Operating System

# Chapter 3. Process

*Lynn Choi*

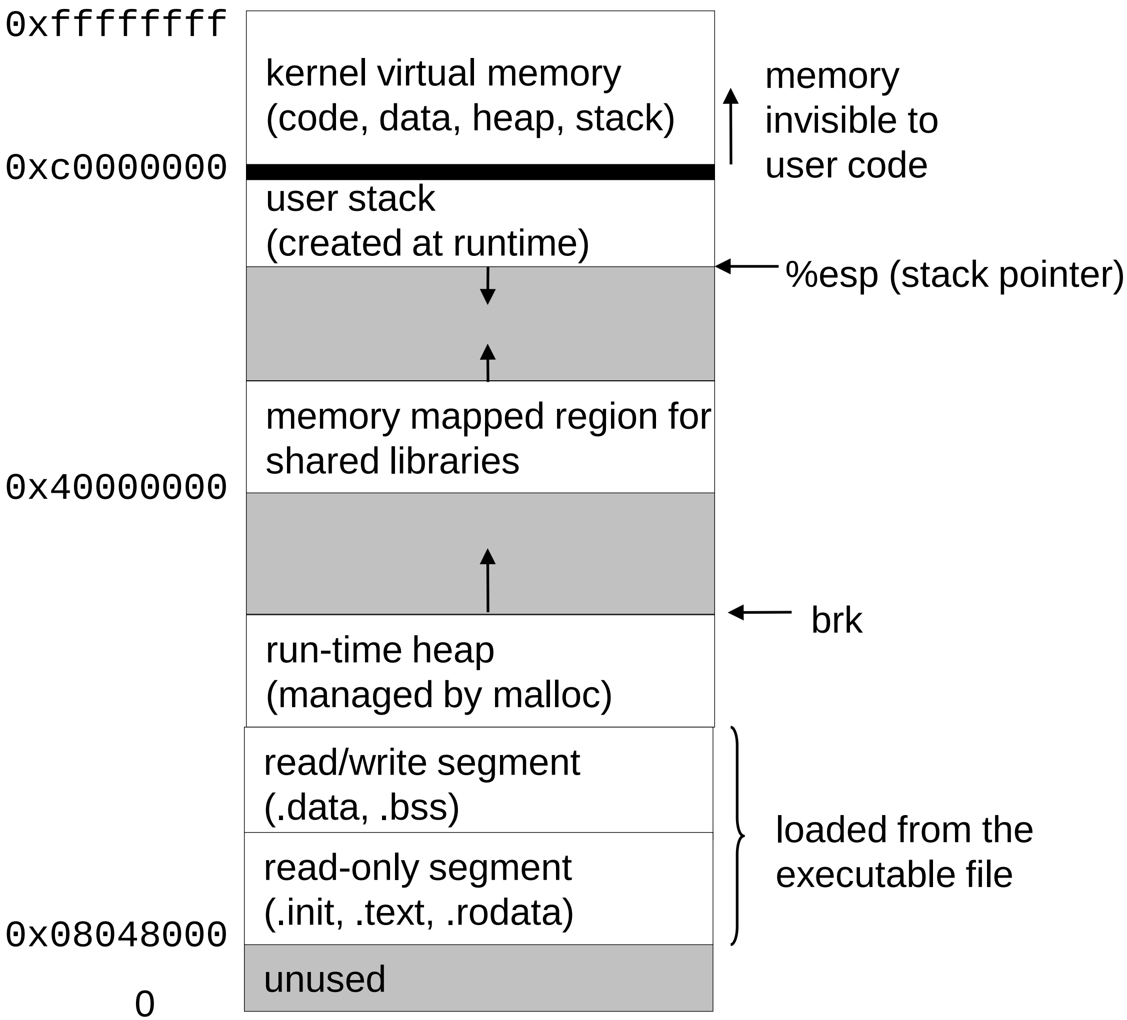
*School of Electrical Engineering*

# 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.




Source: Pearson

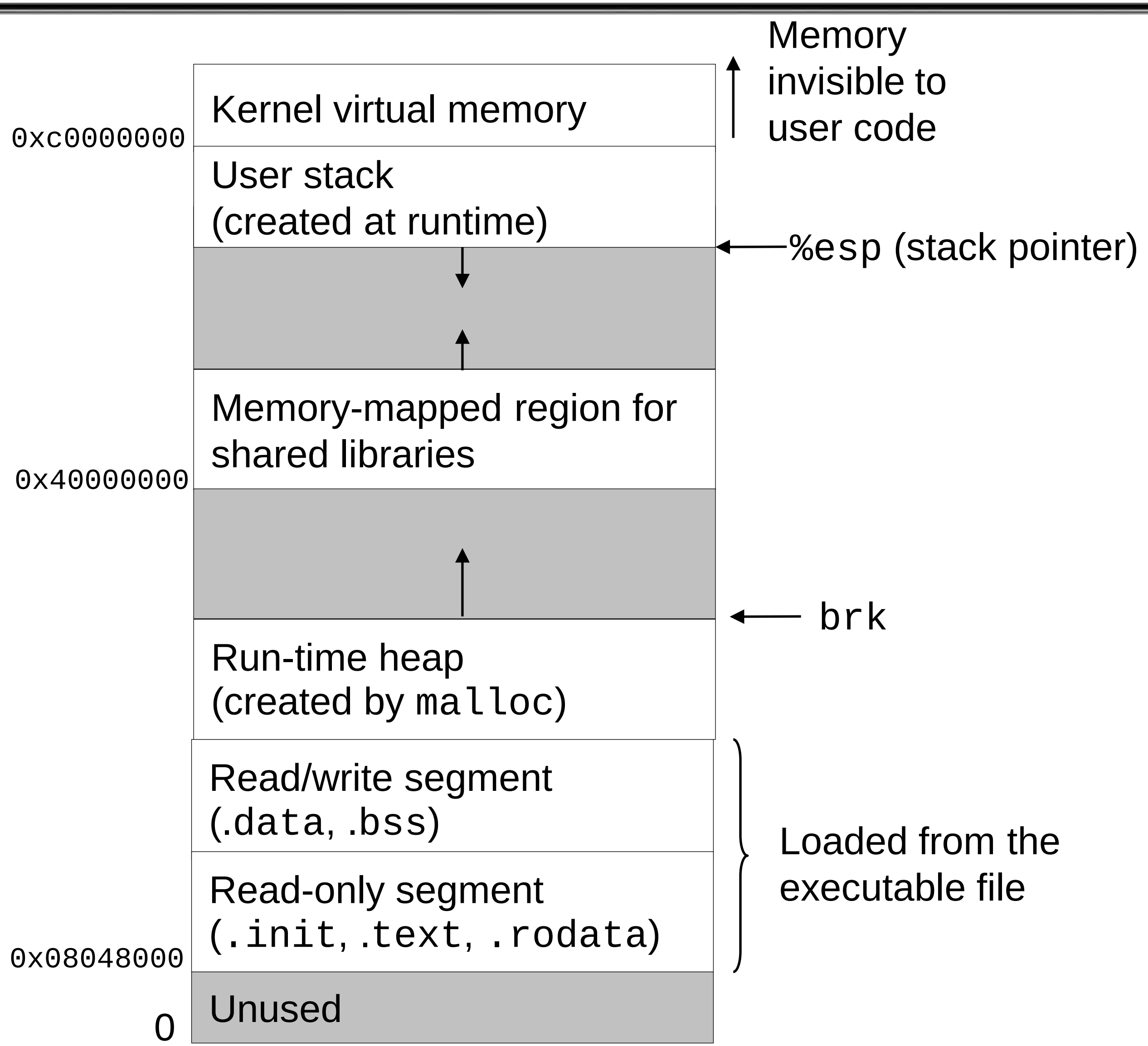
# Life and Scope of an Object

- ***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

