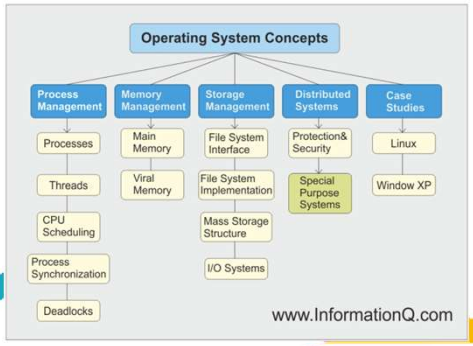
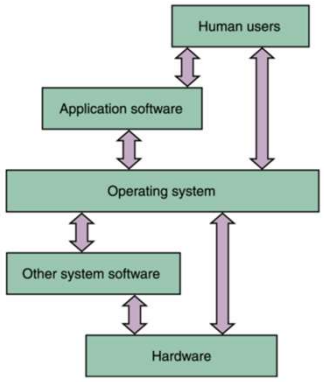
Operating Systems

"Computer System Overview"

Operating System

- Exploits the hardware resources of one or more processors
- Provides a set of services to system users
- Manages secondary memory and I/O devices





Users and Processes access the Computer's resources through the Operating System

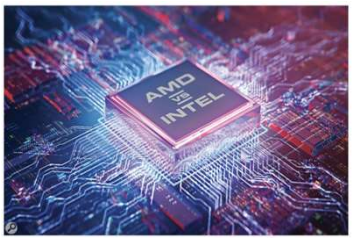
https://infonw.laurensstoop.nl/Introduction/OSrole.html

2

Basic Elements

- Processor
  - Two internal registers
    - Memory address register (MAR)
      - Specifies the address for the next read or write
    - Memory buffer register (MBR)
      - Contains data written into memory or receives data read from memory
  - I/O address register
  - I/O buffer register

Products of microelectronics and microprogramming: microprocessor, motherboard of an electronic computer (source: www.intel.com)



https://www.soundonsound.com/sound-advice/core-wars-amd-intel-cpus-tested

3

Basic Elements

- Main Memory
  - Volatile
  - Referred to as real memory or primary memory
- I/O Modules
  - Secondary Memory Devices
  - Communications equipment
  - Terminals
- System bus
  - Communication among processors, main memory, and I/O modules

4

Computer Components: Top-Level View

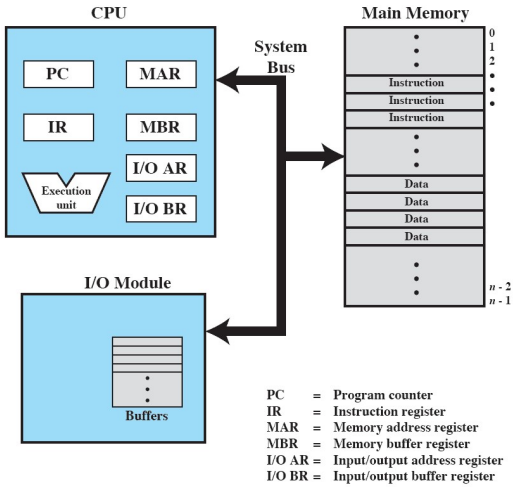


Figure 1.1 Computer Components: Top-Level View

5

Processor Registers

- User-visible registers
  - Enable programmer to minimize main memory references by optimizing register use
- Control and status registers
  - Used by processor to control operating of the processor
  - Used by privileged OS routines to control the execution of programs

6

User-Visible Registers

- May be referenced by machine language
- Available to all programs – application programs and system programs
- Data register: either general purpose or dedicated
- Address register
  - Index register: Adding an index to a base value to get the effective address
  - Segment pointer: When memory is divided into segments, memory is referenced by a segment and an offset
  - Stack pointer: Points to top of stack-push/pop (no address)

7

Control and Status Registers

- Program counter (PC)
  - Contains the address of an instruction to be fetched
- Instruction register (IR)
  - Contains the instruction most recently fetched
- Program status word (PSW)
  - Contains status information: interrupt enable/disable bit, a kernel/user mode bit, etc.
- Condition codes or flags
  - Bits set by processor hardware as a result of operations
  - Example
    - Positive, negative, zero, or overflow result

8

Instruction Execution

- Two steps
  - Processor reads (fetches) instructions from memory
  - Processor executes each instruction

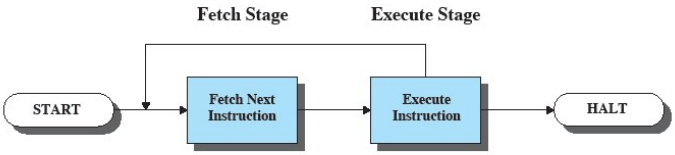


Figure 1.2 Basic Instruction Cycle

9

Instruction Fetch and Execute

- The processor fetches the instruction from memory
- Program counter (PC) holds address of the instruction to be fetched next
- PC is incremented after each fetch

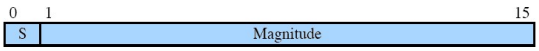
10

Instruction Register

- Fetched instruction loaded into instruction register
- Categories of the instructions
  - Processor-memory, processor-I/O, data processing, control

Characteristics of a Hypothetical Machine

(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 1.3 Characteristics of a Hypothetical Machine

Example of Program Execution

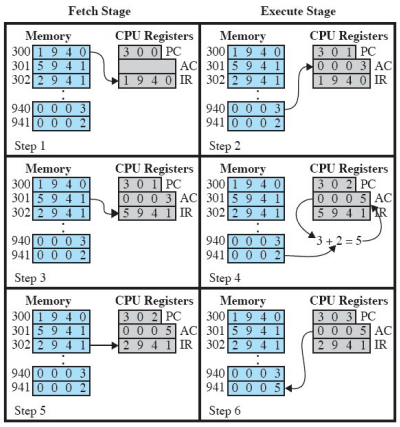


Figure 1.4 Example of Program Execution
(contents of memory and registers in hexadecimal)

13

Interrupts

- Interrupt the normal sequencing of the processor
- Most I/O devices are slower than the processor
  - Processor must pause to wait for device

14

Classes of Interrupts

Table 1.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

15

Interrupt Stage

- Processor checks for interrupts
- If interrupt
  - Suspend execution of program
  - Execute interrupt-handler routine

16

Transfer of Control via Interrupts

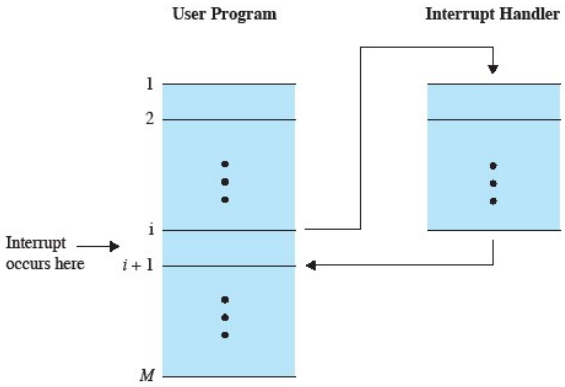
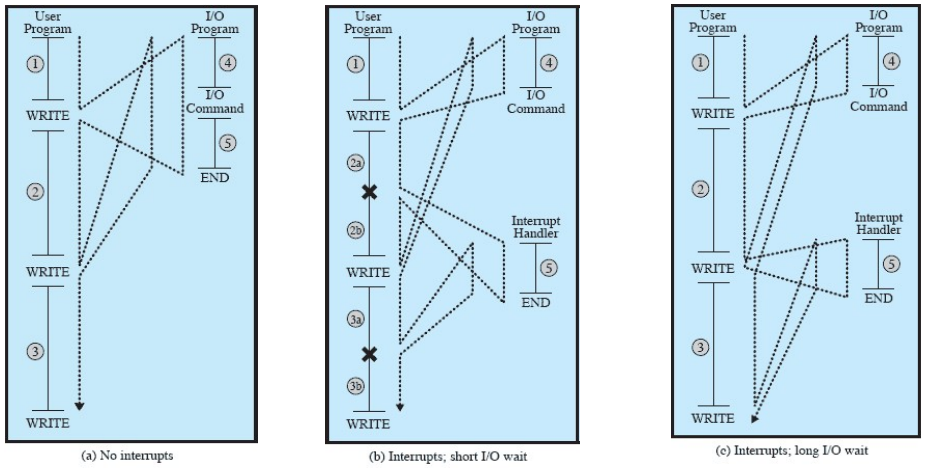


Figure 1.6 Transfer of Control via Interrupts

17

Program Flow of Control



18

Instruction Cycle with Interrupts

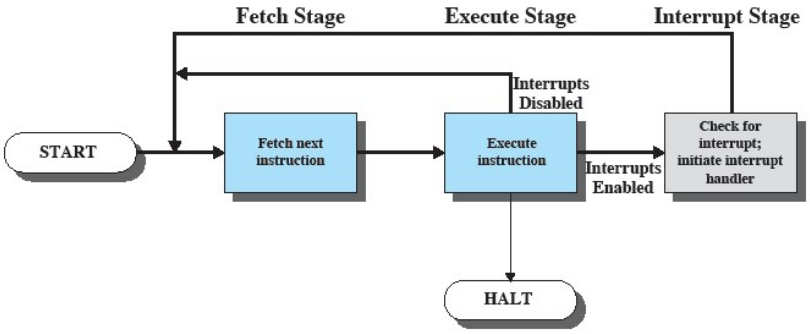


Figure 1.7 Instruction Cycle with Interrupts

Program Timing

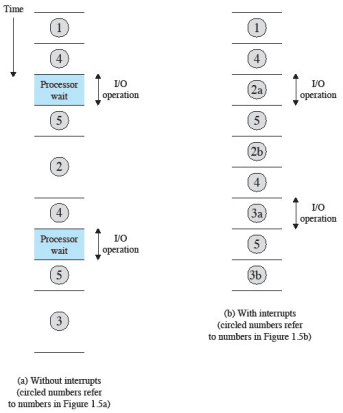


Figure 1.8 Program Timing: Short I/O Wait

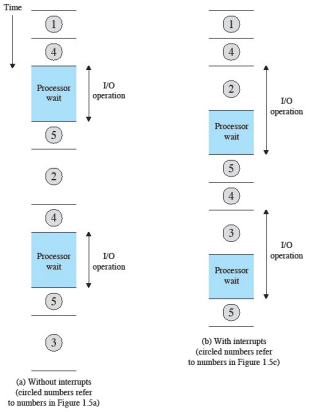


Figure 1.9 Program Timing: Long I/O Wait

Simple Interrupt Processing

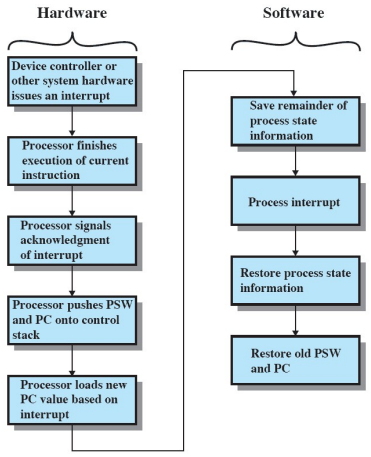
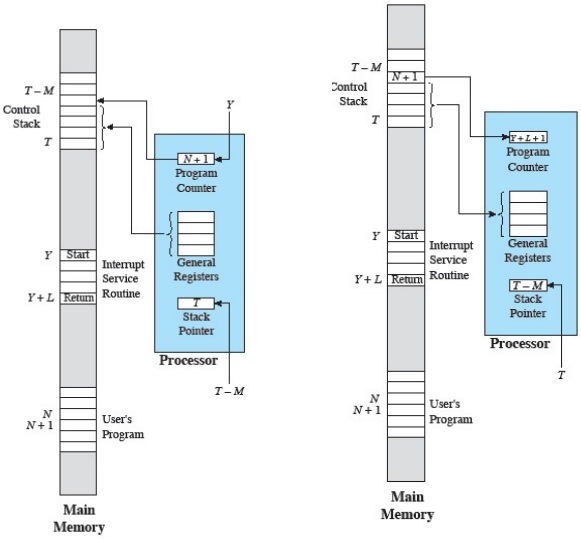


Figure 1.10 Simple Interrupt Processing

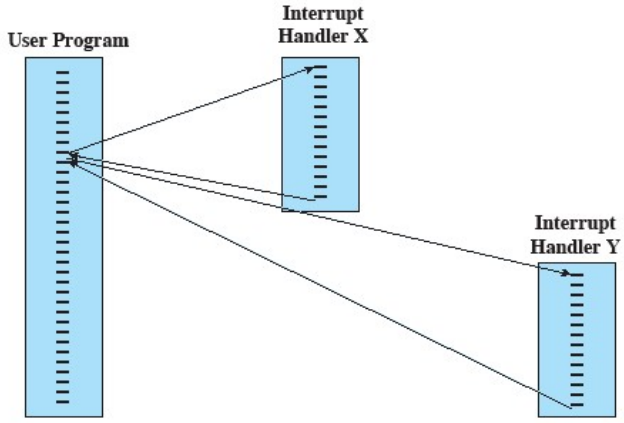
21

Changes in Memory and Registers for an Interrupt



22

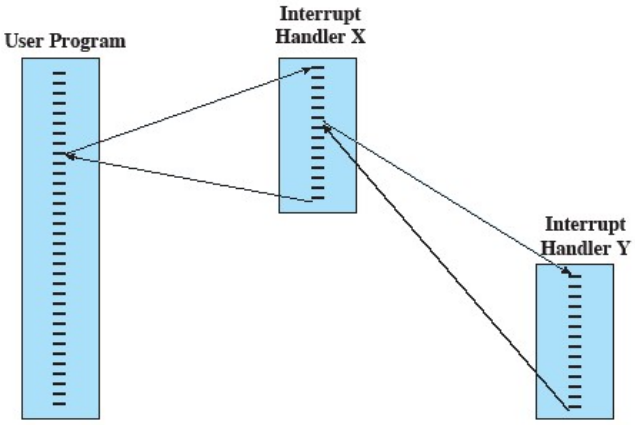
Sequential Interrupt Processing



(a) Sequential interrupt processing

23

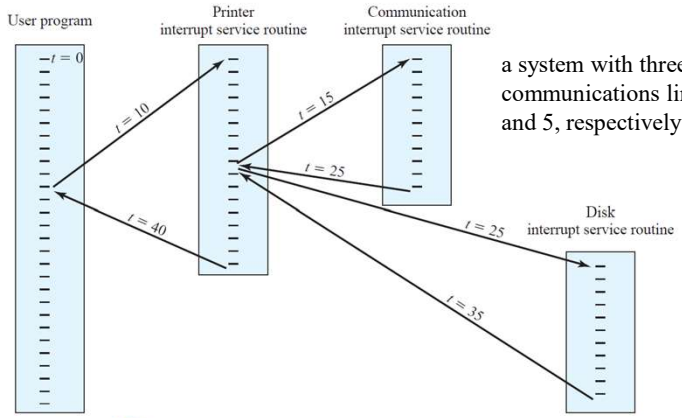
Nested Interrupt Processing



(b) Nested interrupt processing

24

Multiple interrupt



a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively

25

Multiprogramming

- Processor has more than one program to execute
- The sequence in which programs are executed depend on their relative priority and whether they are waiting for I/O
- After an interrupt handler completes, control may not return to the program that was executing at the time of the interrupt

26

Memory Hierarchy

- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access speed

→ key characteristics of memory: namely, capacity, access time, and cost

27

The Memory Hierarchy

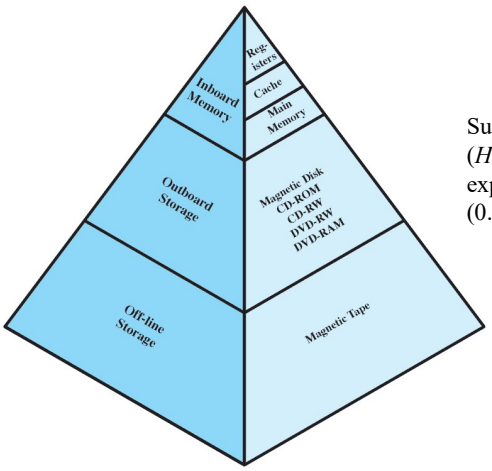


Figure 1.14 The Memory Hierarchy

Suppose 95% of the memory accesses are found in the cache (H = 0.95).Then the average time to access a byte can be expressed as
(0.95) (0.1 μs) + (0.05) (0.1 μs + 1 μs) = 0.095 + 0.055 = 0.15 μs

28

Going Down the Hierarchy

- Decreasing cost per bit
- Increasing capacity
- Increasing access time
- Decreasing frequency of access to the memory by the processor

Secondary Memory

- Auxiliary memory
- External
- Nonvolatile
- Used to store program and data files

Cache Memory

- Processor speed faster than memory access speed
- Exploit the principle of locality with a small fast memory

Cache and Main Memory

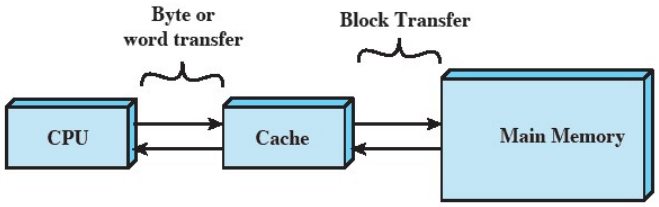
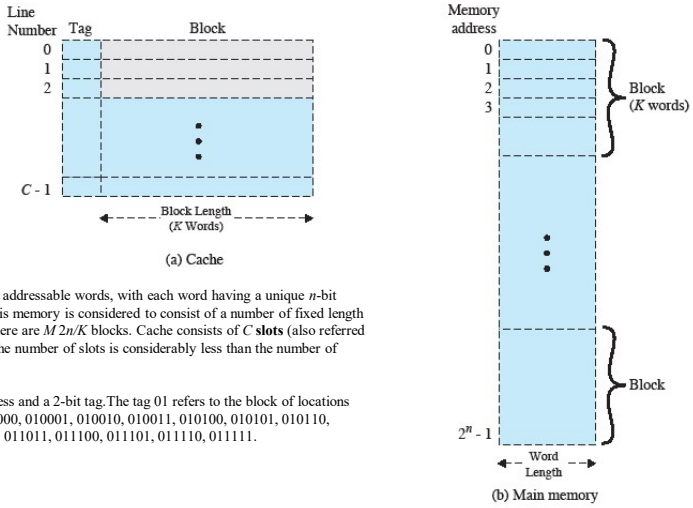


Figure 1.16 Cache and Main Memory

Cache Principles

- Contains copy of a **portion** of main memory
- Processor first checks cache
- If desired data item not found, relevant block of memory read into cache
- Because of locality of reference, it is likely that future memory references are in that block

Cache/Main-Memory Structure



Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address. For mapping purposes, this memory is considered to consist of a number of fixed length **blocks** of K words each. That is, there are $M = 2^n/K$ blocks. Cache consists of C **slots** (also referred to as *lines*) of K words each, and the number of slots is considerably less than the number of main memory blocks ($C << M$).

Suppose that we have a 6-bit address and a 2-bit tag. The tag 01 refers to the block of locations with the following addresses: 010000, 010001, 010010, 010011, 010100, 010101, 010110, 010111, 011000, 011001, 011010, 011011, 011100, 011101, 011110, 011111.

Figure 1.17 Cache/Main-Memory Structure

Cache Read Operation

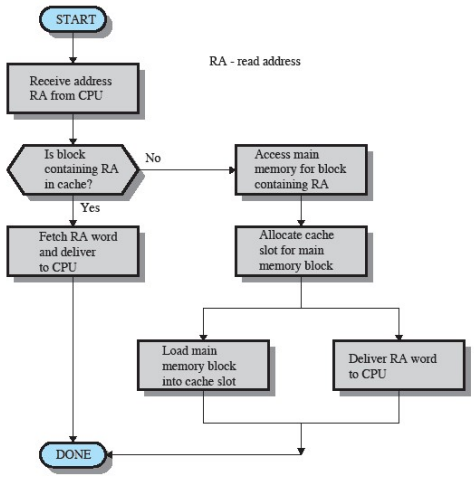


Figure 1.18 Cache Read Operation

35

Cache Principles

- Cache size
  - Even small caches have significant impact on performance
- Block size
  - The unit of data exchanged between cache and main memory
  - Larger block size yields more hits until probability of using newly fetched data becomes less than the probability of reusing data that have to be moved out of cache

36

Cache Principles

- Mapping function
  - Determines which cache location the block will occupy
  - Two constraints
    - when one block is read in, another may have to be replaced (which block will be used in the near future ?)
    - the more flexible the mapping function, the more complex is the circuitry required to search the cache to determine if a given block is in the cache
- Replacement algorithm
  - Chooses which block to replace
    - Least-recently-used (LRU) algorithm
    - First in First Out (FIFO) algorithm
    - Least Frequently Used (LFU) algorithm

37

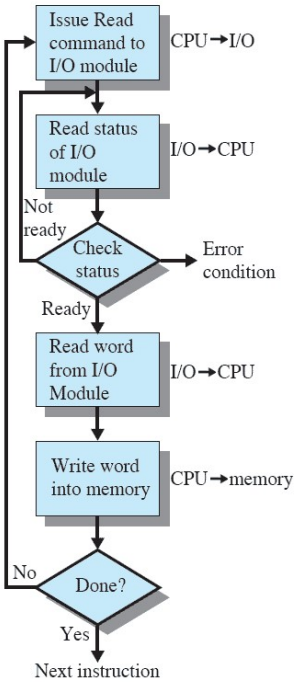
Cache Principles

- Write policy
  - Dictates when the memory write operation takes place
    - Can occur every time the block is updated
    - Can occur when the block is replaced
      - Minimize write operations
      - Leave main memory which are in an obsolete state

38

I/O COMMUNICATION TECHNIQUES
Programmed I/O

- I/O module performs the action, not the processor
- Sets the appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete
- Main disadvantage: a time-consuming process that keeps the processor busy needlessly
- I/O instructions
  - Control
  - Status
  - transfer

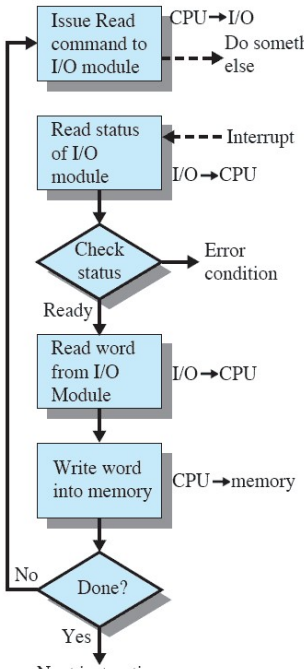


(a) Programmed I/O

39

I/O COMMUNICATION TECHNIQUES
Interrupt-Driven I/O

- Processor is interrupted when I/O module ready to exchange data
- Processor saves context of program executing and begins executing interrupt-handler

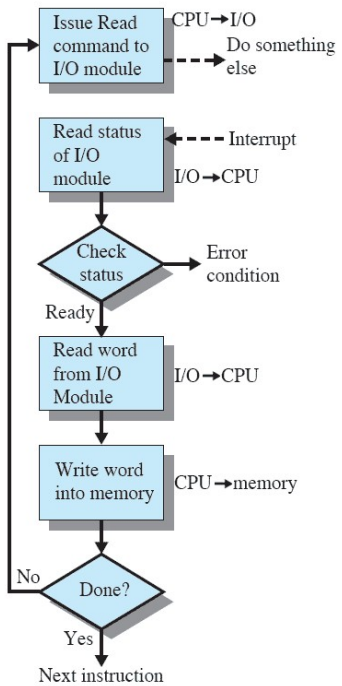


(b) Interrupt-driven I/O

40

I/O COMMUNICATION TECHNIQUES
Interrupt-Driven I/O

- No needless waiting, more efficient than programmed I/O
- Consumes a lot of processor time because every word read or written passes through the processor

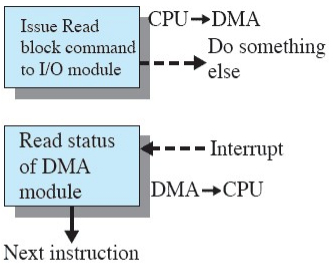


(b) Interrupt-driven I/O

41

I/O COMMUNICATION TECHNIQUES
Direct Memory Access

- The processor continues with other work. It has delegated this I/O operation to the DMA module
- Transfers a block of data directly to or from memory
- An interrupt is sent when the transfer is complete
- More efficient



(c) Direct memory access

42

Thank You

Operating Systems

“Process Description and Control”

Roadmap

- How are processes represented and controlled by the OS.
- **Process states** which characterize the behaviour of processes.
- **Data structures** used to manage processes.
- Ways in which the OS uses these data structures to control process execution.

Operating System

- How are processes represented and controlled by the OS.
- Process states which characterize the behaviour of processes.
- Data structures used to manage processes.
- Ways in which the OS uses these data structures to control process execution.

Requirements of an Operating System

- *Fundamental Task: Process Management*
- 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.

The OS Manages Execution of Applications

- Resources are made available to multiple applications
- The processor is switched among multiple application
- The processor and I/O devices can be used efficiently

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

Process Elements

- A process is comprised of:
  - Program code (possibly shared)
  - A set of data
- A number of elements including
  - Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers
  - Context data
  - I/O status information
  - Accounting information

Process Control Block

- Contains the process elements
- Created and manage by the operating system
- Allows support for multiple processes

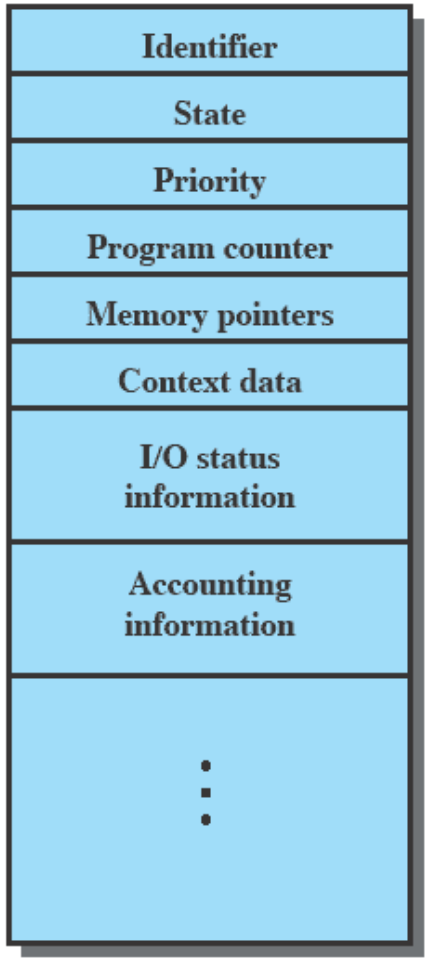
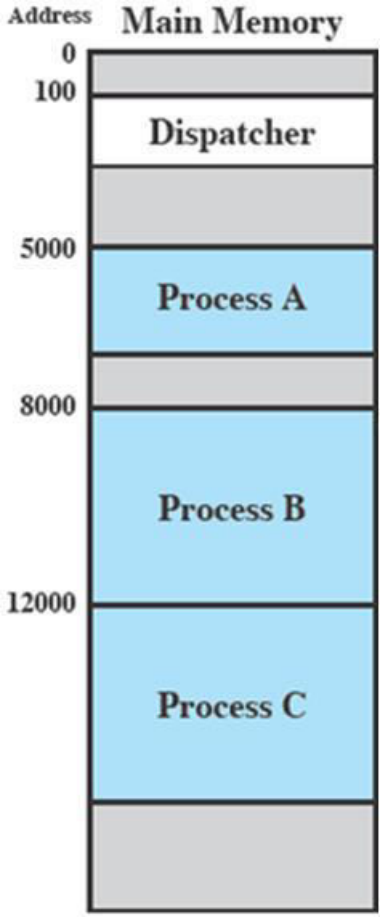


Figure 3.1 Simplified Process Control Block

Trace of the Process

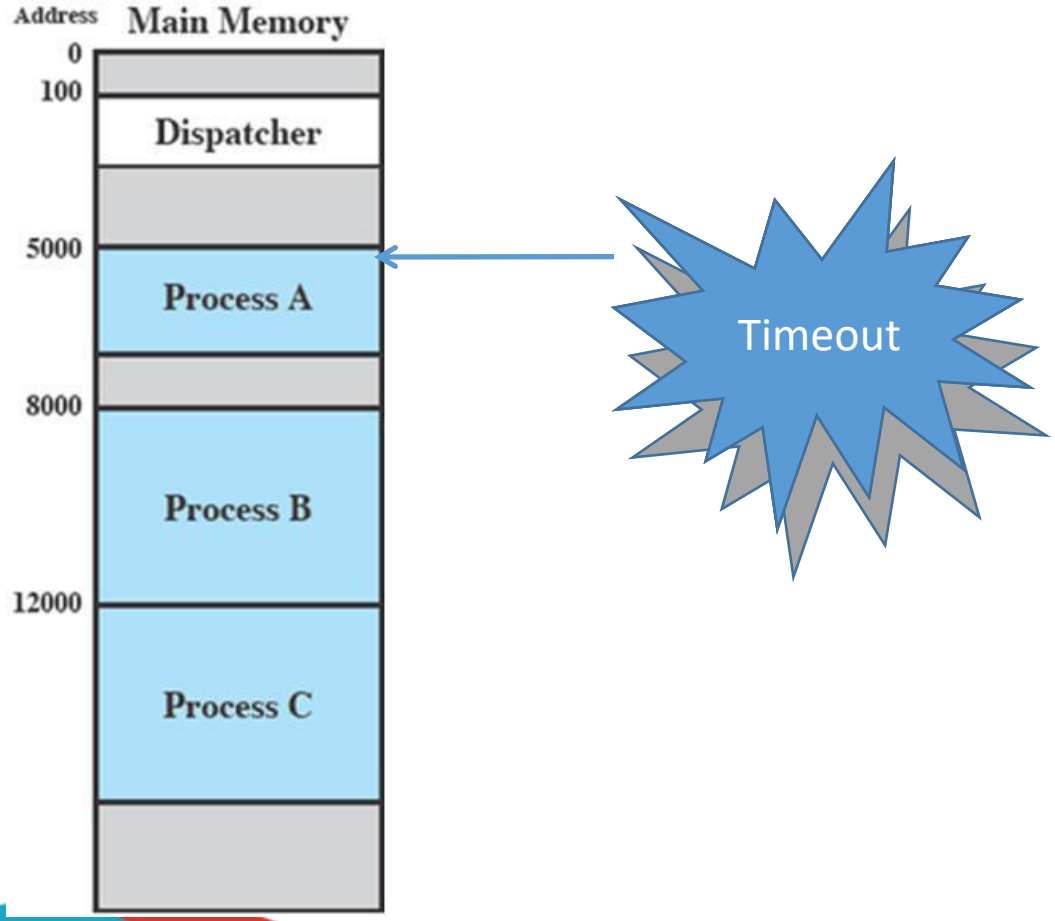
- The behavior of an individual process is shown by listing the sequence of instructions that are executed
- This list is called a ***Trace***
- ***Dispatcher*** is a small program which switches the processor from one process to another

Process Execution



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

Trace from Processors point of view



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

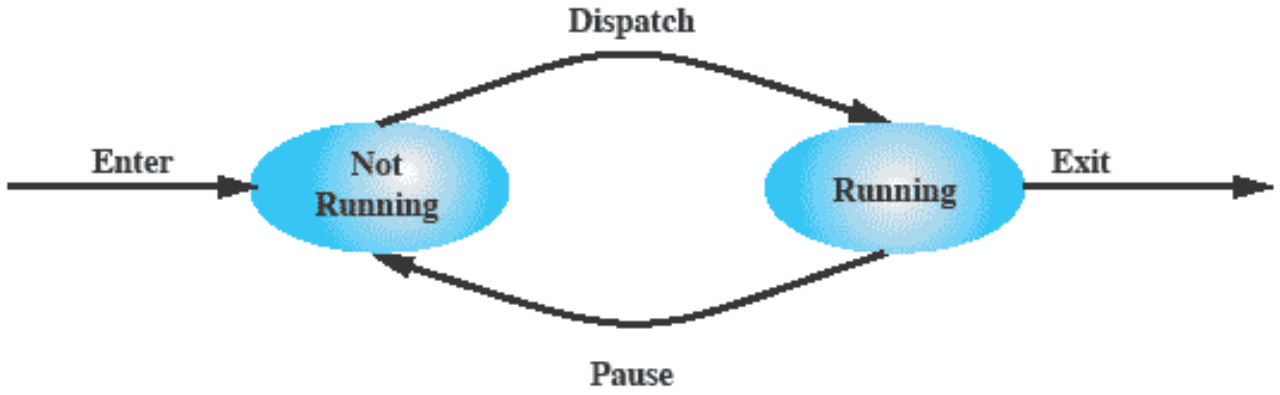
Figure 3.4 Combined Trace of Processes of Figure 3.2

Roadmap

- How are processes represented and controlled by the OS.
- ***Process states*** which characterize the behaviour of processes.
- ***Data structures*** used to manage processes.
- Ways in which the OS uses these data structures to control process execution.
- Discuss process management in UNIX SVR4.

