UNIT-1 PART-2
PROCESS AND THREADS

Roadmap - part-2.1

- Process & Threads
  - **The concept of Process, PCB**
  - Creation and Termination of Process,
  - Process States and Models,
  - Process and Threads,
  - Types of Threads,
  - Process Vs. Threads

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.

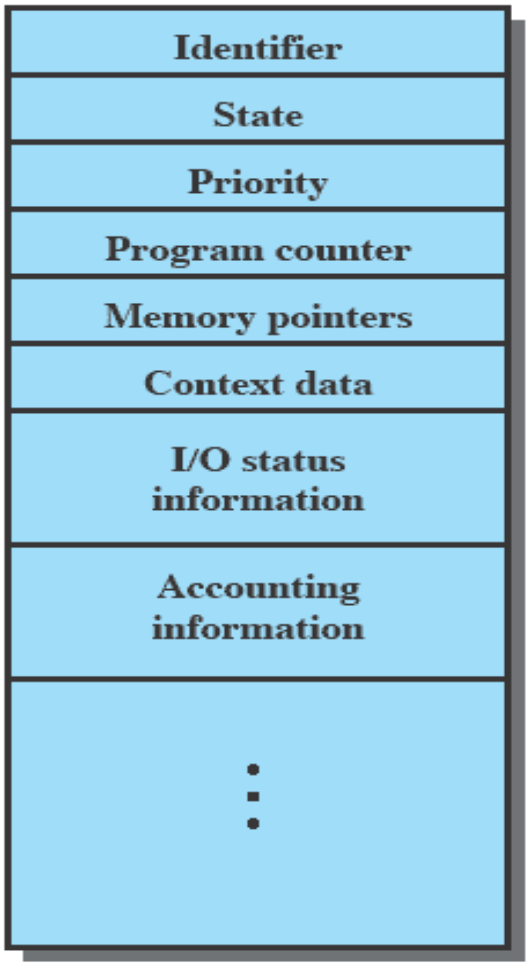


Figure 3.1 Simplified Process Control Block

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.

Roadmap - Part-1

- Process & Threads
  - The concept of Process, PCB
  - **Creation and Termination of Process,**
  - **Process States and Models,**
  - Process and Threads,
  - Types of Threads,
  - Process Vs. Threads

Two-State Process Model

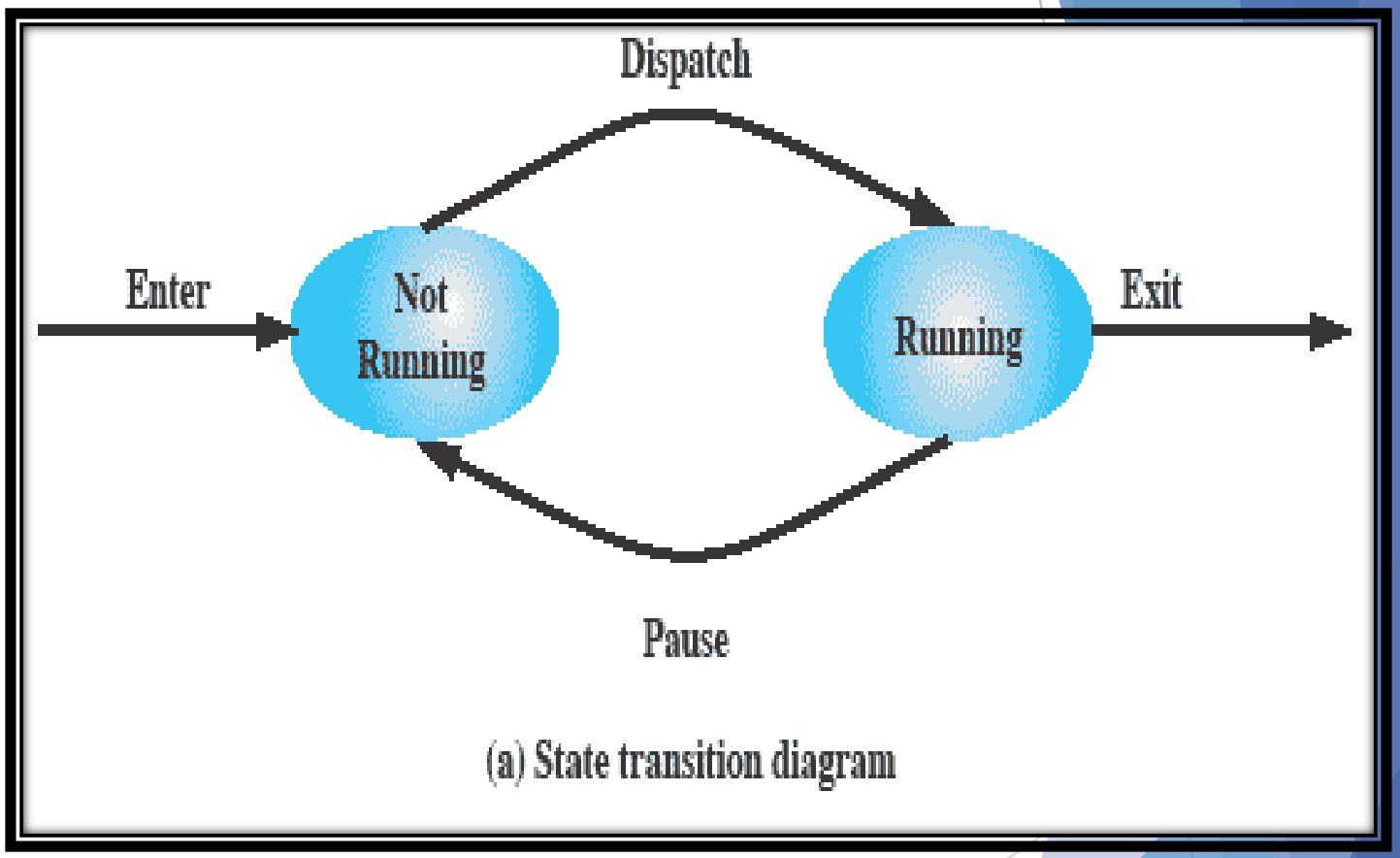
- The operating system's principal responsibility is controlling the execution of processes;
  - ✓This includes determining the interleaving pattern for execution and allocating resources to processes.
- The first step in designing an OS to control processes is to describe the behaviour that we would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

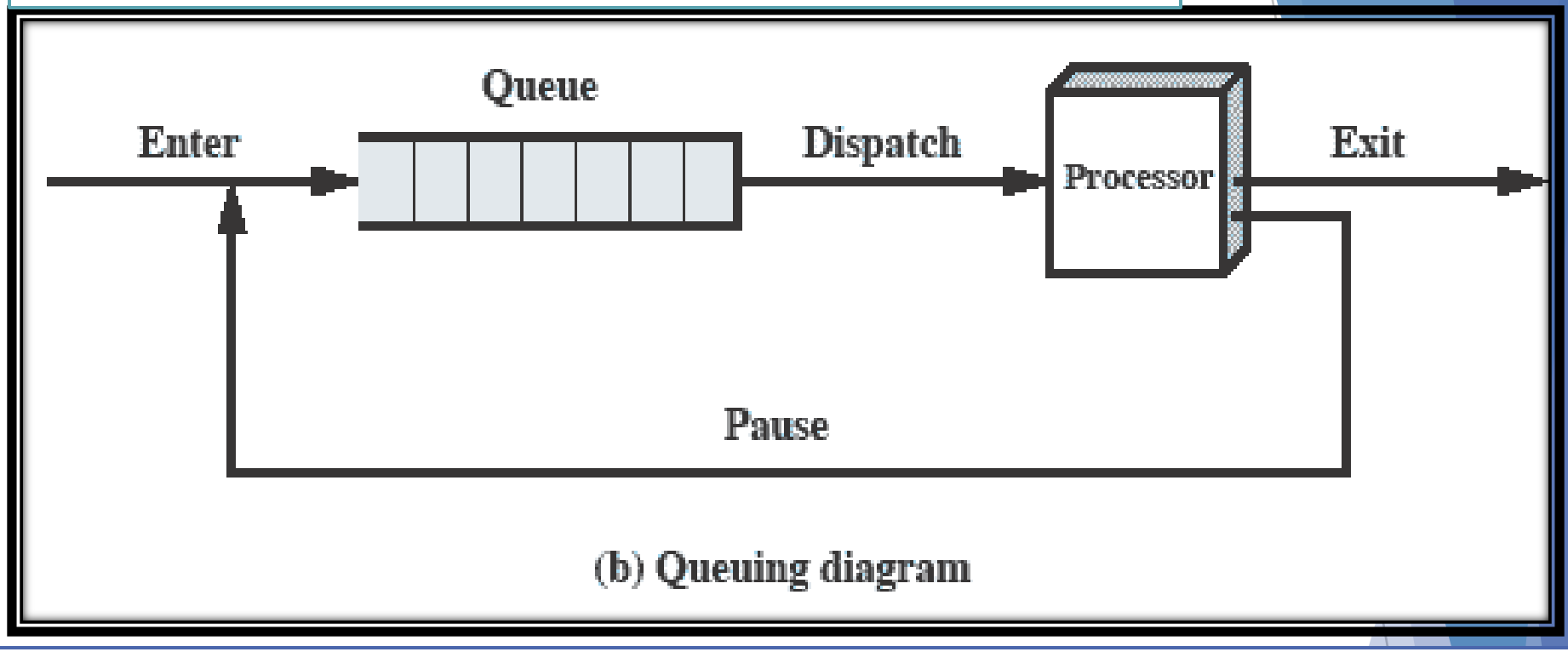
- The process exists, is known to the OS, and is waiting for an opportunity to execute.
- From time to time, the currently running process will be interrupted and the dispatcher portion of the OS will select some other process to run.

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.



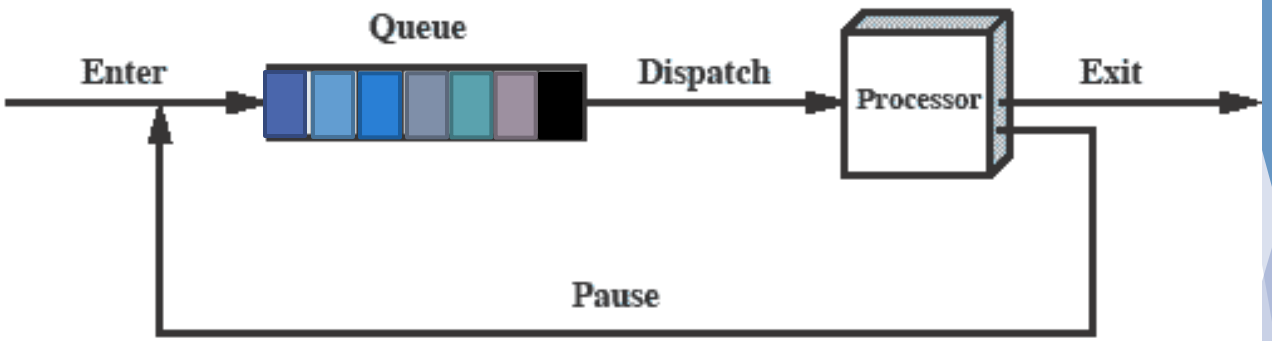
Queuing Diagram for Two state Process Model



(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

Process Creation and Termination

Process Creation

- When a new process is to be added to those currently being managed,
- The OS builds the data structures that are used to manage the process and allocates address space in main memory to the process.
- **Four** common events lead to the creation of a process, as indicated in Table 3.1.
- **1st,** In a batch environment, a process is created in response to the submission of a job.
- **2nd** In interactive environment, a process is created when a new user logs on.
- **3rd** An OS may create a process on behalf of an application. For example, if a user requests for printing, Os creates a process that will manage the printing.
- **4th** One process will create the another. For example, an application process may generate another process to receive data that the application is generating.
  - **Process Spawning:-**
    - When the OS creates a process at the explicit request of another process, the action is referred to as **process spawning.**

Process Creation

Table 3.1 Reasons for Process Creation

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Process Termination

- There must be some way that a process can indicate completion.
- This indication may be:
  - A HALT instruction generating an interrupt alert to the OS.
  - A user action (e.g. log off, quitting an application)
  - A fault or error
  - Parent process terminating

Table 3.2 Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.

