MODULE 1:
Introduction: Concept of Operating Systems, Generations of Operating systems, Types of Operating Systems, OS Services, System Calls, Structure of an OS-Layered, Monolithic, Microkernel Operating Systems, Concept of Virtual Machine, Case study on UNIX and WINDOWS Operating System.

MODULE 2:
Processes: Definition, Process Relationship, Different states of a Process, Process State transitions, Process Control Block (PCB), Context switching Thread: Definition, Various states, Benefits of threads, Types of threads, Concept of multithreads

Process Scheduling: Foundation and Scheduling objectives, Types of Schedulers, Scheduling criteria: CPU utilization, Throughput, Turnaround Time, Waiting Time, Response Time; Scheduling algorithms: Pre-emptive and Non pre-emptive, FCFS, SJF, RR; Multiprocessor scheduling: Real Time scheduling: RM and EDF, Process management in UNIX

MODULE 3:
Inter-process Communication: Critical Section, Race Conditions, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer-Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing, Classical IPC Problems: Reader's & Writer Problem, Dinning Philosopher Problem etc., System V IPC

MODULE 4:
Deadlocks: Definition, Necessary and sufficient conditions for Deadlock, Deadlock Prevention, Deadlock Avoidance: Banker's algorithm, Deadlock detection and Recovery.

MODULE 5:

Memory Management: Basic concept, Logical and Physical address map, Memory allocation: Contiguous Memory allocation – Fixed and variable partition–Internal and External fragmentation and Compaction; Paging: Principle of operation – Page allocation –Hardware support for paging, Protection and sharing, Disadvantages of paging.

Virtual Memory: Basics of Virtual Memory – Hardware and control structures –Locality of reference, Page fault, Working Set, Dirty page/Dirty bit – Demand paging, Page Replacement algorithms: Optimal, first in First Out (FIFO), Second Chance (SC), Not recently used (NRU) and Least Recently used (LRU), Memory Management in UNIX

MODULE 6:

I/O Hardware: I/O devices, Device controllers, Direct memory access Principles of I/O Software: Goals of Interrupt handlers, Device drivers, Device independent I/O software, Secondary-Storage Structure: Disk structure, Disk scheduling algorithms File Management: Concept of File, Access methods, File types, File operation, Directory structure, File System structure, Allocation methods (contiguous, linked, indexed), Free-space management (bit vector, linked list, grouping), directory implementation (linear list, hash table), efficiency and performance. Disk Management: Disk structure, Disk scheduling - FCFS, SSTF, SCAN, C-SCAN, Disk reliability, Disk formatting, Boot-block, Bad blocks

Books

Operating System by P.B. Galvin

G.Gagne

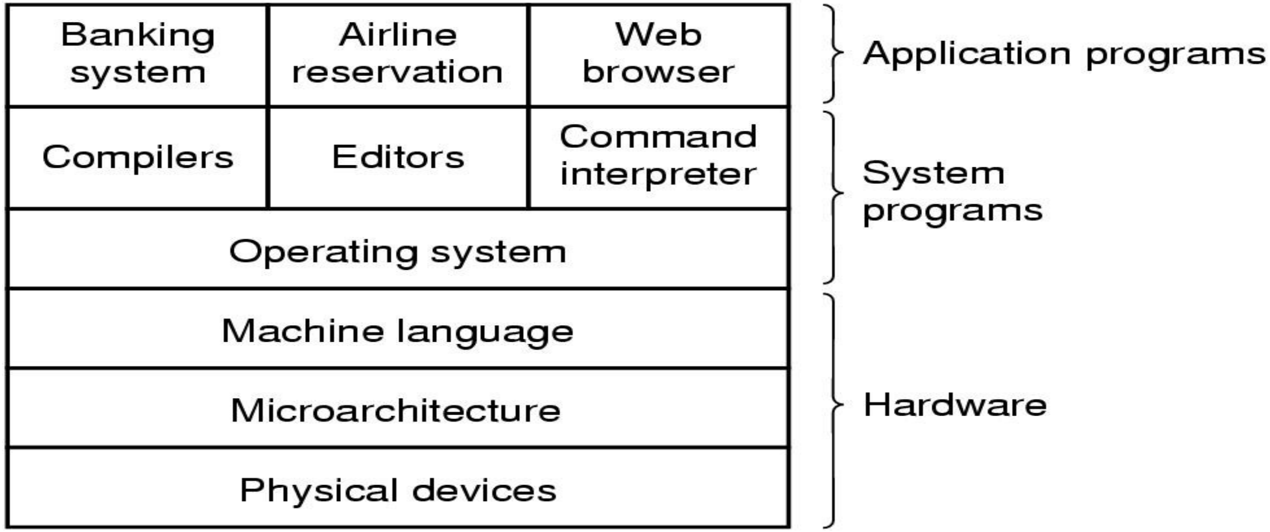
A. Silberschatz

Operating System by Prof. D. M. Dhamdhere

1.1 General Definition

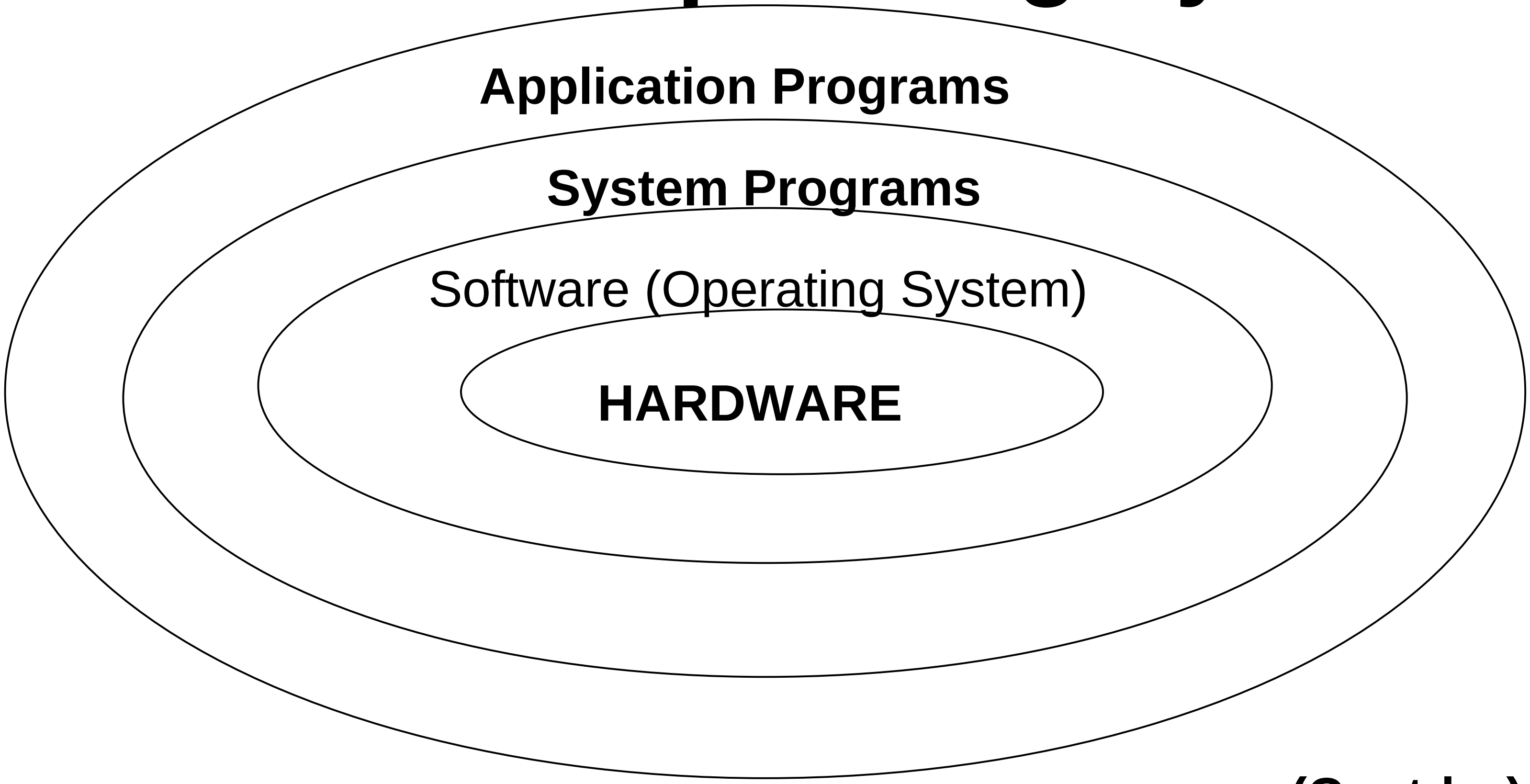
- An OS is a program which acts as an *interface* between computer system users and the computer hardware.
- It provides a user-friendly environment in which a user may easily develop and execute programs.
- Otherwise, hardware knowledge would be mandatory for computer programming.
- So, it can be said that an OS hides the complexity of hardware from uninterested users.

Introduction



- A computer system consists of
  - hardware
  - system programs
  - application programs

Structure of Operating System:



(Contd…)

Structure of Operating System (Contd…):

- The structure of OS consists of 4 layers:
  1. **Hardware**

     Hardware consists of CPU, Main memory, I/O Devices, etc,

  2. **Software (Operating System)**

     Software includes process management routines, memory management routines, I/O control routines, file management routines.

     (Contd…)

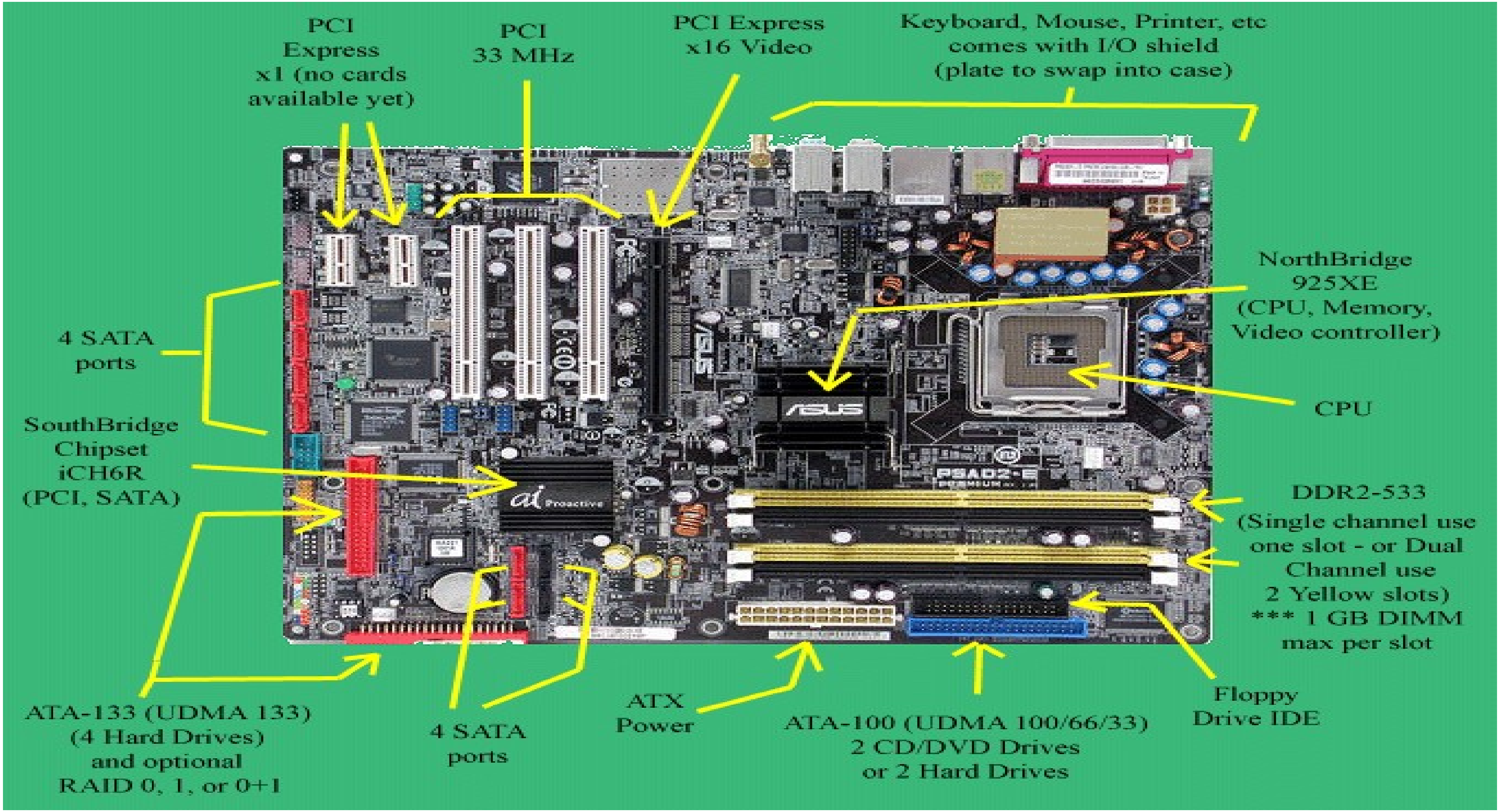
Structure of Operating System (Contd…):

3. **System programs**

   This layer consists of compilers, Assemblers, linker etc.

4. **Application programs**

   This is dependent on users need. Ex. Railway reservation system, Bank database management etc.,


PCI Express x1 (no cards available yet)
PCI 33 MHz
PCI Express x16 Video
Keyboard, Mouse, Printer, etc comes with I/O shield (plate to swap into case)
NorthBridge 925XE (CPU, Memory, Video controller)
4 SATA ports
SouthBridge Chipset iCH6R (PCI, SATA)
CPU
DDR2-533 (Single channel use one slot - or Dual Channel use 2 Yellow slots) *** 1 GB DIMM max per slot
ATA-133 (UDMA 133) (4 Hard Drives) and optional RAID 0, 1, or 0+1
4 SATA ports
ATX Power
ATA-100 (UDMA 100/66/33) 2 CD/DVD Drives or 2 Hard Drives
Floppy Drive IDE

- We buy the computer hardware.
- Then we ask the vendor to install the operating system (Windows , Linux , MAC....etc).
- Then we install the device drivers and anti virus.
- Then we can install the application software(MS word, VlC player ….etc)
- Then we start working on it.

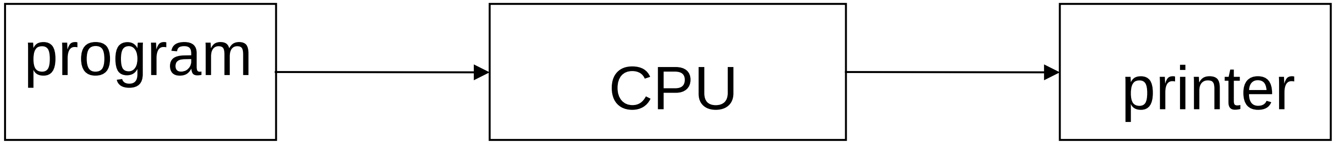
History of Operating Systems

- First generation 1945 - 1955
  - vacuum tubes, plug boards
- Second generation 1955 - 1965
  - transistors, batch systems
- Third generation 1965 – 1980
  - ICs and multiprogramming
- Fourth generation 1980 – present
  - personal computers

History

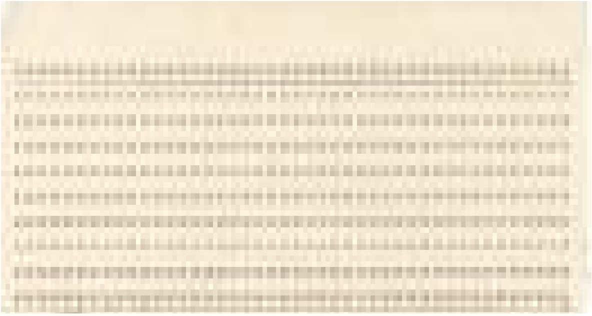
- Pre 1950 : the very first electronic computers
  - valves and relays
  - no OS
  - single program (written with 0 and 1) with dedicated function

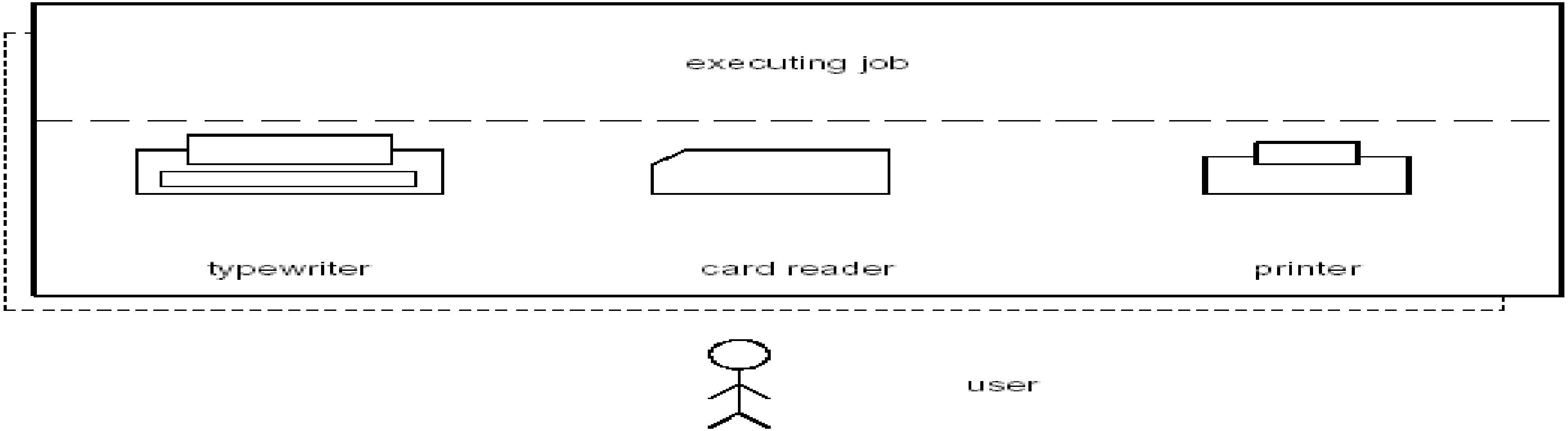
- Pre 1960 : stored program valve machines
  - single job at a time
  - Still program written with 0 and 1.
  - OS just consists of a program loader



Early Systems

- Structure
  - Single user system.
  - Programmer/User as operator (Open Shop).
  - Large machines run from console.





Example of an early computer system

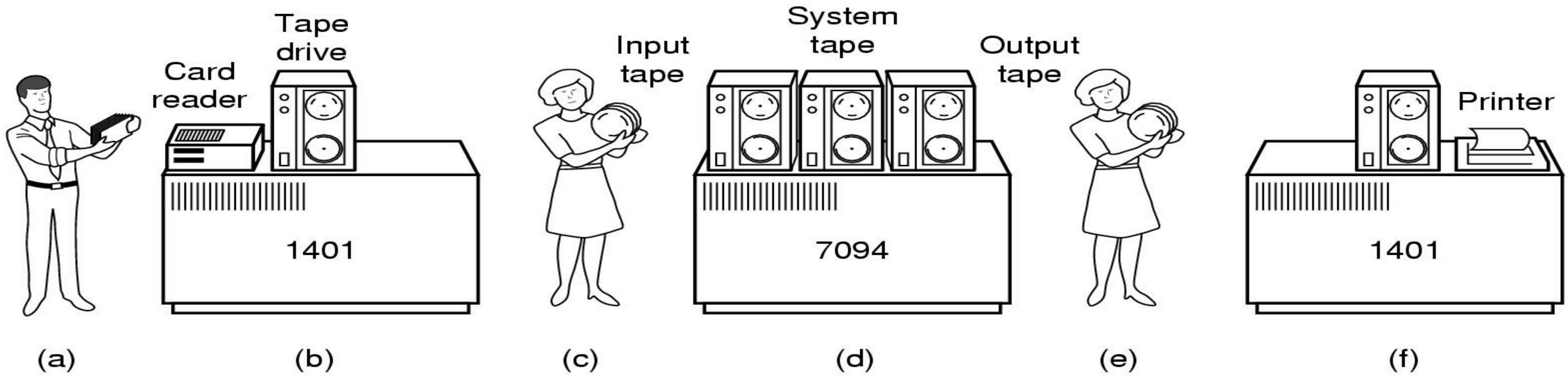
First generation: direct input

- Run one job at a time
  - Enter it into the computer (might require rewiring!)
  - Run it
  - Record the results
- Problem: lots of wasted computer time!
  - Computer was idle during first and last steps
  - Computers were ***very*** expensive!
- Goal: make better use of an expensive commodity: computer time

solution

- Paper tape is slower than the tape drive.
- New device called tape is invented.
- Remember video or audio cassette

History of Operating Systems (1)



Early batch system
- bring cards to 1401
- read cards to tape
- put tape on 7094 which does computing
- put tape on 1401 which prints output

To prepare a FORTRAN program for execution

1. LOAD the FORTRAN compiler in to the computer. The compiler was normally kept on a magnetic tape . So it would need to be mounted on a tape drive.
2. The program would be read through the card reader and written onto another tape.
3. The compiler would then take the program as input and then produce assembly language output.
4. The assembler would now need to assemble the above output. This would mean mounting another tape with the assembler.
5. The output of the assembler would need to be linked with its supporting library routines.
6. Finally the binary object form of the program would be ready to execute.

Problem

1. Setup time for execution

2. CPU utilization.

Solution

Batch programming:

Computer operators batch similar types of jobs together and execute.

All C programming jobs

Then all JAVA programming jobs.

