Operating systems

Lecture 3, Michal Vrábel, 23/10/2019

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

Process Management Activities

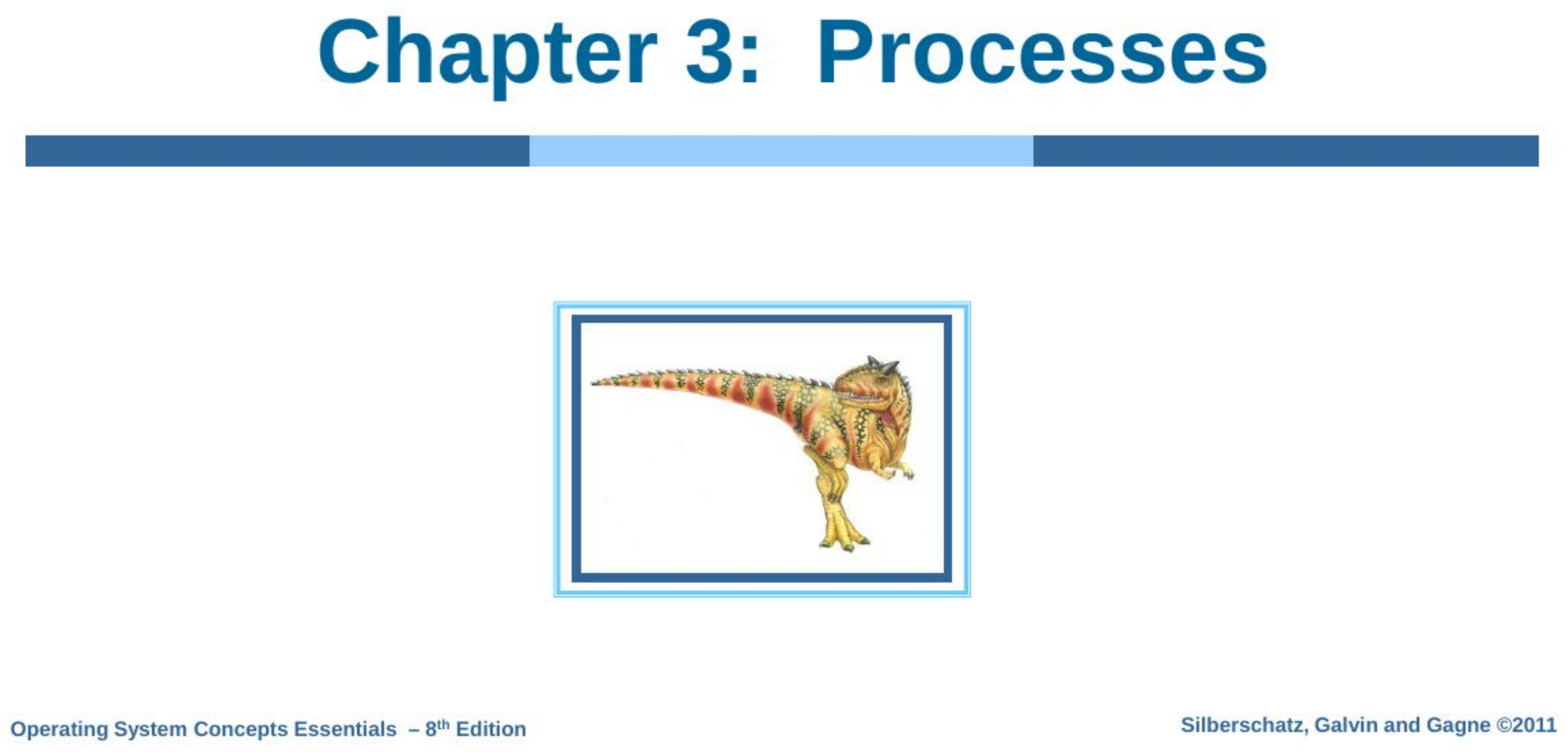
The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Processes

https://www.os-book.com/OS8/os8c/slide-dir/

https://www.os-book.com/OS8/os8c/slide-dir/PDF-dir/ch3.pdf



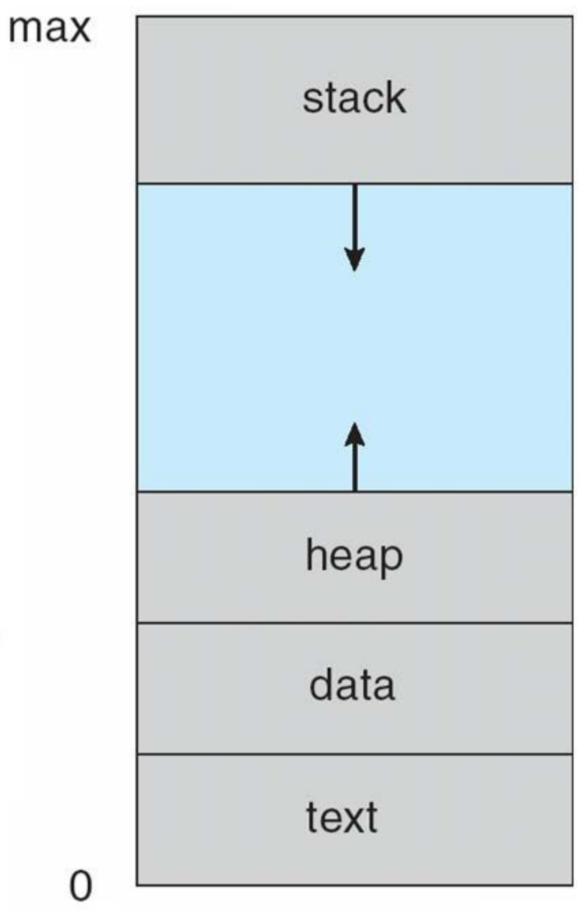
Chapter 3: Processes

Operating System Concepts Essentials – 8th Edition

Silberschatz, Galvin and Gagne ©2011

The Process

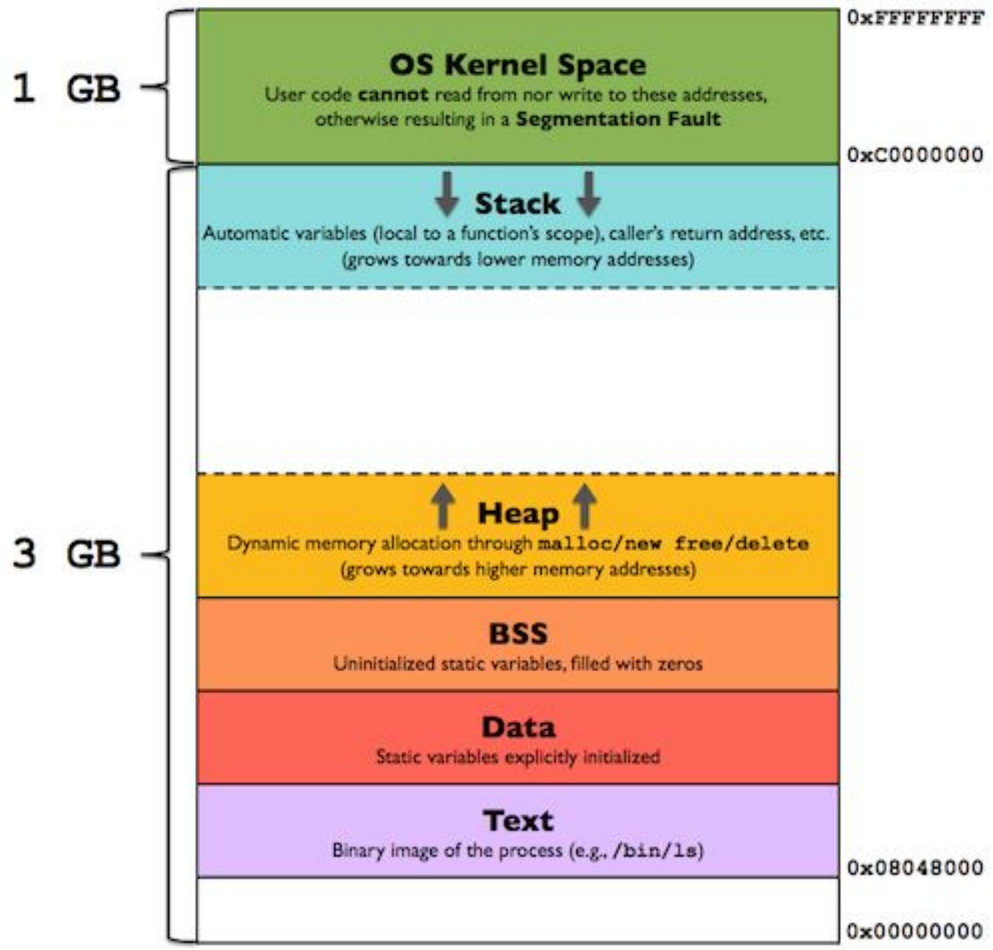
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program



