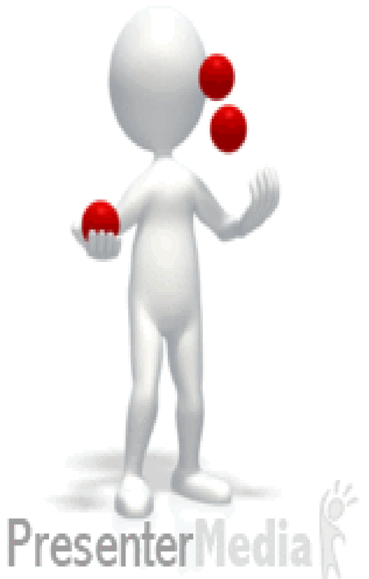
UNIT-2
PROCESS MANAGEMENT AND THREAD MANAGEMENT



Roadmap - Part-1

- Process
  - Process Control Block
  - Process State model
  - Process concepts
  - Process Control Structure

Requirements of an Operating System

- **Fundamental Task: Process Management**
  - The **process** retains the attributes of resource ownership,
  - The **thread** retains the attributes of multiple, concurrent execution streams running within a process.

- **The Operating System must**
  - Interleave the execution of multiple processes
  - Allocate resources to processes, and protect the resources of each process from other processes,
  - Enable processes to share and exchange information,
  - Enable synchronization among processes.

What is a "process"?

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions

- A **process** is comprised of:
  - Program code (possibly shared)
  - A set of data
- A number of attributes describing the state of the process

Process Control Block

- Contains the process elements
- Created and manage by the operating system
- Allows support for multiple processes
- Emphasise that the Process Control Block contains sufficient information so that it is possible to interrupt a running process and later resume execution as if the interruption had not occurred.
- (**Figure 3.1**) shows process control block that is created and managed by the OS.

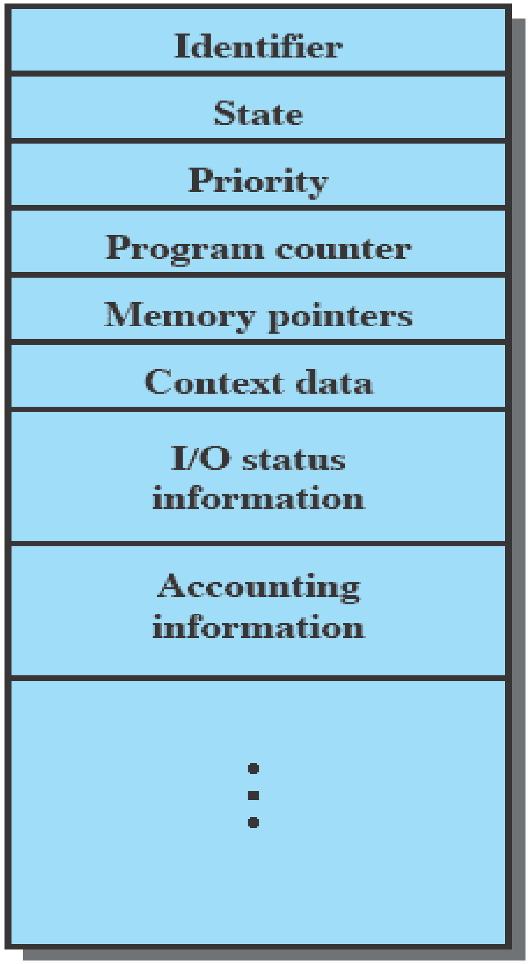


Figure 3.1 Simplified Process Control Block

Process Control Block

- While the program is executing, this process can be uniquely characterized by a number of elements, including the following:
- **Identifier:**
  - A unique identifier associated with this process, to distinguish it from all other processes.
- **State:**
  - If the process is currently executing, it is in the running state.
- **Priority:**
  - Priority level relative to other processes.
- **Program counter:**
  - The address of the next instruction in the program to be executed.
- **Memory pointers**:
  - Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:**
  - These are data that are present in registers in the processor while the process is executing.

•**I/O status information:**

—Includes outstanding I/O requests, I/O devices assigned to this process, a list of files in use by the process, and so on.

•**Accounting information:**

—May include the amount of processor time and clock time used, time limits, account numbers, and so on.

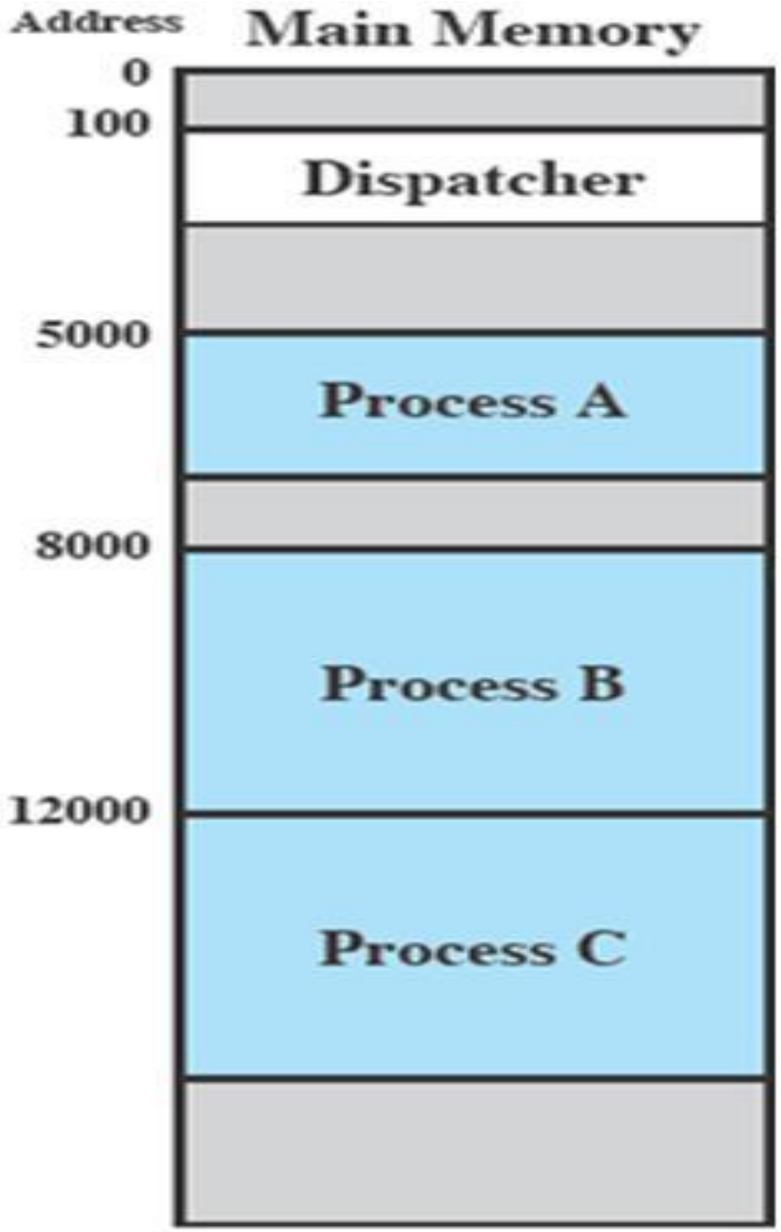
Trace of the Process

- **Trace:-**
  - The behavior of an individual process is shown by listing the sequence of instructions that are executed This list is called a **Trace.**
- **Dispatcher:-**
  - It is a small program which switches the processor from one process to another.

Process Execution Example

- Consider three processes being executed
- All are in memory (plus the dispatcher)
- Lets ignore virtual memory for this.

Process Execution Example

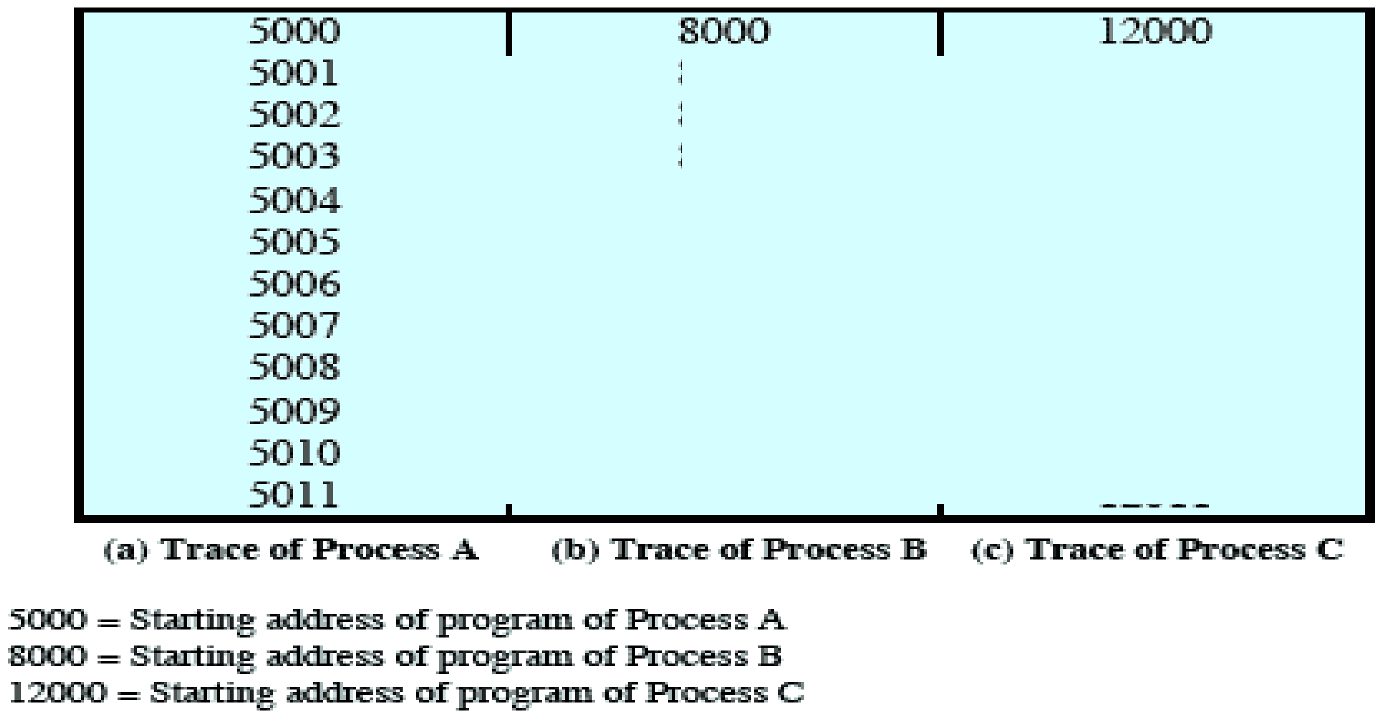


- We assume that the OS only allows a process to continue execution for a maximum of six instruction cycles,

