Process and Process Management

Prof. P.C.P. Bhatt

What is a Process?

- Recall from Module 1 that *a process is a program in execution.*
- A *process in execution needs resources* like processing resource, memory and IO resource.
- Imagine a program written in C – *my_prog.c.*
- After compilation we get an executable.
- If we now give a command like ./a.out it becomes a *process*.

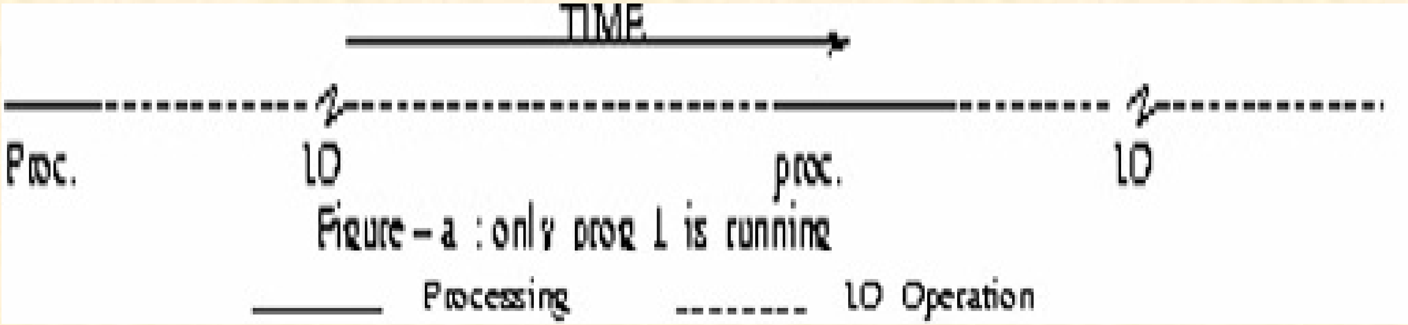
What is a Process - 2

- A computer can have *several process* running or active *at any given time.*
- In case of *multiple users* on a system, all users *share a common processing resource*.

Multi-programming and Time Sharing

- Let us consider a system with only one processor and one user running one program: *prog_1.*
- *IO* and *processing* will happen *alternately*.
- *When IO* is required, say keyboard input, the *processor idles.* This is because we are nearly a *million times* slower than the processor !!!

One Program - One User - Uni-processor Operation



*Note: Most of the time the processor is idling !!

Processor Utilization - 1

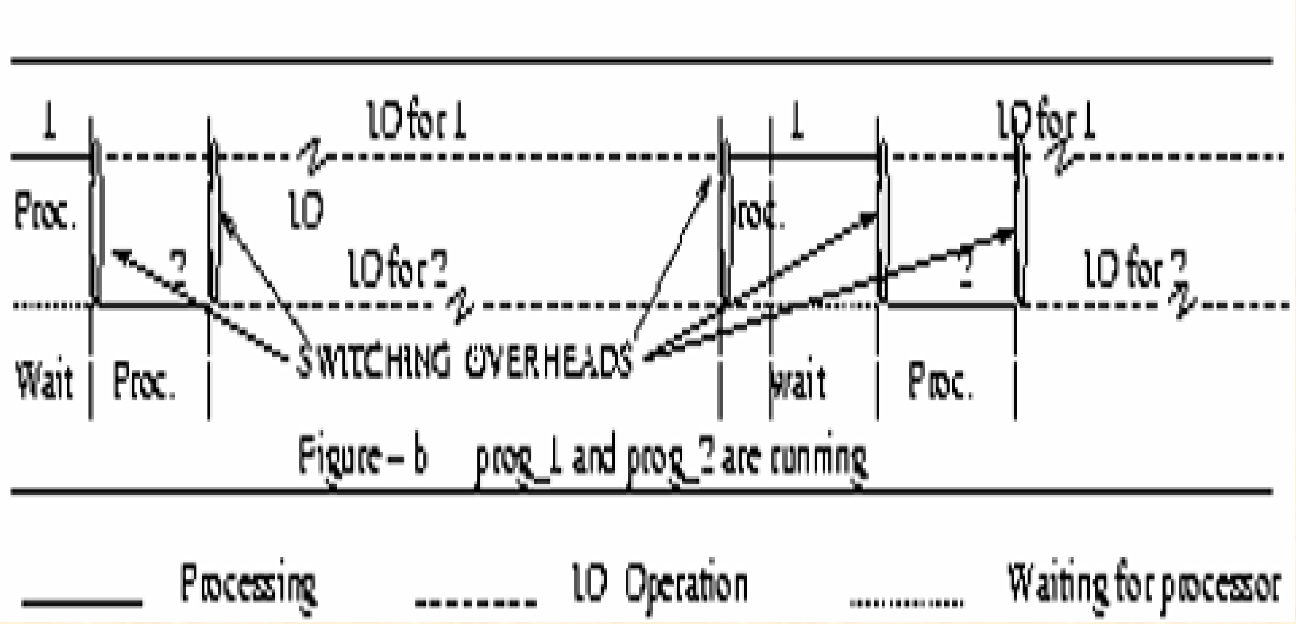
Now *think* about the processor utilization.