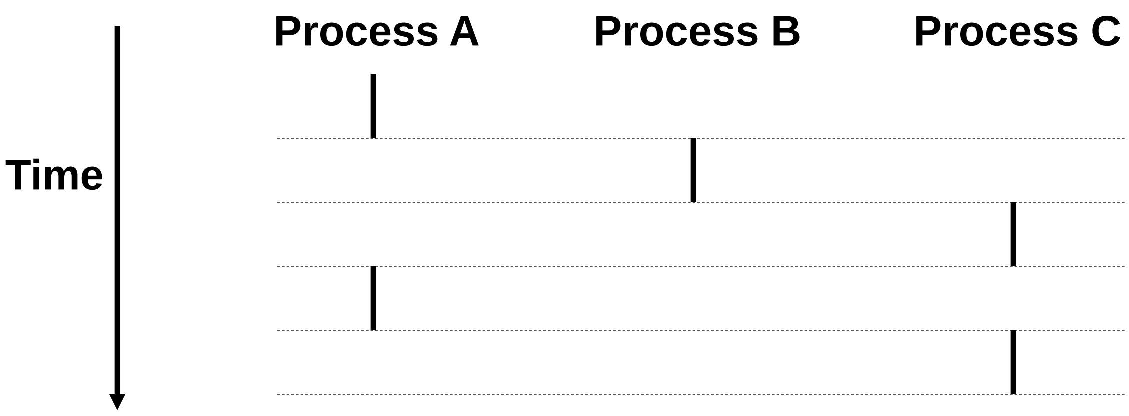
# Linux Run-time Memory Image




*Source: Pearson*

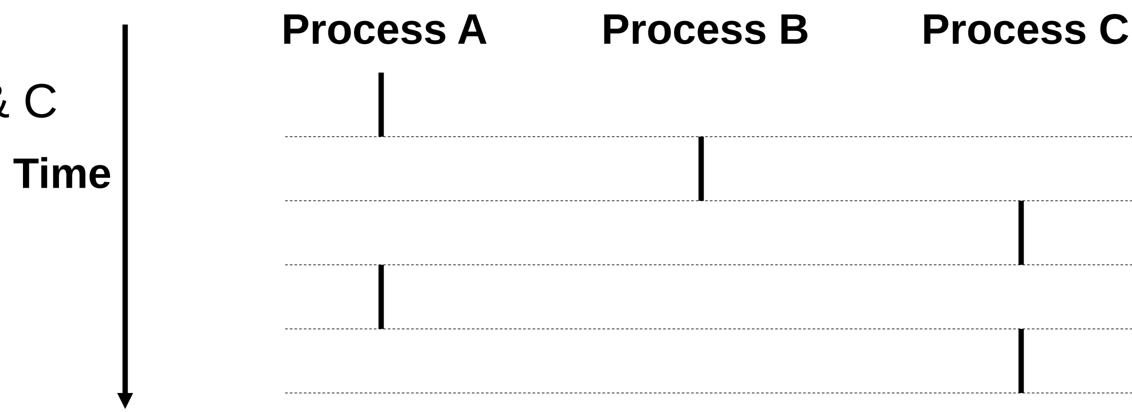
# Logical Control Flows

**Each process has its own logical control flow**

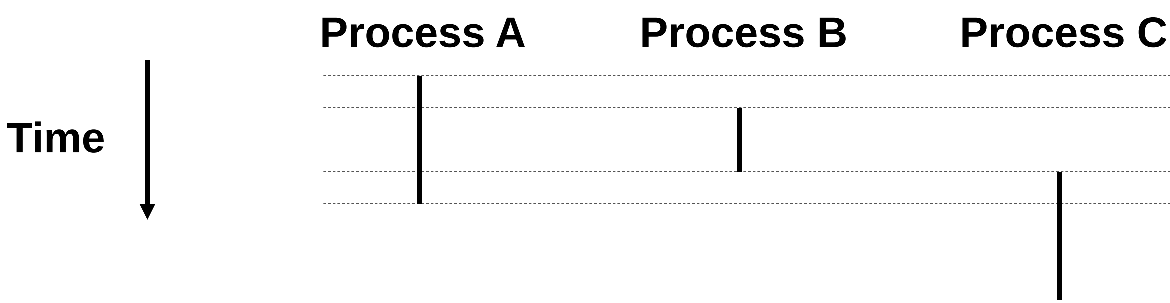


# Concurrent Processes

- **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.