- After which it is interrupted; this prevents any single process from monopolizing processor time

Trace from the processes point of view:

- Each process runs to completion

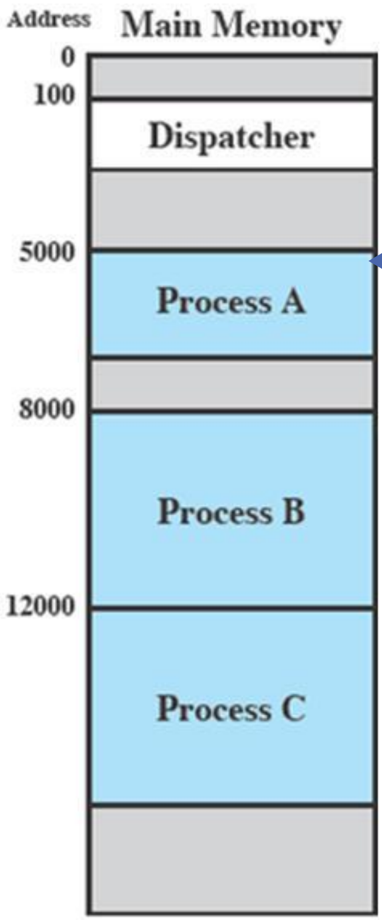


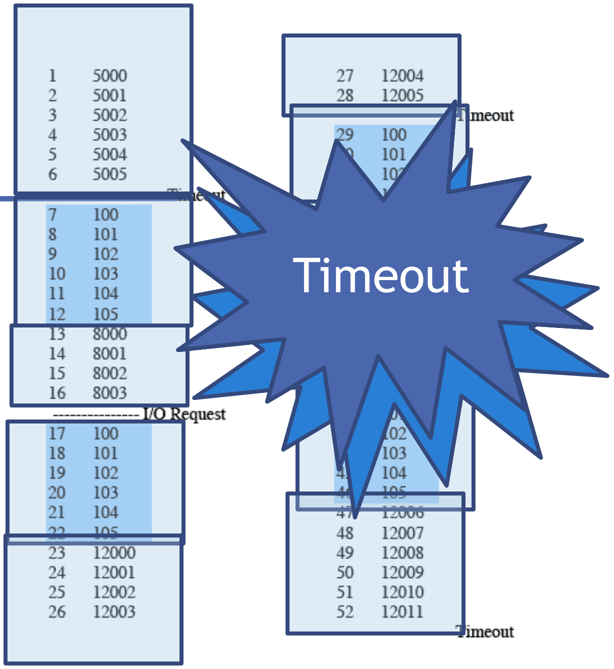
(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	:	
5002	:	
5003	:	
5004		
5005		
5006		
5007		
5008		
5009		
5010		
5011		

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

Trace from Processors point of view





```
1    5000               27   12004
2    5001               28   12005
3    5002               ------------- Timeout
4    5003               29   100
5    5004                    101
6    5005                    10
------------- Timeout        
7    100                     
8    101                     
9    102                     
10   103                     
11   104                     
12   105                     
13   8000                    
14   8001                    
15   8002                    
16   8003                    
------------- I/O Request    
17   100                     102
18   101                     103
19   102                4    104
20   103                     105
21   104                47   12006
22   105                48   12007
23   12000              49   12008
24   12001              50   12009
25   12002              51   12010
26   12003              52   12011
                        ------------- Timeout
```

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

Two-State Process Model

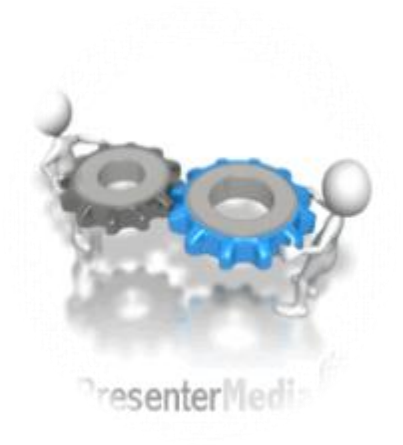
- The operating system's principal responsibility is controlling the execution of processes;
  - ✓This includes determining the interleaving pattern for execution and allocating
    resources to processes.
- The first step in designing an OS to control processes is to describe the behaviour that
  we would like the processes to exhibit.

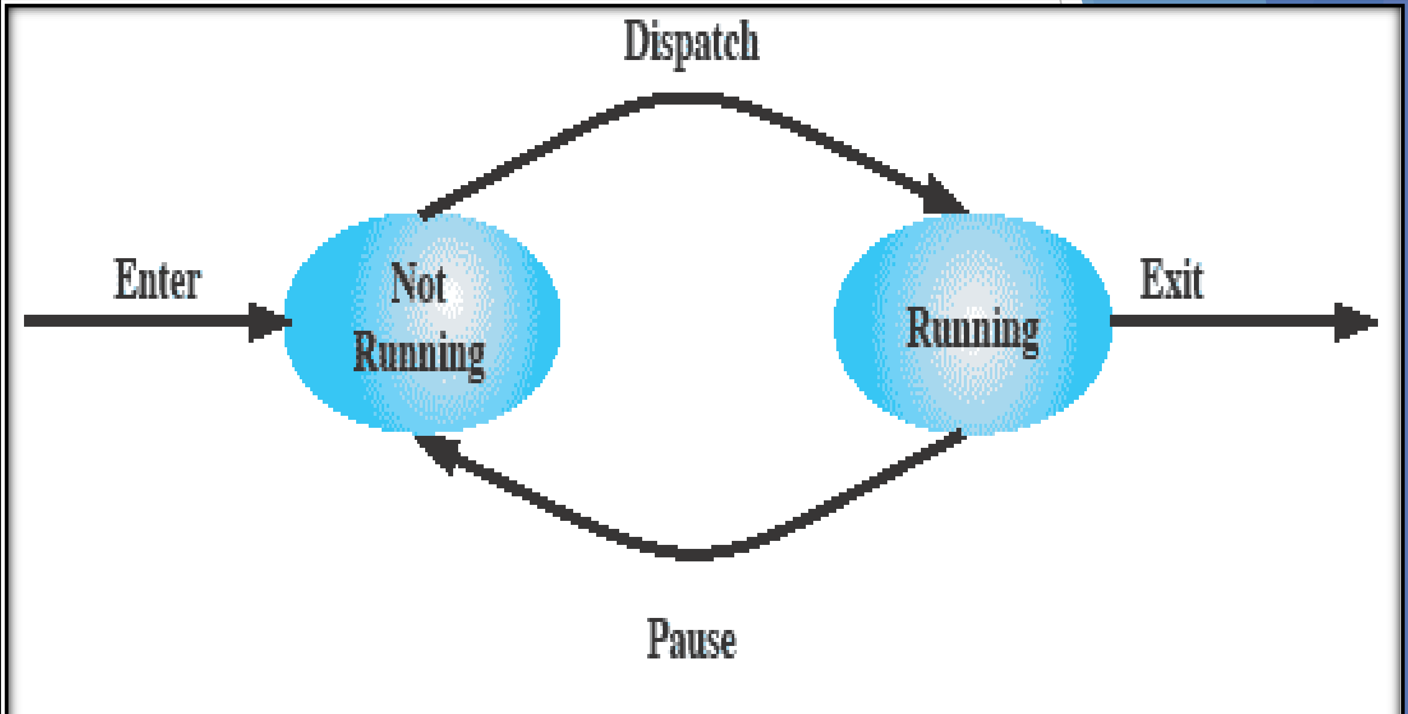
- When the OS creates a new process, it creates a process control block for the process
  and enters that process into the system in the Not Running state.

- The process exists, is known to the OS, and is waiting for an opportunity to execute.
- From time to time, the currently running process will be interrupted and the dispatcher portion of the OS will select some other process to run.
- The former process moves from the Running state to the Not Running state, and one
  of the other processes moves to the Running state.

Two-State Process Model

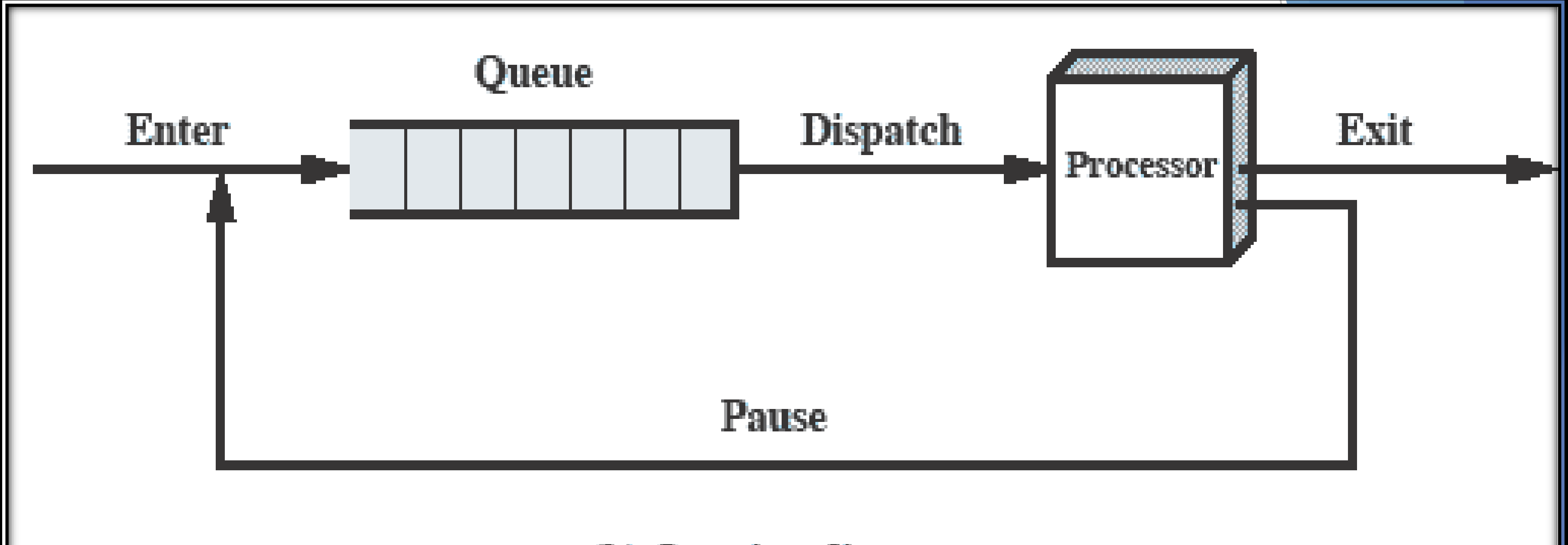
- In this model, a process may be in one of two states:
  - ✓Running or Not Running, as shown in Figure 3.5a.