Memory layout

In-Memory Layout of a program

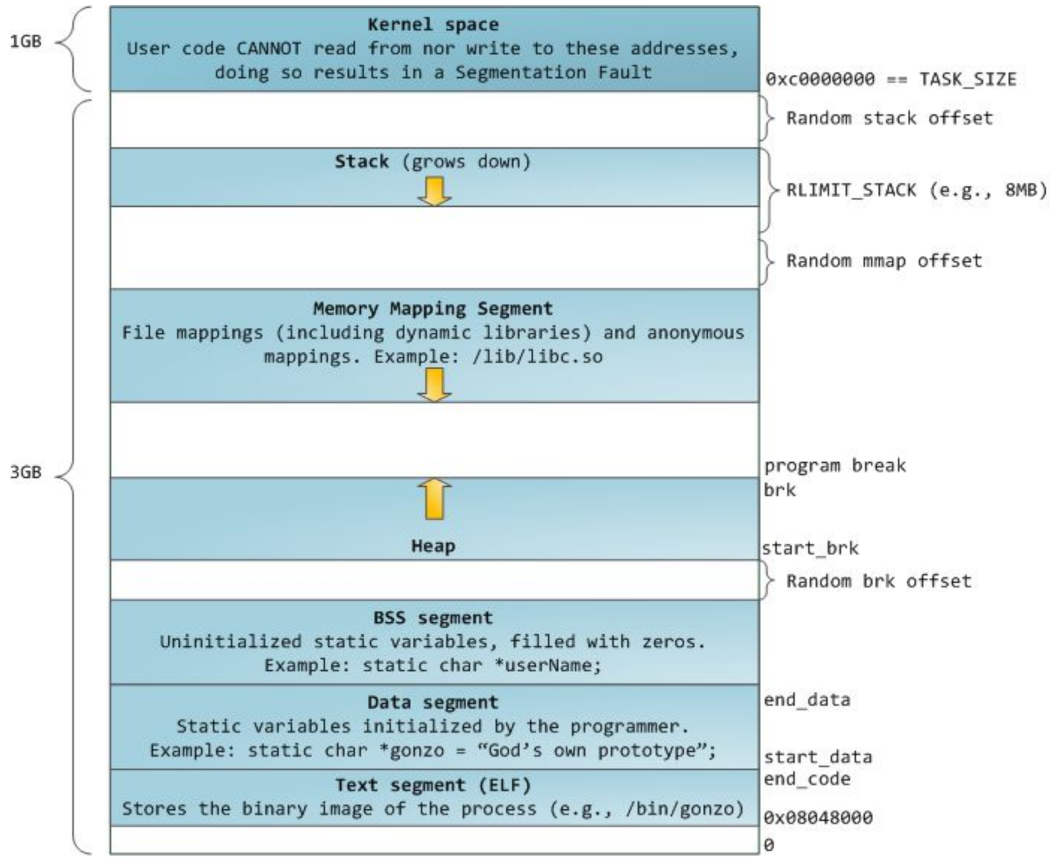
typical 32-bit Linux OS



1 GB
3 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

standard segment layout in a Linux process:



1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

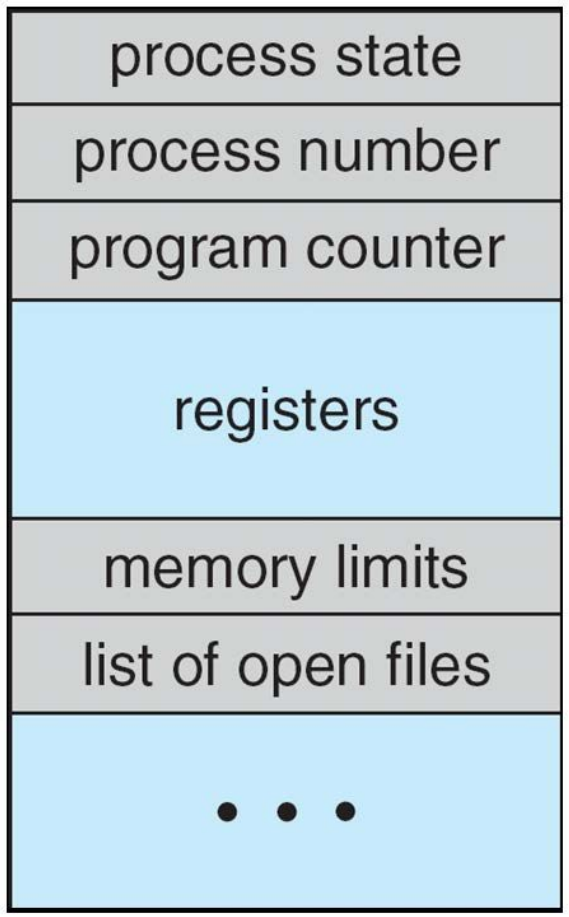
Process Creation

1) Assign a unique process identifier to the new process.
2) Allocate space for the process.
3) Initialize the process control block.
4) Set the appropriate linkages.
5) Create or expand other data structures.

Process Control Block (PCB)

130
CHAPTER 3 / PROCESS DESCRIPTION AND CONTROL

- categories:
  - Process identification
  - Processor state information
  - Process control information
- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information



PCB obsahuje informácie o procese, z ktorých niektoré sú: – ukazovateľ na zásobník procesu, – stav procesu –nový, pripravený, bežiaci, čakajúci atď., – hodnota čítača inštrukcií –indikuje adresu inštrukcie, ktorá bude vykonaná ako nasledujúca, – registre CPU–počet a typ registrov sa mení podľa architektúry počítača. • počítadlo inštrukcií, akumulátory, index registre, ukazovatele zásobníkov, univerzálne registre, inštrukcie o podmienených kódoch a iné. – informácie pre plánovanie procesu–priorita procesu, ukazovatele na fronty pre plánovanie a iné, – informácie pre správu pamäte –hodnoty limitných a bázových registrov, tabuľky stránok alebo segmentov, podľa použitej techniky správy pamäte, – účtovacie informácie – spotrebovaný čas CPU, časové limity pre proces atď., – V/V informácie –obsahujú zoznam V/V zariadení, ktoré sú pridelené procesu, zoznam otvorených súborov atď

Table 3.5 Typical Elements of a Process Control Block

Process Identification

Identifiers
Numeric identifiers that may be stored with the process control block include

- Identifier of this process
- Identifier of the process that created this process (parent process)
- User identifier

Processor State Information

User-Visible Registers
A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers
These are a variety of processor registers that are employed to control the operation of the processor. These include

- **Program counter:** Contains the address of the next instruction to be fetched
- **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- **Status information:** Includes interrupt enabled/disabled flags, execution mode

Stack Pointers
Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

Process Control Information

Scheduling and State Information
This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- **Process state:** Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- **Priority:** One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable).
- **Scheduling-related information:** This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- **Event:** Identity of event the process is awaiting before it can be resumed.

Data Structuring
A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent–child (creator–created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication
Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

Process Privileges
Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

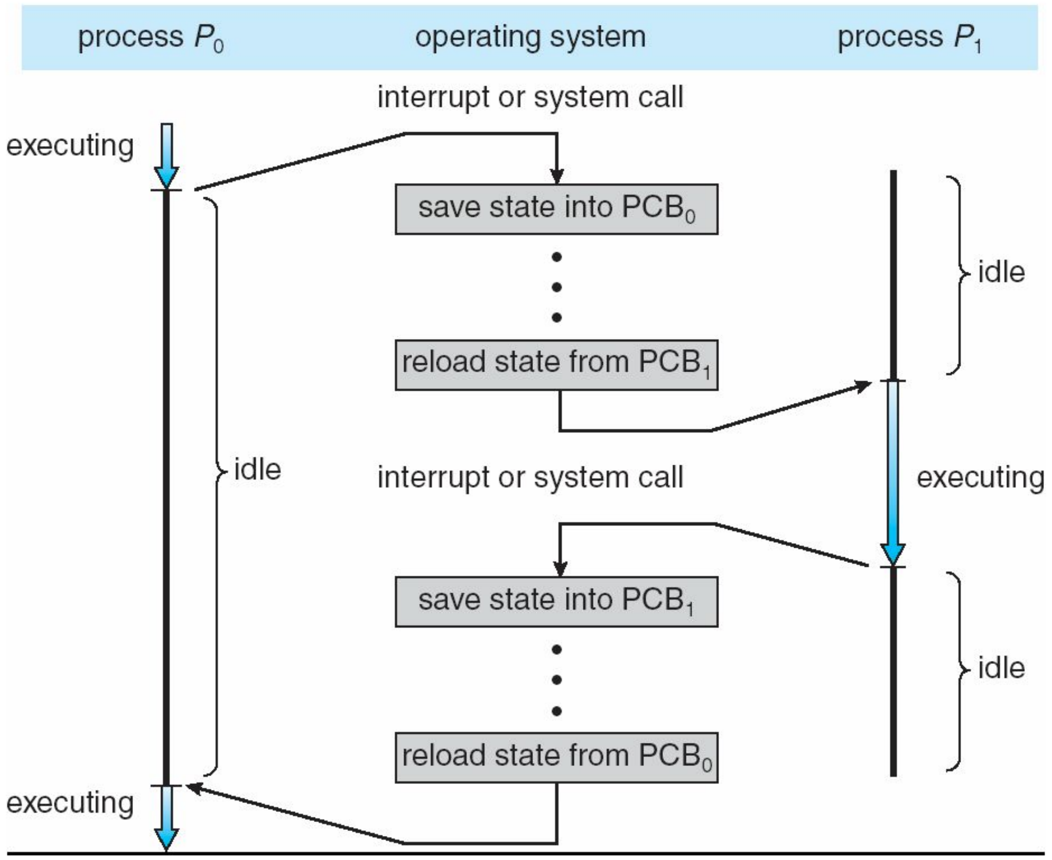
Memory Management
This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization
Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