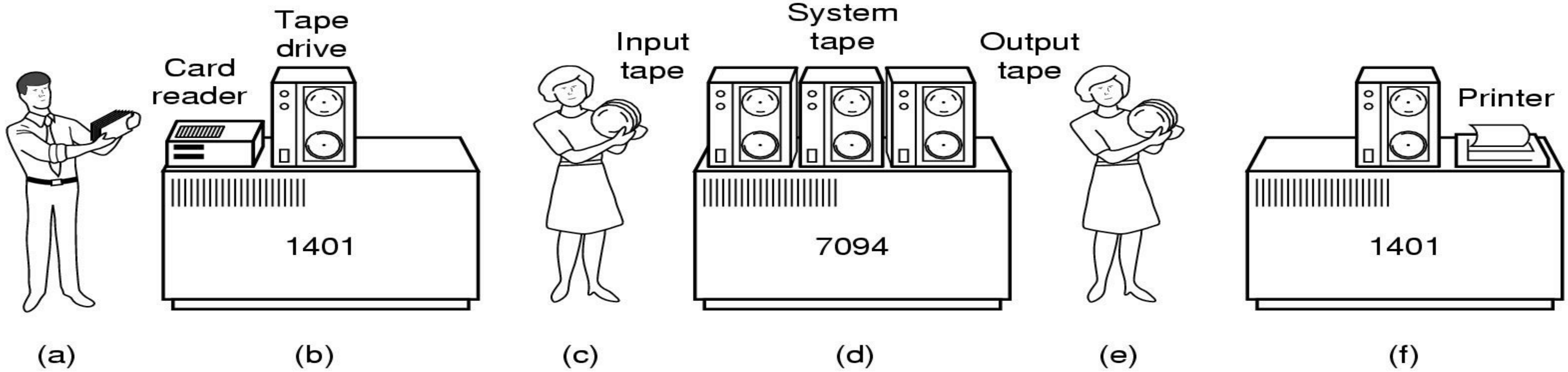
SPOOLING

When a job is involved with I/O operation the CPU sit idle because the I/O devices are very slow compared to CPU.

To overcome this off-line processing was used.

CPU does not read or write to cards or printers(tapes).

spooling


Tape
drive
Card
reader
1401
Input
tape
System
tape
Output
tape
7094
Printer
1401
(a)
(b)
(c)
(d)
(e)
(f)

SPOOLING

There was a problem with tape....

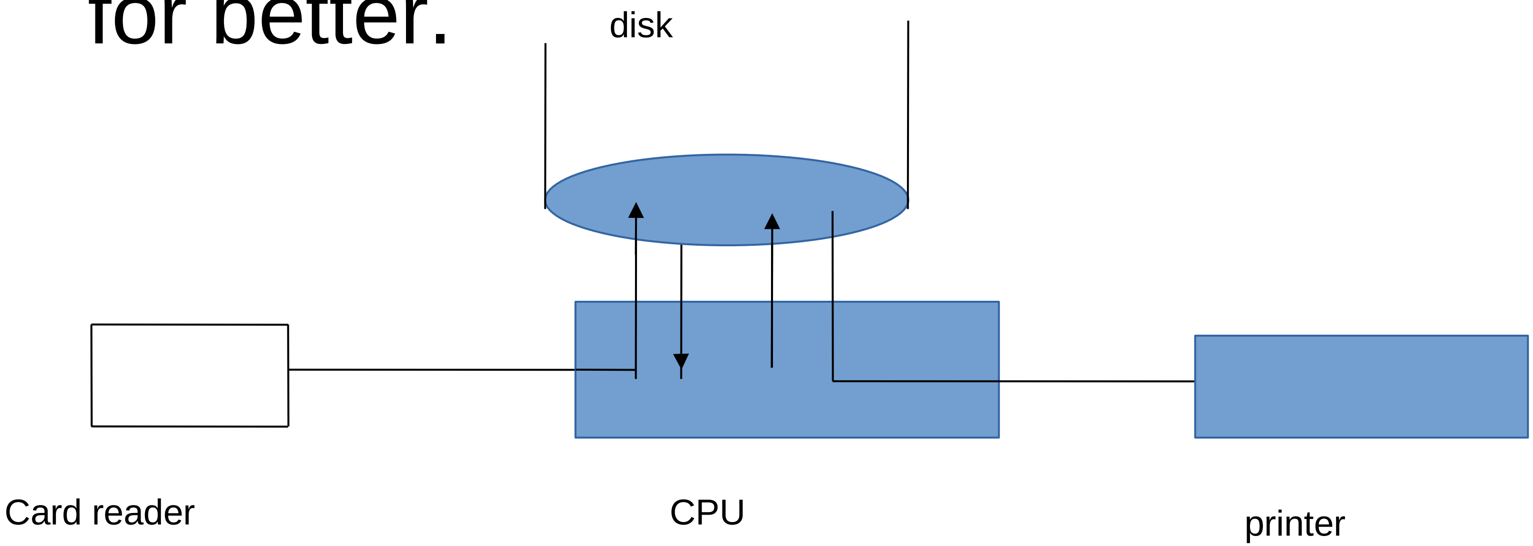
Tape is sequential access device

If some one is writing into the beginning of the tape no one can read from the end of the tape...

Because there is single head to read and write in the tape.

SPOOLING

Disk became widely available and things changed for better.



SPOOLING

Read and write can be done simultaneously in Disk.

And it is a random access device(example HDD
Vlc and programming)

SPOOLING (Simultaneous Peripheral Operation On-Line)

In a disk system cards are read directly from the card to the disk.

When a job is executed and a card is needed as input the equivalent record is read from the disk.

Similarly the output is written onto the disk.

This is known as SPOOLING.

Spooling can keep both the CPU and the I/O devices work at much higher speed.

BY that time memory became large...

More than one processes can be loaded in the memory at the same time.

Still problem

CPU still not utilized fully....

E.g:

When a job stopped the operator would have to notice that by observing the console , Determine why it stopped(normal / abnormal) and then take a necessary actions.

CPU Switching from one job to another takes time

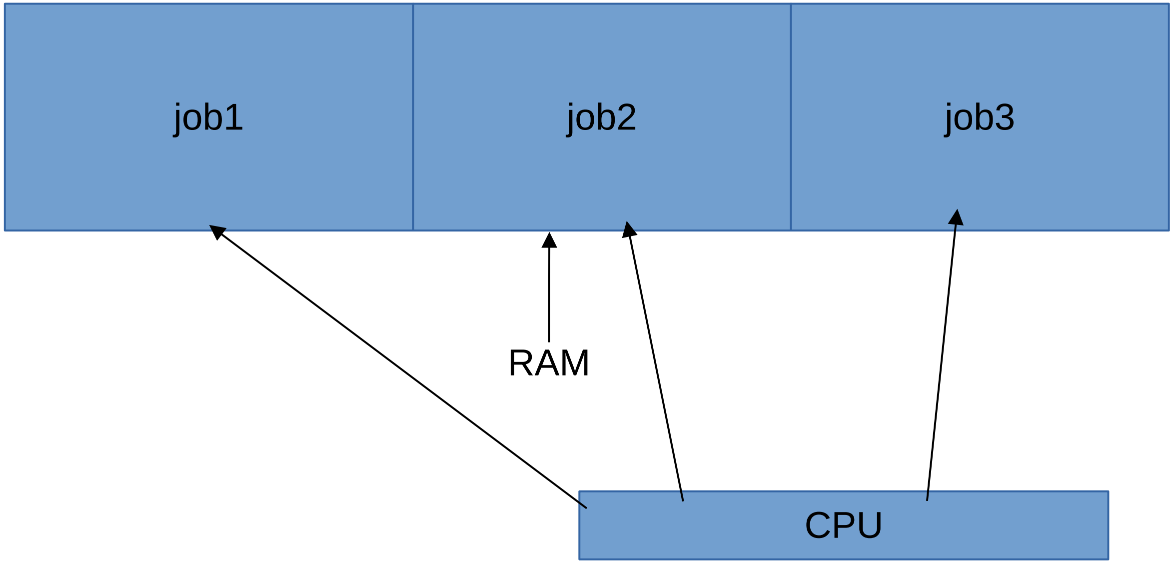
Still problem

CPU still not utilized fully....

E.g:

When a job stopped the operator would have to notice that by observing the console , Determine why it stopped(normal / abnormal) and then take a necessary actions.

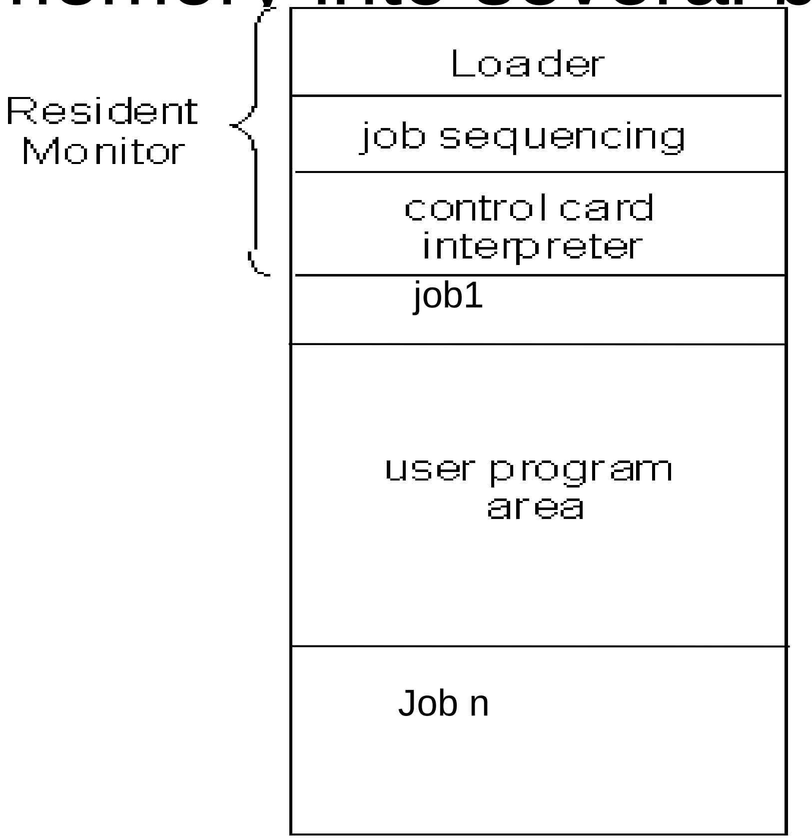
CPU Switching from one job to another takes time


job1
job2
job3
RAM
CPU

Solution

Resident monitor...... ?

The first operating systems were designed to improve the utilization. The solution was simple: break up the memory into several blocks, as shown



One program was always loaded into memory, and continuously running, the resident monitor.

If a user program was loaded into memory, the resident monitor would transfer control to the program.

When the program halted, it would return control to the resident monitor, which could then transfer control to the next program.

Even at this stage, interaction with the computer was not via terminals.

The user programs were read from punched cards, and utilities like the FORTRAN compiler from tape.

To improve efficiency, jobs were run in batches - programs that required the same utility (compiler, e.g.) would be run together, to avoid multiple loading/unloading of the compiler.

In modern computers, I/O is much faster.

Usually, the computer reads the executable program from peripheral memory devices such as floppies, or hard disks.

Output, if it is to be printed, is transmitted to printers and displayed on the screen. Still, compared to the CPU speed, I/O is many magnitudes slower.

Another bottleneck arises due to interactive programs.

If the execution requires input from the user, the CPU has to wait a long time while a user enters the input via keyboard (for instance).

Solution?????

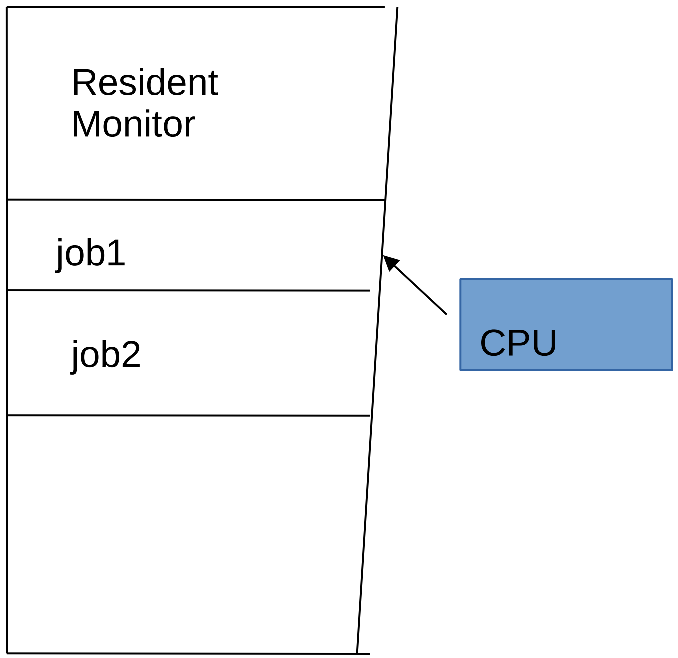
most operating systems schedule many different processes simultaneously.

This is called multiprogramming.

Here, many programs are loaded into different parts of the memory. The OS starts executing the first one.

If/when the execution requires I/O or some such delay, the CPU immediately switches to the next job and starts it; and so on.

The concept is similar to how human managers operates.

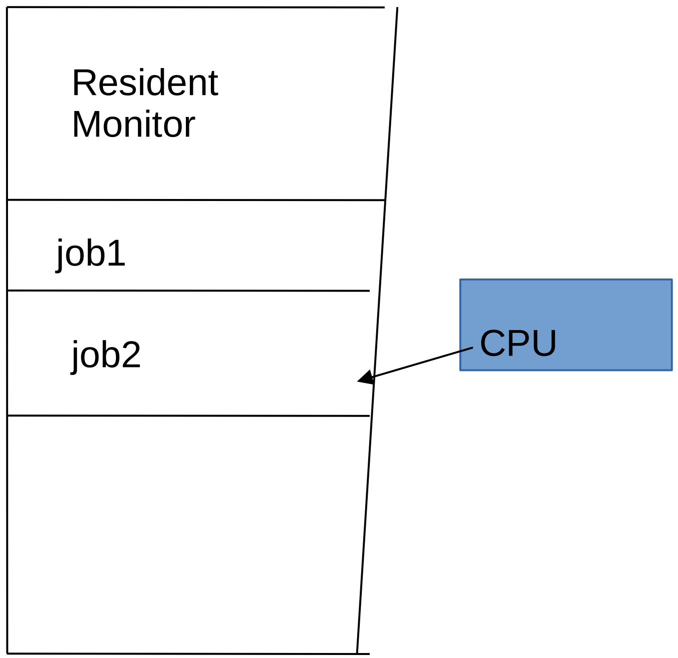



```
#include<stdio.h>
Int main()
{
        Int I;
        printf("\n enter the number");
        Scanf("%d",&i);
        printf("\n i=%d",i);
        Return 0;
}
```

I/o

job1

When job1 is doing I/o cpu sit idle




```
#include<stdio.h>
Int main()
{
        Int I;
        printf("\n enter the number");
        Scanf("%d",&i);
        printf("\n i=%d",i);
        Return 0;
}
```

I/o

job1

When job1 is doing I/o cpu sit idle

```
#include<stdio.h>
Int main()
{
        Int p;
        P=90+20;
        Return 0;
}
```

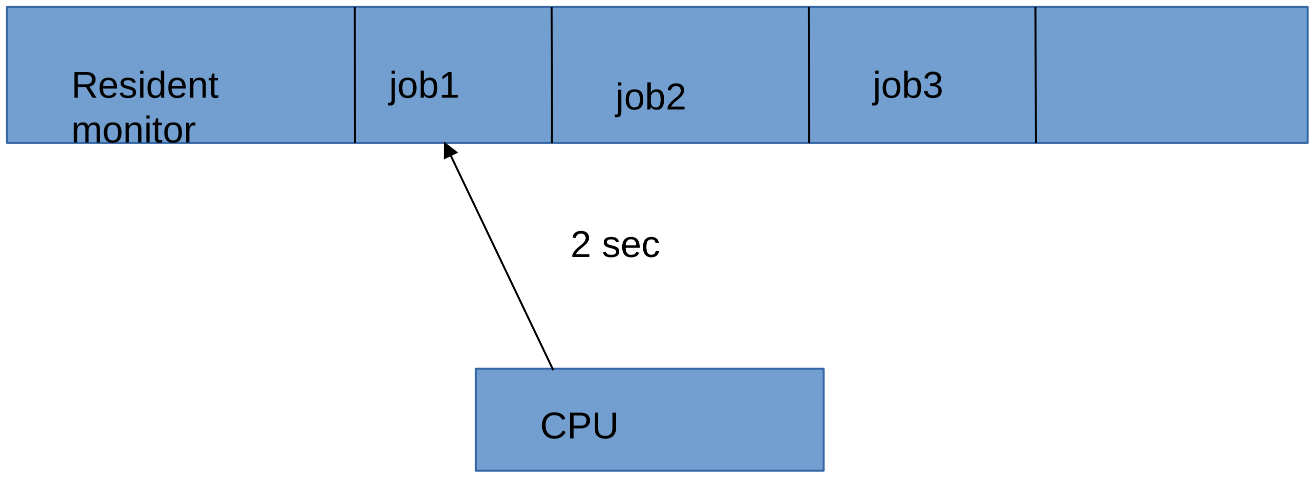
job2

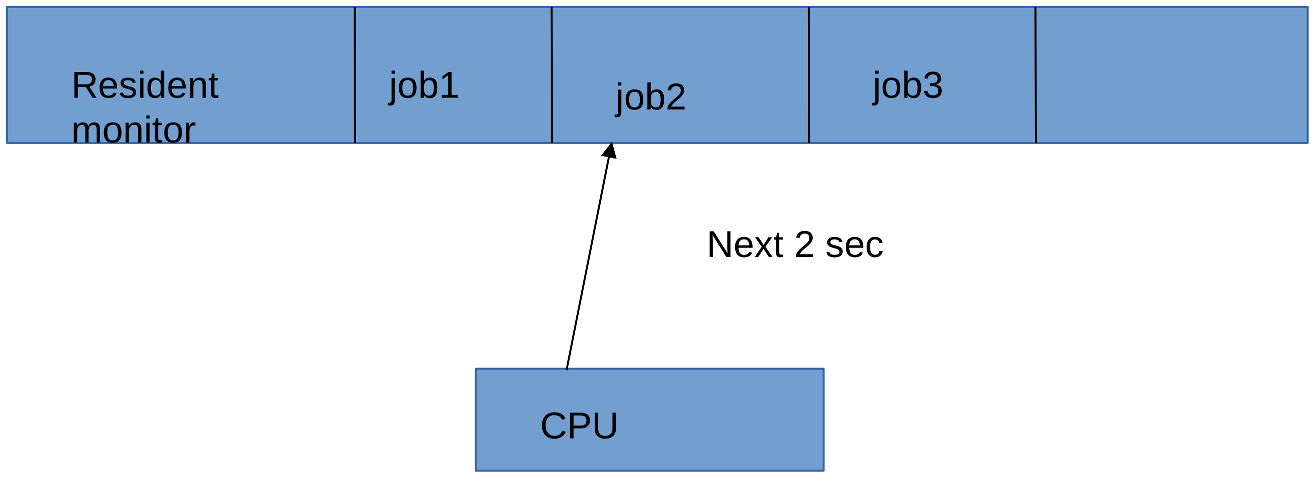
THE OTHER WAY IS
TIME SHARING(or Multi tasking)

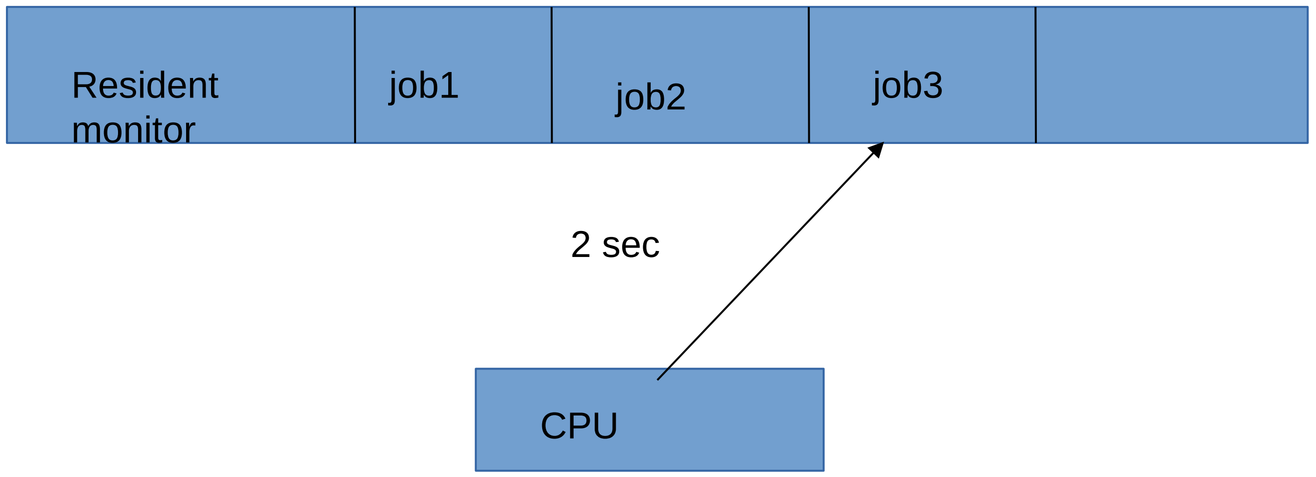
. Is an extension of multiprogramming.

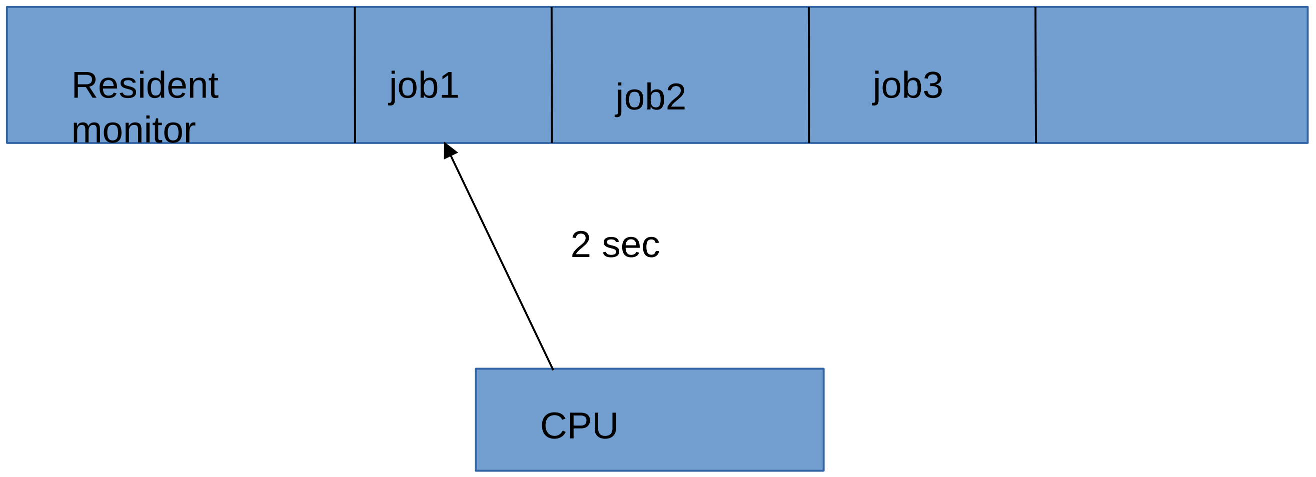
. Like in multiprogramming multiple jobs are executed by the cpu switching between them but the switches occur so frequently that the users may interact with each program while it is running.