(a) State transition diagram

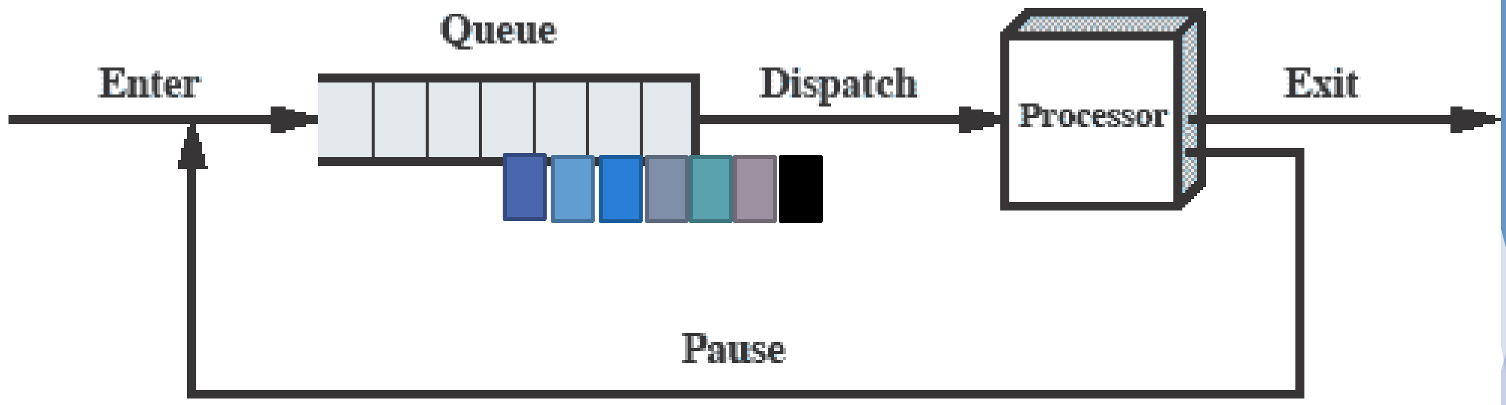
Queuing Diagram for Two state Process Model



(b) Queuing diagram

- Processes that are not running must be kept in some sort of queue, waiting their turn to execute.
- Figure 3.5b suggests a structure.
- There is a single queue in which each entry is a pointer to the process control block of a particular process.
- A process that is interrupted is transferred to the queue of waiting processes.
- If the process has completed or aborted, it is discarded
- In either case, the dispatcher takes another process from the queue to execute.

Queuing Diagram



(b) Queuing diagram

Etc … processes moved by the dispatcher of the OS to the CPU then back to the queue until the task is competed

Five-State Process Model

Problem With Two State Process Model:-

- If all processes were always ready to execute, then the queuing discipline suggested by Figure 3.5b would be effective.

- But, when some processes in the Not Running state
    - are ready to execute, and
    - While others are blocked, waiting for an I/O operation to complete.

- Thus, using a single queue, the dispatcher could not just select the process at the oldest end of the queue.

- Rather, the dispatcher would have to scan the list looking for the process that is not blocked and that has been in the queue the longest.

- The solution to this situation is to split the Not Running state into two states:
    - ✓Ready and Blocked.
    - ✓ As shown in Figure 3.6

Five-State Process Model

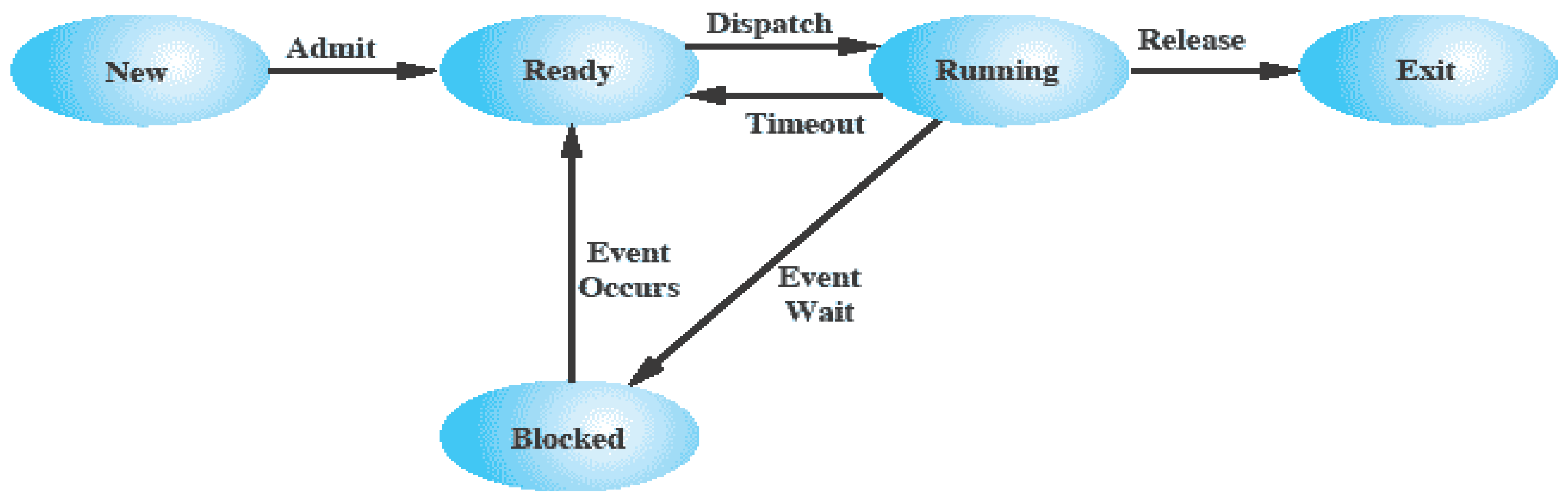


Figure 3.6 Five-State Process Model

Five-State Process Model

The five states in this new diagram are as follows:

• **Running:-**

The process that is currently being executed.

• **Ready:-**

A process that is prepared to execute when given the opportunity.

•**Blocked/Waiting:-**

A process that cannot execute until some event occurs, such as the completion of an I/O operation.

•**New:-**

A process that has just been created but has not yet been admitted to the pool of executable processes by the OS.

•**Exit:**

A process that has been released from the pool of executable processes by the OS.

Possible state transitions for this model are as follows:-

- **Null -> New:** A new process is created to execute a program.
- **New -> Ready:**
  - The OS will move a process from the New state to the Ready state when it is prepared to take on an additional process.
- **Ready -> Running:**
  - When it is time to select a process to run, the OS chooses one of the processes in the Ready state.
- **Running -> Exit:**
  - The currently running process is terminated by the OS if it is completed.
- **Running -> Ready:**
  - The most common reason for this transition is that the running process has timeout.
- **Running ->Blocked:**
  - A process is put in the Blocked state if it requests something for which it must wait.
- **Blocked ->Ready:**
  - A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs.
- **Ready -> Exit:**
  - For clarity, this transition is not shown on the state diagram. E.g If Parent terminates all its child processes.
- **Blocked -> Exit:**
  - The comments under the preceding item apply.