Process Switching

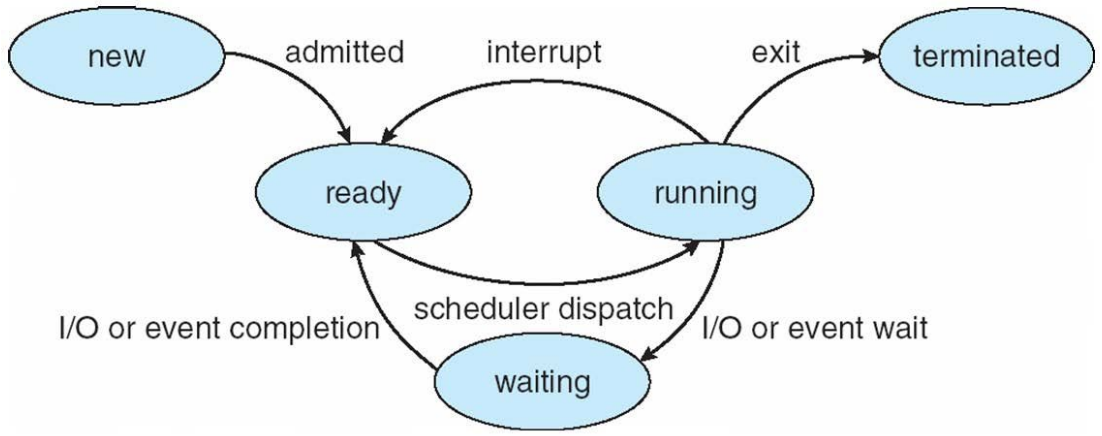
- **Clock interrupt:** The OS determines whether the currently running process has been executing for he maximum allowable unit of time, referred to as a time slice
  - That is, a time slice is the maximum amount of time that a process can execute before being interrupted. If so, this process must be switched to a Ready state and another process dispatched.
- **I/O interrupt:** The OS determines what I/O action has occurred.
  - If the I/O action constitutes an event for which one or more processes are waiting, then the OS moves all of the corresponding blocked processes to the Ready state (and Blocked/Suspend processes to the Ready/Suspend state). The OS must then decide whether to resume execution of the process currently in the Running state or to preempt that process for a higher-priority Ready process.
- **Memory fault:** The processor encounters a virtual memory address reference for a word that is not in main memory.
  - The OS must bring in the block (page or segment) of memory containing the reference from secondary memory to main memory. After the I/O request is issued to bring in the block of memory, the process with the memory fault is placed in a blocked state; the OS then performs a process switch to resume execution of another process. After the desired block is brought into memory, that process is placed in the Ready state.

CPU Switch From Process to Process

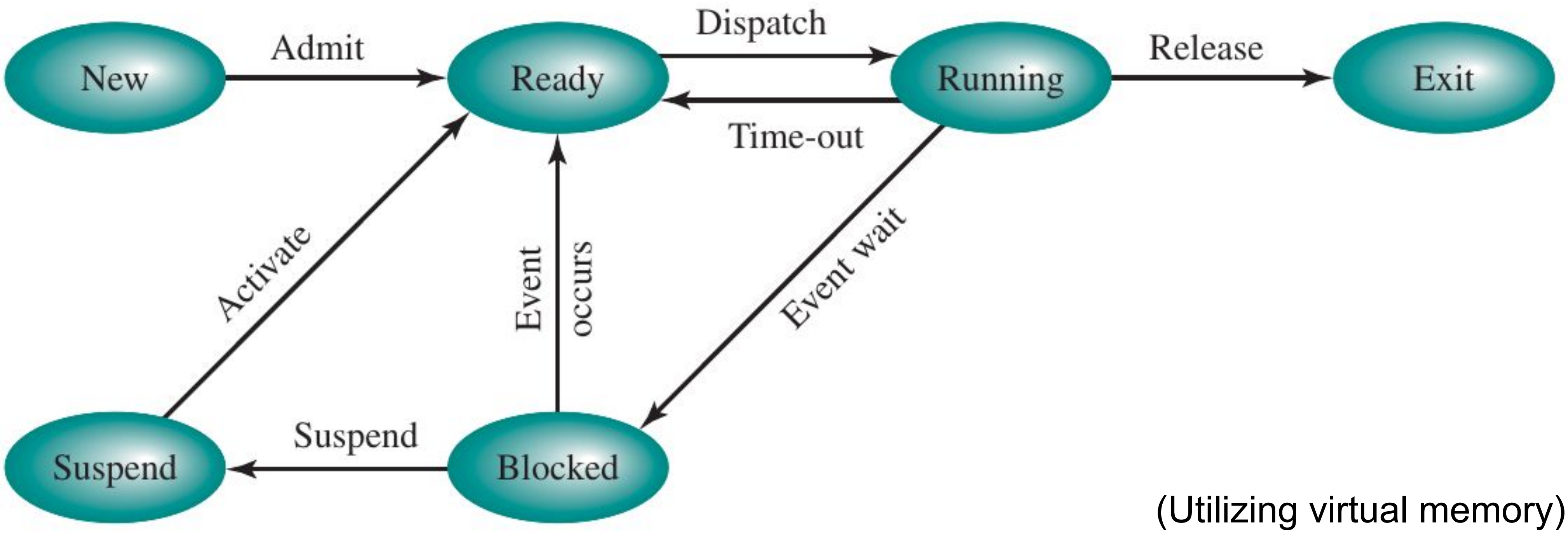


Process State

- As a process executes, it changes state
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution



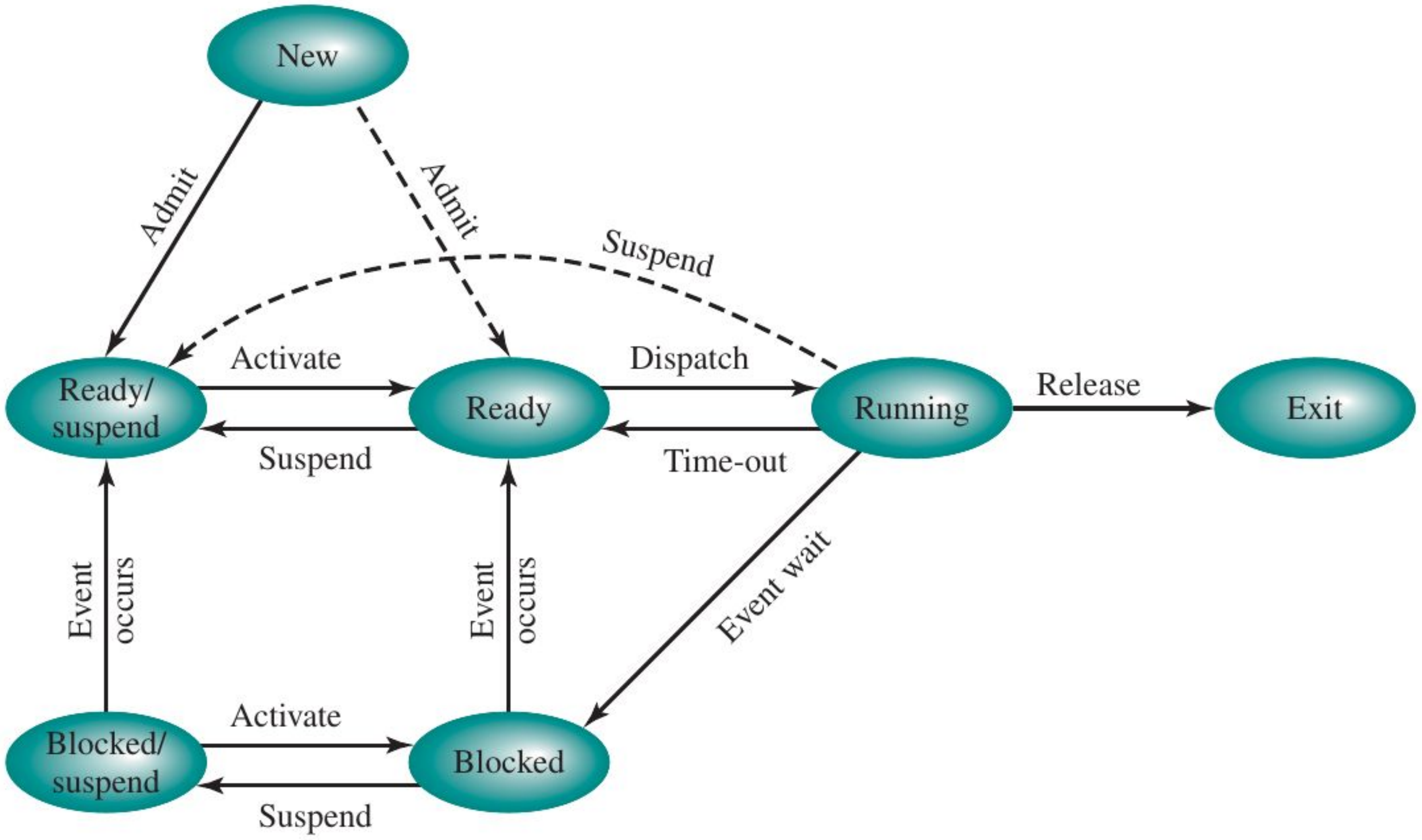
Suspended Processes - THE NEED FOR SWAPPING - Page 121



(Utilizing virtual memory)

- Swapping - involves moving part or all of a process from main memory to disk.
- When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue. This is a queue of existing processes that have been temporarily kicked
- out of main memory, or suspended.
- The OS then brings in another process from the suspend queue, or it honors a new-process request.
- Execution then continues with the newly arrived process.
- Swapping, however, is an I/O operation, and therefore there is the potential for making the problem worse, not better.

