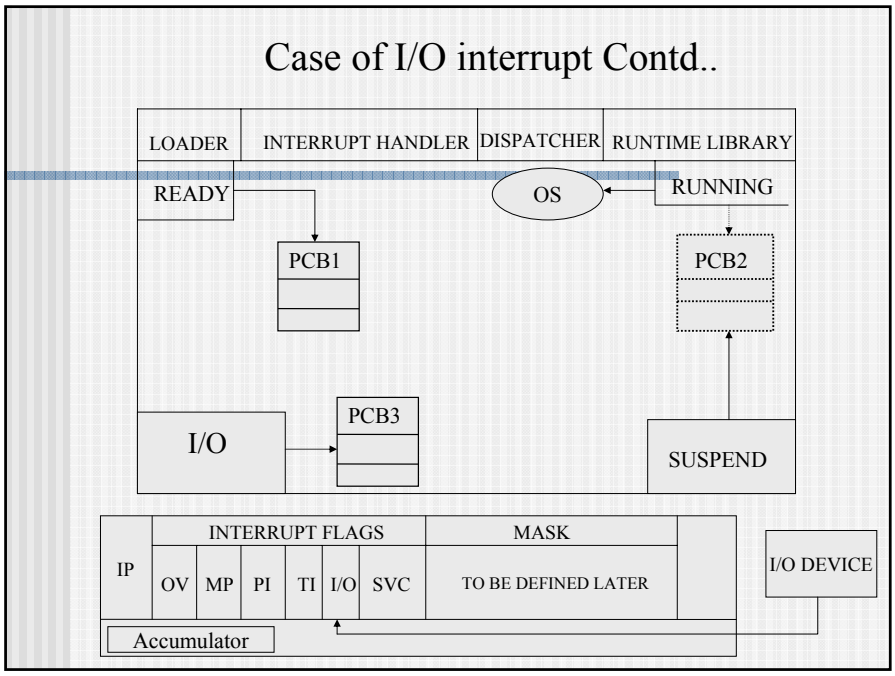

Context switching
EXECUTING
P1
OS
P2
WAITING
TIMER
Save state into PCB1
WAITING
WAITING
Reload state from PCB2
DISPATCH
WAITING
EXECUTING
TIMER
Save state into PCB2
WAITING
WAITING
Reload state from PCB1
DISPATCH
EXECUTING
WAITING


Case of I/O interrupt
RUNNING
READY
Start I/O (System call)
I/O Interrupt
WAITING

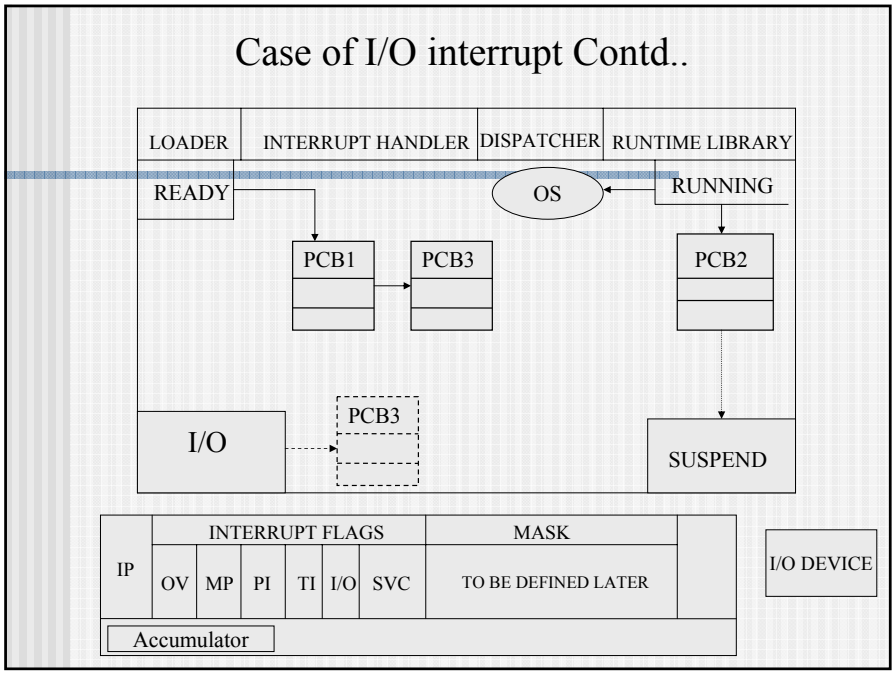

Case of I/O interrupt Contd..
LOADER
INTERRUPT HANDLER
DISPATCHER
RUNTIME LIBRARY
READY
RUNNING
PCB1
PCB2
PCB3
SIO
I/O
IP
INTERRUPT FLAGS
MASK
OV
MP
PI
TI
I/O
SVC
TO BE DEFINED LATER
I/O DEVICE
Accumulator