Process State Example :-

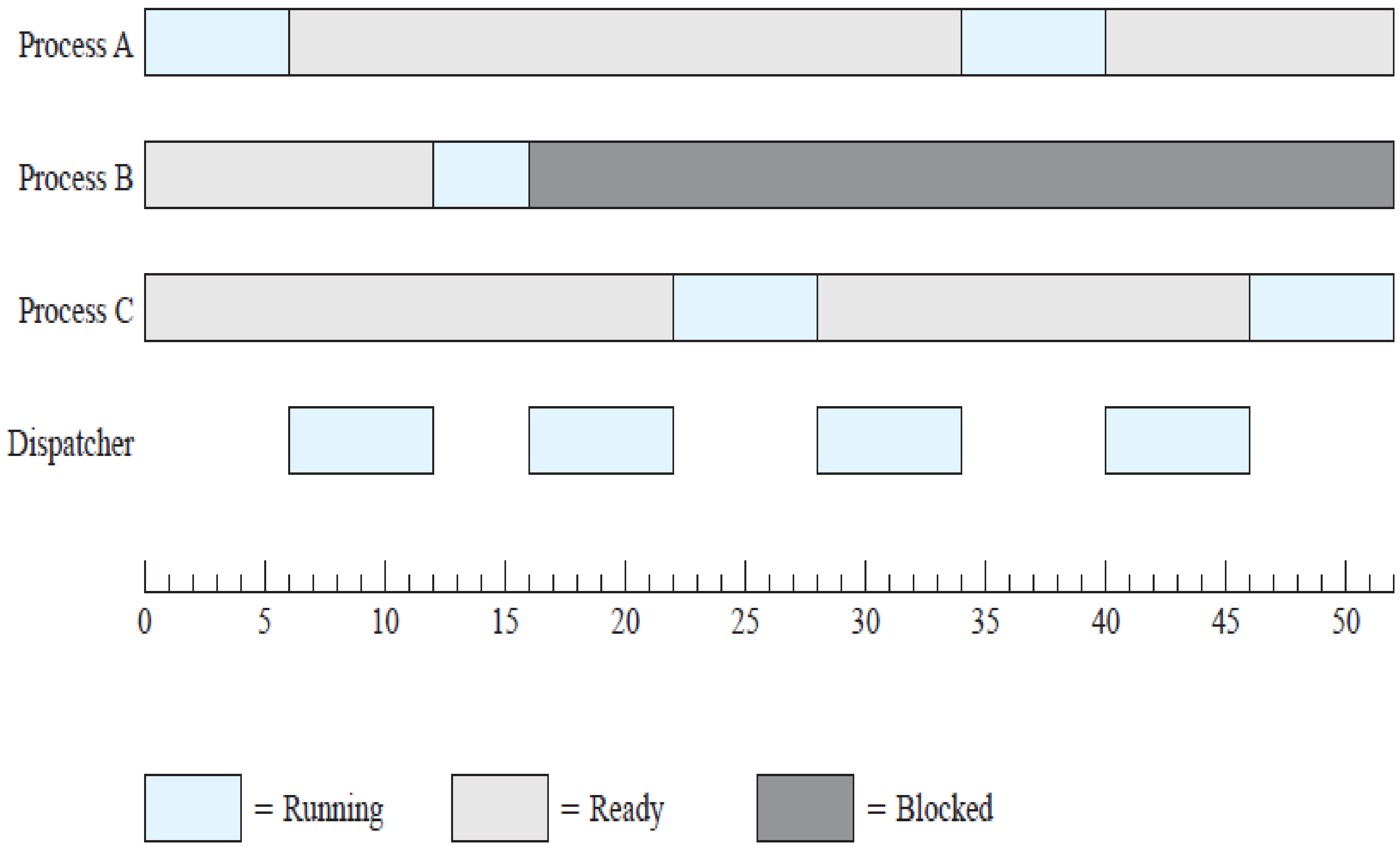
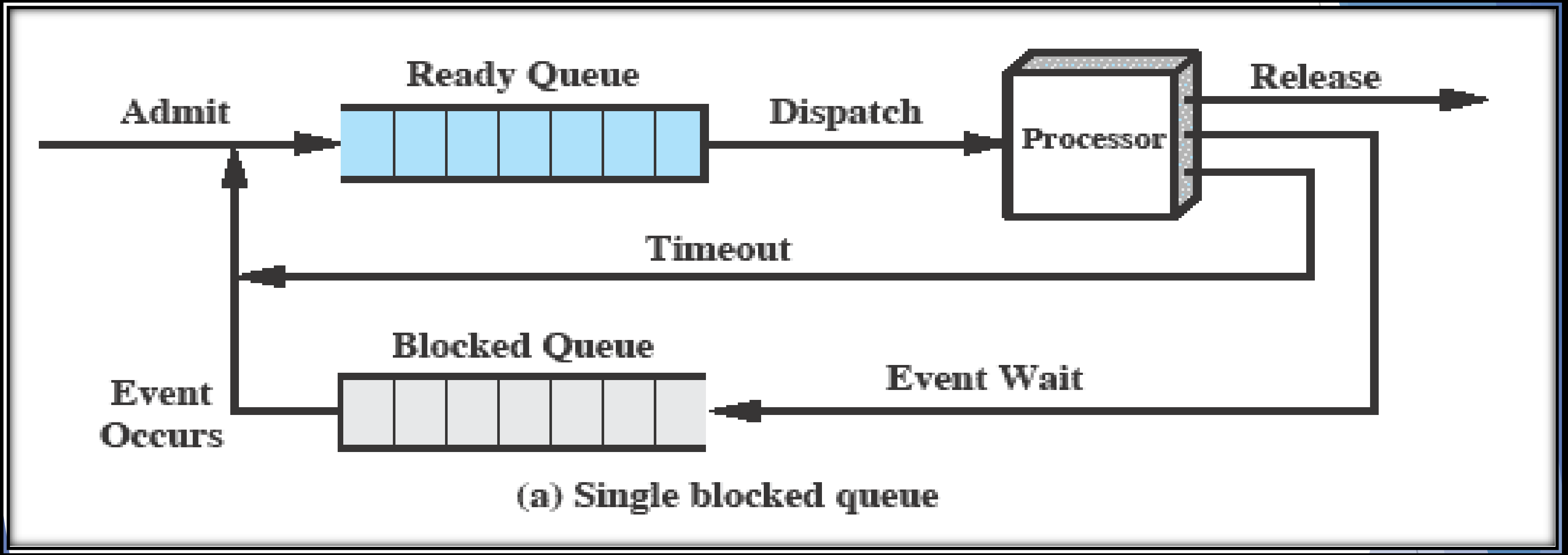


Figure 3.7 Process States for the Trace of Figure 3.4

Queuing Diagram for Two state Process Model

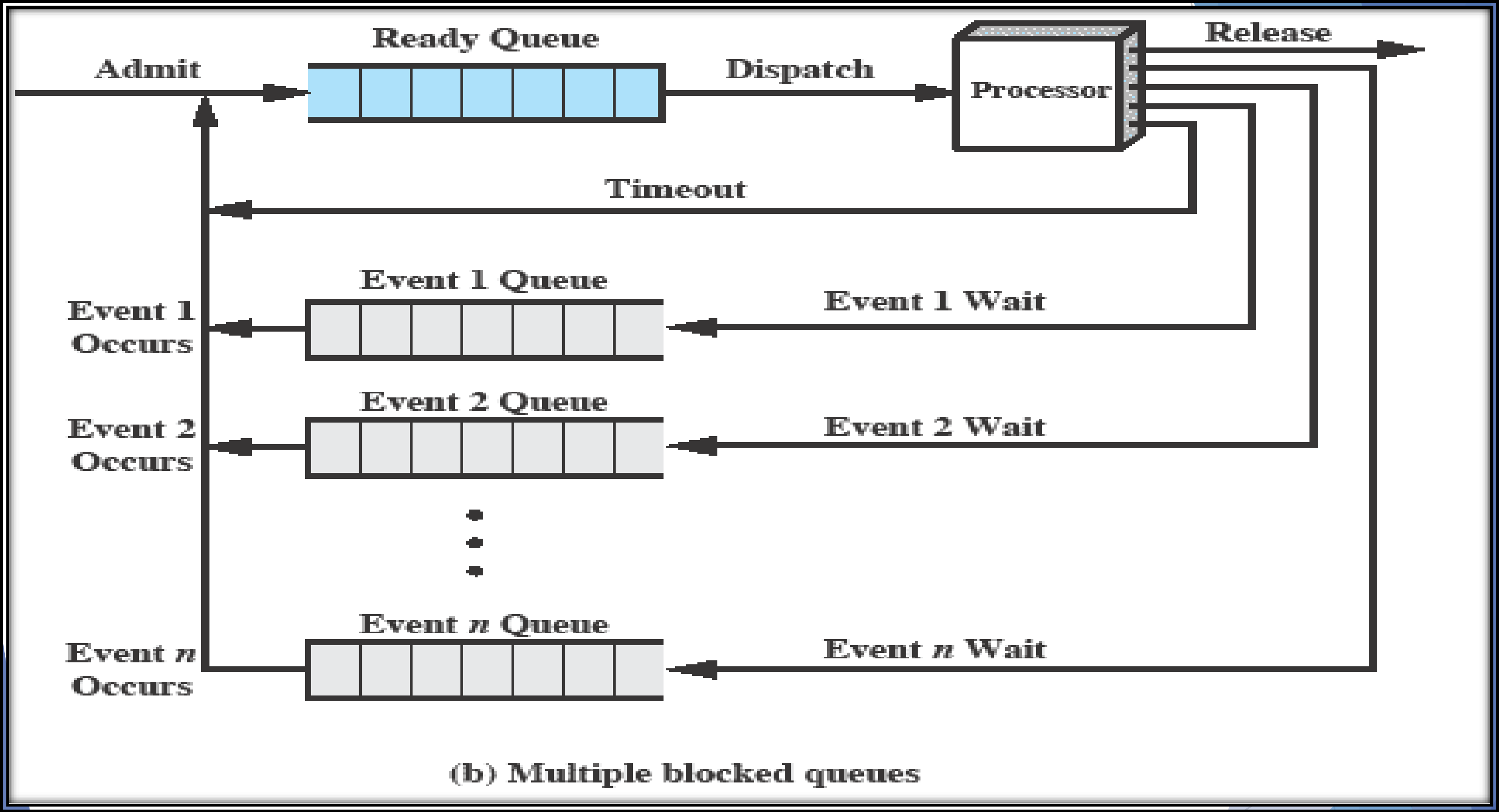
(1) Using Two Queues:-



(a) Single blocked queue

• In the simplest solution, this model would require an additional queue for the blocked
processes.
• **But** when an event occurs the dispatcher would have to cycle through the entire queue to see which process is waiting for the event.
•This can cause huge overhead when their may be 100's or 1000's of processes.
• Solution to above is that use multiple queues for blocked processes.

(2) Using Multiple Queues:-



(b) Multiple blocked queues

More efficient to have a separate ‘blocked’ queue for each type of event.

Suspended Processes

The Need For Swapping

- Each process to be executed must be loaded fully into main memory.
- Thus, in Figure 3.8b, all of the processes in all of the queues must be resident in main memory.
- Processor is faster than I/O so if all processes in memory are waiting for I/O then processor, in uniprogramming, is idle much of time.
  - ✓ One solution is Main memory could be expanded to accommodate more processes but that is costly.
  - ✓ Another solution is Swapping these processes to disk to free up more memory and use processor on more processes
- Blocked state becomes **suspend** state when swapped to disk