. It gives an impression that s/he has his or her own computer. Whereas actually one computer is being shared among many users.


Resident monitor
job1
job2
job3
2 sec
CPU


Resident monitor
job1
job2
job3
Next 2 sec
CPU


Resident monitor
job1
job2
job3
2 sec
CPU


Resident monitor
job1
job2
job3
2 sec
CPU

Multitasking and Multiprogramming

An interactive computer system provides online communication between the user and the system.

The user gives instruction to the operating system or to the program directly and receives an immediate response.

Time sharing os is more complex than the Multiprogramming.

1. several jobs must be kept simultaneously in memory, which requires some form of memory management and protection.

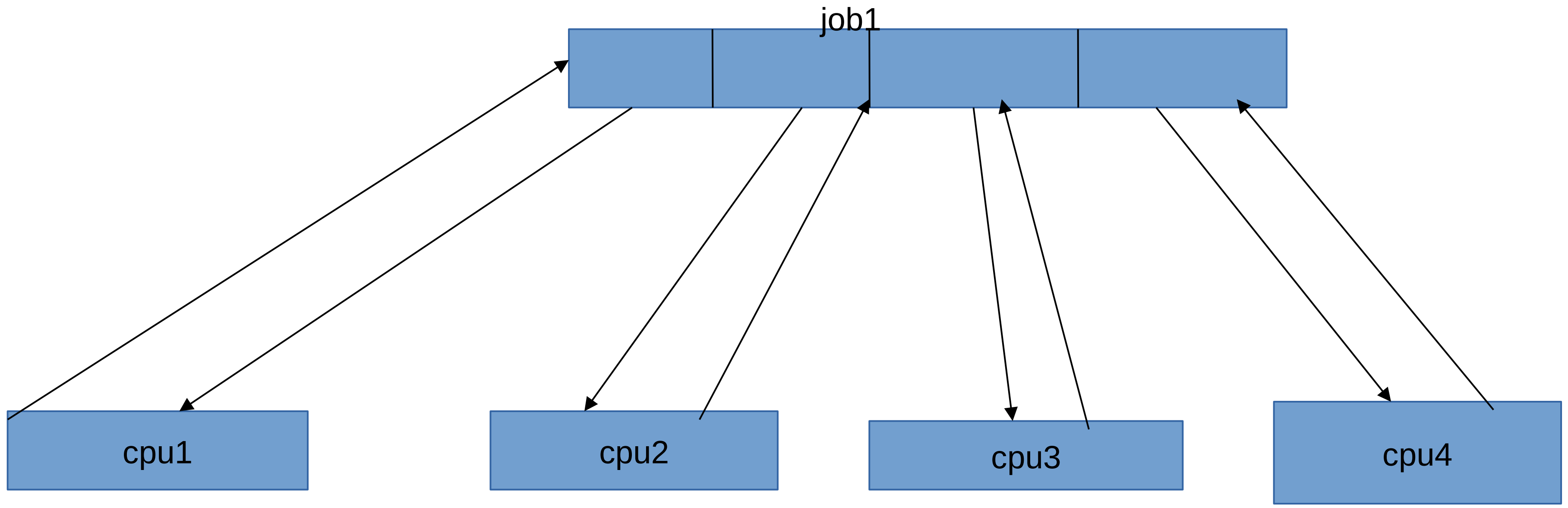
2. Time sharing system must also provide an online file system. The file system resides on a collection of disk hence , disk management is required.

3. They also provide a mechanism for concurrent execution, which required sophisticated CPU scheduling policies.

4. They also must provide mechanism for job synchronization and communication and must ensure that dead lock does not occure.

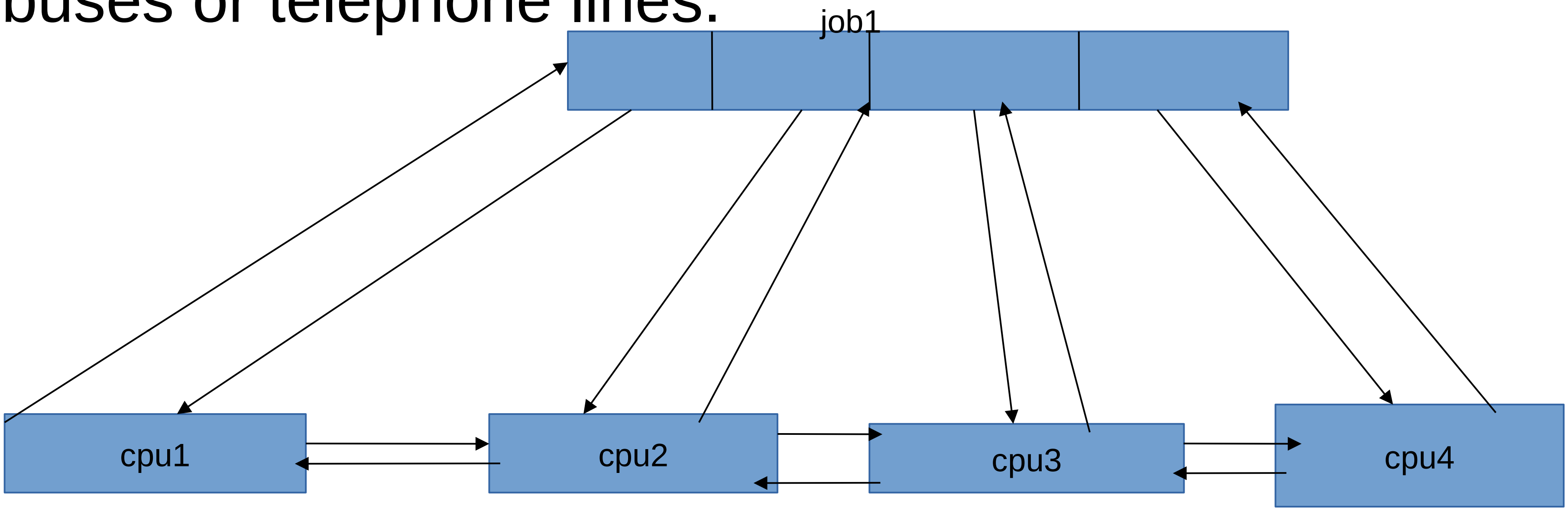
Distributed OS

Distribute the jobs among many processors.



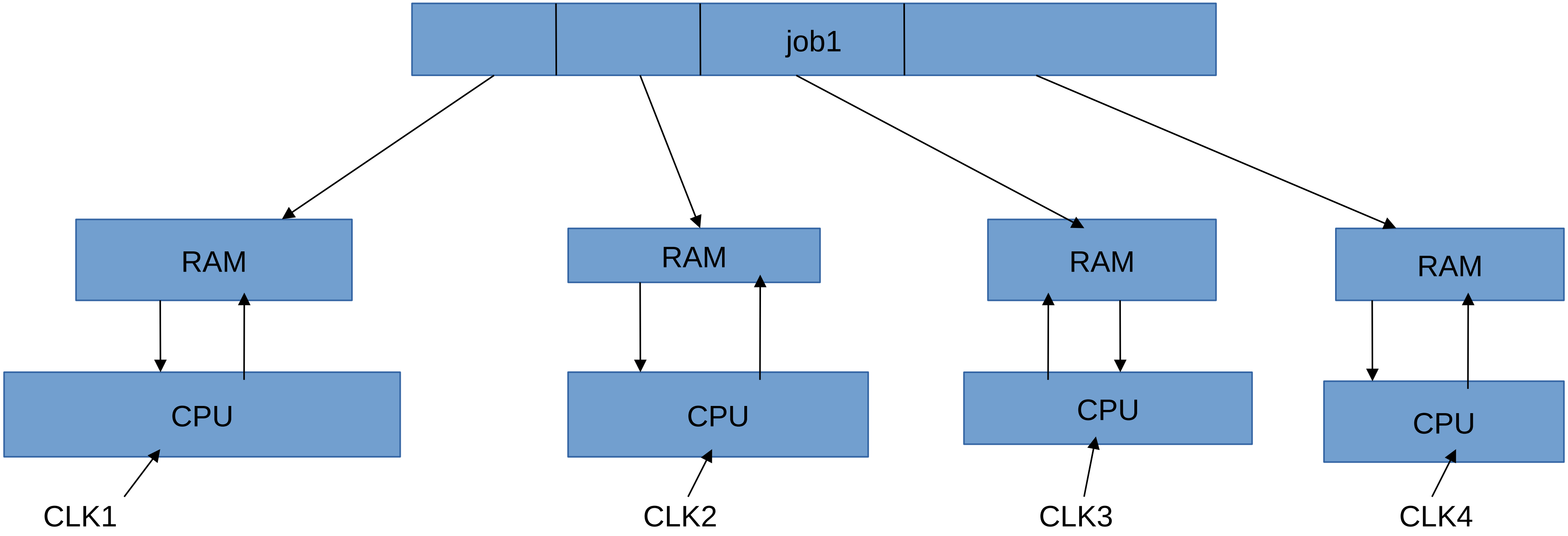
Distributed OS

The processors communicate with each other through various communication lines, such as high speed buses or telephone lines.



Distributed OS
Loosely coupled system

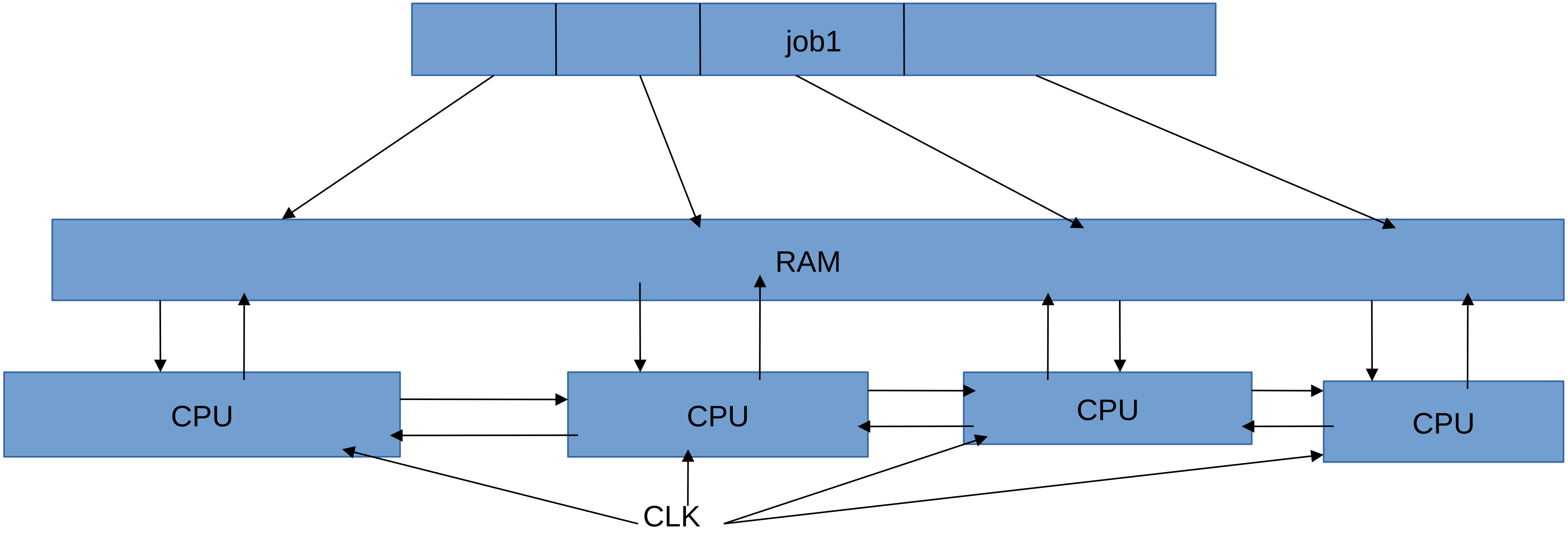
It does not share memory or a clock.



Distributed OS

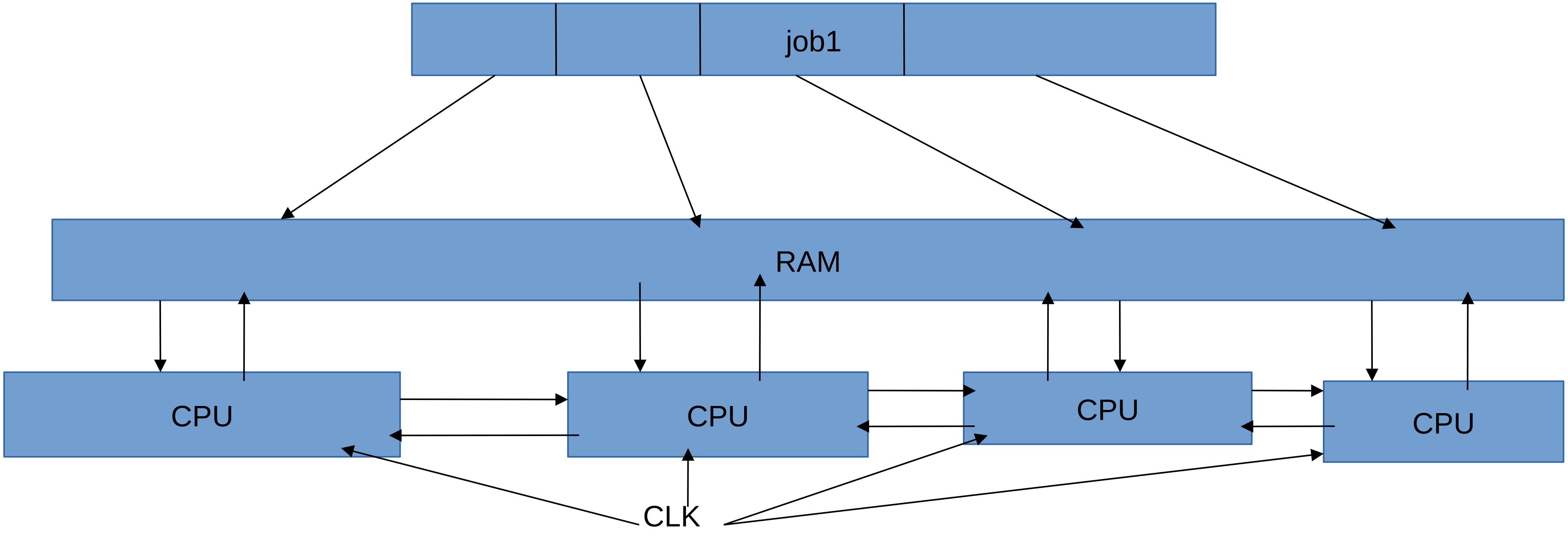
Tightly coupled system

One in which there are multiple processors which communicate with each other by sharing computer bus,the clock,memory etc.



Distributed OS
Tightly coupled system

One in which there are multiple processors which communicate with each other by sharing computer bus,the clock,memory etc.



Distributed OS

Also known as parallel system or multiprocessor system.

Why Distributed system?

1. Resource sharing: files, printers etc...

2. Computation speed up: A problem can be divided into sub problems and can be run concurrently in different CPU and result is found quickly.

3. Reliability: If one CPU fails others are still working.

4. Communication: When a number of CPUs are connected communication takes places and information is exchanged.

Real time os(RTOS)

A real time os is considered to function correctly only if it returns the correct result the correct result within any time constraints.