Two-State Process Model

- Process may be in one of two states
  - Running
  - Not-running



(a) State transition diagram

Process creation and termination

Creation	Termination
New batch job	Normal Completion
Interactive Login	Memory unavailable
Created by OS to provide a service	Protection error
Spawned by existing process	Operator or OS Intervention

See tables 3.1 and 3.2 for more

Process Creation

- The OS builds a data structure to manage the process
- Traditionally, the OS created all processes
  - But it can be useful to let a running process create another
- This action is called ***process spawning***
  - ***Parent Process*** is the original, creating, process
  - ***Child Process*** is the new process

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

Five-State Process Model

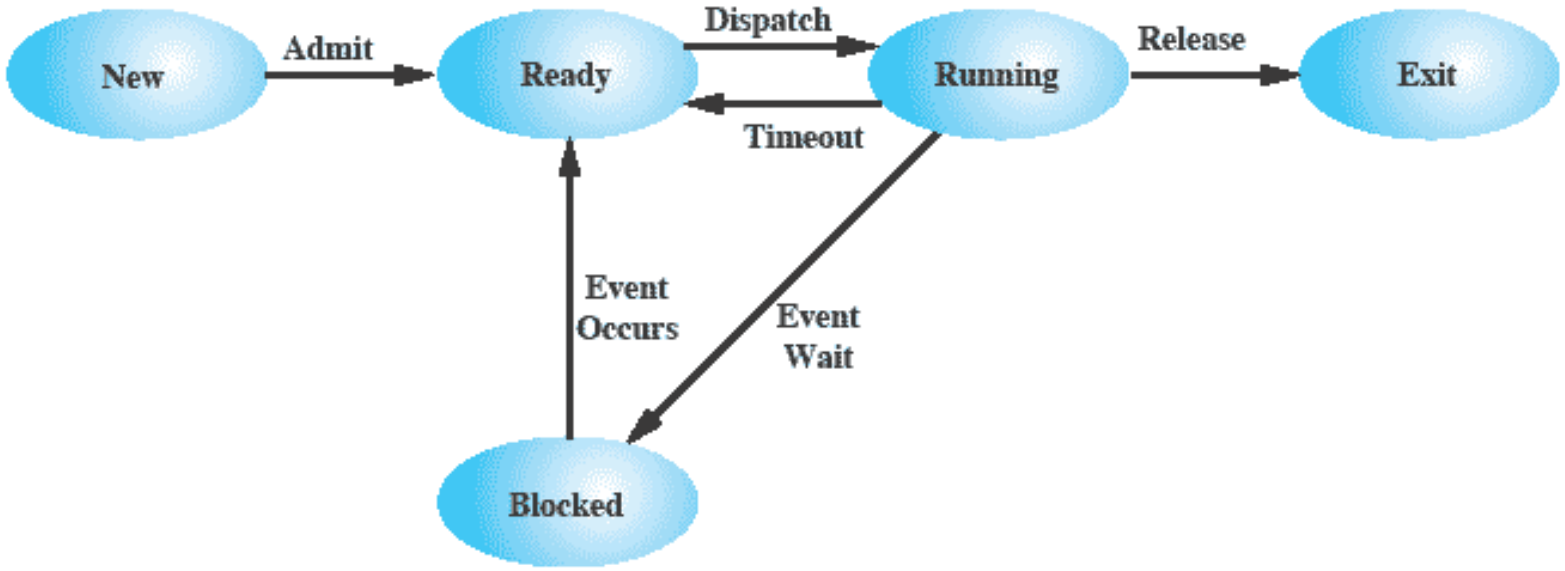
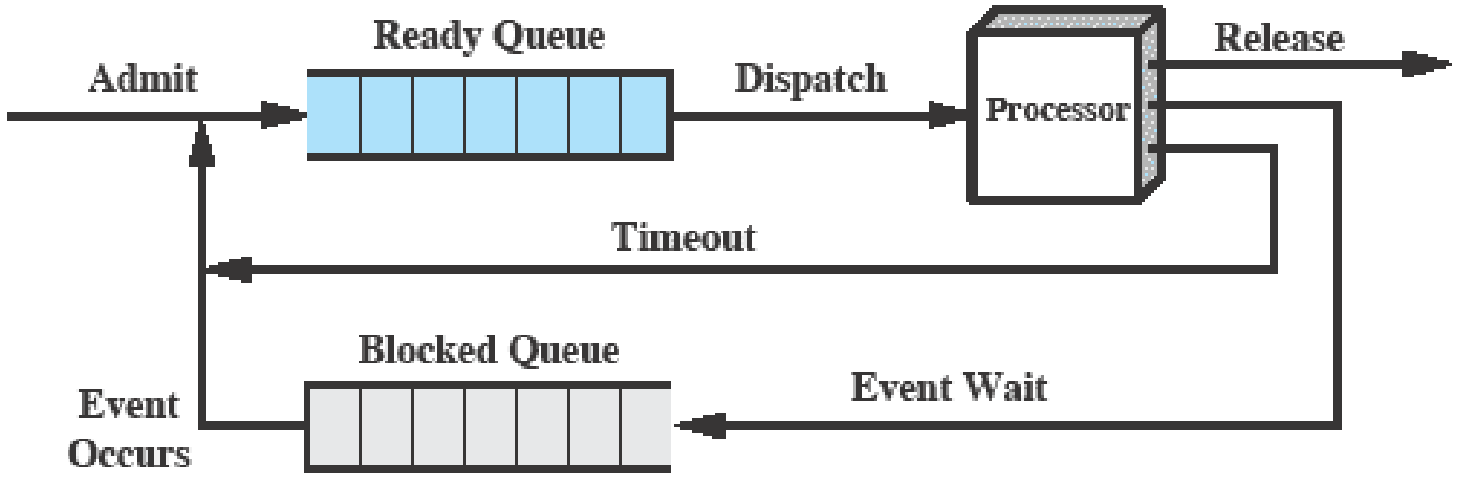


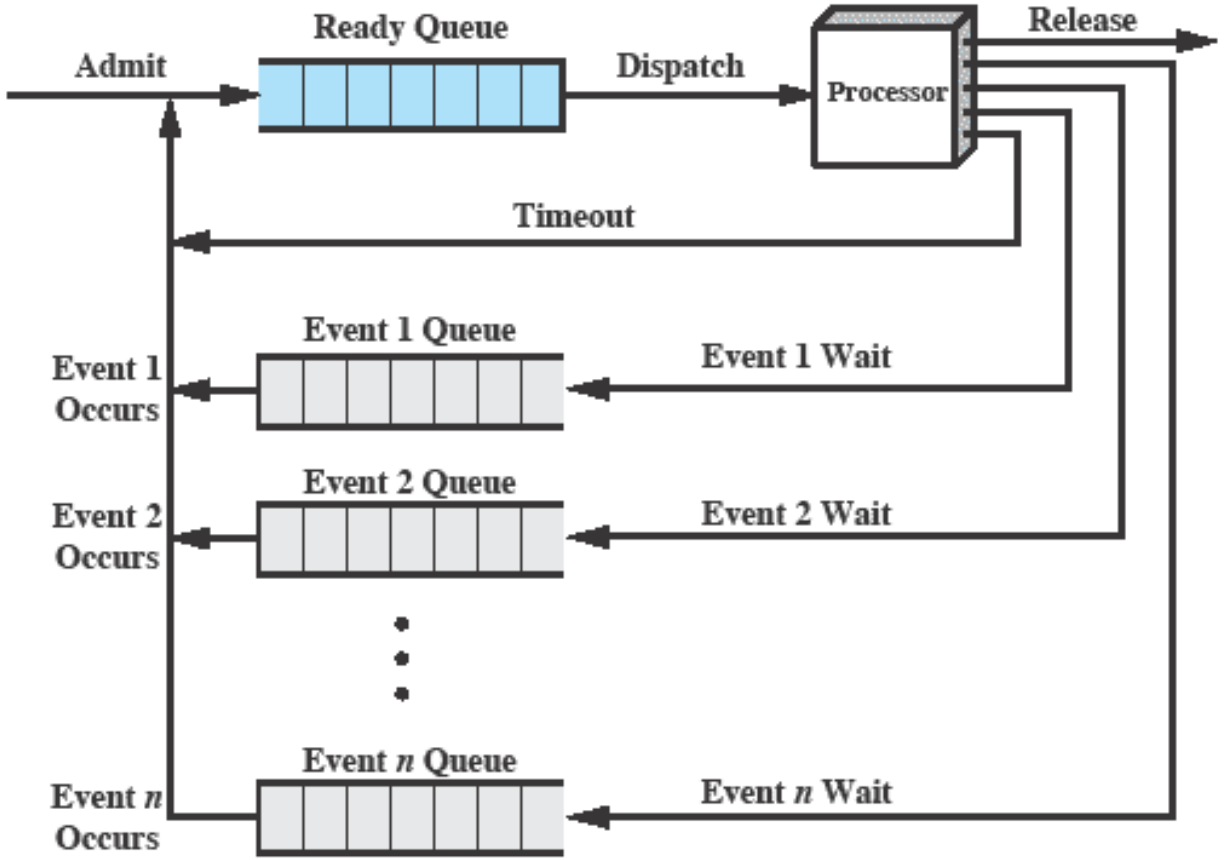
Figure 3.6 Five-State Process Model

Using Two Queues



(a) Single blocked queue

Multiple Blocked Queues

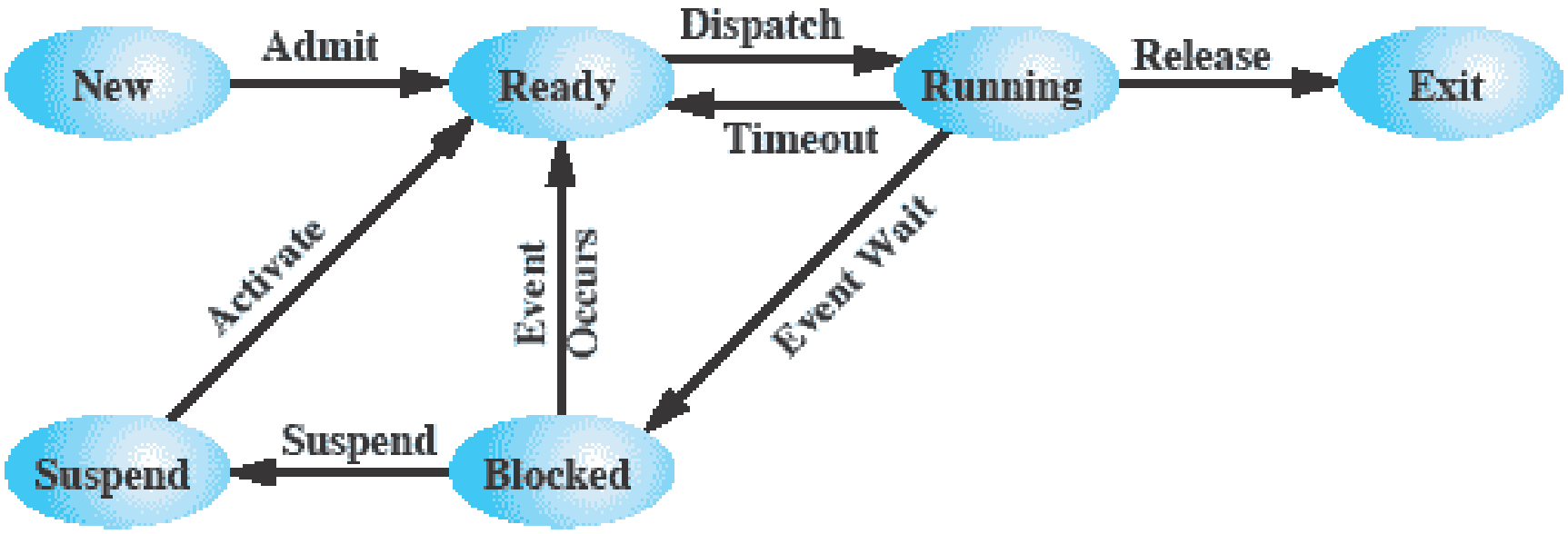


(b) Multiple blocked queues

Suspended Processes

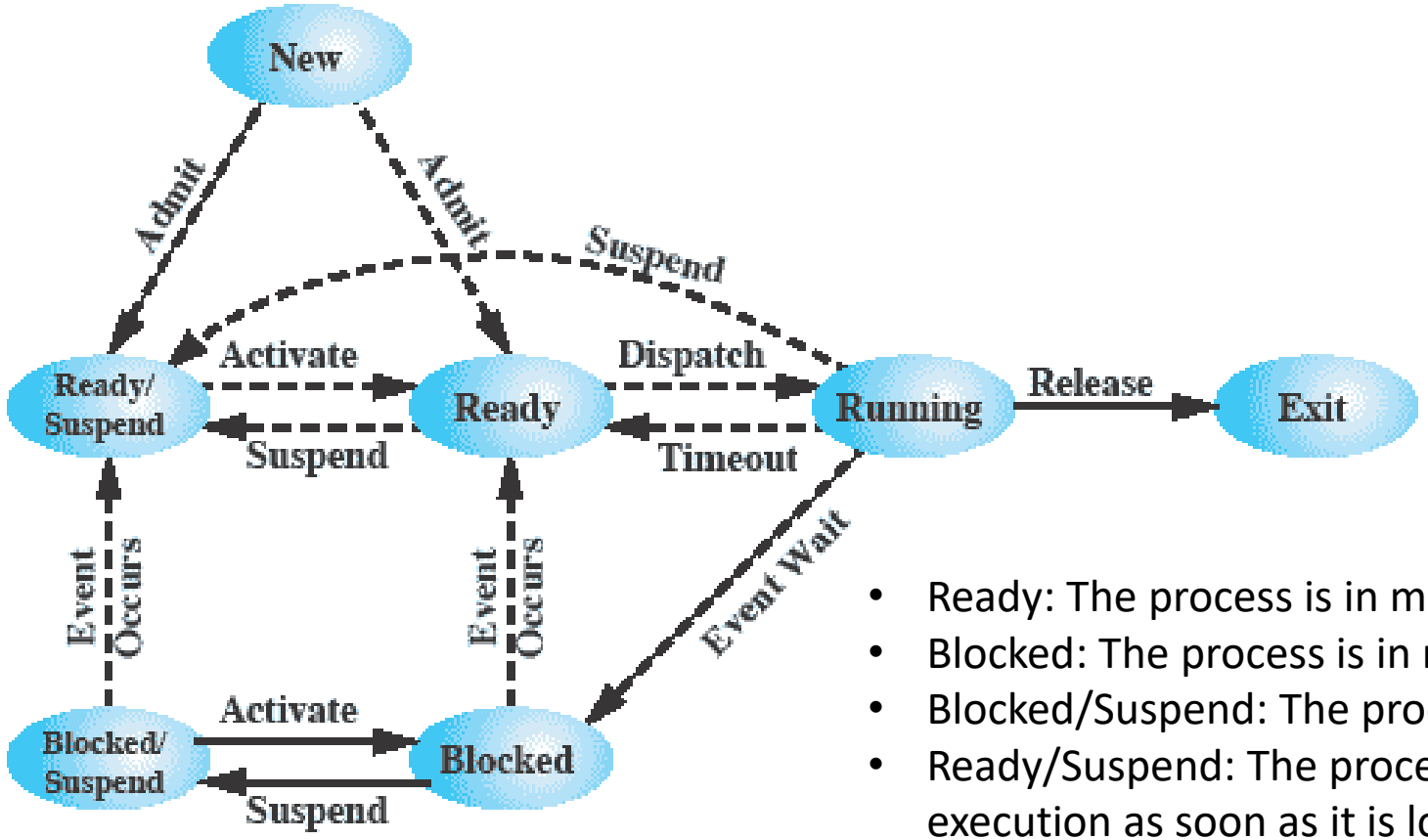
- Processor is faster than I/O so all processes could be waiting for I/O
  - Swap these processes to disk to free up more memory and use processor on more / other processes
- Blocked state becomes ***suspend*** state when swapped to disk (secondary storage)
- Two new states
  - Blocked/Suspend … from memory → disc
  - Ready/Suspend … from disc → memory

One Suspend State



(a) With One Suspend State

Two Suspend States



(b) With Two Suspend 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.

Reason for Process Suspension

Reason	Comment
Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS Reason	OS suspects process of causing a problem.
Interactive User Request	e.g. debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time.
Parent Process Request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Table 3.3 Reasons for Process Suspension

Roadmap

- How are processes represented and controlled by the OS.
- ***Process states*** which characterize the behaviour of processes.
- ***Data structures*** used to manage processes.
- Ways in which the OS uses these data structures to control process execution.

Processes and Resources

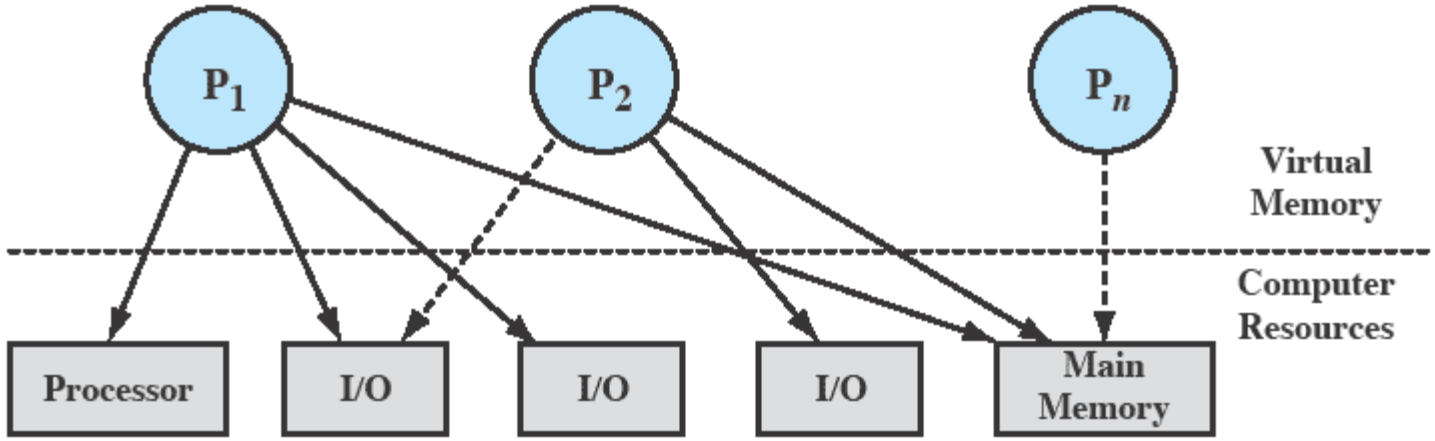


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

Operating System Control Structures

- For the OS is to manage processes and resources, it must have information about the current status of each process and resource.
- Tables are constructed for each entity the operating system manages

OS Control Tables

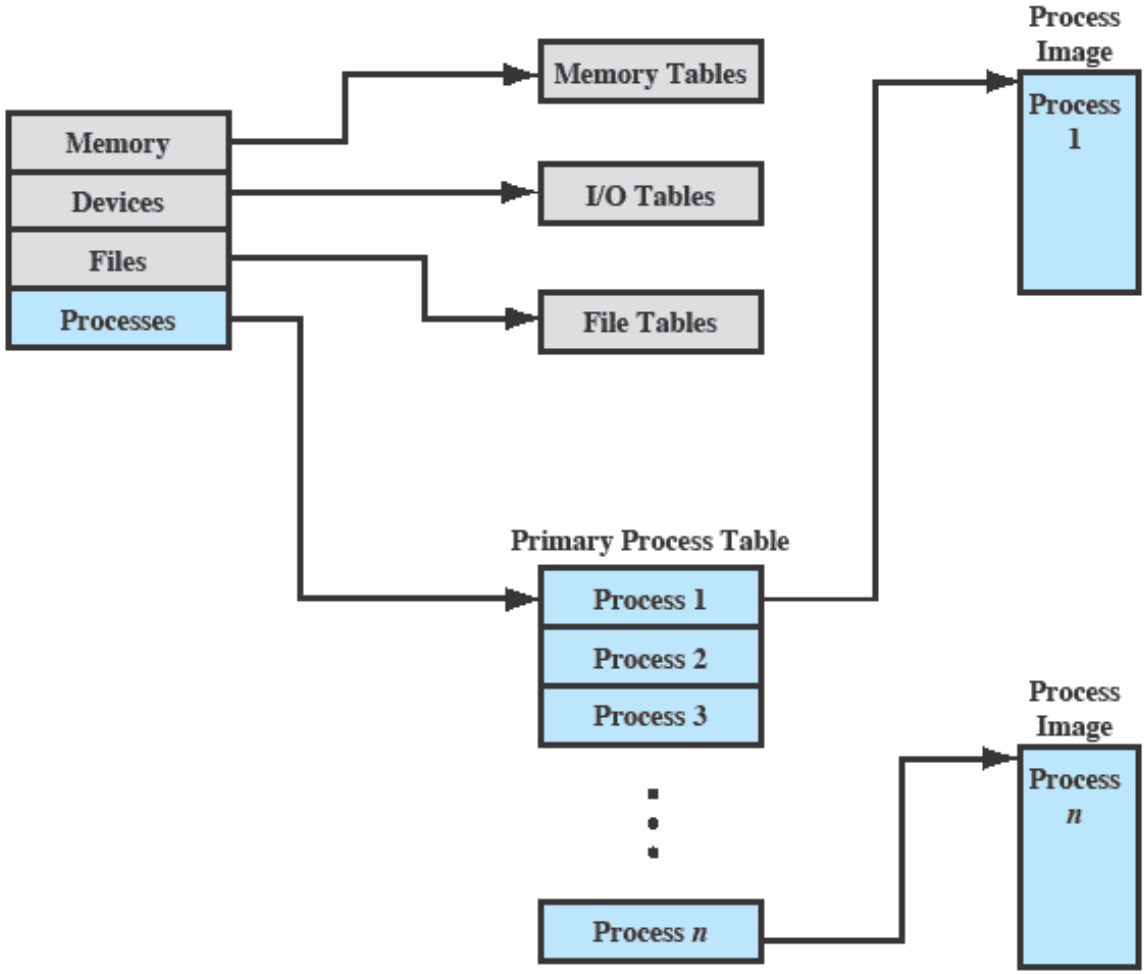


Figure 3.11 General Structure of Operating System Control Tables

Memory Tables

- Memory tables are used to keep track of both main and secondary memory.
- Must include this information:
  - Allocation of main memory to processes
  - Allocation of secondary memory to processes
  - Protection attributes for access to shared memory regions
  - Information needed to manage virtual memory

I/O Tables

- Used by the OS to manage the I/O devices and channels of the computer.
- The OS needs to know
  - Whether the I/O device is available or assigned
  - The status of I/O operation
  - The location in main memory being used as the source or destination of the I/O transfer

File Tables

- These tables provide information about:
  - Existence of files
  - Location on secondary memory
  - Current Status
  - other attributes.
- Sometimes this information is maintained by a file management system

Process Tables

- To manage processes the OS needs to know details of the processes
  - Current state
  - Process ID
  - Location in memory
  - etc
- Process control block
  - ***Process image*** is the collection of program. Data, stack, and attributes

Process Attributes

- We can group the process control block information into three general categories:
  - Process identification
  - Processor state information
  - Process control information

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.

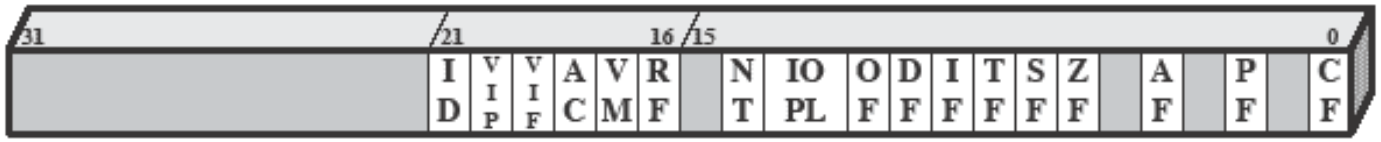
Process Identification

- Each process is assigned a unique numeric identifier.
- Many of the other tables controlled by the OS may use process identifiers to cross-reference process tables

Processor State Information

- This consists of the contents of processor registers.
  - User-visible registers
  - Control and status registers
  - Stack pointers
- Program status word (PSW)
  - contains status information
  - Example: the EFLAGS register on Pentium processors

Pentium II EFLAGS Register



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

Also see Table 3.6

Figure 3.12 Pentium II EFLAGS Register

Process Control Information

- This is the additional information needed by the OS to control and coordinate the various active processes.
  - See table 3.5 for scope of information

Structure of Process Images in Virtual Memory

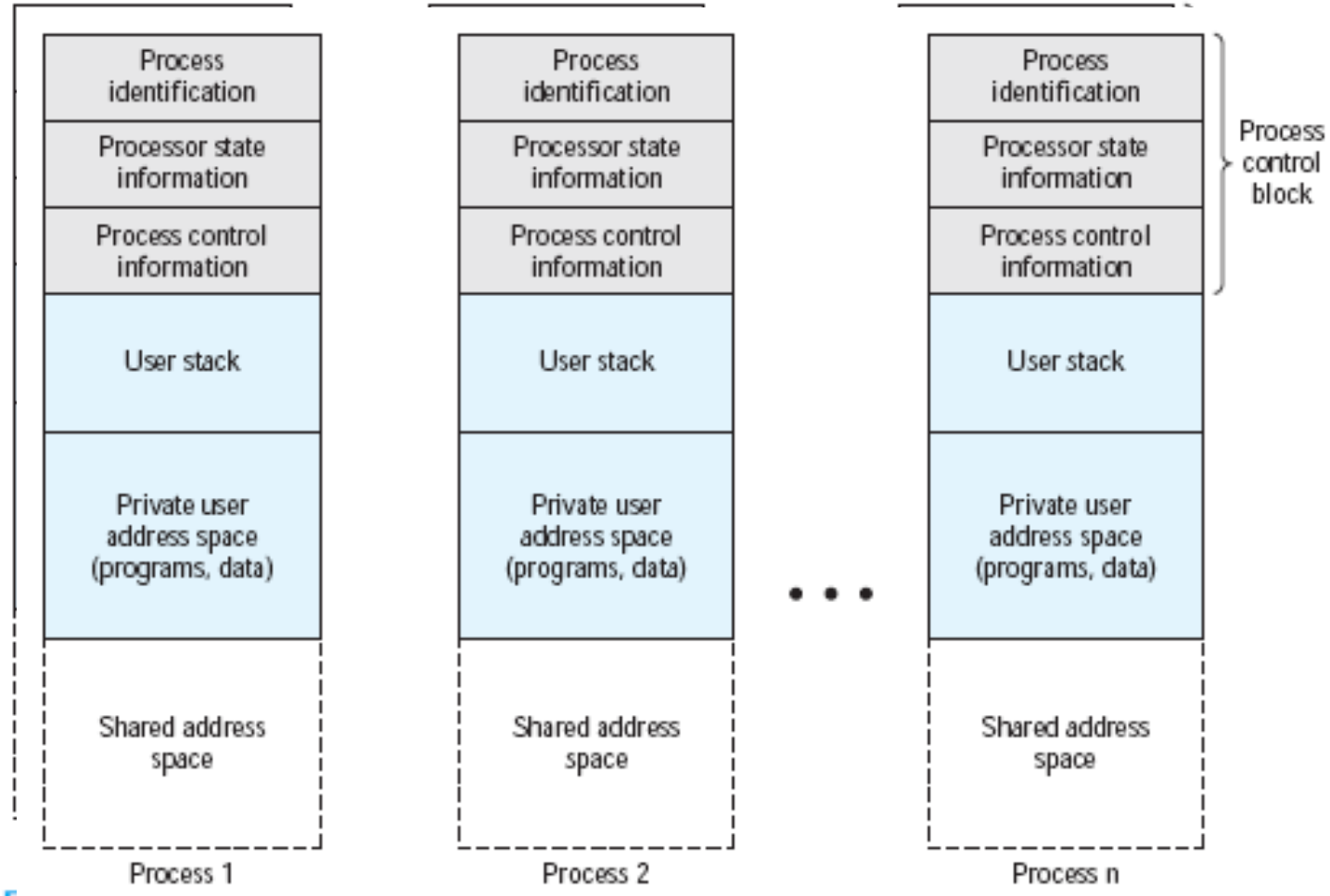


Figure 3.13 User Processes in Virtual Memory

Role of the Process Control Block

- The most important data structure in an OS
  - It defines the state of the OS
- Process Control Block requires protection
  - A faulty routine could cause damage to the block destroying the OS's ability to manage the process
  - Any design change to the block could affect many modules of the OS

Roadmap

- How are processes represented and controlled by the OS.
- ***Process states*** which characterize the behaviour of processes.
- ***Data structures*** used to manage processes.
- Ways in which the OS uses these data structures to control process execution.

Modes of Execution

- Most processors support at least two modes of execution
- User mode
  - Less-privileged mode
  - User programs typically execute in this mode
- System mode
  - More-privileged mode
  - Kernel of the operating system

Typical Functions of an Operating System Kernel

Process Management

- Process creation and termination
- Process scheduling and dispatching
- Process switching
- Process synchronization and support for interprocess communication
- Management of process control blocks

Memory Management

- Allocation of address space to processes
- Swapping
- Page and segment management

I/O Management

- Buffer management
- Allocation of I/O channels and devices to processes

Support Functions

- Interrupt handling
- Accounting
- Monitoring

Process Creation

- Once the OS decides to create a new process it:
  - Assigns a unique process identifier
  - Allocates space for the process
  - Initializes process control block
  - Sets up appropriate linkages
  - Creates or expand other data structures

Switching Processes

- Several design issues are raised regarding process switching
  - What events trigger a process switch?
  - We must distinguish between mode switching and process switching.
  - What must the OS do to the various data structures under its control to achieve a process switch?