Suspended Processes

With one suspend state:-

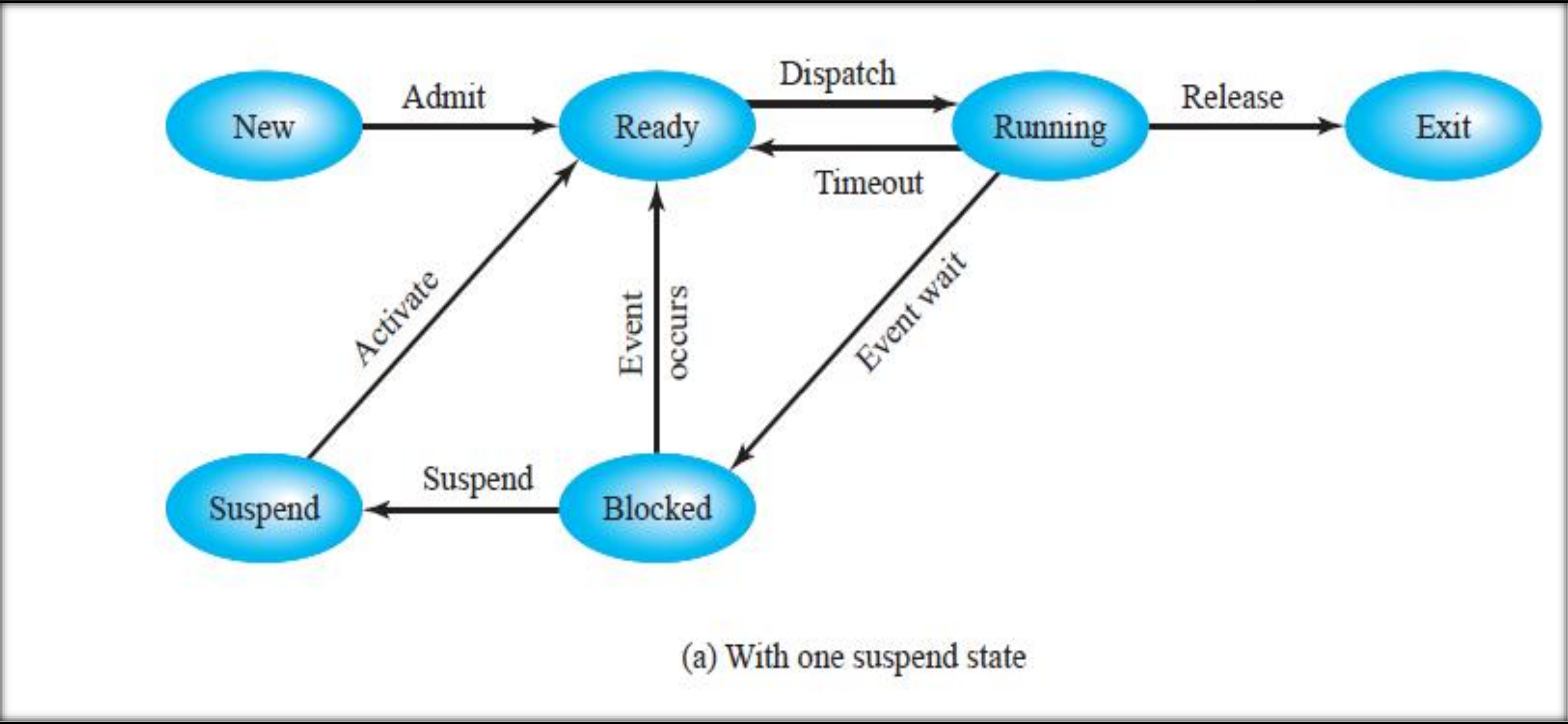


Figure 3.9 **Process State Transition Diagram with Suspend States**

Again, the simple solution is to add a single state – but this only allows processes which are blocked to be swapped out.

Problem With one suspend state:-

- When all of the processes in main memory are in the Blocked state, the OS can suspend one process by putting it in the Suspend state and transferring it to disk.

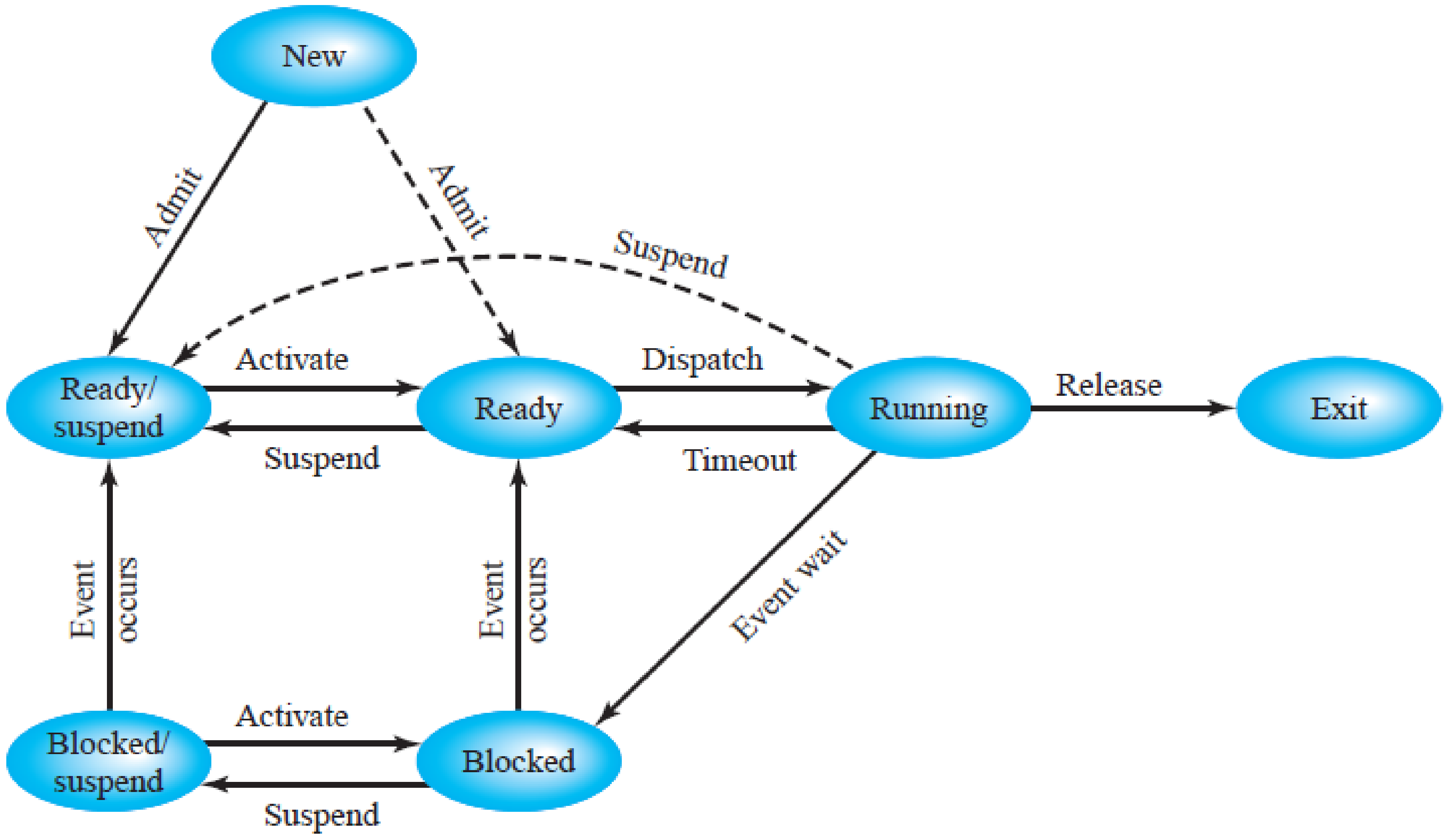
- When Os has to bring in another process, it is possible that the event for which process has blocked, occurred and that process is ready for execution.

- But all the processes which are blocked and which are available for execution are in the same state suspend state.

- So Os has to search entire queue to find the process which is ready and suspended.

- So We can add two suspend states:- **Ready / suspend** and **Blocked / suspend**

Seven State Process Model



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States

With two suspend states:-

Two suspend states allow all processes which are not actually running to be swapped.
Run through the four states:

- **Ready:**
  - The process is in main memory and available for execution.
- **Blocked:**
  - The process is in main memory and awaiting an event.
- **Blocked/Suspend:**
  - The process is in secondary memory and awaiting an event.
- **Ready/Suspend:**
  - The process is in secondary memory but is available for execution as soon as it is loaded into main memory

Important new transitions are the following:

- **Blocked -> Blocked/Suspend:**
  - If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked.

- **Blocked/Suspend -> Ready/Suspend:**
  - A process in the Blocked/Suspend state is moved to the Ready/Suspend state when the event for which it has been waiting occurs.

- **Ready/Suspend -> Ready:**
  When there are no ready processes in main memory, the OS will need to bring one in to continue execution.
- **Ready -> Ready/Suspend:**
  Normally, it is not possible , it may be necessary to suspend a ready process if that is the only way to free up a sufficiently large block of main memory.
- **New → Ready/Suspend and New →Ready:**
  When a new process is created, it can either be added to the Ready queue or the Ready/Suspend queue.
- **Blocked/Suspend -> Blocked:**
  - There is a process in the (Blocked/Suspend) queue with a higher priority than any of the processes in the (Ready/Suspend) queue and
  - The OS has reason to believe that the blocking event for that process will occur soon.
- **Running -> Ready/Suspend:**
  The OS could move the running process directly to the (Ready/Suspend) queue to free some main memory.
- **Any State -> Exit:**
  - Typically, a process terminates while it is running, either because it has completed or because of some fatal fault condition.
  - A process can be terminated by the parent process or if the parent has been terminated then it is moved to the exit state.

ROADMAP – PART-2 THREADS

- Multithreading
- Benefits of threads and uses of threads
- Process Vs Thread
- Thread Categories

THREADS

- A thread is a single sequence stream within a process.
- Because threads have some of the properties of processes, They are called lightweight processes.
- A thread (or lightweight process) consists of:
  - program counter, register set and stack space
  - A thread shares the following with peer threads:
  - code section, data section and OS resources (open files, signals)

- Threads share code section, data section etc with other threads so the threads are not independent of one another like processes.

Multithreading

The ability of an OS to support multiple, concurrent paths of execution within a single process.

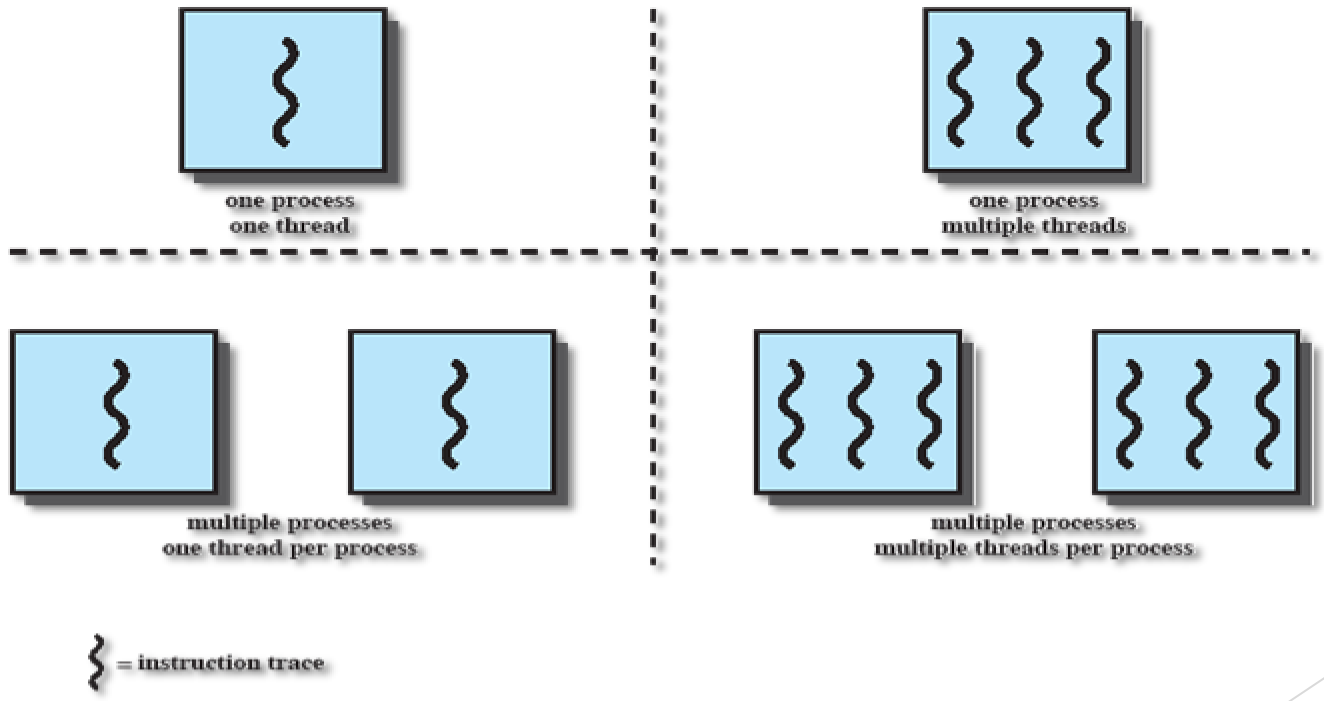


Figure 4.1 Threads and Processes [ANDE97]