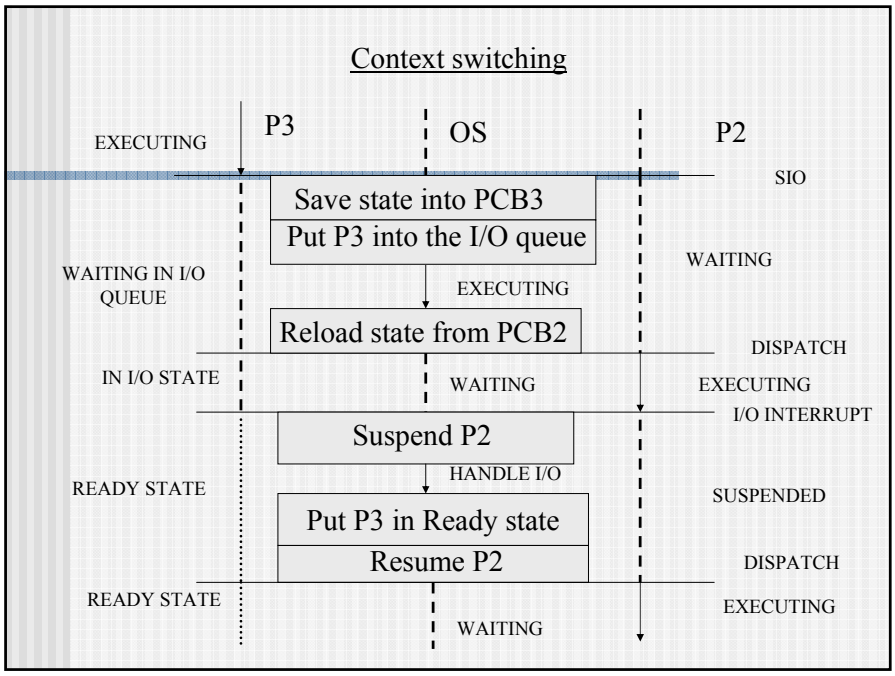

Case of I/O interrupt Contd..
LOADER
INTERRUPT HANDLER
DISPATCHER
RUNTIME LIBRARY
READY
RUNNING
PCB1
PCB2
PCB2
PCB3
PCB3
I/O
IP
INTERRUPT FLAGS
MASK
OV
MP
PI
TI
I/O
SVC
TO BE DEFINED LATER
I/O DEVICE
Accumulator

Case of I/O interrupt Contd..



Case of I/O interrupt Contd..

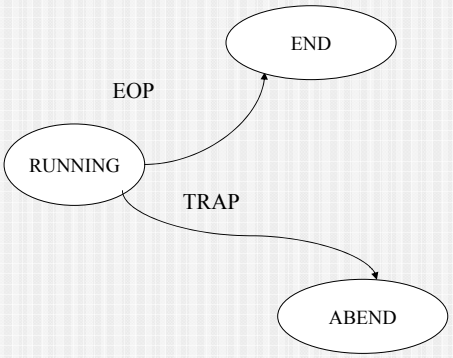


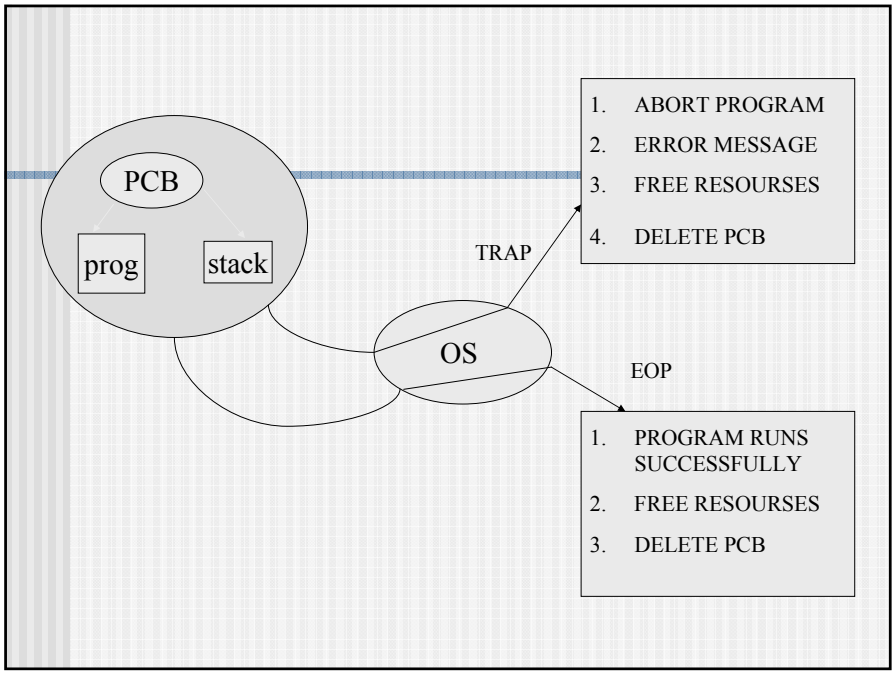


- Handle I/O:

  Here the process waiting in the I/O queue is moved back to the ready state after the I/O request is completed.

End of Process





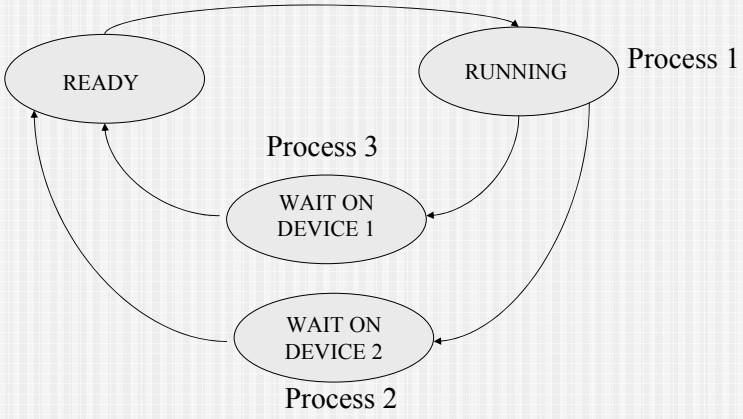
Process Synchronization

Concurrency

- Definition:

  Two or more processes execute concurrently when they execute different activities on different devices at the same time.

Concurrency Contd..



Concurrency Contd..

- In a multiprogramming system CPU time is multiplexed among a set of processes.
- Users like to share their programs and data and the OS must keep information integrity.
- Processes are allowed to use shared data through threads. The concurrent access to shared data may result in data inconsistency.

Concurrency Contd..

- Example :

Consider a variable X =4 and two programs running concurrently

P1
{
Load X ← *Timer interrupt*
X ← X+10
Store X
}

P2
{
Load X
X ← X+2
Store X
}

Concurrency Contd..

