Operating System

Chapter 3. Process

Lynn Choi

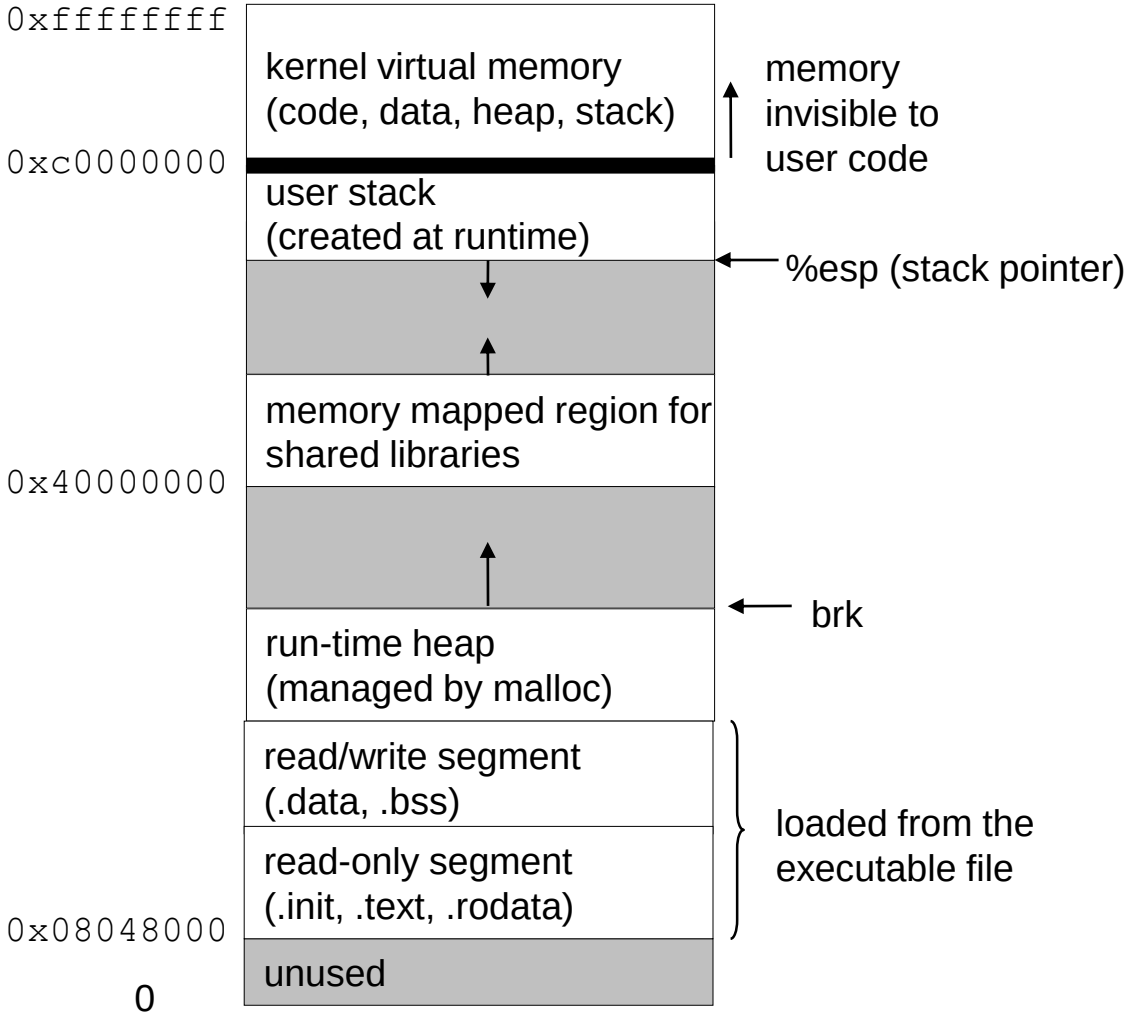
School of Electrical Engineering

Process

- **Def: A process is *an instance of a program in execution*.**
  - One of the most profound ideas in computer science.
  - Not the same as “program” or “processor”
- **Process provides two key abstractions:**
  - Logical control flow
    - Each process has an exclusive use of the processor.
  - Private address space
    - Each process has an exclusive use of private memory.
- **How are these Illusions maintained?**
  - Multiprogramming(multitasking): process executions are interleaved
    - In reality, many other programs are running on the system.
    - Processes take turns in using the processor
      - Each time period that a process executes a portion of its flow is called a *time slice*
  - Virtual memory: OS provides a private space for each process
    - The private space is called the *virtual address space*, which is a linear array of bytes, addressed by n bit virtual address (0, 1, 2, 3, … 2^n-1)

Private Address Spaces

- Each process has its own private address space.



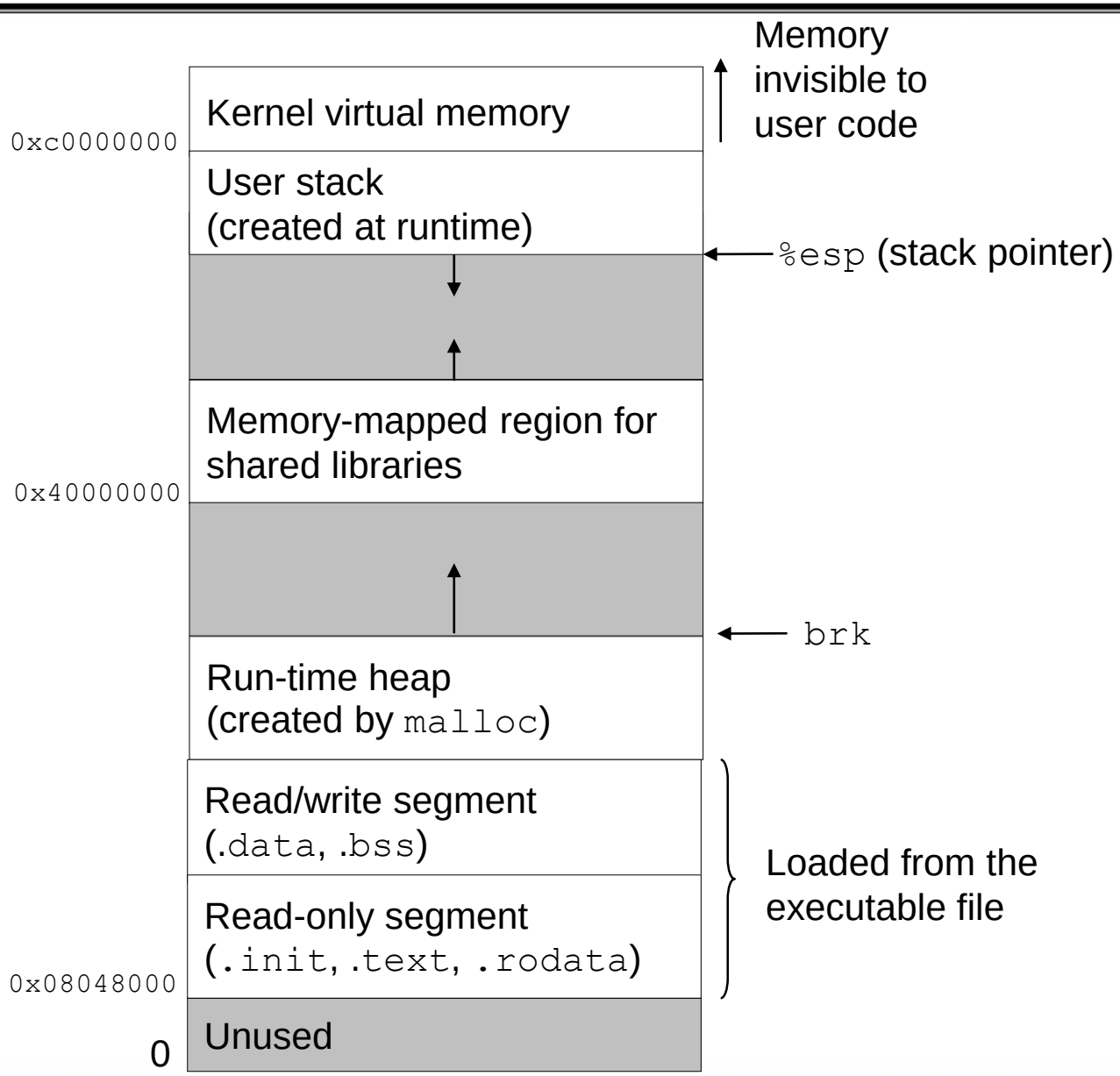
Life and Scope of an Object

- ***Life vs. scope***
  - *Life* of an object determines whether the object is *still in memory* (of the process) whereas the *scope* of an object determines whether the object *can be accessed at this position*
  - It is possible that an object is live but not visible.
  - It is *not* possible that an object is visible but not live.
- ***Local variables***
  - Variables defined inside a function
  - The scope of these variables is only within this function
  - The life of these variables ends when this function completes
  - So when we call the function again, storage for variables is created and values are reinitialized.
  - *Static local* variables - If we want the value to be extent throughout the life of a program, we can define the local variable as "static."
    - Initialization is performed only at the first call and data is retained between func calls.