- What *percentage* of *time* are we *engaging* the *processor?*
- Recall that *Von Neumann* computing requires a *program to reside in main memory* to run.
- Clearly, having *just one program* would result in *gross under utilization of the processor.*
- To *enhance utilization*, we should try to have *more than one* ready-to-run program resident *in main-memory.*

Processor Utilization - 2

- A *processor* is the central element in a computer's operation.
- A *computer's throughput* depends upon the *extent of utilization of* its *processor.*
- *Previous figure* shows *processor idling* for very long periods of time when only one program is executed.
- Now let us consider two ready-to-run memory resident programs and their execution sequence.

Multi-programming Support in OS - 1



Consider two programs : *prog_1* and *prog_2* resident in main memory.

Multi-programming Support in OS - 2

In the figure,

- When *prog_1* is *not engaging* the *processor may be utilized* to run *another ready-to-run program.*
- Clearly, these *two programs* can be *processed without* significantly *sacrificing* the *time required* to process either of them.

Multi-programming Support in OS - 3

- *Disadvantage* - overhead faced while switching the *context* of use of the processor.
- *Advantage* - computer *resource utilization* is improved.
- *Advantage – memory utilization* is improved with multiple processes residing in main memory.
- A system would give maximum *throughput* when *all its components are busy all the time.*

Response Time - 1

Consider the following scenario :

- The *number of ready-to-run programs* must be maximized to *maximize throughput* of processor.
- These *programs* could belong to *different users*.

Response Time - 2

- A system with its resources being used by multiple users is called *time sharing system.*
- For example, a system with multiple terminals. Also a *web server serving multiple clients.*

However, such a usage has *overheads.* Lets see some of the overheads.

Response Time - 3

- In case of a switch in the context of the use of the processor, we must know *where in the program sequence* the program was suspended.
- In addition, *intermediate results stored in registers* have to be *safely stored* in a location *before suspension*.

Response Time - 4

- When a large number of resident user programs compete for the processor resource, the *frequency of storage, reloads* and *wait* periods also increase.
- If overheads are high, users will have to wait longer for their programs to execute *=>Response Time* of the system becomes longer.

Response Time *is the time interval which spans the time from when the last character has been input to the time when the first character of the output appears.*

Response Time : Some Facts

- In a time sharing system, it is important to achieve an acceptable *response time.*
- In a plant with an *on-line system,* system devices are continuously monitored : to determine the *criticality of a plant condition.* Come to think of it even a library system is an on-line system.
- If an online system produces a response time within acceptable limits, we say it is a *real-time system*.

