



OPERATING SYSTEMS

Week 12

Much of the material on these slides comes from the recommended textbook by William Stallings

Detailed content

Weekly program

- ✓ Week 1 – Operating System Overview
- ✓ Week 2 – Processes and Threads
- ✓ Week 3 – Scheduling
- ✓ Week 4 – Real-time System Scheduling and Multiprocessor Scheduling
- ✓ Week 5 – Concurrency: Mutual Exclusion and Synchronization
- ✓ Week 6 – Concurrency: Deadlock and Starvation
- ✓ Week 7 – Memory Management
- ✓ Week 8 – Memory Management II
- ✓ Week 9 – Disk and I/O Scheduling
- ✓ Week 10 – File Management
- ✓ Week 11 – Security and Protection
- **Week 12 – Revision of the course**
- Week 13 – Extra revision (if needed)

Week 12 Lecture Outline

Revision

- ✓ Week 1 – Operating System Overview
- ✓ Week 2 – Processes and Threads
- ✓ Week 3 – Scheduling
- ✓ Week 4 – Real-time System Scheduling and Multiprocessor Scheduling
- ✓ Week 5 – Concurrency: Mutual Exclusion and Synchronization
- ✓ Week 6 – Concurrency: Deadlock and Starvation
- ✓ Week 7 – Memory Management
- ✓ Week 8 – Memory Management II
- ✓ Week 9 – Disk and I/O Scheduling
- ✓ Week 10 – File Management
- ✓ Week 11 – Security and Protection

Operating System Overview

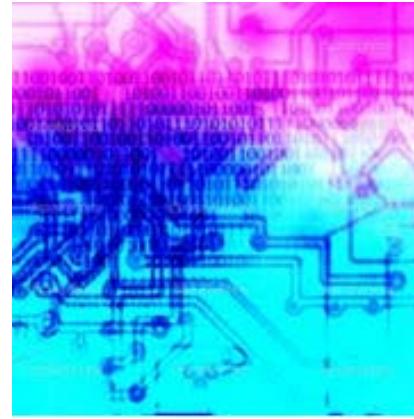
WEEK 1

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

What is an OS?

- Operating System as an Abstract Machine
- Operating System as a Service Provider
 - Program development and execution
 - Access I/O devices
 - Controlled access to files
 - System access
 - Error detection and response
 - Accounting
- Operating System as a Resource Manager
- Still OS is a Software

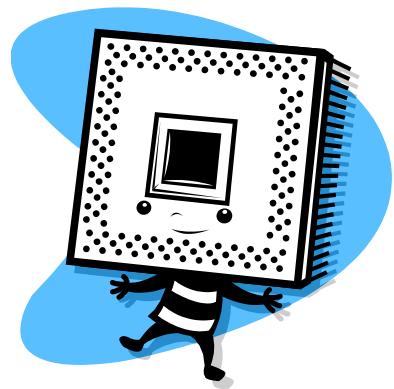


Operating System Objectives

- **Convenience**
 - Makes a computer more convenient to use
- **Efficiency**
 - Allows the computer resources to be used in an efficient manner
- **Ability to evolve**
 - Should permit the effective development, testing and introduction of new system functions without interfering with service.