E.g: Missile technology

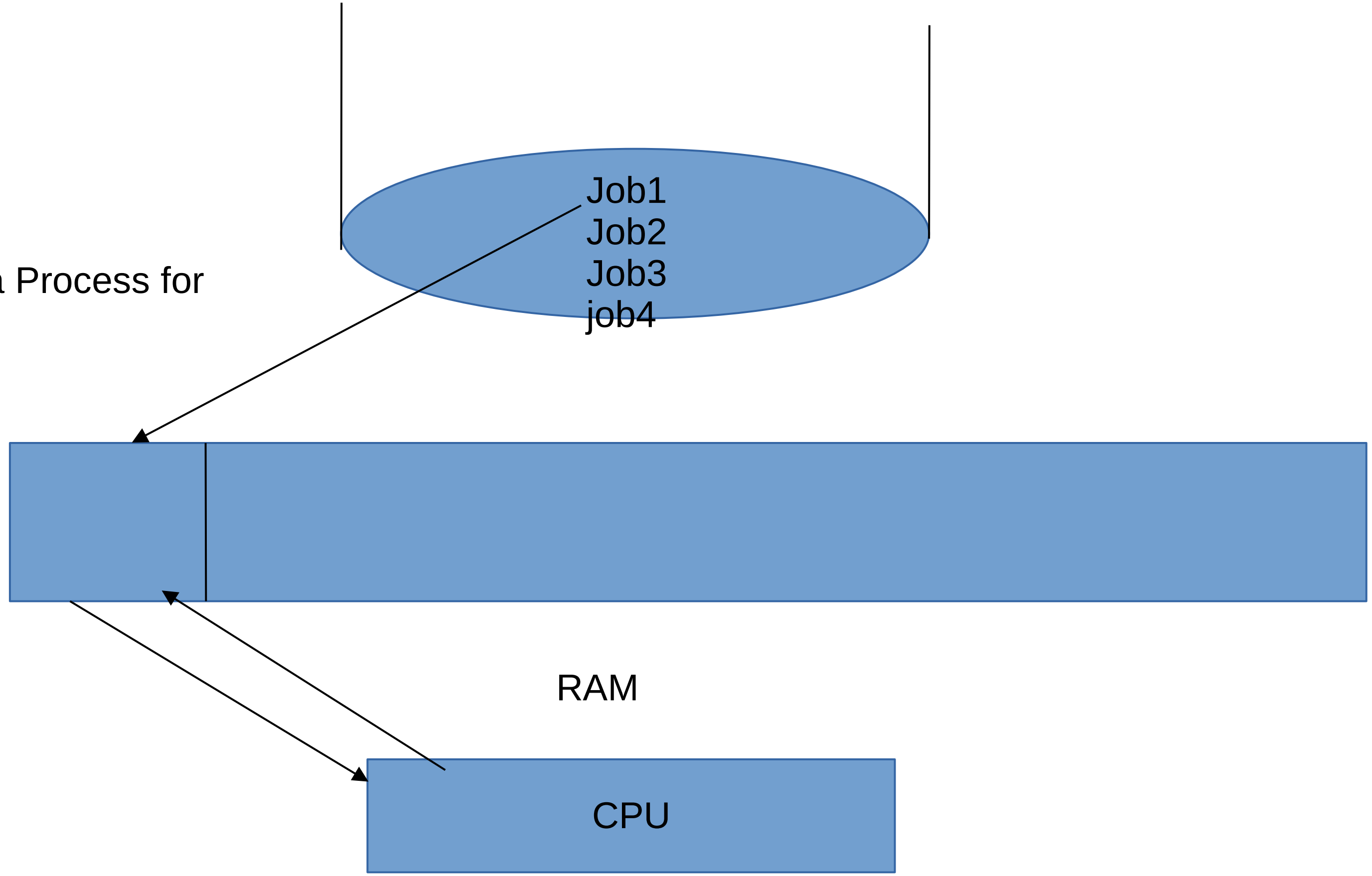
- Space technology
- Robotics

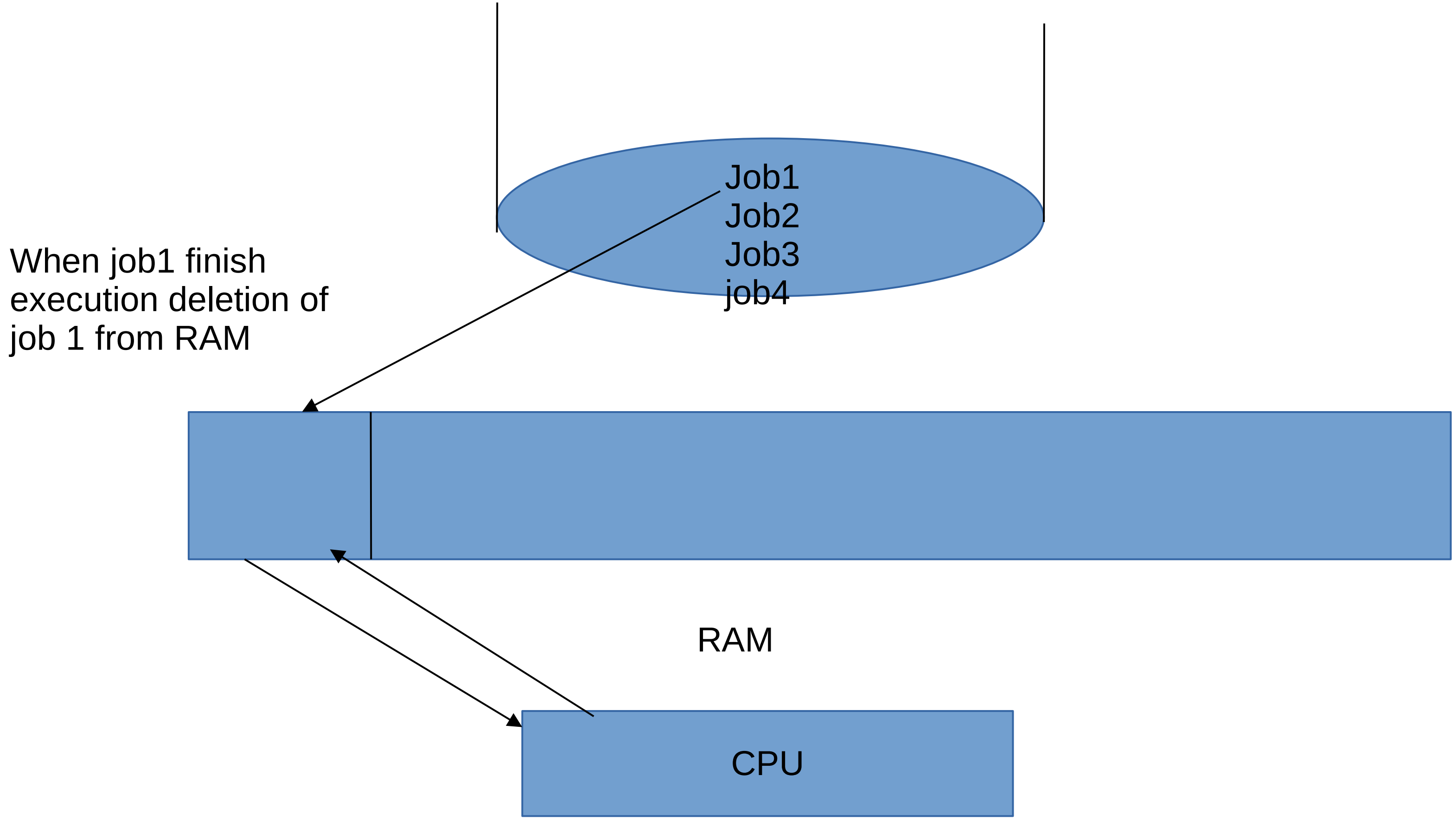
OS Components

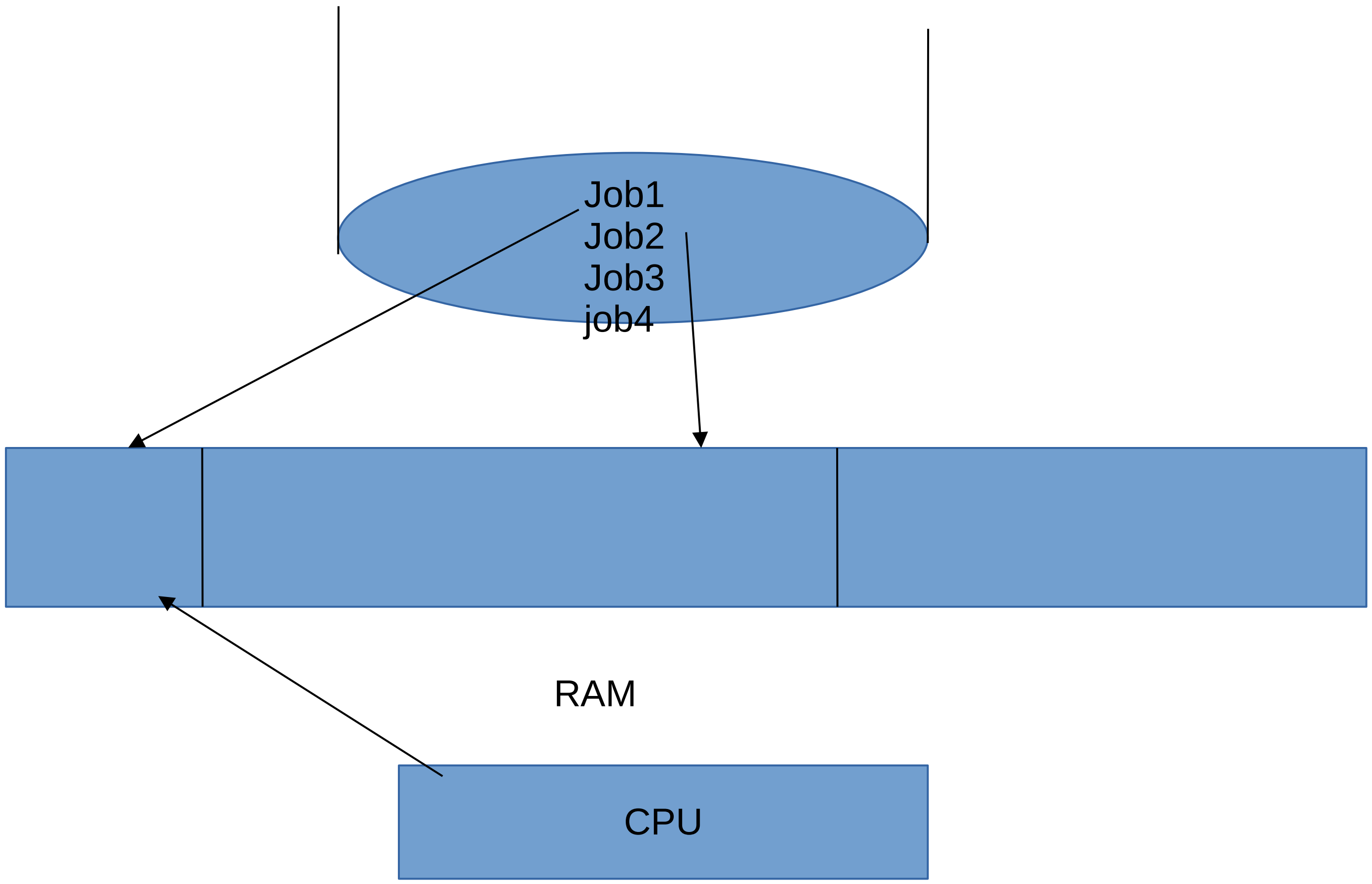
1. Process management:

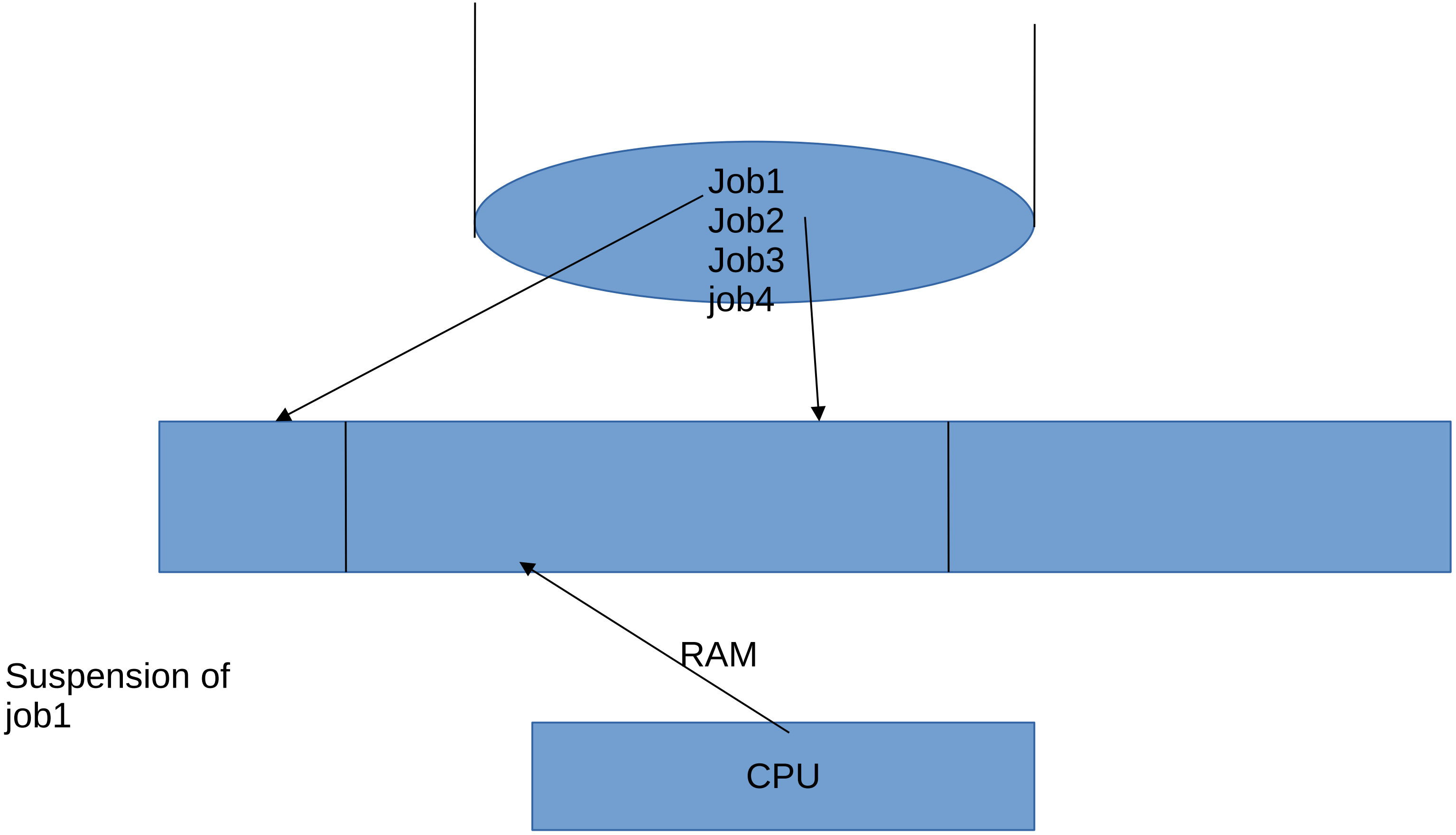
- Creation and deletion of process.
- Suspension and resumption of process.
- Process communication.
- Deadlock handling.

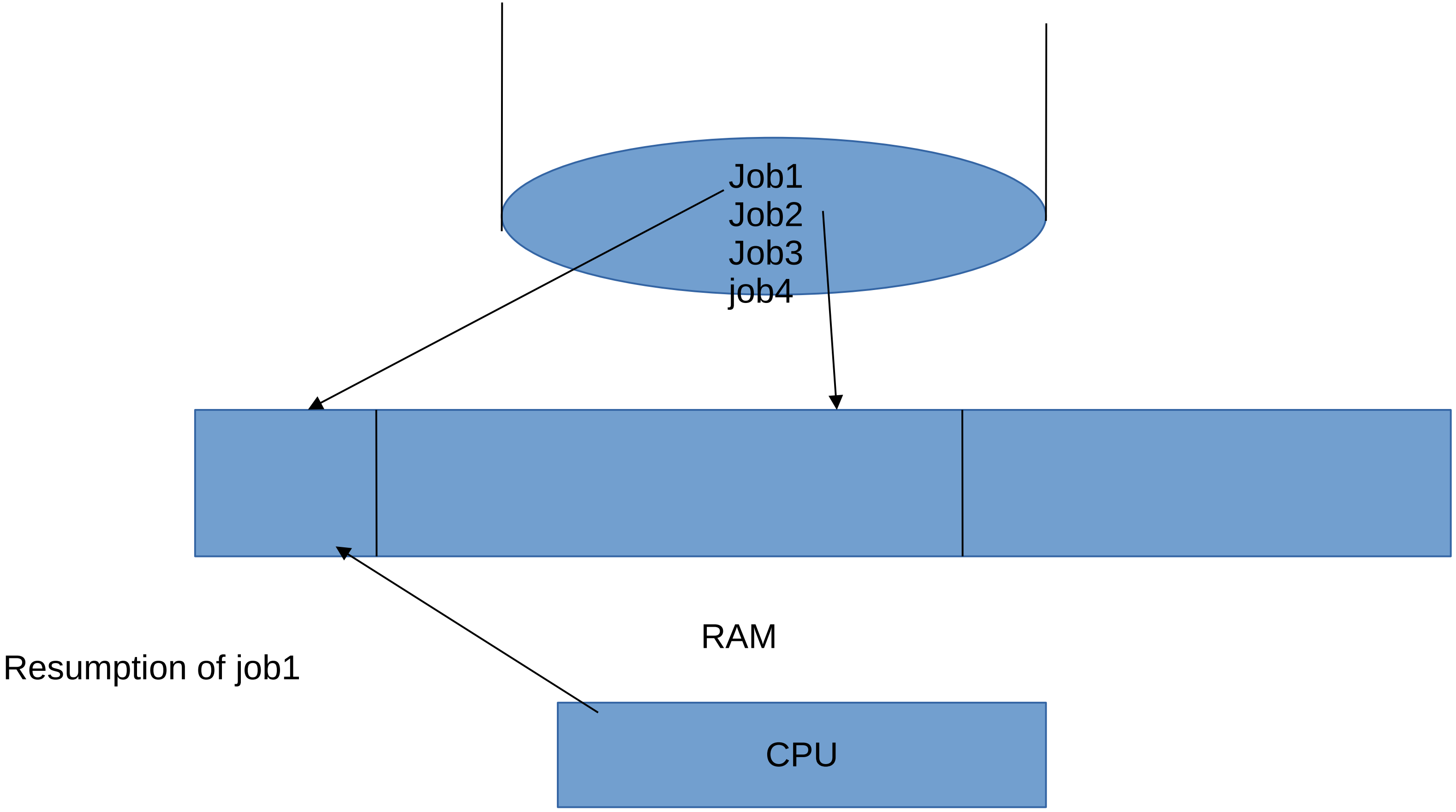
Creating a Process for execution

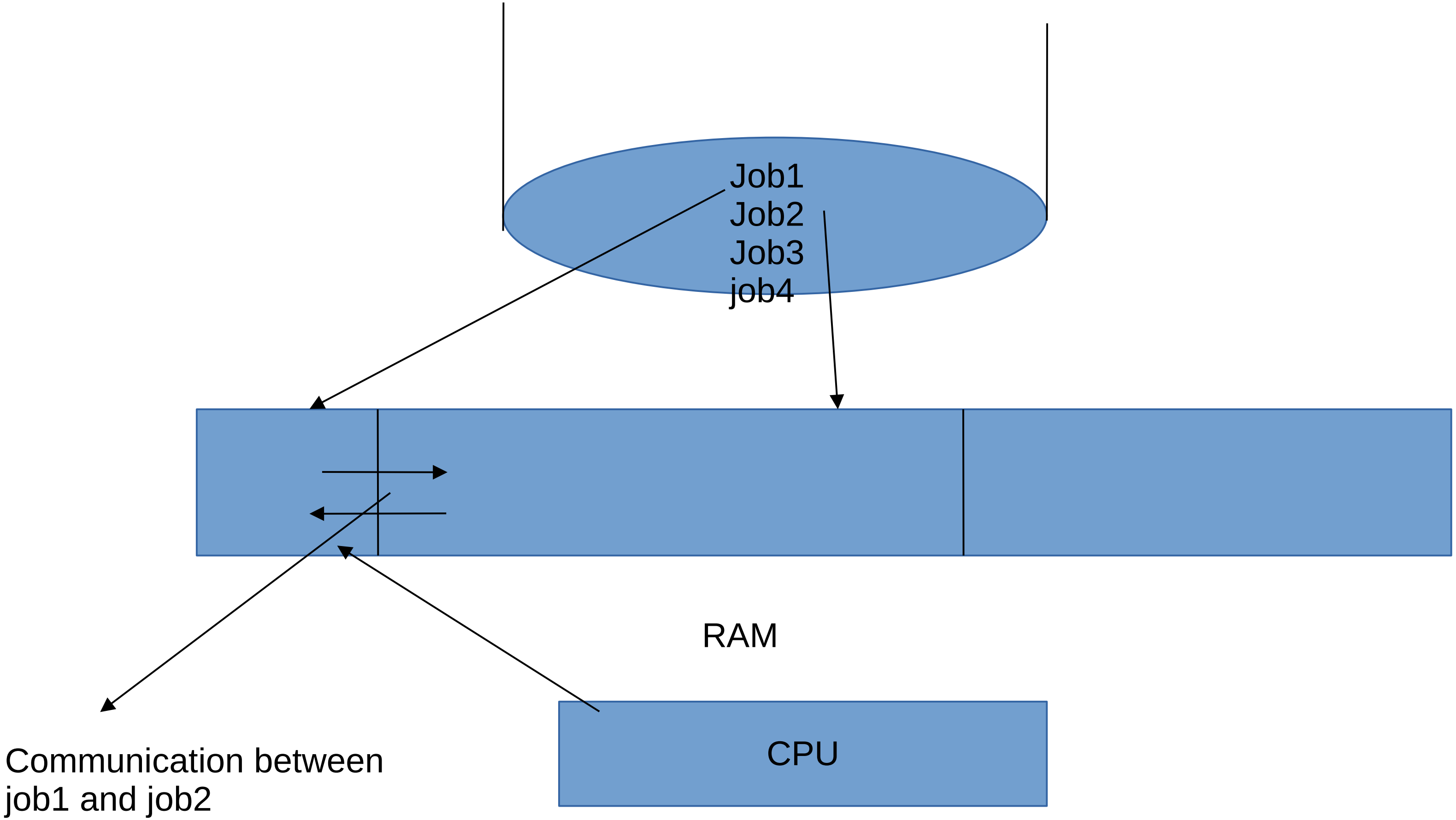



Job1
Job2
Job3
job4
When job1 finish execution deletion of job 1 from RAM
RAM
CPU

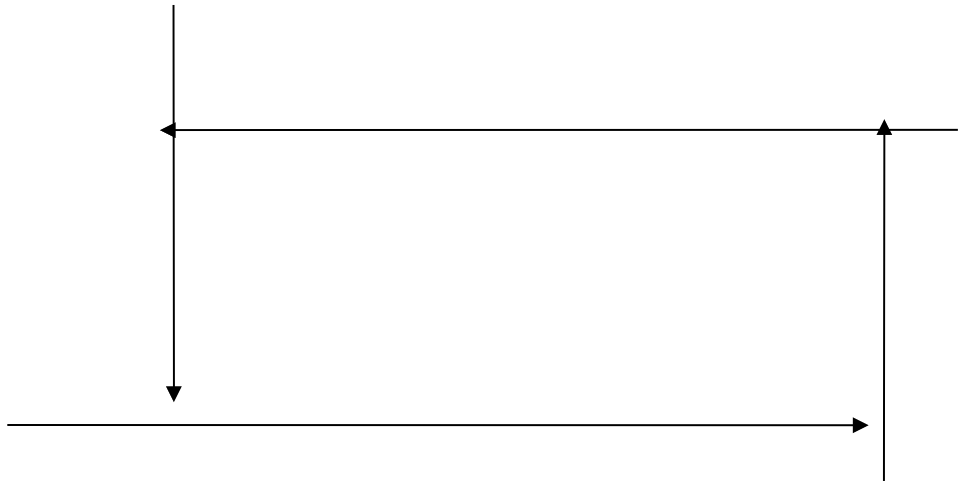

Job1
Job2
Job3
job4
RAM
CPU


Job1
Job2
Job3
job4
RAM
Suspension of job1
CPU


Job1
Job2
Job3
job4
RAM
Resumption of job1
CPU


Job1
Job2
Job3
job4
RAM
CPU
Communication between
job1 and job2

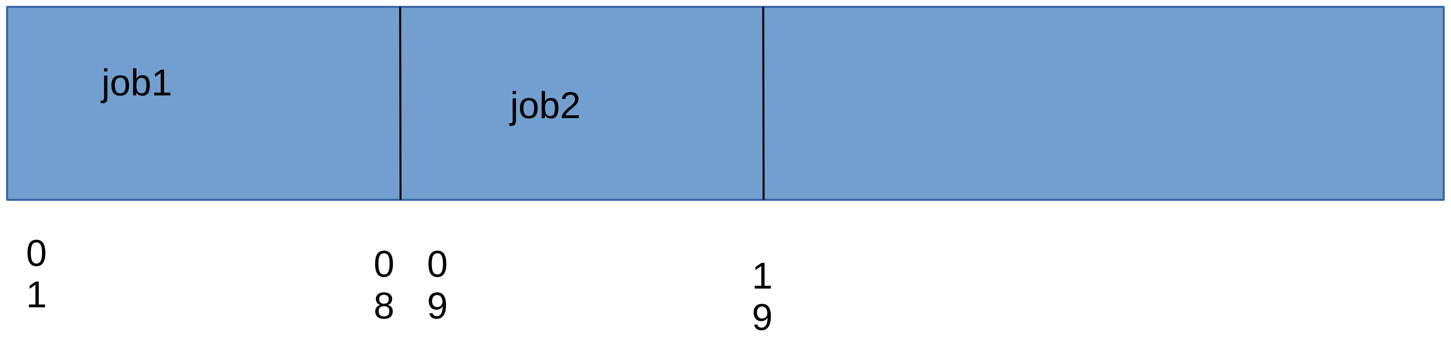
DEADLOCK????



OS Components

2. Main memory management:

- Keep track of which part of the memory are currently being used by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.



	From	To
Job1	01	08
Job2	09	19

When job2 finishes

	From	To
Job1	01	08
Free	09	99

When job2 and job4 finishes

	From	To
Job1	01	08
Job2	09	20
Job3	21	31
Job4	32	99

When job2 and job4 finishes

	From	To
Job1	01	08
Job3	21	31
free	09	20
	32	99

New jobn is ready and it requires
10 bytes to fit

	From	To
Job1	01	08
Job3	21	31
free		
	09	20
	32	99

3. File Management

- The creation and deletion of files.
- The creation and deletion of directories.
- Modifying a file.
- Mapping of files into secondary storage.