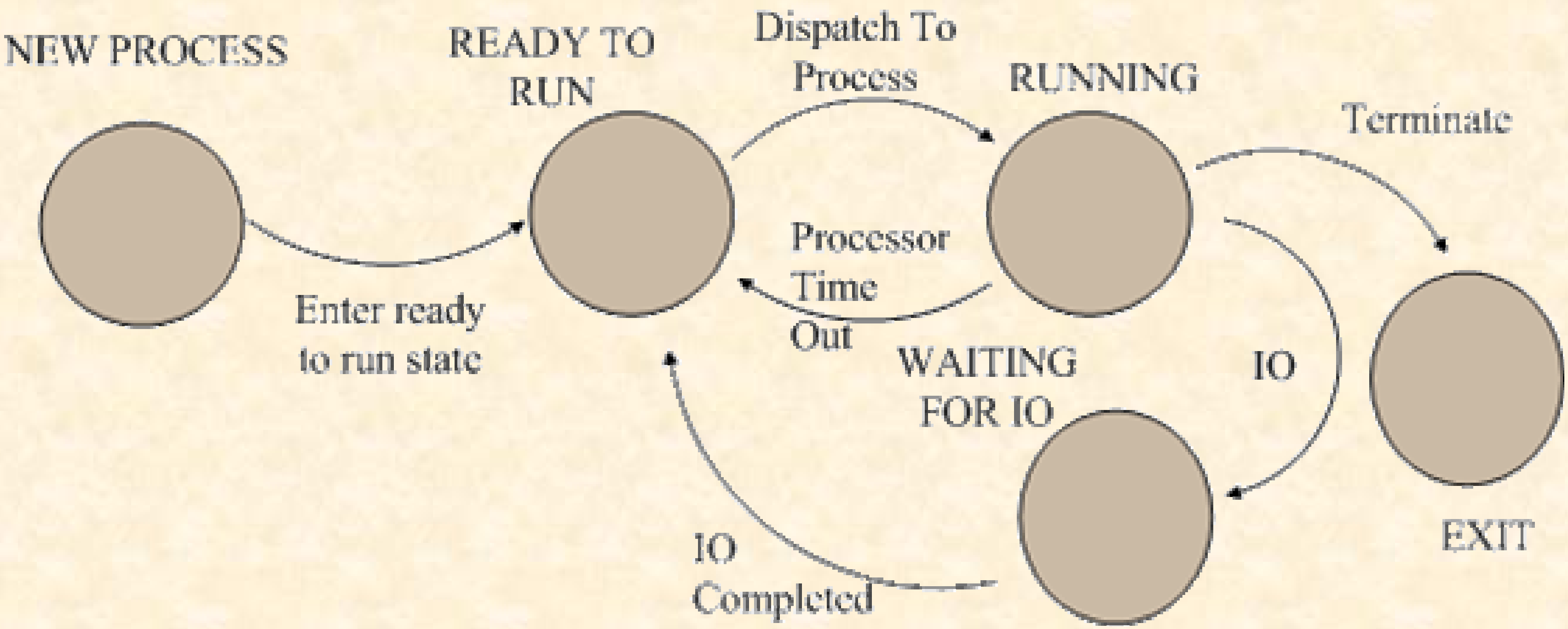
Process States

In all the previous examples, we said

- A process is in "*RUN*" state if is engaging the processor,
- A process is in "*WAIT*" state if it is waiting for *IO* to be completed
- In our *simplistic model* we may think of *5 states:*

  - ✓ *New-process*
  - ✓ *Ready-to-run*
  - ✓ *Running*
  - ✓ *Waiting-for-IO and*
  - ✓ *Exit*

Modeling Process States



Process states : Management issues - 1

- When a process is created, OS assigns it an ID - *pid,* and creates a *data structure* to record its progress. The *state* of the process is now *ready-to-run.*
- OS has a *dispatcher* that selects a ready-to-run process and assigns to the processor.
- OS allocates a *time slot* to run this process.
- OS *monitors* the progress of every process during its life time.