Suspended Processes - THE NEED FOR SWAPPING - Page 121



It clearly would not do any good to bring a blocked process back into main memory, because it is still not ready for execution.

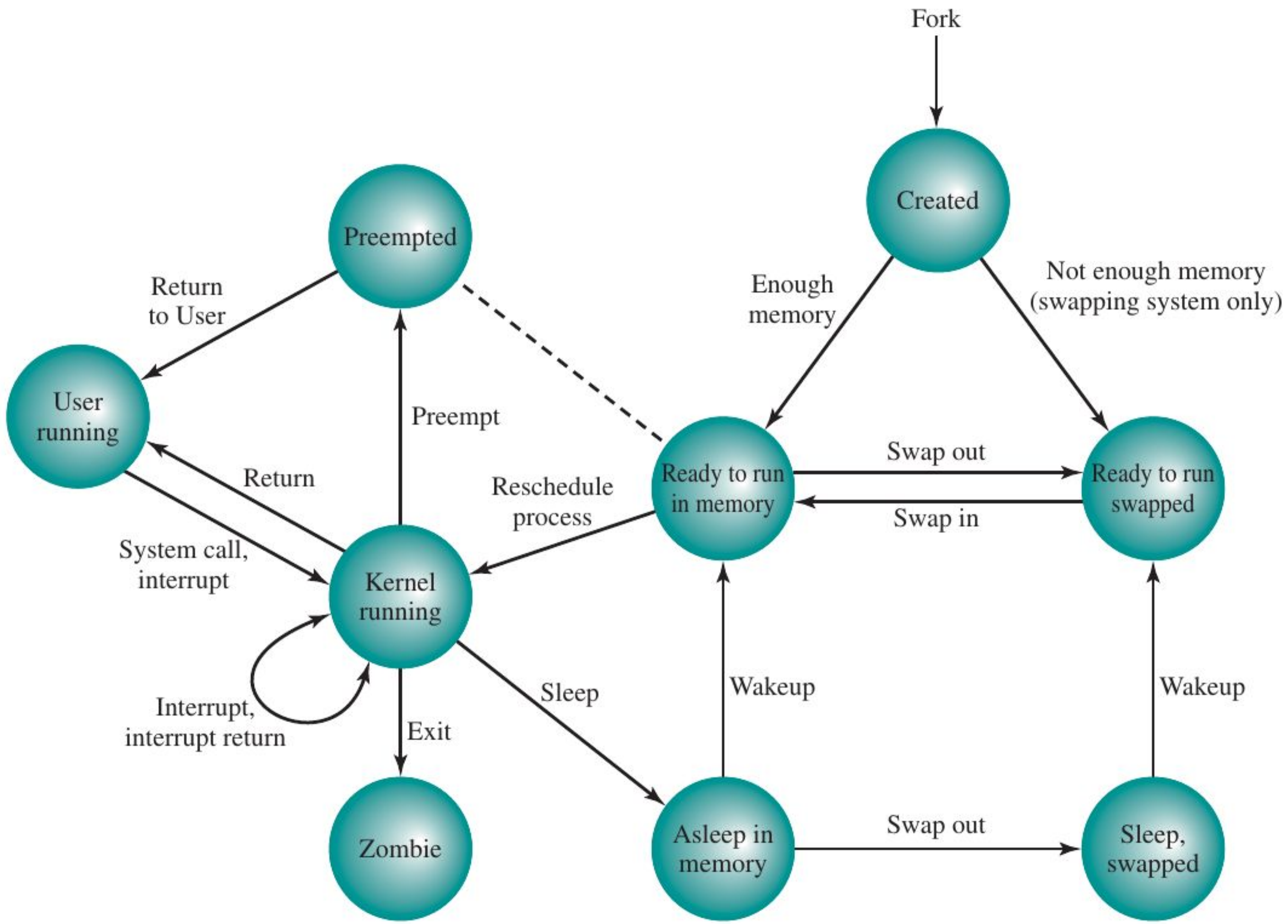


Figure 3.17 **UNIX Process State Transition Diagram**

Context switch

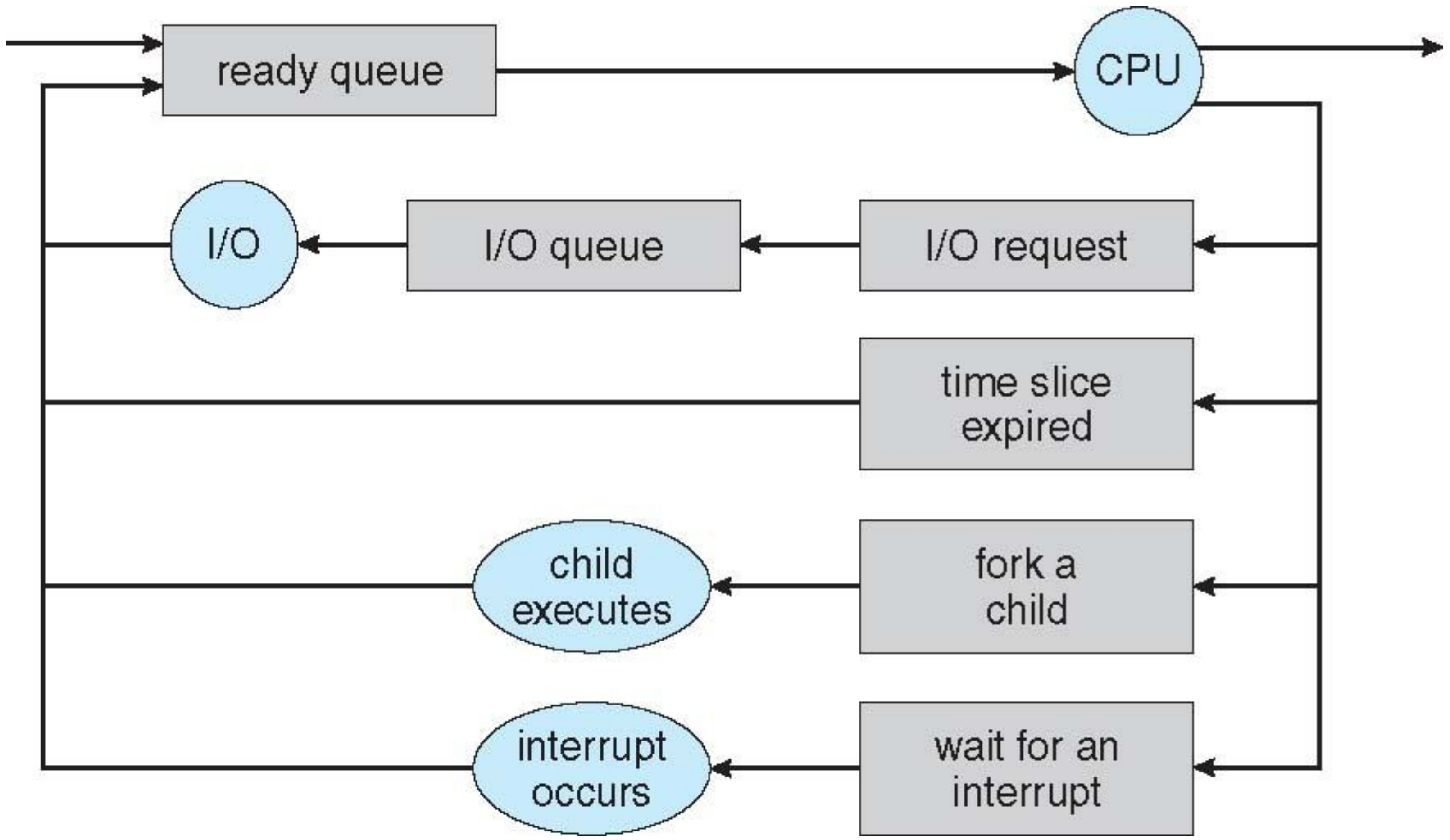
3.4 / PROCESS CONTROL, page 139

1) Save the context of the processor, including program counter and other registers.
2) Update the process control block of the process that is currently in the Running state. This includes changing the state of the process to one of the other states (Ready; Blocked; Ready/Suspend; or Exit). Other relevant fields must also be updated, including the reason for leaving the Running state and accounting information.
3) Move the process control block of this process to the appropriate queue (Ready; Blocked on Event i; Ready/Suspend).
4) Select another process for execution; this topic is explored in Part Four.
5) Update the process control block of the process selected. This includes changing the state of this process to Running.
6) Update memory management data structures. This may be required, depending on how address translation is managed; this topic is explored in Part Three.
7) Restore the context of the processor to that which existed at the time the selected process was last switched out of the Running state, by loading in the previous values of the program counter and other registers.

Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Representation of Process Scheduling



Levels of scheduling

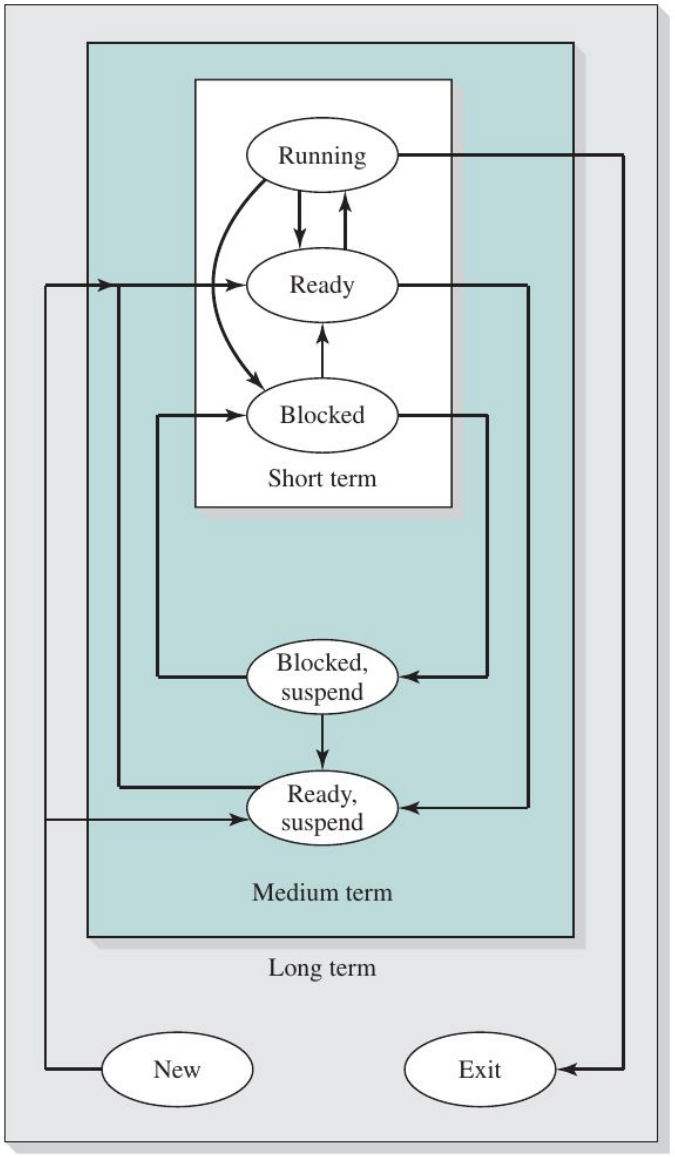


Figure 9.2 **Levels of Scheduling**

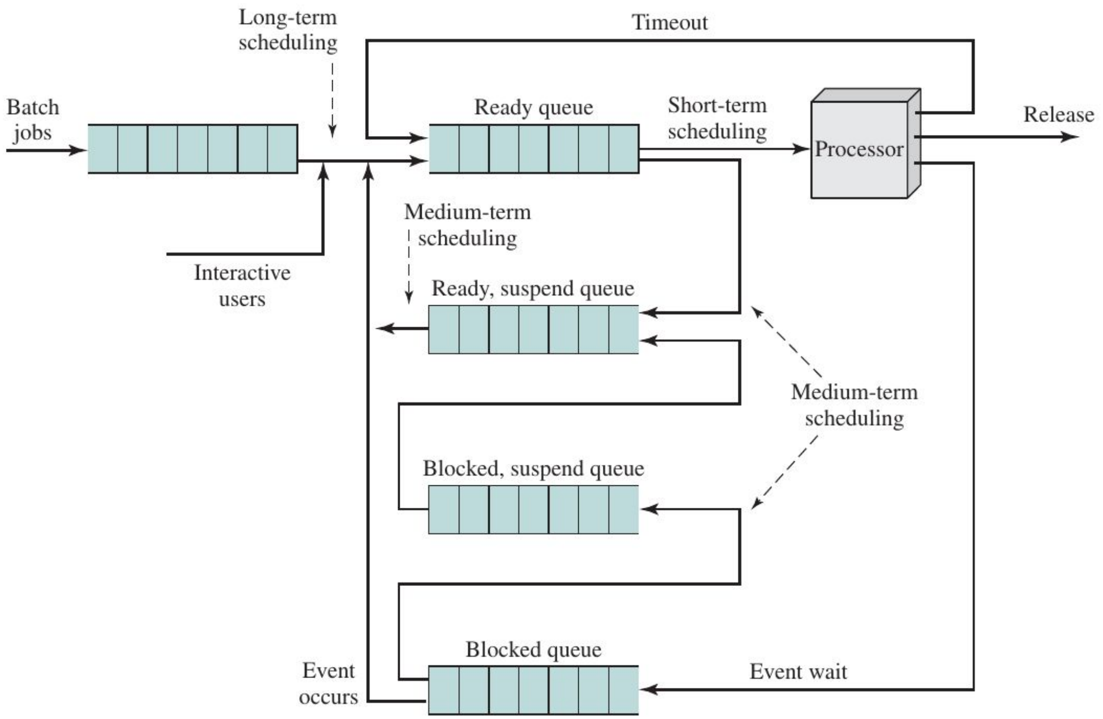


Figure 9.3 **Queueing Diagram for Scheduling**

Levels of scheduling

- **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.
  - Executes relatively infrequently
- **Medium-Term Scheduling**
  - 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.
  - More frequently than the long-term
- **Short-Term Scheduling**
  - Also known as the dispatcher
  - Executes most frequently and makes the fine-grained decision of which process to execute next.
  - 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 favor of another. (Clock interrupts, I/O interrupts, Operating system calls, Signals)

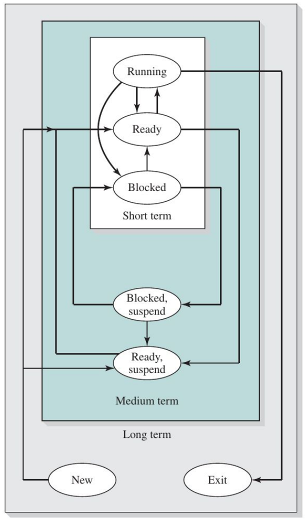


Figure 9.2 Levels of Scheduling

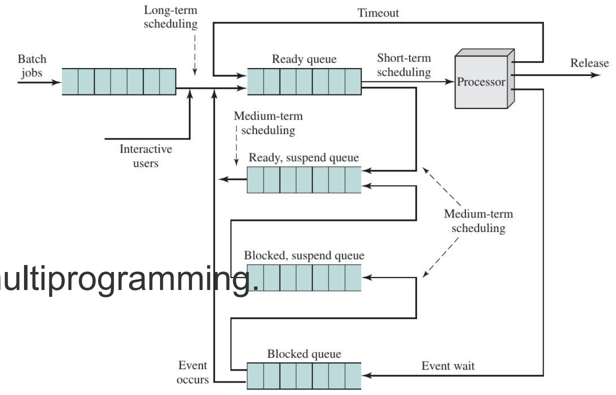


Figure 9.3 Queueing Diagram for Scheduling

Levels of scheduling

(in other words)

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - controls the degree of multiprogramming
  - invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - invoked very frequently (milliseconds) ⇒ (must be fast)
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

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

Effect of Size of Preemption Time Quantum

Applies for the preemptive scheduling policy decision mode

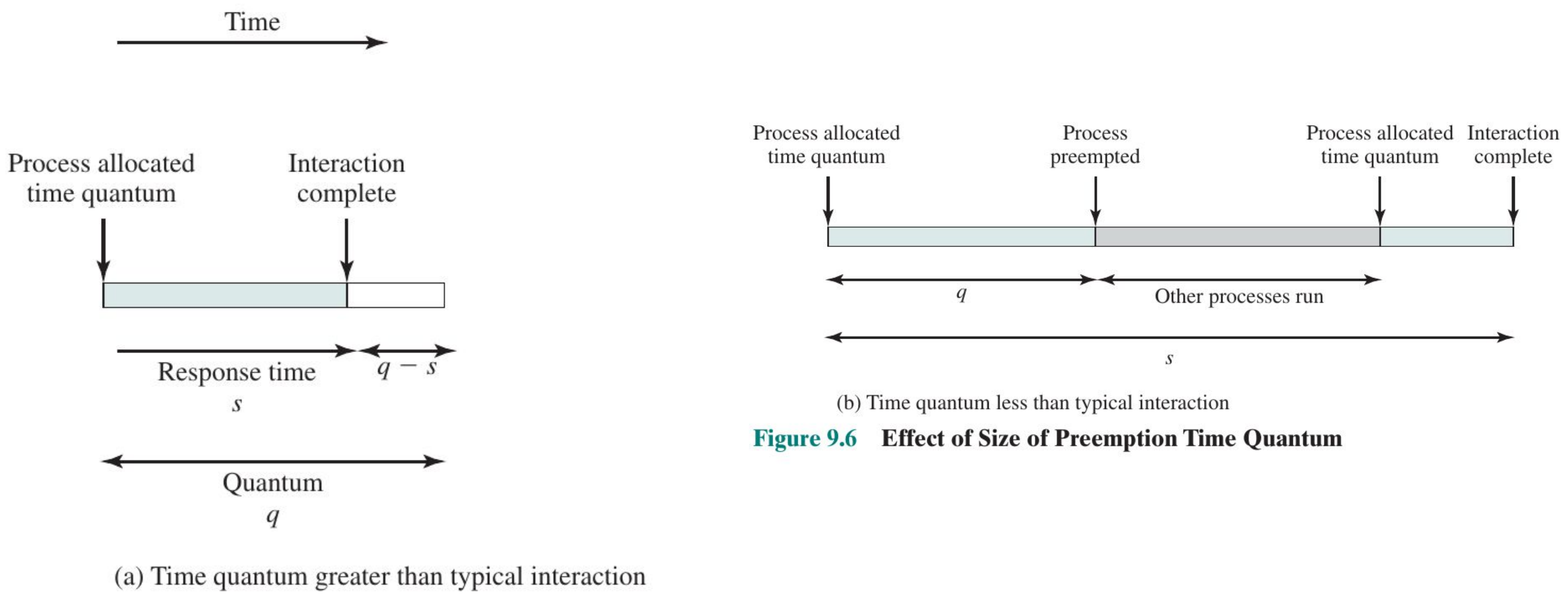

Time
Process allocated time quantum
Interaction complete
Response time s
$q - s$
Quantum q
(a) Time quantum greater than typical interaction
Process allocated time quantum
Process preempted
Process allocated time quantum
Interaction complete
q
Other processes run
s
(b) Time quantum less than typical interaction