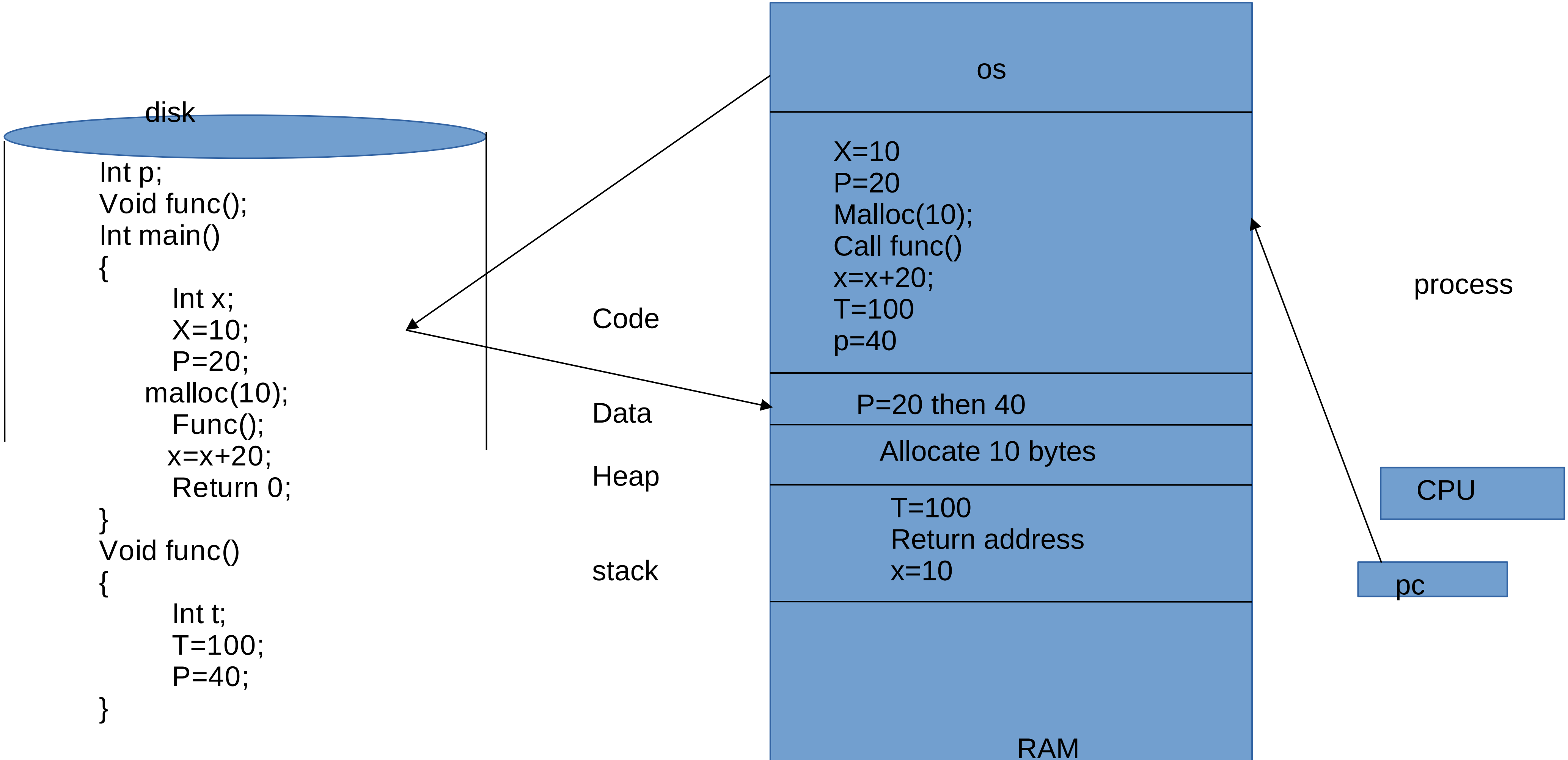
I/O system management

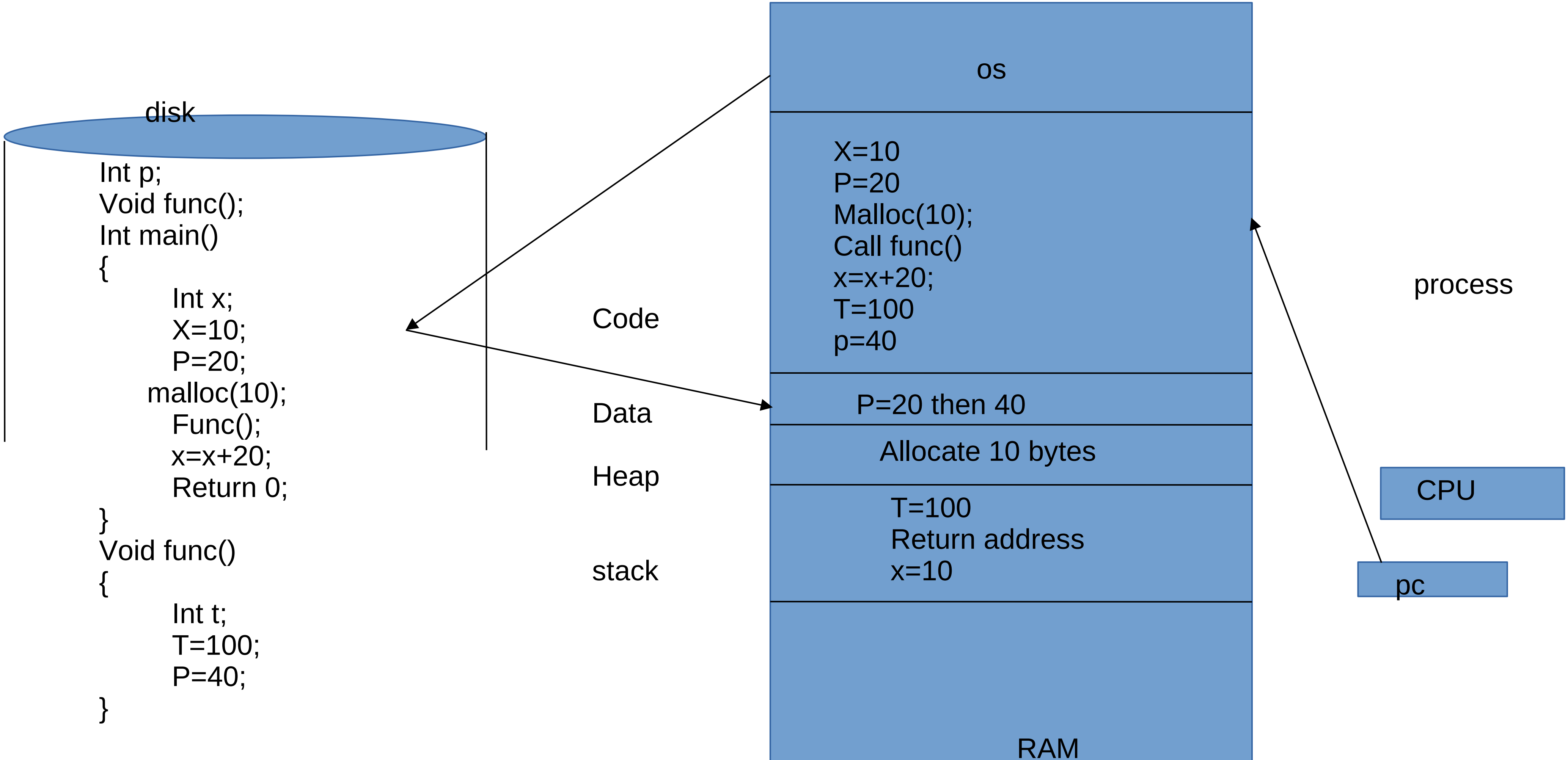
- General device driver interface.
- Drivers for specific hardware devices.
- If a printer is shared between more than one machines then buffering of document to be print.

- Networking
- Secondary storage management.
- Protection system.
- Command line interpreter.

OS services

- Program execution.
- I/O operation.
- File system manipulation.
- Communication.
- Error detection.
- Resource allocation.
- Accounting.
- Protection.


disk
Int p;
Void func();
Int main()
{
Int x;
X=10;
P=20;
malloc(10);
Func();
x=x+20;
Return 0;
}
Void func()
{
Int t;
T=100;
P=40;
}
Code
Data
Heap
stack
os
X=10
P=20
Malloc(10);
Call func()
x=x+20;
T=100
p=40
P=20 then 40
Allocate 10 bytes
T=100
Return address
x=10
RAM
process
CPU
pc


disk
Int p;
Void func();
Int main()
{
Int x;
X=10;
P=20;
malloc(10);
Func();
x=x+20;
Return 0;
}
Void func()
{
Int t;
T=100;
P=40;
}
Code
Data
Heap
stack
os
X=10
P=20
Malloc(10);
Call func()
x=x+20;
T=100
p=40
P=20 then 40
Allocate 10 bytes
T=100
Return address
x=10
RAM
process
CPU
pc

- Os processes execute system code and user processes execute user code.

- The term job and process are used interchangeably.

- Two processes may be associated with the same program. But they are two separate execution sequence.

PROCESS STATE

- New:

A process is being created.

- Running:

Instructions are being executed.

- Waiting:

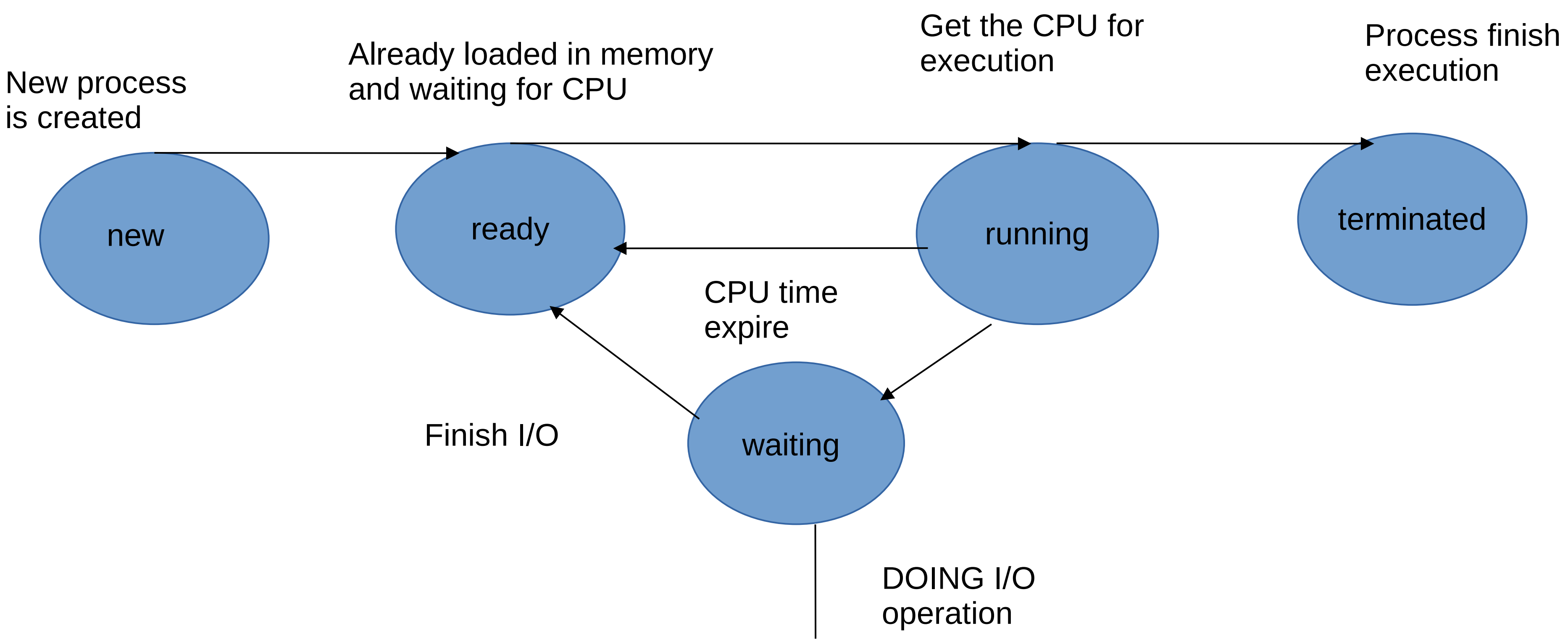
The process is waiting for some event to occur (such as an I/O completion).

- Ready:

The process is waiting to be assigned to a processor .

- Terminated:

The process has finished execution.

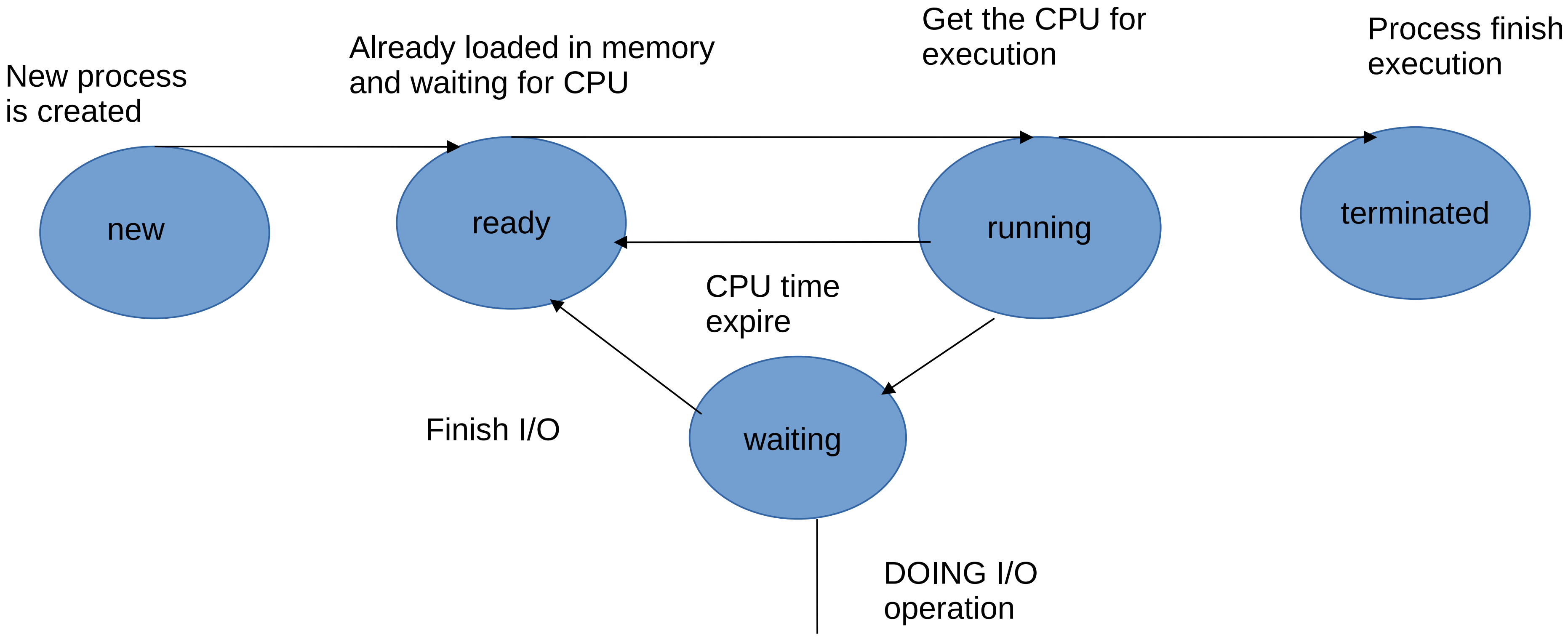

New process
is created
Already loaded in memory
and waiting for CPU
Get the CPU for
execution
Process finish
execution
new
ready
running
terminated
CPU time
expire
Finish I/O
waiting
DOING I/O
operation

Process Control Block

Each process is represented in the OS by a process control block(PCB).

It contains many peace of information:

- Process state: new/ready/running/......
- Program counter information: Address of the next instruction to be execute.
- CPU registers: Registers may be very in numbers.
- CPU scheduling information: Process priority.
- Memory management information: This may include the value of the base and limit address and other informations: page table, segment table etc.
- Account information: This includes the amount of CPU and real time used, process number etc..
- I/O status information: This includes the list of I/O devices allocated to this process.

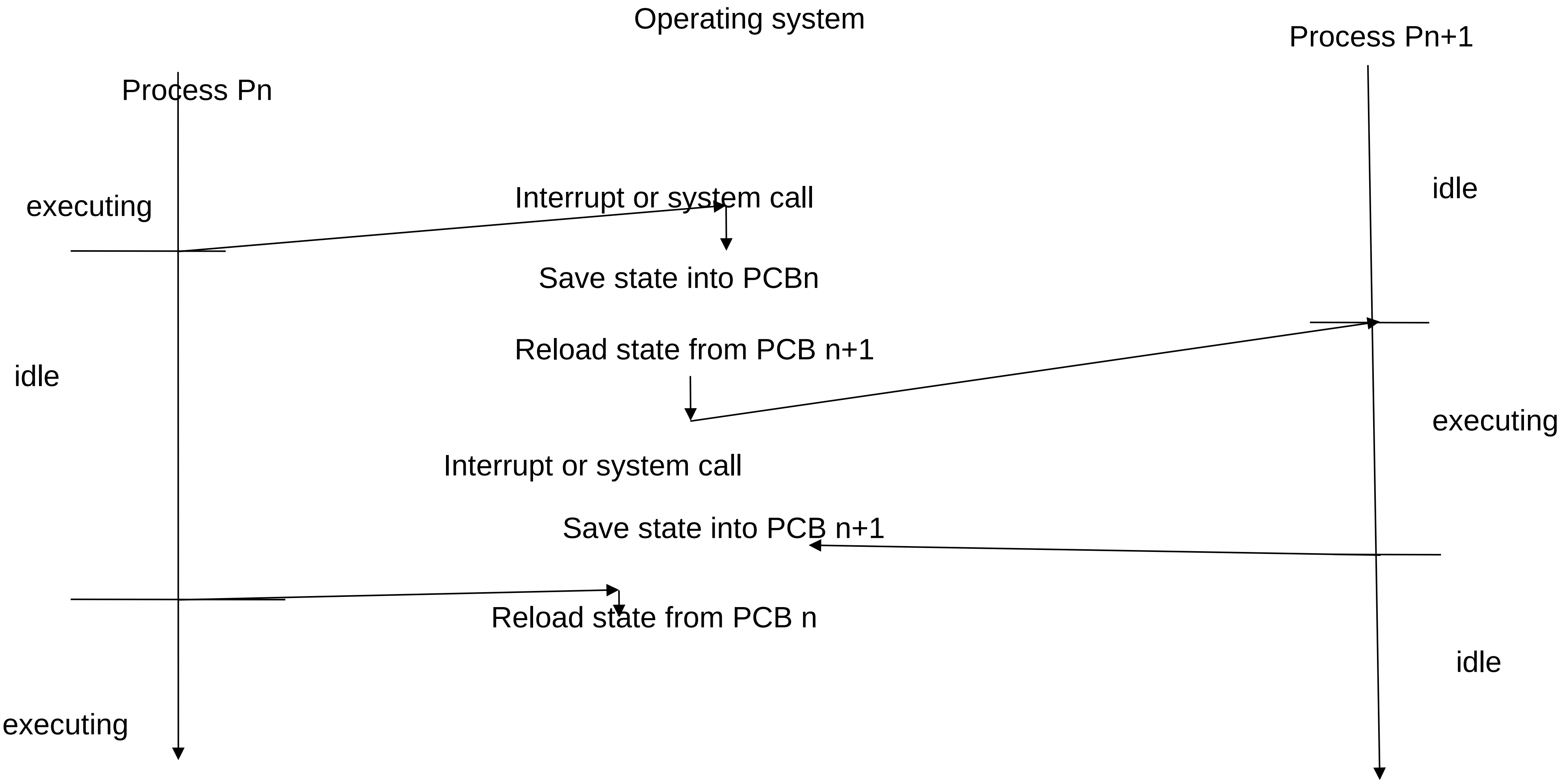

New process
is created
Already loaded in memory
and waiting for CPU
Get the CPU for
execution
Process finish
execution
new
ready
running
terminated
CPU time
expire
Finish I/O
waiting
DOING I/O
operation

Process Control Block

Each process is represented in the OS by a process control block(PCB).

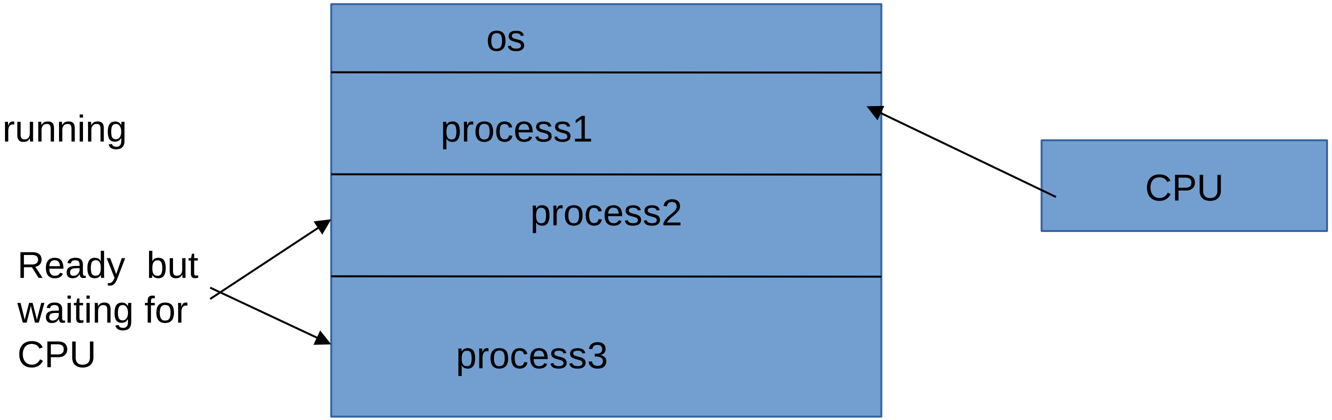
It contains many peace of information:

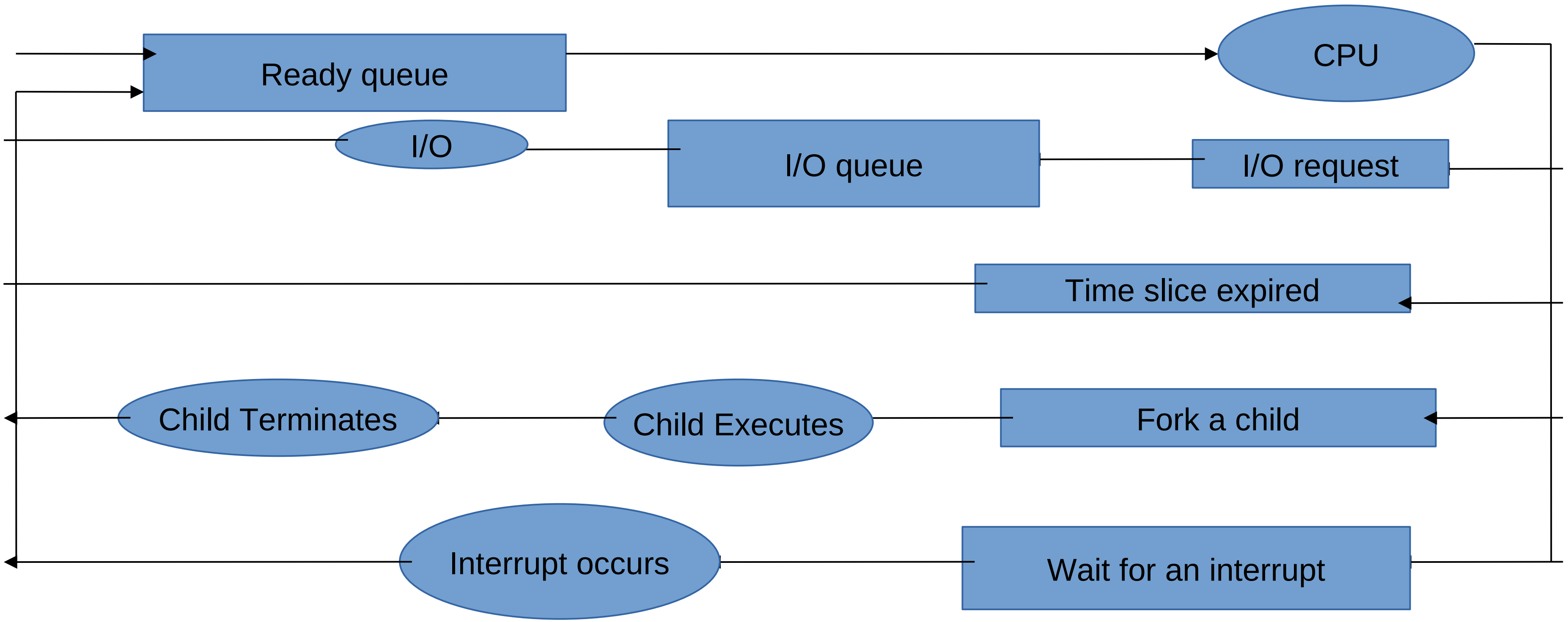
- Process state: new/ready/running/......
- Program counter information: Address of the next instruction to be execute.
- CPU registers: Registers may be very in numbers.
- CPU scheduling information: Process priority.
- Memory management information: This may include the value of the base and limit address and other informations: page table, segment table etc.
- Account information: This includes the amount of CPU and real time used, process number etc..
- I/O status information: This includes the list of I/O devices allocated to this process.