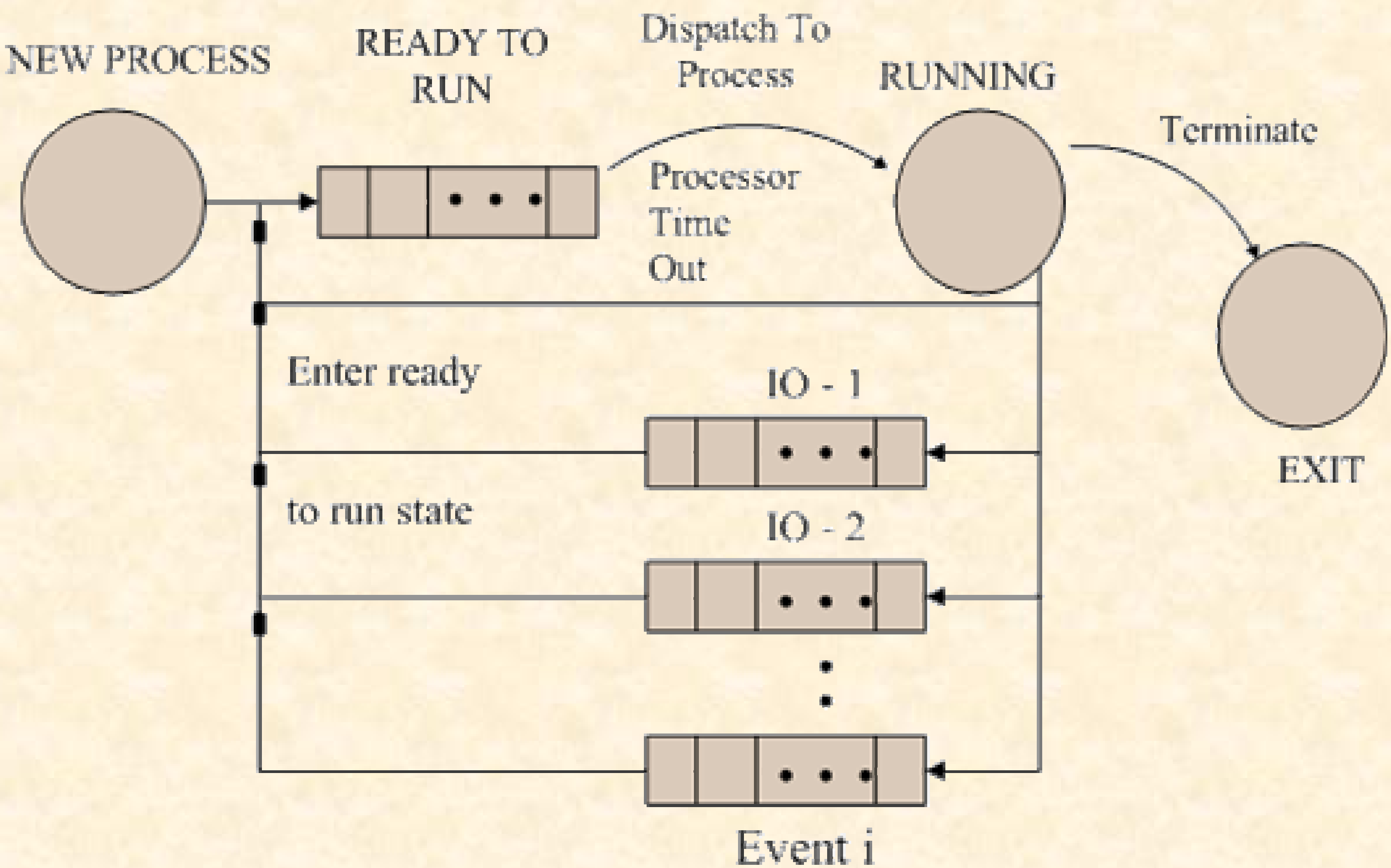
Process states : Management issues - 2

- A process may not progress till a certain event occurs - *synchronizing signal.*
- A process waiting for IO is said to be *blocked for IO.*
- OS *manages all these process migrations between process states.*

A Queuing Model

- Data structures are used for process management.
- OS maintains a *queue* for all *ready-to-run* processes.
- OS may have separate queue for each of the likely events (including completion of IO).

Queues Based Model - 1



Queues Based Model - 2

- This model helps in the study and analysis of chosen OS policies.
- As an example, Consider the *First Come First Served* policy for ready-to-run queue.
- To compare this policy with one that *prioritizes* processes we can study:
- The *average and maximum delays* experienced by the lowest priority process.

Queues Based Model - 3

- Comparison of the *response times and throughputs* in the two cases.
- Processor utilization *in the two cases and so on.* Such studies offer new insights - for instance, what level of *prioritization leads to starvation.*

Scheduling Considerations

- OS maintains process data in *various queues.*
- These queues advance based on *scheduling policies*
  - ✓ *First Come First Served.*
  - ✓ *Shortest Job First*
  - ✓ *Priority Based Scheduling*
  - ✓ *Batch Processing.*
- We also need to understand *Preemptive and Non-Preemptive* operations.
- Note that each policy *affects the performance of the overall system.*

Choosing a Scheduling Policy

- Scheduling policy depends on the *nature of operations*.
- An OS policy may be chosen to *suit situations with specific requirements.*
- Within a computer system, we need policies to *schedule access to processor, memory, disc, IO* and *shared resource (e.g. printers).*

Policy Selection - 1

- A *scheduling policy* is often determined by a *machine's configuration* and *its pattern of usage*.
- Scheduling is considered in the following *context*:
  - ✓ We have only *one processor* in they system
  - ✓ We have a *multi-programming* system – more than one ready-to-run program in memory.

Policy Selection - 2

- We shall study the *effect* on the following quality parameters:
  - ✓ Response time to users
  - ✓ Turn around time
  - ✓ Processor utilization
  - ✓ Throughput of the system
  - ✓ Fairness of allocation
  - ✓ Effect on other resources.
- We see that measures for response time and turn around are *user centered* parameters.

Policy Selection - 3

- Process utilization and throughput are *system centered* considerations.
- *Fairness of allocation* and *effect on other resources* affect both the *system and users.*
- An OS performance can be *tuned* by choosing an appropriate scheduling policy.

Comparison of Policies

- Let us consider 5 processes *P1* through *P5.*
- We shall make the following assumptions:
  - The jobs have to *run to completion.*
  - *No new jobs arrive* till these jobs are *processed.*
  - *Time required* for each job is known *appropriately.*
  - During the *run of jobs* there is *no suspension* for *IO operation.*

Comparison of Three Non-Preemptive Scheduling Policies

(A)

PROCESS NUMBER					
	P1	P2	P3	P4	P5
TIME	20	10	25	15	5

THE PROCESSES FOR PROCESSING

(B)

TIME TO RESPOND					
	P1	P2	P3	P4	P5
TIME	20	30	55	70	75

FCFS (QUEUE) ORDER : P1, P2, P3, P4. P5

AVERAGE TIME TO COMPLETE : 50

P1	P2	P3	P4	P5

GANTT CHART

(C)