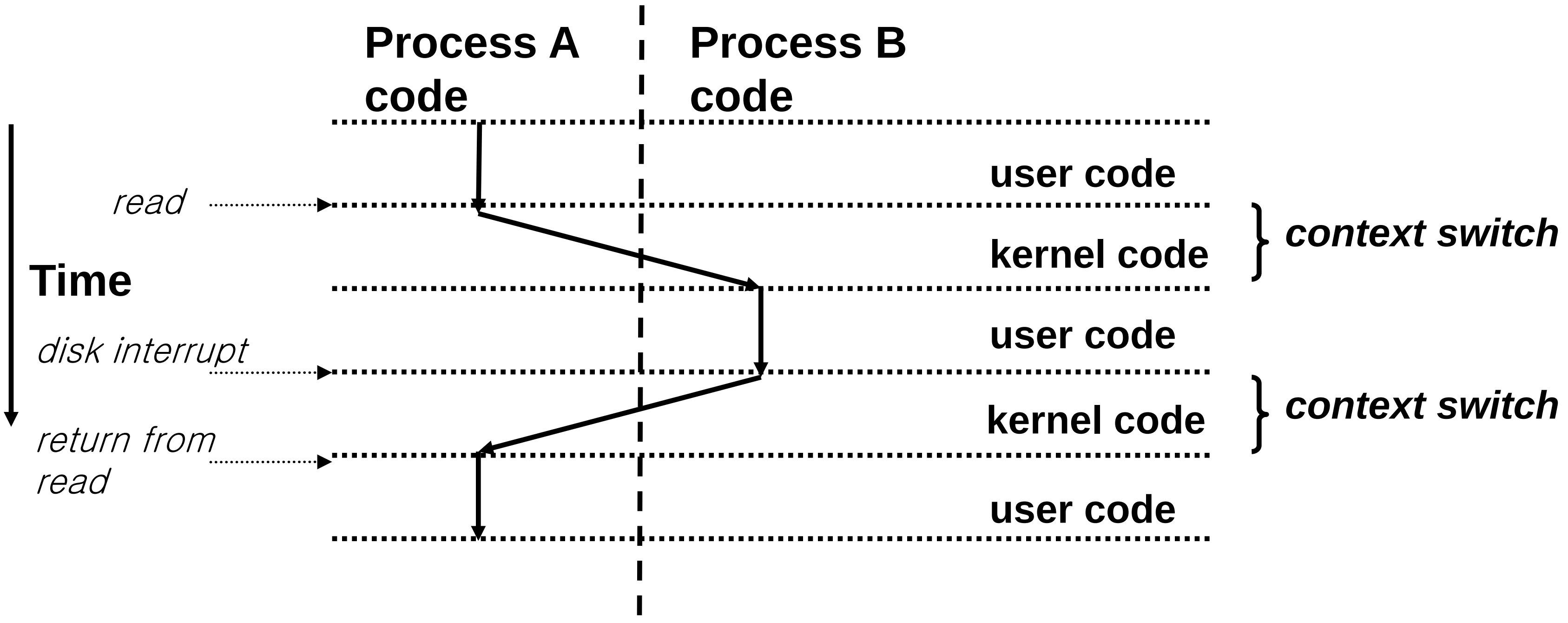
# Context Switching

- **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

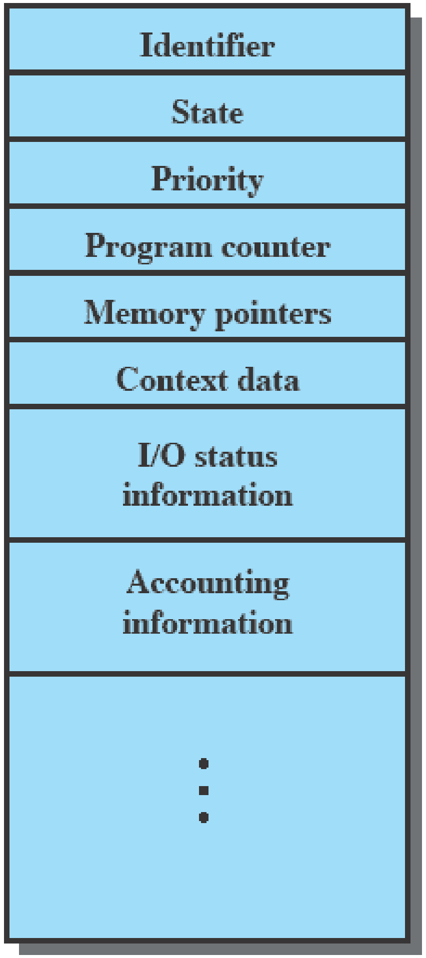
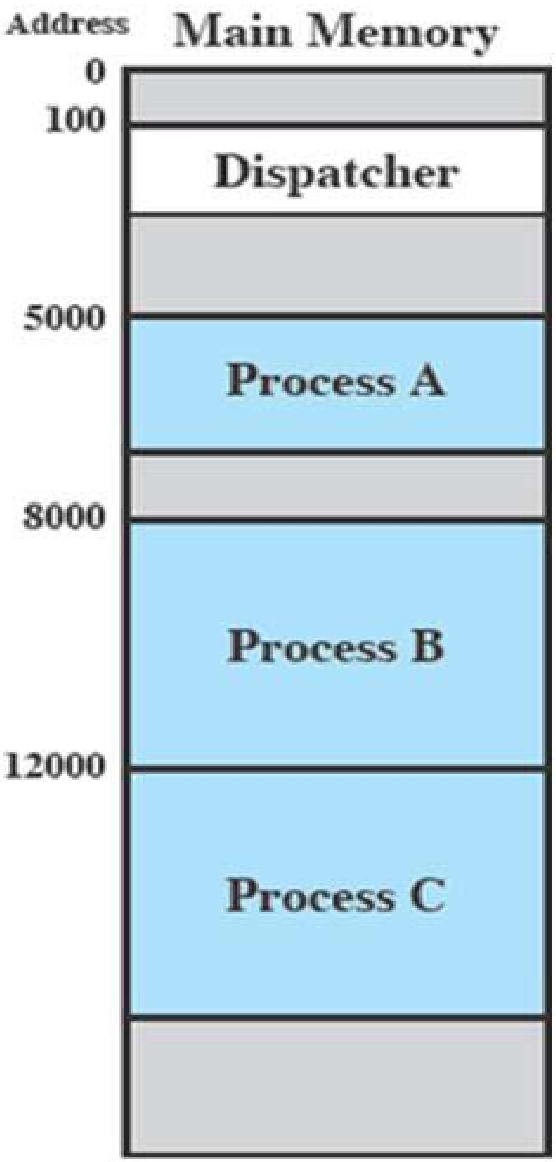



**Figure 3.1 Simplified Process Control Block**

*Source: Pearson*

# Process 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
2 5001
3 5002
4 5003
5 5004
6 5005
------------------ Timeout
7 100
8 101
9 102
10 103
11 104
12 105
13 8000
14 8001
15 8002
16 8003
-------------- I/O Request
17 100
18 101
19 102
20 103
21 104
22 105
23 12000
24 12001
25 12002
26 12003
27 12004
28 12005
------------------ Timeout
29 100
30 101
31 102
32 103
33 104
34 105
35 5006
36 5007
37 5008
38 5009
39 5010
40 5011
------------------ Timeout
41 100
42 101
43 102
44 103
45 104
46 105
47 12006
48 12007
49 12008
50 12009
51 12010
52 12011
------------------ Timeout

100 = Starting address of dispatcher program

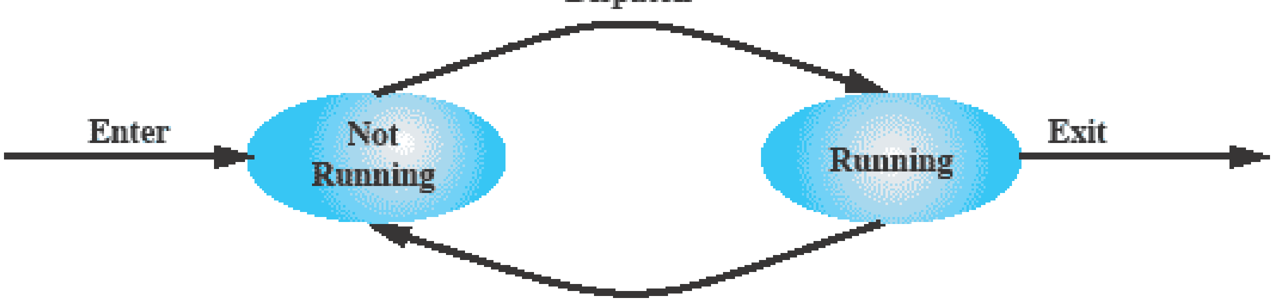
Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