Single Thread Approaches

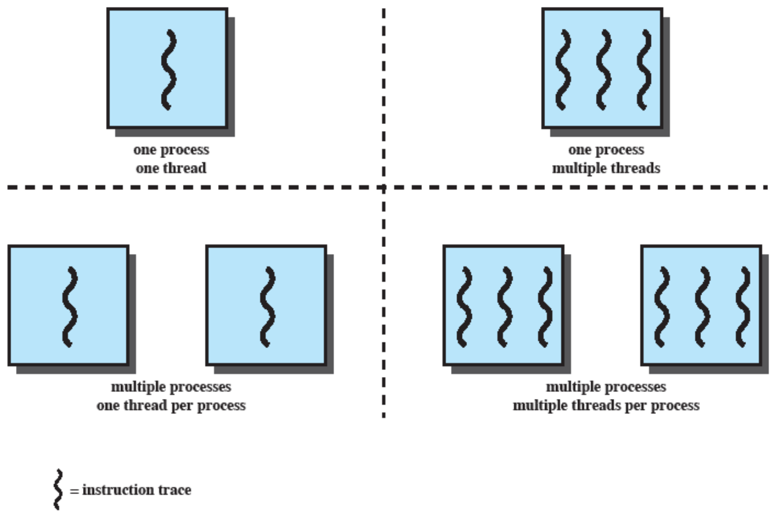


Figure 4.1 Threads and Processes [ANDE97]

- MS-DOS supports a single user process and a single thread.
- Some UNIX, support multiple user processes but only support one thread per process

Multithreading

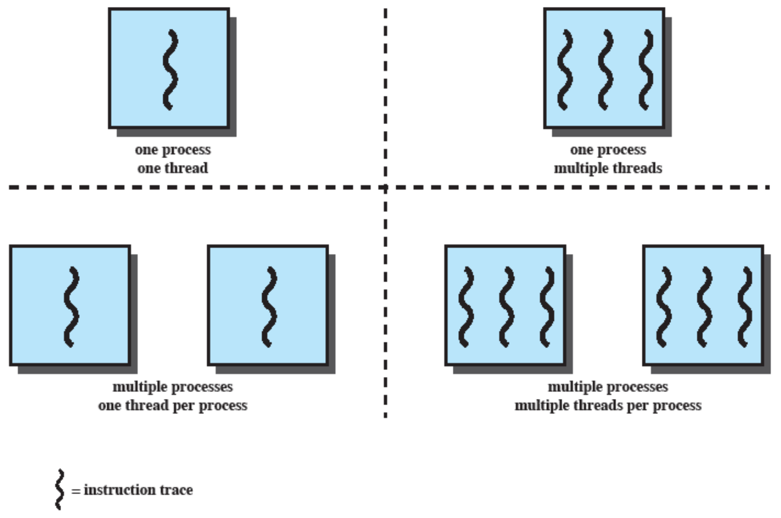


Figure 4.1 Threads and Processes [ANDE97]

- Java run-time environment is a single process with multiple threads
- Multiple processes ***and*** threads are found in Windows, Solaris, and many modern versions of UNIX

Benefits Of Threads

The **key benefits** of threads derive from the performance implications:

(1) It takes far less time to create a new thread in an existing process than to create a brand-new process.

(2) It takes less time to terminate a thread than a process.

(3) It takes less time to switch between two threads within the same process than to switch between processes.

(4) Threads enhance efficiency in communication between different executing programs.

Thread uses in a Single-User System

- Uses of threads in a single-user multiprocessing system:
- Foreground and background work :
  - e.g. Spreadsheet - one thread looking after display and Another thread updating results of formula.
- Asynchronous processing:
  - e.g. Protection against power failure within a word processor-
    -A thread writes random access memory (RAM) buffer to disk once every minute.
    - this thread schedules itself directly with the OS;
- Speed of execution:
  - A multithreaded process can compute one batch of data while reading the next batch from a device.
  - On a multiprocessor system, multiple threads from the same process may be able to execute simultaneously.

Differences:-

SR. NO:	PROCESSES	THREADS
1	Process is unit of allocation i.e. Resources, privileges, etc.	Threads are unit of execution which includes pc,sp and register
2	Each process has one or more threads	Each thread belongs to one process.
3	Inter process communication is expensive i.e Context switching required	Inter thread communication is cheap. Don't required process switching.
4	These are secure because one process can not corrupt another process	These are not secure because a thread can write memory used by another thread.
5	Take more time to create a process.	Takes less time to create a thread.
6	Take more time to terminate a process	Takes less time to terminate a thread
7	Take more time to switch between processes	Take less time to switch between threads
8	It has more communication overheads	It has less communication overheads
9	Processes are independent of one another.	Threads are not independent of one another.

Types of Processor Scheduling

In computing, scheduling is the method by which work is assigned to resources that complete the work. The work may be virtual computation elements such as threads, processes or data flows, which are in turn scheduled onto hardware resources such as processors, network links or expansion cards.

A scheduler is what carries out the scheduling activity. Schedulers are often implemented so they keep all computer resources busy (as in load balancing), allow multiple users to share system resources effectively, or to achieve a target quality of service. Scheduling is fundamental to computation itself, and an intrinsic part of the execution model of a computer system; the concept of scheduling makes it possible to have computer multitasking with a single central processing unit (CPU).

Goals of a Scheduler

A scheduler may aim at one or more goals, for example: maximizing throughput (the total amount of work completed per time unit); minimizing wait time (time from work becoming ready until the first point it begins execution); minimizing latency or response time (time from work becoming ready until it is finished in case of batch activity, or until the system responds and hands the first output to the user in case of interactive activity); or maximizing fairness (equal CPU time to each process, or more generally appropriate times according to the priority and workload of each process).

Types of operating system schedulers

The scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run. Operating systems may feature up to three distinct scheduler types: a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler, and a short-term scheduler. The names suggest the relative frequency with which their functions are performed.

Process scheduler

The process scheduler is a part of the operating system that decides which process runs at a certain point in time. It usually has the ability to pause a running process, move it to the back of the running queue and start a new process; such a scheduler is known as a preemptive scheduler, otherwise it is a cooperative scheduler.

Long-Term Scheduling

- The long-term scheduler determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming.

- Once admitted, a **job** or user program becomes a process and is added to the queue for the short-term scheduler. **OR**

- A newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler.

- The criteria used may include
  - priority, expected execution time, and I/O requirements.

- The decision as to when to create a new process is generally driven by the desired degree of multiprogramming.

Medium-Term Scheduling

- The medium-term scheduler is executed somewhat more frequently.
- Medium-term scheduling is part of the swapping function.
- Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming.
- On a system that does not use virtual memory, memory management is also an issue.
  - Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes

Short-Term Scheduling

- Also known as the dispatcher, executes most frequently and makes the fine-grained decision of which process to execute next.

- The short-term scheduler is invoked whenever an event occurs that may lead to the blocking of the current process or that may provide an opportunity to preempt a currently running process in favour of another.

- Examples of such events include
  - Clock interrupts
  - I/O interrupts
  - Operating system calls
  - Signals (e.g., semaphores)

Long-Term	Short-Term	Medium-Term
Long term is also known as a job scheduler	Short term is also known as CPU scheduler	Medium-term is also called swapping scheduler.
It is either absent or minimal in a time-sharing system.	It is insignificant in the time-sharing order.	This scheduler is an element of Time-sharing systems.
Speed is less compared to the short term scheduler.	Speed is the fastest compared to the short-term and medium-term scheduler.	It offers medium speed.
Allow you to select processes from the loads and pool back into the memory	It only selects processes that is in a ready state of the execution.	It helps you to send process back to memory.
Offers full control	Offers less control	Reduce the level of multiprogramming.