TIME TO RESPOND					
	P3	P1	P2	P5	P4
TIME	25	45	55	60	75

PRIORITY QUEUE : P3, P1, P2, P5, P4

AVERAGE TIME TO COMPLETE : 52

P3	P1	P2	P5	P4

GANTT CHART

(D)

TIME TO RESPOND					
	P5	P2	P4	P1	P3
TIME	5	15	30	50	75

SHORTEST JOB FIRST : P1, P2, P3, P4. P5

AVERAGE TIME TO COMPLETE : 35

P5	P2	P4	P1	P3

GANTT CHART

Summary of Results

- Case B $\longrightarrow$ Average time to complete - 50
  (case of FCFS)

- Case C $\longrightarrow$ Average time to complete - 52
  (case of prioritized)

- Case D $\longrightarrow$ Average time to complete – 35
  (case of Shortest Job First)

Clearly it would seem that shortest job first is the best policy. Infact theoretically also this can be proved

Preemptive Policies - 1

PROCESS NUMBER	P1	P2	P3	P4	P5
TIME	30	10	25	15	5

(A)

THE PROCESSES FOR PROCESSING

TIME TO COMPLETE	P1	P2	P3	P4	P5
TIME	65	35	75	60	25

(B)

FCFS (QUEUE) ORDER : P1, P2, P3, P4. P5

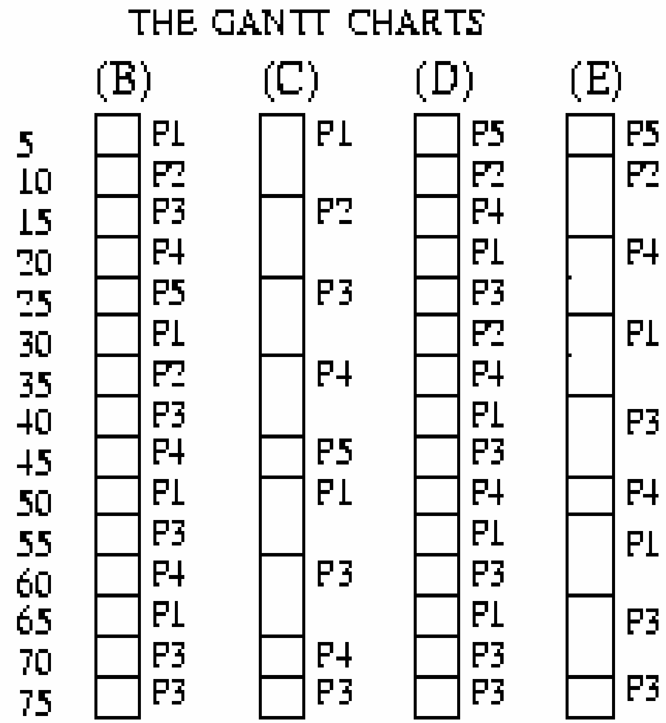
TIME : 5 UNITS; AVERAGE : 52, DIFF : 50

TIME TO COMPLETE	P1	P2	P3	P4	P5
TIME	55	20	75	70	45

(C)

FCFS (QUEUE) ORDER : P1, P2, P3, P4. P5

TIME : 10 UNITS; AVERAGE : 53; DIFF : 55



TIME TO COMPLETE	P1	P2	P3	P4	P5
TIME	65	30	75	50	5

(D)

TIME TO COMPLETE	P1	P2	P3	P4	P5
TIME	60	15	75	50	5

(E)

SHORTEST JOB FIRST ORDER : P5, P2, P4, P1, P3

TIME : 5 UNITS; AVERAGE : 45; DIFF : 70

TIME : 10 UNITS; AVERAGE : 41; DIFF : 70

Preemptive Policies - 2

In the figure, we *compare four cases*:

- *Round-robin* allocation with *time slice = 5 units* (CASE B)
- *Round-robin* allocation with *time slice = 10 units* (CASE C)
- *Shortest Job First* within the *Round-robin*; *time slice = 5 units* (CASE D)
- *Shortest Job First* within the *Round-robin; time slice = 10 units*. (CASE E)

Summary of Results

- Case B → Average time to complete - 52
- Case C → Average time to complete - 53
- Case D → Average time to complete - 45
- Case E → Average time to complete - 41

Clearly it would seem that shortest job first is the best policy. Infact theoretically also this can be proved

Yet Another Variation

- It was assumed before that all jobs were present initially.
- A more *realistic situation* is when *processes arrive at different times.*
- Each job is assumed to arrive with an estimate of time required to complete.
- Considering the *estimated remaining time, variation of SJF* is designed.