Life and Scope of an Object

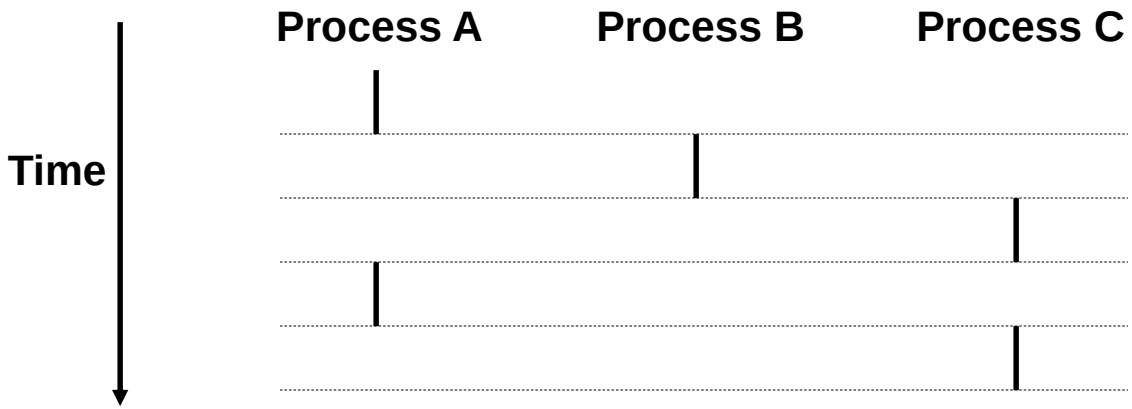
- ***Global variables***
  - Variables defined outside a function
  - The scope of these variables is throughout the entire program
  - The life of these variables ends when the program completes
- ***Static variables***
  - Static variables are local in scope to their module in which they are defined, but life is throughout the program.
  - *Static local variables*: static variables inside a function cannot be called from outside the function (because it's not in scope) but is alive and exists in memory.
  - *Static variables*: if a static variable is defined in a global space (say at beginning of file) then this variable will be accessible only in this file (file scope)
    - If you have a global variable and you are distributing your files as a library and you want others not to access your global variable, you may make it static by just prefixing keyword static

Linux Run-time Memory Image



Logical Control Flows

Each process has its own logical control flow



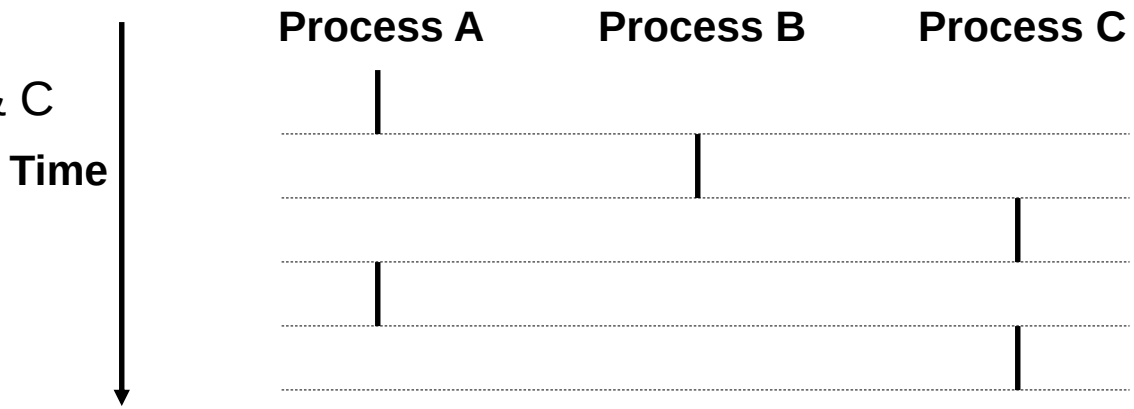
Concurrent Processes

Concurrent processes

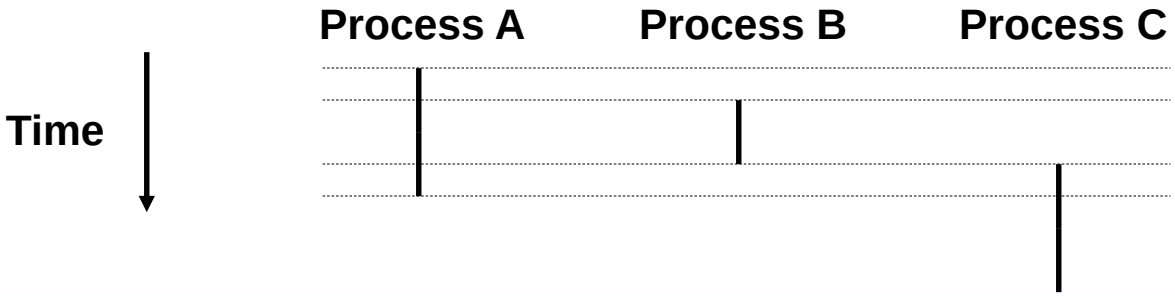
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time.
- Otherwise, they are *sequential.*

Examples:

- Concurrent: A & B, A & C
- Sequential: B & C



- Control flows for concurrent processes are ***physically disjoint*** in time.
- However, we can think of concurrent processes as ***logically running in parallel*** with each other.



Context Switching

- **Processes are managed by OS code called the *kernel***
  - Important: *the kernel is not a separate process*, but rather runs as part of some user process
    - Processors typically provide this capability with a mode bit in some control register
- ***User mode and kernel mode***
  - If the mode bit is set, the process is running in *kernel mode* (*supervisor mode*), and can execute any instruction and can access any memory location
  - If the mode bit is not set, the process is running in *user mode* and is not allowed to execute *privileged instructions*
    - A process running application code is initially in user mode
    - The only way to change from user mode to kernel mode is via an exception and exception handler runs in kernel mode