When to switch processes

A process switch may occur any time that the OS has gained control from the currently running process. Possible events giving OS control are:

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

Table 3.8 Mechanisms for Interrupting the Execution of a Process

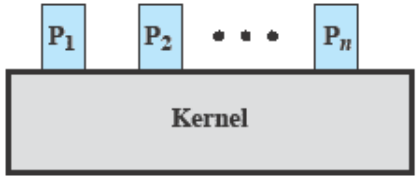
Change of Process State …

- The steps in a process switch are:
  1. Save context of processor including program counter and other registers
  2. Update the process control block of the process that is currently in the Running state
  3. Move process control block to appropriate queue – ready; blocked; ready/suspend
  4. Select another process for execution
  5. Update the process control block of the process selected
  6. Update memory-management data structures
  7. Restore context of the selected process

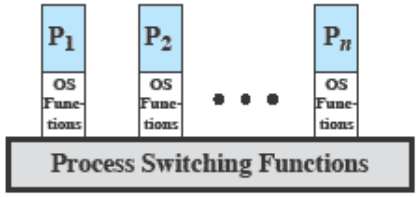
Is the OS a Process?

- If the OS is just a collection of programs and if it is executed by the processor just like any other program, is the OS a process?
- If so, how is it controlled?
  - Who (what) controls it?

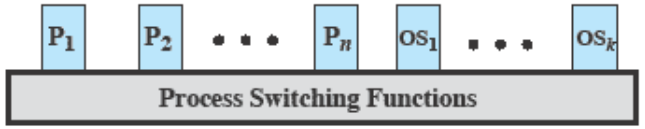
Execution of the Operating System



(a) Separate kernel



(b) OS functions execute within user processes

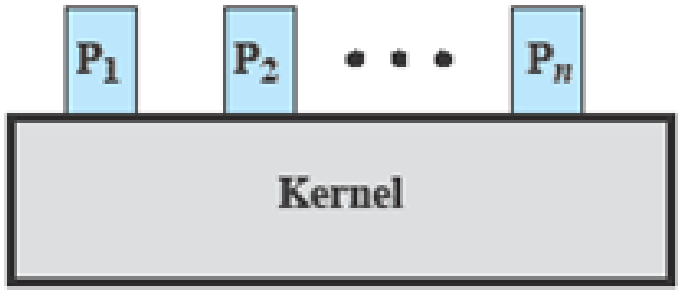


(c) OS functions execute as separate processes

Figure 3.15 Relationship Between Operating System and User Processes

Non-process Kernel

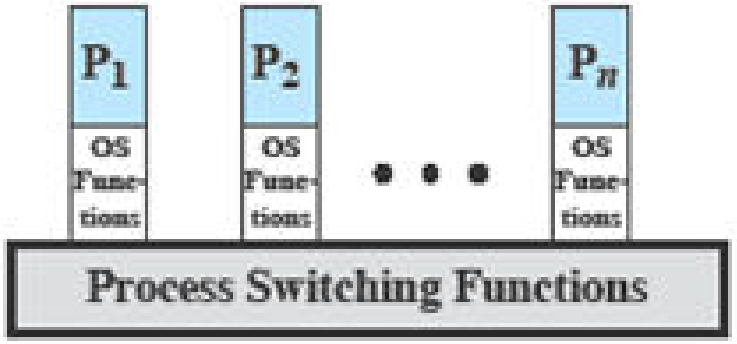
- Execute kernel outside of any process
- The concept of process is considered to apply only to user programs
  - Operating system code is executed as a separate entity that operates in privileged mode



(a) Separate kernel

Execution *Within* User Processes

- Execution Within User Processes
  - Operating system software within context of a user process
  - No need for Process Switch to run OS routine



(b) OS functions execute within user processes

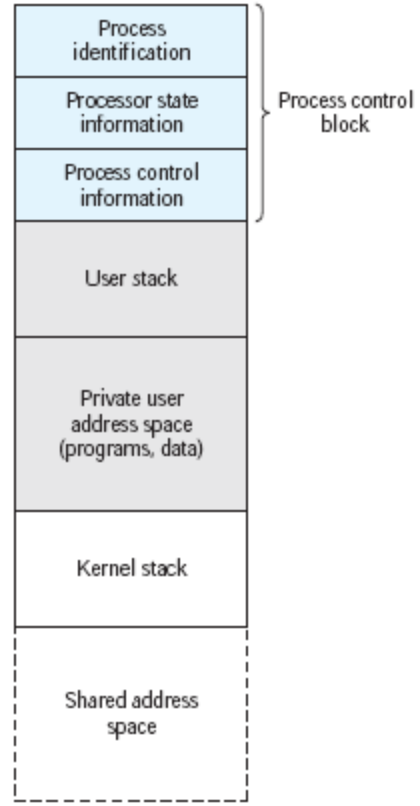


Figure 3.16 Process Image: Operating System Executes within User Space

Process-based Operating System

- Process-based operating system
  - Implement the OS as a collection of system process

(c) OS functions execute as separate processes

Security Issues

- An OS associates a set of privileges with each process.
  - Highest level being administrator, supervisor, or root, access.
- A key security issue in the design of any OS is to prevent anything (user or process) from gaining unauthorized privileges on the system
  - Especially - from gaining root access.

System access threats

- Intruders
  - Masquerader (outsider)
  - Misfeasor (insider)
  - Clandestine user (outside or insider)
- Malicious software (malware)

Countermeasures: Intrusion Detection

- Intrusion detection systems are typically designed to detect human intruder and malicious software behaviour.
- May be host or network based
- Intrusion detection systems (IDS) typically comprise
  - Sensors
  - Analyzers
  - User Interface

Countermeasures: Authentication

- Two Stages:
  - Identification
  - Verification
- Four Factors:
  - Something the individual ***knows***
  - Something the individual ***possesses***
  - Something the individual ***is*** (static biometrics)
  - Something the individual ***does*** (dynamic biometrics)

Countermeasures: Access Control

- A policy governing access to resources
- A security administrator maintains an authorization database
  - The access control function consults this to determine whether to grant access.
- An auditing function monitors and keeps a record of user accesses to system resources.

Countermeasures: Firewalls

- Traditionally, a firewall is a dedicated computer that:
  - interfaces with computers outside a network
  - has special security precautions built into it to protect sensitive files on computers within the network.

Thank You

Operating Systems

"Interprocess Communication (IPC)"

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

Multiple Processes

- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

Concurrency

Concurrency arises in:

- Multiple applications
  - Sharing time
- Structured applications
  - Extension of modular design
- Operating system structure
  - OS themselves implemented as a set of processes or threads

Key Terms

Table 5.1 Some Key Terms Related to Concurrency

atomic operation	A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
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.

Interleaving and Overlapping Processes

- Earlier (Ch2) we saw that processes may be interleaved on uniprocessors

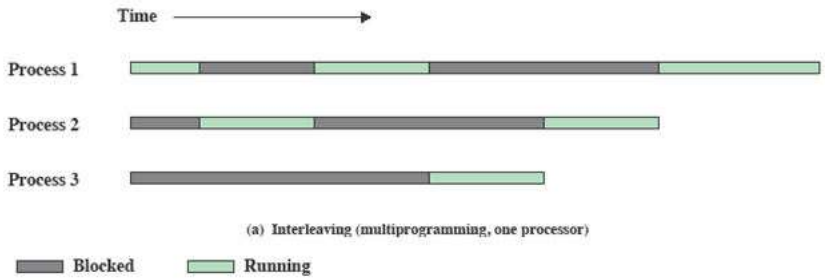


Figure 2.12 Multiprogramming and Multiprocessing

Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors

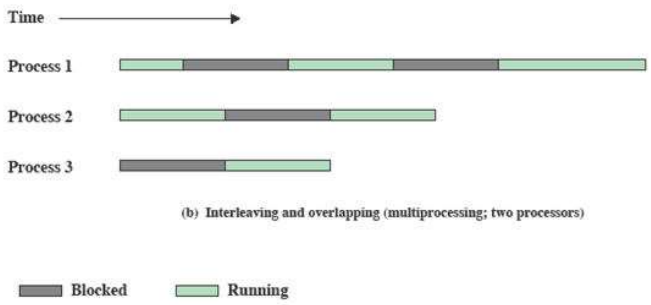


Figure 2.12 Multiprogramming and Multiprocessing

Difficulties of Concurrency

- Sharing of global resources
- Optimally managing the allocation of resources
- Difficult to locate programming errors as results are not deterministic and reproducible.

A Simple Example

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

A Simple Example: On a Multiprocessor

Process P1	Process P2
.	.
chin = getchar();	.
.	chin = getchar();
chout = chin;	chout = chin;
putchar(chout);	.
.	putchar(chout);
.	.

Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then:
- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.

Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed

Process Interaction

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence That One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Competition among Processes for Resources

Three main control problems:

- Need for Mutual Exclusion
  - Critical sections
- Deadlock
- Starvation

Requirements for Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation

Requirements for Mutual Exclusion

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

Disabling Interrupts

- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture

Pseudo-Code

```
while (true) {
   /* disable interrupts */;
   /* critical section */;
   /* enable interrupts */;
   /* remainder */;
}
```

Special Machine Instructions

- Compare&Swap Instruction
  - also called a “compare and exchange instruction”
- Exchange Instruction

Compare&Swap Instruction

```
int compare_and_swap (int *word,
 int testval, int newval)
{
   int oldval;
   oldval = *word;
   if (oldval == testval) *word = newval;
   return oldval;
}
```

Mutual Exclusion (fig 5.2)

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

(a) Compare and swap instruction

Exchange instruction

```
void exchange (int register, int memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Exchange Instruction
(fig 5.2)

```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

Semaphore

- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - initialize,
  - Decrement (`semWait`)
  - increment. (`semSignal`)

Semaphore Primitives

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

Binary Semaphore Primitives

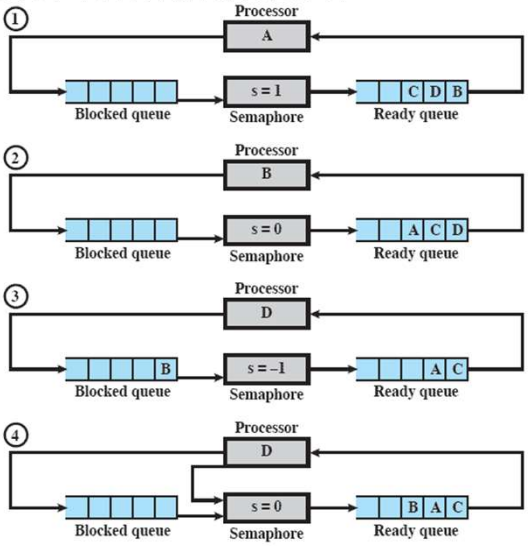
```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
            /* place this process in s.queue */;
            /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Strong/Weak Semaphore

- A queue is used to hold processes waiting on the semaphore
  - In what order are processes removed from the queue?
- ***Strong Semaphores*** use FIFO
- ***Weak Semaphores*** don't specify the order of removal from the queue

Example of Strong Semaphore Mechanism



Example of Semaphore Mechanism

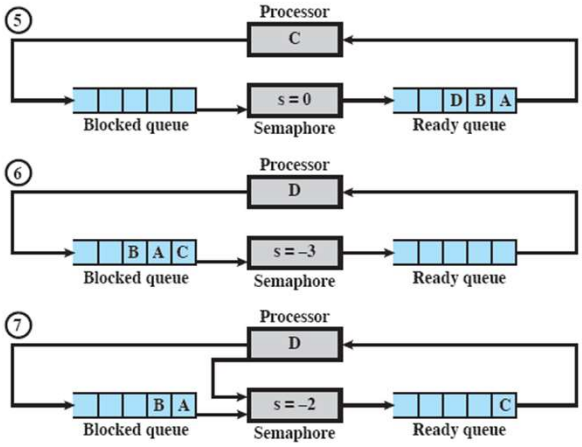


Figure 5.5 Example of Semaphore Mechanism

Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

Processes Using Semaphore

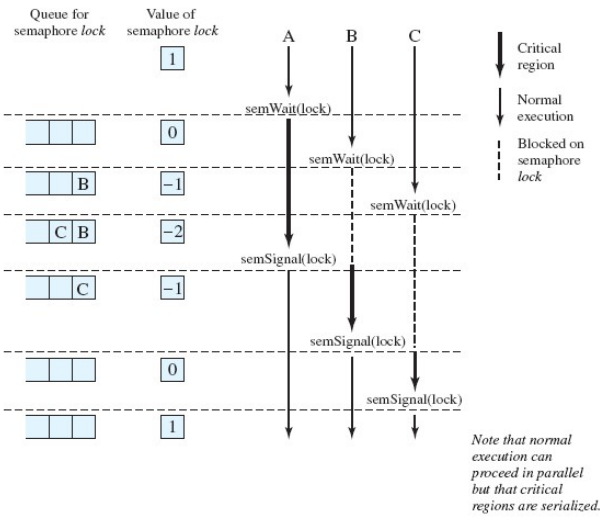


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,
  - Modula-2, Modula-3, and Java.

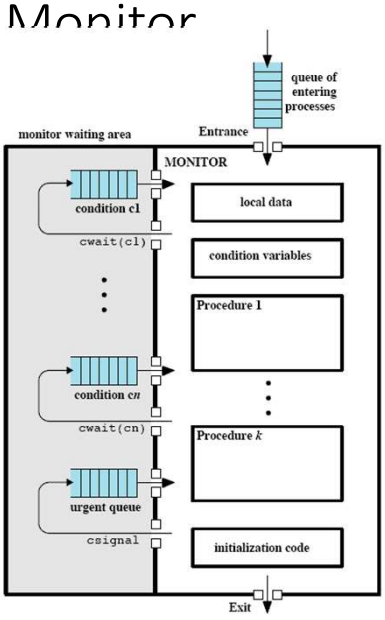
Chief characteristics

- Local data variables are accessible only by the monitor
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time

Synchronization

- Synchronisation achieved by **condition variables** within a monitor
  - only accessible by the monitor.
- Monitor Functions:
  - Cwait(c): Suspend execution of the calling process on condition *c*
  - Csignal(c) Resume execution of some process blocked after a cwait on the same condition

Structure of a Monitor



Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

Process Interaction

- When processes interact with one another, two fundamental requirements must be satisfied:
  - synchronization and
  - communication.
- Message Passing is one solution to the second requirement
  - Added bonus: It works with shared memory *and* with distributed systems

Message Passing

- The actual function of message passing is normally provided in the form of a pair of primitives:
- send (destination, message)
- receive (source, message)

Synchronization

- Communication requires synchronization
  - Sender must send before receiver can receive
- What happens to a process after it issues a send or receive primitive?
  - Sender and receiver may or may not be blocking (waiting for message)

Blocking send, Blocking receive

- Both sender and receiver are blocked until message is delivered
- Known as a *rendezvous*
- Allows for tight synchronization between processes.

Non-blocking Send

- More natural for many concurrent programming tasks.
- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait

Addressing

- Sending process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect Addressing

Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive could know ahead of time which process a message is expected
- Receive primitive could use source parameter to return a value when the receive operation has been performed

Indirect addressing

- Messages are sent to a shared data structure consisting of queues
- Queues are called *mailboxes*
- One process sends a message to the mailbox and the other process picks up the message from the mailbox

Indirect Process Communication

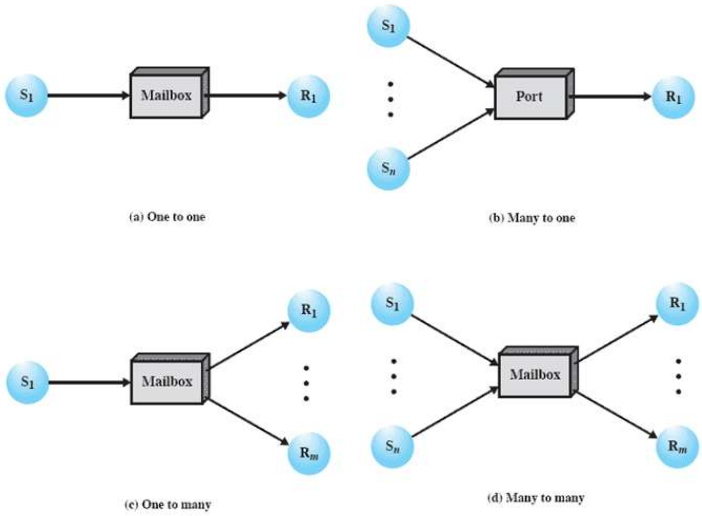


Figure 5.18 Indirect Process Communication

General Message Format

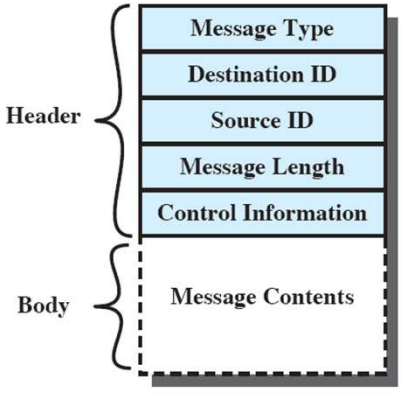


Figure 5.19 General Message Format

Mutual Exclusion Using Messages

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

Producer/Consumer Messages

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

Readers/Writers Problem

- A data area is shared among many processes
  - Some processes only read the data area, some only write to the area
- Conditions to satisfy:
  1. Multiple readers may read the file at once.
  2. Only one writer at a time may write
  3. If a writer is writing to the file, no reader may read it.

interaction of readers and writers.

Readers have Priority

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
     semWait (x);
     readcount++;
     if (readcount == 1) semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true) {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Writers have Priority

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
```