Figure 9.6 **Effect of Size of Preemption Time Quantum**

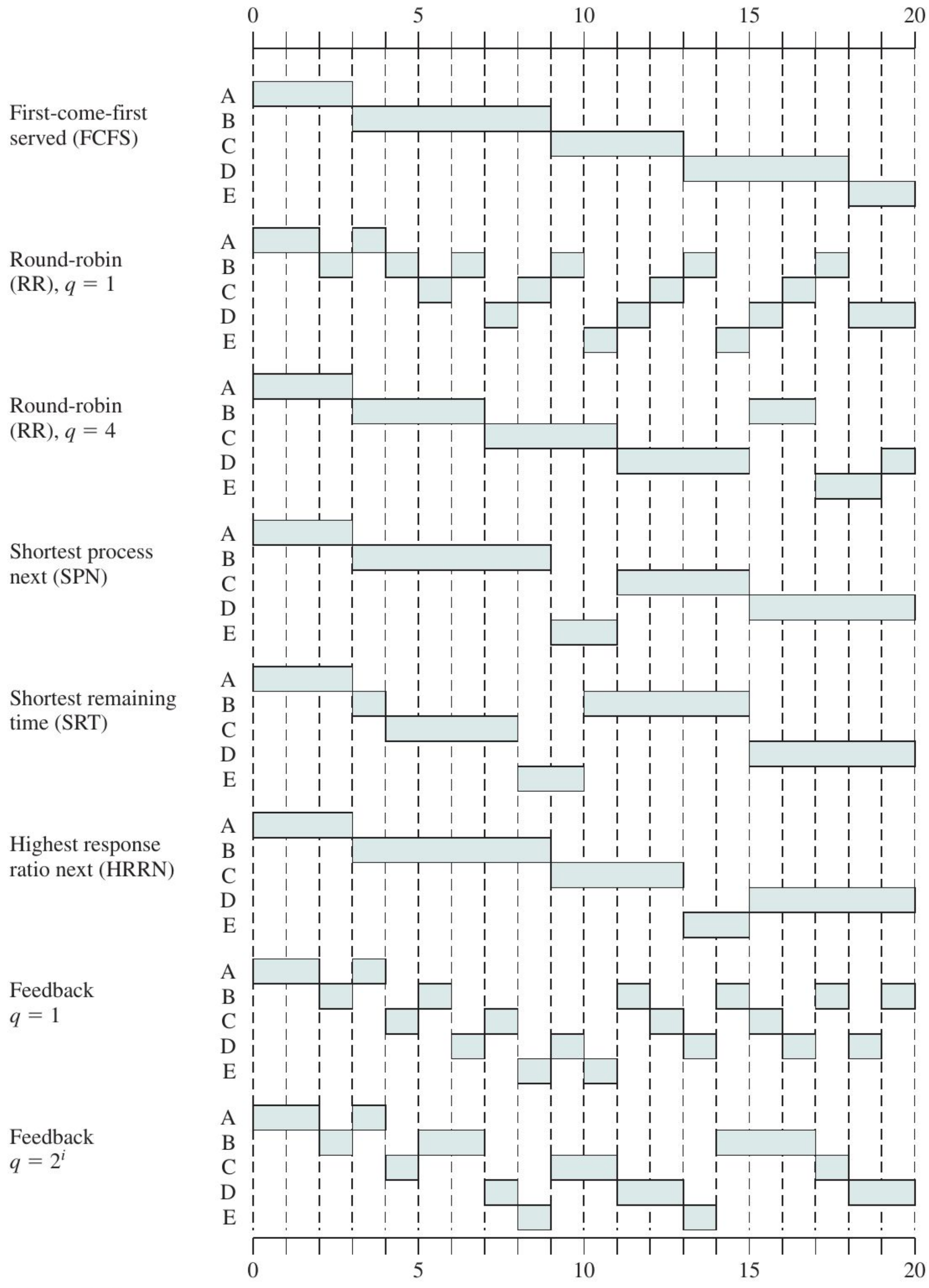


Figure 9.5 A Comparison of Scheduling Policies

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

Table 9.5 A Comparison of Scheduling Policies

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean
FCFS						
Finish Time	3	9	13	18	20	
Turnaround Time (T_r)	3	7	9	12	12	8.60
T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$						
Finish Time	4	18	17	20	15	
Turnaround Time (T_r)	4	16	13	14	7	10.80
T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$						
Finish Time	3	17	11	20	19	
Turnaround Time (T_r)	3	15	7	14	11	10.00
T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN						
Finish Time	3	9	15	20	11	
Turnaround Time (T_r)	3	7	11	14	3	7.60
T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT						
Finish Time	3	15	8	20	10	
Turnaround Time (T_r)	3	13	4	14	2	7.20
T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN						
Finish Time	3	9	13	20	15	
Turnaround Time (T_r)	3	7	9	14	7	8.00
T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$						
Finish Time	4	20	16	19	11	
Turnaround Time (T_r)	4	18	12	13	3	10.00
T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$						
Finish Time	4	17	18	20	14	
Turnaround Time (T_r)	4	15	14	14	6	10.60
T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