Table 3.2 Reasons for Process Termination

Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

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

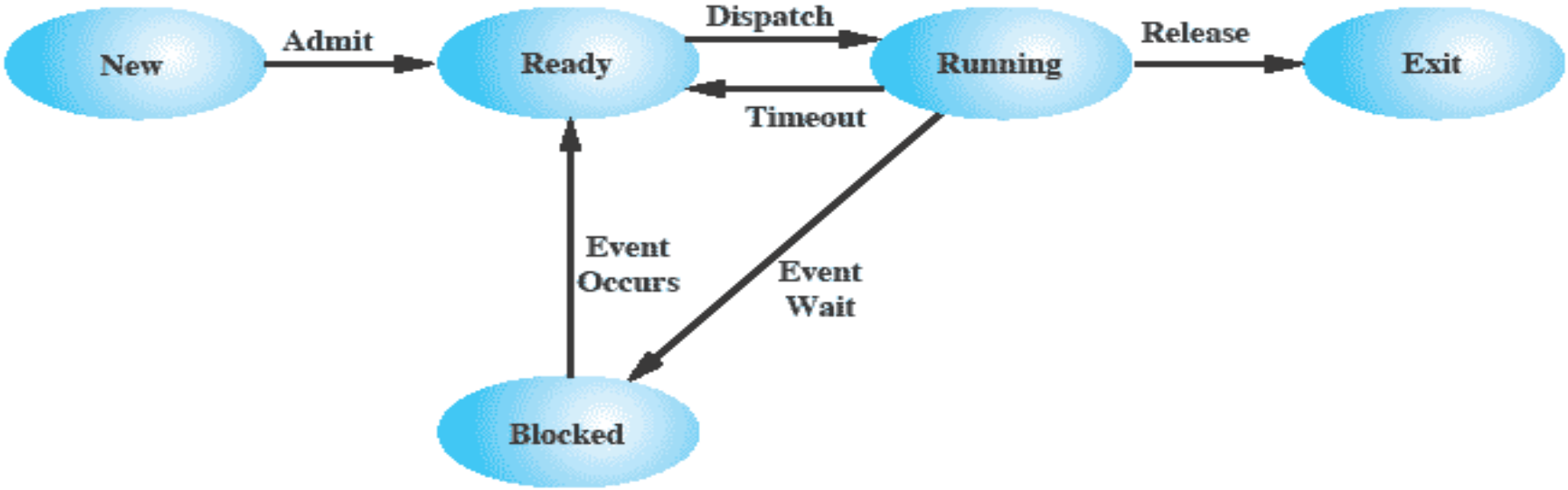


Figure 3.6 Five-State Process Model

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 :-

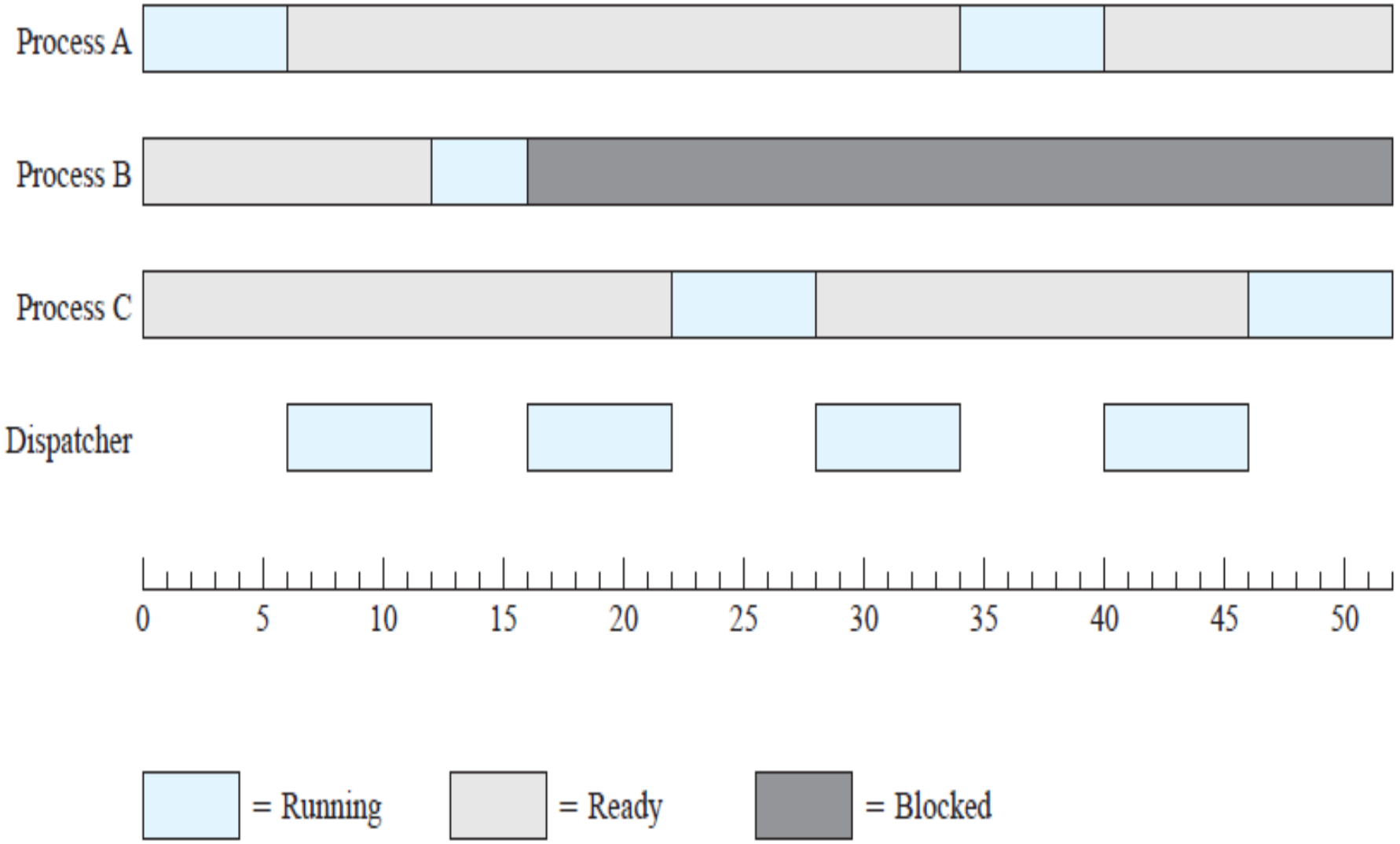
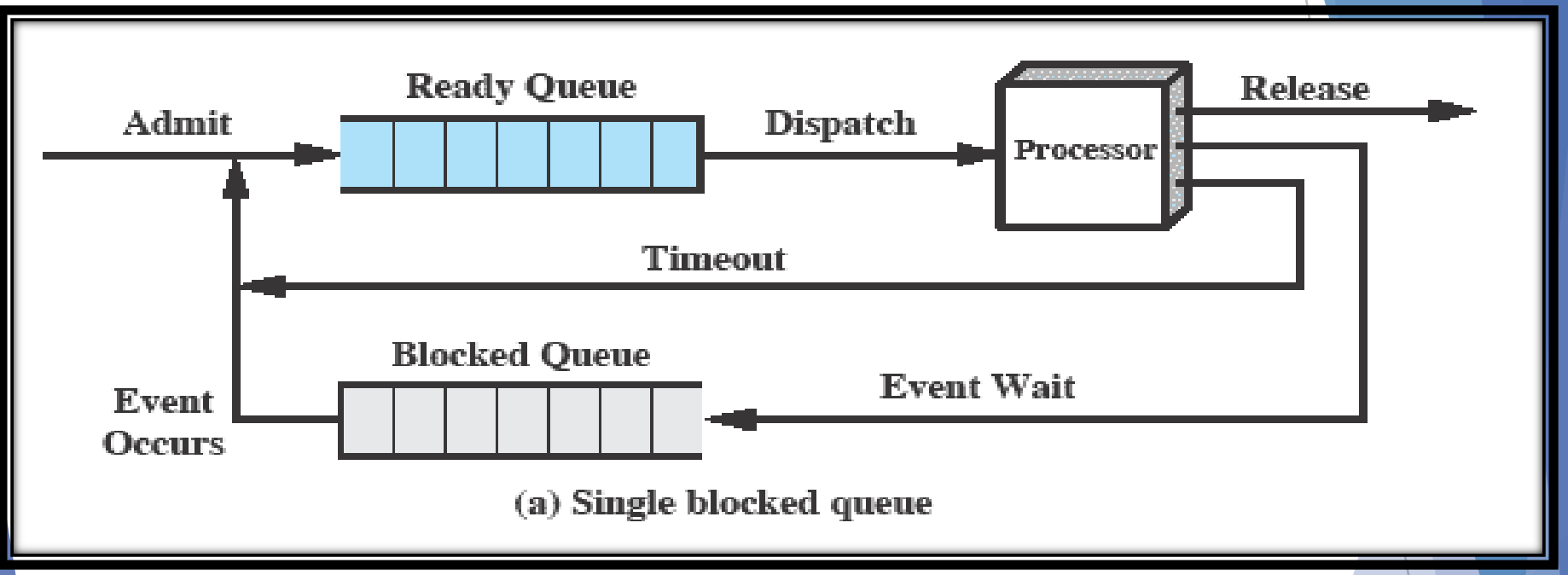