- The state of the process P1 is saved and the process P2 executes. The value of X is now:

4+2 = 6

After process P2 finishes execution, P1 resumes execution. Now the value of X becomes

4+10 =14

We see that there are two different values for X

Concurrency Contd..

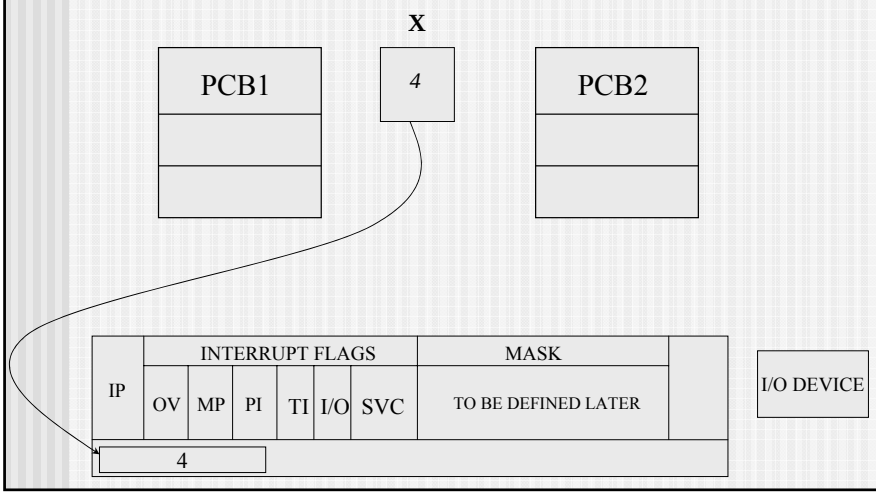
- Consider the case when P1 executes completely. The value of X will be now:

  4+10 = 14

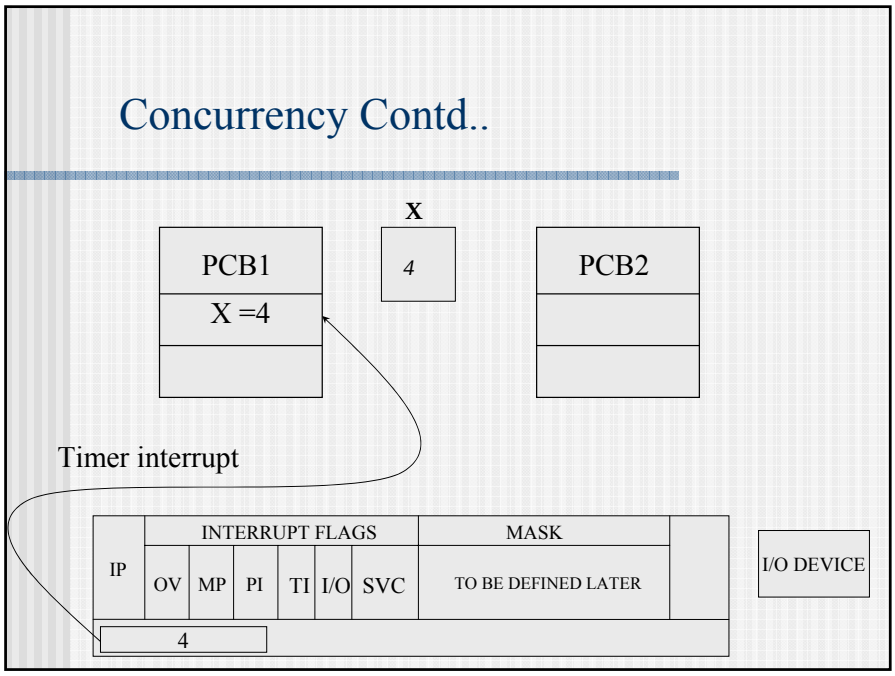
  The process P2 executes and the value of X will be changed to:

  14+2 = 16

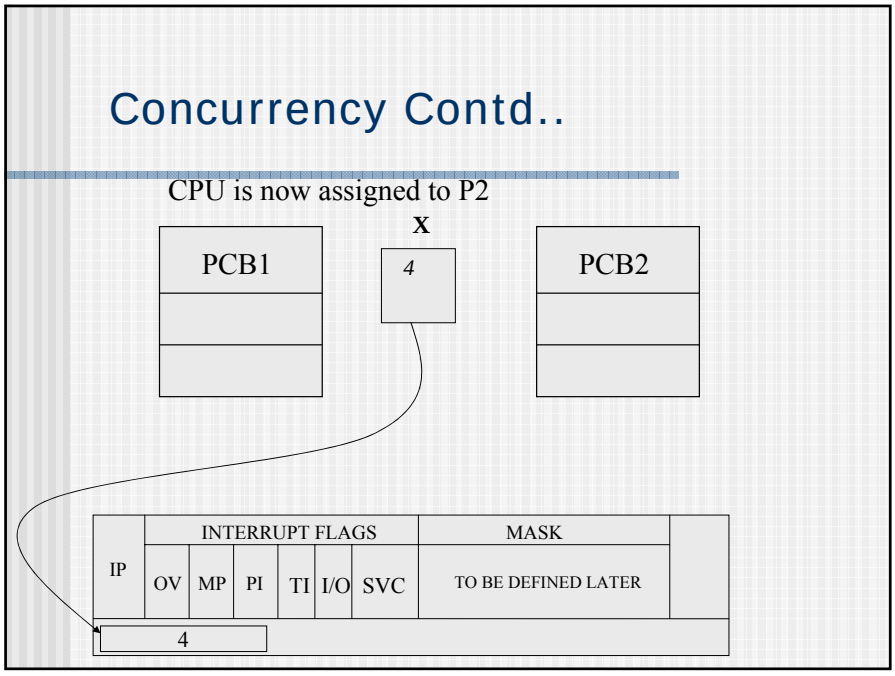
Concurrency Contd..