**Figure 3.4 Combined Trace of Processes of Figure 3.2**

*Source: Pearson*

# Two-State Process Model

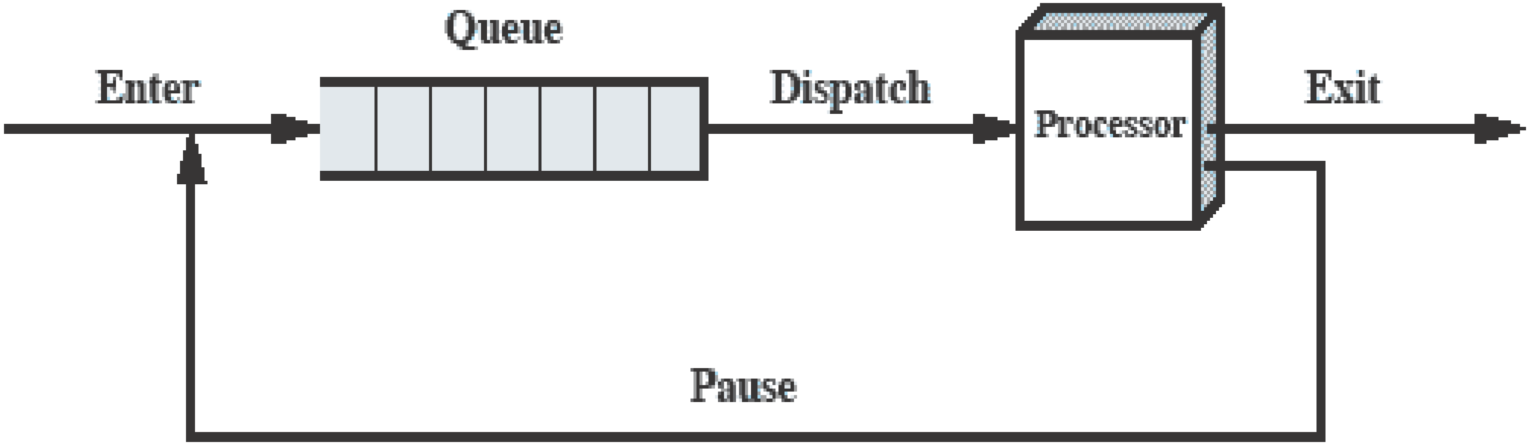
- ***A process may be in one of two states:***
  - Running
  - Not Running




(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 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

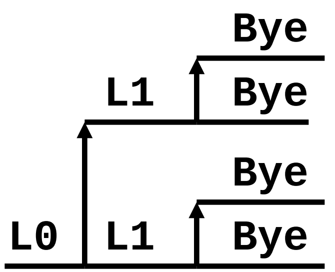
- **Parent and child both run the same code**
  - Distinguish parent from child by return value from `fork`
- **Duplicate but separate address space**
  - Start with same state, but each has private copy
  - Relative ordering of their print statements undefined
- **Shared files**
  - Both parent and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

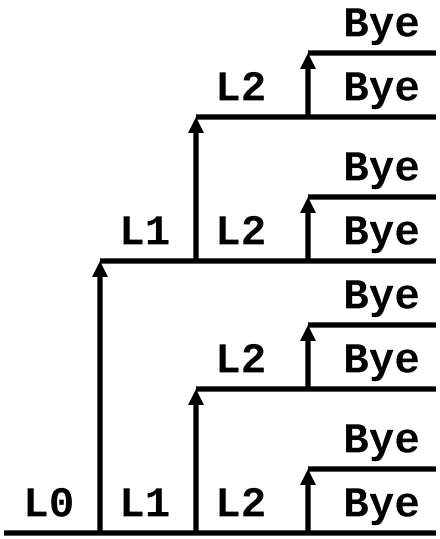


# Fork Example #3

## ❑ Key Points

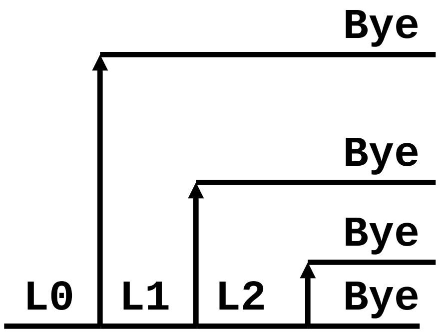
- Both parent and child can continue forking