The Use of Priorities

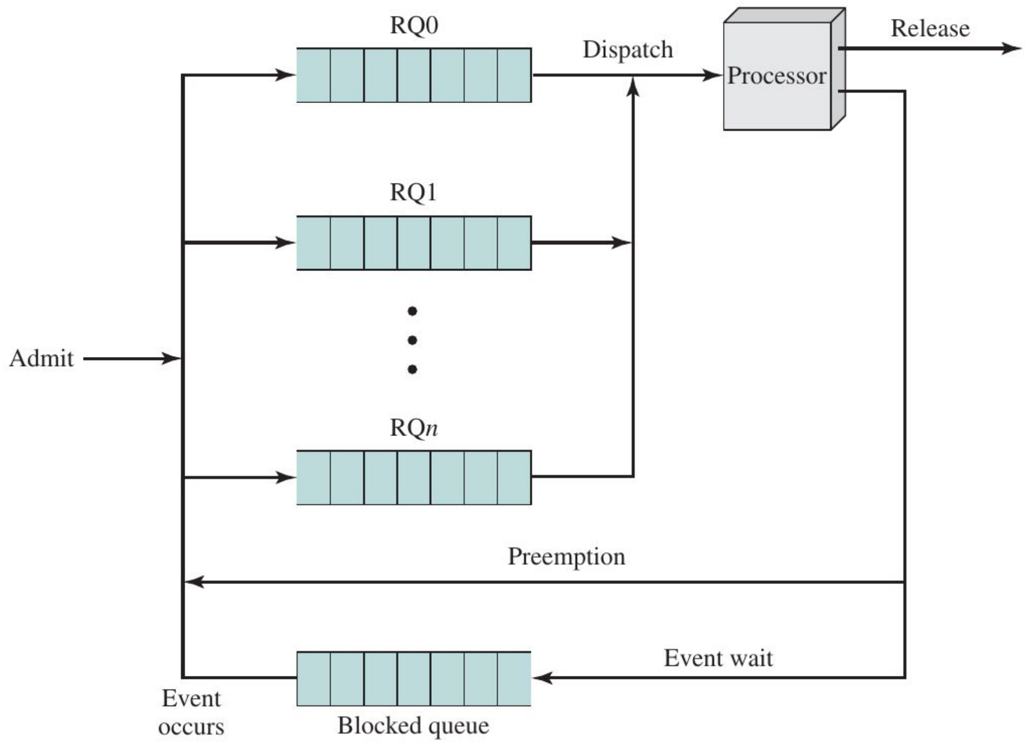


Figure 9.4 **Priority Queueing**

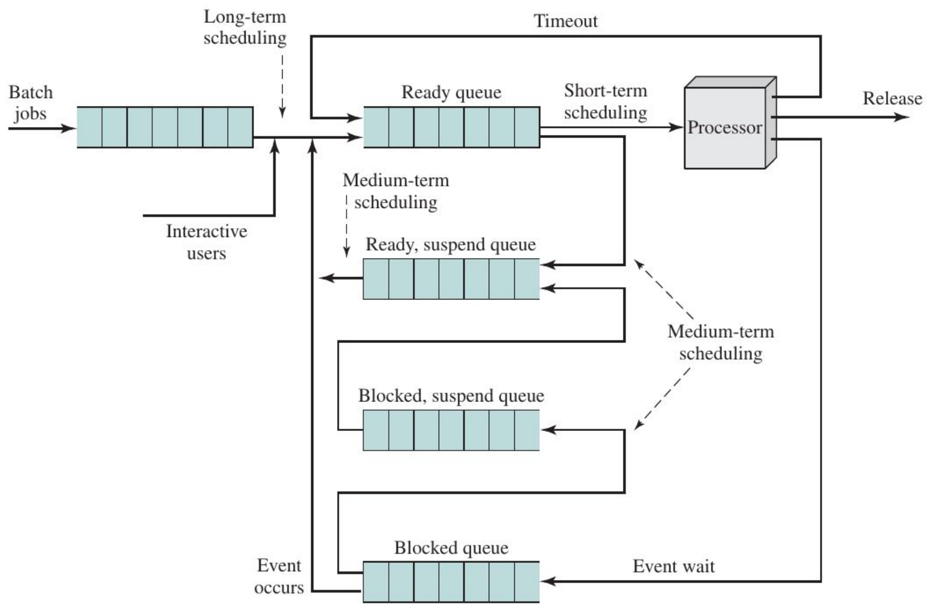


Figure 9.3 **Queueing Diagram for Scheduling**

Multithreading

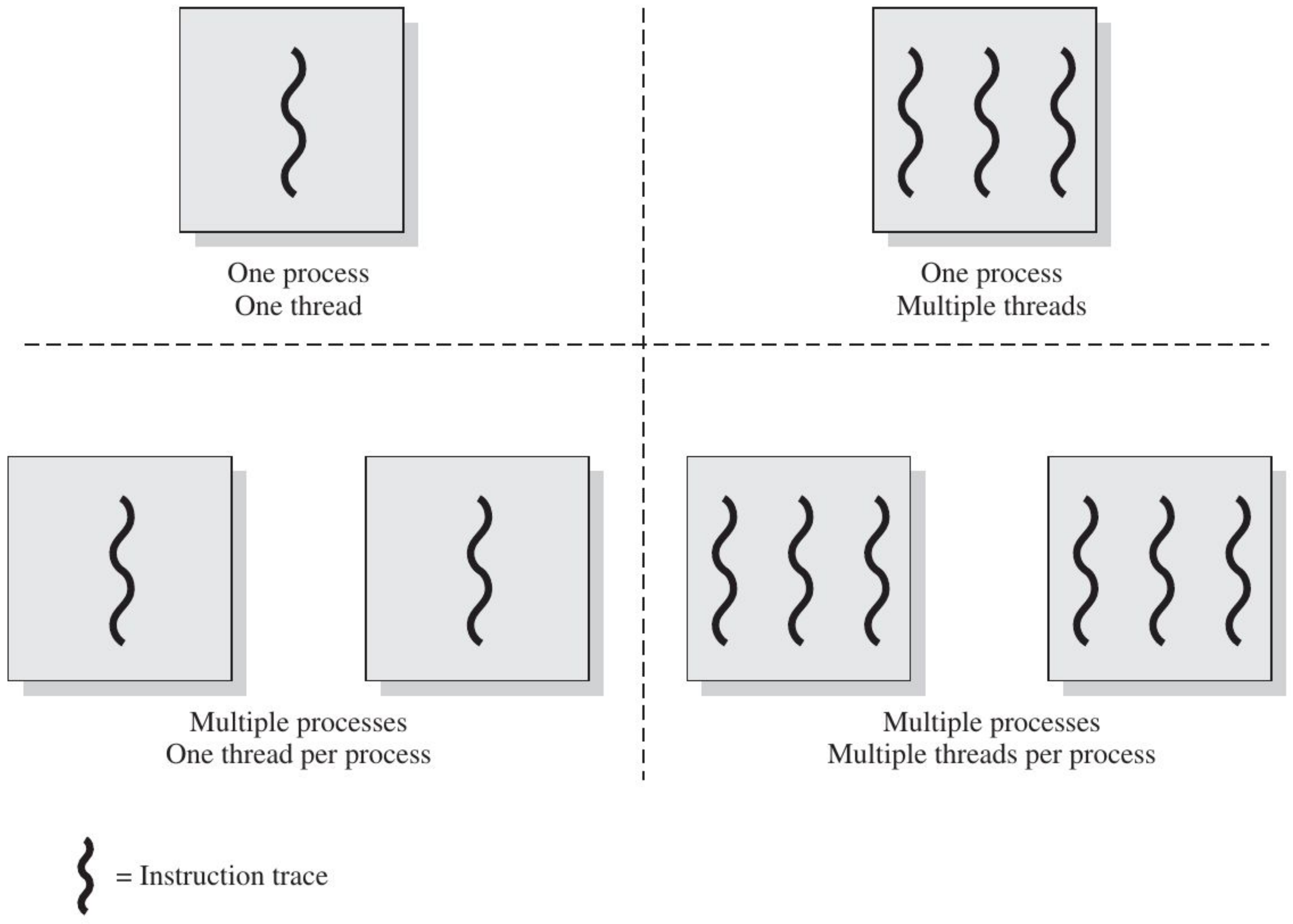


Figure 4.1 Threads and Processes [ANDE97]

Multithreading

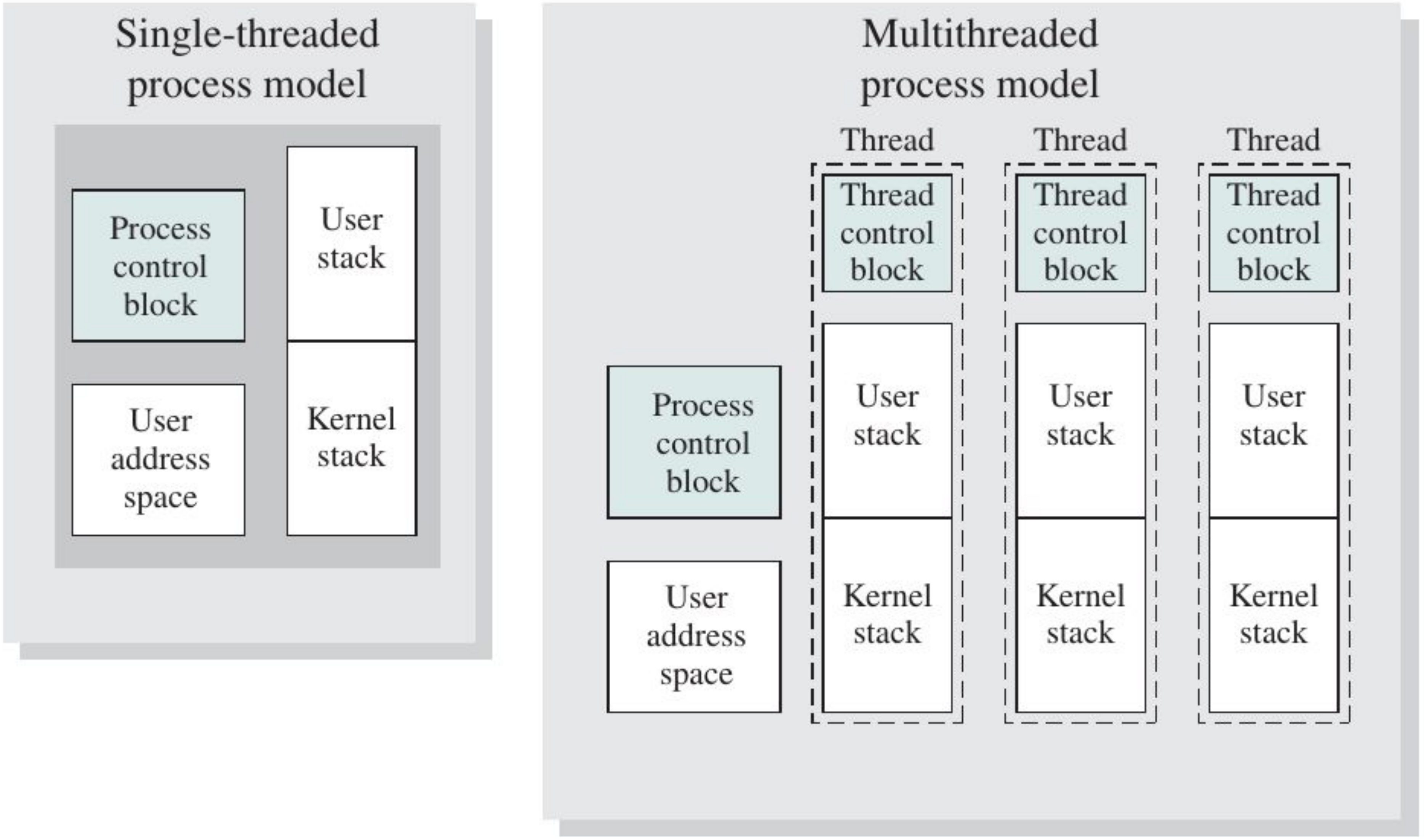


Figure 4.2 Single-Threaded and Multithreaded Process Models

The key benefits of threads

4.1 / PROCESSES AND THREADS Page 161

- 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.