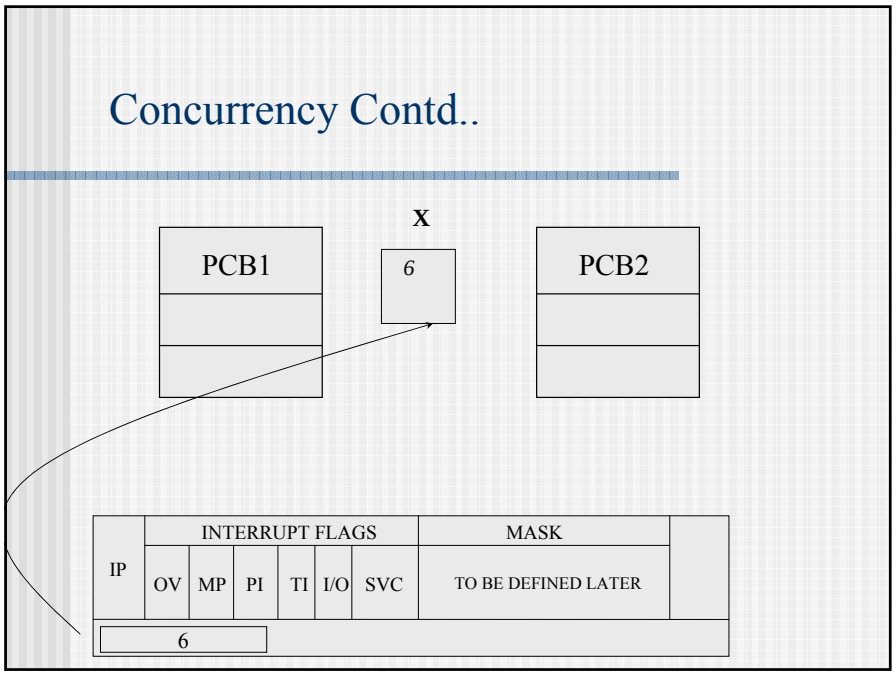
Concurrency Contd..



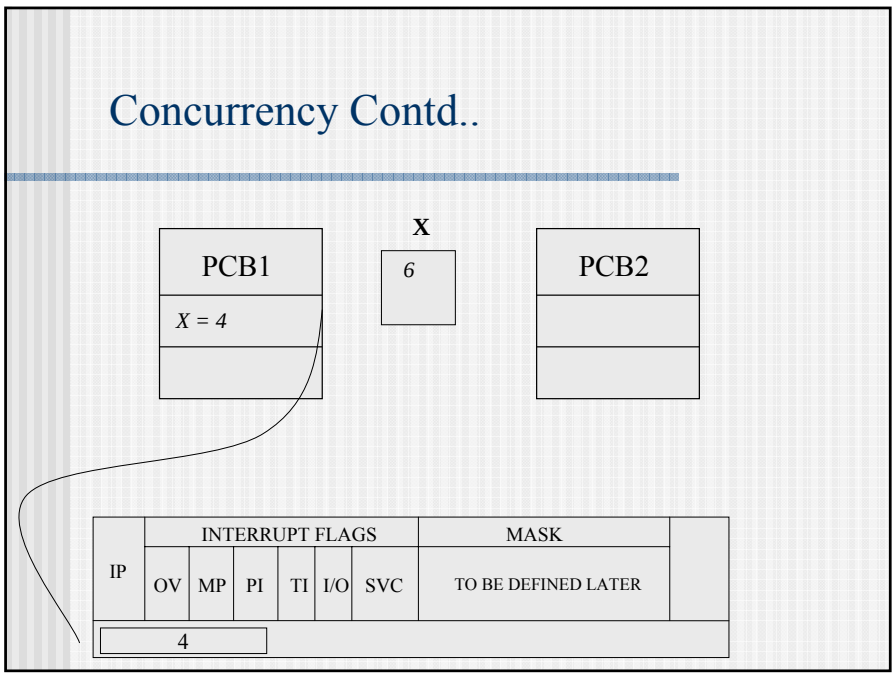
Concurrency Contd..



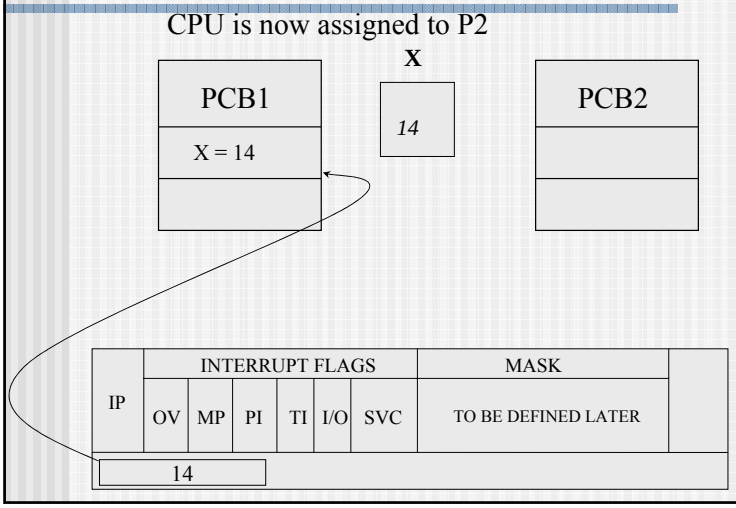
Concurrency Contd..



Concurrency Contd..



Concurrency Contd..



Concurrency Contd..

- Here there are two different values for the same variable X.
- This is called a ***Race Condition***.
- It occurs when processes access shared variables without using an appropriate synchronization mechanism.

Race Condition

- Definition:

  A race condition is an undesirable situation that occurs when two or more operations manipulate data concurrently and the outcome depends on the particular order the operations occur.