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



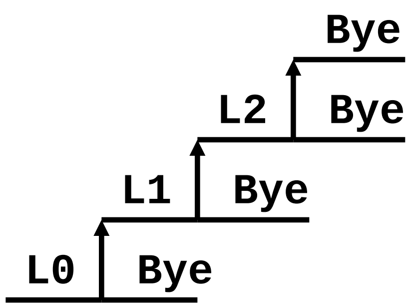
# Fork Example #4

```
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

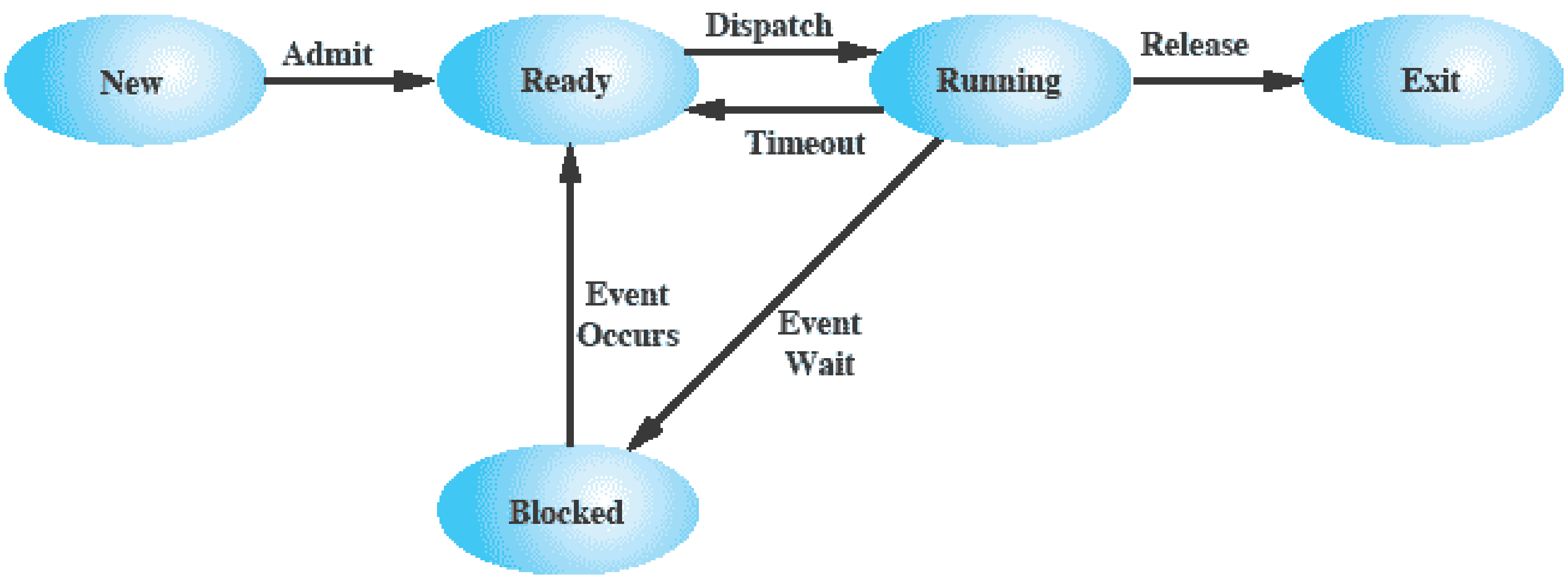



Figure 3.6 Five-State Process Model

*Source: Pearson*

# Example

## Process States for Trace of Figure 3.4

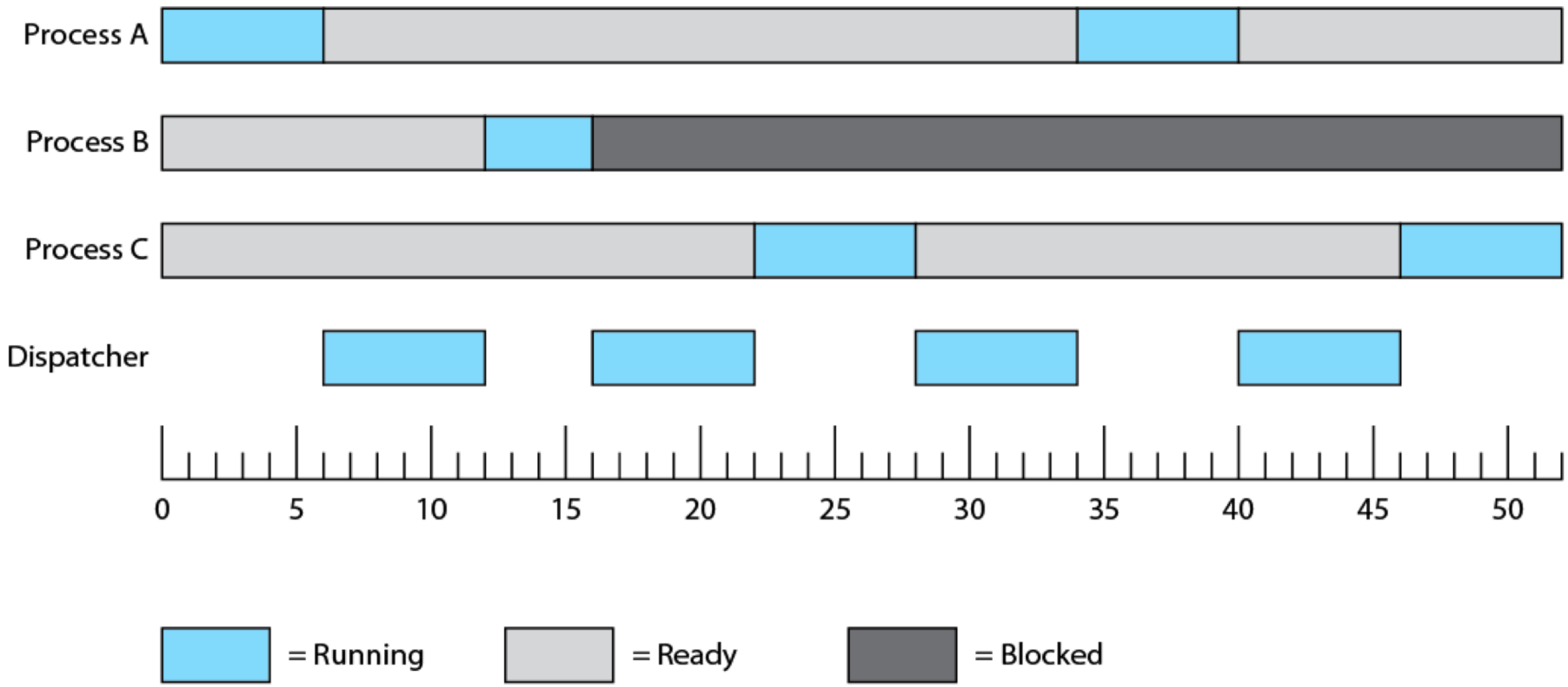
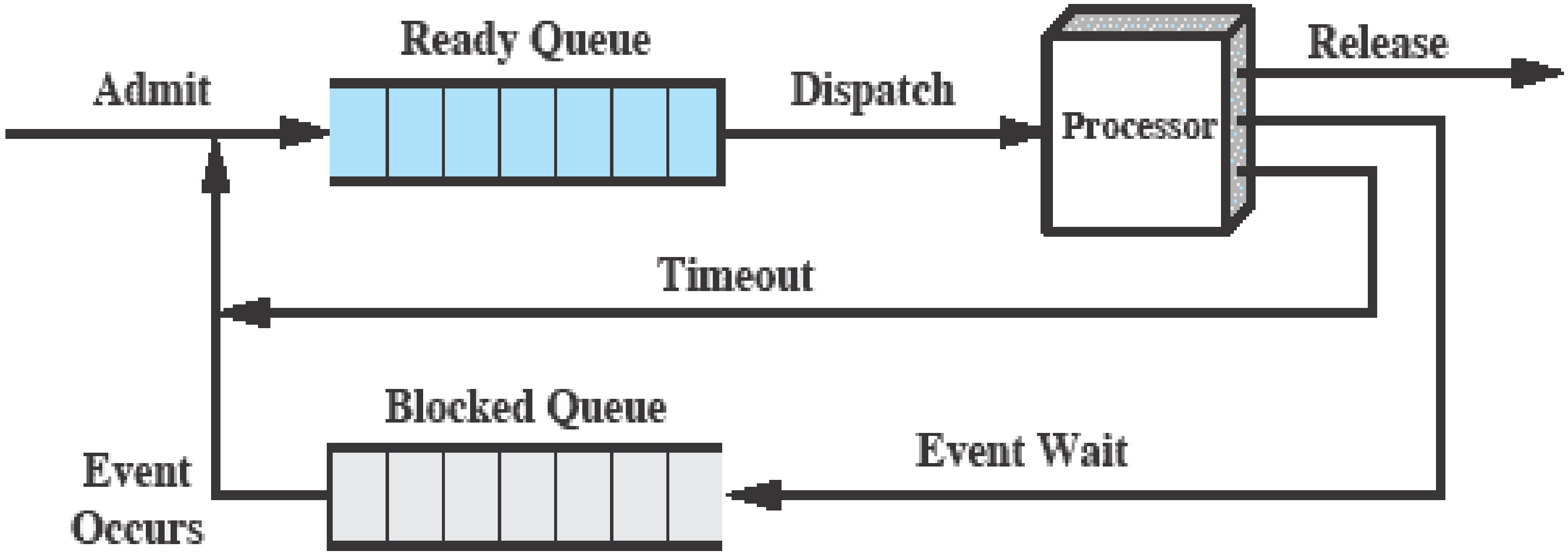



Figure 3.7 Process States for Trace of Figure 3.4

*Source: Pearson*

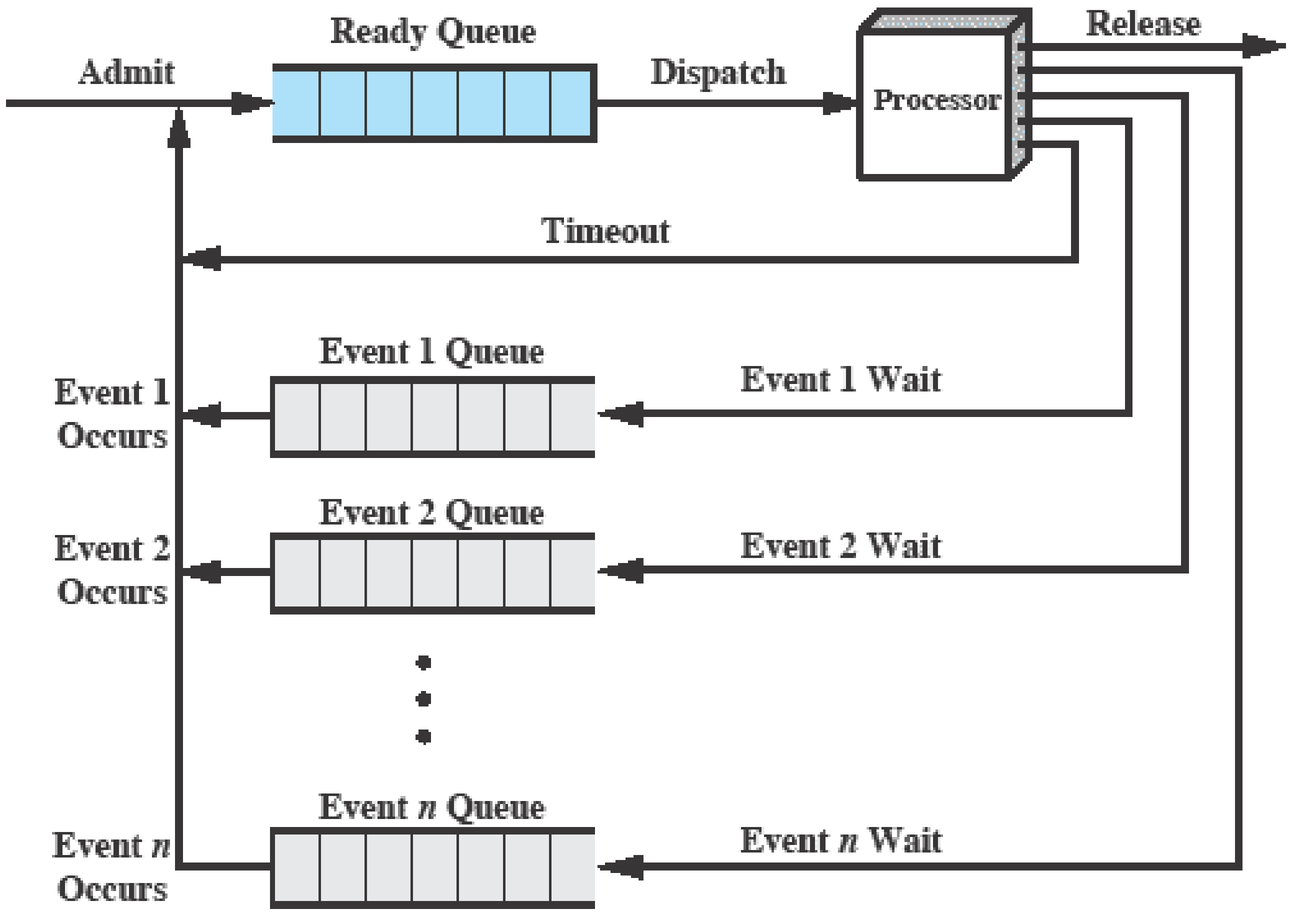
# Using Two Queues




(a) Single blocked queue

*Source: Pearson*

# Multiple Blocked Queues



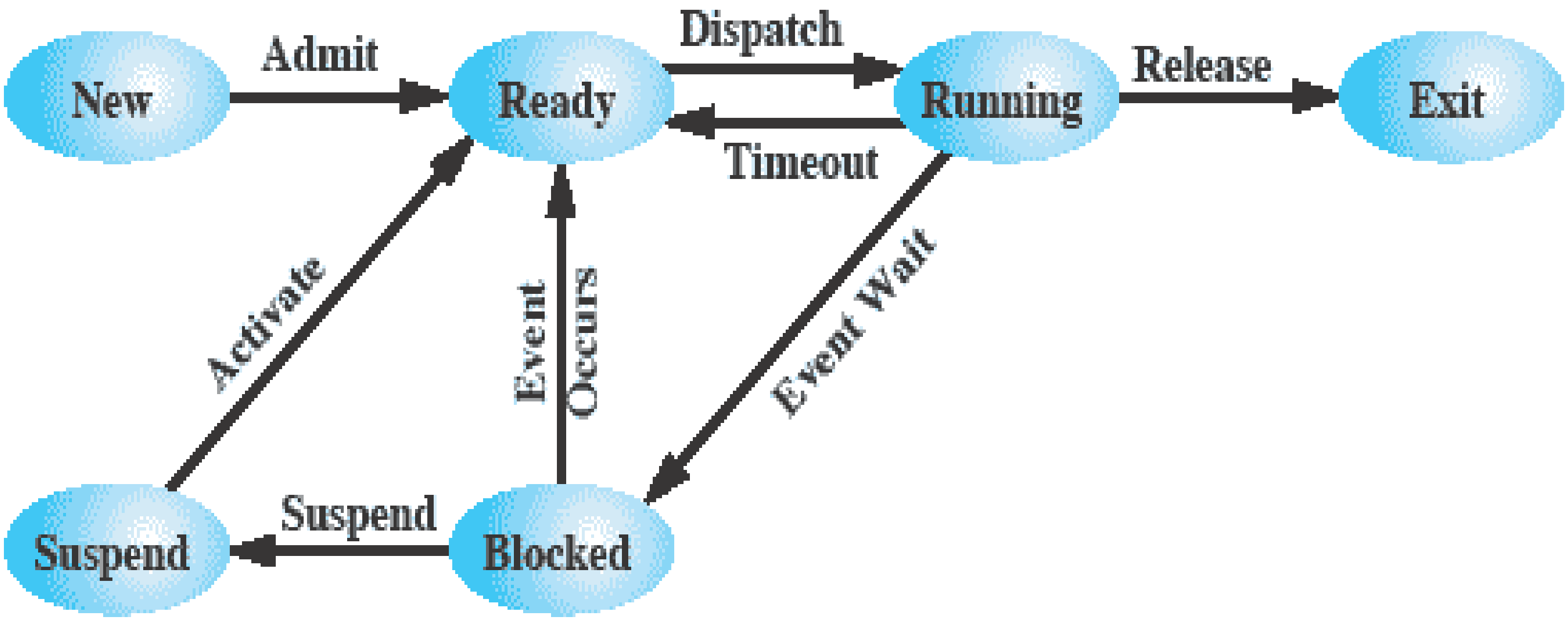

(b) Multiple blocked queues

*Source: Pearson*

# Suspended Processes

- **Swapping**