Process Scheduling Example

- Example set of processes, consider each a batch job

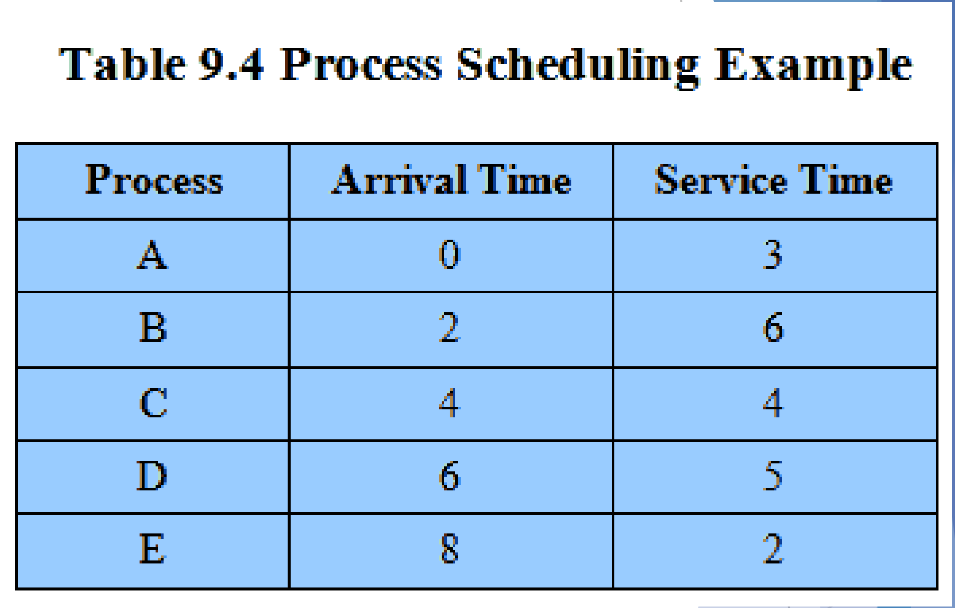


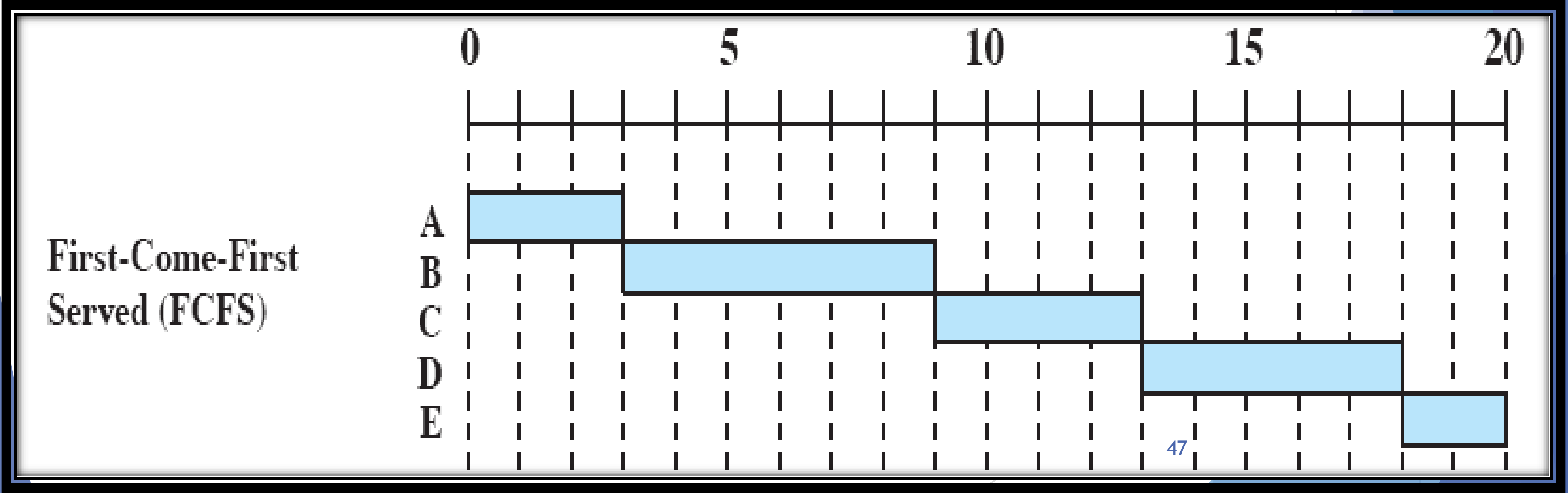
Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

– Service time represents total execution time

FIRST-COME-FIRST-SERVED

- The simplest scheduling policy is first-come-first-served (FCFS),
  - first-in-first-out (FIFO) or a strict queuing scheme.
- As each process becomes ready, it joins the ready queue.
- When the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running.



First-Come- First-Served

Process	A	B	C	D	E	
Arrival TIme	0	2	4	6	8	
Service Time (Ts)	3	6	4	5	2	

FCFS						MEAN
FinishTIme	3	9	13	18	20	
Turnaround Time (Tr) (FT - AT)	3	7	9	12	12	8.60
Tr / Ts	1	1.17	2.25	2.40	6.00	2.56

First-Come- First-Served

Process	Arrival Time	Service Time (Ts)	Start Time	Finish Time	Turnaround Time(Tr) (FT - AT)	Tr / Ts
w	0	1	0	1	1	1
x	1	100	1	101	100	1
y	2	1	101	102	100	100
z	3	100	102	202	199	1.99
MEAN					(400 / 4) 100	(103 .99 /4) 26

FIRST-COME-FIRST-SERVED

- FCFS performs much better for long processes than short ones.

- Another difficulty with FCFS is that it tends to favor processor-bound processes over I/O-bound processes.

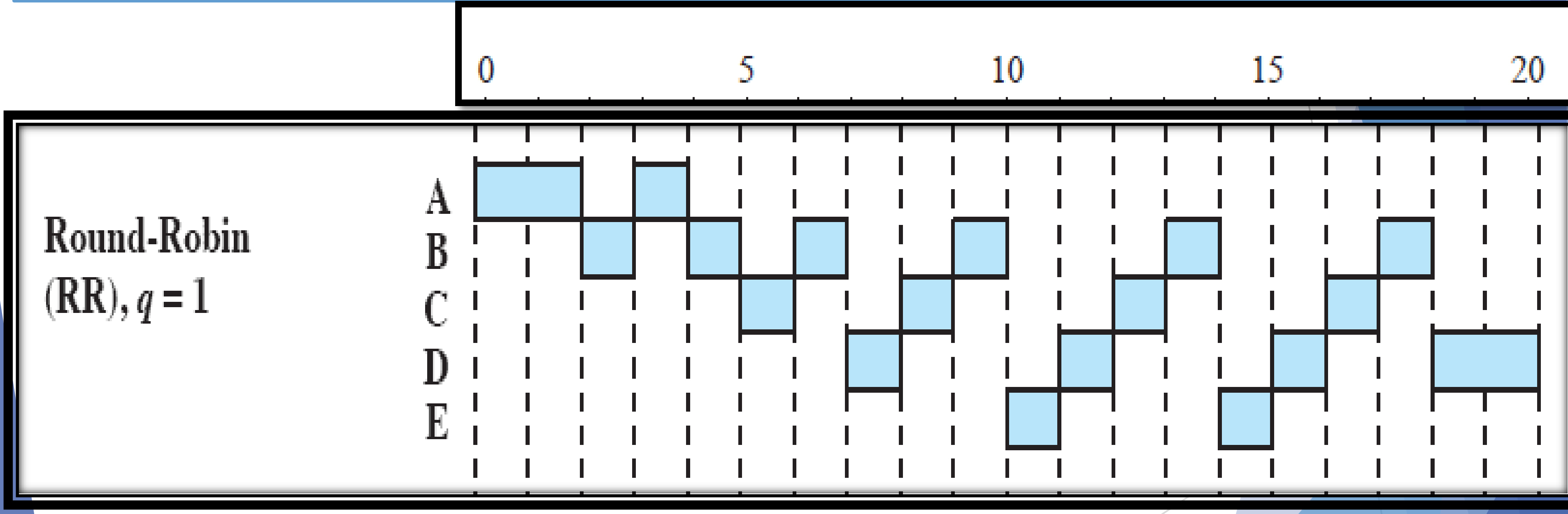
- FCFS is not an attractive alternative on its own for a uniprocessor system.

- However, it is often combined with a priority scheme to provide an effective scheduler.
    - Thus, the scheduler may maintain a number of queues, one for each priority level, and dispatch within each queue on a first-come-first-served basis.

Round Robin

- A straightforward way to reduce the penalty that short jobs suffer with FCFS is to use preemption based on a clock.
    - The simplest such policy is round robin.

- Also known as time slicing, because each process is given a slice of time before being preempted.



Round Robin

- Clock interrupt is generated at periodic intervals
- When an interrupt occurs, the currently running process is placed in the ready queue
  - Next ready job is selected

Round Robin

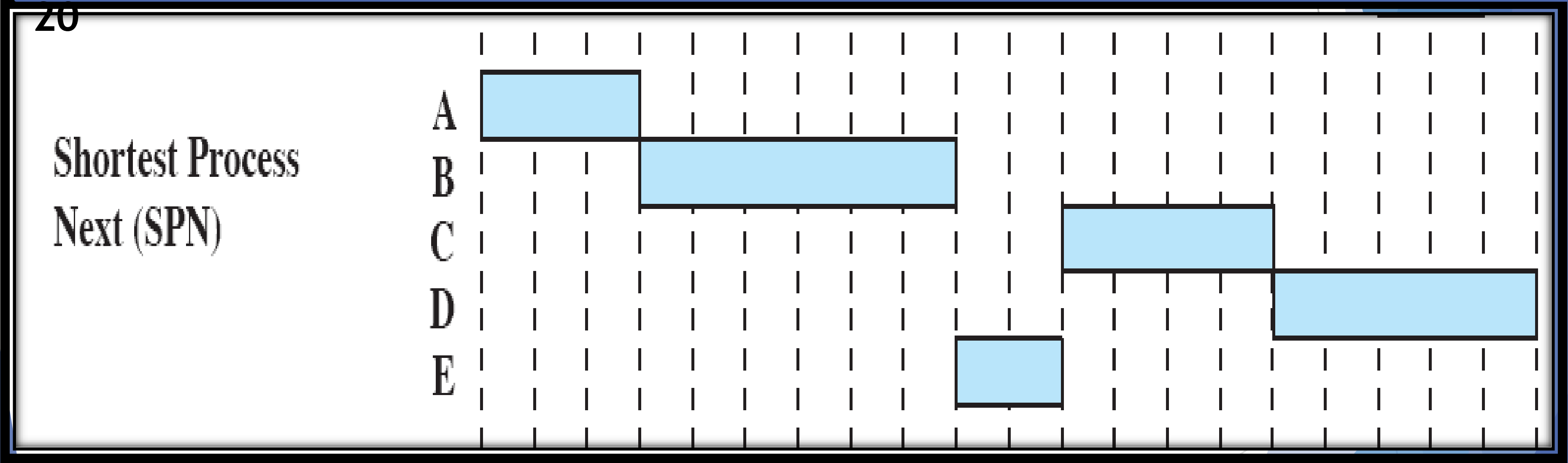
Process	A	B	C	D	E	
Arrival TIme	0	2	4	6	8	
Service Time (Ts)	3	6	4	5	2	

RR (q=1)						MEAN
FinishTIme	4	18	17	20	15	
Turnaround Time (Tr)	4	16	13	14	7	10.80
Tr / Ts	1.33	2.67	3.25	2.80	3.50	2.71
RR (q=4)						**MEAN**
FinishTIme	3	17	11	20	19	
Turnaround Time (Tr)	3	15	7	14	11	10.00
Tr / Ts	1.00	2.5	1.75	2.80	5.50	2.71

Shortest Process Next

- Non preemptive policy
- Process with shortest expected processing time is selected next
- Short process jumps ahead of longer processes

0 5 10 15 20



Shortest Process Next

- Overall performance is significantly improved in terms of response time.
  - However, the variability of response times is increased, especially for longer processes, and thus predictability is reduced.