Shortest Remaining Time Schedule

THE PROCESSES FOR PROCESSING

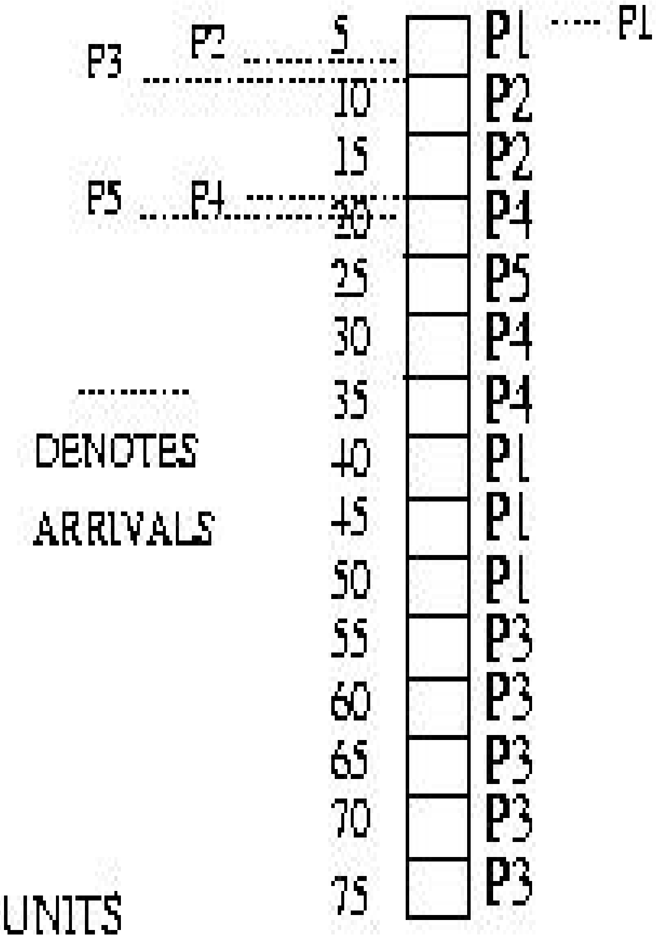
	PROCESS NUMBER				
	P1	P2	P3	P4	P5
TIME	20	10	25	15	5
ARRIVAL	0	3	5	15	17
SERVICE START	0	5	50	15	20
COMPLETED	50	15	75	35	25
DELAY	50	12	70	20	8

(A)

TIME SLICE : 5 UNITS;

AVERAGE TIME TO COMPLETE : 32 TIME UNITS

THE GANTT CHART



How to Estimate Completion Time? - 1

10	10	10	10	10	10	10

Scenario 1

1.7	2.9	1.9	3.7	2.5	1.8

Scenario 2

With the assumption that we are allocating 10 units of time for each burst, we notice: for the first scenario its inadequate where as for the second one it is too large.

How to Estimate Completion Time - 2

The following *strategies* can be observed:

- Allocate the *next larger time slice* to the *time actually used.*
- Allocate the *average over the last several time slice utilizations.* It gives all previous utilizations equal weightages to find the next time slice allocation.
- Use the entire history but *give lower weightages to the utilization in past* (*Exponential Averaging technique*).

Exponential Averaging Technique - 1

We denote our current, *n*th, CPU usage burst by t_n. Also, we denote the average of all past usage bursts up to now by τ_n . Using a weighting factor $0 \le \alpha \le n$ with t_n and $1-\alpha$ with τ_n, we estimate the next CPU usage burst. The predicted value of τ_{n+1} is computed as :