  - Involves moving part or all of a process from main memory to disk
- ***Suspended Process***
  - The process 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

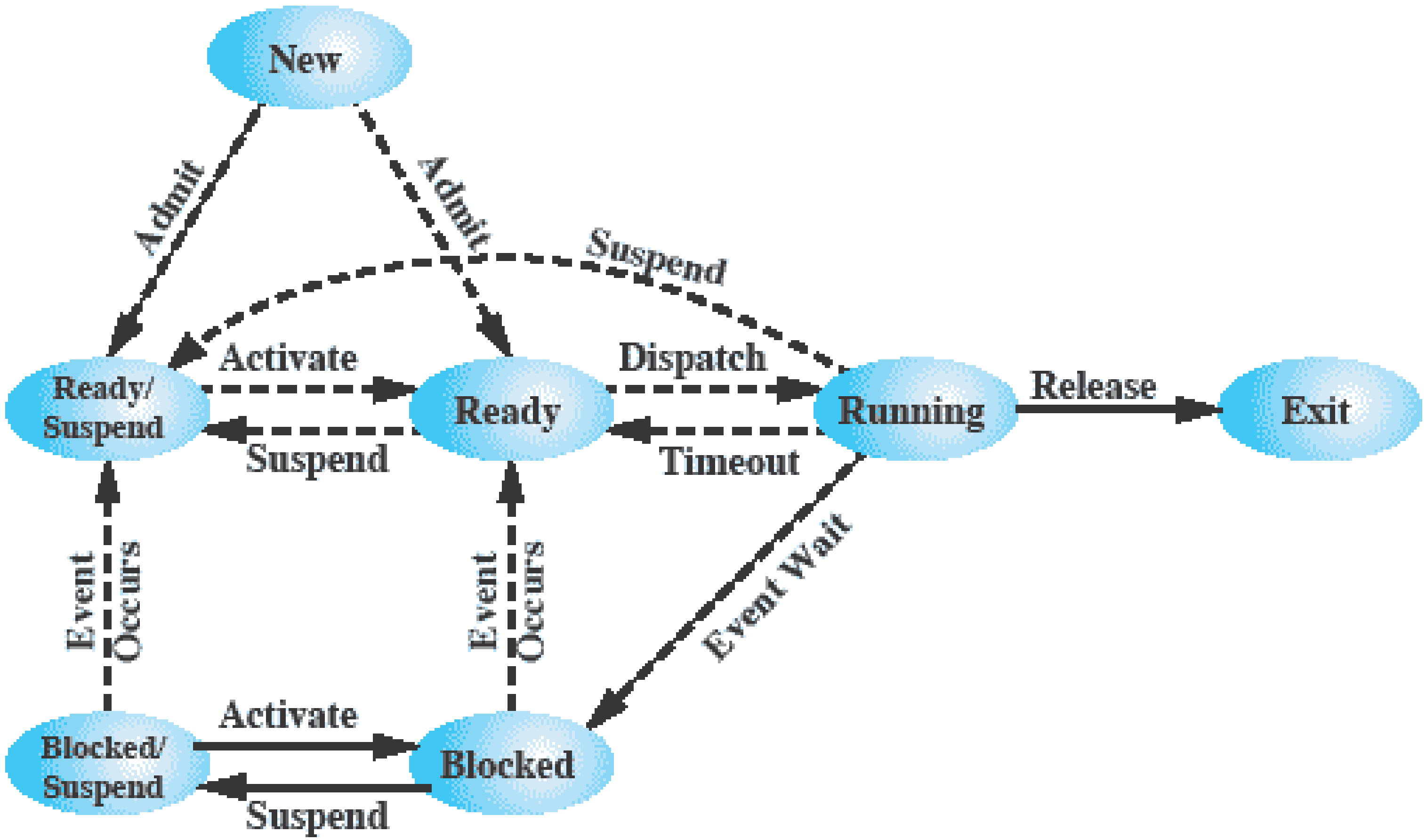
# Suspend State




(a) With One Suspend State

*Source: Pearson*

# Two Suspend States




(b) With Two Suspend States

*Source: Pearson*

# Processes and Resources

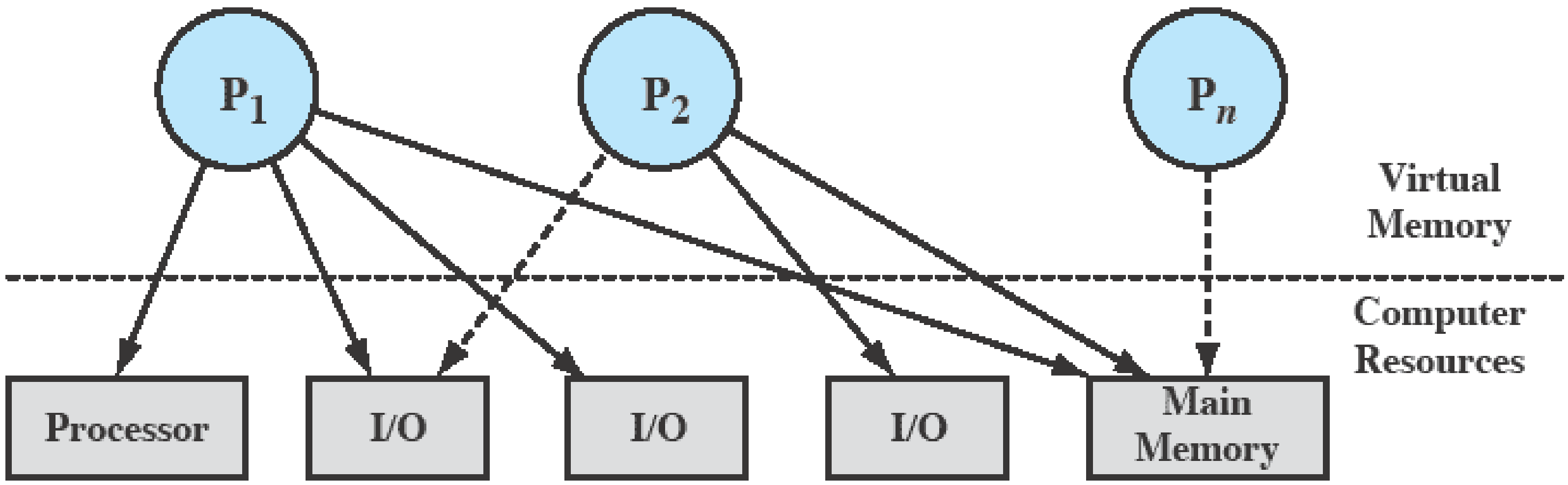



Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

*Source: Pearson*

# Interrupt/Exception

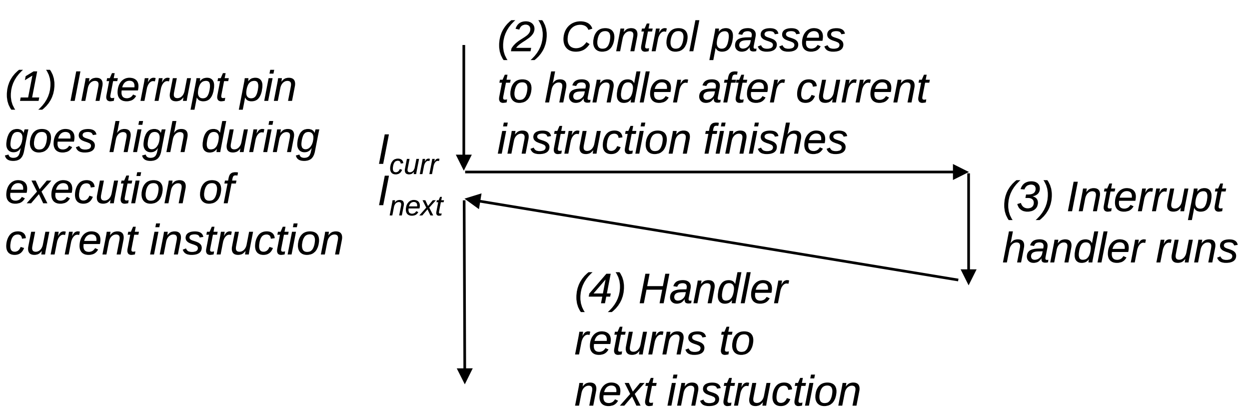
- **Interrupts**
  - Forced transfer of control to a procedure (*handler*) due to external events (*interrupt*) or due to an erroneous condition (*exception*)