- One difficulty with the SPN policy is the need to know or at least estimate the required processing time of each process.
  - For batch jobs, the system may require the programmer to estimate the value and supply it to the operating system.
  - If the programmer’s estimate is substantially under the actual running time, the system may abort the job.

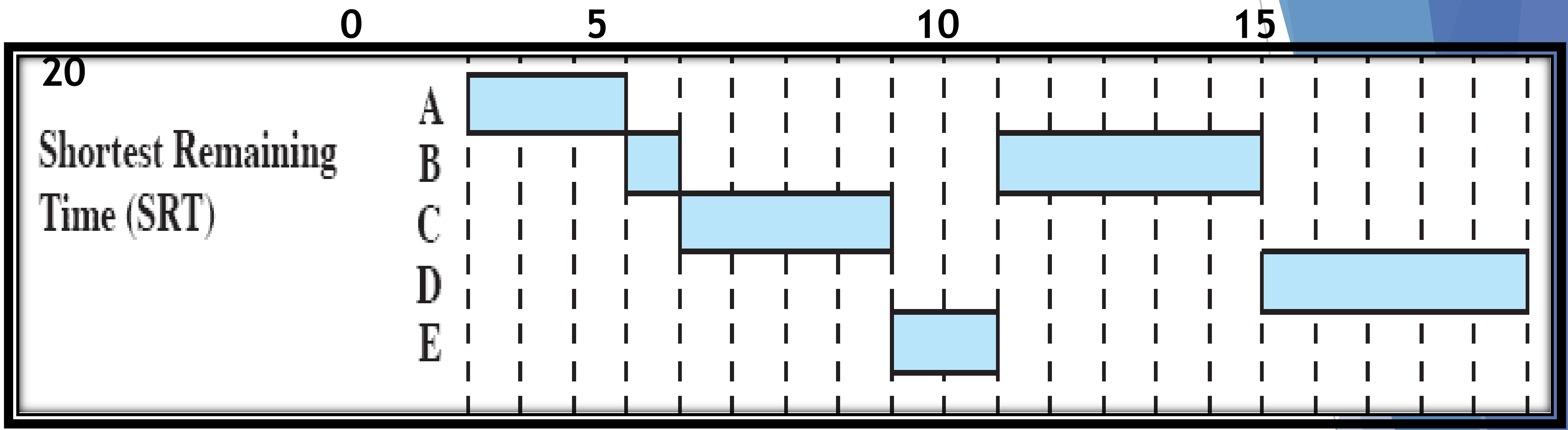
Shortest Process Next

Process	A	B	C	D	E	
Arrival TIme	0	2	4	6	8	
Service Time (Ts)	3	6	4	5	2	

SPN						MEAN
FinishTIme	3	9	15	20	11	
Turnaround Time (Tr)	3	7	11	14	3	7.60
Tr / Ts	1.00	1.17	2.75	2.80	1.50	1.84

Shortest Remaining Time

- Preemptive version of shortest process next policy
- Must estimate processing time and choose the shortest



•A pre-emptive version of SPN.

•In this case, the scheduler always chooses the process that has the shortest expected remaining
processing time.

• SRT does not have the bias in favor of long processes found in FCFS.

• Unlike round robin, no additional interrupts are generated, reducing overhead.

• On the other hand, elapsed service times must be recorded, contributing to overhead.

•SRT should also give superior turnaround time performance to SPN, because a short job is given
immediate preference to a running longer job.

Shortest Remaining Time

Process	A	B	C	D	E	
Arrival TIme	0	2	4	6	8	
Service Time (Ts)	3	6	4	5	2	

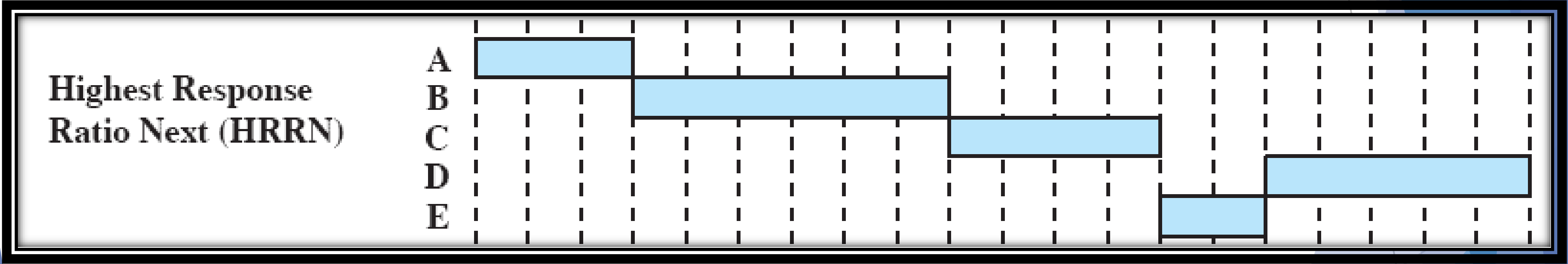
SRT						MEAN
FinishTIme	3	15	8	20	10	
Turnaround Time (Tr)	3	13	4	14	2	7.20
Tr / Ts	1.00	2.17	1.00	2.80	1.00	1.59

Highest Response Ratio Next

- Choose next process with the greatest ratio

$$Ratio = \frac{time\ spent\ waiting + expected\ service\ time}{expected\ service\ time}$$

0 5 10 15 20



A smaller denominator yields a larger ratio so that shorter jobs are favoured, but aging without service increases the ratio so that a longer process will eventually get past competing shorter jobs.

As with SRT and SPN, the expected service time must be estimated to use highest response ratio next (HRRN).

Highest Response Ratio Next

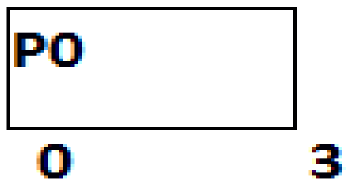
Process	A	B	C	D	E	
Arrival TIme	0	2	4	6	8	
Service Time (Ts)	3	6	4	5	2	

HRRN						MEAN
FinishTIme	3	9	13	20	15	
Turnaround Time (Tr)	3	7	9	14	7	8.00
Tr / Ts	1.00	1.17	2.25	2.80	3.5	2.14

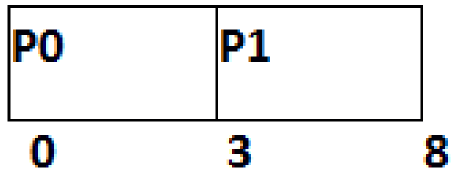
HRNN Example

Process ID	Arrival Time	Burst Time
0	0	3
1	2	5
2	4	4
3	6	1
4	8	2

At time 0, The Process P0 arrives with the CPU burst time of 3 units. Since it is the only process arrived till now hence this will get scheduled immediately



P0 is executed for 3 units, meanwhile, only one process P1 arrives at time 3. This will get scheduled immediately since the OS doesn't have a choice.

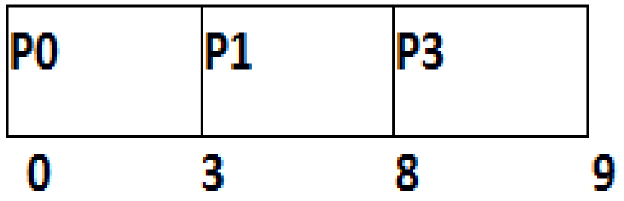


P1 is executed for 5 units. Meanwhile, all the processes get available. We have to calculate the Response Ratio for all the remaining jobs.

RR (P2) = ((8-4) +4)/4 = 2
RR (P3) = (2+1)/1 = 3
RR (P4) = (0+2)/2 = 1

```
Response Ratio = (W+S)/S
Where, W= waiting time
S= Burst time or Service time
```

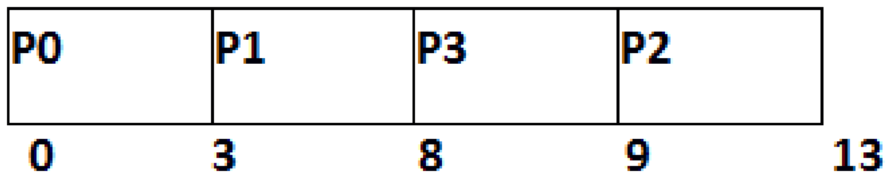
Since, the Response ratio of P3 is higher hence P3 will be scheduled first.



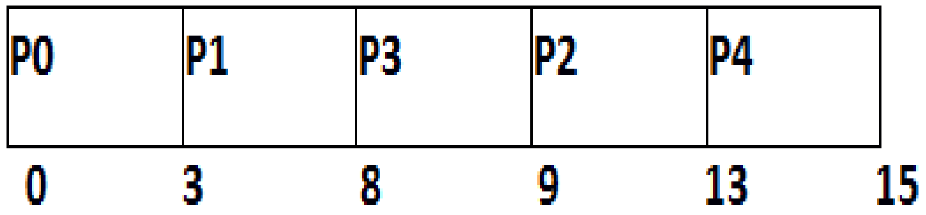
P3 is scheduled for 1 unit. The next available processes are P2 and P4. Let's calculate their Response ratio.

RR (P2) = (5+4)/4 = 2.25
RR (P4) = (1+2)/2 = 1.5

The response ratio of P2 is higher hence P2 will be scheduled.



Now, the only available process is P4 with the burst time of 2 units, since there is no other process available hence this will be scheduled.



Process ID	Arrival Time	Burst Time	Complet ion Time	Turn Around Time	Waiting Time
0	0	3	3	3	0
1	2	5	8	6	1
2	4	4	13	9	5
3	6	1	9	3	2
4	8	2	15	7	5

Average Waiting Time = 13/5

Dispatcher

Another component that is involved in the CPU-scheduling function is the dispatcher, which is the module that gives control of the CPU to the process selected by the short-term scheduler. It receives control in kernel mode as the result of an interrupt or system call.

The functions of a dispatcher mop the following:

- Context switches, in which the dispatcher saves the state (also known as context) of the process or thread that was previously running; the dispatcher then loads the initial or previously saved state of the new process.
- Switching to user mode.
- Jumping to the proper location in the user program to restart that program indicated by its new state.

The dispatcher should be as fast as possible, since it is invoked during every process switch.
During the context switches, the processor is virtually idle for a fraction of time, thus unnecessary context switches should be avoided.
The time it takes for the dispatcher to stop one process and start another is known as the dispatch latency.

UNIT 2 COMPLETED