Figure 3.7 Process States for the Trace of Figure 3.4

Queuing Diagram for Two state Process Model

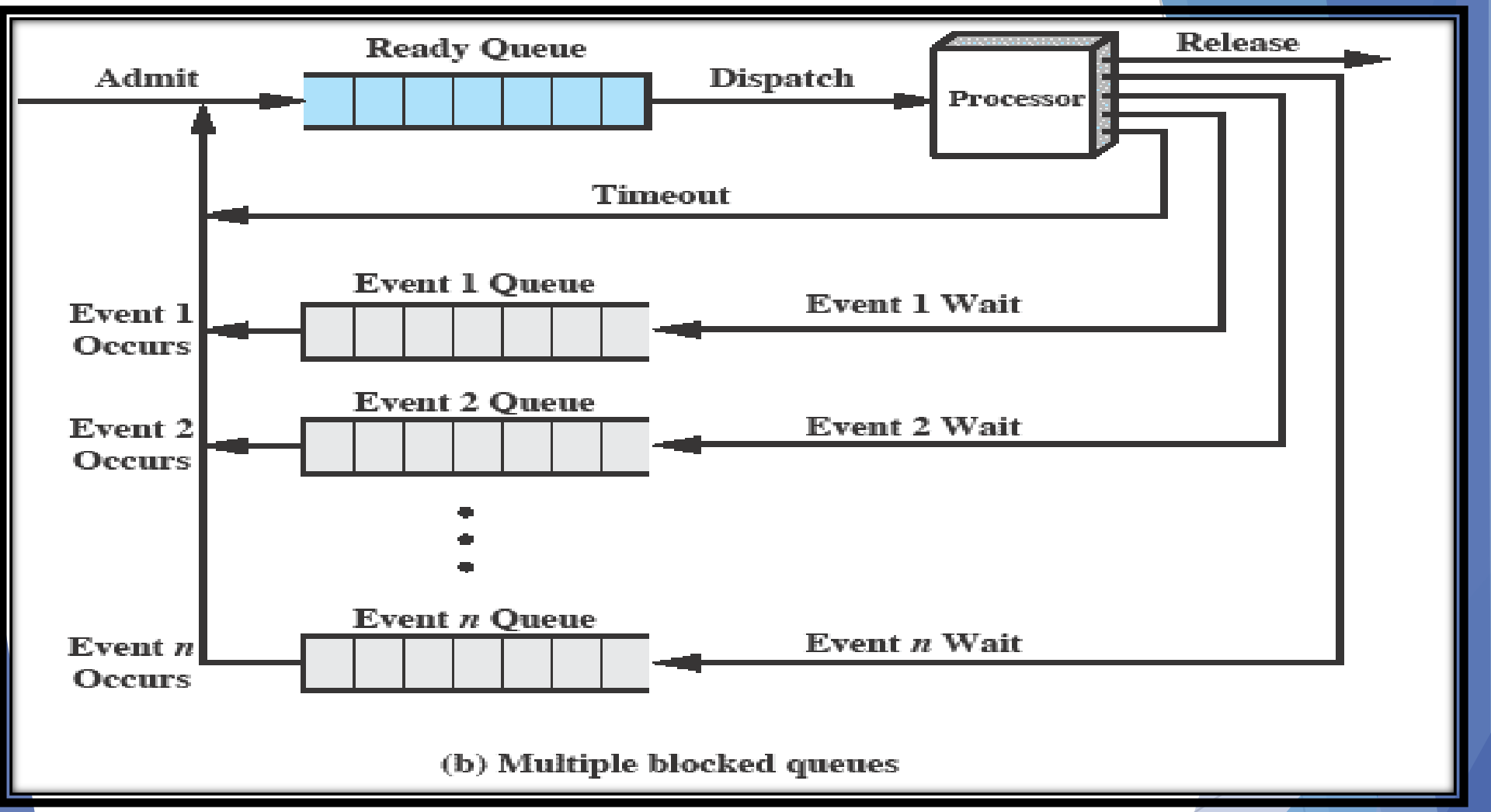
(1) Using Two Queues:-



(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:-

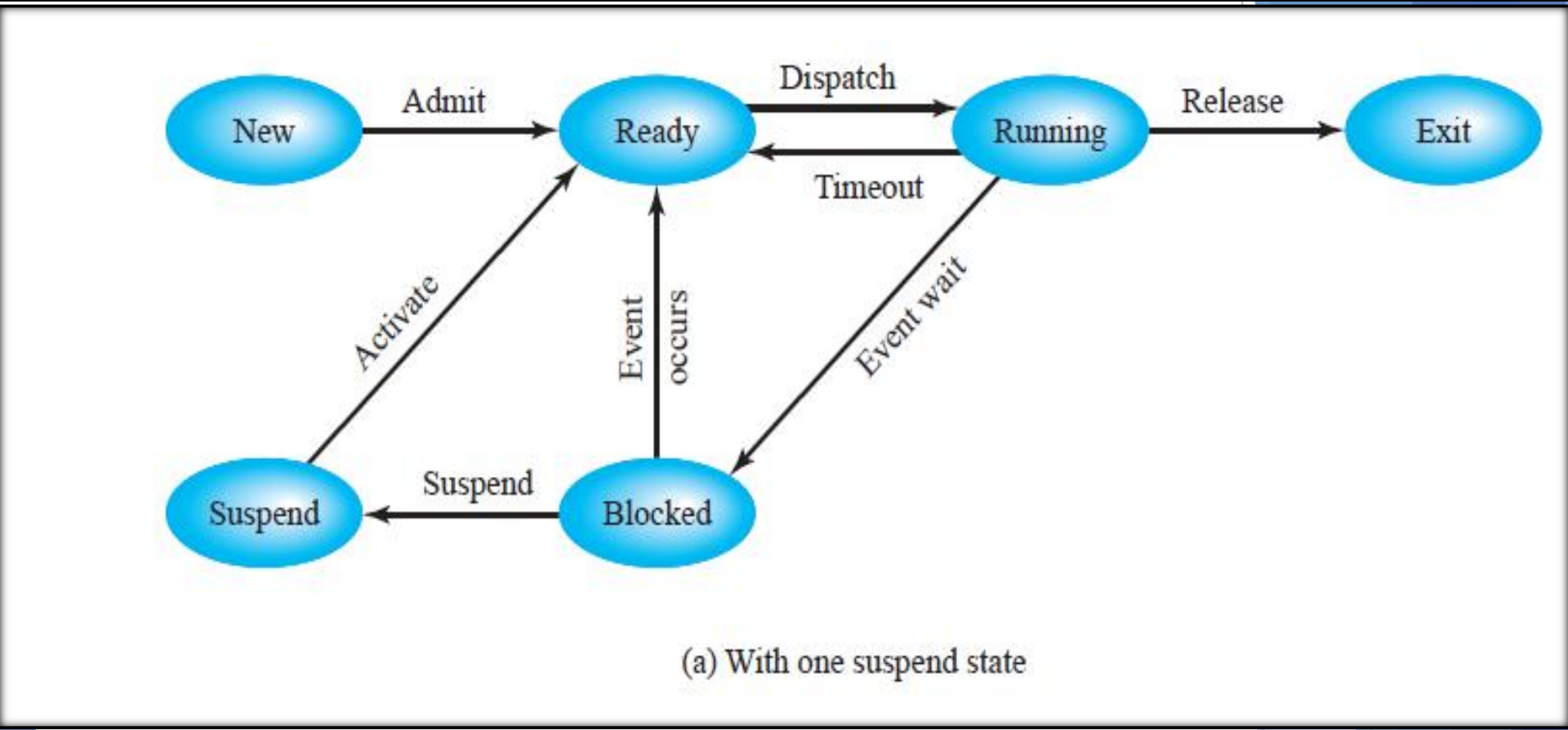


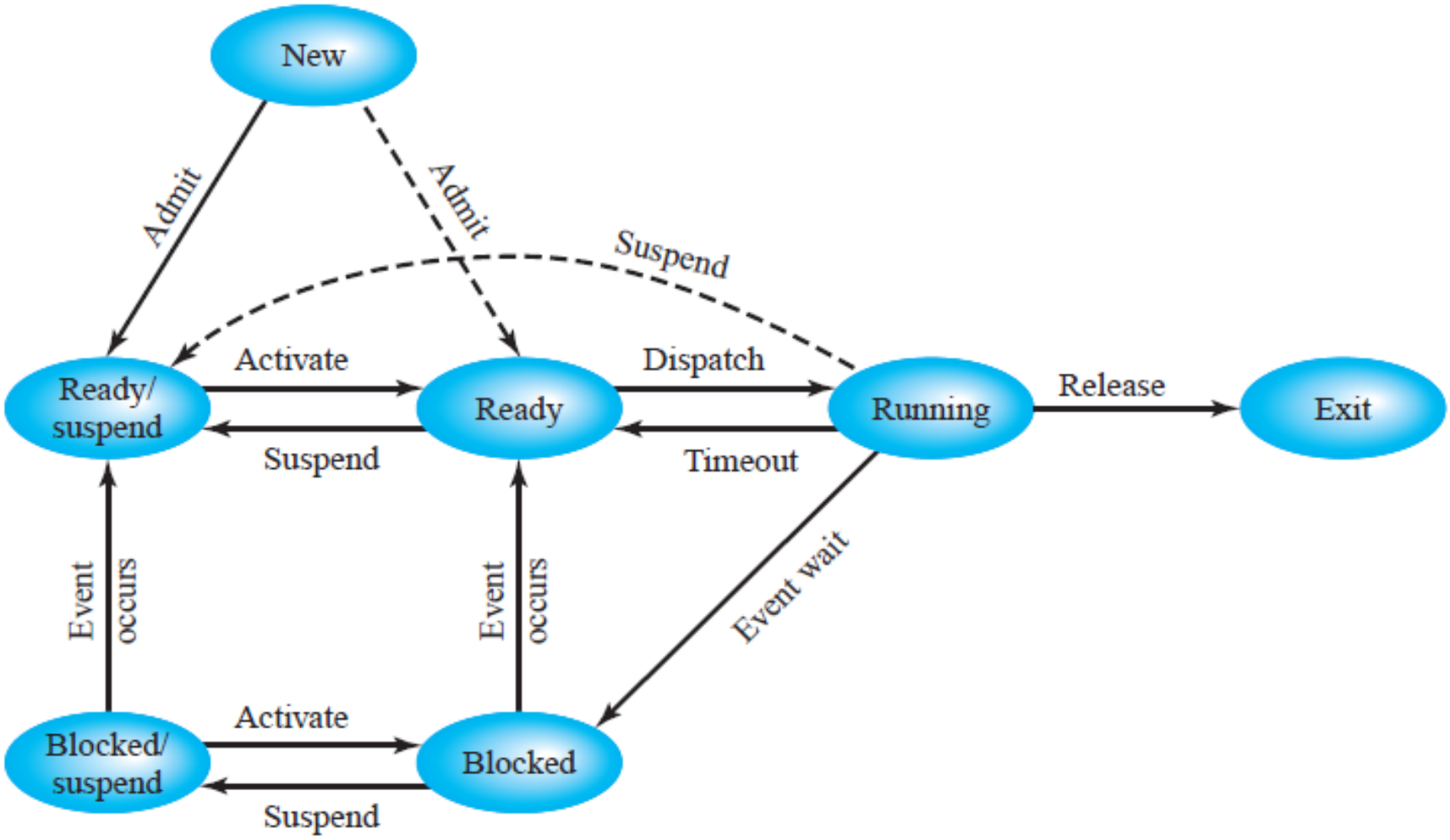
Figure 3.9 Process State Transition Diagram with Suspend States

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.