Context Switching

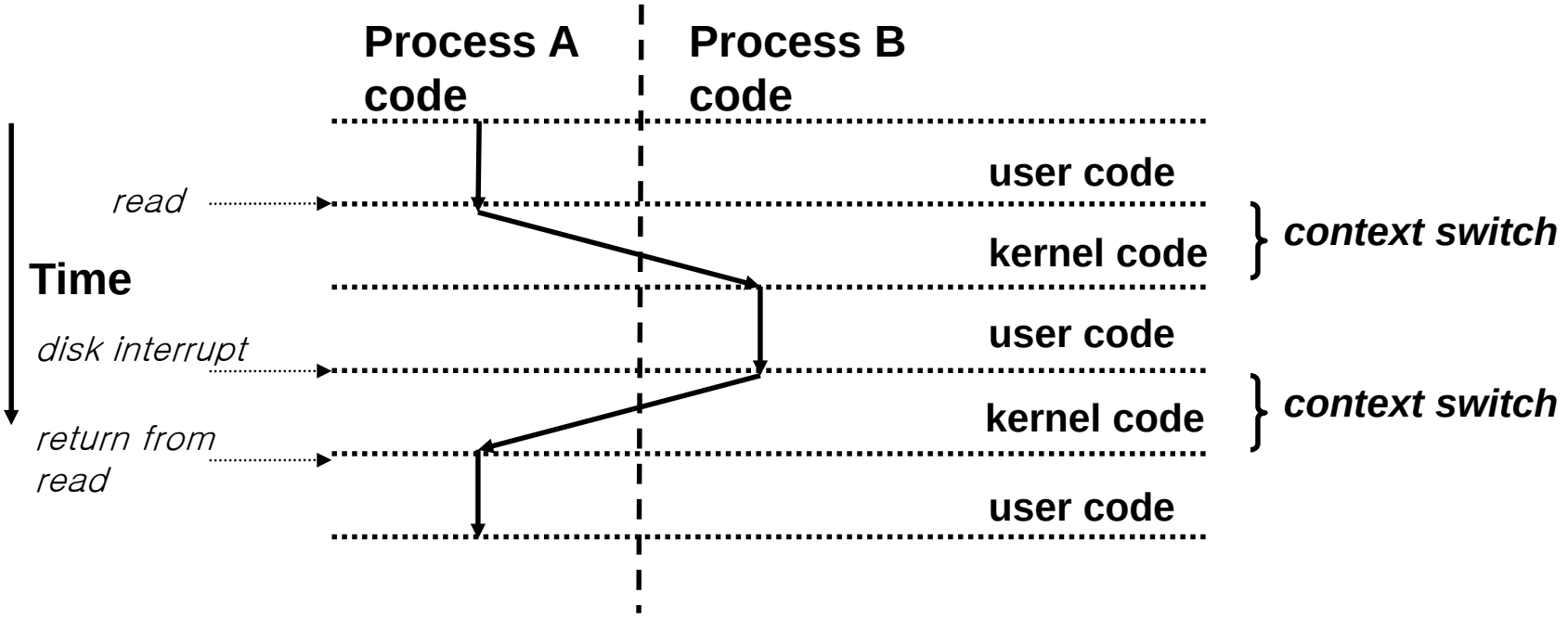
Context

- The kernel maintains a *context* for each process
  - The context is the state of a process that the kernel needs to restart a preempted process
  - Consist of PC, general purpose registers, FP registers, status registers, and various kernel data structures such as page table and file table

Context switching

- The OS kernel implements multitasking using an exceptional control flow
- At certain points during the execution of a process, the kernel decide to preempt the current process and restart a previously preempted process
  - This is called *scheduling* and handled by code in the kernel called *scheduler (or dispatcher)*
- Context switching
  - The kernel first saves the context of the current process
  - The kernel restores the context of some previously preempted process
  - Then, the kernel passes control to this newly restored process

Context Switching



Process Control Block

Process Control Block

- A data structure in the OS kernel that contains the information needed to manage a particular process
- Process ID, state, priority, pointer to register save area, and status tables such as page tables, file tables, IO tables, etc.
- Created and managed by the operating system

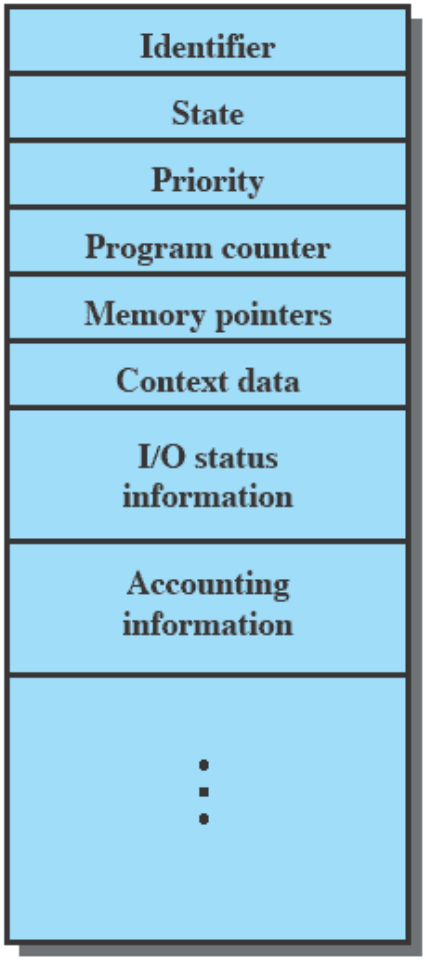


Figure 3.1 Simplified Process Control Block

Source: Pearson

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.

Source: Pearson

Process Control Block

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.

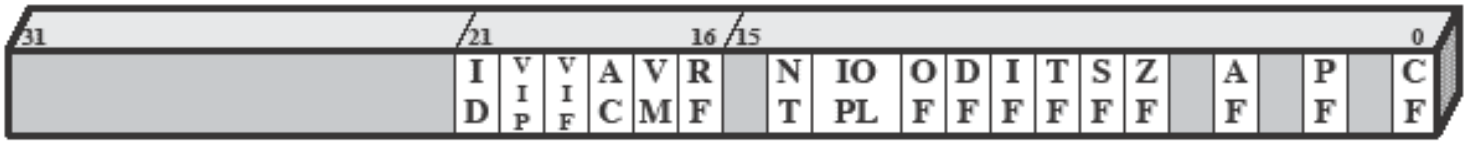
Resource Ownership and Utilization

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.

Source: Pearson

Program Status Word (PSW)

- ***PSW***
  - Contains condition codes and other status information of the currently running process
  - Example: EEFLAGS register on Intel x86 processors



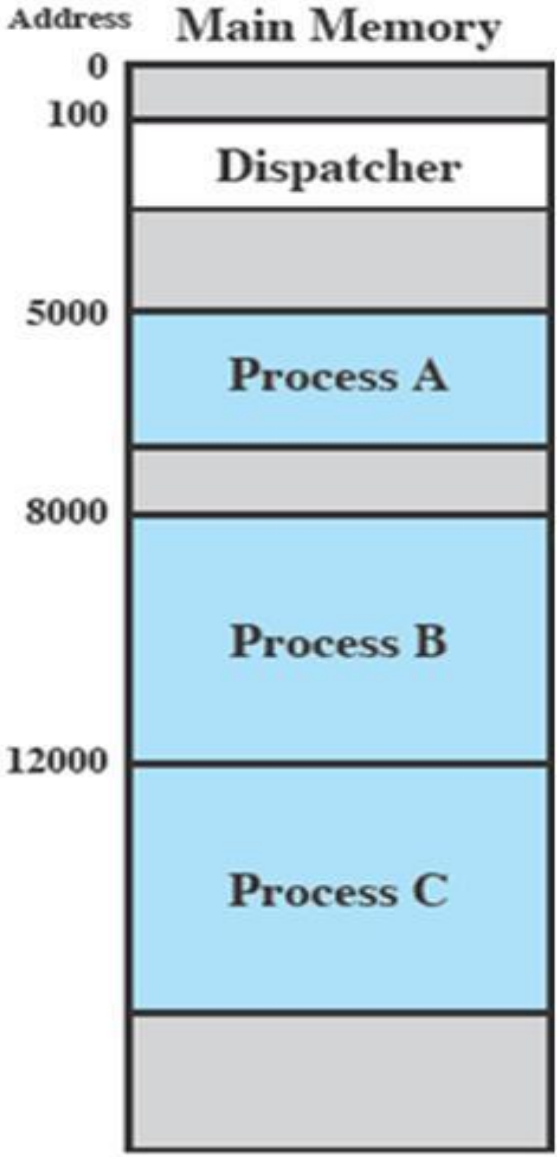
ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag

DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Source: Pearson

Figure 3.12 Pentium II EFLAGS Register

Process Execution and Traces



5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

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

Source: Pearson

Process Execution and Traces

Combined Traces of Processes A, B, and C

1	5000	27	12004
2	5001	28	12005
3	5002	---------- Timeout	
4	5003	29	100
5	5004	30	101
6	5005	31	102
---------- Timeout		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	---------- Timeout	
16	8003	41	100
---------- I/O Request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	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	