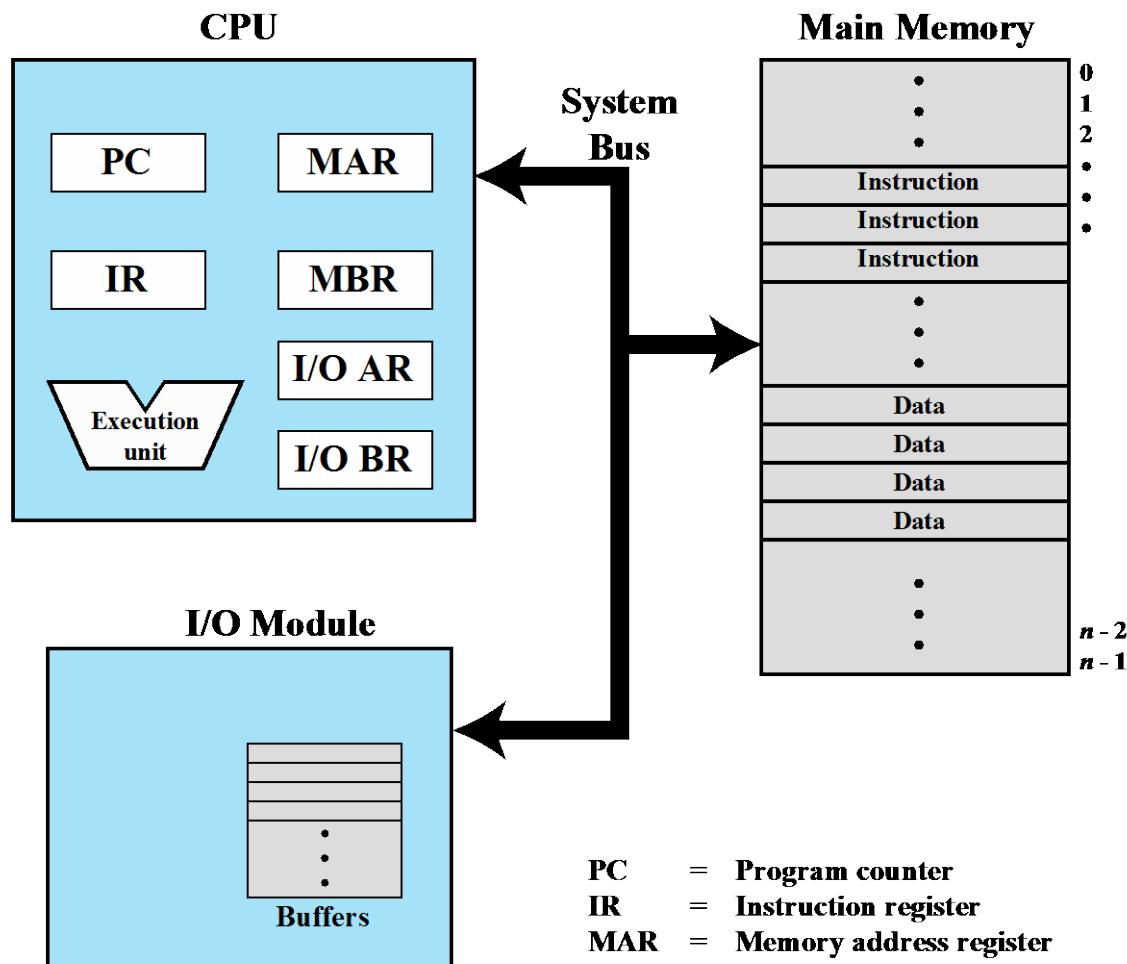
Hardware Review

7



29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au



- PC** = Program counter
- IR** = Instruction register
- MAR** = Memory address register
- MBR** = Memory buffer register
- I/O AR** = Input/output address register
- I/O BR** = Input/output buffer register