OS Control Tables

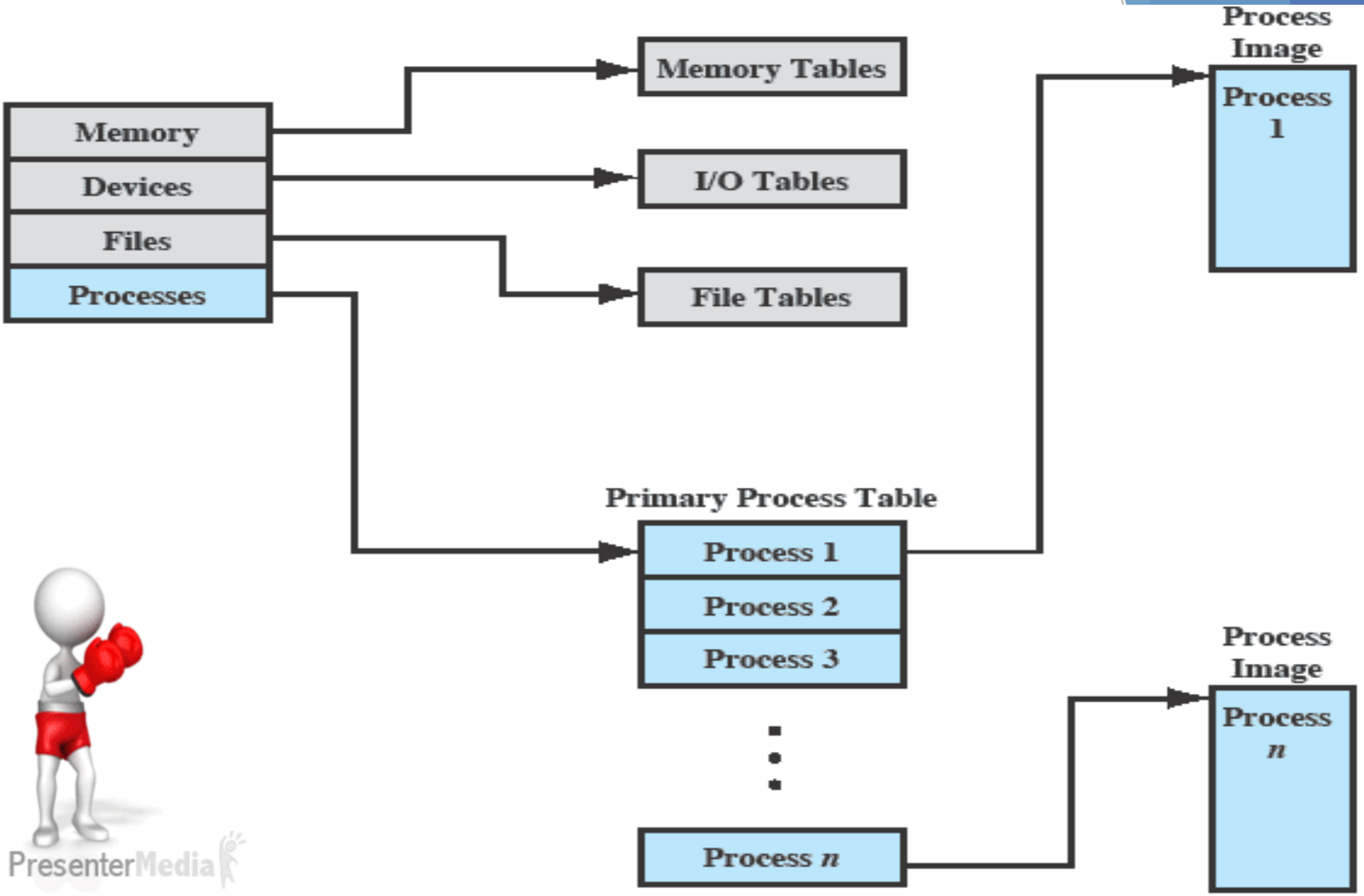


Figure 3.11 General Structure of Operating System Control Tables

Modes of Execution

- Most processors support at least two modes of execution:-
  - **User mode :**The **less-privileged mode** is referred to as the **user mode**,
    - ✓User programs typically would execute in this mode.
  - **Kernel mode :**The **more-privileged mode** is known as the **system , control mode, or kernel mode.**
    - ✓Kernel of the OS, which is that portion of the OS that encompasses the important system functions.
- Certain instructions can only be executed in the more-privileged mode.
  - Including reading or altering a control register, such as the program status word;
  - primitive I/O instructions;
  - Instructions that relate to memory management.

- In addition, certain regions of memory can only be accessed in the more-privileged mode.
- Table 3.7 lists the functions typically found in the kernel of an OS.

- Typically, when a user makes a call to an operating system service or when an operating system routine is executed, the mode is set to the kernel mode and,

  upon return from the service to the user process, the mode is set to user mode.

Roadmap – part 2.1

- Process & Threads
  - The concept of Process, PCB
  - Creation and Termination of Process,
  - Process States and Models,
  - **Process and Threads,**
  - **Types of Threads,**
  - **Process Vs. Threads**

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.

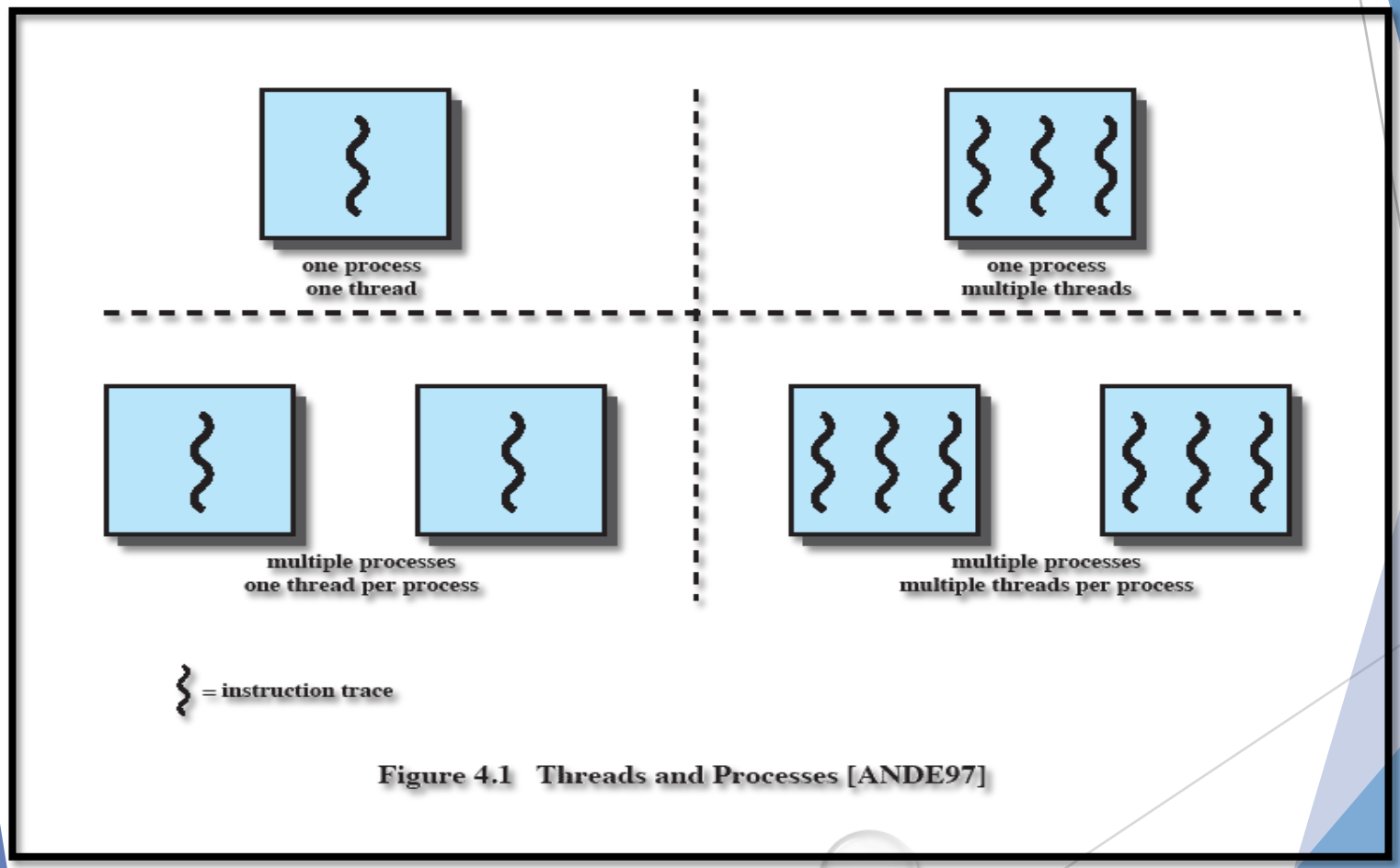


Figure 4.1 Threads and Processes [ANDE97]

Single Thread Approaches

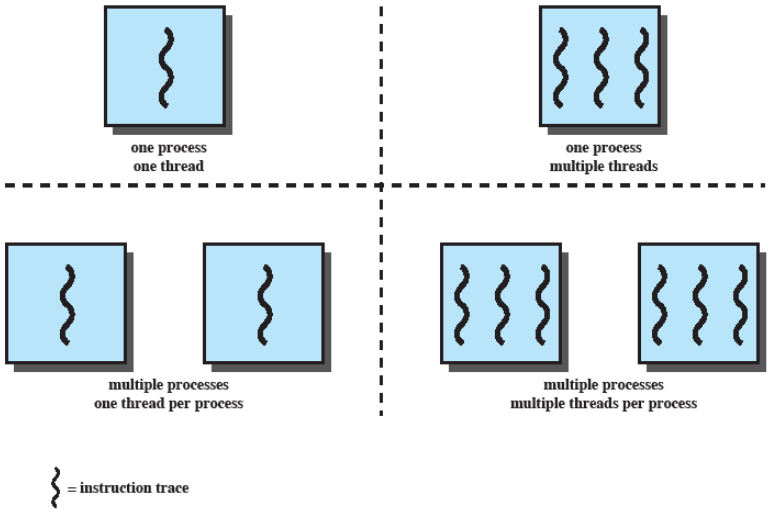


Figure 4.1 Threads and Processes [ANDE97]

- 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

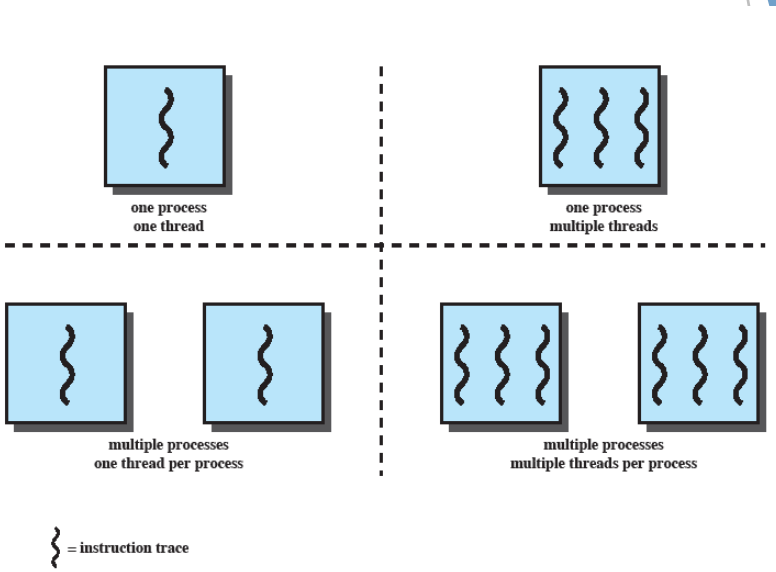


Figure 4.1 Threads and Processes [ANDE97]

- 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.