Operating system
Process Pn+1
Process Pn
idle
executing
Interrupt or system call
Save state into PCBn
Reload state from PCB n+1
idle
executing
Interrupt or system call
Save state into PCB n+1
Reload state from PCB n
idle
executing

Process Scheduling

- For a uniprocessor system there will never be more than one running process.
- If there are more processes, the rest have to wait until the CPU is free and can be rescheduled.




Ready queue
CPU
I/O
I/O queue
I/O request
Time slice expired
Child Terminates
Child Executes
Fork a child
Interrupt occurs
Wait for an interrupt

Job queue: As processes enter into the system, they are kept in this queue.

Ready queue: This queue consists of all processes that are ready and waiting to be executed.

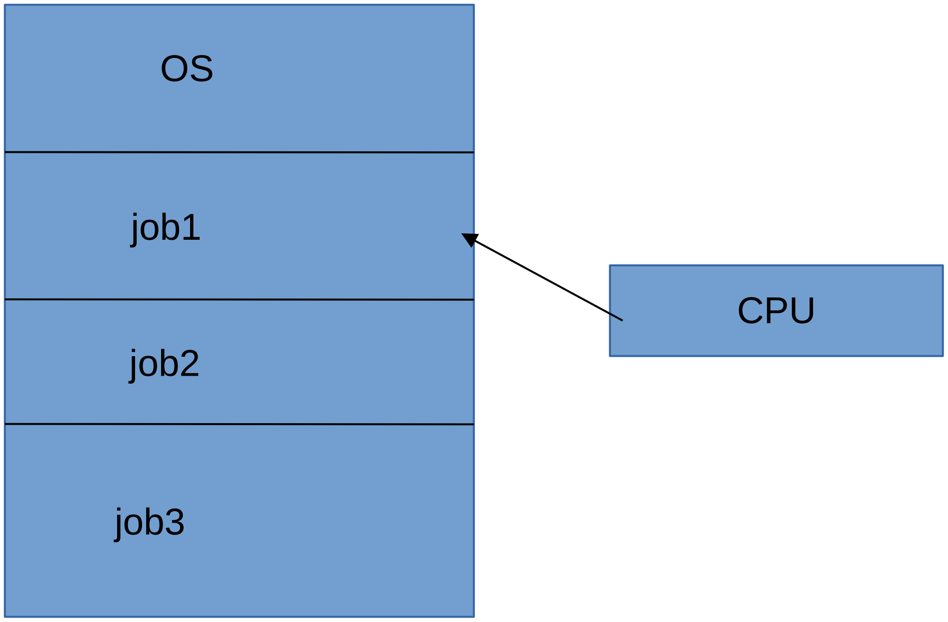
Device queue: This queue consist of all processes waiting for the particular I/O device. Thus there is a device queue for each device.

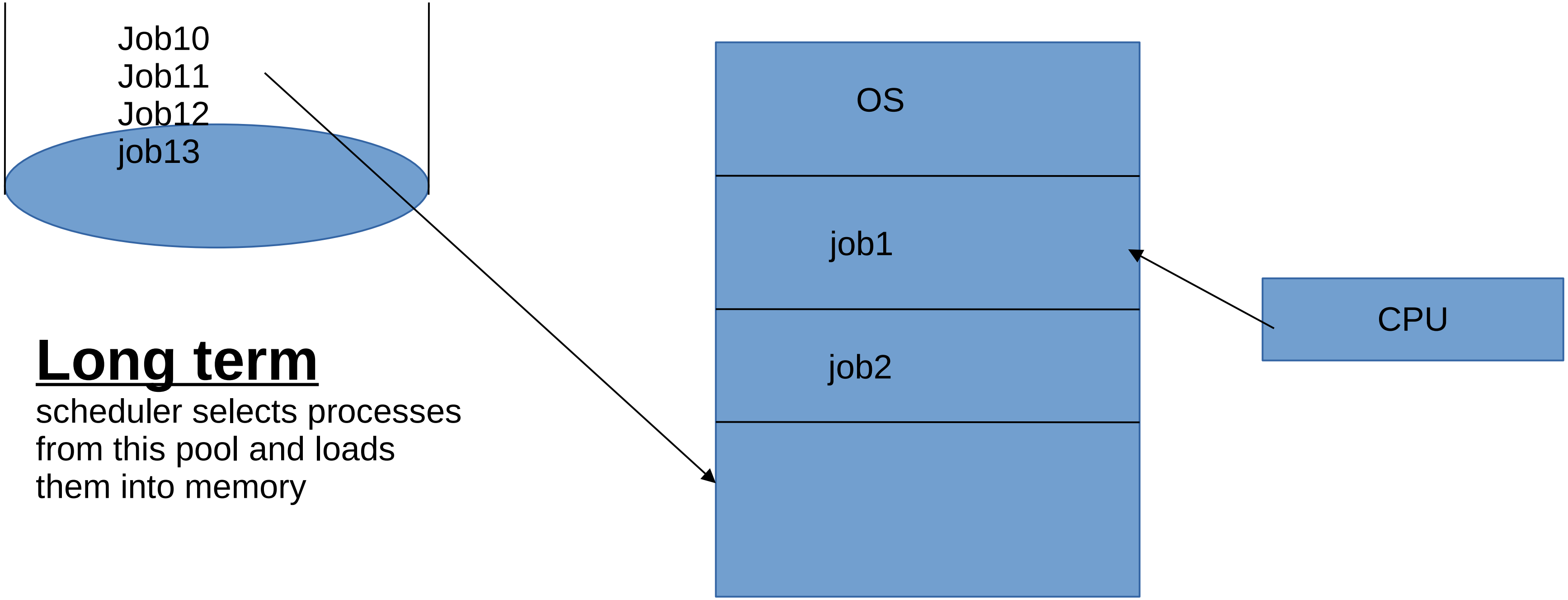
Once a process is executing one of the following events can occur

1. The process could issue an I/O request and then be placed in an I/O queue.
2. The CPU time expire(2 sec for each process).
3. The Process could create a new sub process and wait for its termination.
4. The process could be removed forcibly from the CPU as a result of an interrupt and be put back in the ready queue.

There are sometimes more processes submitted than can be executed immediately.

These processes are kept in a storage device (usually a disk) for later execution.

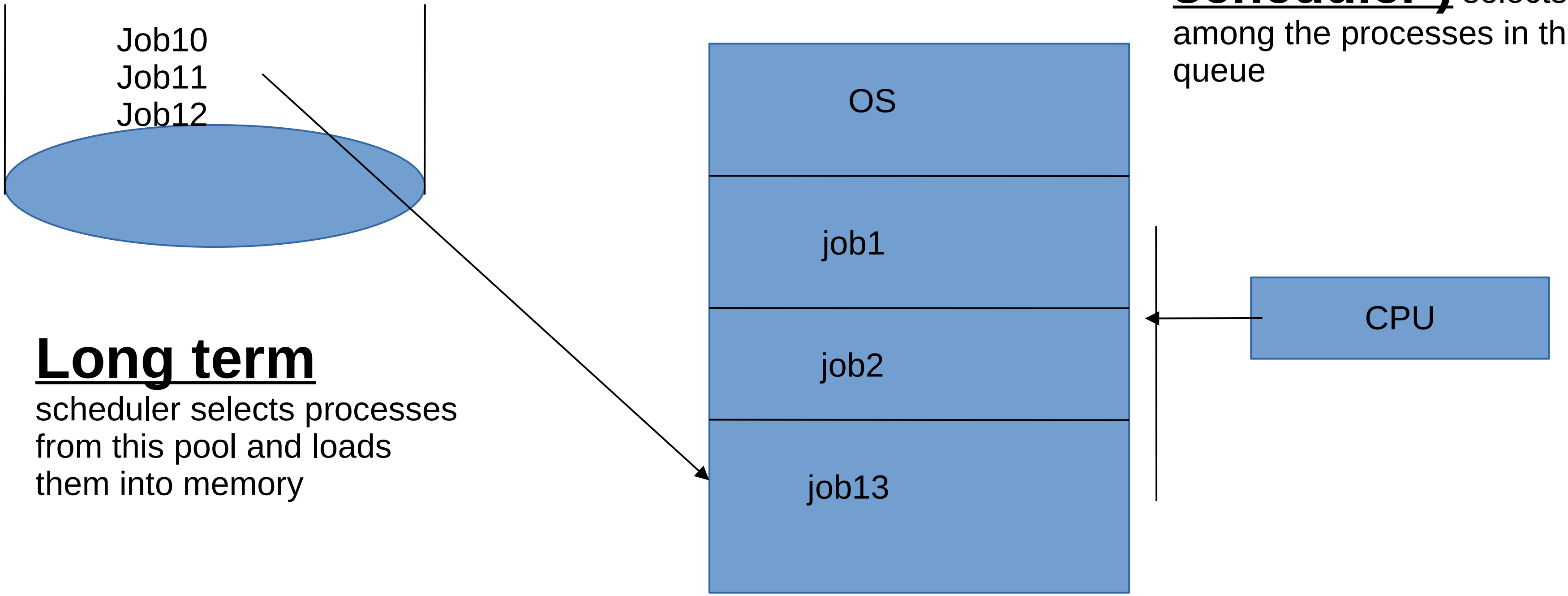




Long term

scheduler selects processes from this pool and loads them into memory

Short term Scheduler(CPU scheduler) selects from among the processes in the ready queue



Long term scheduler selects processes from this pool and loads them into memory

. Long term scheduler executes less frequently than the short term scheduler.

. Long term scheduler controls the degree of multiprogramming.

I/O and CPU bound process

```
#include<stdio.h>
Int main()
{
    Int p;
    P=10;
    printf(“\n initial value=%d”,p);
    While(1)
    {
            p=p+1;
            }
          •   }
```

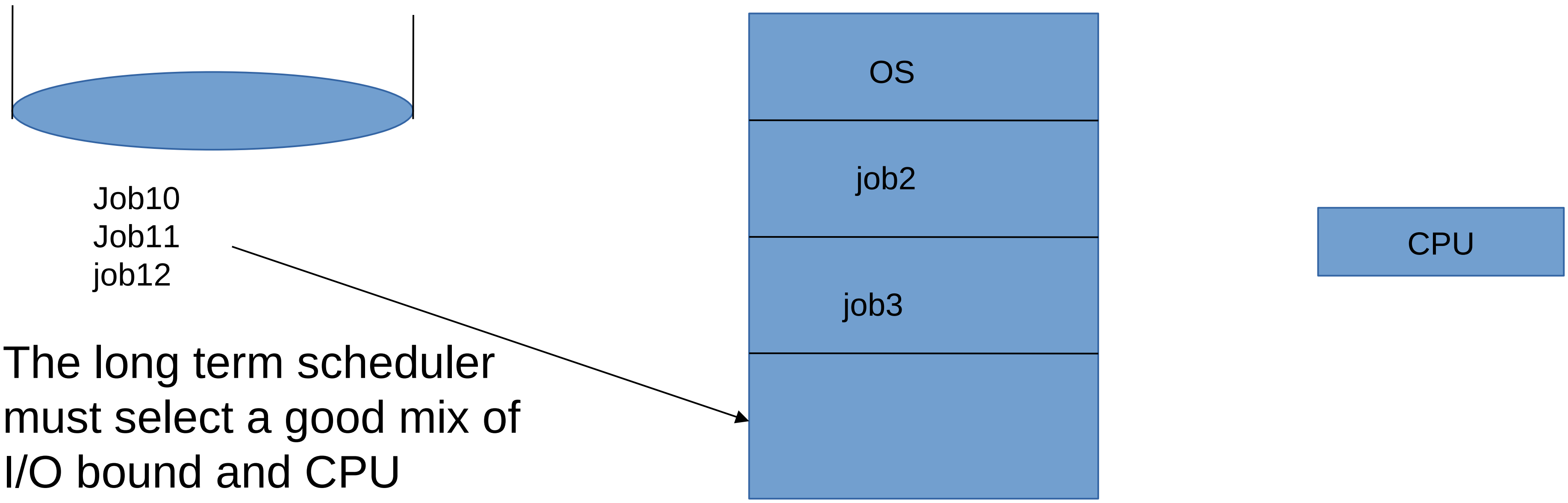
```
#include<stdio.h>
 int main()
   {
       int q;
         p=100;
             while(1)
                   {
                         printf(“\n hello”);
                   }
               }
```

- If all processes are CPU bound processes then the I/O devices sit idle.

- If all processes are I/O bound processes then the CPU sit idle.

- Utilization Problem.

The long term scheduler is responsible for solving this problem.

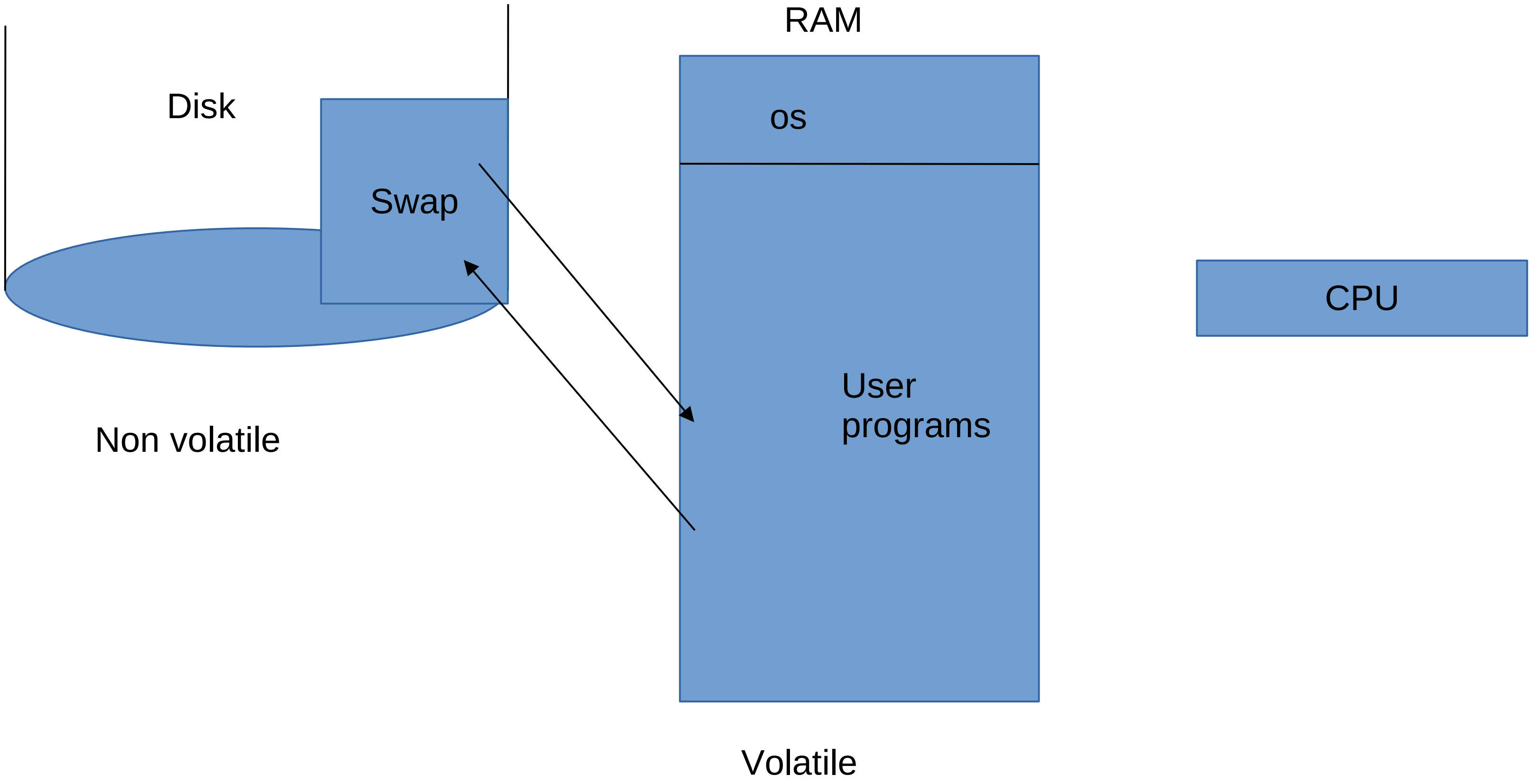


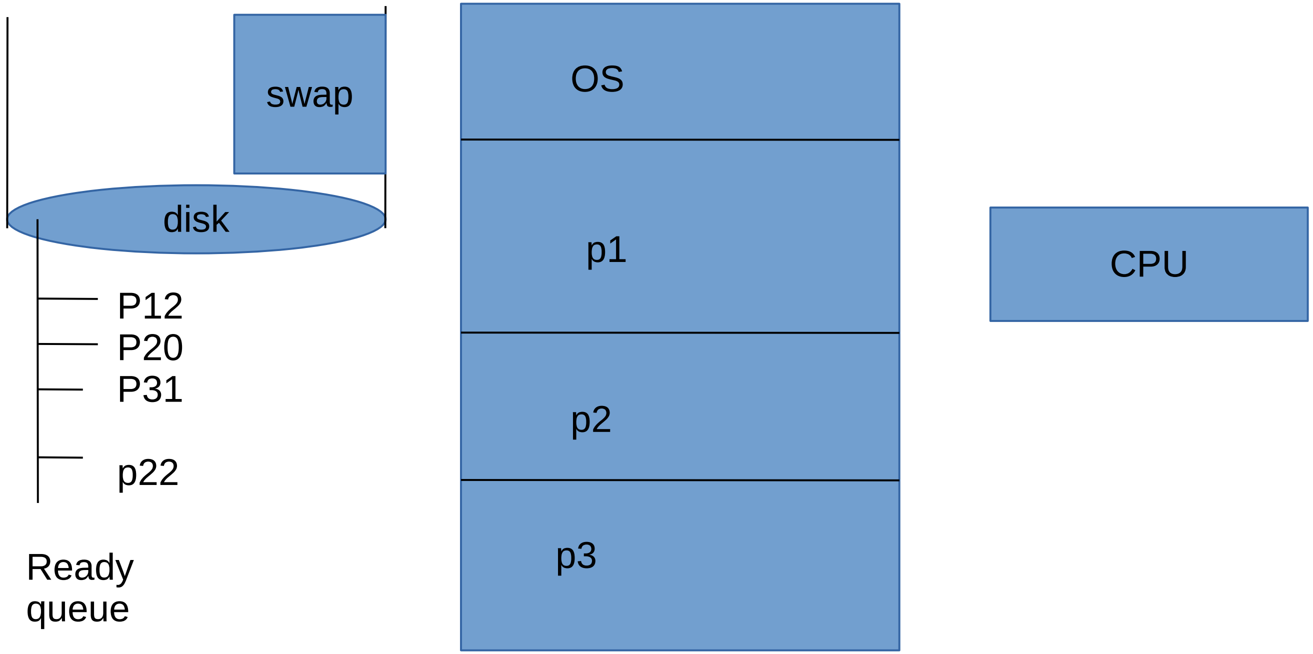
The long term scheduler must select a good mix of I/O bound and CPU bound processes.