Instruction Execution

- A program consists of a set of instructions stored in memory

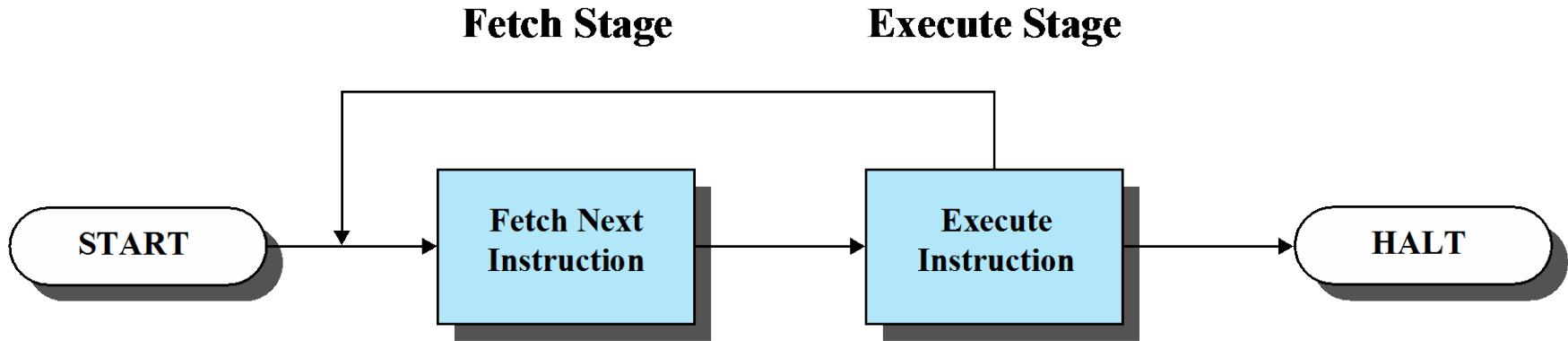


Figure 1.2 Basic Instruction Cycle



(a) Instruction format



(b) Integer format

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

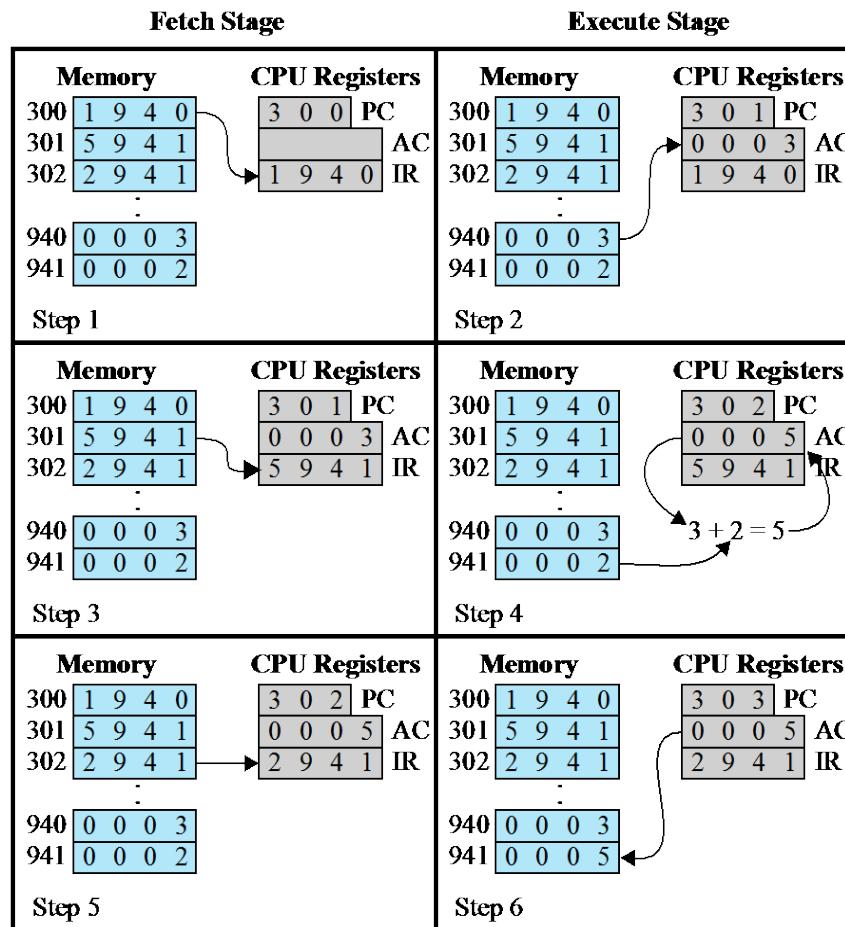
(c) Internal CPU registers

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

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