$$\tau_{n+1} = \alpha * t_n + (1-\alpha) * \tau_n$$

This formula is called an *exponential averaging formula.*

Exponential Averaging Technique - 2

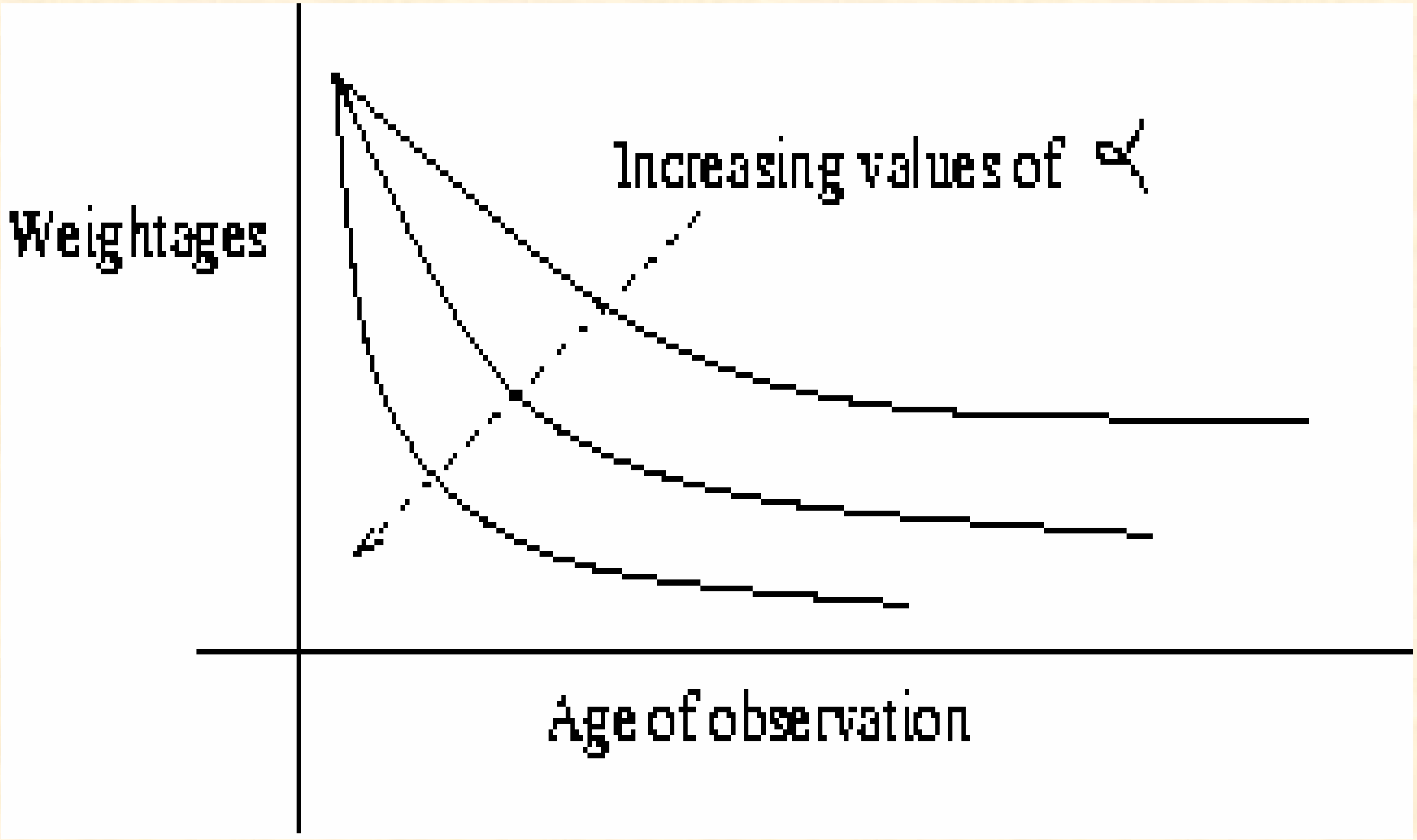
Let us briefly examine the role of α. If α is made 0 then we ignore the immediate past utilisation altogether. Obviously both would be undesirable choices. In choosing a value of α in the range of 0 to 1 we have an opportunity to weigh the immediate past usage, as well as, the previous history of a process with decreasing weightage. It is worth while to expand the formula further.

$$\tau_{n+1} = \alpha * t_n + (1-\alpha) * \tau_n = \alpha * t_n + \alpha * (1-\alpha) * t_{n-1} + (1-\alpha) * \tau_{n-1}$$

which on full expansion gives the following expression:

$$\tau_{n+1} = \alpha * t_n + \alpha * (1-\alpha) * t_{n-1} + \alpha * (1-\alpha)^2 * t_{n-2} + \alpha * (1-\alpha)^3 * t_{n-3} \ldots$$

Exponential Averaging Technique - 3



In figure above we see the effect of the choice of α has in determining the weightages for past utilisations.

Process Context Switching - 1

- OS maintains a lot of information about the *resources used by a running process*.
- The information stored *establishes the context for the process*. Usually the following is stored.
    - ✓ The *program computed.*
    - ✓ The *values in various registers*.
    - ✓ The *process states* etc..