  In order to avoid a race condition, it is to necessary to ensured that only one process, at a time, has exclusive access to the shared data.

Race Condition Contd..

The prevention of other process from accessing a shared variable, while one process is accessing it, is called

mutual exclusion

In order to guarantee mutual exclusion we need some kind of synchronization mechanism.

In most synchronization schemes a physical entity must be used to represent a resource. This entity is often called *Lock Byte* or *Semaphore*.

Process Synchronization

- Concept of Critical Section:

  A *Critical Section is the* segment of code where a shared variable is used.

  If several processes are accessing a shared variable when one process is in its critical section, no other process is allowed to enter its critical section.

Process Synchronization contd..

- Each process must request permission to enter the critical section (CS).
- A solution to CS problem must satisfy the following requirements
  1. Mutual exclusion
  2. Progress

Process Synchronization contd..

- Mutual exclusion: When a process is executing in the critical section other processes can not execute their critical sections.
- Progress: If no process is executing in its critical section and there are processes that wish to enter the critical section, only one of them can enter the critical section.

Process Synchronization contd..

- ***Test and Set***

  Before entering the critical section we need to execute a Lock(x) operation and an Unlock(x) operation before leaving the CS.

P1	P2
.	.
.	.
Lock(x)	Lock(x)
{	{
CS	CS
}	}
Unlock(x)	Unlock(x)

Process Synchronization contd..

- If a system implements Test and Set as a hardware instruction, we can implement mutual exclusion with the help of a Boolean variable, *TS,* that is initialized to "*0*" and two operations.

Lock	*Unlock*
Label: If TS = 1 then goto Label *else TS ← 1*	*TS ← 0*

This is implemented in hardware

Process Synchronization contd..

- The main disadvantage here is that when one process is in the critical section all other processes only used the CPU to execute Test and Set.
- This is called ***busy waiting***.
- To overcome this problem the concept of Semaphores was proposed by Dijkstra.

Concept of Semaphores

- ***Semaphores***:

  A semaphore **S** is an integer variable that apart from initialization, is accessed only through two standard "***atomic***" operations.

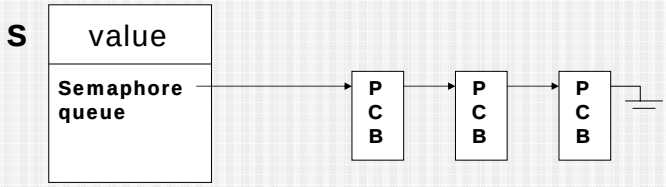
  - ***Wait***
  - ***Signal***

Concept of Semaphores

- When a process executes a ***wait*** operation and finds that the semaphore value is not positive the process ***blocks*** itself, and the OS places the process in the semaphore waiting queue.
- The process will be restarted when some other process executed the ***signal*** operation, which changes the process state from *waiting* to *ready*.

Semaphores contd..

- The operations were originally named as:
  - **P** means Wait
  - **V** means Signal



Semaphores contd..

- The semaphore operations can be defined as follows

```
P(S) :  inhibit interrupts
        S.value    S.value -1
        if S.value < 0
          then {
               add this process to S.queue
                 }
          end;
        enable interrupts
```

Semaphores contd..

```
V(S):
        inhibit interrupts
        S.value :=S.value+1
            if S.value<=0
            then {
                  remove a process from S.queue
                  add process to Ready queue
                   }
             end;
        enable interrupts
```

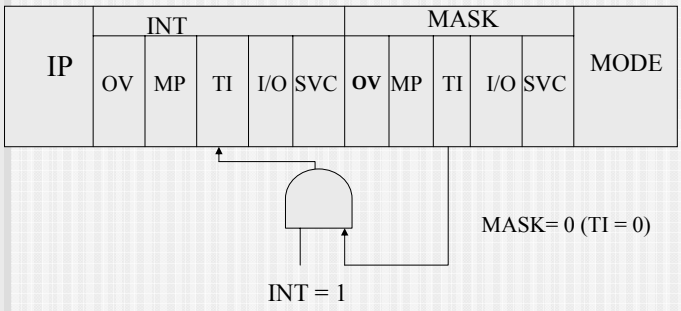
Masking Interrupts…

- We need to disallow or mask the interrupts while the P(s) or the V(s) operations are executed.
- Thus the current sequence of instructions would be allowed to execute without preemption.

Masking Interrupts contd…

- Example

  Consider the PSW



Semaphores contd..

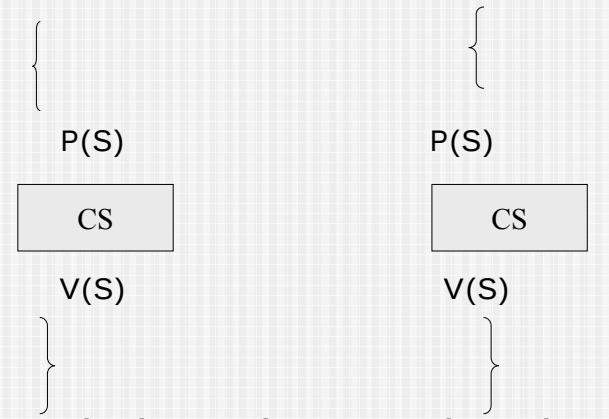
- Algorithms for P and V operations
- P(S)
  1. Decrement value of S by 1
  2. If S<0 then
     - Find current process descriptor
     - Remove from processor queue
     - Add to semaphore queue
  3. Call Dispatcher

Semaphores contd..