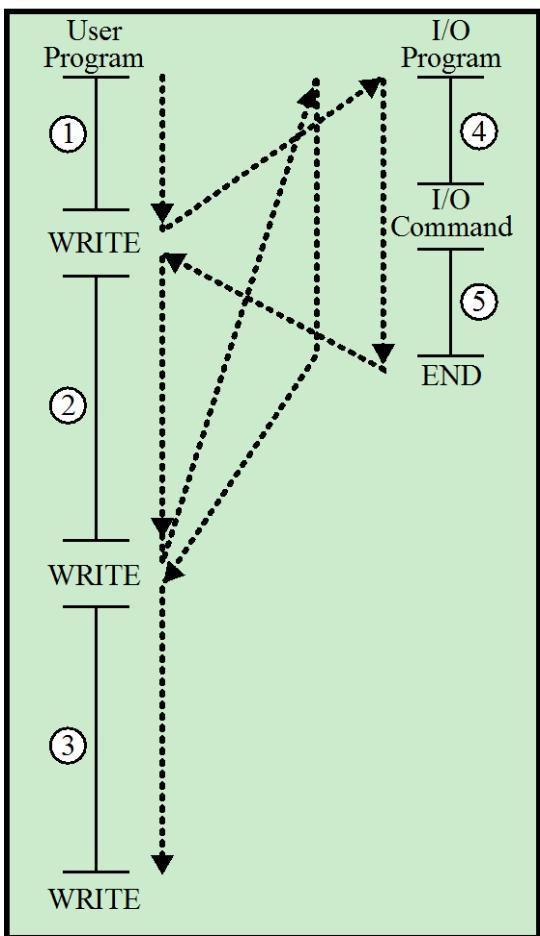


**Figure 1.4 Example of Program Execution
 (contents of memory and registers in hexadecimal)**

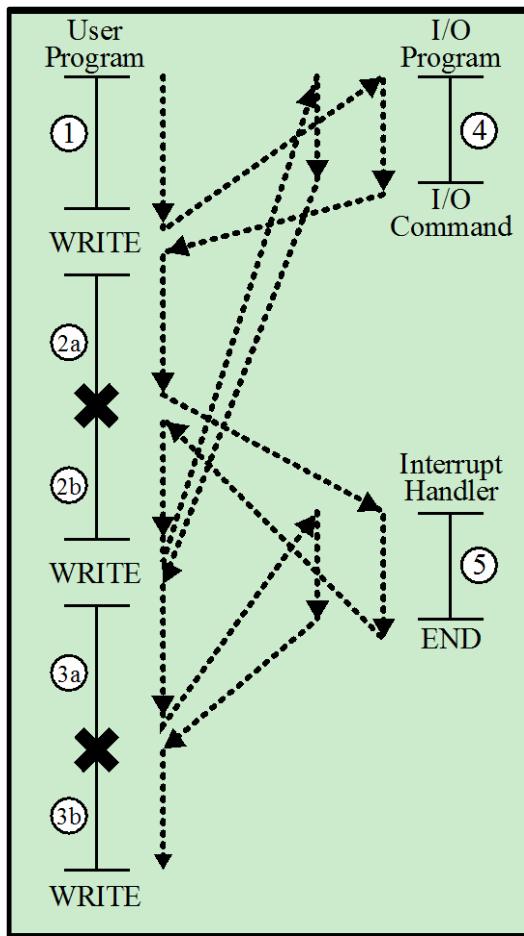
Interrupts

- Interrupt the normal sequencing of the processor
- Provided to improve processor utilization
 - most I/O devices are slower than the processor
 - processor must pause to wait for device
 - wasteful use of the processor