Source: Pearson

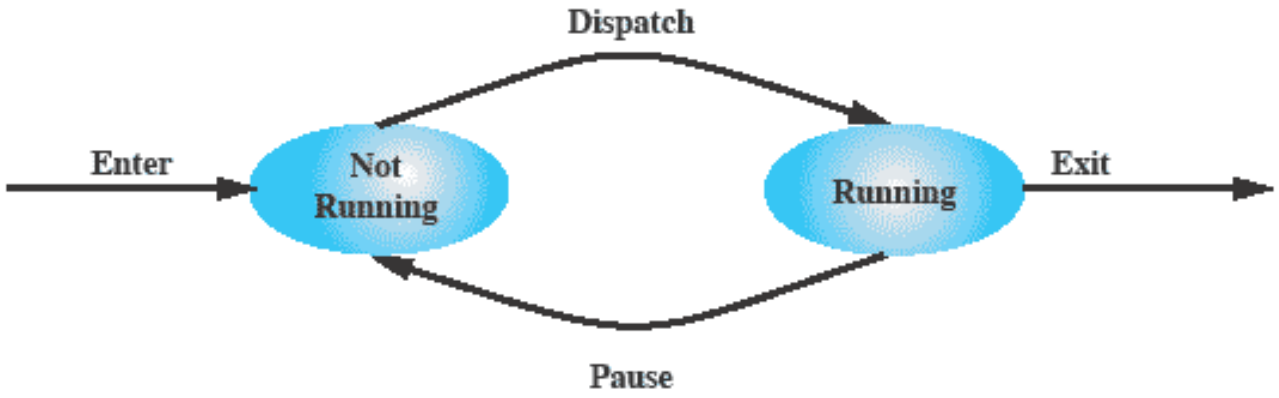
100 = Starting address of dispatcher program

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

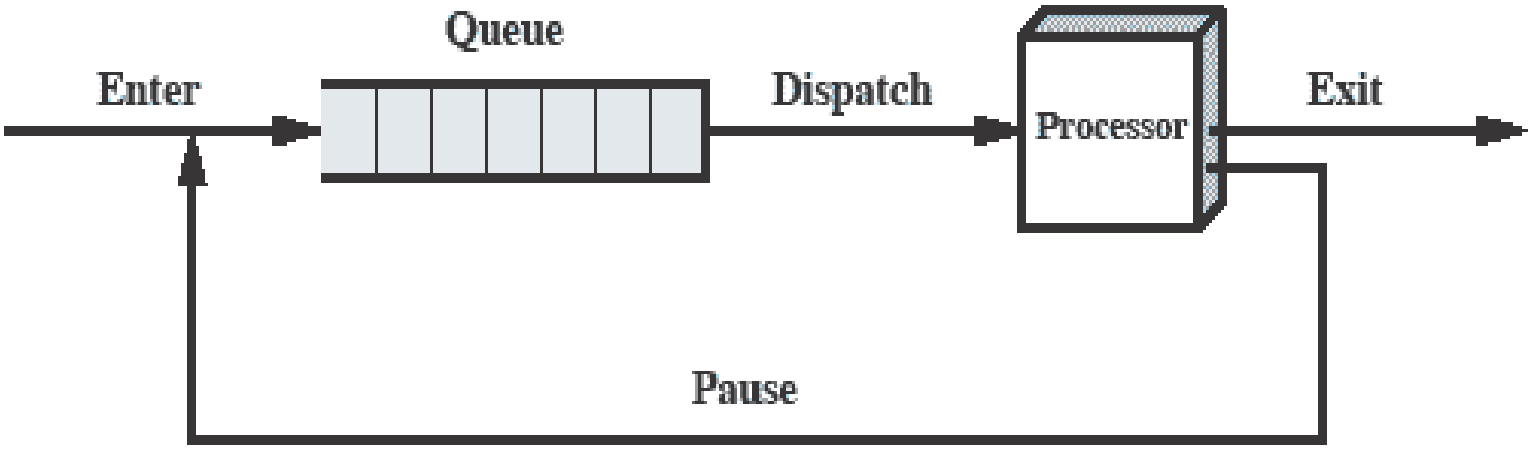
- ***A process may be in one of two states:***
  - Running
  - Not Running



(a) State transition diagram

Source: Pearson

Queuing Diagram



(b) Queuing diagram

Source: Pearson

Process Creation and Termination

- **Process spawning**
  - OS may create a process at the explicit request of another process
    - A new process becomes a *child process* of the *parent process*
- **Process termination**
  - A process may terminate itself by calling a system call called EXIT
    - A batch job include a HALT instruction for termination
    - For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)
  - A process may terminate due to an erroneous condition such as memory unavailable, arithmetic error, or parent process termination, etc.