Writers have Priority

```
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Message Passing

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer_id = msg.id;
               count = count – 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

Message Passing

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer_id = msg.id;
               count = count – 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

Thank You

Operating Systems

"Threads, SMP, and Microkernels"

Roadmap

- Threads: Resource ownership and execution
- Symmetric multiprocessing (SMP).
- Microkernel
- Case Studies of threads and SMP:
  - Windows
  - Solaris
  - Linux

Processes and Threads

- Processes have two characteristics:
  - **Resource ownership** - process includes a virtual address space to hold the process image
  - **Scheduling/execution** - follows an execution path that may be interleaved with other processes
- These two characteristics are treated independently by the operating system

Processes and Threads

- The dispatching unit is referred to as a ***thread*** or lightweight process
- The entity that owns a resource is referred to as a process or ***task***
- One process / task can have one or more threads

Multithreading

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

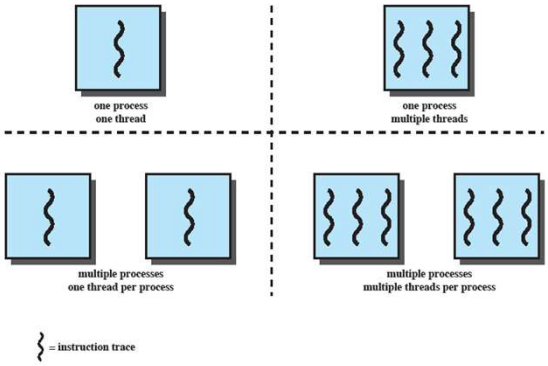


Figure 4.1 Threads and Processes [ANDE97]

Single Thread Approaches

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

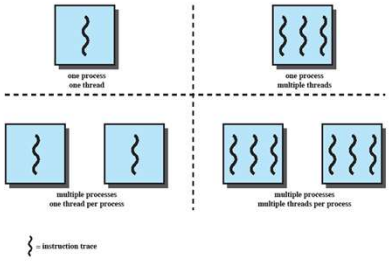


Figure 4.1 Threads and Processes [ANDE97]

Multithreading

- 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

  →**the main topic**

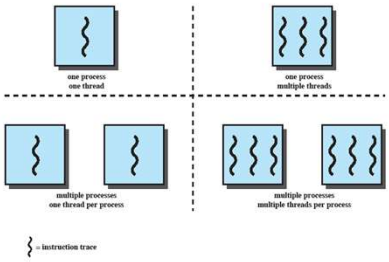


Figure 4.1 Threads and Processes [ANDE97]

Processes

- A virtual address space which holds the process image
- Protected access to
  - Processors,
  - Other processes,
  - Files,
  - I/O resources

One or More Threads in Process

- Each thread has
  - An execution state (running, ready, etc.)
  - Saved thread context when not running
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its process (all threads of a process share this)

One view…

- *One way to view a thread is as an independent program counter operating **within** a process.*

Threads vs. processes

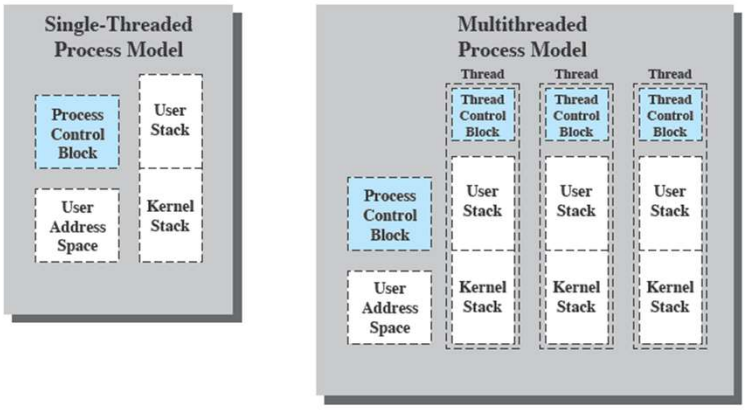


Figure 4.2 Single Threaded and Multithreaded Process Models

Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time that switching processes
- Threads can communicate with each other
  - without invoking the kernel

Thread use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure

Threads

- Several actions that affect all of the threads in a process
  - The OS must manage these at the process level.
- Examples:
  - Suspending a process involves suspending all threads of the process
  - Termination of a process, terminates all threads within the process

Activities similar to Processes

- Threads have execution states and may synchronize with one another.
  - Similar to processes
- We look at these two aspects of thread functionality in turn.
  - States
  - Synchronisation

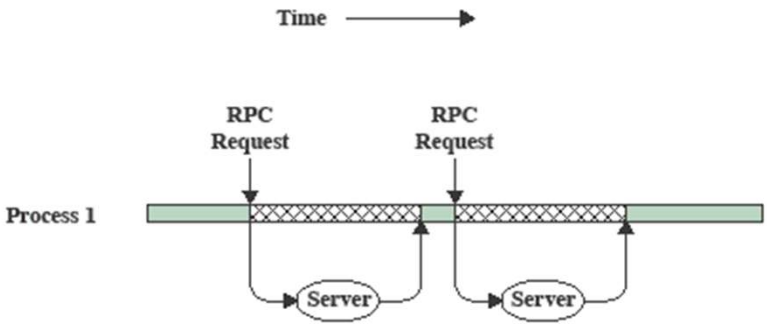
Thread Execution States

- States associated with a change in thread state
  - Spawn (another thread)
  - Block
    - Issue: will blocking a thread block other, or all threads
  - Unblock
  - Finish (thread)
    - Deallocate register context and stacks

Example: Remote Procedure Call

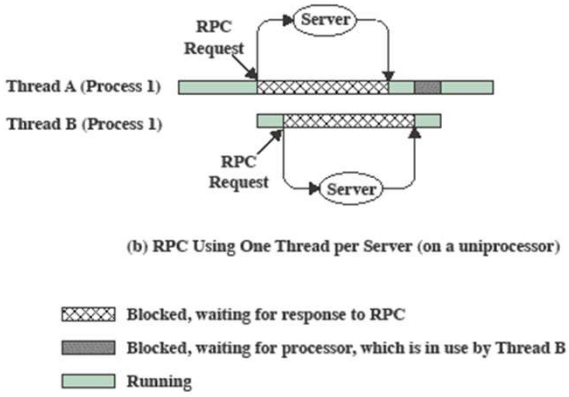
- Consider:
  - A program that performs two remote procedure calls (RPCs)
  - to two different hosts
  - to obtain a combined result.

RPC Using Single Thread



(a) RPC Using Single Thread

RPC Using One Thread per Server



Multithreading on a Uniprocessor

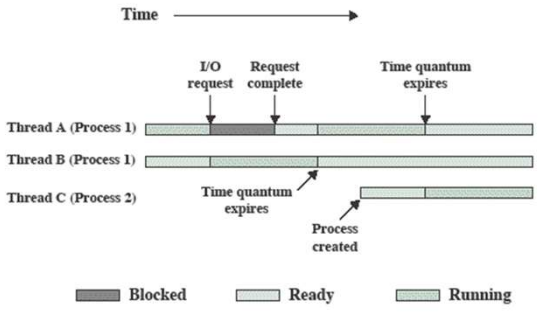


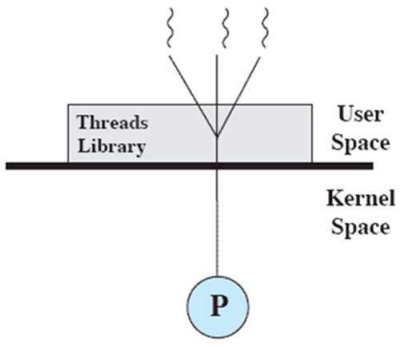
Figure 4.4 Multithreading Example on a Uniprocessor

Categories of Thread Implementation

- User Level Thread (ULT)
  - thread that are managed entirely by user-level code
  - Not requiring any support from the operating system kernel.
- Kernel level Thread (KLT) also called:
  - kernel-supported threads
  - lightweight processes.

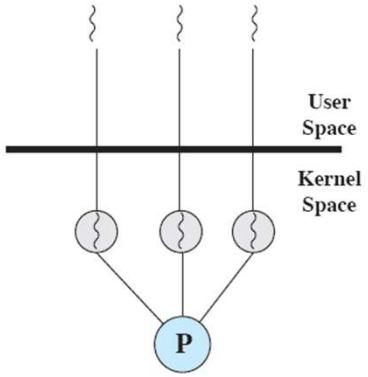
User-Level Threads

- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

Kernel-Level Threads



(b) Pure kernel-level

- Kernel maintains context information for the process and the threads
  - No thread management done by application
- Scheduling is done on a thread basis
- Windows is an example of this approach

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.

Disadvantage of KLT

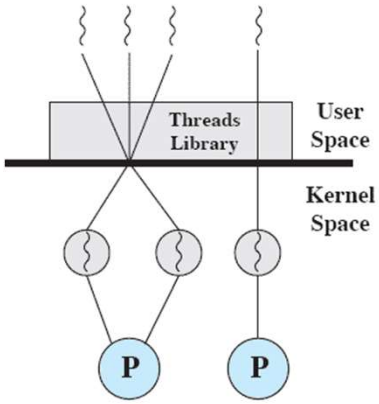
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Comparison ULT and KLT

Criteria	User-Level Threads (ULTs)	Kernel-Level Threads (KLTs)
Management	Managed entirely by user-level code, without kernel support	Managed by the operating system kernel
Scheduling	Scheduled by a user-level thread library or application, without kernel intervention	Scheduled by the operating system kernel, which may provide advanced scheduling algorithms and policies
Context Switching	Context switching occurs entirely in user space, without requiring a system call or trap into kernel mode	Context switching requires a system call or trap into kernel mode, which can incur overhead
Resources	ULTs share the same process address space and resources as the parent process	KLTs have their own kernel-level data structures, which can result in higher overhead and memory usage
Scalability	ULTs are limited to a single processor core and cannot take full advantage of multicore systems	KLTs can be assigned to different processor cores and can take full advantage of multicore systems
Synchronization	ULTs must use user-level synchronization mechanisms, which can be more efficient but may be subject to priority inversion and other issues	KLTs can use both user-level and kernel-level synchronization mechanisms, providing more flexibility and better enforcement of policies
Portability	ULTs are highly portable across different operating systems, as long as a compatible user-level thread library is available	KLTs may be less portable, as different operating systems may have different thread APIs and scheduling mechanisms

Combined Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads by the application
- Example is Solaris



(c) Combined

Relationship Between Thread and Processes

Table 4.2 Relationship Between Threads and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Roadmap

- Threads: Resource ownership and execution
- Symmetric multiprocessing (SMP).
- Microkernel
- Case Studies of threads and SMP:
  - Windows
  - Solaris
  - Linux

Traditional View

- Traditionally, the computer has been viewed as a sequential machine.
  - A processor executes instructions one at a time in sequence
  - Each instruction is a sequence of operations
- Two popular approaches to providing parallelism
  - Symmetric Multi Processors (SMPs)
  - Clusters (ch 16)

Categories of Computer Systems

- Single Instruction Single Data (SISD) stream
  - Single processor executes a single instruction stream to operate on data stored in a single memory
- Single Instruction Multiple Data (SIMD) stream
  - Each instruction is executed on a different set of data by the different processors

Categories of Computer Systems

- Multiple Instruction Single Data (MISD) stream
  - A sequence of data is transmitted to a set of processors, each of execute a different instruction sequence
- Multiple Instruction Multiple Data (MIMD)
  - A set of processors simultaneously execute different instruction sequences on different data sets

Parallel Processor Architectures

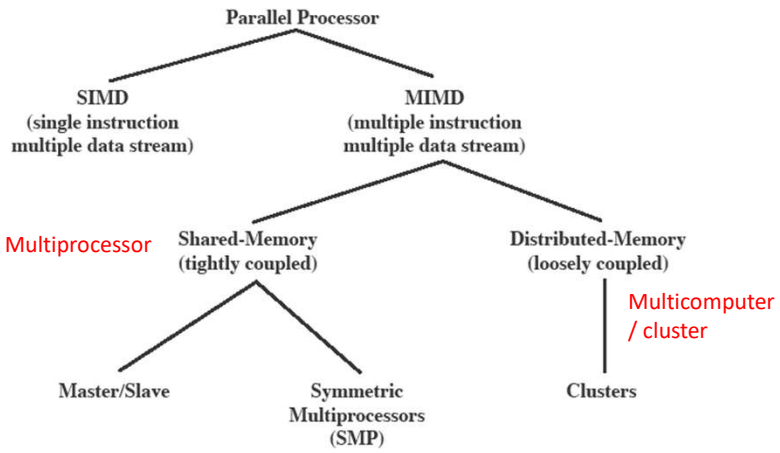


Figure 4.8 Parallel Processor Architectures

Symmetric Multiprocessing

- Kernel can execute on any processor
  - Allowing portions of the kernel to execute in parallel
- Typically, each processor does self-scheduling from the pool of available process or threads

Typical SMP Organization

symmetricity

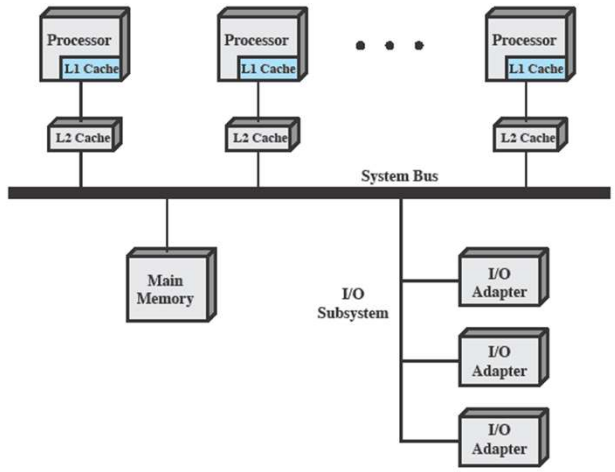


Figure 4.9 Symmetric Multiprocessor Organization

Multiprocessor OS Design Considerations

- The key design issues include
  - Simultaneous concurrent processes or threads
  - Scheduling
  - Synchronization
  - Memory Management
  - Reliability and Fault Tolerance

Roadmap

- Threads: Resource ownership and execution
- Symmetric multiprocessing (SMP).
- Microkernel
- Case Studies of threads and SMP:
  - Windows
  - Solaris
  - Linux

Microkernel

- A microkernel is a small OS core that provides the foundation for modular extensions.
- Big question is how small must a kernel be to qualify as a microkernel
  - *Must* drivers be in user space?
- In theory, this approach provides a high degree of flexibility and modularity.

Kernel Architecture

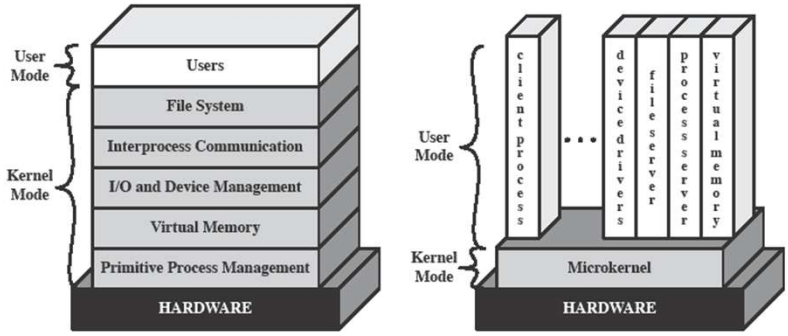


Figure 4.10 Kernel Architecture

Microkernel Design: Memory Management

- Low-level memory management - Mapping each virtual page to a physical page frame
  - Most memory management tasks occur in user space

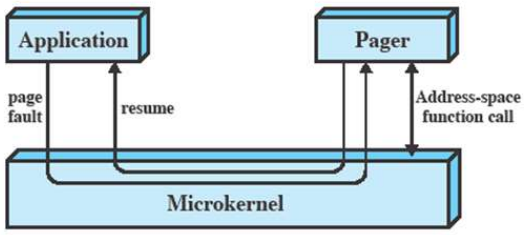


Figure 4.11 Page Fault Processing

Microkernel Design: Interprocess Communication

- Communication between processes or threads in a microkernel OS is via messages.
- A message includes:
  - A header that identifies the sending and receiving process and
  - A body that contains direct data, a pointer to a block of data, or some control information about the process.

Microkernal Design: I/O and interrupt management

- Within a microkernel, it is possible to handle hardware interrupts as messages and to include I/O ports in address spaces.
  - a particular user-level process is assigned to the interrupt and the kernel maintains the mapping.

Benefits of a Microkernel Organization

- Uniform interfaces on requests made by a process.
- Extensibility
- Flexibility
- Portability
- Reliability
- Distributed System Support
- Object Oriented Operating Systems

Comparison Microkernel vs Monolitic kernel

Criteria	Microkernel	Monolithic Kernel
Architecture	Minimalistic kernel that provides basic services, with additional services implemented as user-space processes	Entire kernel including device drivers, system calls, and other services are executed in kernel space
Security	More secure due to smaller attack surface, as most operating system services are run in user space	Exposes entire kernel to potential security vulnerabilities
Flexibility	More flexible and easier to customize and maintain as new services can be added without modifying the kernel itself	Requires modifications to kernel code to add new functionality
Performance	Generally slower due to increased reliance on interprocess communication and message passing	Better performance due to all system services being executed in kernel space, avoiding the overhead of interprocess communication
Debugging and Maintenance	Easier to debug and maintain due to modular design, where different services run as independent processes	Debugging and maintenance can be more difficult due to all services running in the same kernel space

Roadmap

- Threads: Resource ownership and execution
- Symmetric multiprocessing (SMP).
- Microkernel
- Case Studies of threads and SMP:
  - Windows
  - Solaris
  - Linux

Which OS uses microkernel / monolithic kernel

- Windows:
  - Windows NT and later versions use a hybrid kernel that combines elements of a microkernel and a monolithic kernel.
  - Earlier versions of Windows (such as Windows 95, 98, and ME) use a monolithic kernel.
- Linux:
  - a monolithic kernel, but it can also support loadable kernel modules that can be dynamically loaded and unloaded at runtime.
- macOS:
  - a hybrid kernel that combines elements of a microkernel and a monolithic kernel.
- iOS:
  - a hybrid kernel that combines elements of a microkernel and a monolithic kernel.
- Android:
  - a monolithic kernel, but it also includes a user-level component called the Binder IPC mechanism that provides some microkernel-like functionality.

Which OS uses microkernel / monolithic kernel

- some research and niche operating systems that use microkernels, such as
  - QNX, Minix, and L4 microkernel.
- Minix can be downloaded from the official website: http://www.minix3.org/download/.
  - select the version of Minix you want to download (e.g., stable or development) and the architecture (e.g., x86, ARM, or PowerPC), and the format (e.g., ISO or USB image).
- Install Minix on virtual machine (VM) such as VirtualBox and VMware

Different Approaches to Processes

- Differences between different OS's support of processes include
  - How processes are named
  - Whether threads are provided
  - How processes are represented
  - How process resources are protected
  - What mechanisms are used for inter-process communication and synchronization
  - How processes are related to each other

Windows Processes

- Processes and services provided by the Windows Kernel are relatively simple and general purpose
  - Implemented as objects
  - An executable process may contain one or more threads
  - Both processes and thread objects have built-in synchronization capabilities

Relationship between Process and Resources

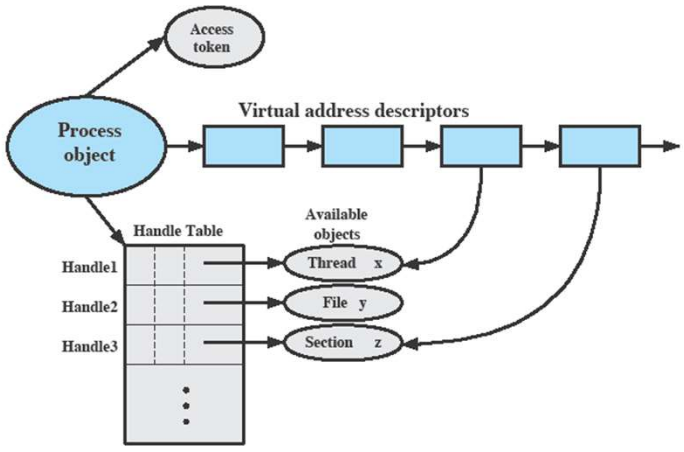
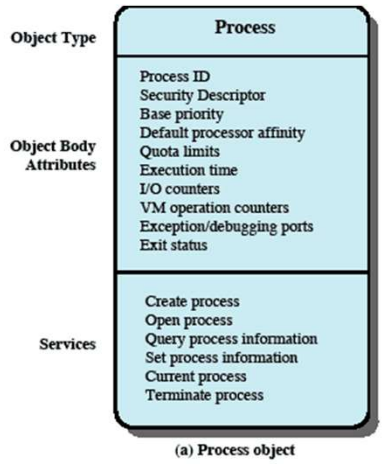


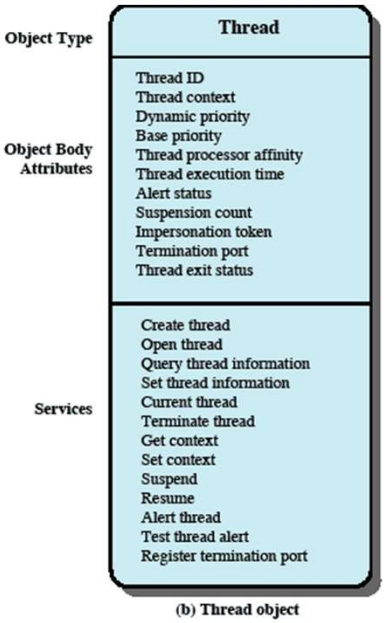
Figure 4.12 A Windows Process and Its Resources

Windows Process Object



(a) Process object

Windows Thread Object



(b) Thread object

Thread States

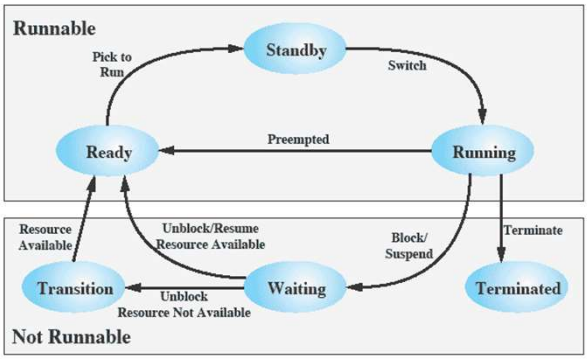


Figure 4.14 Windows Thread States

Windows SMP Support

- Threads can run on any processor
  - But an application can restrict affinity / natural liking
- Soft Affinity
  - The dispatcher tries to assign a ready thread to the same processor it last ran on.
  - This helps reuse data still in that processor's memory caches from the previous execution of the thread.
- Hard Affinity
  - An application restricts threads to certain processor

Solaris

- Solaris implements multilevel thread support designed to provide flexibility in exploiting processor resources.
- Processes include the user's address space, stack, and process control block

Solaris Process

- Solaris makes use of four separate thread-related concepts:
  - Process: includes the user's address space, stack, and process control block.
  - User-level threads: a user-created unit of execution within a process.
  - Lightweight processes: a mapping between ULTs and kernel threads.
  - Kernel threads

Relationship between Processes and Threads

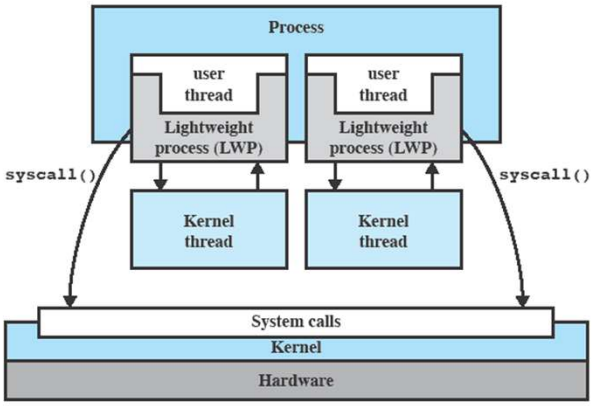


Figure 4.15 Processes and Threads in Solaris [MCDO07]

Traditional Unix vs Solaris

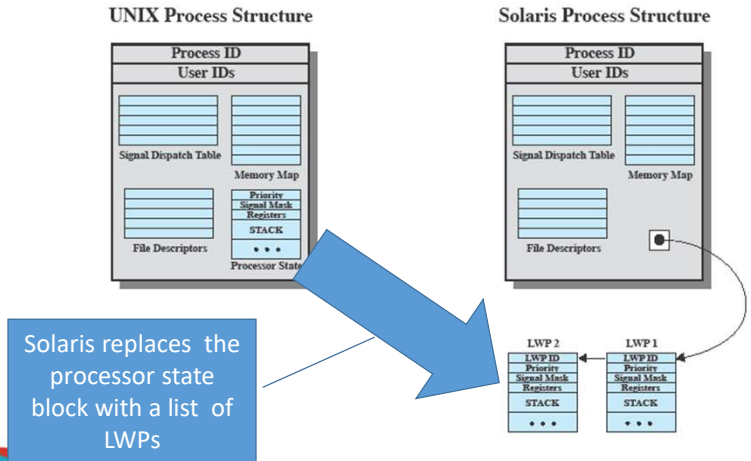


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

LWP Data Structure

- An LWP identifier
- The priority of this LWP
- A signal mask
- Saved values of user-level registers
- The kernel stack for this LWP
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

Solaris Thread States

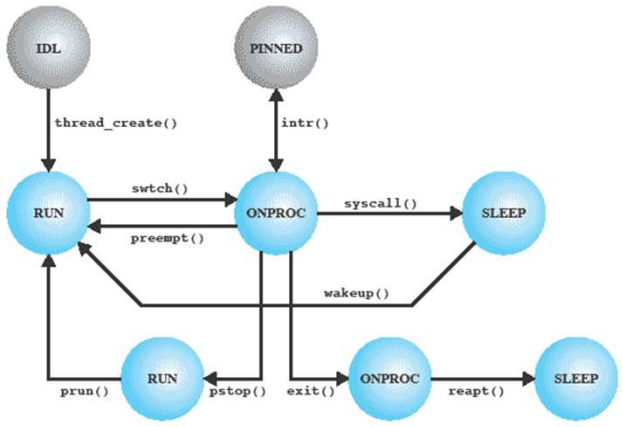


Figure 4.17 Solaris Thread States [MCDO07]

Linux Tasks

- A process, or task, in Linux is represented by a task_struct data structure
- This contains a number of categories including:
  - State
  - Scheduling information
  - Identifiers
  - Interprocess communication
  - And others

Linux Process/Thread Model

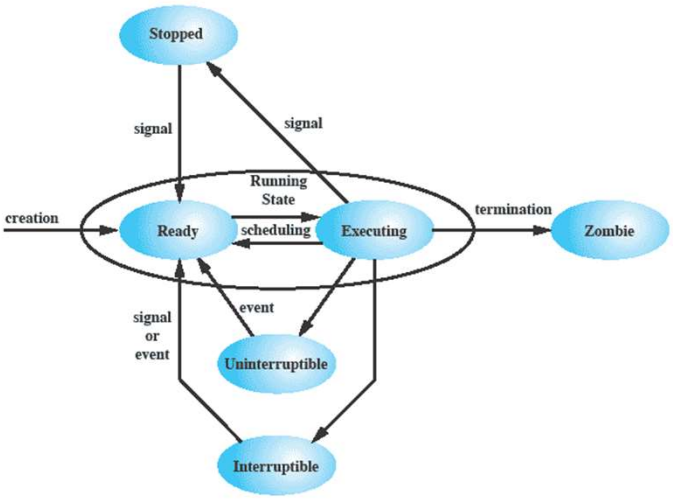


Figure 4.18 Linux Process/Thread Model

Thank You

Operating Systems

"Synchronization and Semaphores"

Outline

- Fundamentals of semaphore
- Mutual Exclusion Using Semaphores
- Examples
- Analysis
- Implementation

Semaphores

- Fundamental Principle:
  - Two or more processes want to cooperate by means of simple signals
- Special Variable: **`semaphore s`**
  - A special kind of “int” variable
  - Can’t just modify or set or increment or decrement it

Semaphores

- Before entering critical section
  - **`semWait(s)`**
    - Receive signal via semaphore **`s`**
    - “down” on the semaphore
    - Other term: **`P`** – proberen (“to try, prove”)
- After finishing critical section
  - **`semSignal(s)`**
    - Transmit signal via semaphore **`s`**
    - “up” on the semaphore
    - Other term : **`V`** – verhogen (to make or become higher)
- Implementation requirements
  - **`semSignal`** and **`semWait`** must be atomic

Inside a Semaphore

- Requirement
  - No two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time!
- Critical section
  - **`wait()`** and **`signal()`** code
  - Now have busy waiting in critical section implementation
    - + Implementation code is short
    - + Little busy waiting if critical section rarely occupied
    - – Bad for applications may spend lots of time in critical sections

Inside a Semaphore

- Add a waiting queue
- Multiple process waiting on **`s`**
  - Wakeup one of the blocked processes upon getting a signal
- Semaphore data structure

```
typedef struct {
  int count;
  queueType queue;
  /* queue for procs. waiting on s */
} SEMAPHORE;
```

Binary Semaphores

```
typedef struct bsemaphore {
  enum {0,1} value;
  queueType queue;
 } BSEMAPHORE;

void semWaitB(bsemaphore s) {
        if (s.value == 1)
                s.value = 0;
        else {
                place P in s.queue;
                block P;
        }
}

void semSignalB (bsemaphore s) {
  if (s.queue is empty())
        s.value = 1;
  else {
        remove P from s.queue;
        place P on ready list;
  }
}
```

General Semaphore

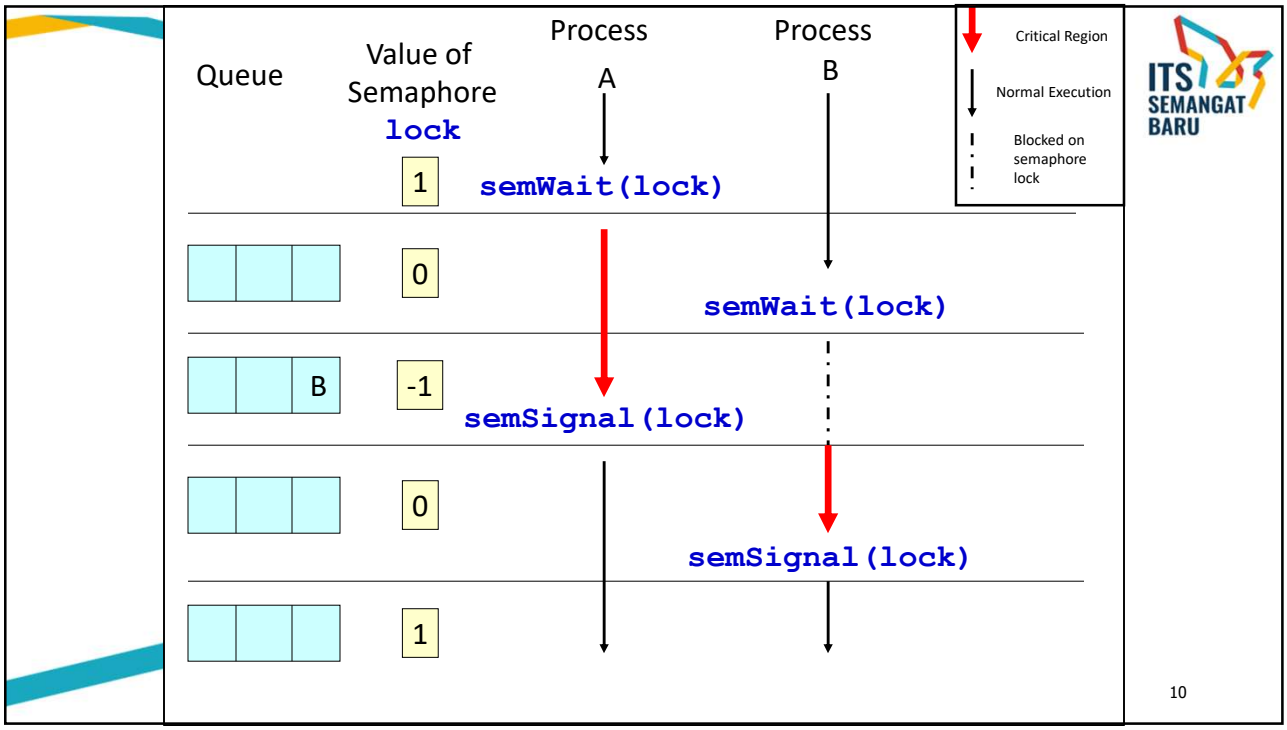
```
typedef struct {
  int count;
   queueType queue;
} SEMAPHORE

void semWait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
}

void semSignal(semaphore s) {
  s.count++;
  if (s.count ≤ 0) {
        remove P from s.queue;
        place P on ready list;
  }
}
```

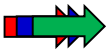
Mutual Exclusion Using Semaphores

```
semaphore s = 1;
Pi {
 while(1)     {
     semWait(s);
     /* Critical Section */
     semSignal(s);
     /* remainder */
 }
}
```



Semaphore Example 1

- What happens?
- When might this be desirable?

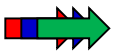


```
semaphore s = 2;
Pi {
  while(1)   {
       semWait(s);
       /* CS */
       semSignal(s);
       /* remainder */
  }
}
```

s = 2 1 0 -1

Semaphore Example 2

- What happens?
- When might this be desirable?



```
semaphore s = 0;
Pi {
  while(1)   {
       semWait(s);
       /* CS */
       semSignal(s);
       /* remainder */
  }
}
```

s = 0 -1 -2 -3

Semaphore Example 3

- What happens?
- When might this be desirable?

```
semaphore s = 0;
P1 {
  /* do some stuff */
  semWait(s);
  /* do some more stuff */
}
```

s = ~~0~~ ~~-1~~ 1

Semaphore Example 4

- Two processes
  - Two semaphores: S and Q
  - Protect two critical variables ‘a’ and ‘b’.
- What happens in the pseudocode if Semaphores S and Q are initialized to 1 (or 0)?

Process 1 executes:

```
while(1) {
      semWait(S);
      a;
      semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
      semWait(Q);
      b;
      semSignal(S);
}
```

Semaphore Example 4

Process 1 executes:

```
while(1) {
    semWait(S);
    a;
    semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
    semWait(Q);
    b;
    semSignal(S);
}
```

S = ~~0~~ -1

Q = ~~0~~ -1

Semaphore Example 4

Process 1 executes:

```
while(1) {
    semWait(S);
    a;
    semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
    semWait(Q);
    b;
    semSignal(S);
}
```

S = ~~0~~ ~~-1~~ ~~0~~ 0

Q = ~~0~~ ~~-1~~ ~~0~~ 0

Semaphore Example 4

Process 1 executes:

```
while(1) {
    semWait(S);
    a;
    semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
    semWait(Q);
    b;
    semSignal(S);
}
```

S = ~~1~~ ~~0~~ ~~-1~~ ~~-2~~ 1

Q = ~~1~~ ~~0~~ ~~-1~~ 0

Analysis

Deadlock or Violation of Mutual Exclusion?

1
```
semSignal(s);
critical_section();
semWait(s);
```

2
```
semWait(s);
critical_section();
```

3
```
critical_section();
semSignal(s);
```

4
```
semWait(s);
critical_section();
semWait(s);
```

5
```
semWait(s);
semWait(s);
critical_section();
semSignal(s);
semSignal(s);
```

Analysis

Deadlock or Violation of Mutual Exclusion?

Mutual exclusion violation

1
```
semSignal(s);
critical_section();
semWait(s);
```

Possible deadlock

2
```
semWait(s);
critical_section();
```

Mutual exclusion violation

3
```
critical_section();
semSignal(s);
```

Certain deadlock!

4
```
semWait(s);
critical_section();
semWait(s);
```

Deadlock again!

5
```
semWait(s);
semWait(s);
critical_section();
semSignal(s);
semSignal(s);
```

POSIX Semaphores

- Named Semaphores
  - Provides synchronization between related process, between threads and unrelated process
  - Kernel persistence
  - System-wide and limited in number
  - Uses **`sem_open`**
- Unnamed Semaphores
  - Provides synchronization between between related process and between threads
  - Thread-shared or process-shared
  - Uses **`sem_init`**

Example: bank balance

- Want to shared variable **balance** to be protected by semaphore when used in:
  - **decshared** – a function that decrements the current value of **balance**
  - **incshared** – a function that increments the **balance** variable.

Example: bank balance

```
int decshared() {
   while (sem_wait(&balance_sem) == -1)
      if (errno != EINTR)
         return -1;
   balance--;
   return sem_signal(&balance_sem);
}

int incshared() {
   while (sem_wait(&balance_sem) == -1)
      if (errno != EINTR)
         return -1;
   balance++;
   return sem_signal(&balance_sem);
}
```

Thank You

Operating Systems

"Deadlock and Starvation"

Learning Objectives

- List and explain the conditions for deadlock.
- Define deadlock prevention and its strategies related to each of the conditions for deadlock.
- Explain the difference between deadlock prevention and deadlock avoidance.
- Understand two approaches to deadlock avoidance.
- Explain the fundamental difference in approach between deadlock detection and deadlock prevention or avoidance.
- Understand how an integrated deadlock strategy can be designed.
- Analyze the dining philosopher's problem.
- Explain the concurrency and synchronization methods used in UNIX, Linux, Solaris, and Windows 7.

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes

Deadlock (illustration)

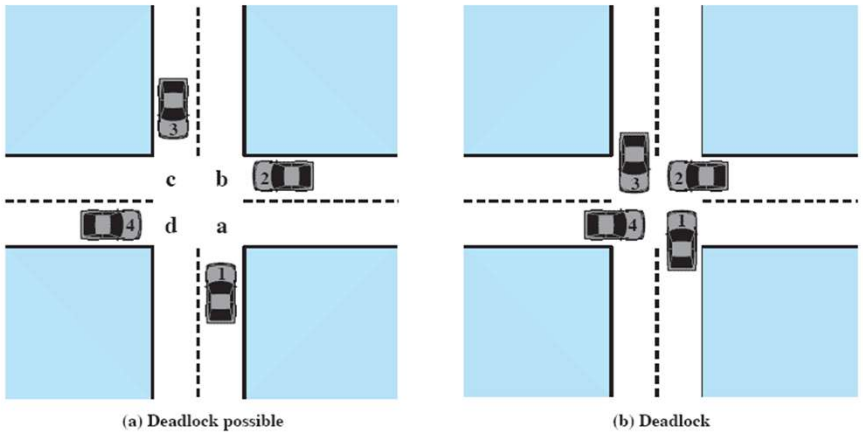


Figure 6.1 Illustration of Deadlock

Example of Deadlock

Two processes, P and Q, have the following general form:

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Deadlock

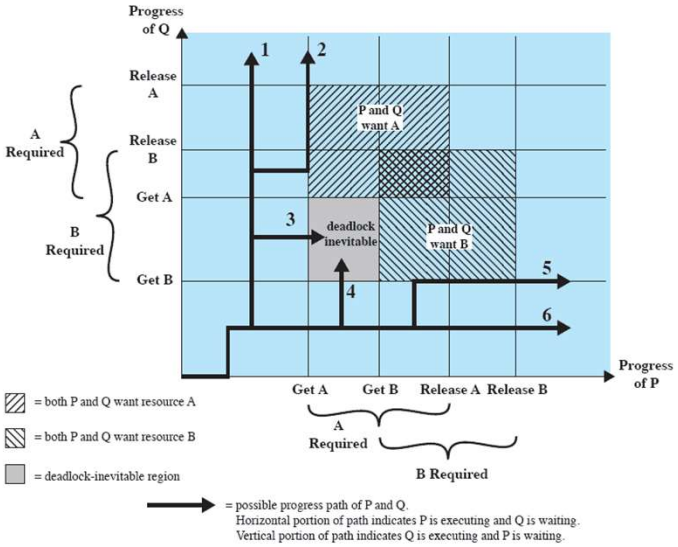


Figure 6.2 Example of Deadlock

Example of No Deadlock

- Changing the dynamics of the execution

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Deadlock

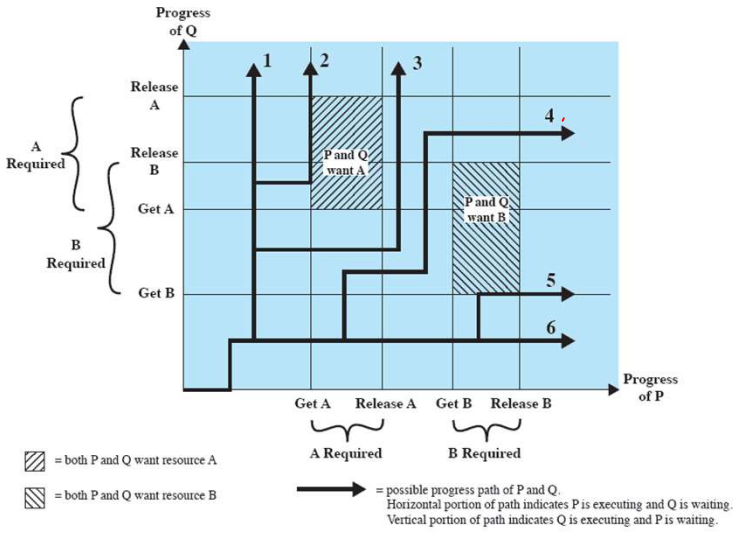


Figure 6.3 Example of No Deadlock [BACO03]

Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Example of reusable resources:
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

Reusable Resources

Process P		Process Q	
Step	**Action**	**Step**	**Action**
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Reusable Resources

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

P1	P2
. . . Request 80 Kbytes; . . . Request 60 Kbytes;	. . . Request 70 Kbytes; . . . Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Example of Deadlock

- Deadlock occurs if receives blocking

```
P1
. . .
Receive(P2);
. . .
Send(P2, M1);
```

```
P2
. . .
Receive(P1);
. . .
Send(P1, M2);
```

Resource Allocation Graphs

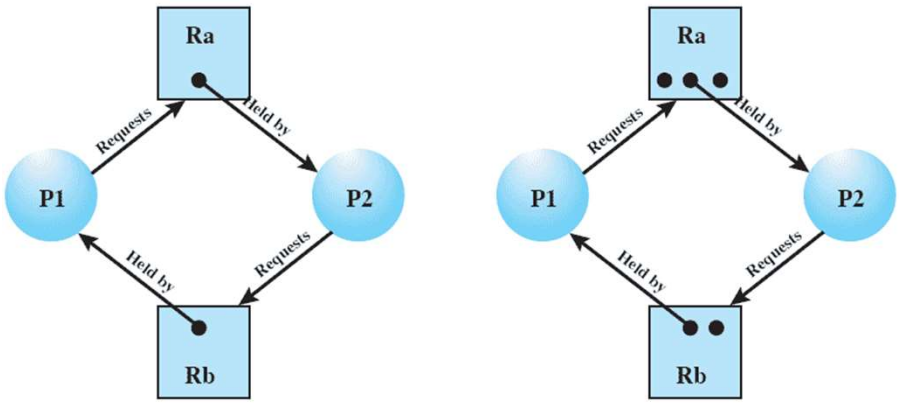
- Directed graph that depicts a state of the system of resources and processes

(a) Resouce is requested (b) Resource is held

Conditions for Deadlock

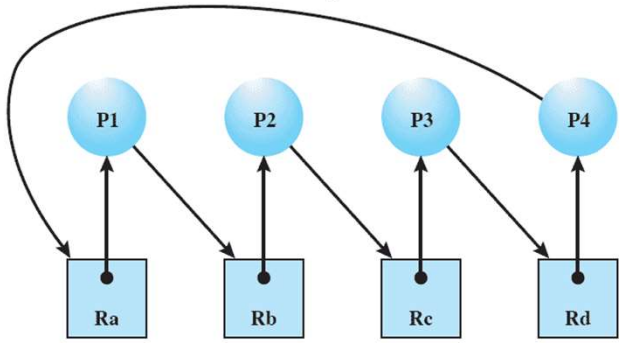
- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others
- No preemption
  - No resource can be forcibly removed form a process holding it
- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Resource Allocation Graphs



(c) Circular wait (d) No deadlock

Resource Allocation Graphs



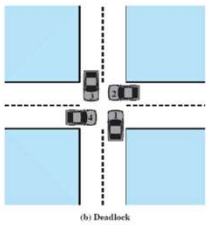


Figure 6.6 Resource Allocation Graph for

To summarize

Possibility of Deadlock	Existence of Deadlock
1. Mutual exclusion	1. Mutual exclusion
2. No preemption	2. No preemption
3. Hold and wait	3. Hold and wait
	4. Circular wait

Methods

- **Deadlock prevention**: Disallow one of the three necessary conditions (possible deadlock) for deadlock occurrence or prevent circular wait condition from happening.
- **Deadlock avoidance**: Do not grant a resource request if this allocation might lead to deadlock.
- **Deadlock detection**: Grant resource requests, when possible, but periodically check for the presence of deadlock and take action to recover.

Deadlock Prevention

- Mutual Exclusion
  - Must be supported by the OS and cannot be disallowed
- Hold and Wait
  - Require a process request all of its required resources at one time
  - To give required resources of a process at the same time
- No Preemption
  - Process must release resource and request again
  - OS may preempt a process to require it releases its resources
- Circular Wait
  - Define a linear ordering of resource types

Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests
- Approaches:
  - Do not start a process if its demands might lead to deadlock
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Resource Allocation Denial

- Referred to as **the banker's algorithm**
- State of the system is the current allocation of resources to process
- Safe state is where there is at least one sequence that does not result in deadlock
- Unsafe state is a state that is not safe

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Determination of an Unsafe State (practice)

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

Deadlock Avoidance

- Deadlock avoidance has the advantage that it is not necessary to preempt and rollback processes, as in deadlock detection, and is less restrictive than deadlock prevention. However, it does have a number of restrictions on its use:
  - Maximum resource requirement must be stated in advance
  - Processes under consideration must be independent; no synchronization requirements
  - There must be a fixed number of resources to allocate
  - No process may exit while holding resources

Deadlock Avoidance Logic