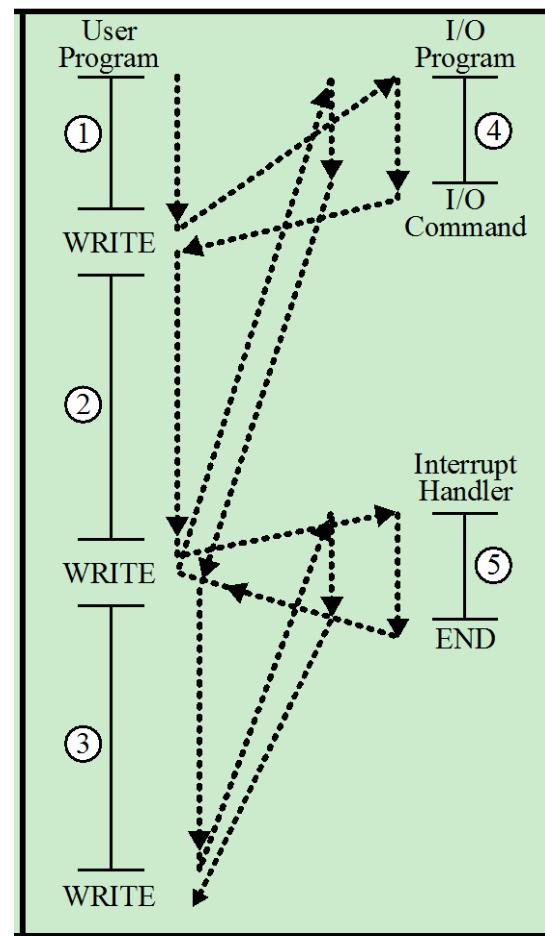




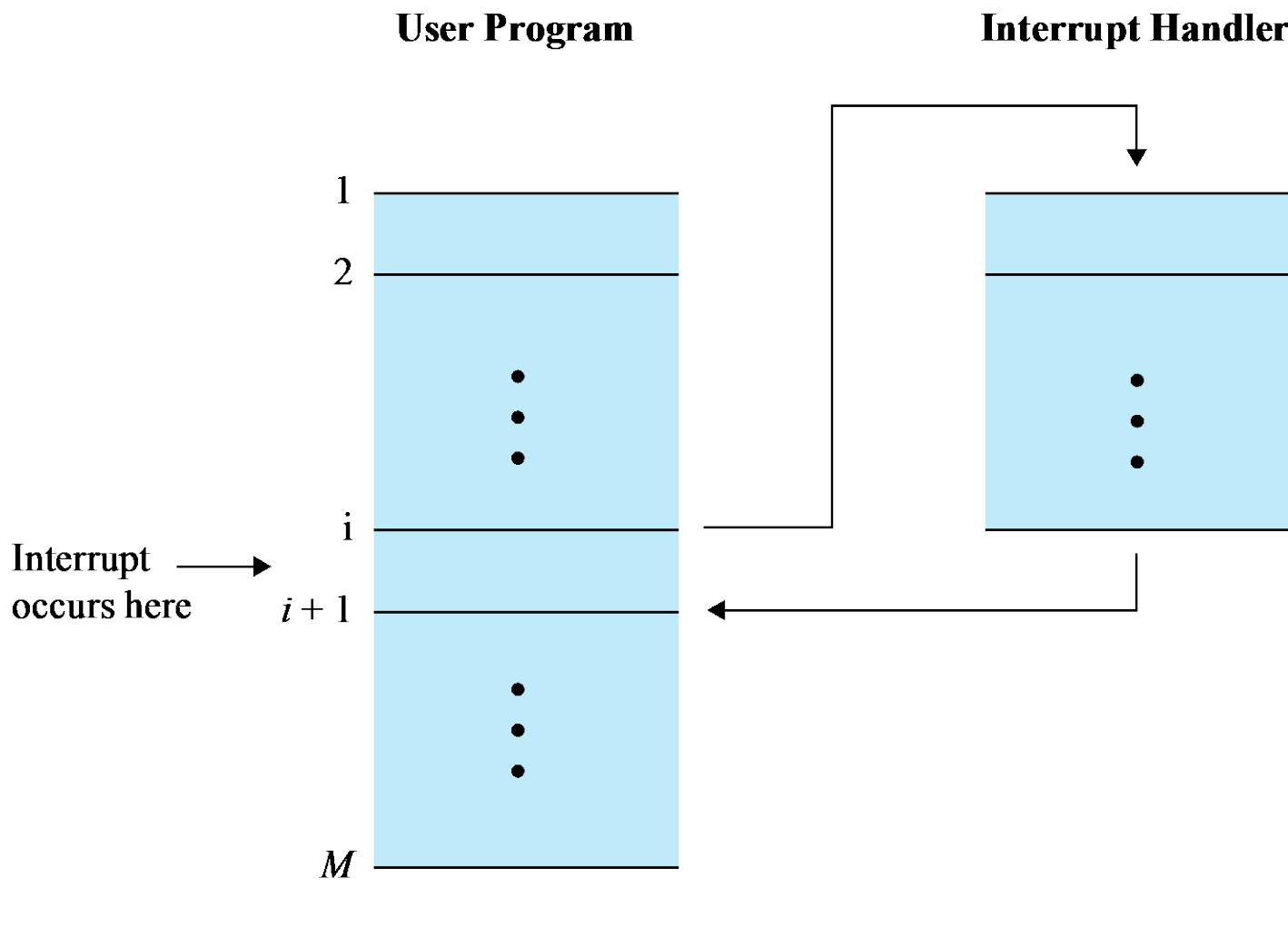