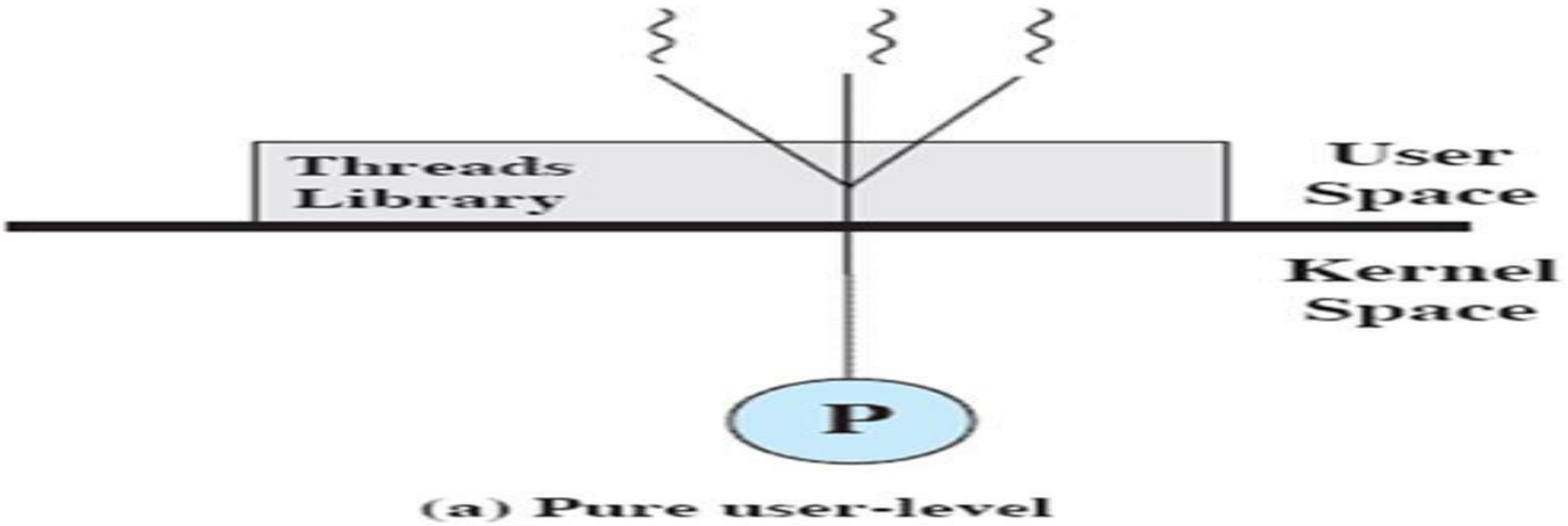
Categories of Thread Implementation

- There are two broad categories of thread implementation: user-level threads
  - User Level Thread (ULT)
  - Kernel level Thread (KLT) also called:
    - kernel-supported threads
    - lightweight processes.

User Threads

- Thread management done by user-level threads library
- Supported above the kernel, via a set of library calls at the user level.
  - Threads do not need to call OS and cause interrupts to kernel - fast.
- Example thread libraries:
  - POSIX threads
  - Win32 threads
  - Java threads

User-Level Threads



(a) Pure user-level

- All thread management is done by the application
- The kernel is not aware of the existence of threads
- Figure 4.6a illustrates the pure ULT approach.
- Any application can be programmed to be multithreaded by using a threads library, which is a package of routines for ULT management.
- By default, an application begins with a single thread and begins running in that thread.
- This application and its thread are allocated to a single process managed by the kernel.

Disadvantages Of User-Level Threads

There are two distinct disadvantages of ULTs compared to KLTs:

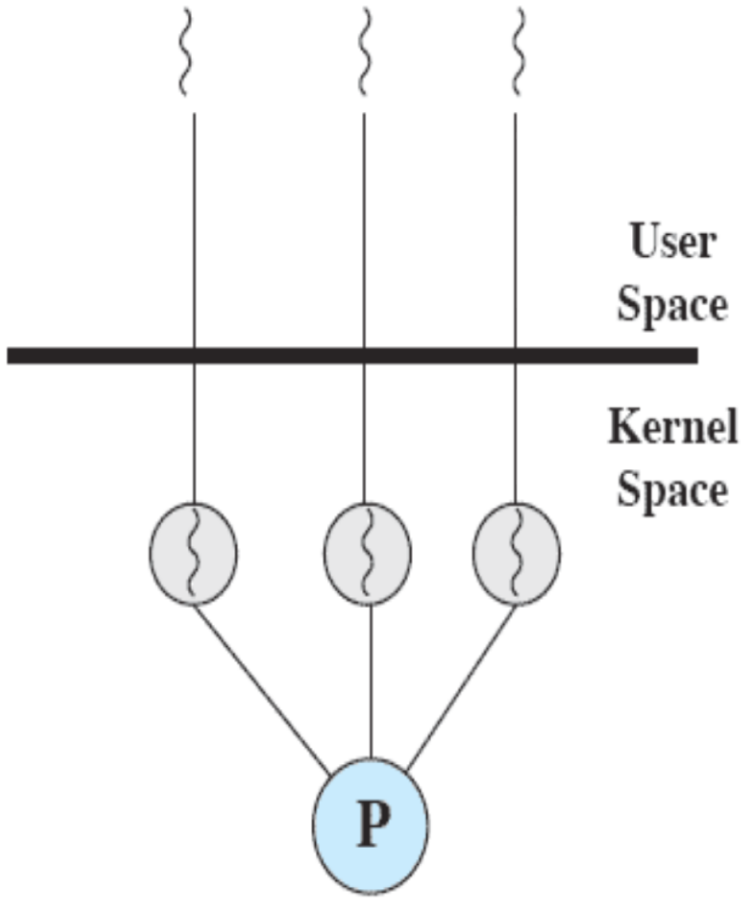
1.

- ✓ In a typical OS, many system calls are blocking.
- ✓ As a result, when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked.

2.

- ✓ In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing.
- ✓ A kernel assigns one process to only one processor at a time.
- ✓ Therefore, only a single thread within a process can execute at a time.

Kernel -Level Threads



(b) Pure kernel-level

- All of the work of thread management is done by the kernel.
- There is no thread management code in the application level.
- Figure 4.6b depicts the pure KLT approach.
- The kernel maintains context information for the process as a whole and for individual threads within the process.
- Scheduling by the kernel is done on a thread basis.

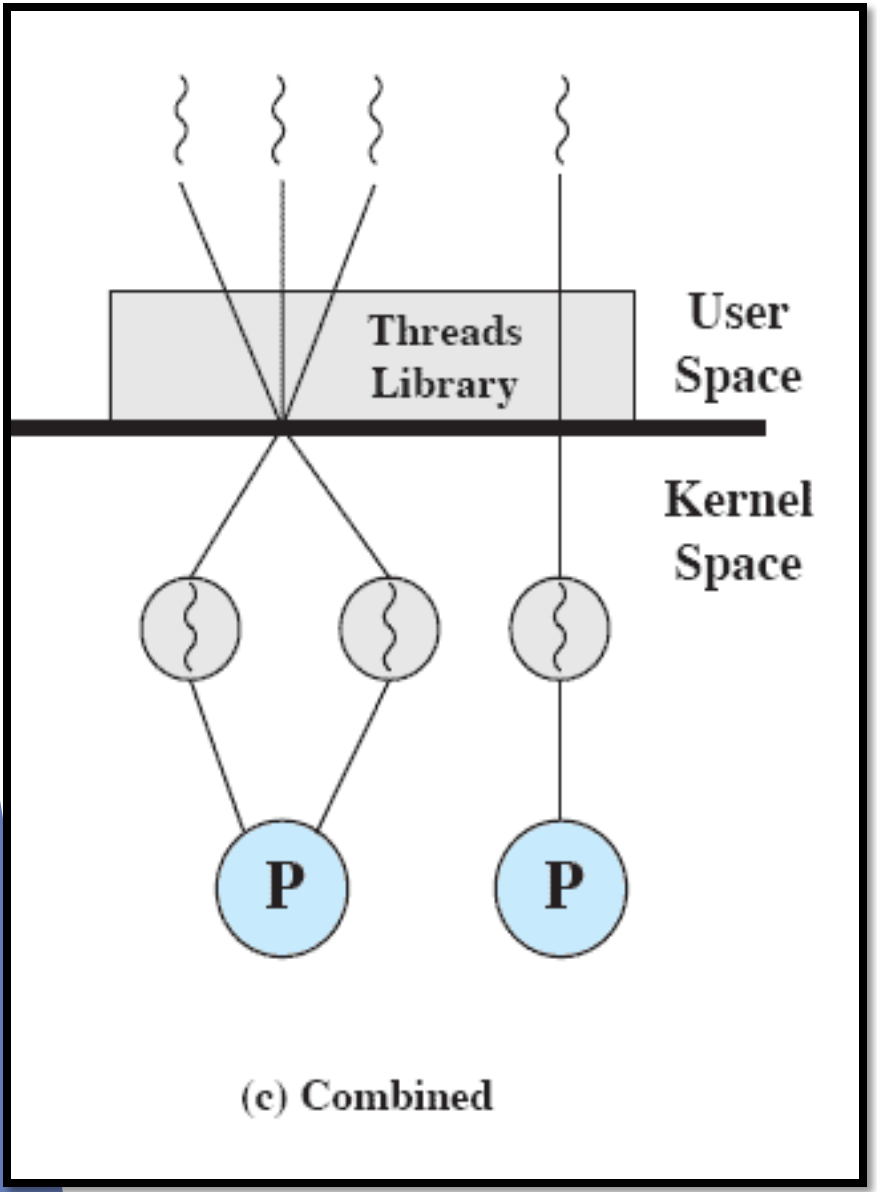
Advantages of KLT

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages of KLT

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Combined Approaches



- Thread creation is done completely in user space, as is the bulk of the scheduling and synchronization of threads within an application.
- The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs
- Multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process

ULT VS. KLT

Sr. No	User-Level Threads	Kernel-Level Threads
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage
2	Implementation is by a thread library at the user level	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system	Kernel-level thread is specific to the operating system
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded

Roadmap - Part-2.2

- Scheduling
  - Types of processor scheduling,
  - Scheduling algorithms

Scheduling

- An OS must allocate resources amongst competing processes.
- The resource provided by a processor is execution time
  - The resource is allocated by means of a schedule