- V(S)
  1. Increment value of S by 1
  2. If S<=0 then
     -Dequeue some process descriptor from semaphore queue
     - Add the process to ready queue
  3. Call Dispatcher

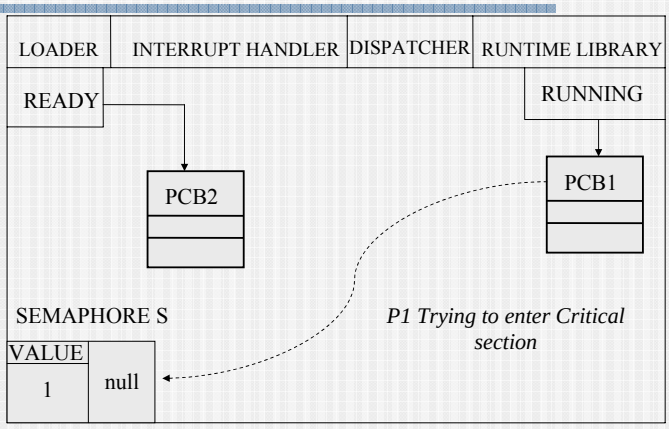
Semaphores contd..

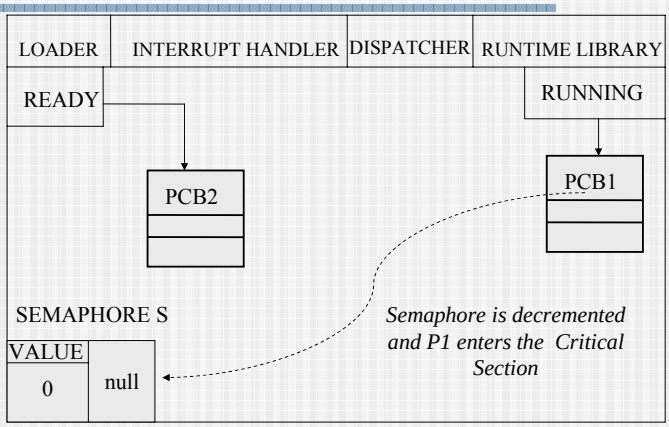
- **P1** **P2**

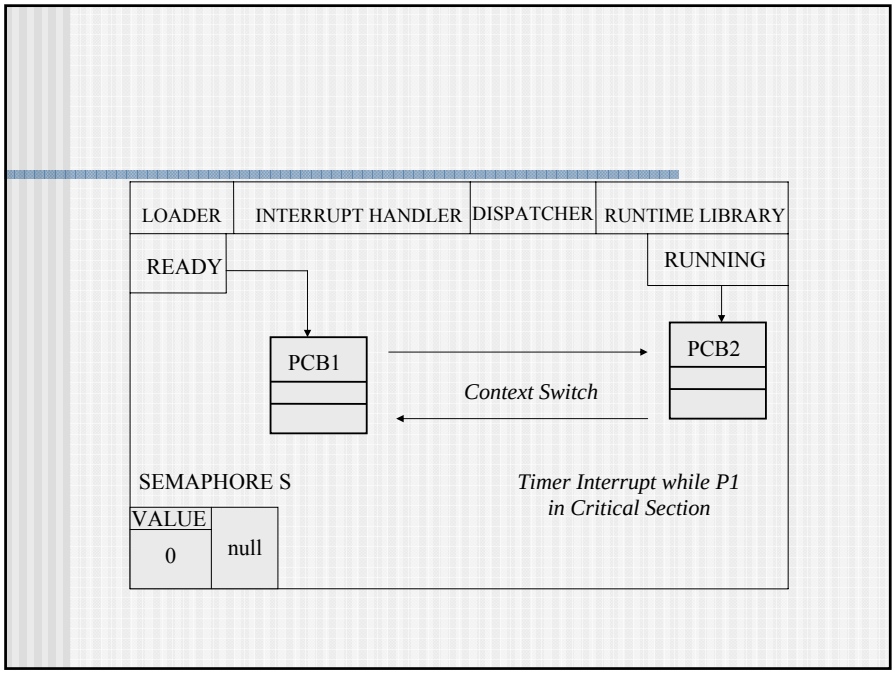


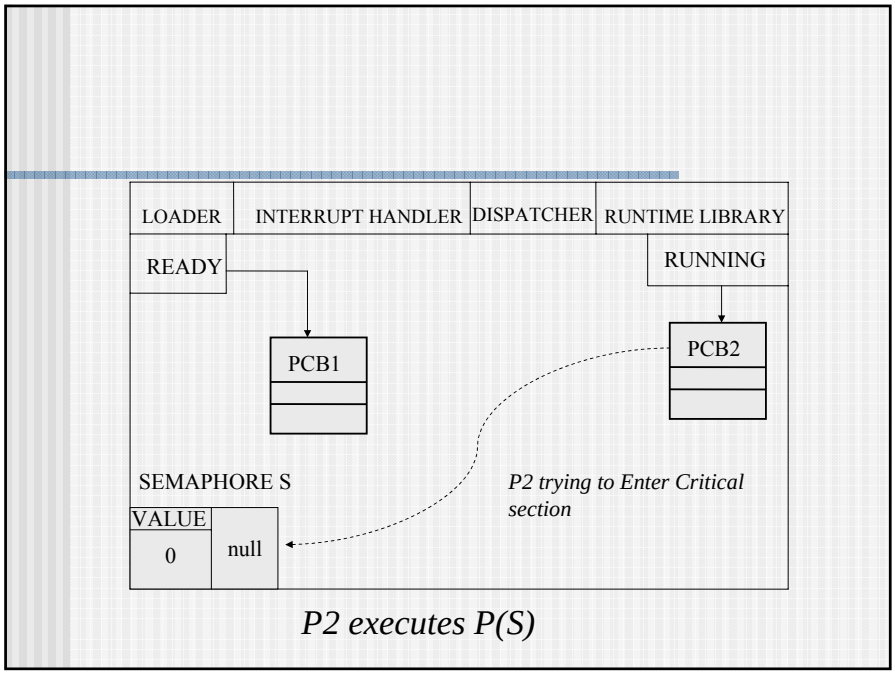
Mutual exclusion implementation with semaphores

Semaphores contd..