Uses of threads in a single-user multi-processing system

- Foreground and background work
  - This arrangement often increases the perceived speed of the application by allowing the program to prompt for the next command before the previous command is complete
- Asynchronous processing
  - Example: periodic file saving
- Speed of execution
  - Even though one thread may be blocked for an I/O operation to read in a batch of data, another thread may be executing
- Modular program structure
  - Ease of program design

Categorization of threads by level

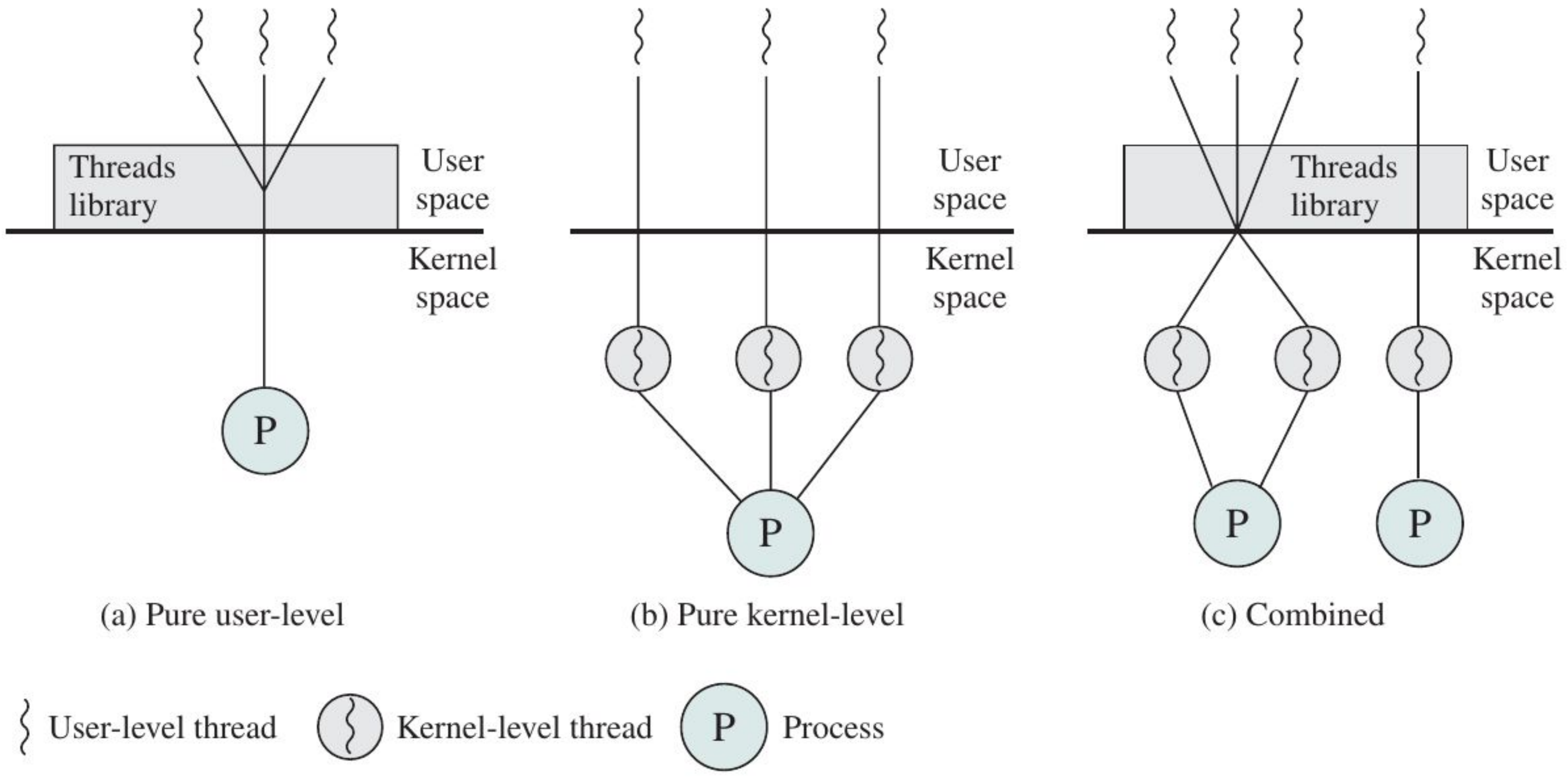


Figure 4.5 **User-Level and Kernel-Level Threads**

Linux Threads

- Traditional UNIX systems support a single thread of execution per process, while modern UNIX systems typically provide support for multiple kernel-level threads per process.
- As with traditional UNIX systems, older versions of the Linux kernel offered no support for multithreading. Instead, applications would need to be written with a set of user-level library functions, the most popular of which is known as pthread (POSIX thread) libraries, with all of the threads mapping into a single kernel-level process.
- We have seen that modern versions of UNIX offer kernel-level threads.
- Linux provides a unique solution in that **it does not recognize a distinction between threads and processes**. Using a mechanism similar to the lightweight processes of Solaris, user-level threads are mapped into kernel-level processes.
- **Multiple user-level threads that constitute a single user-level process are mapped into Linux kernel-level processes that share the same group ID.**
- This enables these processes to share resources such as files and memory and to avoid the need for a context switch when the **scheduler switches among processes in the same group**.
- **A new process is created in Linux by copying the attributes of the current process.** A new process can be cloned so that it shares resources, such as files, signal handlers, and virtual memory. When the two processes share the same virtual memory, they function as threads within a single process. However, no separate type of data structure is defined for a thread. In place of the usual fork() command, processes are created in Linux using the clone() command. This command includes a set of flags as arguments.
- The traditional fork() system call is implemented by Linux as a clone() system call with all of the clone flags cleared.

Example: Process creation in UNIX

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  int pid;
  pid = fork(); /* process creation */
  if (pid < 0) { /* error handling */
     perror("Fork failed");
     exit(-1);
  }
  else if (pid == 0) { /* child process */
     execlp("/bin/ls","ls",NULL);
  }
  else { /* parent process */
     wait(NULL); /* waiting for child to finish */
     printf(„Child finished");
     exit(0);
  }
}
```

Next topics

- Interprocess Communication (IPC)
- Synchronization
- …
- Memory management

Concurrency

The central themes of operating system design are all concerned with the management of processes and threads:

- **Multiprogramming:** The management of multiple processes within a uniprocessor system
- **Multiprocessing**: The management of multiple processes within a multiprocessor
- **Distributed processing:** The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system.

Concurrency arises in three different contexts:

- **Multiple applications:** Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.
- **Structured applications:** As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.
- **Operating system structure:** The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.

Key Terms Related to Concurrency

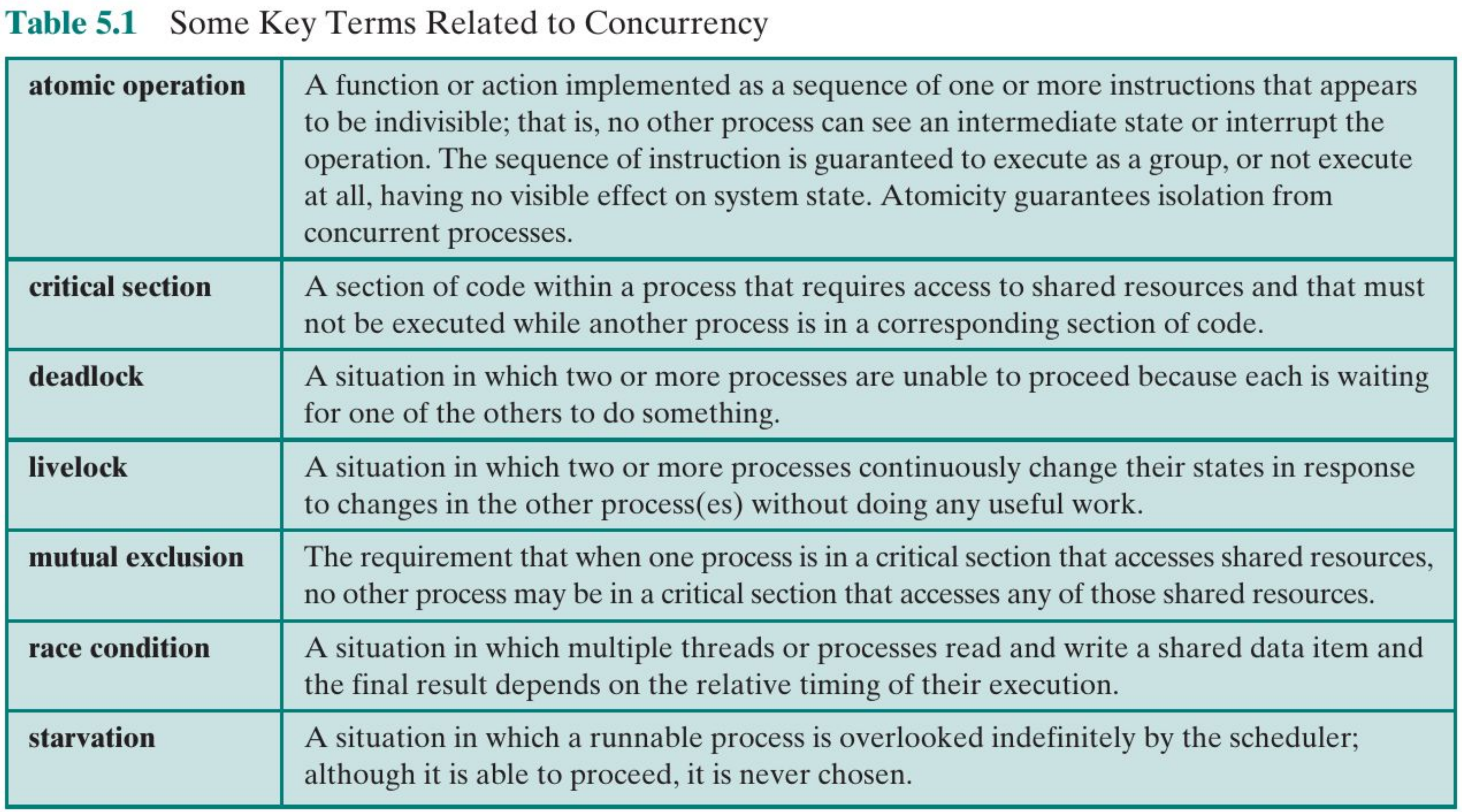


Table 5.1 Some Key Terms Related to Concurrency

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.