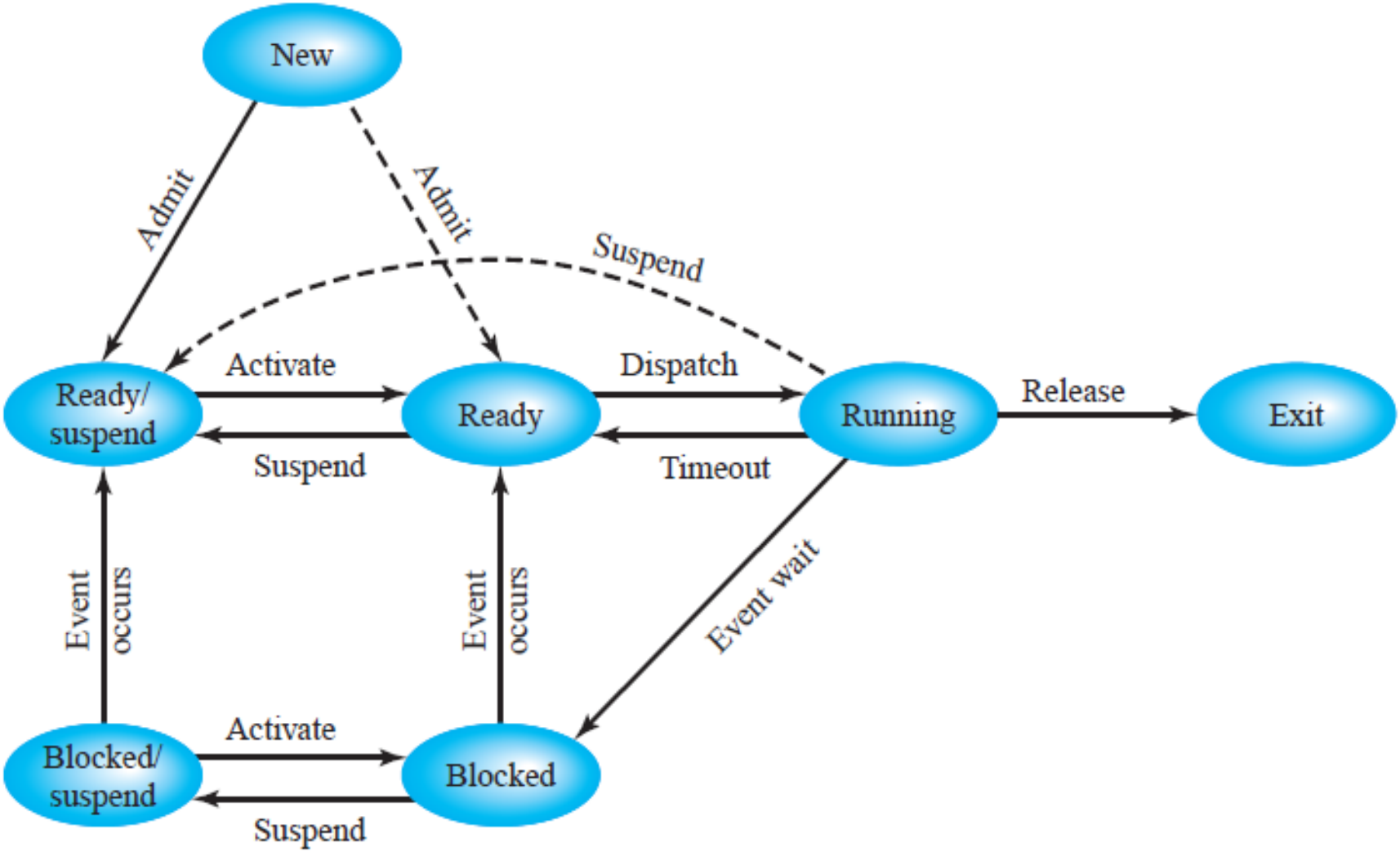
Overall Aim of Scheduling

- The aim of processor scheduling is to assign processes to be executed by the processor over time,
  - in a way that meets system objectives, such as response time, throughput, and processor efficiency.

Table 9.1 Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

Two Suspend States



(b) With two suspend states

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

Scheduling and Process State Transitions

- Figure 9.1 relates the scheduling functions to the process state transition diagram (first shown in Figure 3.9b).
- Long-term scheduling is performed when a new process is created.
- This is a decision whether to add a new process to the set of processes that are currently active.
- Medium-term scheduling is a part of the swapping function.
- This is a decision whether to add a process to those that are at least partially in main memory and therefore available for execution.
- Short-term scheduling is the actual decision of which ready process to execute next.

SCHEDULING AND PROCESS STATE TRANSITIONS

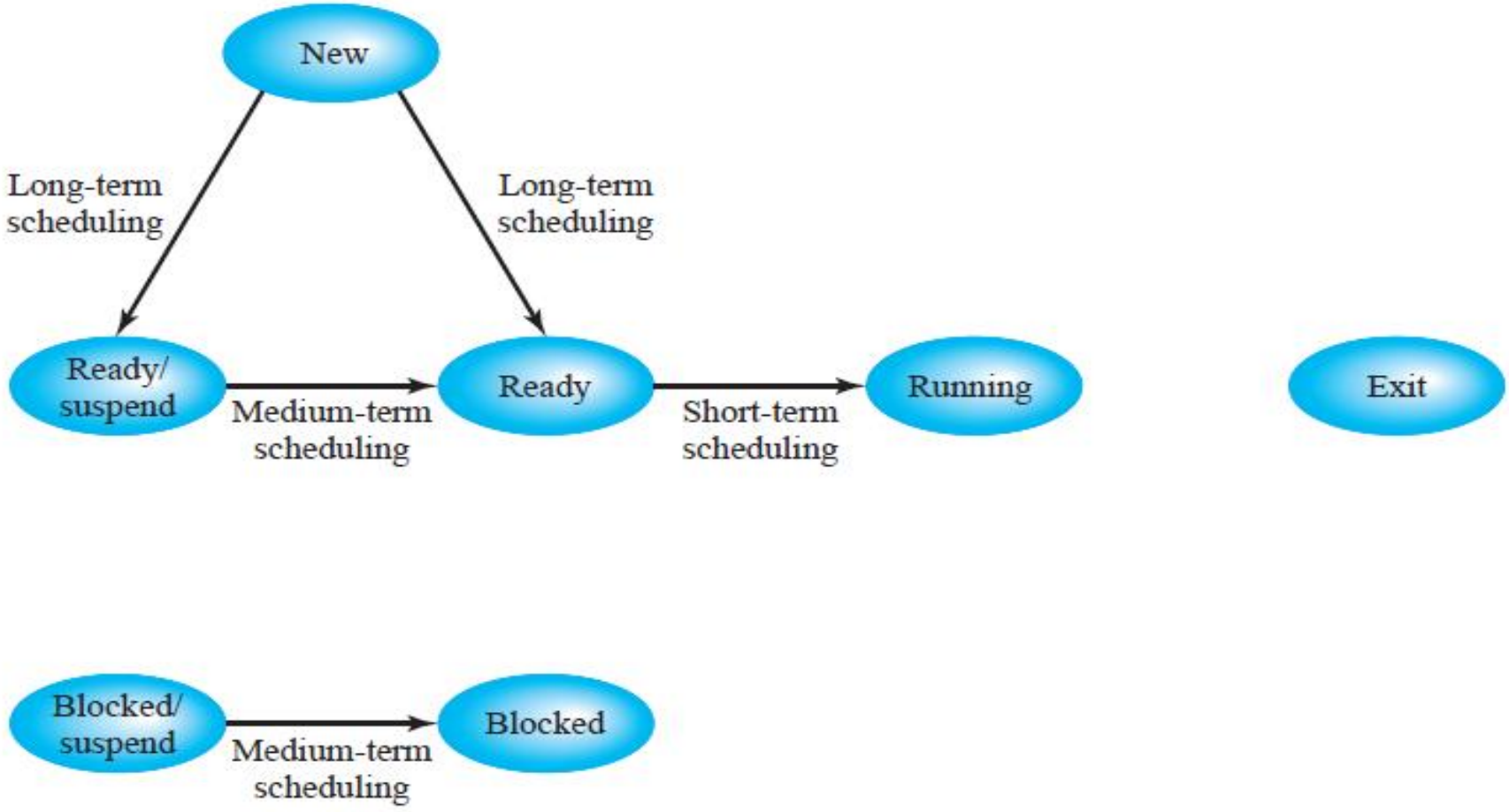


Figure 9.1 Scheduling and Process State Transitions

- **Above are the scheduling functions to the process state transition diagram for**

Nesting of Scheduling Functions

Figure 9.2 reorganizes the state transition diagram of Figure 3.9b to suggest the nesting of scheduling functions.