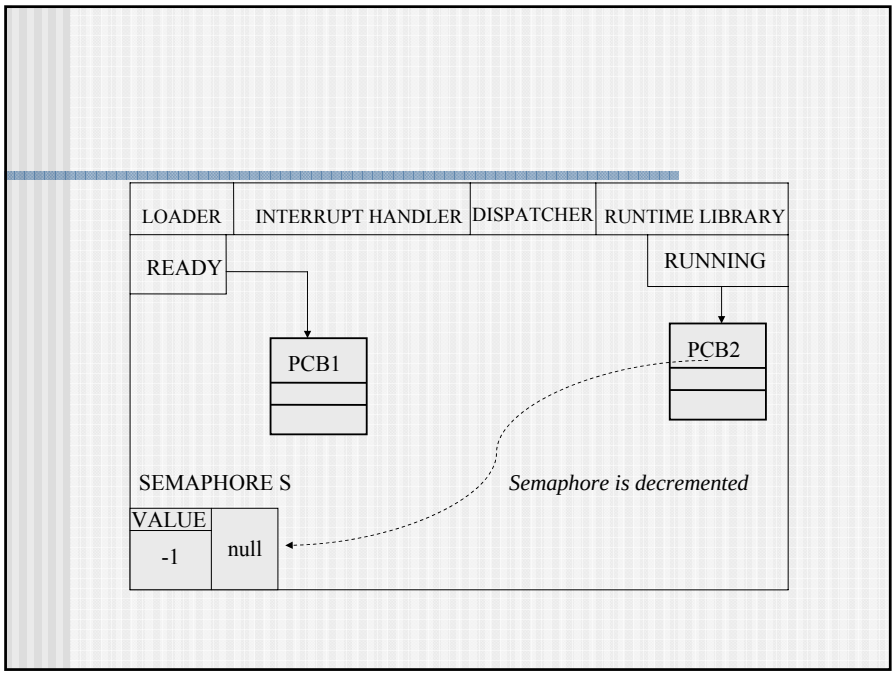




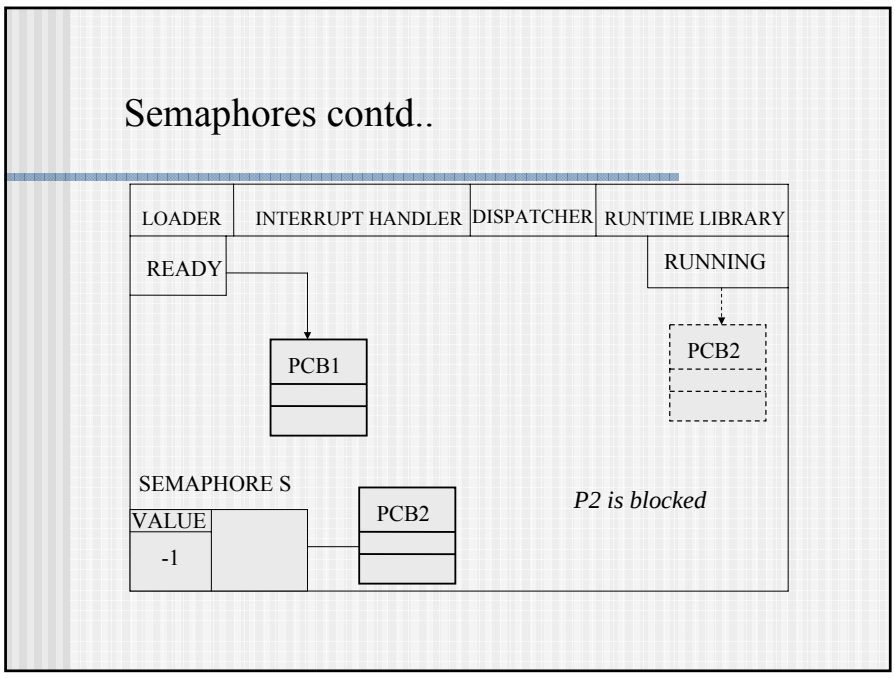




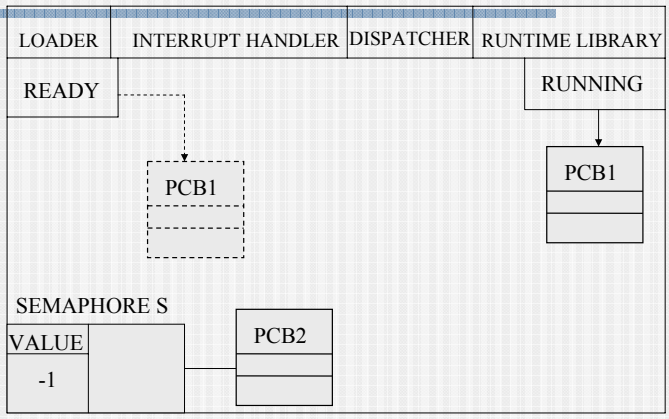
P2 executes P(S)



Semaphores contd..

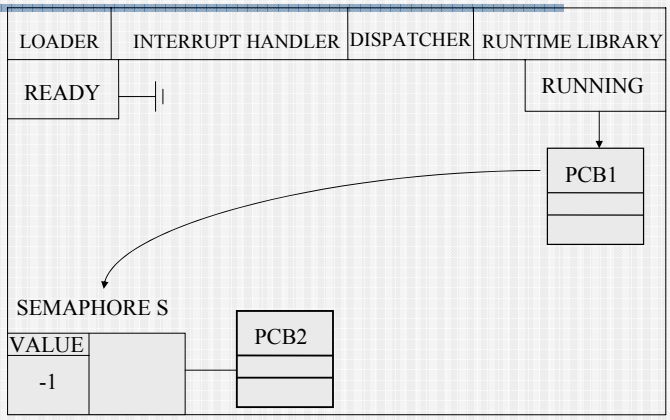


Semaphores contd..