Process Context Switching - 2

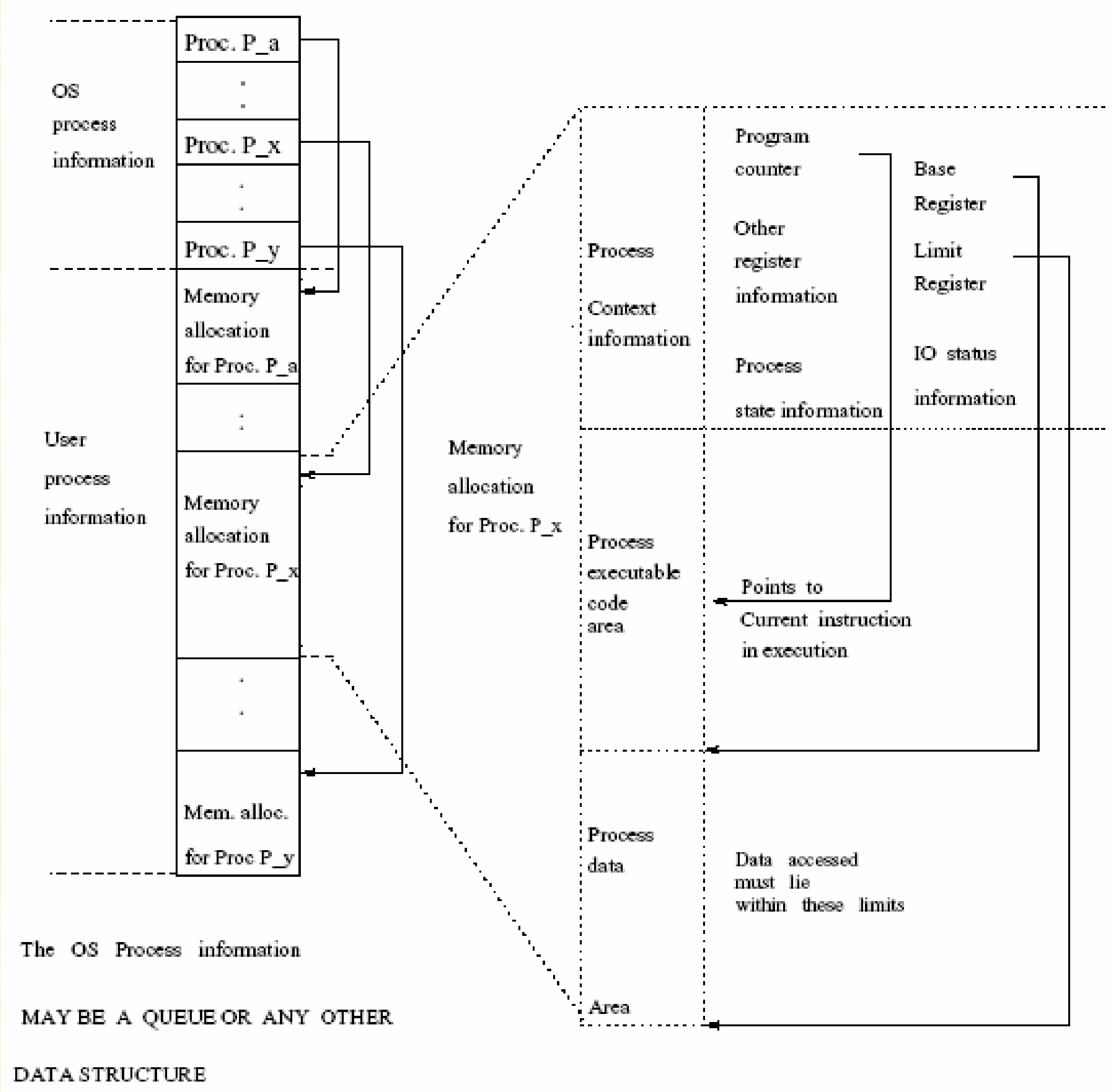
When a process is switched the context information needs to be changed as follows :

- *For the outgoing process*: Store the information of the current process in the some area of memory
- *For incoming process*: Copy the previously stored information from memory.

The information context that is switched is illustrated in the figure in the following slide.

Process Context Switching - 3

Execution Context of a process



Process Context Switching - 4

An OS maintains, and keeps updating, a lot of information about the resources in use for a running process. For instance, each process in execution uses the program counter, registers and other resources within the CPU. So, whenever a process is switched, the OS moves out, and brings in, considerable amount of context switching information as shown in the previous figure. We see that process P_x is currently executing (note that the program counter is pointing in executable code area of P_x).

Process Context Switching - 5

Let us now switch the context in favor of running process P_y.

The following must happen:

- All the current context information about process P_x must be updated in its own context area.
- All context information about process P_y must be downloaded in its own context area.
- The program counter should have an address value to an instruction of process P_y. and process P_y must be now marked as "running".

Process Context Switching - An Example

The process context area is also called *process control block.*

As an example when the process P x is switched the information stored is:

1. Program counter
2. Registers (like stack, index etc.) currently in use
3. Changed state (changed from Running to ready-to-run)
4. The base and limit register values
5. IO status (Files opened; IO blocked or completed etc.)
6. Accounting
7. Scheduling information
8. Any other relevant information.

When the process P_y is started its context must be loaded and then alone it can run.