Mid Term Scheduler

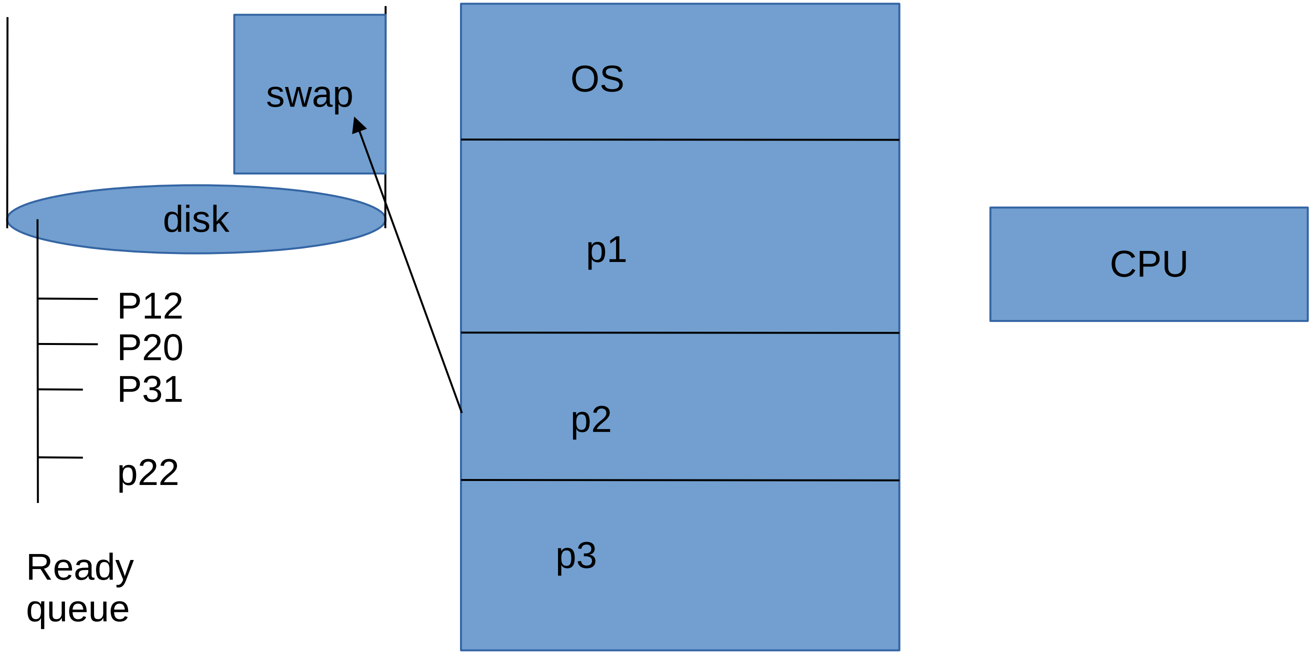
- Swapping

Hibernate





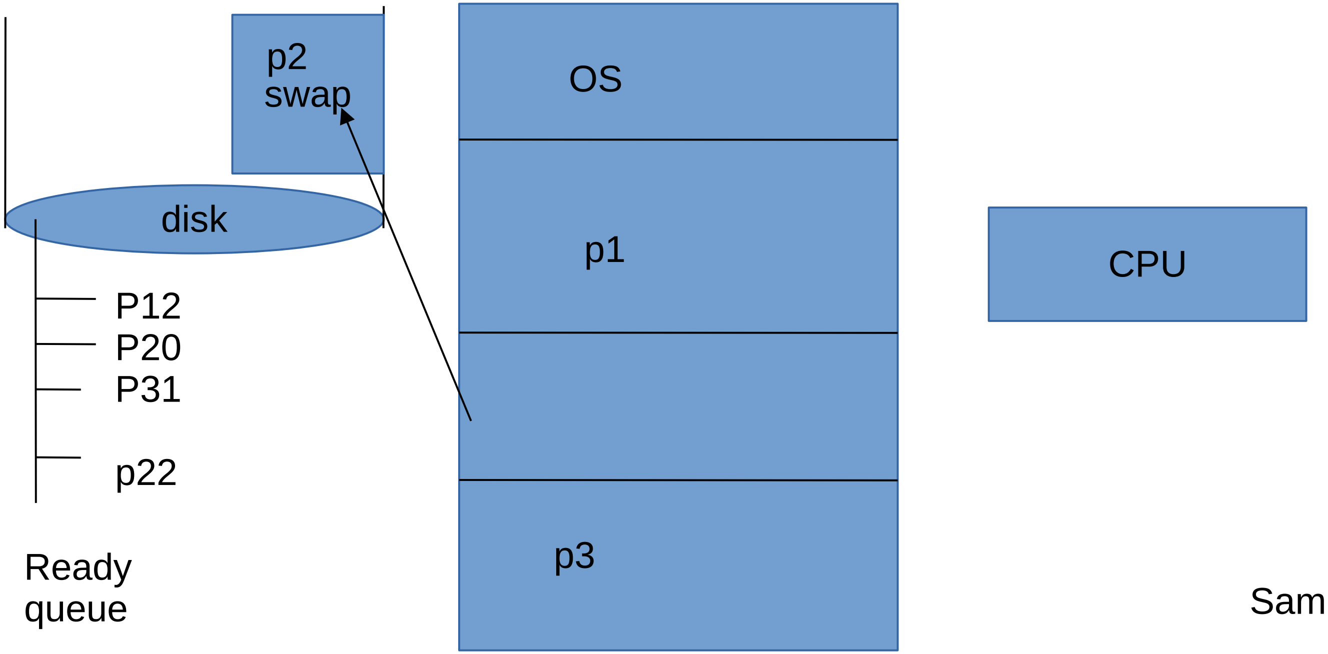

```c
#include<stdio.h>
Int main()
{
        Int x=0;
        x=x+1;
        While(x<10)
        {
                printf("\n %d",x);
                x=x+1;
        }
        Return 0;
}
```




```
#include<stdio.h>
Int main()
{
        Int x=0;
        x=x+1;
        While(x<10)
        {
                printf("\n %d",x);
                x=x+1;
        }
        Return 0;
}
                p2
```

When p2 is doing I/O then, is p2 required to be in main memory?

NO



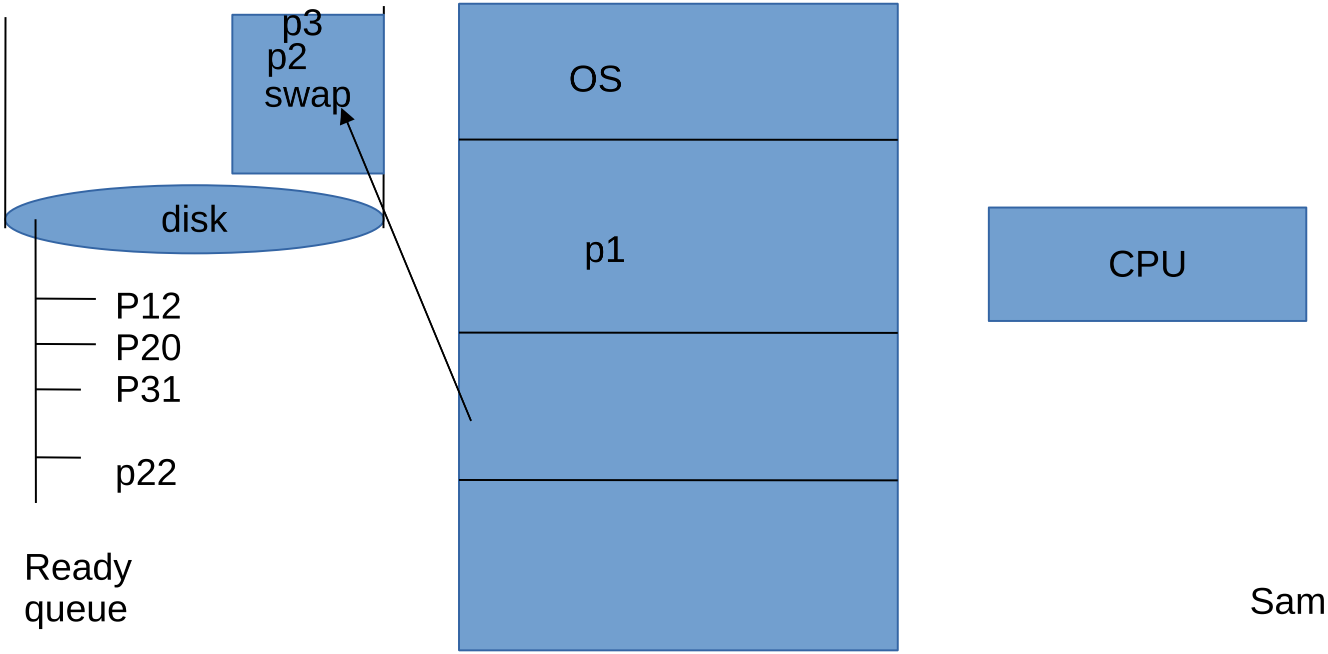

```
#include<stdio.h>
Int main()
{
        Int x=0;
        x=x+1;
        While(x<10)
        {
                printf("\n %d",x);
                x=x+1;
        }
        Return 0;
}
```

p2

When p2 is doing I/O then, is p2 required to be in main memory?

NO

Same thing might happened to p3 also



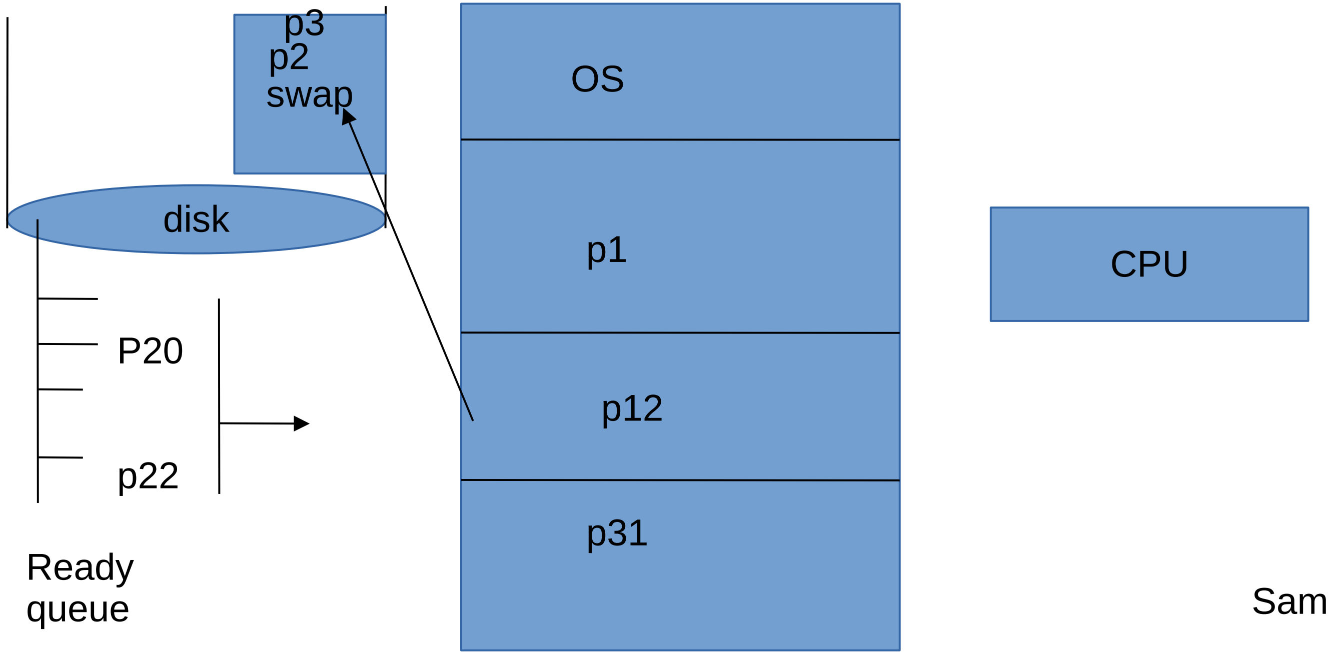

```
#include<stdio.h>
Int main()
{
        Int x=0;
        x=x+1;
        While(x<10)
        {
                printf("\n %d",x);
                x=x+1;
        }
        Return 0;
}
```

p2

When p2 is doing I/O then, is p2 required to be in main memory?

NO

Same thing might happened to p3 also




```
#include<stdio.h>
Int main()
{
        Int x=0;
        x=x+1;
        While(x<10)
        {
                printf("\n %d",x);
                x=x+1;
        }
        Return 0;
}
```

p2

When p2 is doing I/O then, is p2 required to be in main memory?

NO

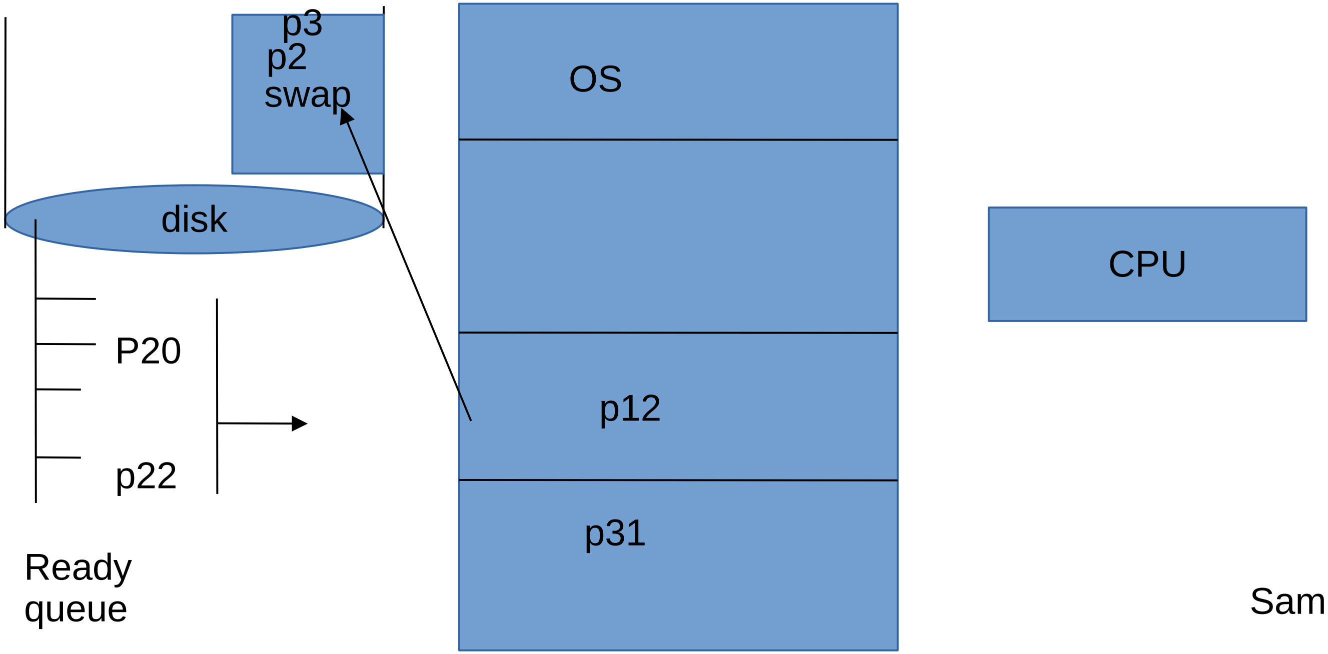
Same thing might happened to p3 also

In the free memory only one can be fitted.. who will get the chance to be in main memory?

By the time p1 finish its operation and both p2 and p3 finish I/O..

```
#include<stdio.h>
Int main()
{
        Int x=0;
        x=x+1;
        While(x<10)
        {
                printf("\n %d",x);
                x=x+1;
        }
        Return 0;
}
```

p2



When p2 is doing I/O then, is p2 required to be in main memory?

NO

Same thing might happened to p3 also

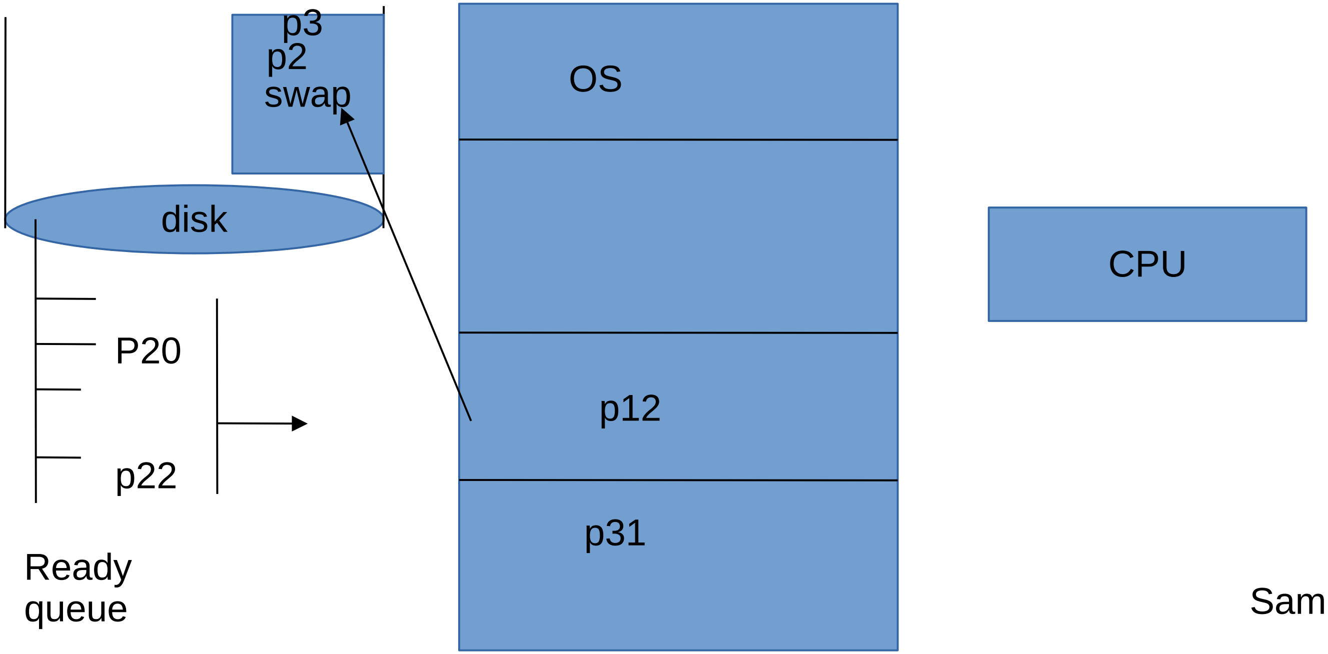
Middle term scheduler

In the free memory only one can be fitted.. who will get the chance to be in main memory?

By the time p1 finish its operation and both p2 and p3 finish I/O..

```
#include<stdio.h>
Int main()
{
        Int x=0;
        x=x+1;
        While(x<10)
        {
                printf("\n %d",x);
                x=x+1;
        }
        Return 0;
}
```

p2



When p2 is doing I/O then, is p2 required to be in main memory?

NO

Same thing might happened to p3 also

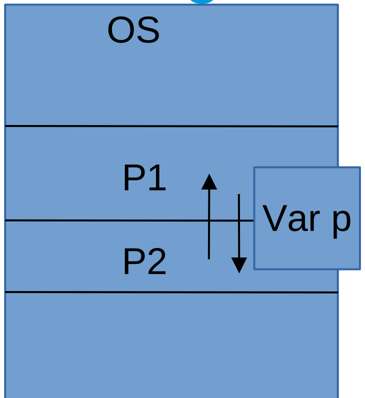
Process synchronization

A Co-operating process is one that can effect or be effected by the other co-operating processes executing in the system.

Co-operating process often share some common storage that can be read or write by both processes.

Storage may be a variable or a file ….

Concurrent access of the storage may result in data inconsistency.



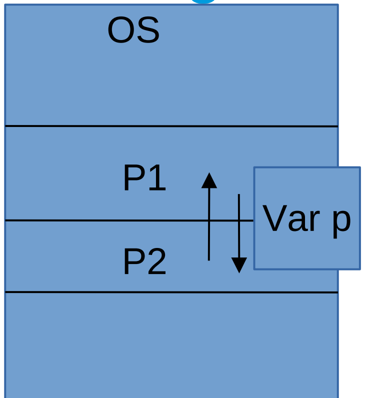
Process synchronization

A Co-operating process is one that can effect or be effected by the other co-operating processes executing in the system.

Co-operating process often share some common storage that can be read or write by both processes.

Storage may be a variable or a file ….

Concurrent access of the storage may result in data inconsistency.