```
struct state {
      int resource[m];
      int available[m];
      int claim[n][m];
      int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                                /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Deadlock Avoidance Logic

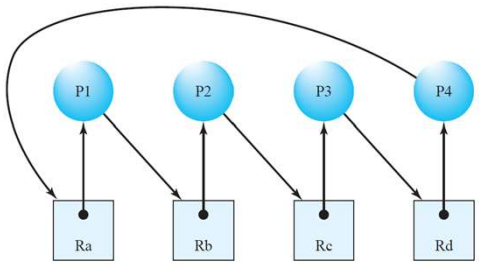
```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                          /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

Deadlock Detection

- Requested resources are granted to processes whenever possible.
- Periodically, the OS performs an algorithm that allows it to detect the circular wait condition described earlier in condition



Deadlock Detection Alg

- A request matrix *Q* is defined such that *Qij* represents the amount of resources of type *j* requested by process *i*. The algorithm proceeds by marking processes that are not deadlocked. Initially, all processes are unmarked. Then the following steps are performed:
  - Mark each process that has a row in the Allocation matrix of all zeros. A process that has no allocated resources cannot participate in a deadlock.
  - Initialize a temporary vector **W** to equal the Available vector
  - Find an index i such that process i is currently unmarked and the ith row of **Q** is less than or equal to **W**. That is, $Qik <= Wk$, for $1 <= k <= m$. If no such row is found, terminate the algorithm → **deadlock occurs**
  - If such a row is found, mark process i and add the corresponding row of the allocation matrix to W. That is, set $Wk = Wk + Aik$, for $1 <= k <= m$. Return to step 3.

Deadlock Detection

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Figure 6.10 Example for Deadlock Detection

Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Advantages and Disadvantages

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLOS80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Dining Philosophers Problem

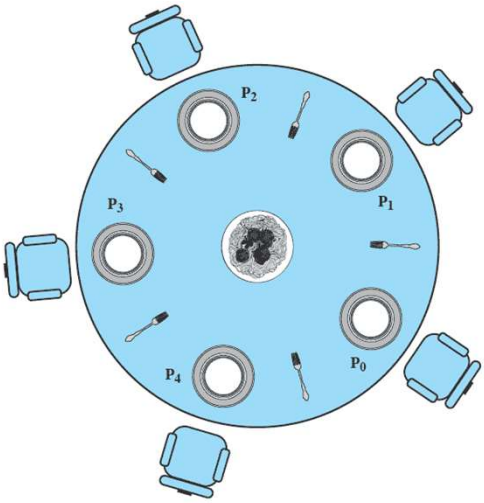


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem

```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

Dining Philosophers Problem

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

Dining Philosophers Problem

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)         /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);         /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);         /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

Dining Philosophers Problem

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true) {
      <think>;
      get forks(k);         /* client requests two forks via monitor */
      <eat spaghetti>;
      release forks(k);     /* client releases forks via the monitor */
   }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

UNIX Signals

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

Linux Kernel Concurrency Mechanism

- Includes all the mechanisms found in UNIX
- Atomic operations execute without interruption and without interference

Linux Atomic Operations

Table 6.3 Linux Atomic Operations

Atomic Integer Operations	
`ATOMIC_INIT (int i)`	At declaration: initialize an atomic_t to i
`int atomic_read(atomic_t *v)`	Read integer value of v
`void atomic_set(atomic_t *v, int i)`	Set the value of v to integer i
`void atomic_add(int i, atomic_t *v)`	Add i to v
`void atomic_sub(int i, atomic_t *v)`	Subtract i from v
`void atomic_inc(atomic_t *v)`	Add 1 to v
`void atomic_dec(atomic_t *v)`	Subtract 1 from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Subtract i from v; return 1 if the result is zero; return 0 otherwise
`int atomic_add_negative(int i, atomic_t *v)`	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
`int atomic_dec_and_test(atomic_t *v)`	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
`int atomic_inc_and_test(atomic_t *v)`	Add 1 to v; return 1 if the result is zero; return 0 otherwise

Linux Atomic Operations

Atomic Bitmap Operations	
`void set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr
`void clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr
`void change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr
`int test_and_set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr; return the old bit value
`int test_bit(int nr, void *addr)`	Return the value of bit nr in the bitmap pointed to by addr

Linux Spinlocks

`void spin_lock(spinlock_t *lock)`	Acquires the specified lock, spinning if needed until it is available
`void spin_lock_irq(spinlock_t *lock)`	Like spin_lock, but also disables interrupts on the local processor
`void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`	Like spin_lock_irq, but also saves the current interrupt state in flags
`void spin_lock_bh(spinlock_t *lock)`	Like spin_lock, but also disables the execution of all bottom halves
`void spin_unlock(spinlock_t *lock)`	Releases given lock
`void spin_unlock_irq(spinlock_t *lock)`	Releases given lock and enables local interrupts
`void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)`	Releases given lock and restores local interrupts to given previous state
`void spin_unlock_bh(spinlock_t *lock)`	Releases given lock and enables bottom halves
`void spin_lock_init(spinlock_t *lock)`	Initializes given spinlock
`int spin_trylock(spinlock_t *lock)`	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
`int spin_is_locked(spinlock_t *lock)`	Returns nonzero if lock is currently held and zero otherwise

Linux Semaphores

Traditional Semaphores	
`void sema_init(struct semaphore *sem, int count)`	Initializes the dynamically created semaphore to the given count
`void init_MUTEX(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
`void init_MUTEX_LOCKED(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 0 (initially locked)
`void down(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
`int down_interruptible(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an `up` operation is received.
`int down_trylock(struct semaphore *sem)`	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
`void up(struct semaphore *sem)`	Releases the given semaphore
Reader-Writer Semaphores	
`void init_rwsem(struct rw_semaphore, *rwsem)`	Initalizes the dynamically created semaphore with a count of 1
`void down_read(struct rw_semaphore, *rwsem)`	Down operation for readers
`void up_read(struct rw_semaphore, *rwsem)`	Up operation for readers
`void down_write(struct rw_semaphore, *rwsem)`	Down operation for writers
`void up_write(struct rw_semaphore, *rwsem)`	Up operation for writers

Linux Memory Barrier Operations

Table 6.6 Linux Memory Barrier Operations

`rmb()`	Prevents loads from being reordered across the barrier
`wmb()`	Prevents stores from being reordered across the barrier
`mb()`	Prevents loads and stores from being reordered across the barrier
`Barrier()`	Prevents the compiler from reordering loads or stores across the barrier
`smp_rmb()`	On SMP, provides a `rmb()` and on UP provides a `barrier()`
`smp_wmb()`	On SMP, provides a `wmb()` and on UP provides a `barrier()`
`smp_mb()`	On SMP, provides a `mb()` and on UP provides a `barrier()`

SMP = symmetric multiprocessor
UP = uniprocessor

Solaris Thread Synchronization Primitives

- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables

Solaris Synchronization Data Structures

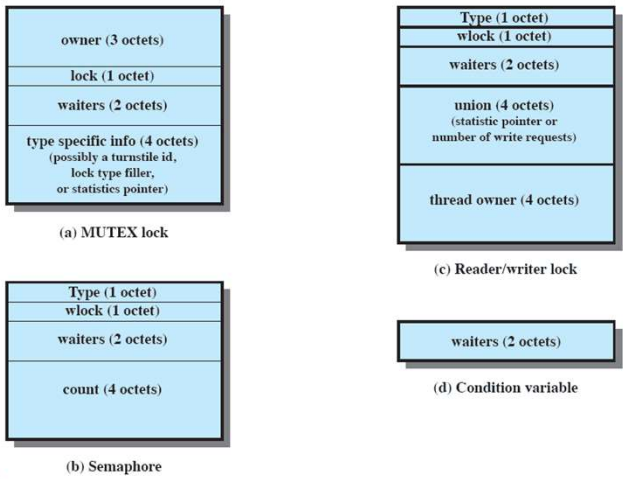


Figure 6.15 Solaris Synchronization Data Structures

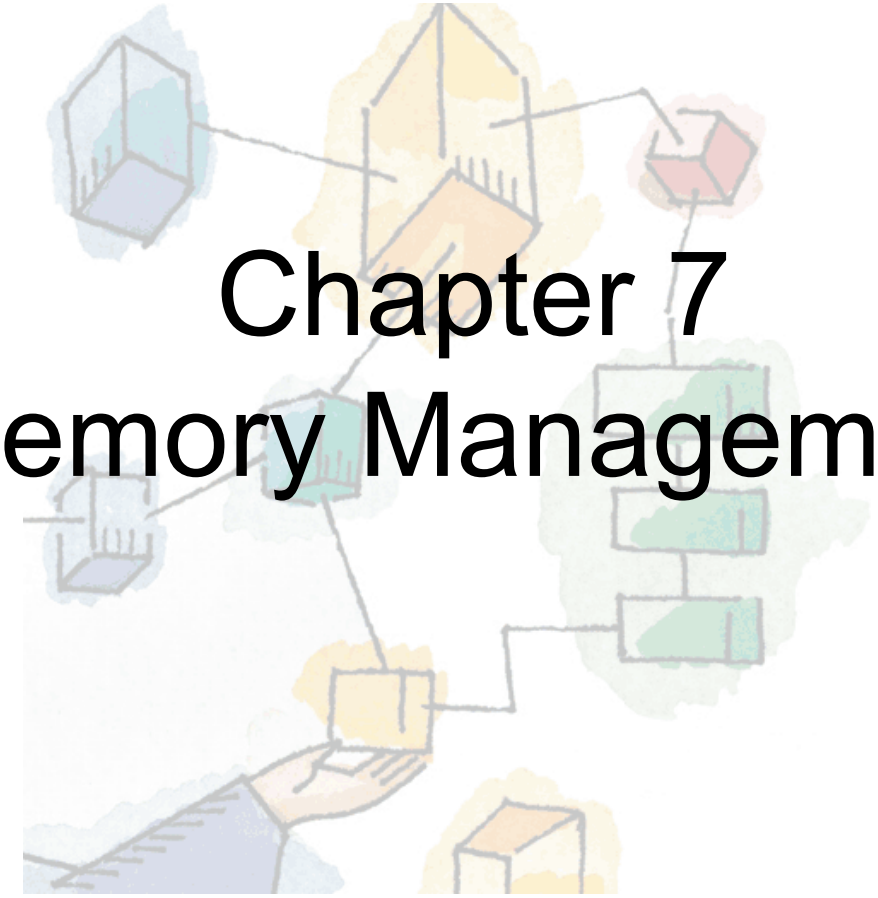
Windows Synchronization Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Note: Shaded rows correspond to objects that exist for the sole purpose of synchronization.

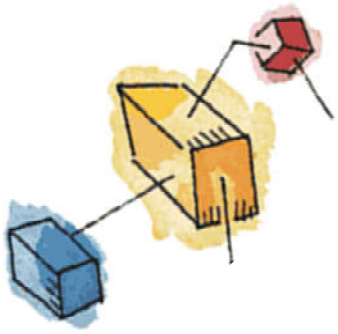
Thank You

Operating Systems:
Internals and Design Principles, 9/E
William Stallings



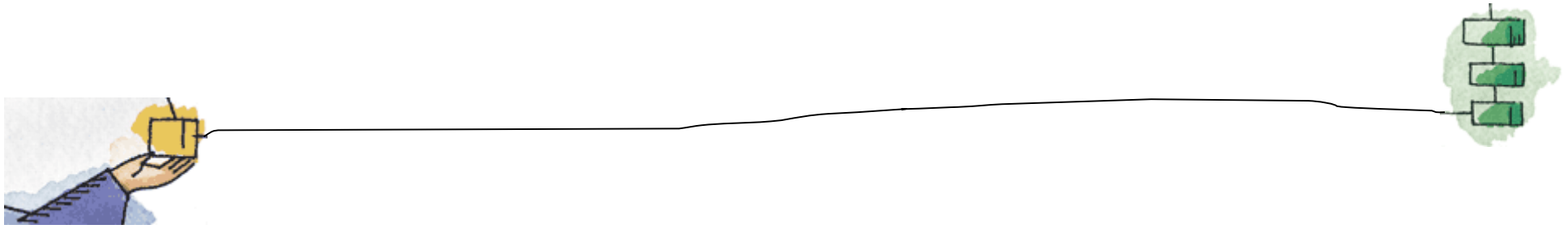
Chapter 7
Memory Management

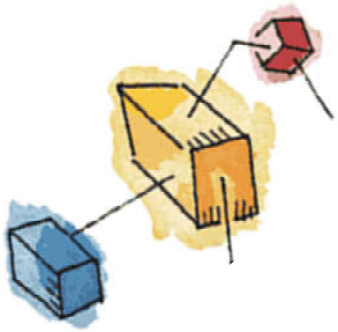
Patricia Roy
Manatee Community College, Venice, FL



Why?

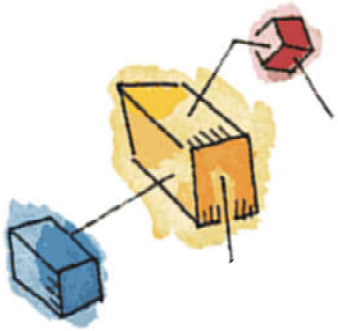
Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time





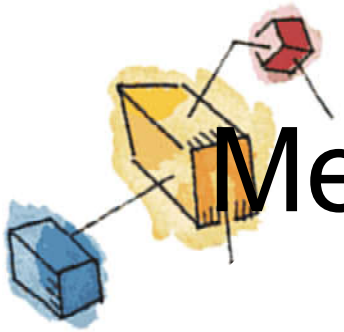
Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- Basic blocks of memory management
  - Paging
  - Segmentation



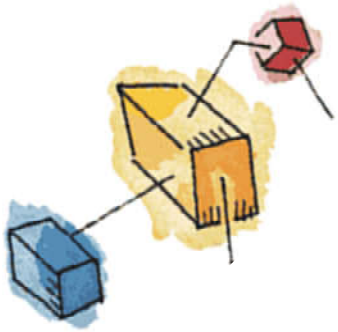
The need for memory management

- Memory is cheap today, and getting cheaper
  - But applications are demanding more and more memory, there is never enough!
- Memory Management, involves swapping blocks of data from secondary storage.
- Memory I/O is slow compared to a CPU
  - The OS must cleverly time the swapping to maximise the CPU's efficiency



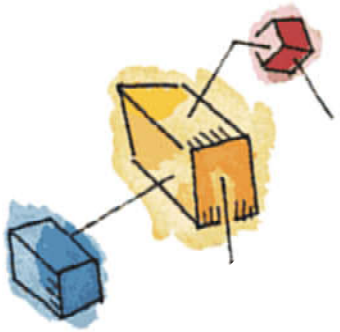
Memory Management scope

- Relocation
- Protection
- Sharing
- Logical organisation
- Physical organisation



Relocation

- The programmer does not know where the program will be placed in memory when it is executed,
  - it may be swapped to disk and return to main memory at a different location (relocated)
- Memory references must be translated to the actual physical memory address



Memory Management Terms

Table 7.1 Memory Management Terms

Term	Description
Frame	***Fixed***-length block of main memory.
Page	***Fixed***-length block of data in secondary memory (e.g. on disk).
Segment	***Variable-length*** block of data that resides in secondary memory.

Addressing

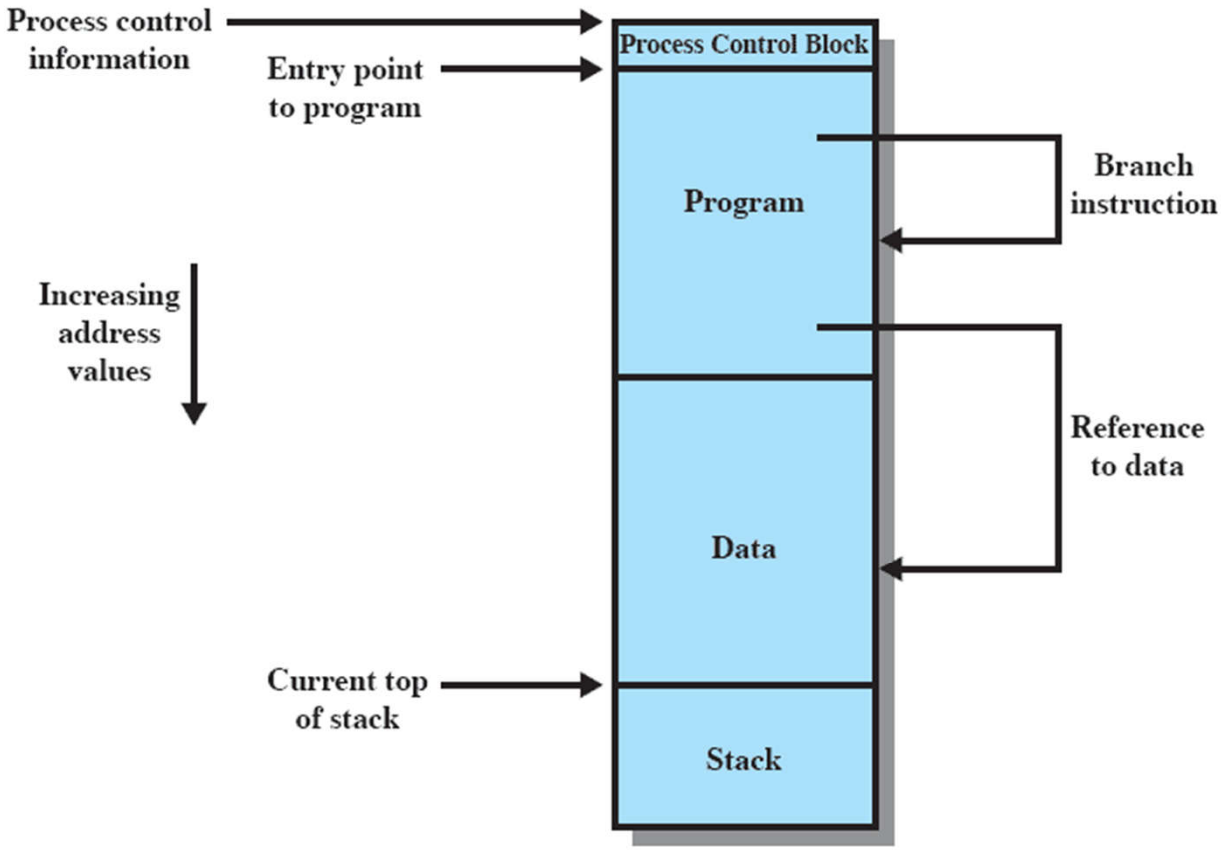
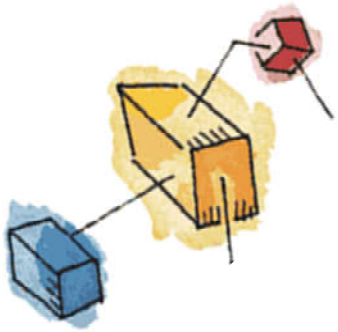
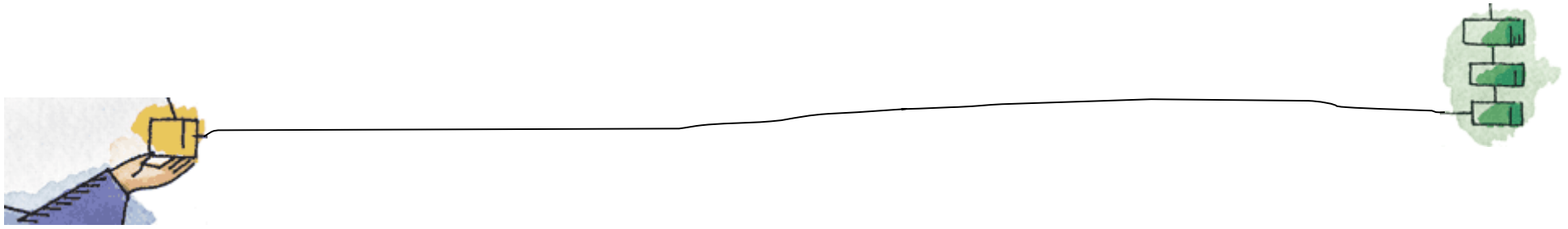


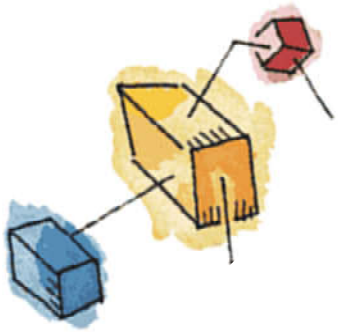
Figure 7.1 Addressing Requirements for a Process



Protection

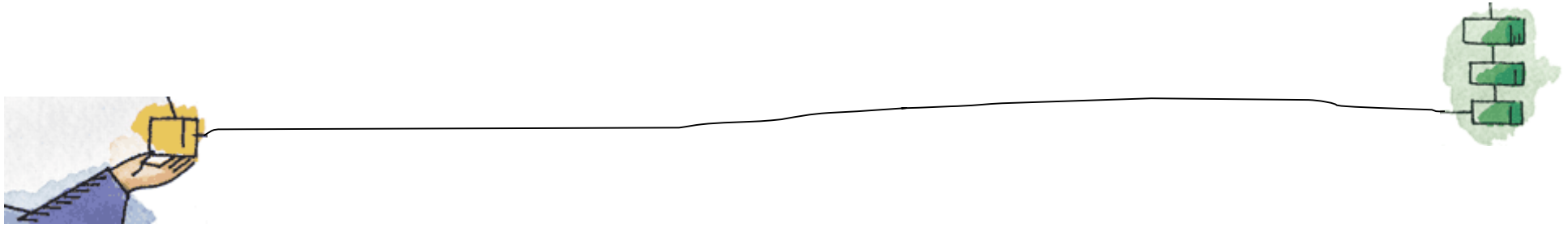
- Processes should not be able to reference memory locations in another process without permission
- Impossible to check absolute addresses at compile time
- Must be checked at run time

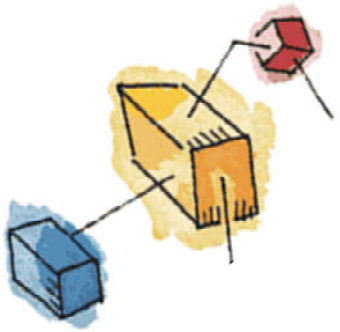




Sharing

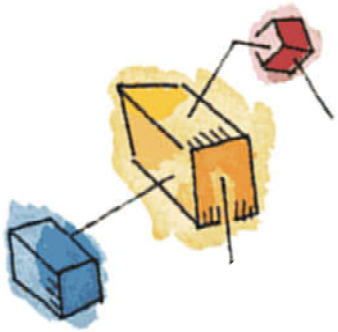
- Allow several processes to access the same portion of memory
- Better to allow each process access to the same copy of the program rather than have their own separate copy





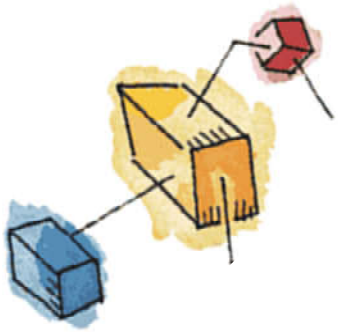
Logical Organization

- Memory is organized linearly (usually)
- Programs are written in modules
  - Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Share modules among processes



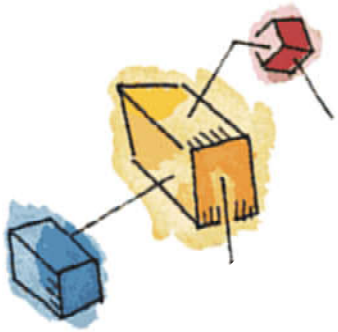
Physical Organization

- Cannot leave the programmer with the responsibility to manage memory
- Memory available for a program plus its data may be insufficient
  - Overlaying allows various modules to be assigned the same region of memory but is time consuming to program
- Programmer does not know how much space will be available



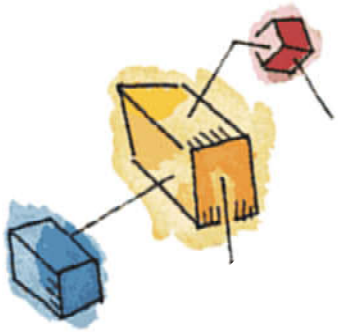
Partitioning

- An early method of managing memory
- divides primary **memory** into multiple **memory partitions**, usually contiguous areas of **memory**
- Each **partition** might contain all the information for a specific job or task.
- **Memory management** consists of allocating a **partition** to a job when it starts and unallocating it when the job ends.



Types of Partitioning

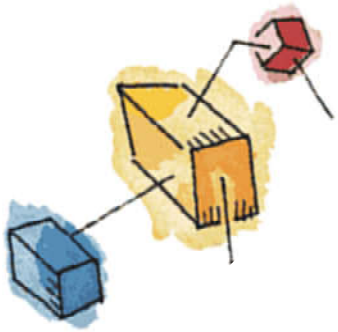
- Fixed Partitioning
- Dynamic Partitioning
- Simple Paging
- Simple Segmentation
- Virtual Memory Paging
- Virtual Memory Segmentation



Fixed Partitioning

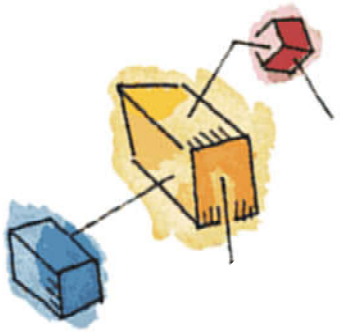
- Equal-size partitions (see fig 7.3a)
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
- The operating system can swap a process out of a partition
  - If none are in a ready or running state

(a) Equal-size partitions



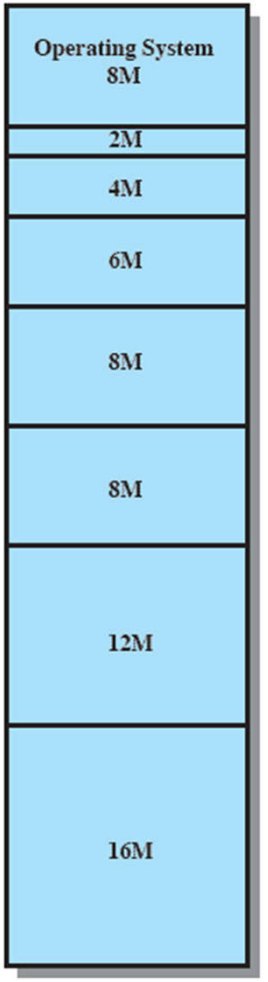
Fixed Partitioning Problems

- A program may not fit in a partition.
  - The programmer must design the program with overlays
- Main memory use is inefficient.
  - Any program, no matter how small, occupies an entire partition.
  - This is results in ***internal fragmentation.***

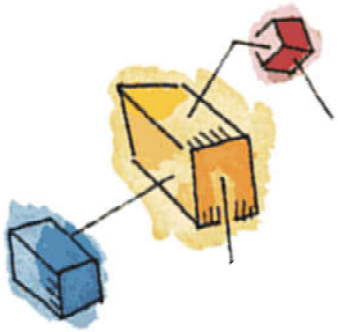


Solution – Unequal Size Partitions

- Lessens both problems
  - but doesn't solve completely
- In Fig 7.3b,
  - Programs up to 16M can be accommodated without overlay
  - Smaller programs can be placed in smaller partitions, reducing internal fragmentation
- Still a possible internal fragmentation problem