fork: Creating new processes

- **Process control**
  - Unix provides a number of system calls for manipulating processes
  - Obtain Process ID, Create/Terminate Process, etc.
- ***int fork(void)***
  - Creates a new process (child process) that is identical to the calling process (parent process)
  - Returns 0 to the child process
  - Returns child's `pid` to the parent process

```
if (fork() == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Fork Example #1

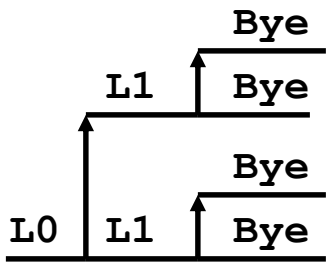
- **Parent and child both run the same code**
  - Distinguish parent from child by return value from `fork`
- **Duplicate but separate address space**
  - Start with same state, but each has private copy
  - Relative ordering of their print statements undefined
- **Shared files**
  - Both parent and child print their output on the same screen

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

- **Both parent and child can continue forking**
- **Process graph**
  - Each horizontal arrow corresponds to a process
  - Each vertical arrow corresponds to the execution of a *fork* function

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

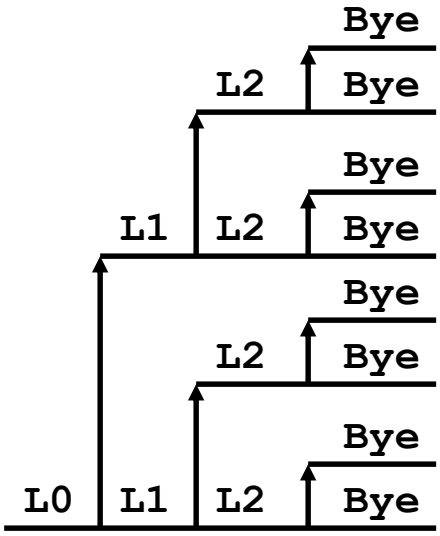


Fork Example #3

Key Points

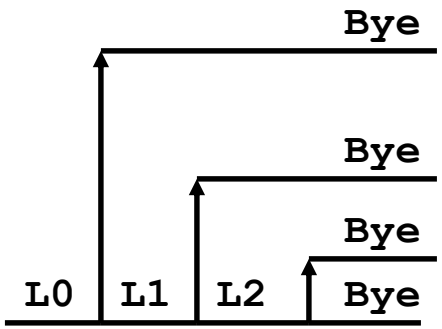
- Both parent and child can continue forking

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



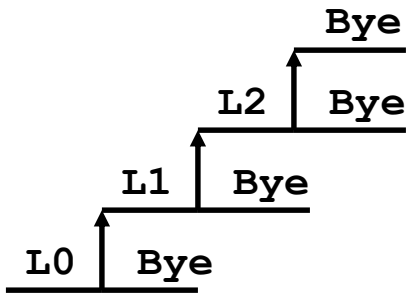
Fork Example #4

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Fork Example #5

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



exit: Destroying Process

- ***void exit(int status)***
  - Terminate a process with an *exit status*
    - Normally with status 0
  - `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Five-State Process Model

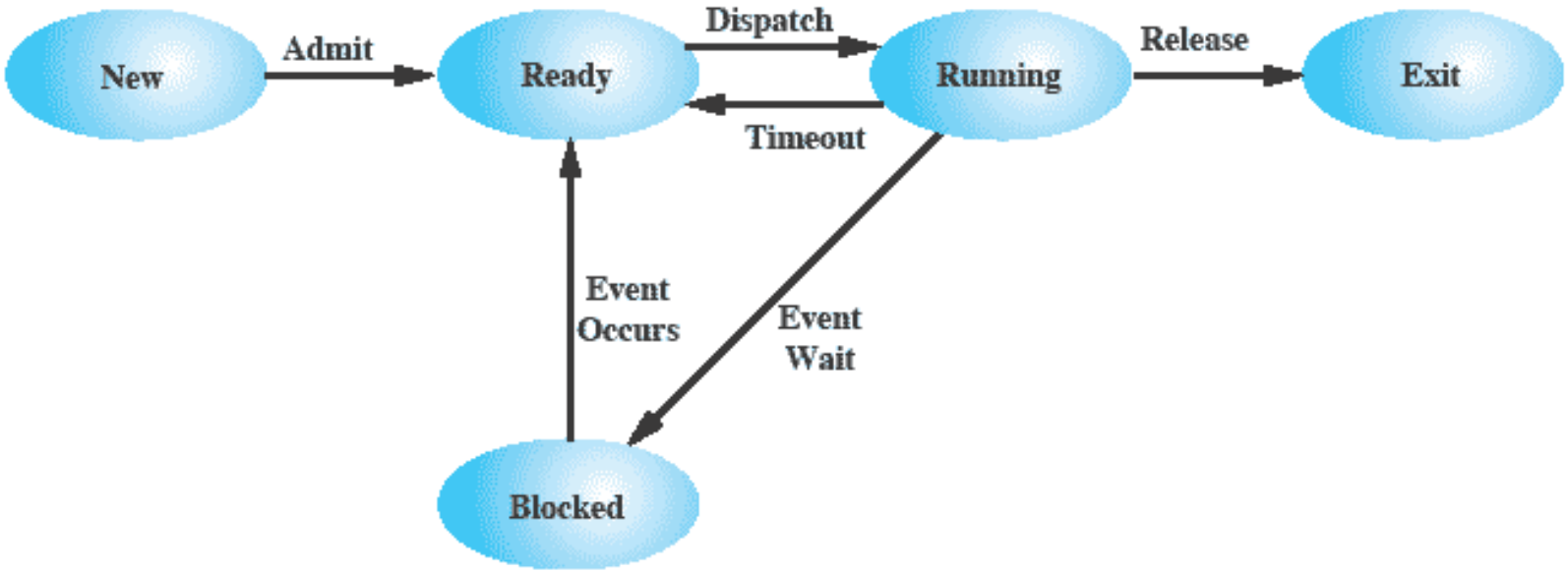


Figure 3.6 Five-State Process Model

Source: Pearson

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.

Source: Pearson

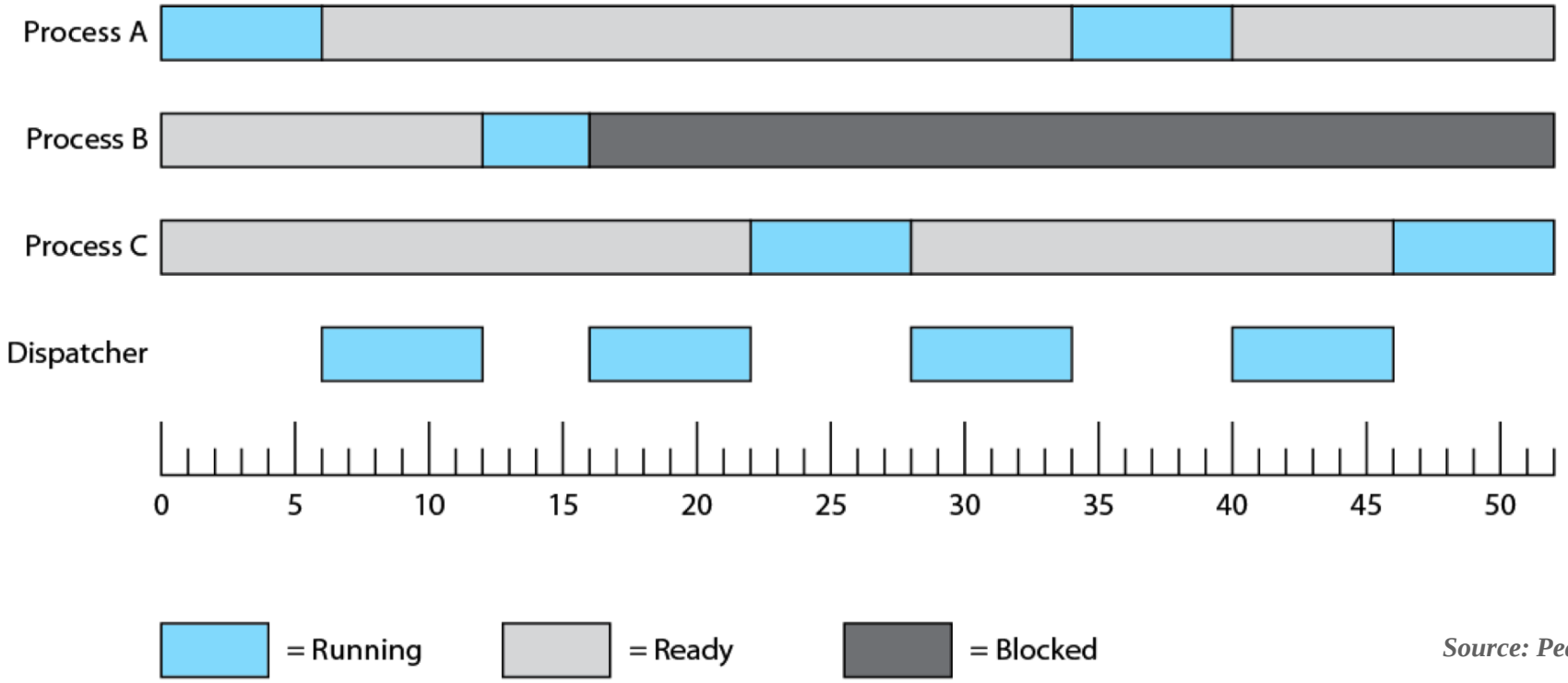
Reasons for Process Termination

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.
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 (e.g., 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.

Source: Pearson

Example

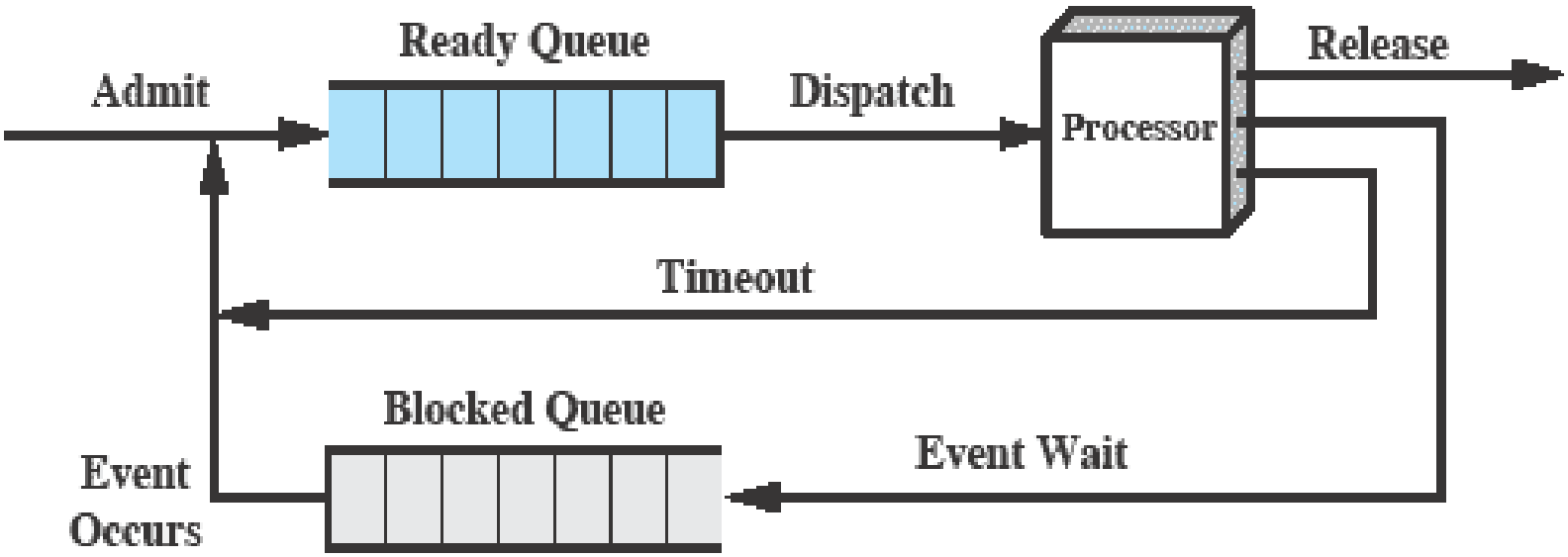
Process States for Trace of Figure 3.4



Source: Pearson

Figure 3.7 Process States for Trace of Figure 3.4