The Producer and consumer Problem

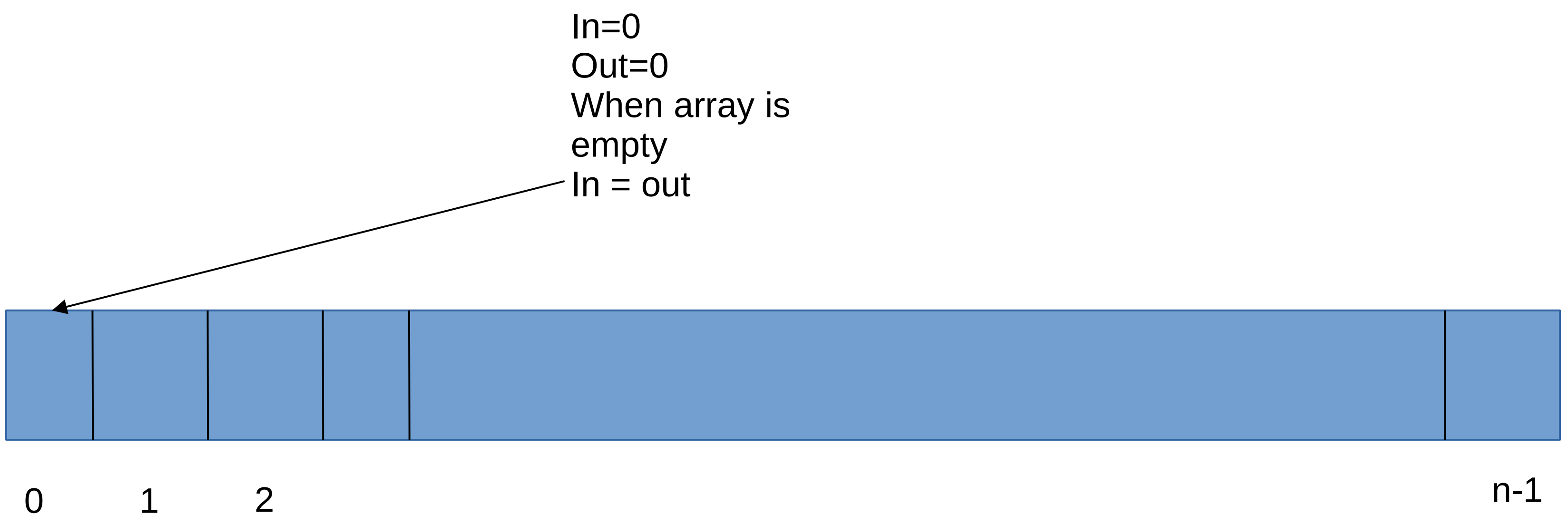
Both producer and consumer processes share the following variables:

- var n;
- type item=....
- var buffer: array[0..n-1] of item //implemented as circular array with 2 logical pointers in and out
- in,out: 0 … n-1 //initialized to 0

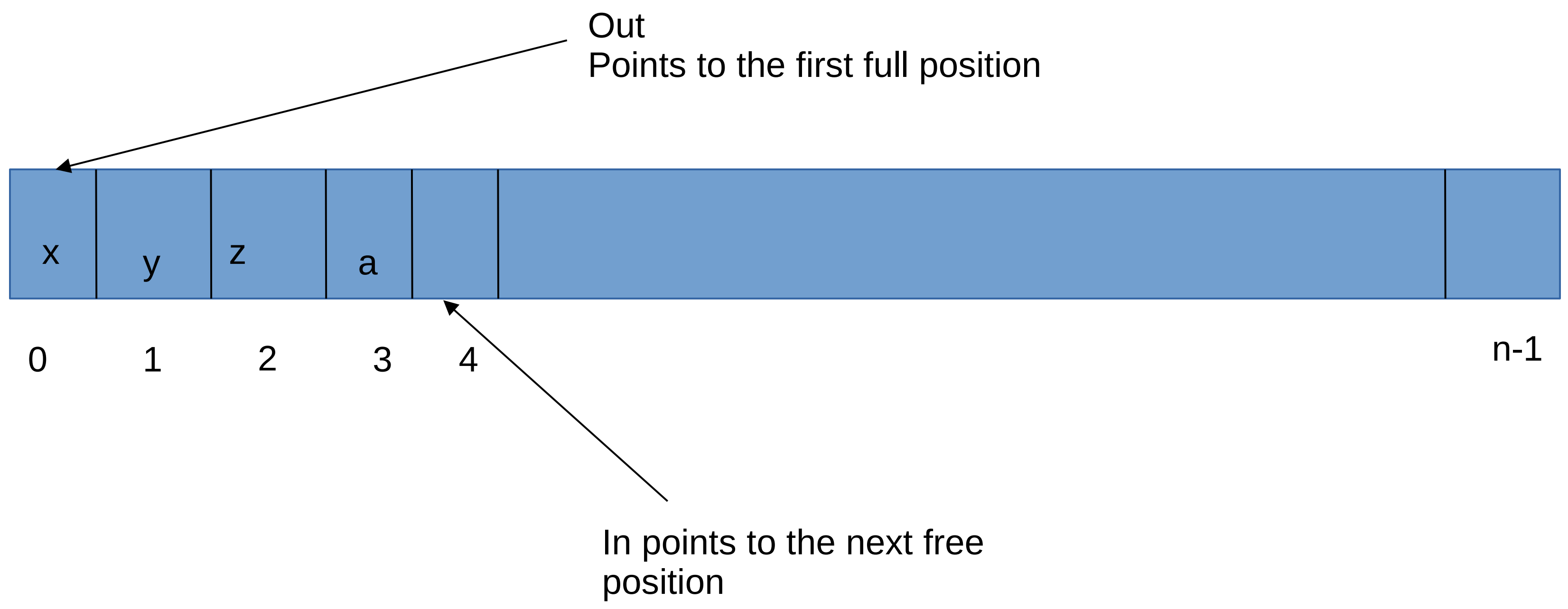
in points to next free position in the buffer and out points to the first full position in the buffer.

The buffer is empty when in =out;

The buffer is full when (out+1)mod n= in;


In=0
Out=0
When array is
empty
In = out
0
1
2
n-1

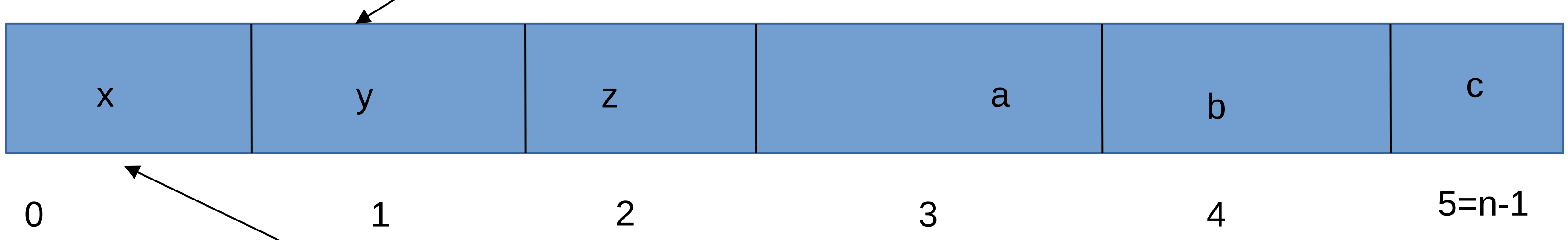
In=0
Out=4


Out
Points to the first full position
x
y
z
a
0
1
2
3
4
n-1
In points to the next free
position

In=0
Out=4

Suppose n= 6

Out
Points to the first full position



In points to the next free position

When the array is full
(in+1)mod n=out
(o+1)mod 6=1

The no-op is a do nothing instruction
Nextp is a local variable in which the new item produces is stored .

Code for producer

Repeat…….

Produce an item in nextp……

While (in+1)mod n=out do no-op//while buffer is full

buffer[in]=nextp

in=(in+1)mod n;

Until false

Nextc is a local variable in which the item to be consumed is stored.

Repeat

While in=out do no-op; //while buffer is empty

nextc=buffer[out];

out=(out+1)mod n;

…….

Consume the next item in nextc;

Until false

This algorithm allows at most n-1 items in the buffer at the same time.

Some one need to count the number of items present in the array at a particular time.

Use a counter variable by both processes.

- Every time a new item is added counter is incremented by 1.
- Every time an item is removed from the array counter is decremented by 1.

Code for producer

Repeat…….

Produce an item in nextp……

While (in+1)mod n=out do no-op//while buffer is full

buffer[in]=nextp

in=(in+1)mod n;

counter=counter+1;

Until false

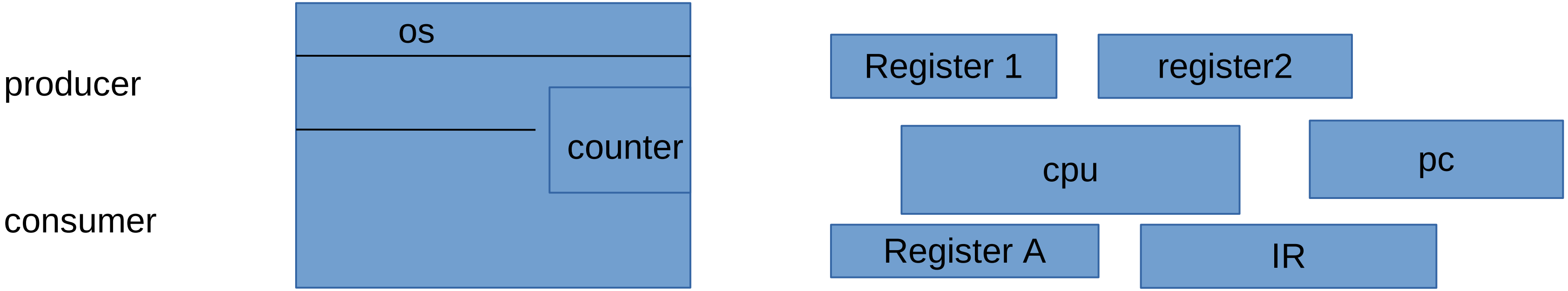
Consumer code

```
Repeat
While in=out do no-op; //while buffer is empty
nextc=buffer[out];
out=(out+1)mod n;
counter=counter-1;
…….
Consume the next item in nextc;
Until false
```

counter=counter+1; counter=counter-1;

In machine language that statements are implemented as:

register1=counter;	register2=counter;
register1=register1+1;	register2=register2—1;
counter=register1;	counter=register2;



Concurrent execution of producer and consumer

Remember CPU switch from one job to another in multi tasking? (2 sec for each process).......

Lets begin …. assume counter = 4 at that moment and a new item is added first then another consumed next....

T0: producer: execute : register1=counter [register1=4]

T1: producer: execute : register1=register1+1 [register1=5]

CPU switches from producer to consumer

T2: consumer: execute: register2=counter [register2=4]

T3: consumer: execute: register2=register2-1 [register2=3]

CPU switches from consumer to producer

T4: Producer : execute: counter=register1 [counter= 5]

T5: one more statement in producer executes

CPU switches from producer to consumer

T6: consumer : executes : counter=register2 [counter=3]

T7: consumer executes one more statement...

We have come to an incorrect state counter=3, the correct is counter=4;

BUT WHY?

We allow both the processes to manipulate the shared variable counter concurrently

A situation where more than one processes access the same variable simultaneously and the outcome of the execution depends on the particular order in which the access takes place is called a RACE CONDITION

Solution?

Ensure that only one process at a time can access/ manipulate the shared variables/files.....

So some kind of synchronization required. That is known as process synchronization.

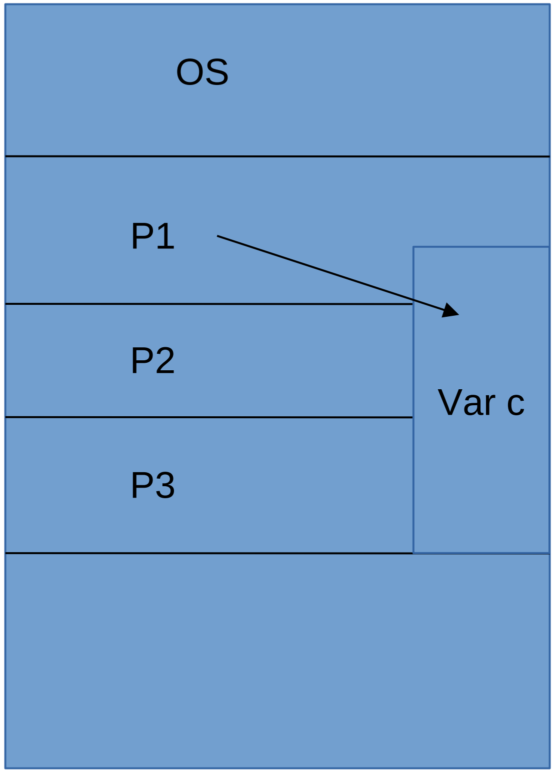
Since OS shares different resources so it should be taken care by the OS.

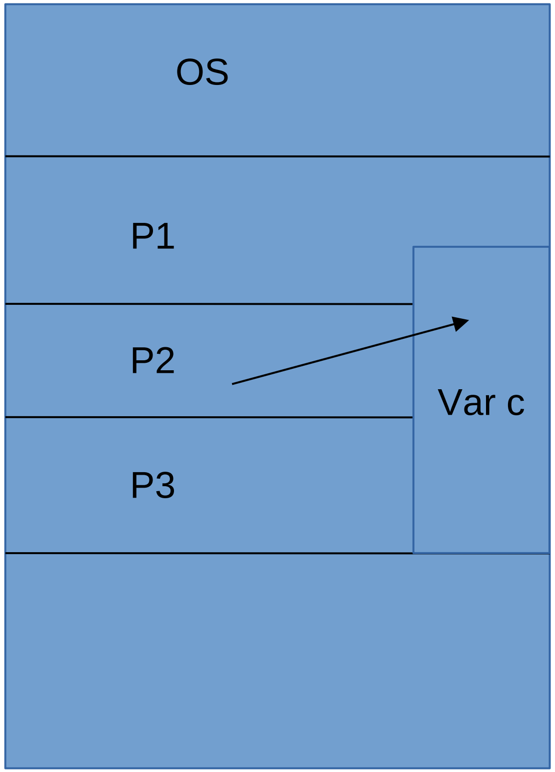
The Critical Section Problem

. Consider a system consisting of n processes .

. Each process has a segment code called critical section in which the process may be changing common variables, updating a table, writing a file and so on.

. When a process is executing in its critical section no other process is allowed to execute in its critical section.


OS
P1
P2
Var c
P3



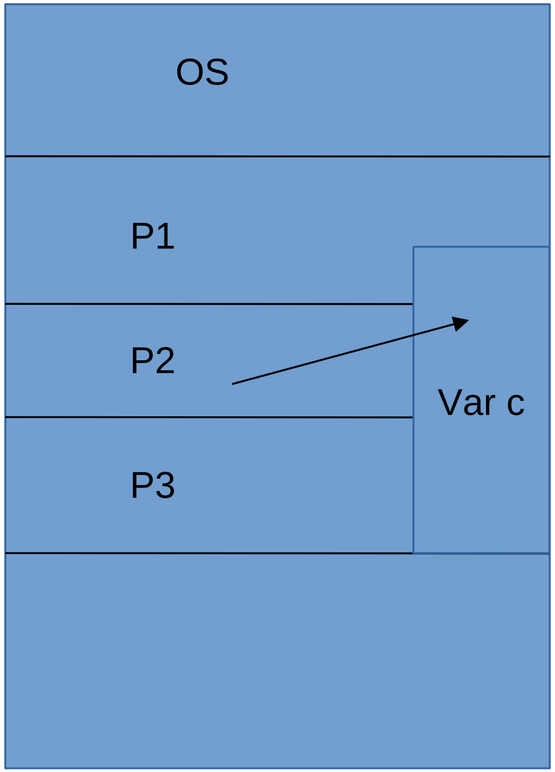
OS
P1
P2
P3
Var c

Each process must request permission to enter its critical section.

This section of code is known as entry section.

The critical section is followed by exit section.

Process P2 want to enter the CS



Entry section (To take the permission to enter in to the CS and it ensures that no one is in CS)

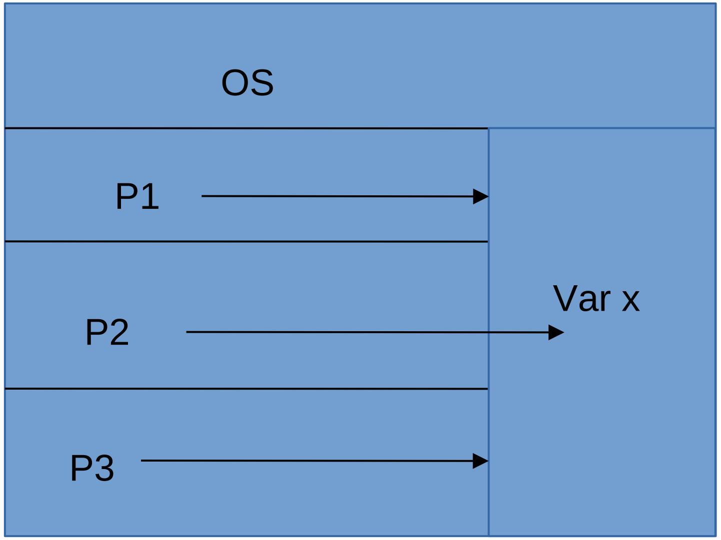
c=c+1;

Exit section (To release the control of CS to others, so that others can enter the CS)

Remainder section

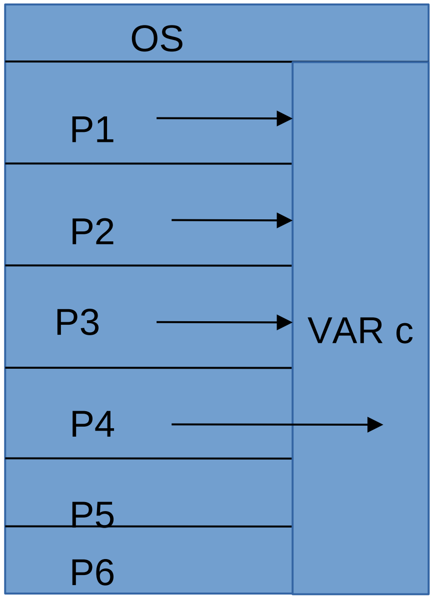
Mutual Exclusion

If a process P1 is executing in its Critical section then no other process can be executing in its CS.



Progress

If no process is executing in its cs and there exist some processes that wish to enter their cs then only those processes that are not executing in their remainder section can participate in the decision of which will enter its CS next and this selection can not be postponed indefinitely.



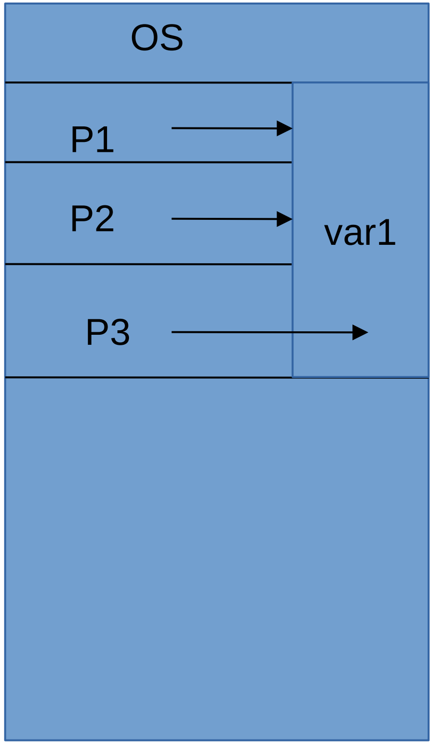
P5 and P6 finish executing their CS

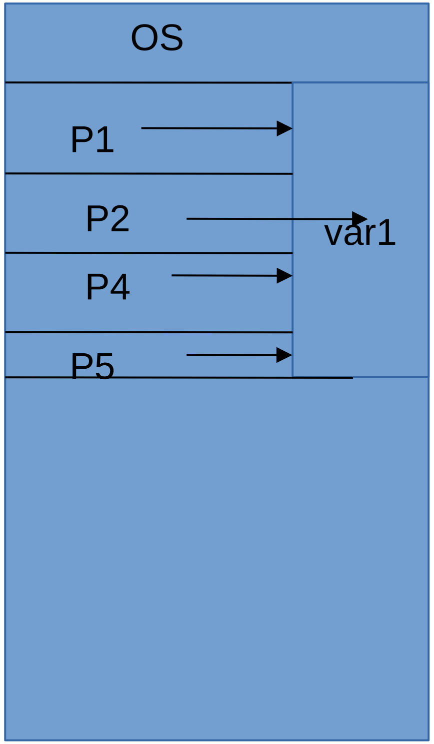
P1 , P2, P3 are trying to enter their CS, When P4 finish its CS

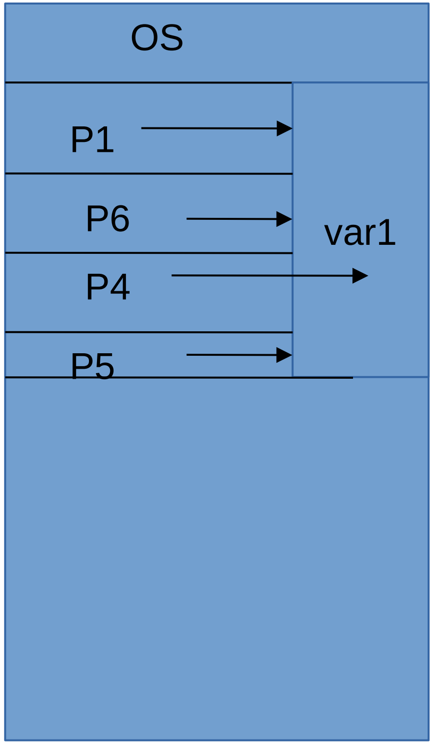
P4 will make the decision who will enter the CS next

Bounded Waiting

There must exist a bound on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS and before that request is granted.




OS
P1
P2
var1
P4
P5



P1 is always waiting for CS …..it should not be like this way....there should be a time limit

Entry section

Critical section

Exit section

Remainder section

Solution 1
var turn=0/1(turn is a atomic instruction)

P1	P2
Repeat	repeat
While turn!=0 do no_op	while turn!=1 do no_op
CS	CS
Turn=1	turn=0
Remainder section	remainder section

Disadvantage?

P1 p2

Cs Normal code CS

..

..

..

» ..

» CS

It requires strict alteration , E.g: if turn=0 and process 2 is ready to enter the CS, process 2 can not do so even though process 1 may be in its remainder section.

Solution 1
var turn=0/1(turn is a atomic instruction)

P1	P2
Repeat	repeat
While turn!=0 do no_op	while turn!=1 do no_op
CS	CS
Turn=1	turn=0
Remainder section	remainder section

Disadvantage?

P1 p2

Cs Normal code CS

..

..

..

» ..

» CS

It requires strict alteration , E.g: if turn=0 and process 2 is ready to enter the CS, process 2 can not do so even though process 1 may be in its remainder section.

Solution
replace turn with an array

Var array flag[0..1] of boolean

P0	P1
flag[0]=1	flag[1]=1
While flag[1]==1 do no_op	while flag[0]==1 do no_op
CS	CS
Flag[0]=0	flag[1]=0
Remainder section	remainder section

Flag[0]=1 means that process 0 is ready to enter the CS. But flag[0]=0 means that it is no longer needed to be in its CS

Disadvantage

p0 sets flag[0]=1

P1 sets flag[1]=1

P0 and p1 will be looping forever in their while statement.

Algo-3
By combining the key idea of algo1 and 2 we got a correct solution

P0	p1
Flag[0]=1	flag[1]=1
Turn=1	turn=0
While (flag[1]==1 and turn==1){	while(flag[0]==1 and turn==0){
Do no_op }	do no_op}
CS	CS
Flag[0]=0	flag[1]=0
Turn =1	turn=0

N processes

var

number:array[0...n-1] of integer

(a,b)<(c,d) if a<c or if a=c and b<d

max(a0,….an-1)is a number,k such that k>=ai for i=0...n-1