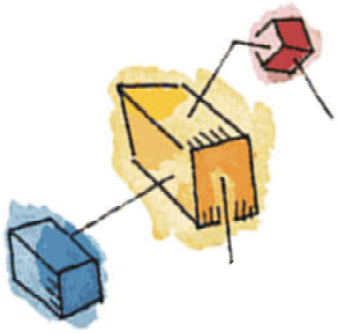


(b) Unequal-size partitions

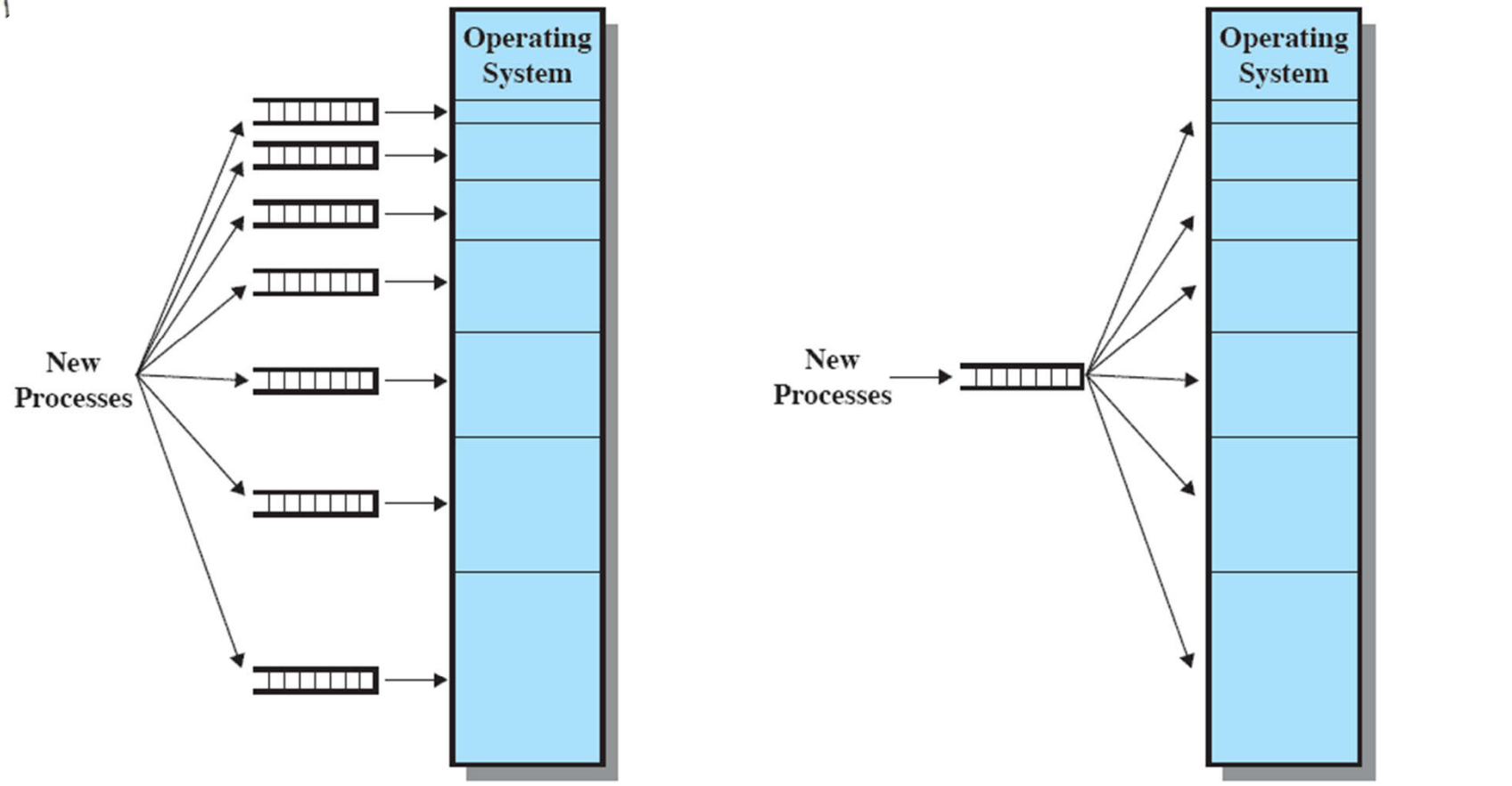


Placement Algorithm

- Equal-size
  - Placement is trivial (no options)
- Unequal-size
  - Can assign each process to the smallest partition within which it will fit
  - Queue for each partition
  - Processes are assigned in such a way as to minimize wasted memory within a partition



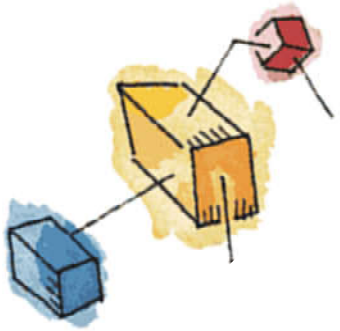
Fixed Partitioning



(a) One process queue per partition

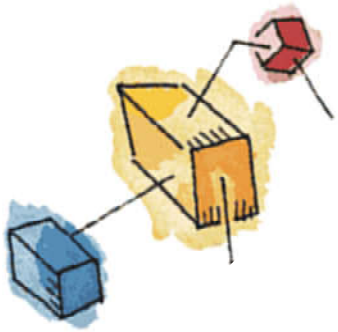
(b) Single queue

Figure 7.3 Memory Assignment for Fixed Partitioning



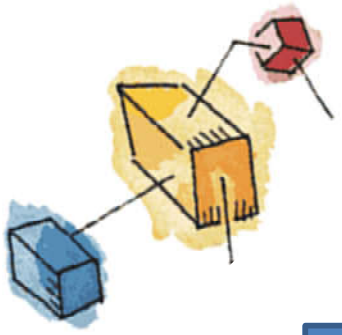
Remaining Problems with Fixed Partitions

- The number of active processes is limited by the system
  - i.e. limited by the pre-determined number of partitions
- A large number of very small process will not use the space efficiently
  - In either fixed or variable length partition methods



Dynamic Partitioning

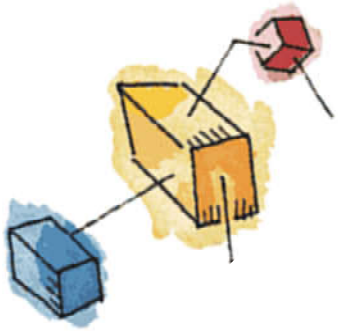
- Partitions are of variable length and number
- Process is allocated exactly as much memory as required



Dynamic Partitioning Example

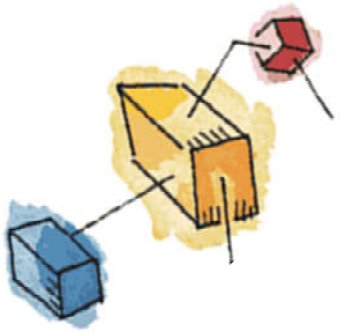
- ***External Fragmentation***
- Memory external to all processes is fragmented
- Can resolve using ***compaction***
  - OS moves processes so that they are contiguous
  - Time consuming and wastes CPU time

Refer to Figure 7.4



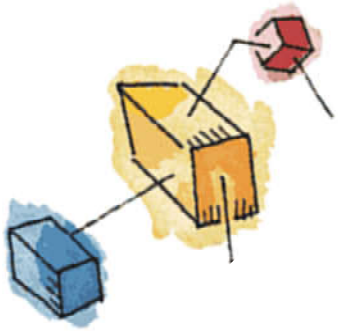
Dynamic Partitioning

- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often



Dynamic Partitioning

- First-fit algorithm
  - Scans memory form the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many process loaded in the front end of memory that must be searched over when trying to find a free block



Dynamic Partitioning

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory

Allocation

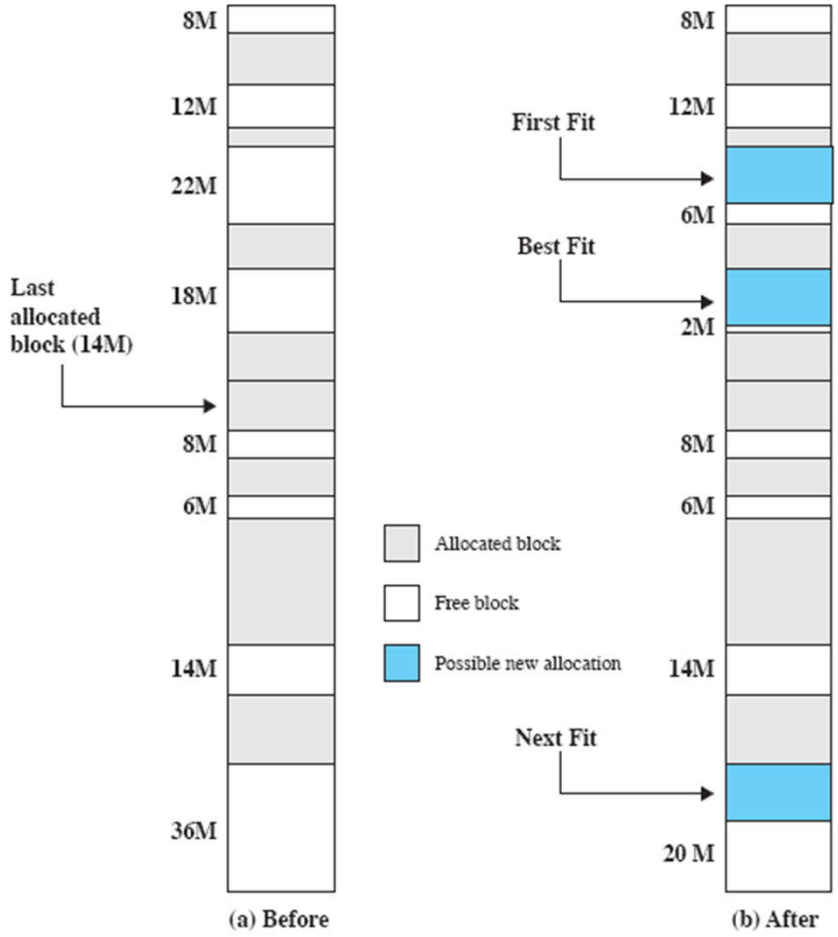
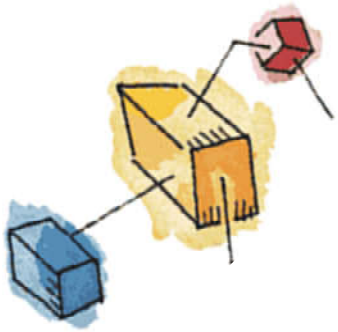
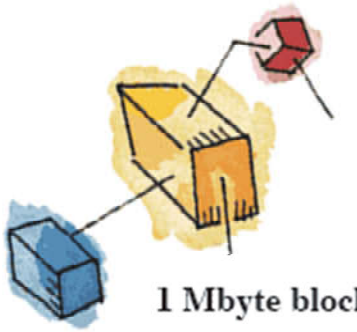


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block



Buddy System

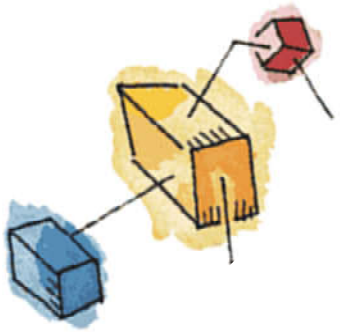
- Entire space available is treated as a single block of 2^U
- If a request of size *s* where $2^{U-1} < s <= 2^U$
  - entire block is allocated
- Otherwise, block is split into two equal buddies
  - Process continues until smallest block greater than or equal to *s* is generated



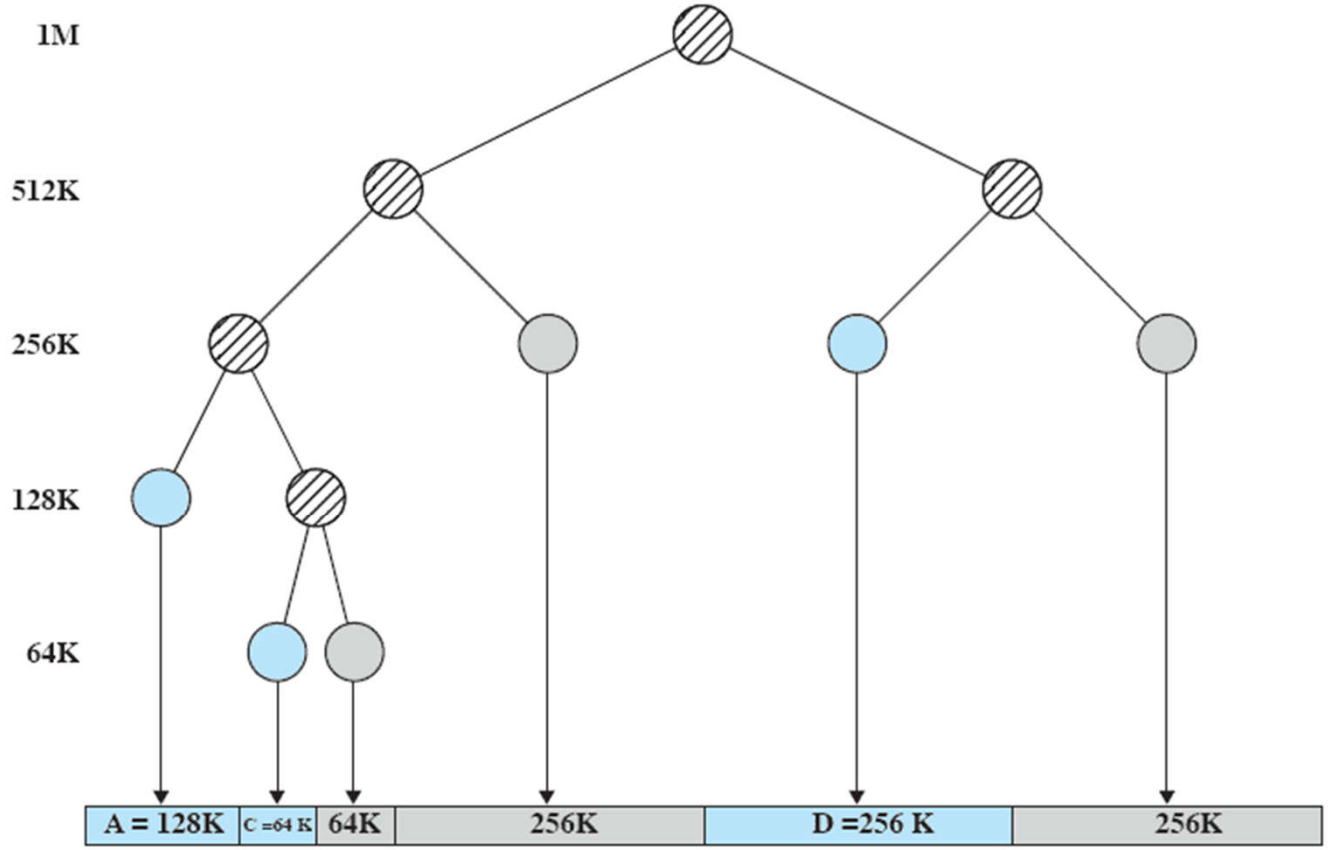
Example of Buddy System

1 Mbyte block	1 M				
Request 100 K	A = 128K	128K	256K	512K	
Request 240 K	A = 128K	128K	B = 256K	512K	
Request 64 K	A = 128K	C = 64K	64K	B = 256K	512K
Request 256 K	A = 128K	C = 64K	64K	B = 256K	D = 256K	256K
Release B	A = 128K	C = 64K	64K	256K	D = 256K	256K
Release A	128K	C = 64K	64K	256K	D = 256K	256K
Request 75 K	E = 128K	C = 64K	64K	256K	D = 256K	256K
Release C	E = 128K	128K	256K	D = 256K	256K	
Release E	512K			D = 256K	256K	
Release D	1M				

Figure 7.6 Example of Buddy System



Tree Representation of Buddy System



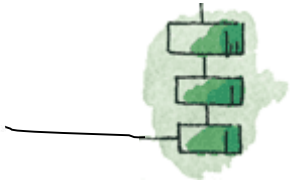
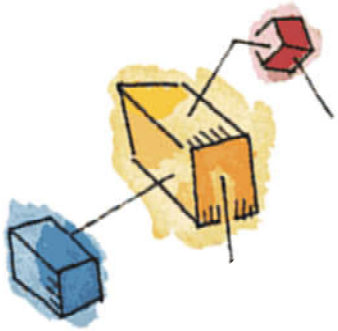
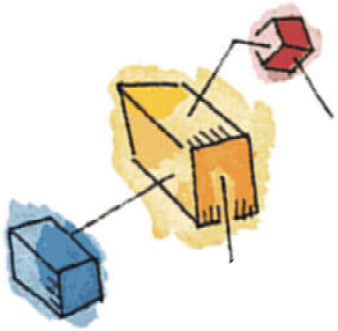


Figure 7.7 Tree Representation of Buddy System



Relocation

- When program loaded into memory the actual (absolute) memory locations are determined
- A process may occupy different partitions which means different absolute memory locations during execution, due to:
  - Swapping
  - Compaction



Addresses

- Logical
  - Reference to a memory location independent of the current assignment of data to memory.
- Relative
  - Address expressed as a location relative to some known point.
- Physical or Absolute
  - The absolute address or actual location in main memory.

Relocation

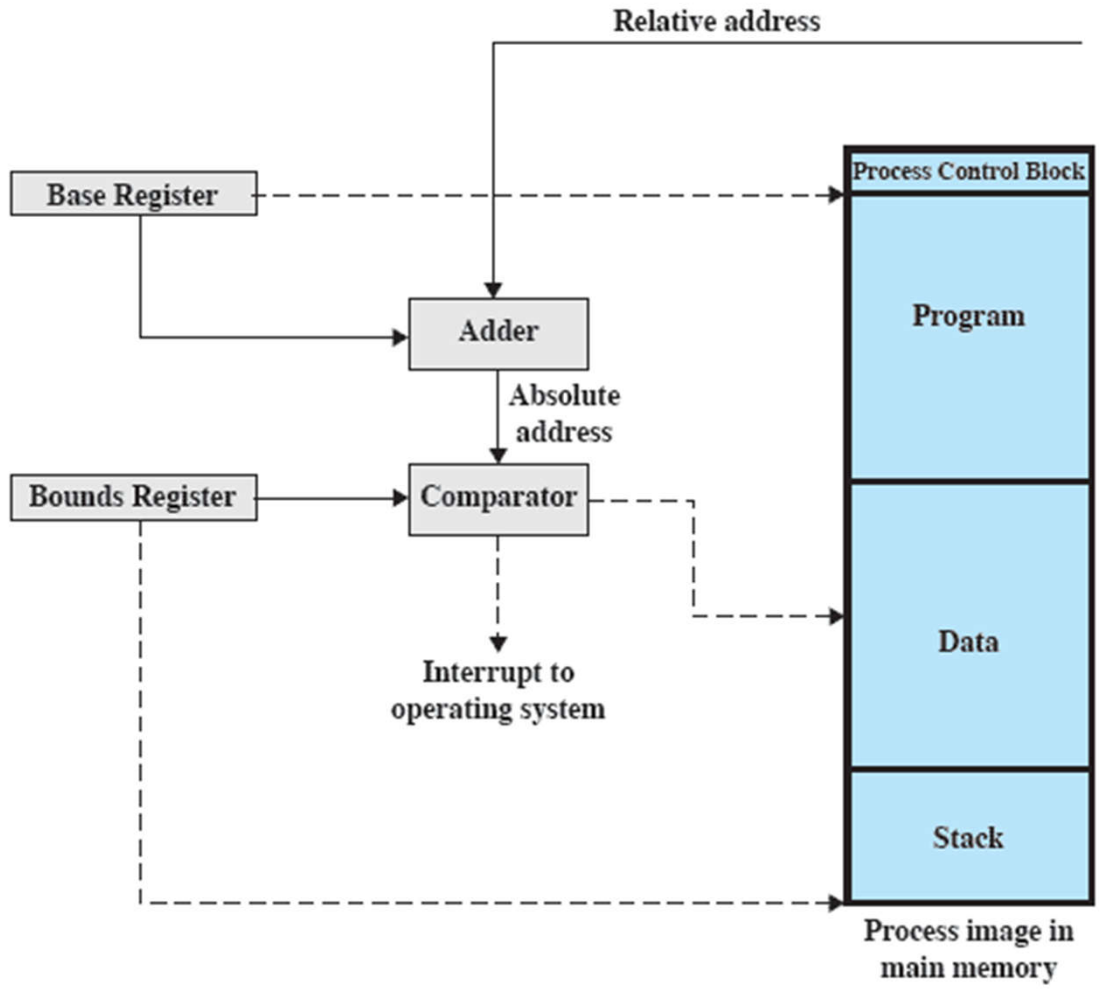
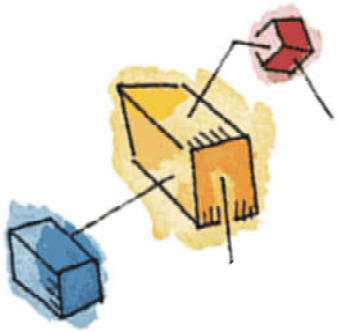
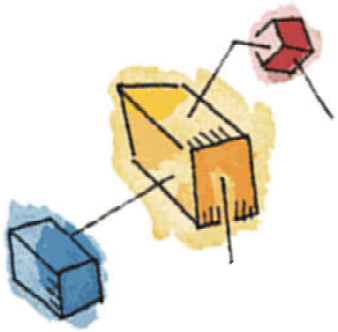


Figure 7.8 Hardware Support for Relocation



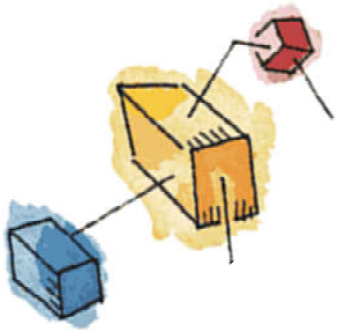
Registers Used during Execution

- Base register
  - Starting address for the process
- Bounds register
  - Ending location of the process
- These values are set when the process is loaded or when the process is swapped in



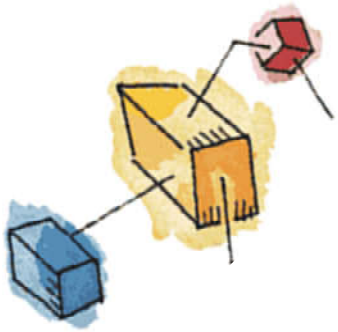
Registers Used during Execution

- The value of the base register is added to a relative address to produce an absolute address (physical address)
- The resulting address is compared with the value in the bounds (the max address in a memory) register
- If the address is not within bounds, an interrupt is generated to the operating system



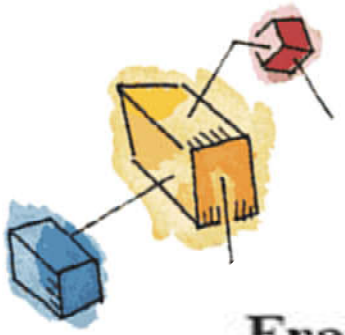
Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called ***pages***
- The chunks of memory are called ***frames***



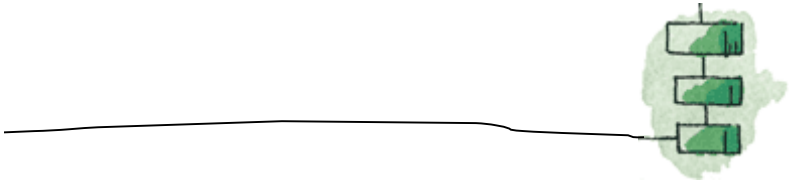
Paging

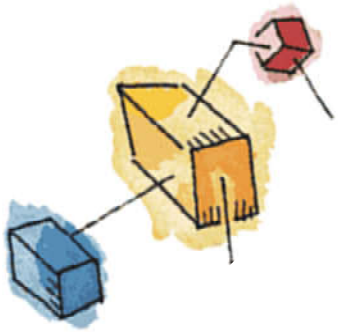
- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a page number and offset within the page



Processes and Frames

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	





Page Table

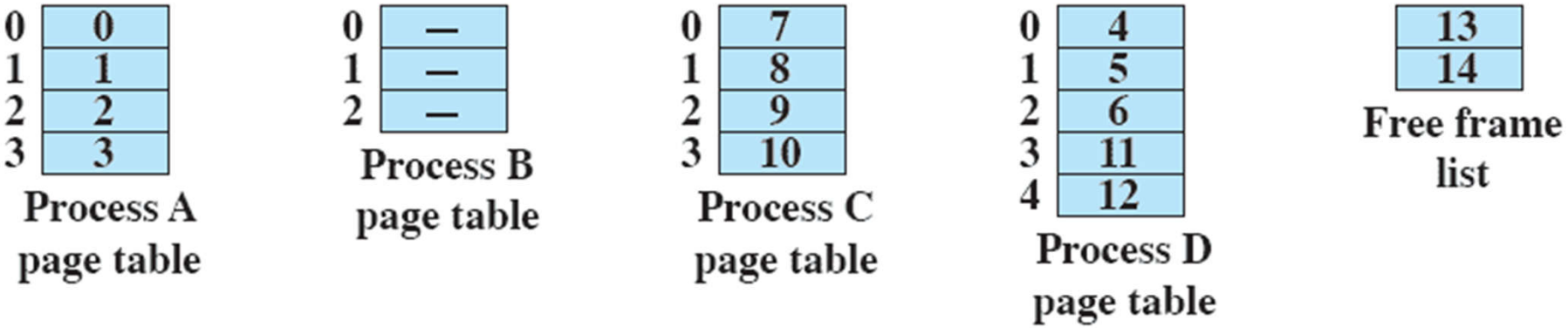
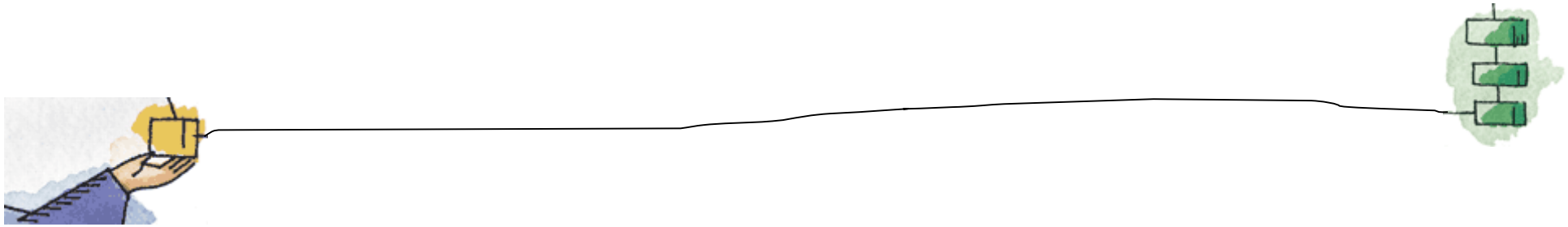
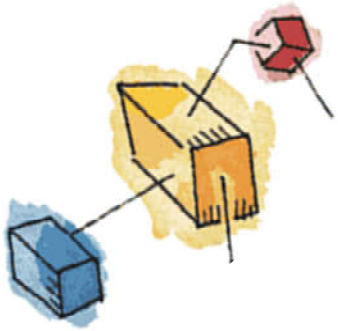


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)





Segmentation

- A program can be subdivided into segments
  - Segments may vary in length
  - There is a maximum segment length
- Addressing consist of two parts
  - a segment number and
  - an offset
- Segmentation is similar to dynamic partitioning

Logical Addresses

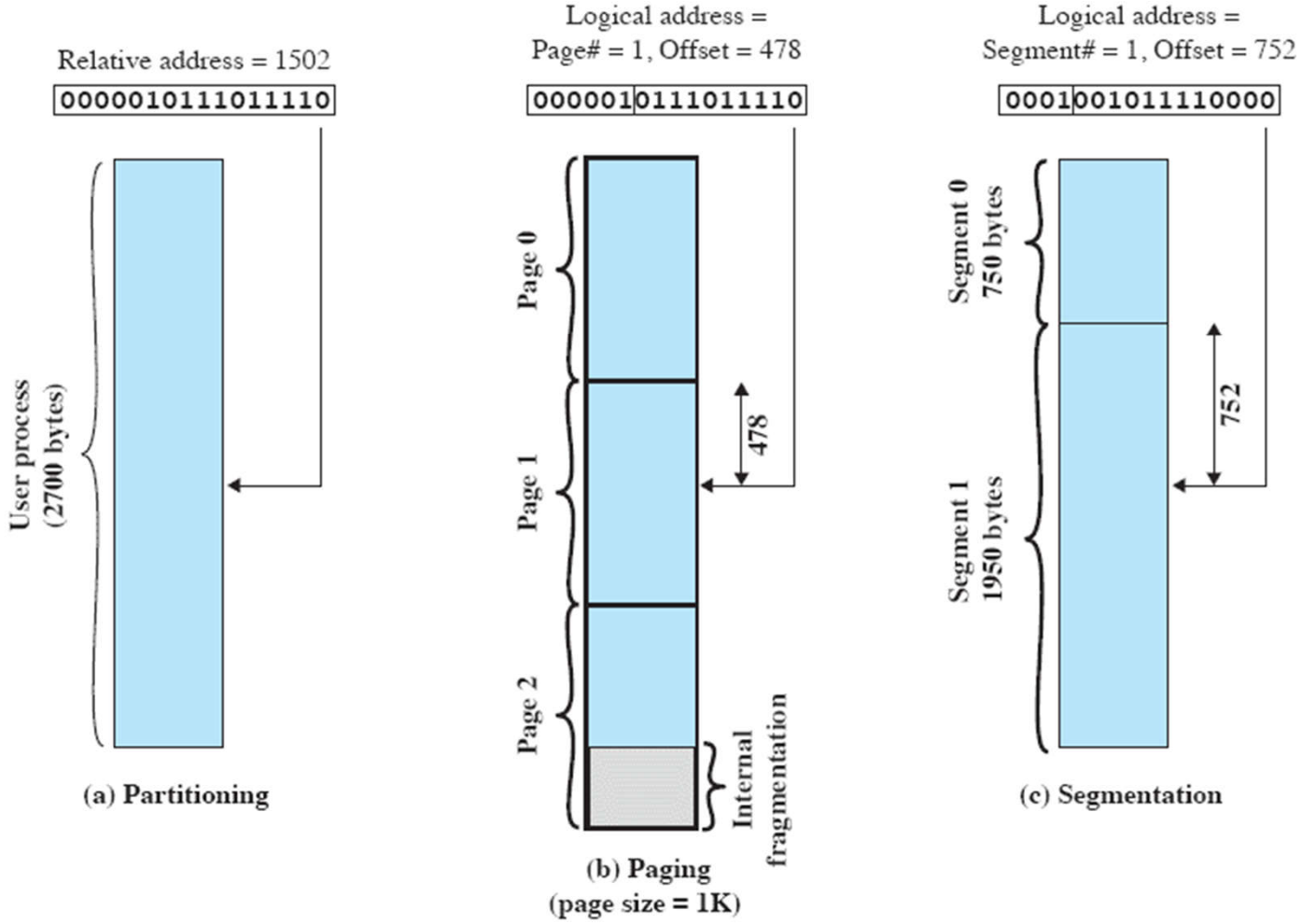
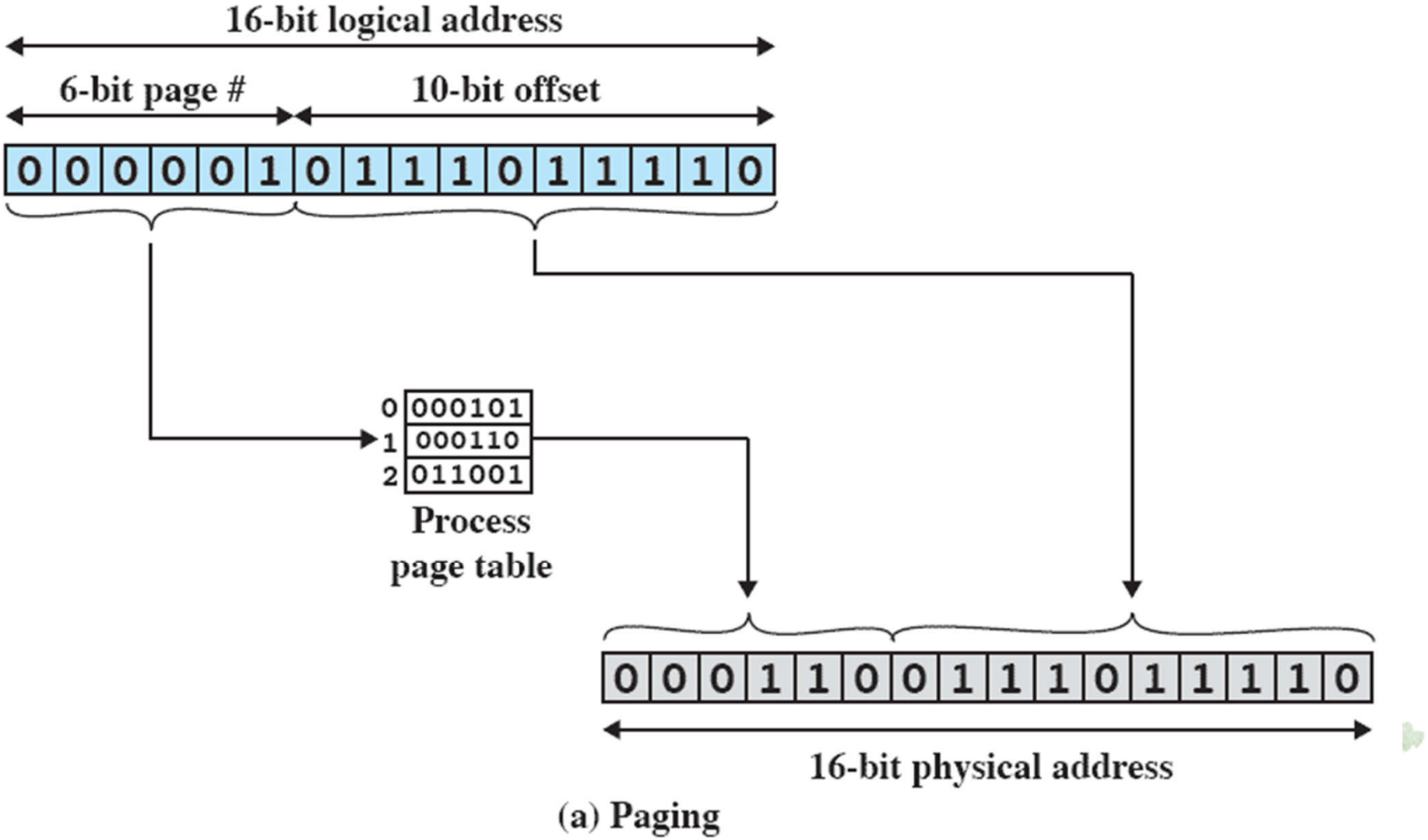