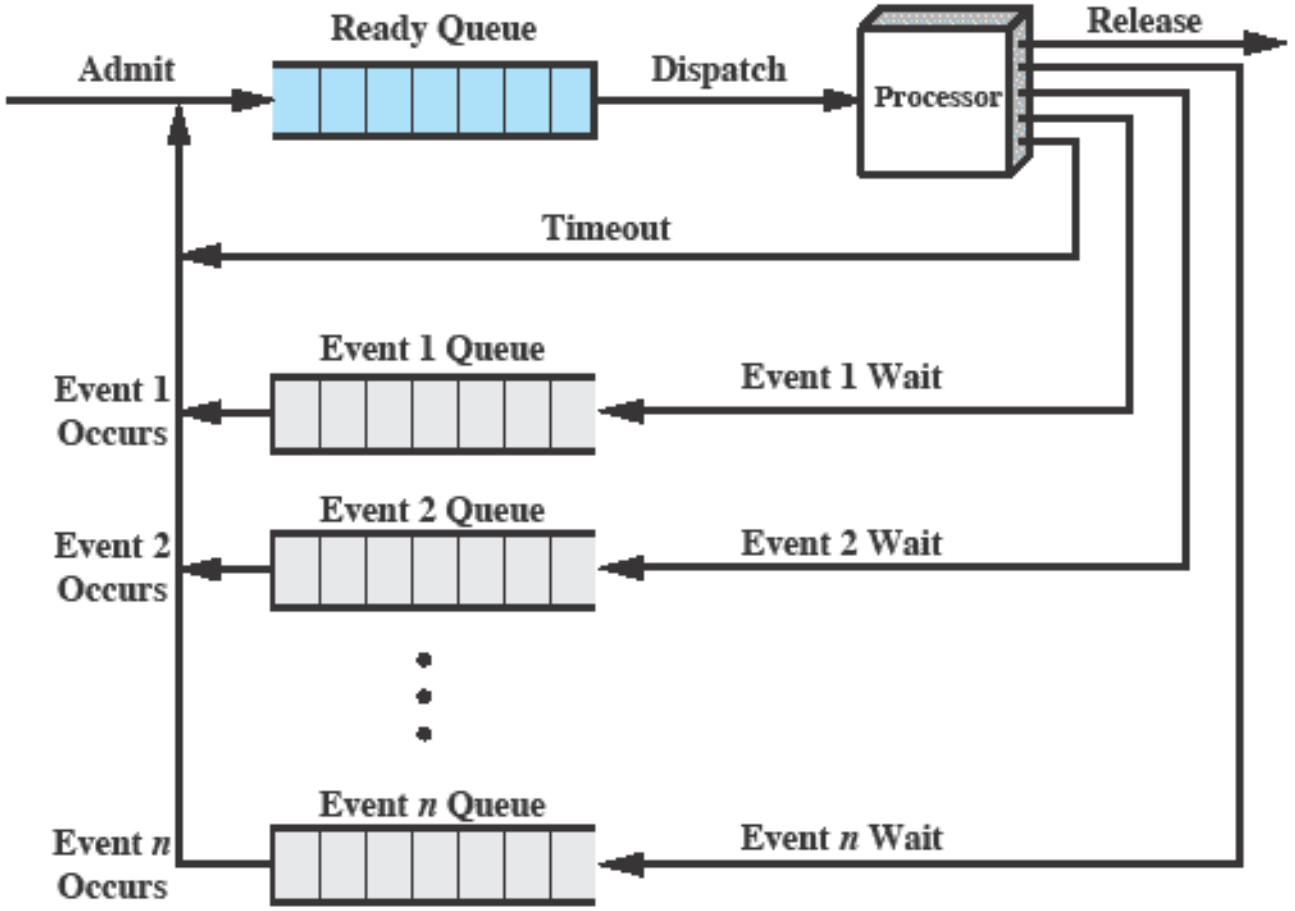
Using Two Queues



(a) Single blocked queue

Source: Pearson

Multiple Blocked Queues



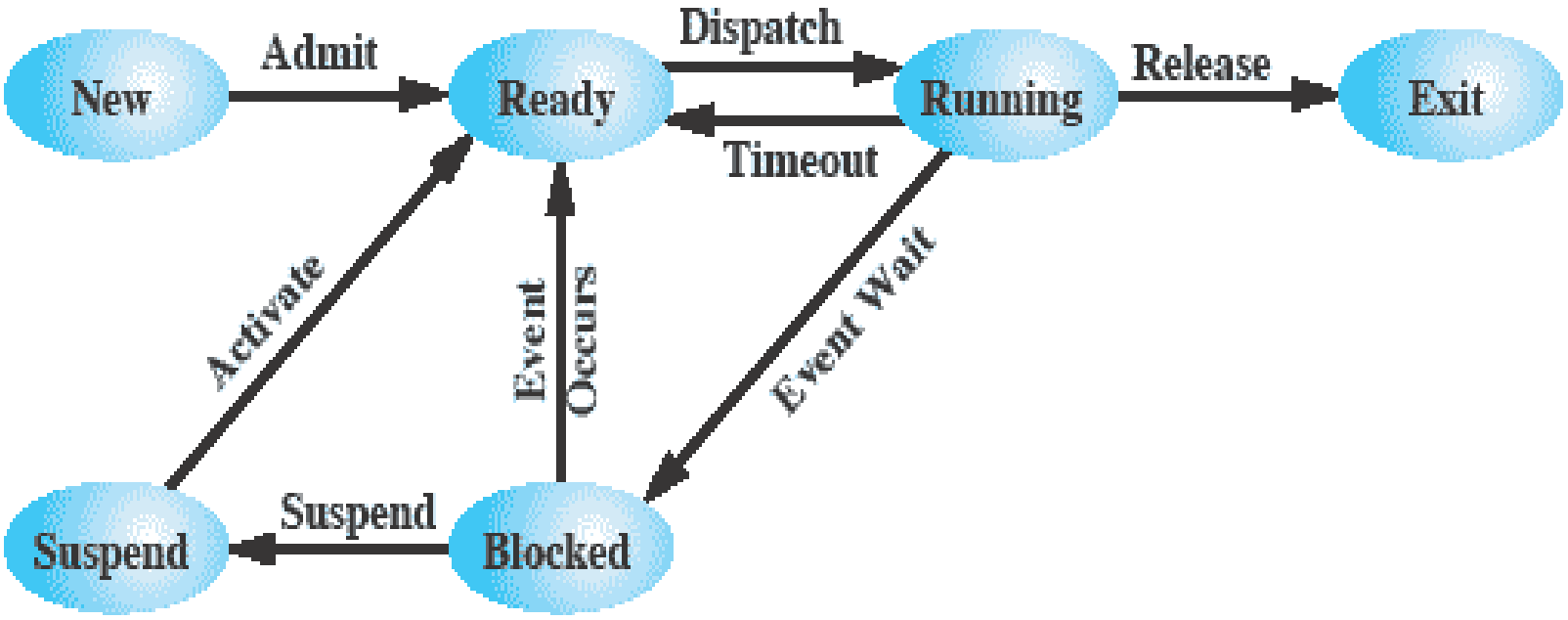
(b) Multiple blocked queues

Source: Pearson

Suspended Processes

- **Swapping**
  - Sometimes, all the processes in memory are waiting for I/O
  - Involves moving part or all of a process from main memory to disk
  - Then, OS brings in another (new or suspended) process into memory
- ***Suspended Process***
  - The process is swapped out and is not immediately available for execution
  - The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution
  - The process may or may not be waiting on an event

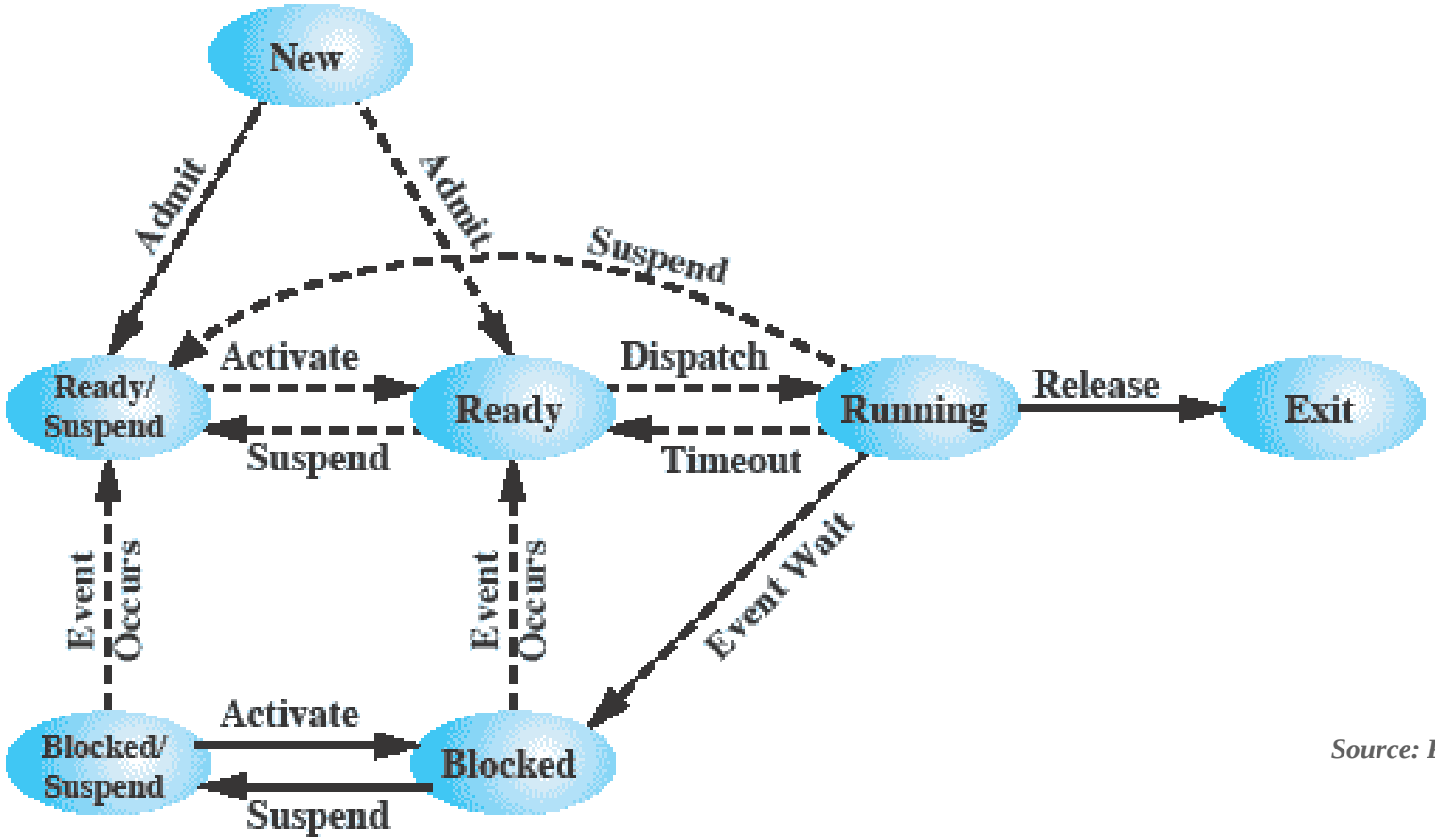
Suspend State



(a) With One Suspend State

Source: Pearson

Two Suspend States



Source: Pearson

(b) With Two Suspend States

Processes and Resources

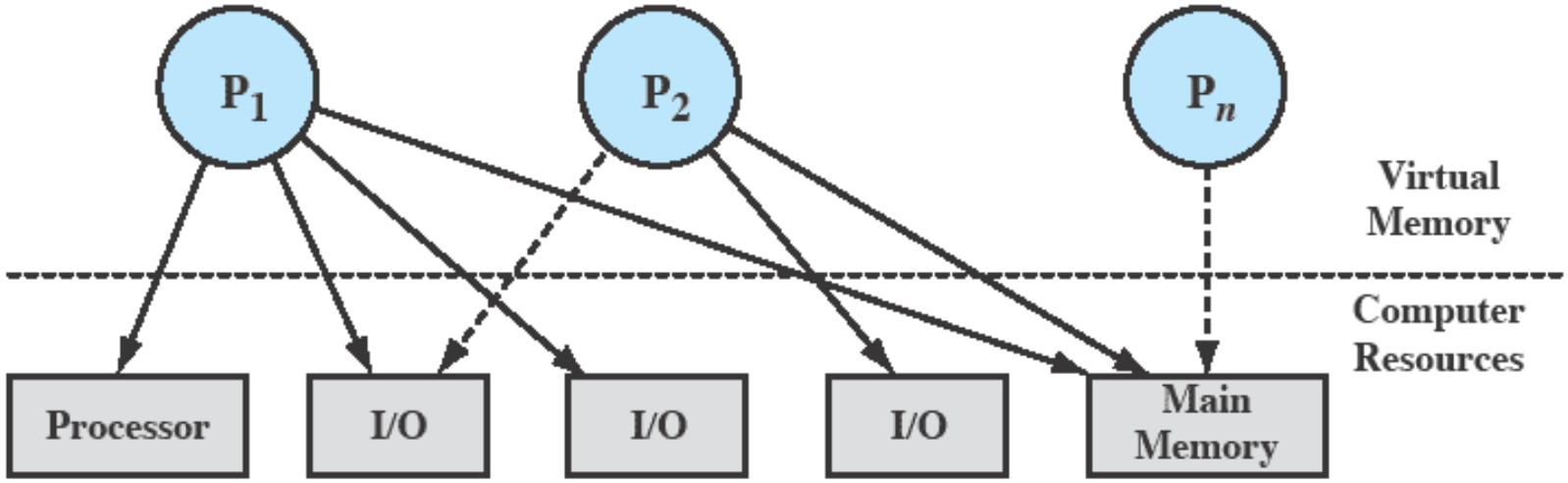


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

Source: Pearson

Interrupt/Exception

Interrupts

- Forced transfer of control to a procedure (*handler*) due to external events (*interrupt*) or due to an internal erroneous condition during program execution (*exception*)
- Exception
  - Generated by an instruction in the currently running process due to an erroneous condition
  - Synchronous, internal
- Interrupt
  - Asynchronous, external events

Interrupt handling mechanism

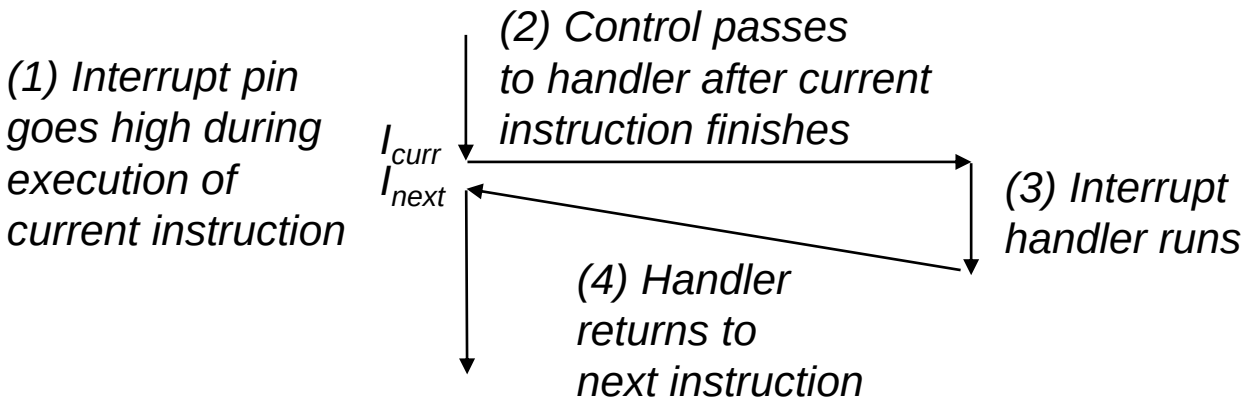
- Should allow interrupts/exceptions to be handled transparently to the executing process (application programs and operating system)
- Procedure
  - When an interrupt is received or an exception condition is detected, the current task is suspended and the control automatically transfers to a handler
  - After the handler is complete, the interrupted task resumes without loss of continuity, unless recovery is not possible or the interrupt causes the currently running task to be terminated.

(Synchronous) Exceptions

- **Caused by an event that occurs as a result of executing an instruction:**
- ***Traps***
  - *Intentional* exceptions
  - Examples: system calls, breakpoints (debug)
  - Returns control to "*next*" instruction
- ***Faults***
  - *Unintentional* but possibly recoverable
  - Examples: page faults (recoverable), protection faults (unrecoverable), arithmetic exceptions (i.e. divide by zero, NaN).