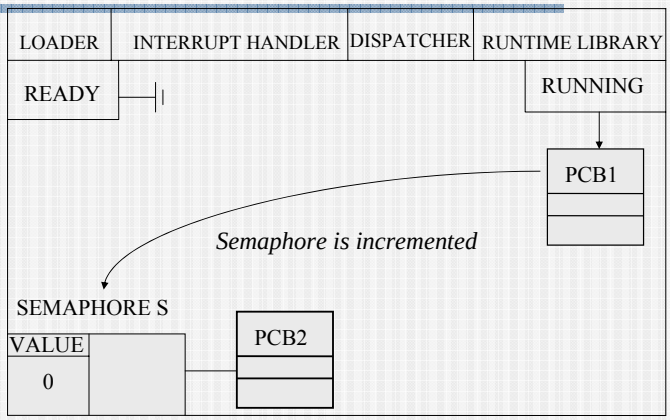
CPU assigned to P1

Semaphores contd..

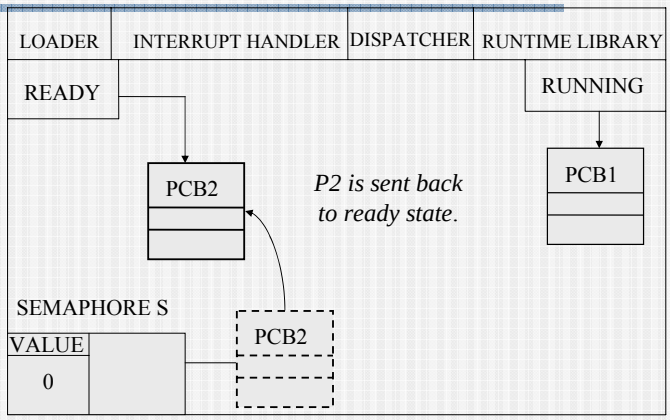


P1 executes V(S)

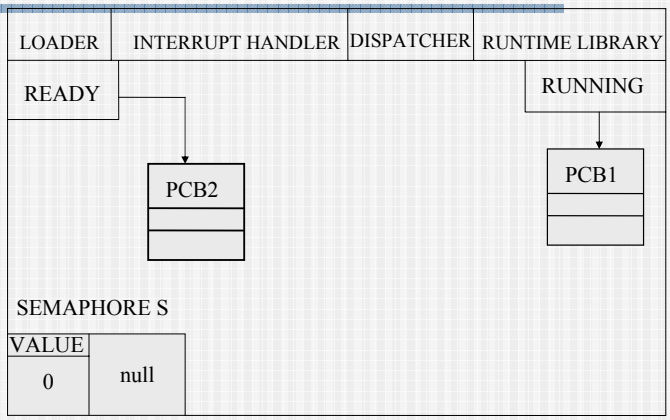
Semaphores contd..



Semaphores contd..



Semaphores contd..



I/O Procedures

I/O Procedures

- We shall now consider how the operating system handles a request for I/O from a user process.
- A request from a process will be a system call to the operating system of the form.
- DOIO(*device, mode, amount, destination, semaphore*)

I/O Procedures contd…

- DOIO — is the name of a system I/O procedure.
- *Device* — is the number of the device on which the I/O operation will take place.
- *Mode* — indicates the operation and sometimes the character code to be used.
- *Amount* — amount of data to be transferred.
- *Destination* — location into which the transfer is to occur
- *Semaphore* — is the address of a semaphore request serviced

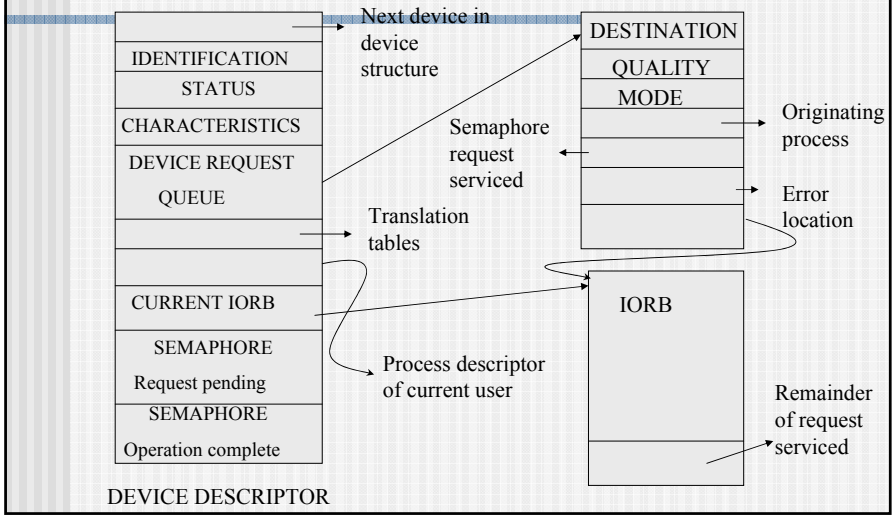
I/O Procedures contd…

- The I/O procedure assembles the parameters of the request into an *I/O request block* and adds it to the I/O request queue.
- The *I/O request queue* is associated to the descriptor of the concerned device and is serviced by a separate process called device handler.

I/O Procedures contd…

- The I/O procedure notifies the device handler that a request has been placed on the I/O request queue by the request pending signal and when the operation is complete the device handler notifies the user by the means of *request serviced* .
- A device handler operates in a continuous cycle during which it removes an IORB from the request queue initiates the corresponding I/O operation and waits for that operation to be completed.

I/O Procedures contd…



Sketch of I/O system