Nesting of Scheduling Functions

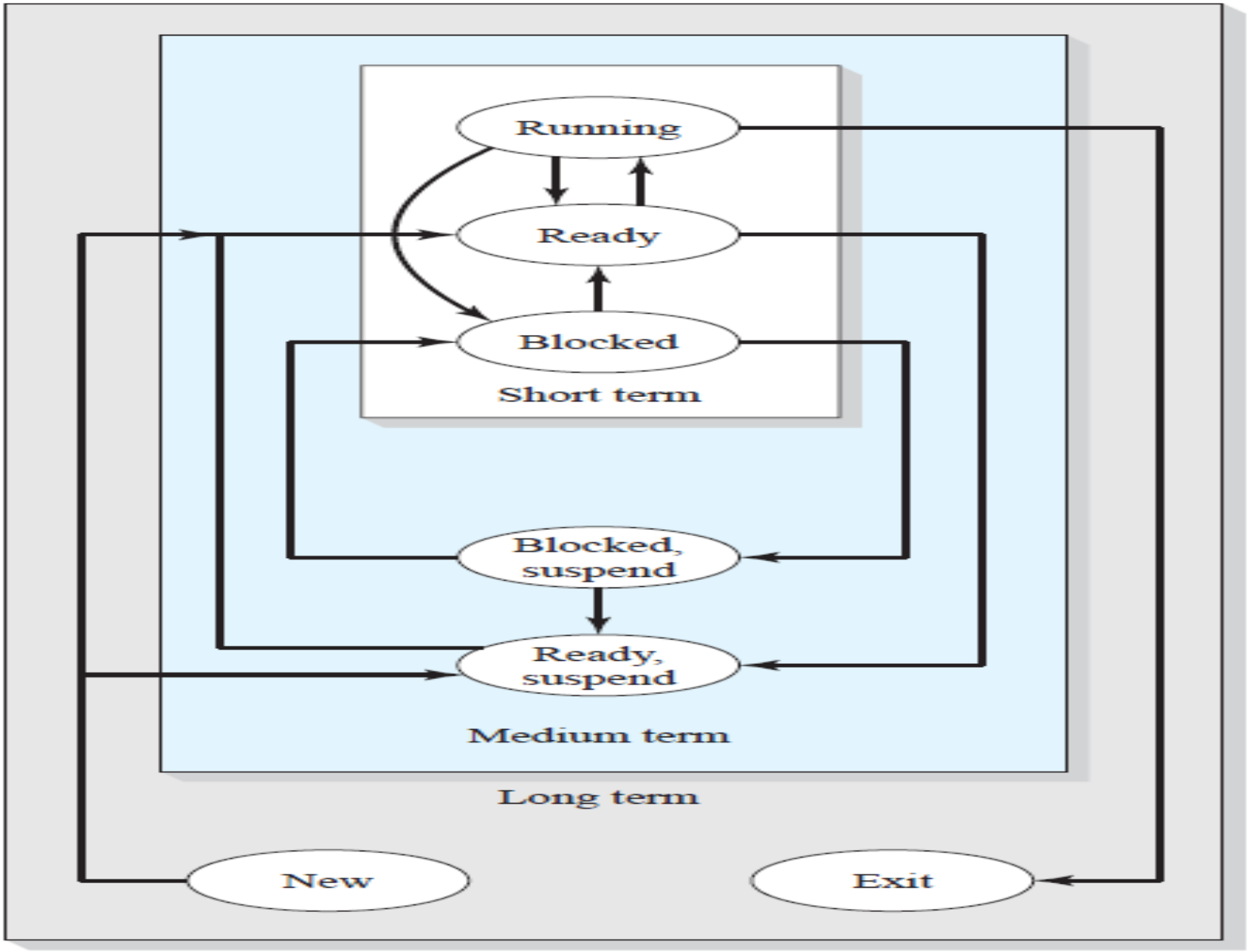


Figure 9.2 **Levels of Scheduling**

Queuing Diagram For Scheduling

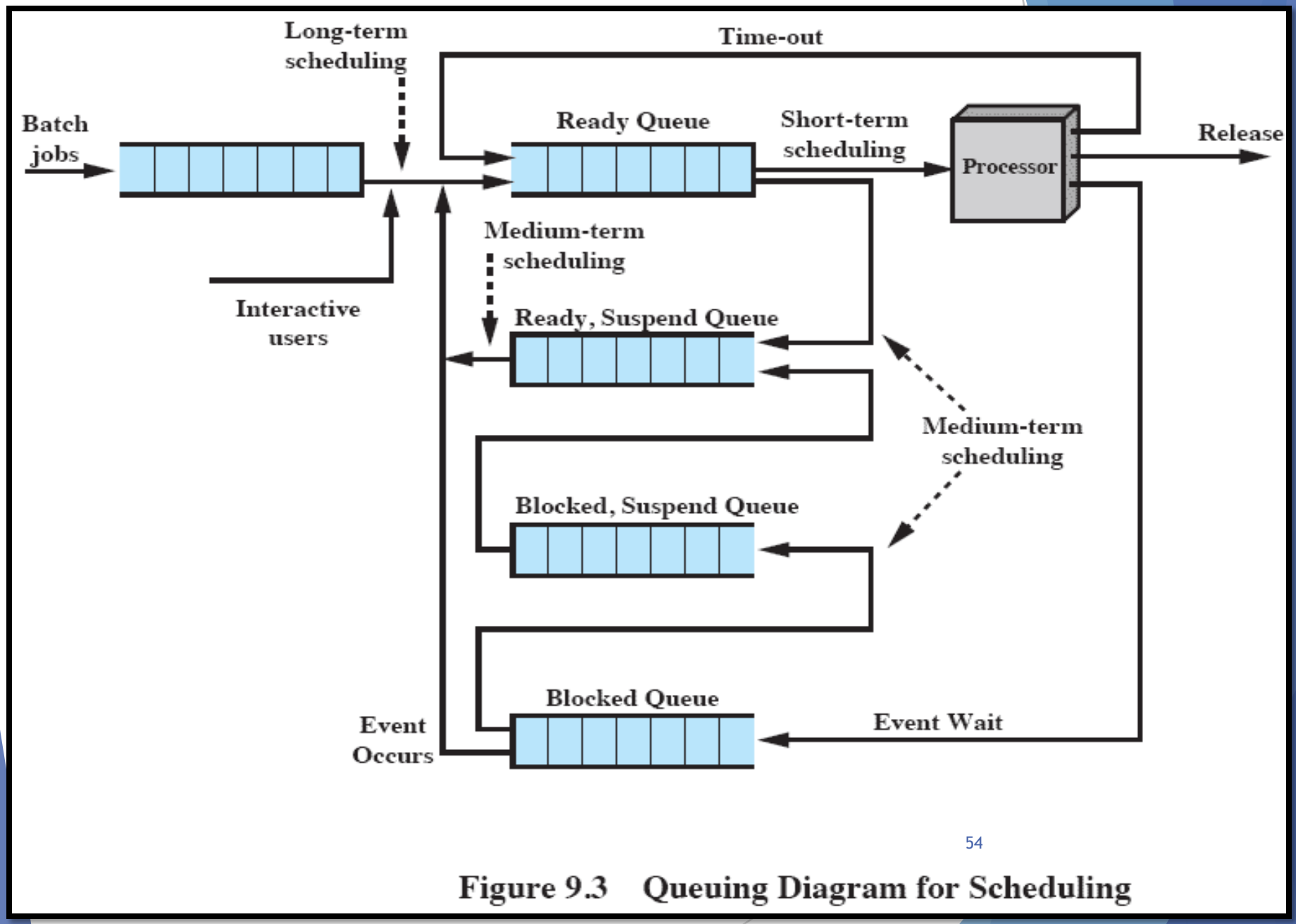


Figure 9.3 Queuing Diagram for Scheduling

Queuing Diagram For Scheduling

- This figure shows that scheduling affects the performance of the system because it determines which processes will wait and which will progress.
- The figure shows the queues involved in the state transitions of a process.
- For simplicity, new processes are shown going directly to the Ready state, whereas the earlier figures (9.1 and 9.2) show the option of either the Ready state or the Ready/Suspend state.
- Fundamentally, scheduling is a matter of managing queues to minimize queuing delay and to optimize performance in a queuing environment.

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

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)

ROADMAP – PART 2.2

- Types of Processor Scheduling

- **Scheduling Algorithms**

Aim of Short Term Scheduling

- Main objective is to allocate processor time to optimize certain aspects of system behaviour.
- A set of criteria is needed to evaluate the scheduling policy.

Short-Term Scheduling Criteria: User vs. System

•The commonly used criteria can be categorized along two dimensions.

•We can make a distinction between **user-oriented** and **system-oriented criteria**.

• **User oriented criteria** relate to the behaviour of the system as perceived by the individual user or process.

  • E.g. response time in an interactive system which is the elapsed time between the submission of a request until the response begins to appear as output.

  • We would like a scheduling policy that provides "good" service to various users.

•**System oriented criteria** is the focused on effective and efficient utilization of the processor.

  • An example is throughput, i.e. the rate at which processes are completed. Thus, throughput is of concern to a system administrator but not to the user population.

Short-Term Scheduling Criteria: Performance

- Another dimension along which criteria can be classified is those that are performance related and those that are not directly performance related.

- Performance-related criteria are quantitative and generally can be readily measured.
  - e.g. include response time and throughput.

- Criteria that are not performance related are either qualitative in nature or do not lend themselves readily to measurement and analysis.
  - e.g. predictability. We would like for the service provided to users to exhibit the same characteristics over time, independent of other work being performed by the system.

Interdependent Scheduling Criteria

Table 9.2 Scheduling Criteria

User Oriented, Performance Related

Turnaround time This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.

Response time For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user's point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.

Deadlines When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.

User Oriented, Other

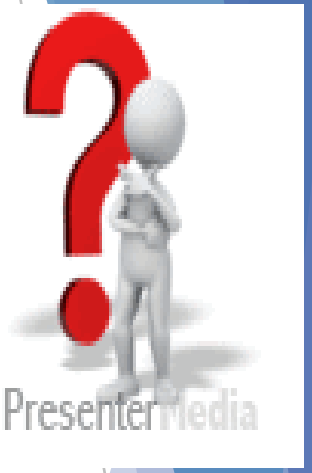
Predictability A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.

Interdependent Scheduling Criteria

- This table summarizes key scheduling criteria.

- These are interdependent, and it is impossible to optimize all of them simultaneously.
  - Thus, the design of a scheduling policy involves compromising among competing requirements;
  - The relative weights given the various requirements will depend on the nature and intended use of the system.

- In most interactive operating systems, whether single user or time shared, adequate response time is the critical requirement.



Interdependent Scheduling Criteria cont.

System Oriented, Performance Related

Throughput The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.

Processor utilization This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.

System Oriented, Other

Fairness In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.

Enforcing priorities When processes are assigned priorities, the scheduling policy should favor higher-priority processes.

Balancing resources The scheduling policy should keep the resources of the system busy. Processes that will underutilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling.

Alternative Scheduling Policies

Table 9.3 Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s − e]	$\max\left(\frac{w+s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

Selection Function

- **Selection function** determines which process, among ready processes, is selected next for execution.
  - May be based on priority, resource requirements, or the execution characteristics of the process.

- In the latter case, three quantities are significant:
  - ***w*** = time spent in system so far, waiting
  - ***e*** = time spent in execution so far
  - ***s*** = total service time required by the process, including *e*;

Decision Mode

- Specifies the instants in time at which the selection function is exercised.
- Two categories:
  - Nonpreemptive
  - Preemptive

Non preemptive vs. Preemptive

Non-preemptive

In this case, once a process is in the Running state, it continues to execute until

(a)it terminates or

(b)it blocks itself to wait for I/O or to request some operating system service.

Pre-emptive:

The currently running process may be interrupted and moved to the Ready state by the operating system.

The decision to preempt may be performed

- when a new process arrives;
- when an interrupt occurs that places a blocked process in the Ready state;
- or periodically, based on a clock interrupt.