  - Either re-executes faulting ("*current*") instruction or terminate the process
- ***Aborts***
  - Unintentional and *unrecoverable fatal* errors
  - Examples: parity error, machine check abort.
  - Aborts the current process, and probably the entire system

(Asynchronous) Interrupt

- **Caused by an event external to the processor**
  - Indicated by setting the processor's interrupt pins (#INT, #NMI)
  - Handler returns to "*next*" instruction.
- ***Examples:***
  - I/O interrupts
    - Hitting ctl-c at the keyboard, arrival of a packet from the network, arrival of a data sector from a disk
  - Hard reset interrupt: hitting the reset button
  - Soft reset interrupt: hitting ctl-alt-delete on a PC



(External) Interrupt

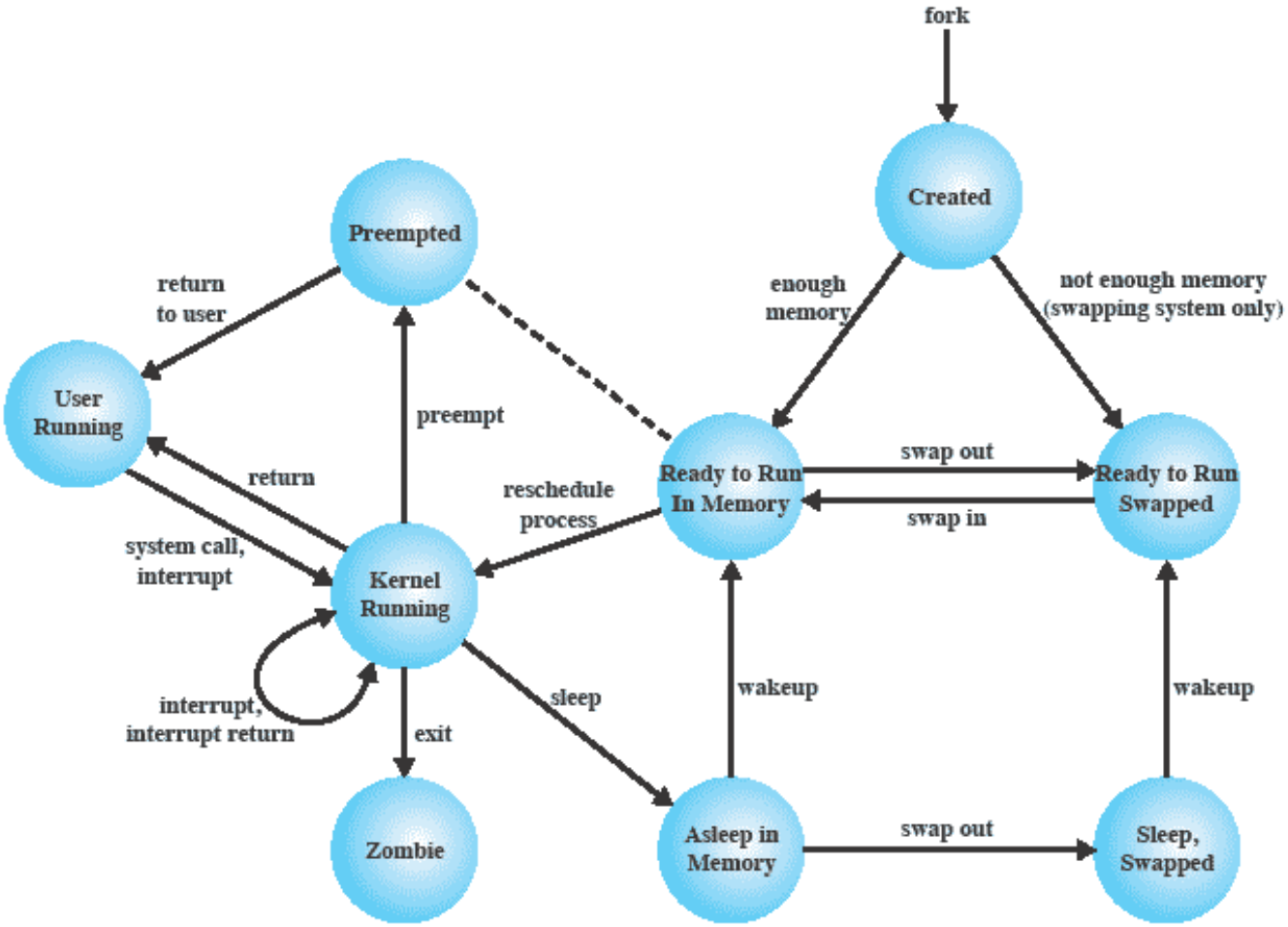
Interrupt Classification

- Maskable interrupt
  - Can be disabled/enabled by an instruction
  - Generated by asserting INT pin
- Non-maskable interrupt (NMI)
  - Cannot be disabled by program
  - Received on the processor's NMI pin
- Software interrupt
  - Generated by INT n instruction
    - INT instruction can be used to generate an interrupt or an exception by using a vector number as an operand
  - Viewed as an implicit call to interrupt handler of interrupt vector n
  - No mechanism for masking interrupts

UNIX System V Process Management

- ***Two categories of processes***
  - System Process
    - Run in kernel mode and execute OS code
    - Perform administrative and housekeeping functions such as memory allocation and process swapping
  - User Process
    - Run in user mode to execute user programs
    - Enter kernel mode by issuing a system call, by an exception or by an interrupt
- ***Process states***
  - 9 states with the following characteristics
    - 2 running states: '*kernel running*', '*user running*'
    - A distinction is made between '*Ready to run in memory*' and '*preempted*' states
      - They are essentially the same state

UNIX Process States



Source: Pearson

Figure 3.17 UNIX Process State Transition Diagram

UNIX Process Context

User-Level Context	
Process text	Executable machine instructions of the program
Process data	Data accessible by the program of this process
User stack	Contains the arguments, local variables, and pointers for functions executing in user mode
Shared memory	Memory shared with other processes, used for interprocess communication