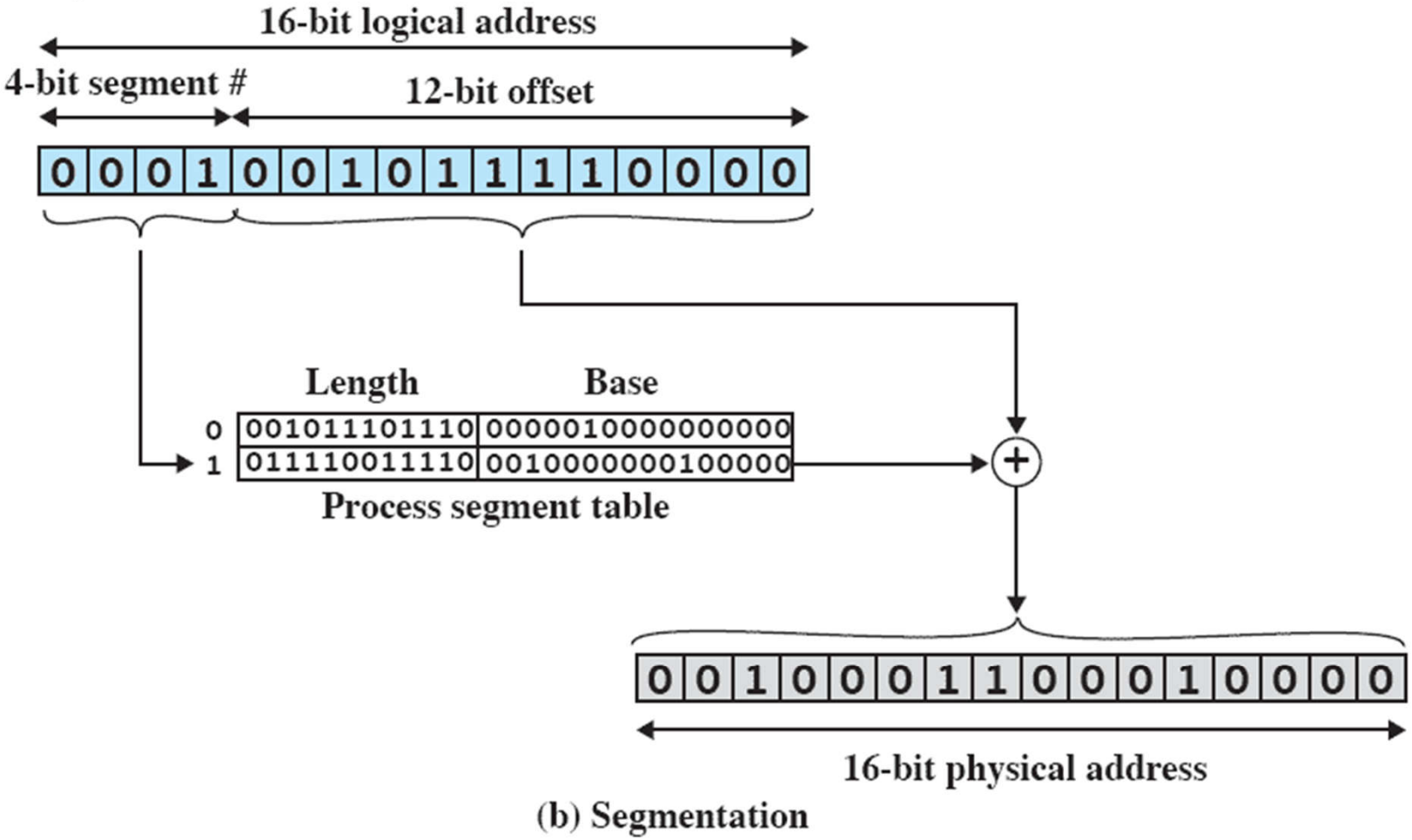
Figure 7.11 Logical Addresses

Paging



(a) Paging

Segmentation



(b) Segmentation

Figure 7.12 Examples of Logical-to-Physical Address Translation

Operating Systems

“Virtual Memory”

Roadmap

- Hardware and Control Structures
- Operating System Software

Terminology

Table 8.1 Virtual Memory Terminology

Virtual memory	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.
Virtual address	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
Virtual address space	The virtual storage assigned to a process.
Address space	The range of memory addresses available to a process.
Real address	The address of a storage location in main memory.

Real and Virtual Memory

- Real memory
  - Main memory, the actual RAM
- Virtual memory
  - Memory on disk
  - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

Key points in Memory Management

1) Memory references are logical addresses dynamically translated into physical addresses at run time
   - A process may be swapped in and out of main memory occupying different regions at different times during execution
2) A process may be broken up into pieces that do not need to be located contiguously in main memory

Breakthrough in Memory Management

- **If both** of those two characteristics are present,
  - then it is not necessary that all of the pages or all of the segments of a process be in main memory during execution.
- If the next instruction, and the next data location are in memory then execution can proceed
  - at least for a time

Execution of a Process

- Operating system brings into main memory a few pieces of the program
- Resident set - portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory
- Operating system places the process in a blocking state

Execution of a Process

- Piece of process that contains the logical address is brought into main memory
  - Operating system issues a disk I/O Read request
  - Another process is dispatched to run while the disk I/O takes place
  - An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state

Implications of this new strategy

- More processes may be maintained in main memory
  - Only load in some of the pieces of each process
  - With so many processes in main memory, it is very likely a process will be in the Ready state at any particular time
- A process may be larger than all of main memory
  - The operating system automatically loads pieces of a process into main memory as required.

Characteristics of Paging and Segmentation

Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation
Main memory partitioned into small fixed-size chunks called frames	Main memory partitioned into small fixed-size chunks called frames	Main memory not partitioned	Main memory not partitioned
Program broken into pages by the compiler or memory management system	Program broken into pages by the compiler or memory management system	Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)	Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)
Internal fragmentation within frames	Internal fragmentation within frames	No internal fragmentation	No internal fragmentation
No external fragmentation	No external fragmentation	External fragmentation	External fragmentation
Operating system must maintain a page table for each process showing which frame each page occupies	Operating system must maintain a page table for each process showing which frame each page occupies	Operating system must maintain a segment table for each process showing the load address and length of each segment	Operating system must maintain a segment table for each process showing the load address and length of each segment
Operating system must maintain a free frame list	Operating system must maintain a free frame list	Operating system must maintain a list of free holes in main memory	Operating system must maintain a list of free holes in main memory
Processor uses page number, offset to calculate absolute address	Processor uses page number, offset to calculate absolute address	Processor uses segment number, offset to calculate absolute address	Processor uses segment number, offset to calculate absolute address
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory frames for the process to run. Segments may be read in as needed
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk

Thrashing

- A state in which the system spends most of its time swapping pieces rather than executing instructions.
- To avoid this, the operating system tries to guess which pieces are least likely to be used in the near future.
  - The guess is based on recent history

Principle of Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Therefore, it is possible to make intelligent guesses about which pieces will be needed in the future
- This suggests that virtual memory may work efficiently

The locality map

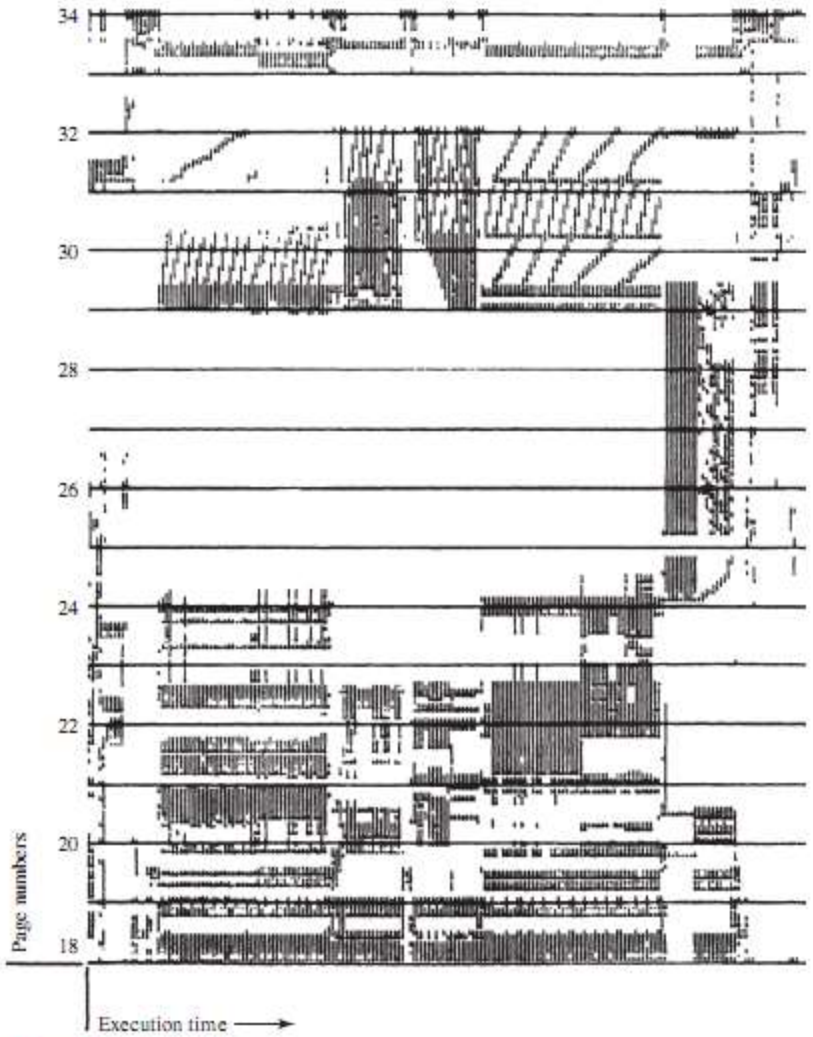


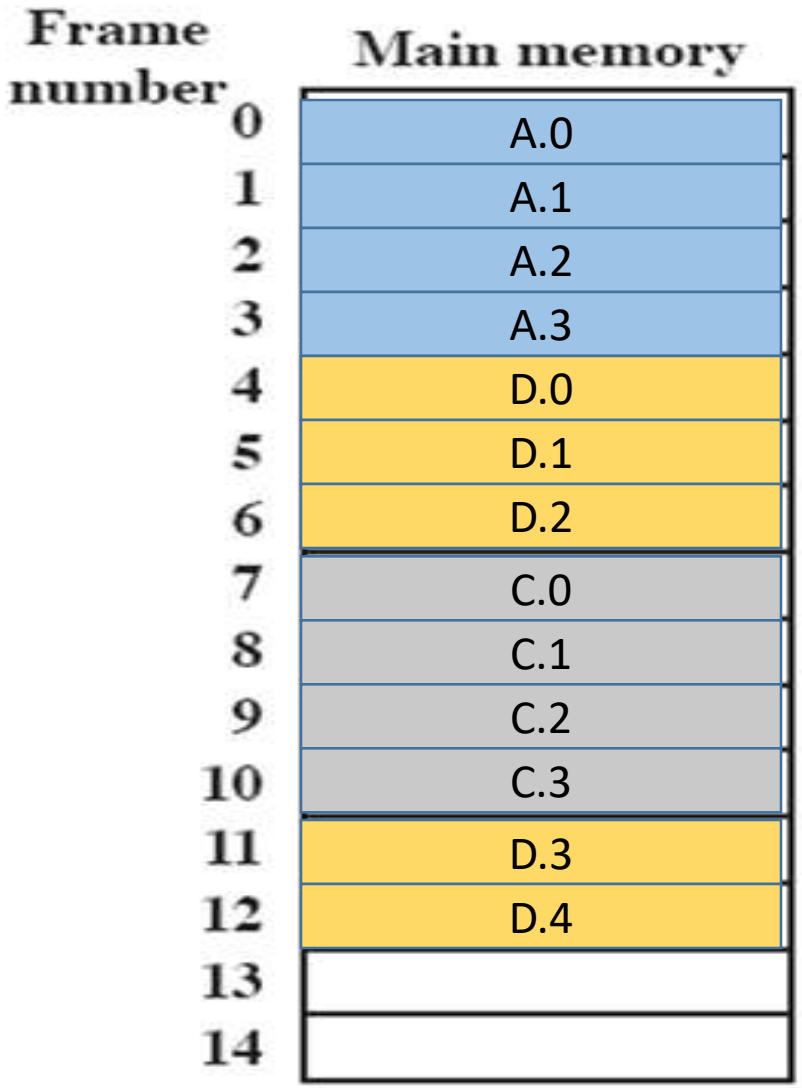
Figure 8.1 Paging Behavior

- Note that during the lifetime of the process, references are confined to a subset of pages.

Paging

- Each process has its own page table
- Each page table entry contains the frame number of the corresponding page in main memory
- Two extra bits are needed to indicate:
  - whether the page is in main memory or not
  - Whether the contents of the page has been altered since it was last loaded

Processes and Frames



Page Table

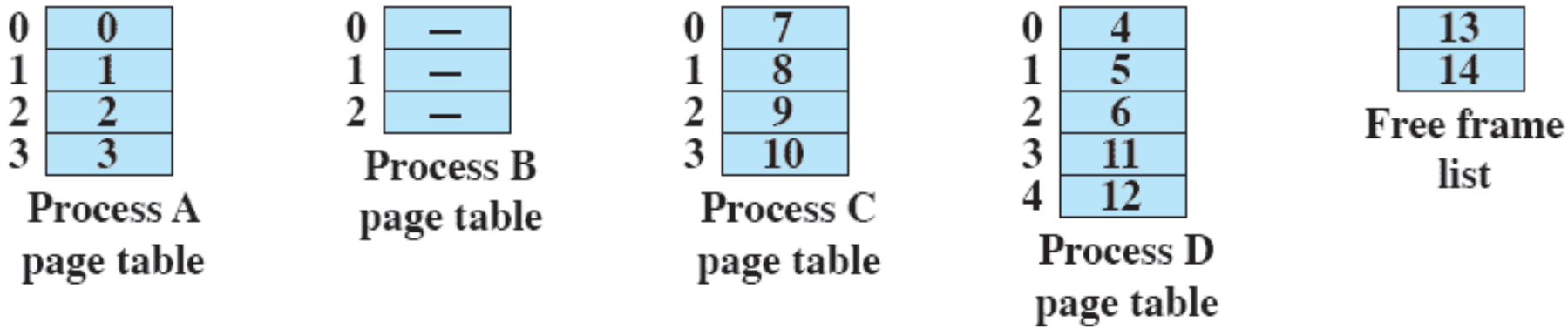


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

Address Translation

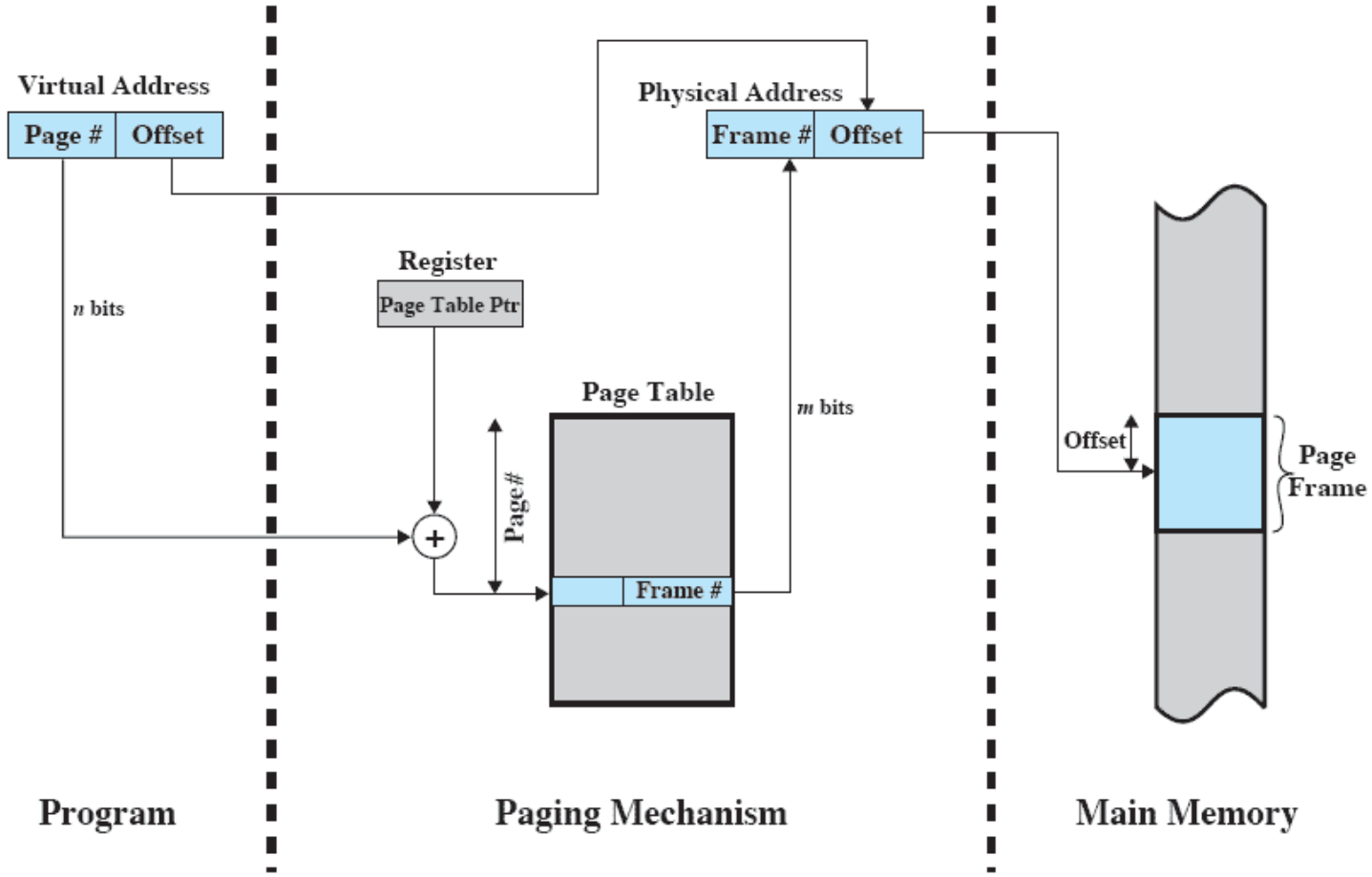
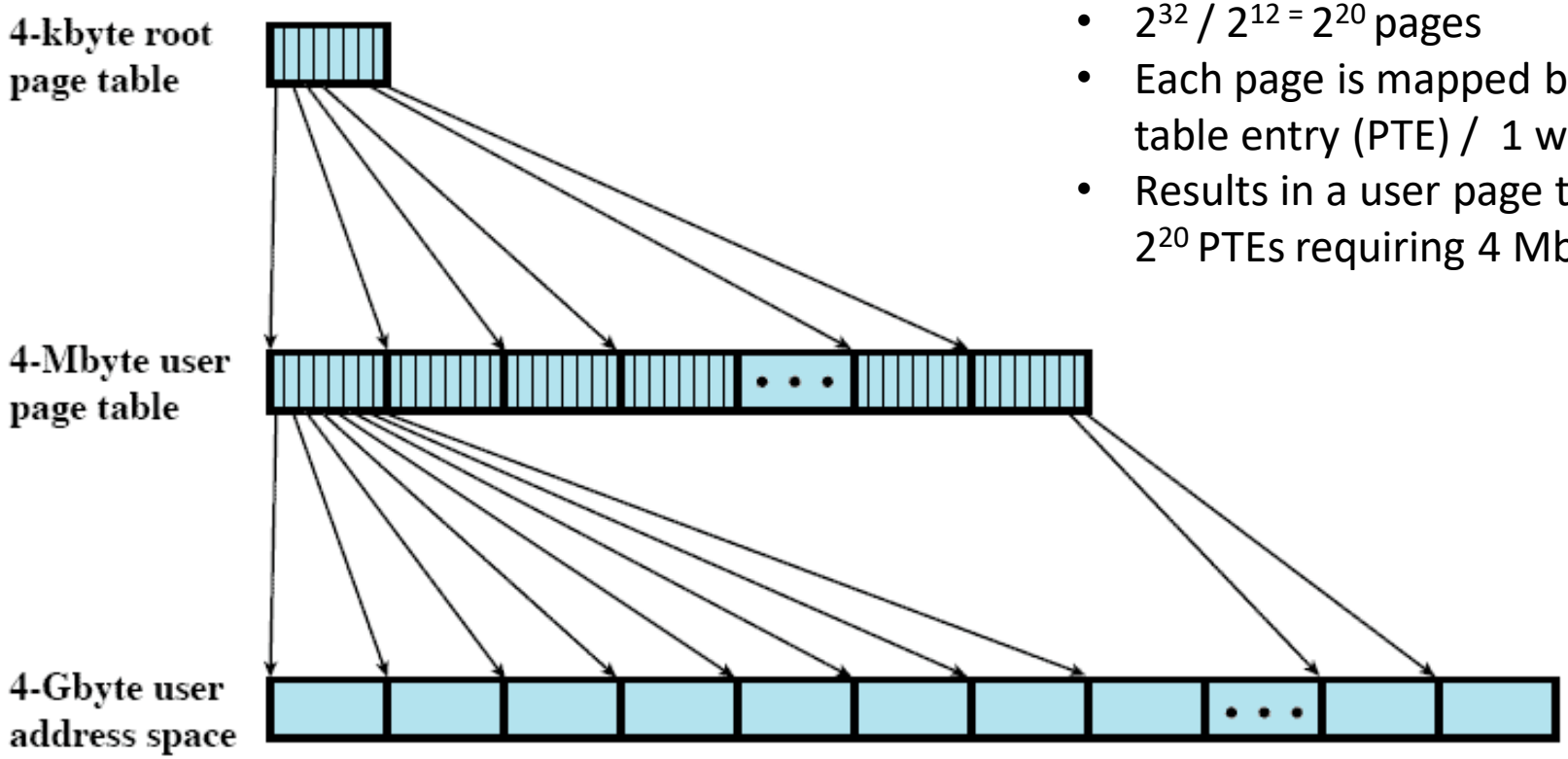


Figure 8.3 Address Translation in a Paging System

Two-Level Hierarchical Page Table



- $2^{32} / 2^{12} = 2^{20}$ pages
- Each page is mapped by a 4-byte page table entry (PTE) / 1 word
- Results in a user page table composed of 2^{20} PTEs requiring 4 Mbyte (2^{22}) bytes

Figure 8.4 A Two-Level Hierarchical Page Table

Address Translation for Hierarchical page table

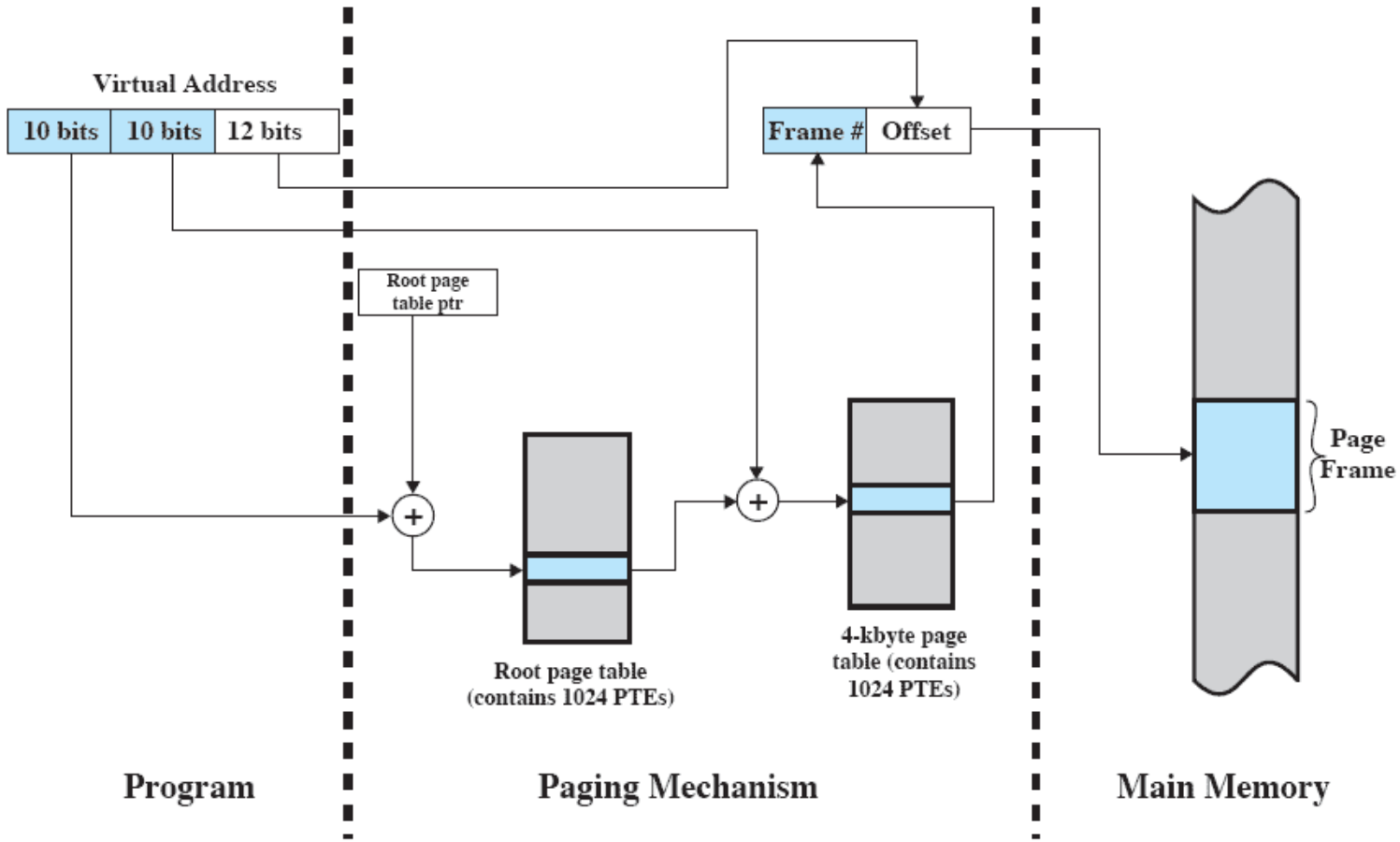


Figure 8.5 Address Translation in a Two-Level Paging System

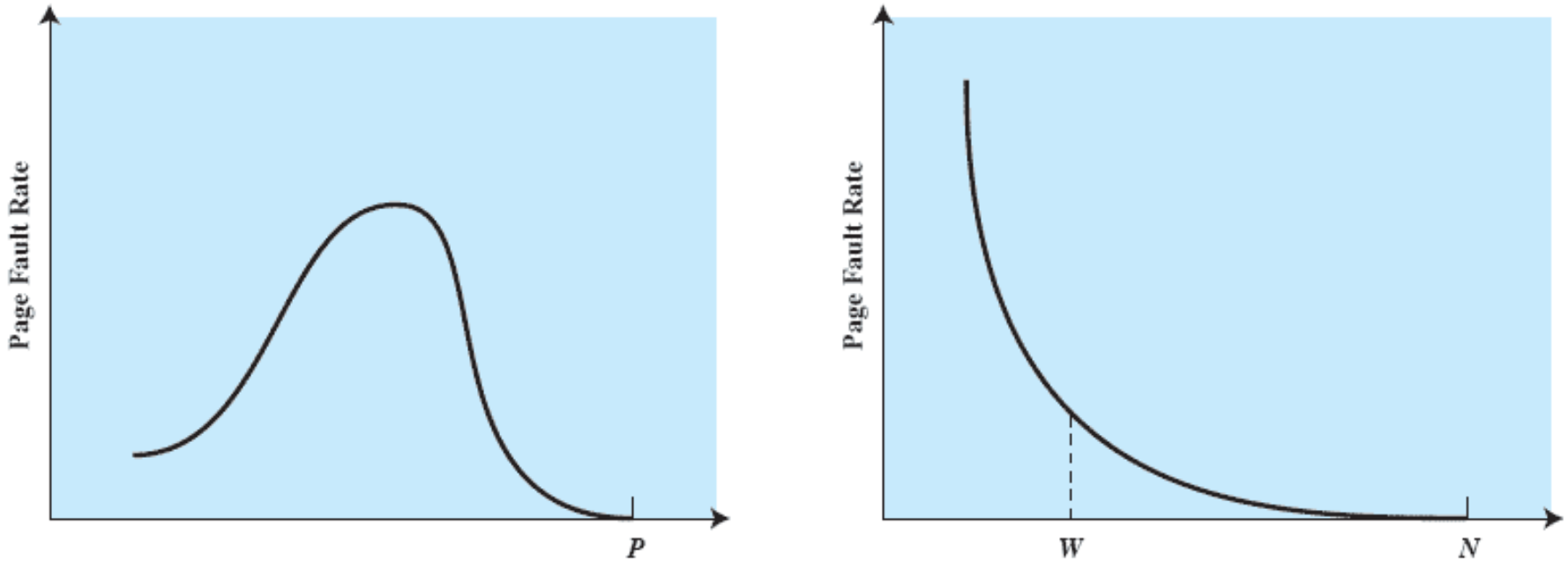
Page Size

- Smaller page size, less amount of internal fragmentation
- But Smaller page size, more pages required per process
  - More pages per process means larger page tables
- Larger page tables means large portion of page tables in virtual memory

Further complications to Page Size

- Small page size, large number of pages will be found in main memory
- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.
- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.