Process Scheduling Example

- Example set of processes, consider each a batch job

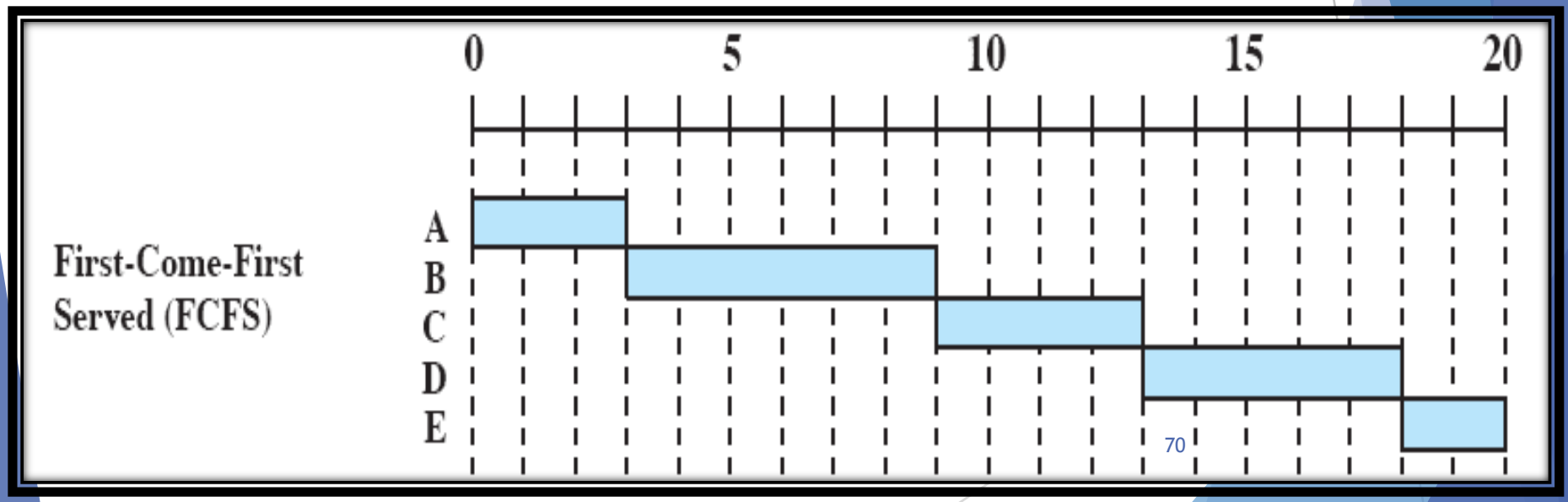
Table 9.4 Process Scheduling Example

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(AT)	0	2	4	6	8	
Service Time (Ts)	3	6	4	5	2	

FCFS						MEAN
FinishTIme(FT)	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

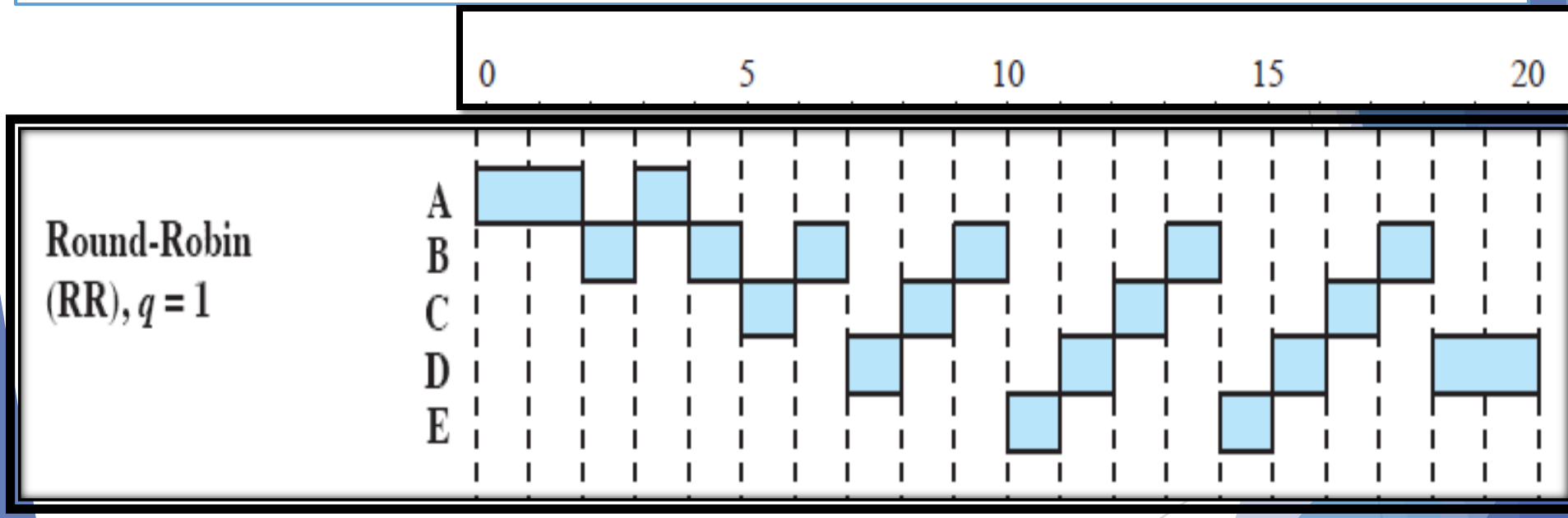
FIRST-COME- FIRST-SERVED

- 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

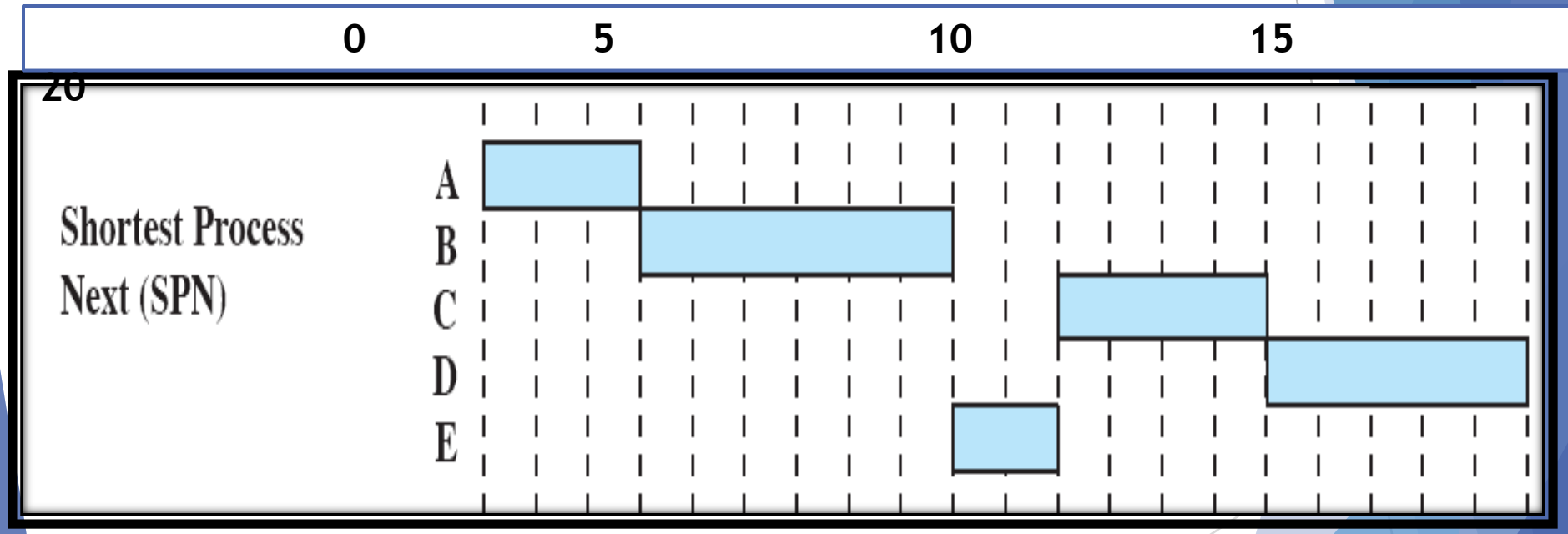
Round Robin

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



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.

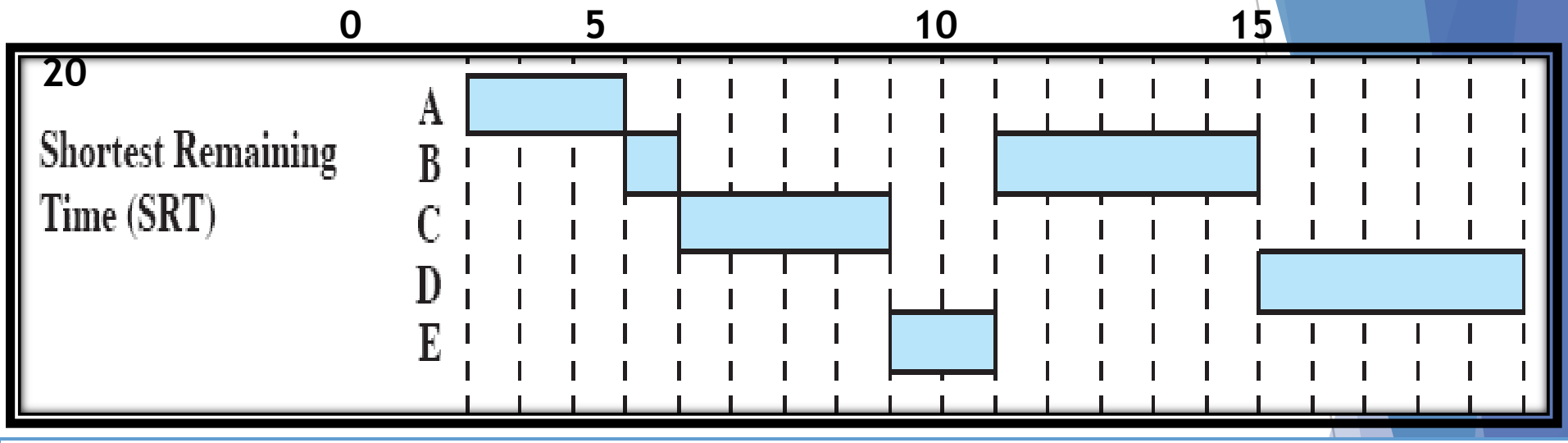
Shortest Process Next

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

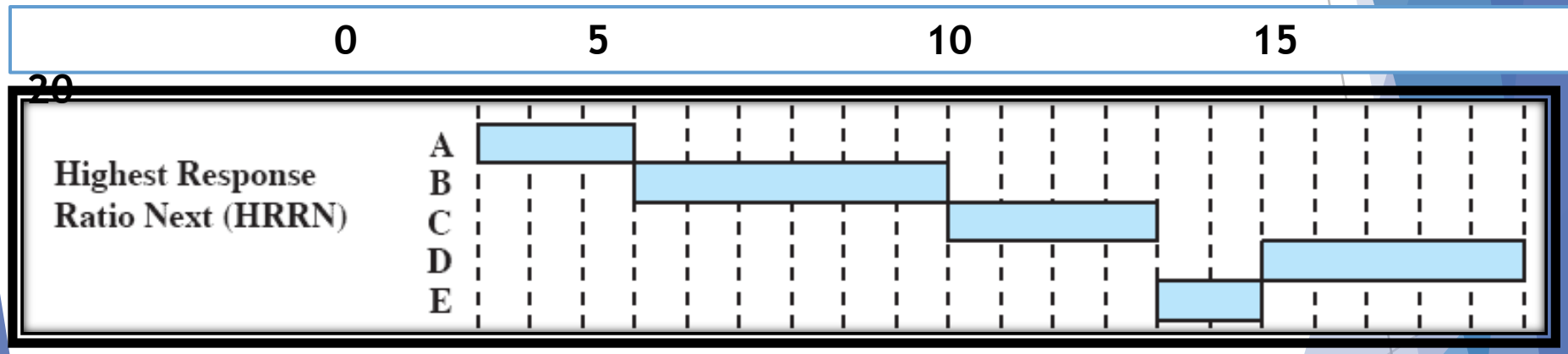
Process	A	B	C	D	E	
Arrival TIme	0	2	4	6	8	
Service Time (Ts)	3	6	4	5	2	

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

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



A smaller denominator yields a larger ratio so that shorter jobs are favored, 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