- **Interrupt handling mechanism**
  - Should allow interrupts/exceptions to be handled transparently to the executing process (application programs 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).
  - 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
  - External interrupt controllers
    - Intel 8259 PIC (programmable interrupt controller) delivers the *interrupt vectors* on the system bus during interrupt acknowledge cycle
- Non-maskable interrupt (NMI)
  - Cannot be disabled by program
  - Received on the processor's NMI pin

# UNIX Process States

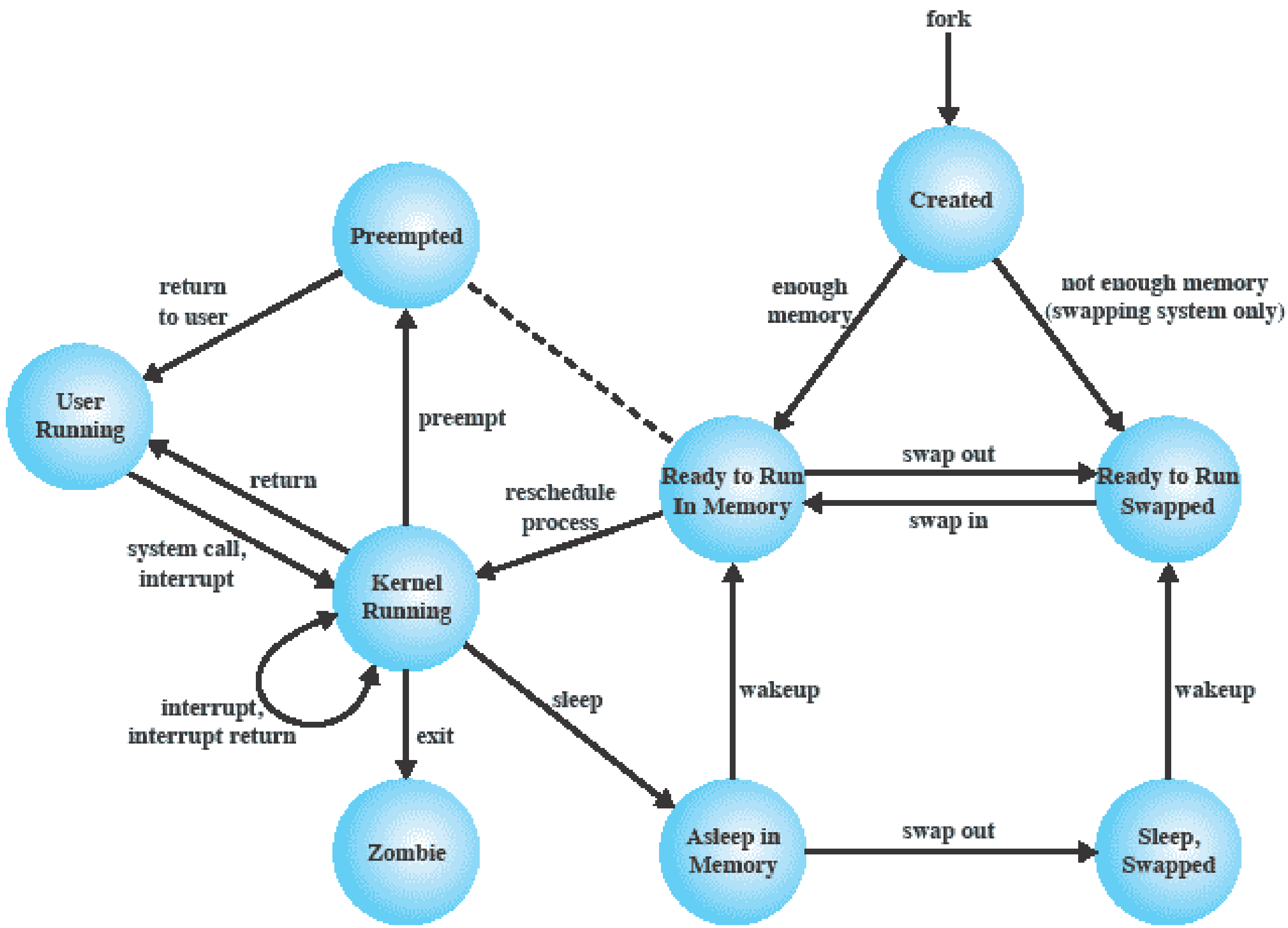



Figure 3.17 UNIX Process State Transition Diagram

*Source: Pearson*

# 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*

# Summary

- The most fundamental concept in a modern OS is the process
- The principal function of the OS is to create, manage, and terminate processes
- Process control block contains all of the information that is required for the OS to manage the process, including its current state, resources allocated to it, priority, and other relevant data
- The most important states are Ready, Running and Blocked
- The running process is the one that is currently being executed by the processor
- A blocked process is waiting for the completion of some event
- A running process is interrupted either by an interrupt or by executing a supervisor call to the OS

# Homework 2

- *3.1*
- *3.3*
- *3.5*
- *3.7*
- *3.9*
- *Read Chapter 4*