Page Size



(a) Page Size

(b) Number of Page Frames Allocated

P = size of entire process
W = working set size
N = total number of pages in process

Figure 8.11 Typical Paging Behavior of a Program

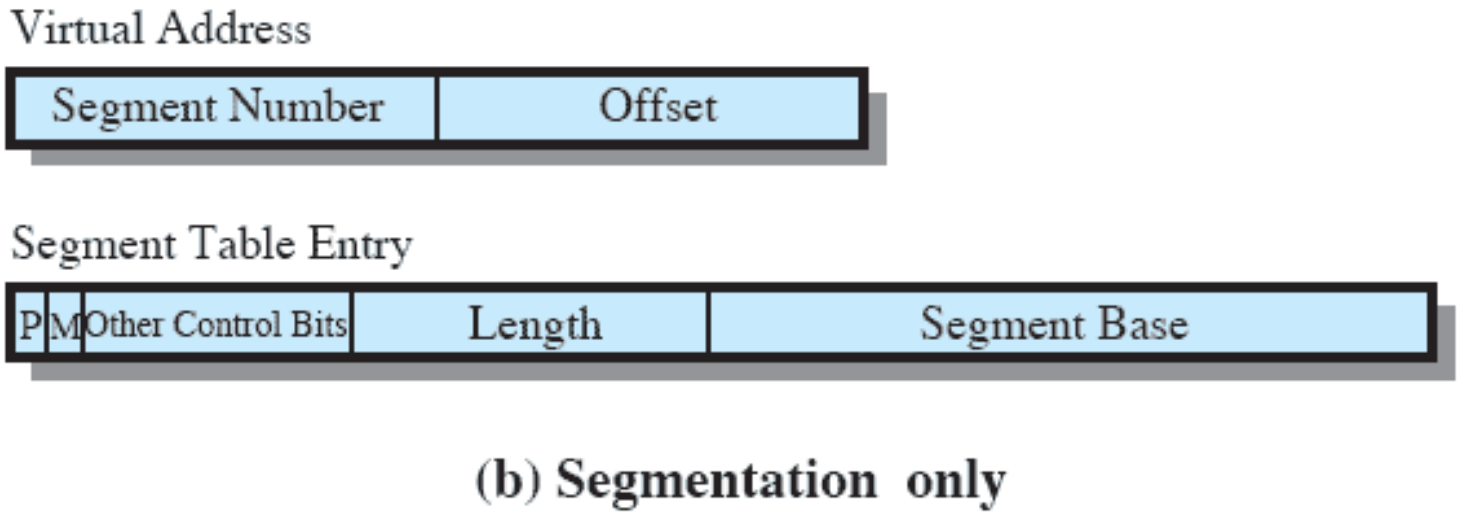
Segmentation

- A program can be subdivided into segments
  - Segments may vary in length
  - There is a maximum segment length
- Addressing consist of two parts
  - a segment number and
  - an offset
- Segmentation is similar to dynamic partitioning

Segment Organization

- Starting address corresponding segment in main memory
- Each entry contains the length of the segment
- A bit is needed to determine if segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory
- Same as in the page organization

Segment Table Entries



(b) Segmentation only

Address Translation in Segmentation

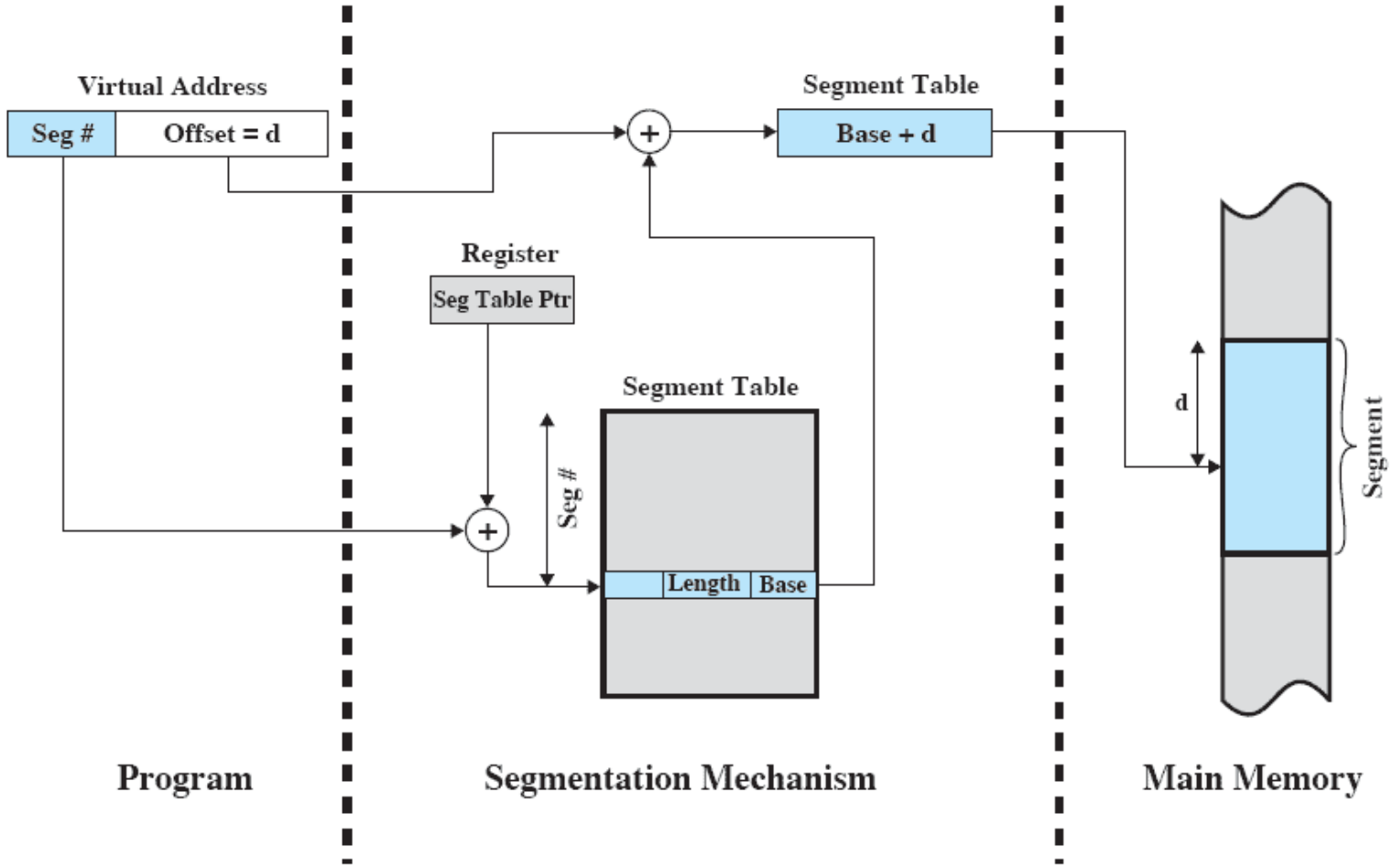
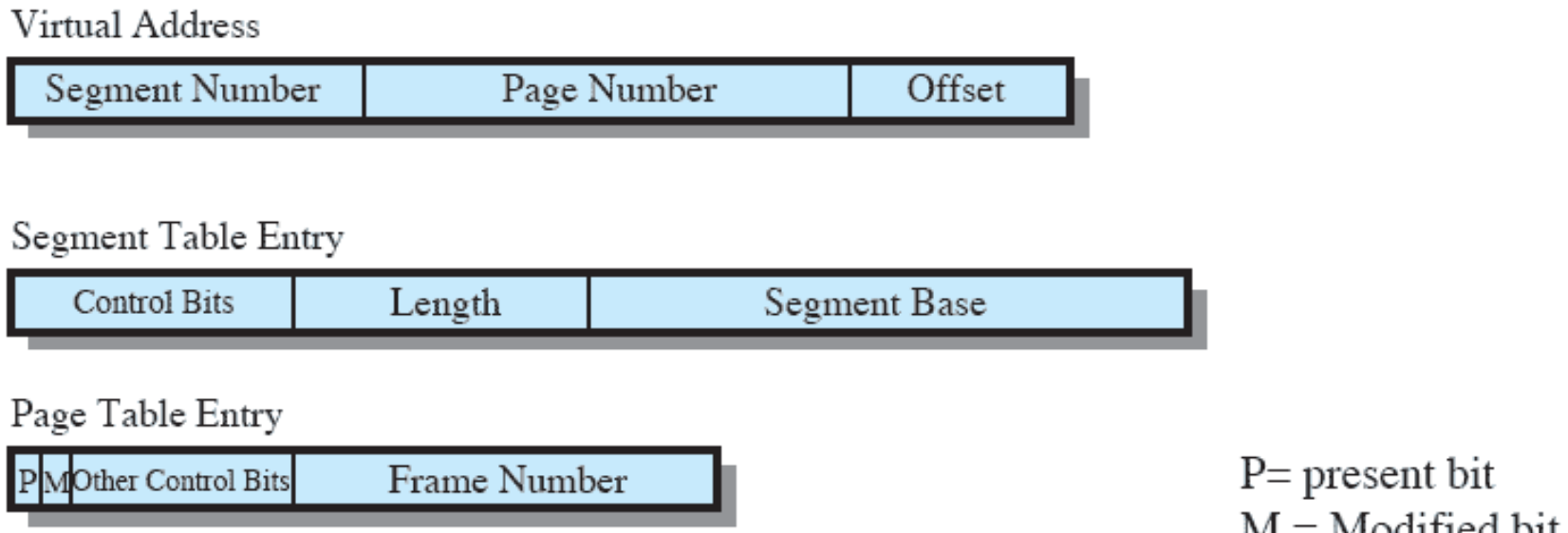


Figure 8.12 Address Translation in a Segmentation System

Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size pages

Combined Paging and Segmentation



P= present bit
M = Modified bit

(c) Combined segmentation and paging

Address Translation

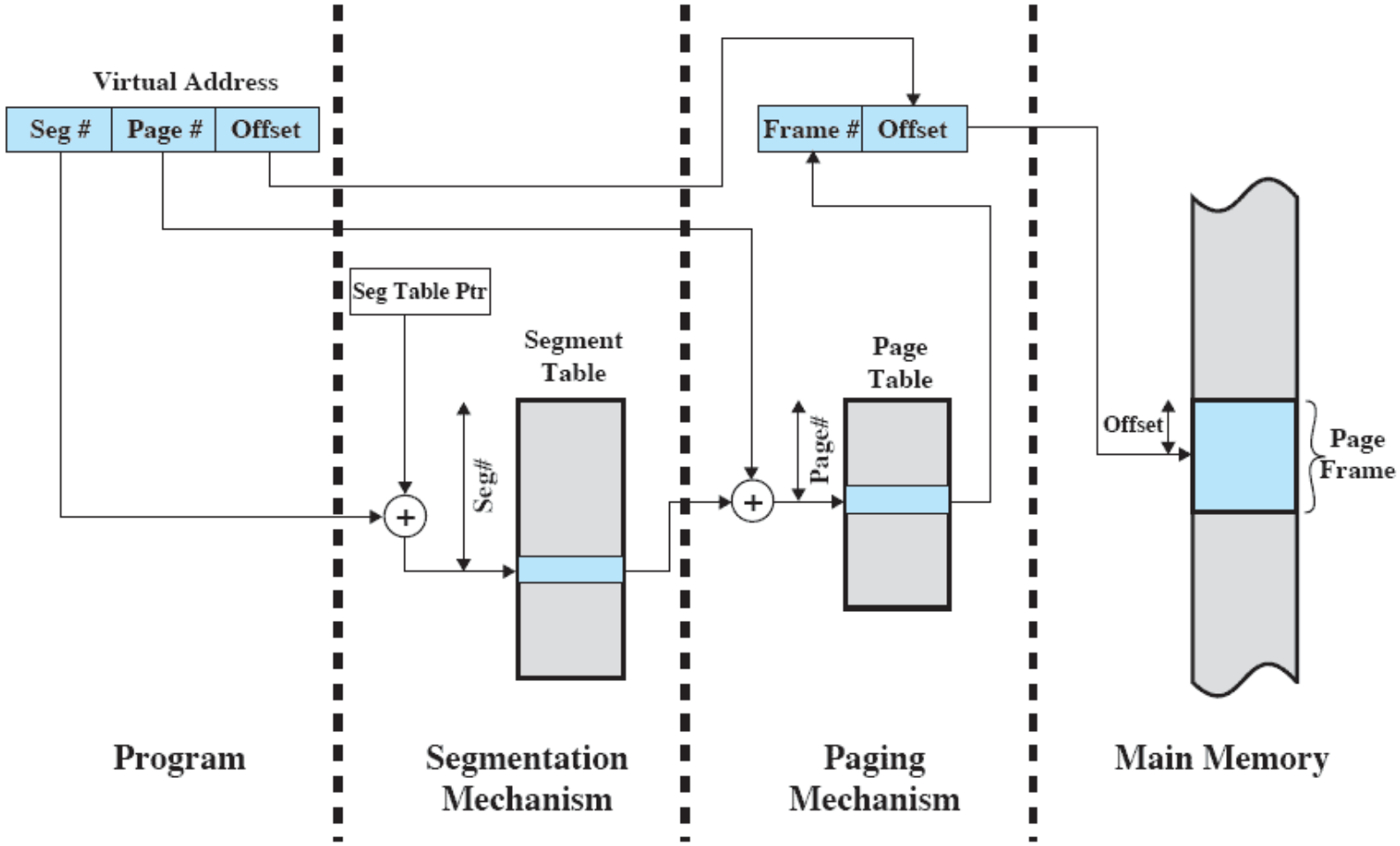


Figure 8.13 Address Translation in a Segmentation/Paging System

Roadmap

- Hardware and Control Structures
- Operating System Software

Memory Management Decisions

- Three fundamental areas of choice:
  - Whether or not to use virtual memory techniques
  - The use of paging or segmentation or both
  - The algorithms employed for various aspects of memory management

Key Design Elements

Table 8.4 Operating System Policies for Virtual Memory

Fetch Policy Demand Prepaging **Placement Policy** **Replacement Policy** Basic Algorithms Optimal Least recently used (LRU) First-in-first-out (FIFO) Clock Page buffering	**Resident Set Management** Resident set size Fixed Variable Replacement Scope Global Local **Cleaning Policy** Demand Precleaning **Load Control** Degree of multiprogramming

- Key aim: Minimise page faults
  - No definitive best policy

Fetch Policy

- Determines when a page should be brought into memory
- Two main types:
  - Demand Paging
  - Prepaging

Demand Paging and Prepaging

- **Demand paging**
  - only brings pages into main memory when a reference is made to a location on the page
  - Many page faults when process first started
- **Prepaging**
  - brings in more pages than needed
  - More efficient to bring in pages that reside contiguously on the disk
  - Don’t confuse with “swapping”

Placement Policy

- Determines where in real memory a process piece is to reside, e.g. best-fit, first-fit, next-fit
- Important in a segmentation system
- Paging or combined paging with segmentation hardware performs address translation

Replacement Policy

- When all of the frames in main memory are occupied and it is necessary to bring in a new page
- the replacement policy determines which page currently in memory is to be replaced.
- Page removed should be the page least likely to be referenced in the near future
  - How is that determined?
  - Principal of locality again
- Most policies predict the future behavior on the basis of past behavior

Basic Replacement Algorithms

- There are certain basic algorithms that are used for the selection of a page to replace, they include
  - Optimal
  - Least recently used (LRU)
  - First-in-first-out (FIFO)
  - Clock
- Examples

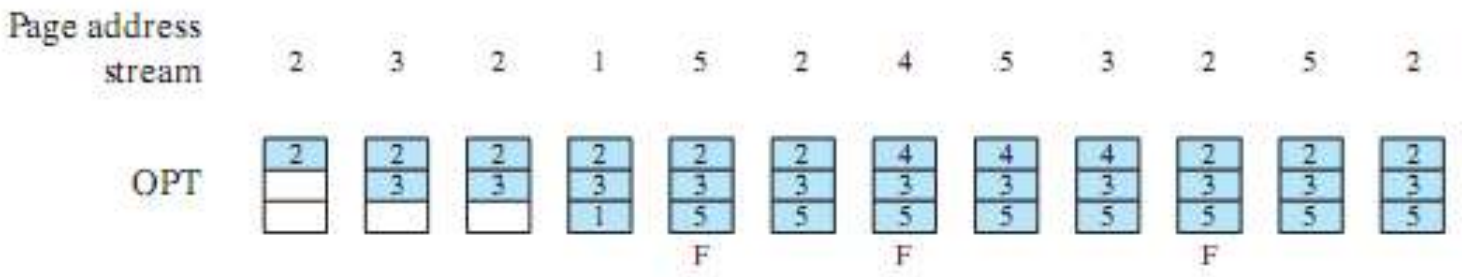
Examples

- An example of the implementation of these policies will use a page address stream formed by executing the program is
  - 2 3 2 1 5 2 4 5 3 2 5 2
- Which means that the first page referenced is 2,
  - the second page referenced is 3,
  - And so on.

Optimal policy

- Selects for replacement that page for which the time to the next reference is the longest
- But Impossible to have perfect knowledge of future events

Optimal Policy Example



F = page fault occurring after the frame allocation is initially filled

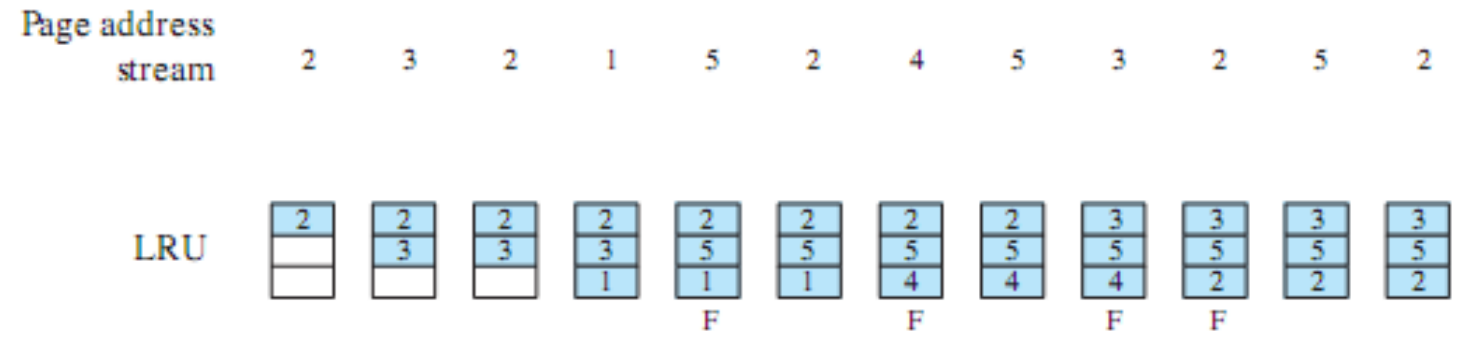
Figure 8.15 Behavior of Four Page Replacement Algorithms

- The optimal policy produces three page faults after the frame allocation has been filled.

Least Recently Used (LRU)

- Replaces the page that has not been referenced for the longest time
- By the principle of locality, this should be the page least likely to be referenced in the near future
- Difficult to implement
  - One approach is to tag each page with the time of last reference.
  - This requires a great deal of overhead.

LRU Example



F= page fault occurring after the frame allocation is initially filled

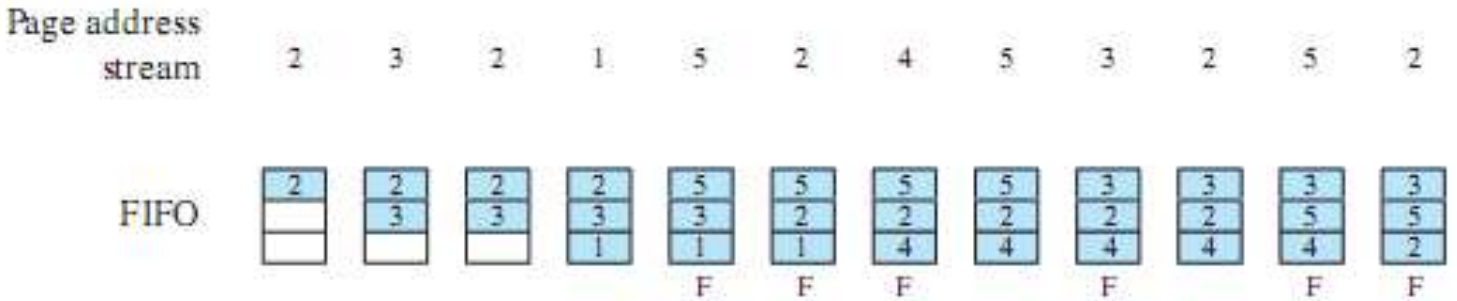
Figure 8.15 Behavior of Four Page Replacement Algorithms

- The LRU policy does nearly as well as the optimal policy.
  - In this example, there are four page faults

First-in, first-out (FIFO)

- Treats page frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
  - Simplest replacement policy to implement
- Page that has been in memory the longest is replaced
  - But, these pages may be needed again very soon if it hasn't truly fallen out of use

FIFO Example



F = page fault occurring after the frame allocation is initially filled

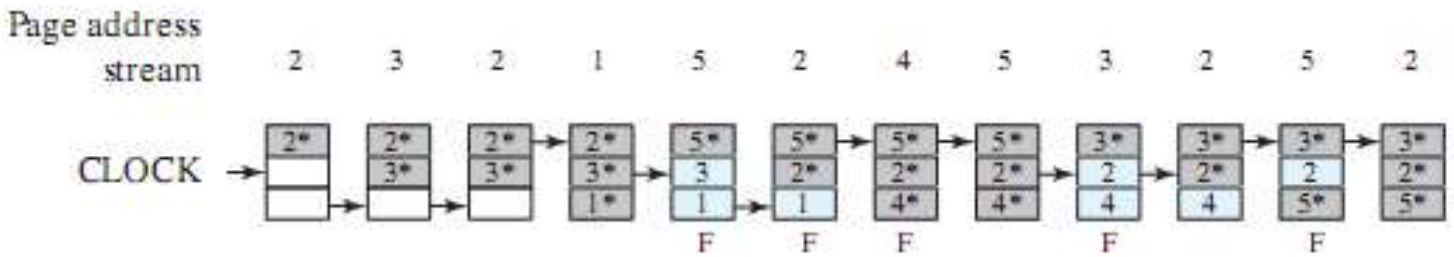
Figure 8.15 **Behavior of Four Page Replacement Algorithms**

- The FIFO policy results in six page faults.
  - Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

Clock Policy

- Uses and additional bit called a “use bit”
- When a page is first loaded in memory or referenced, the use bit is set to 1
- When it is time to replace a page, the OS scans the set flipping all 1’s to 0
- The first frame encountered with the use bit already set to 0 is replaced.

Clock Policy Example



F = page fault occurring after the frame allocation is initially filled

Figure 8.15 **Behavior of Four Page Replacement Algorithms**

- Note that the clock policy is adept at protecting frames 2 and 5 from replacement.

Combined Examples

Figure 8.15 **Behavior of Four Page Replacement Algorithms**

Comparison

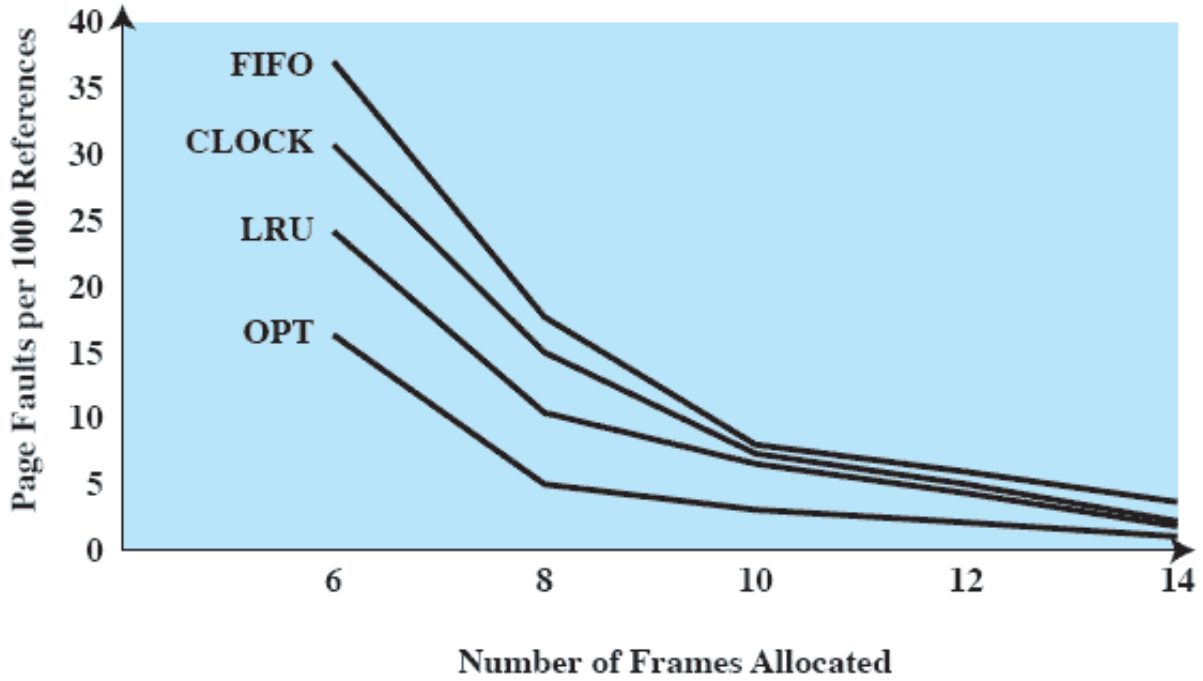


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

Resident Set Management

- The OS must decide how many pages to bring into main memory
  - The smaller the amount of memory allocated to each process, the more processes that can reside in memory.
  - Small number of pages loaded increases page faults.
  - Beyond a certain size, further allocations of pages will not affect the page fault rate.

Resident Set Size

- Fixed-allocation
  - Gives a process a fixed number of pages within which to execute
  - When a page fault occurs, one of the pages of that process must be replaced
- Variable-allocation
  - Number of pages allocated to a process varies over the lifetime of the process

Replacement Scope

- The scope of a replacement strategy can be categorized as *global* or *local*.
  - Both types are activated by a page fault when there are no free page frames.
  - A local replacement policy chooses only among the resident pages of the process that generated the page fault
  - A global replacement policy considers all unlocked pages in main memory

Fixed Allocation, Local Scope

- Decide ahead of time the amount of memory allocation to be given to a process
- If allocation is too small, there will be a high page fault rate
- If allocation is too large there will be too few programs in main memory
  - Increased processor idle time or
  - Increased swapping.

Variable Allocation, Local Scope

- When new process added, allocate number of page frames based on application type, program request, or other criteria
- When page fault occurs, select page from among the resident set of the process that suffers the fault
- Reevaluate allocation from time to time

Variable Allocation, Global Scope

- Easiest to implement
  - Adopted by many operating systems
- Operating system keeps list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no free frame, replaces one from another process
  - The challenge is, which frame of process to replace.

Resident Set Management Summary

Table 8.5 Resident Set Management

	Local Replacement	Global Replacement
Fixed Allocation	• Number of frames allocated to process is fixed. • Page to be replaced is chosen from among the frames allocated to that process.	• Not possible.
Variable Allocation	• The number of frames allocated to a process may be changed from time to time, to maintain the working set of the process. • Page to be replaced is chosen from among the frames allocated to that process.	• Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

Cleaning Policy

- A cleaning policy is concerned with determining when a modified page should be written out to secondary memory.
- Demand cleaning
  - A page is written out only when it has been selected for replacement
- Precleaning
  - Pages are written out in batches

Cleaning Policy

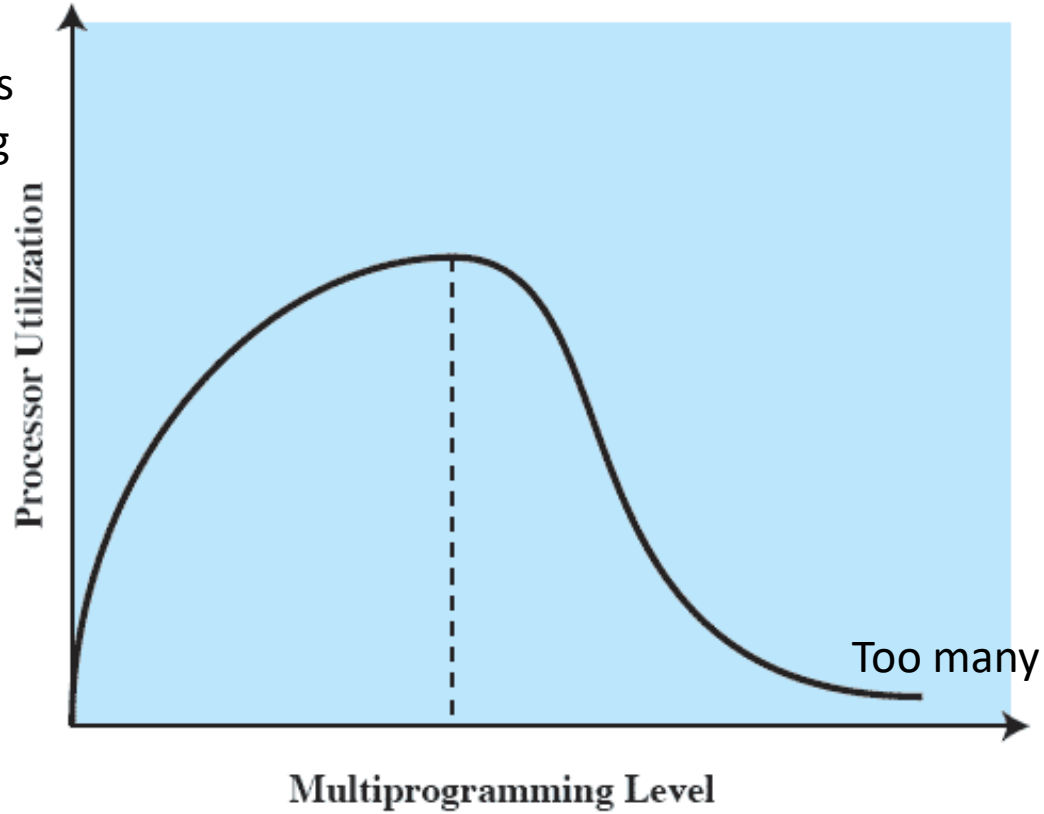
- Best approach uses page buffering
- Replaced pages are placed in two lists
  - Modified and unmodified
- Pages in the modified list are periodically written out in batches
- Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page

Load Control

- Determines the number of processes that will be resident in main memory
  - especially in *multiprogramming* environment
- Two things to consider:
  - Too few processes, many occasions when all processes will be blocked, and much time will be spent in swapping
  - Too many processes will lead to thrashing

Multiprogramming

Too few processes leads to swapping



Too many processes leads to trashing

Figure 8.21 Multiprogramming Effects

Process Suspension

- If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be suspended (swapped out).
- Six categories of suspension:
  - Lowest priority process
  - Faulting process
    - This process does not have its working set in main memory so it will be blocked anyway
  - Last process activated
    - This process is least likely to have its working set resident
  - Process with smallest resident set
    - This process requires the least future effort to reload
  - Largest process
    - Obtains the most free frames
  - Process with the largest remaining execution window

Thank You