Register Context	
Program counter	Address of next instruction to be executed; may be in kernel or user memory space of this process
Processor status register	Contains the hardware status at the time of preemption; contents and format are hardware dependent
Stack pointer	Points to the top of the kernel or user stack, depending on the mode of operation at the time or preemption
General-purpose registers	Hardware dependent
System-Level Context	
Process table entry	Defines state of a process; this information is always accessible to the operating system
U (user) area	Process control information that needs to be accessed only in the context of the process
Per process region table	Defines the mapping from virtual to physical addresses; also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute
Kernel stack	Contains the stack frame of kernel procedures as the process executes in kernel mode

Source: Pearson

UNIX Process Table Entry

Process status	Current state of process.
Pointers	To U area and process memory area (text, data, stack).
Process size	Enables the operating system to know how much space to allocate the process.
User identifiers	The **real user ID** identifies the user who is responsible for the running process. The **effective user ID** may be used by a process to gain temporary privileges associated with a particular program; while that program is being executed as part of the process, the process operates with the effective user ID.
Process identifiers	ID of this process; ID of parent process. These are set up when the process enters the Created state during the fork system call.
Event descriptor	Valid when a process is in a sleeping state; when the event occurs, the process is transferred to a ready-to-run state.
Priority	Used for process scheduling.
Signal	Enumerates signals sent to a process but not yet handled.
Timers	Include process execution time, kernel resource utilization, and user-set timer used to send alarm signal to a process.
P_link	Pointer to the next link in the ready queue (valid if process is ready to execute).
Memory status	Indicates whether process image is in main memory or swapped out. If it is in memory, this field also indicates whether it may be swapped out or is temporarily locked into main memory.

Source: Pearson

Homework 2

- *3.1*
- *3.2*
- *3.3*
- *3.5*
- *3.6*
- *Read Chapter 4*