(a) No interrupts



(b) Interrupts; short I/O wait



(c) Interrupts; long I/O wait



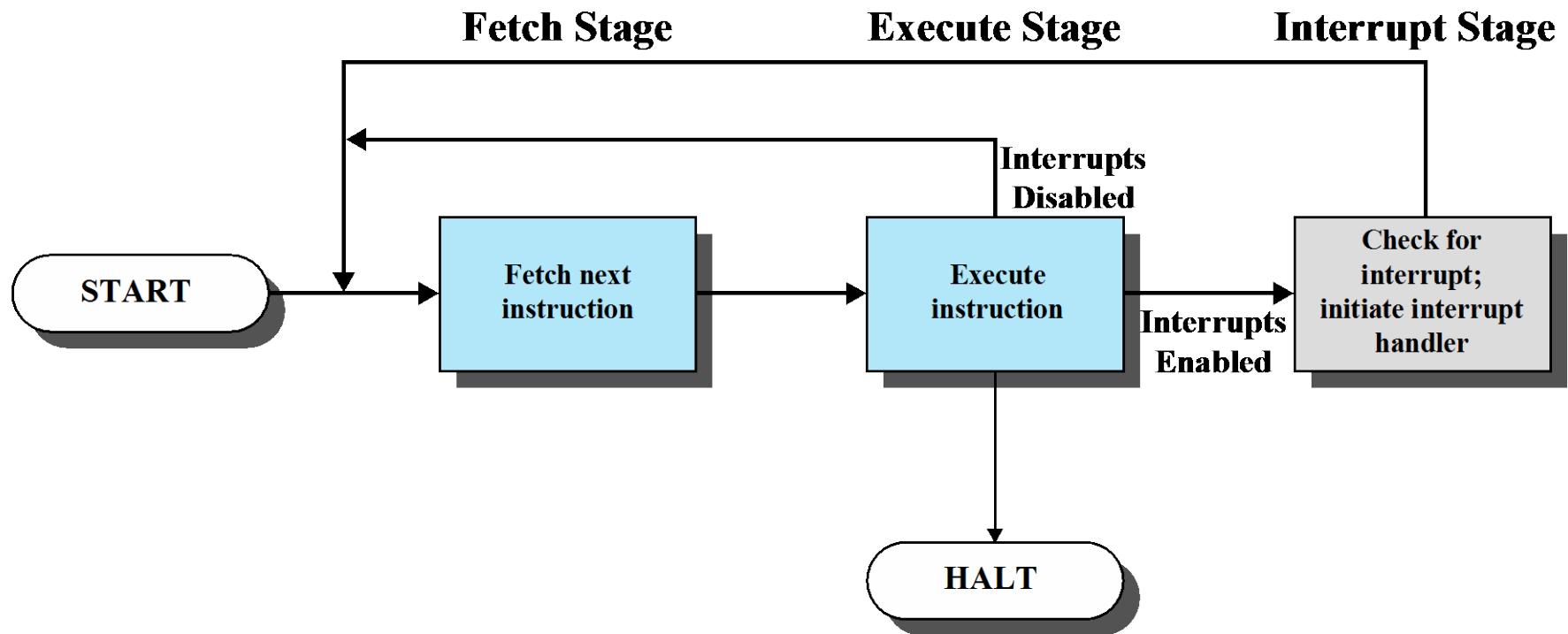
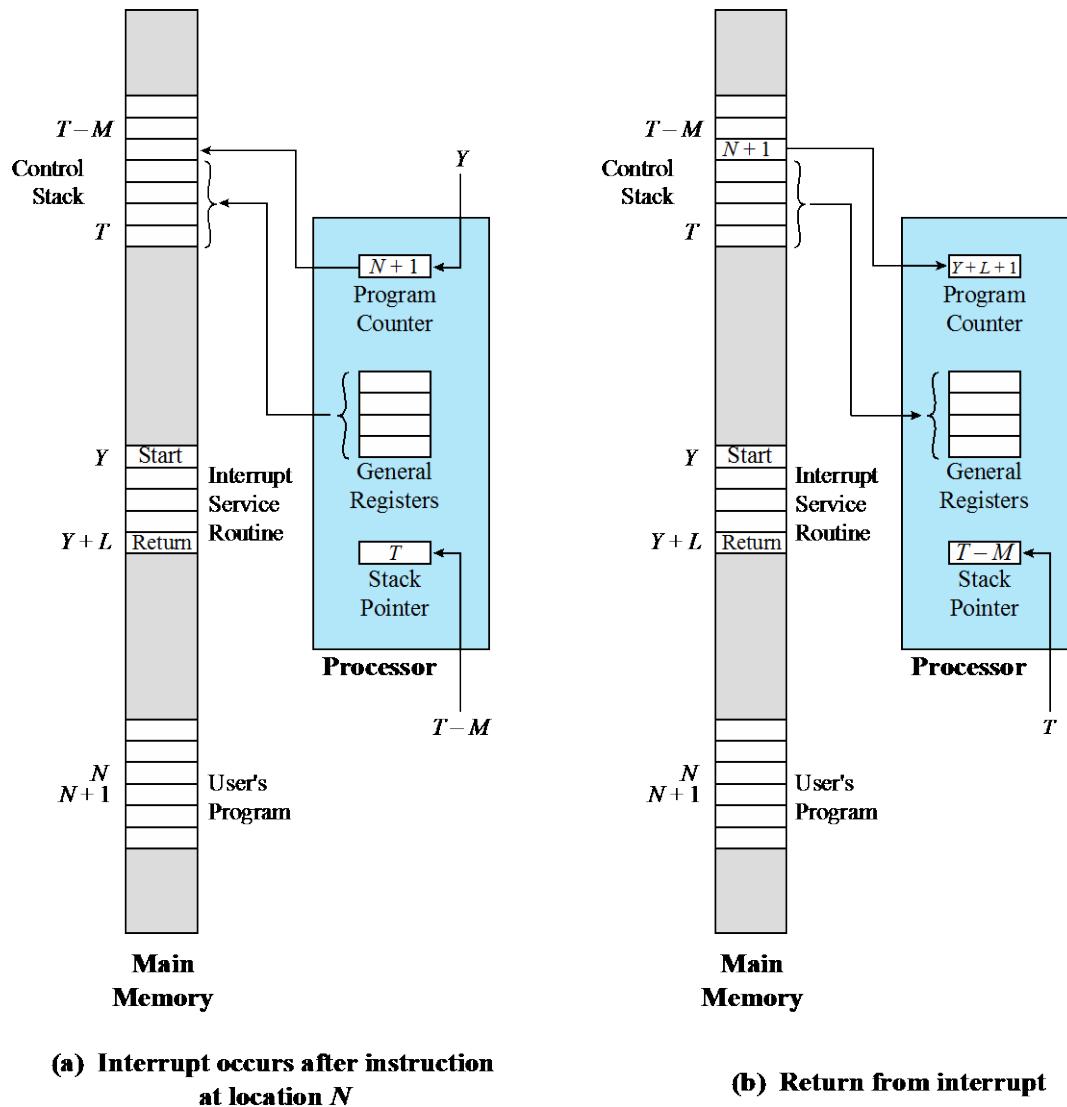
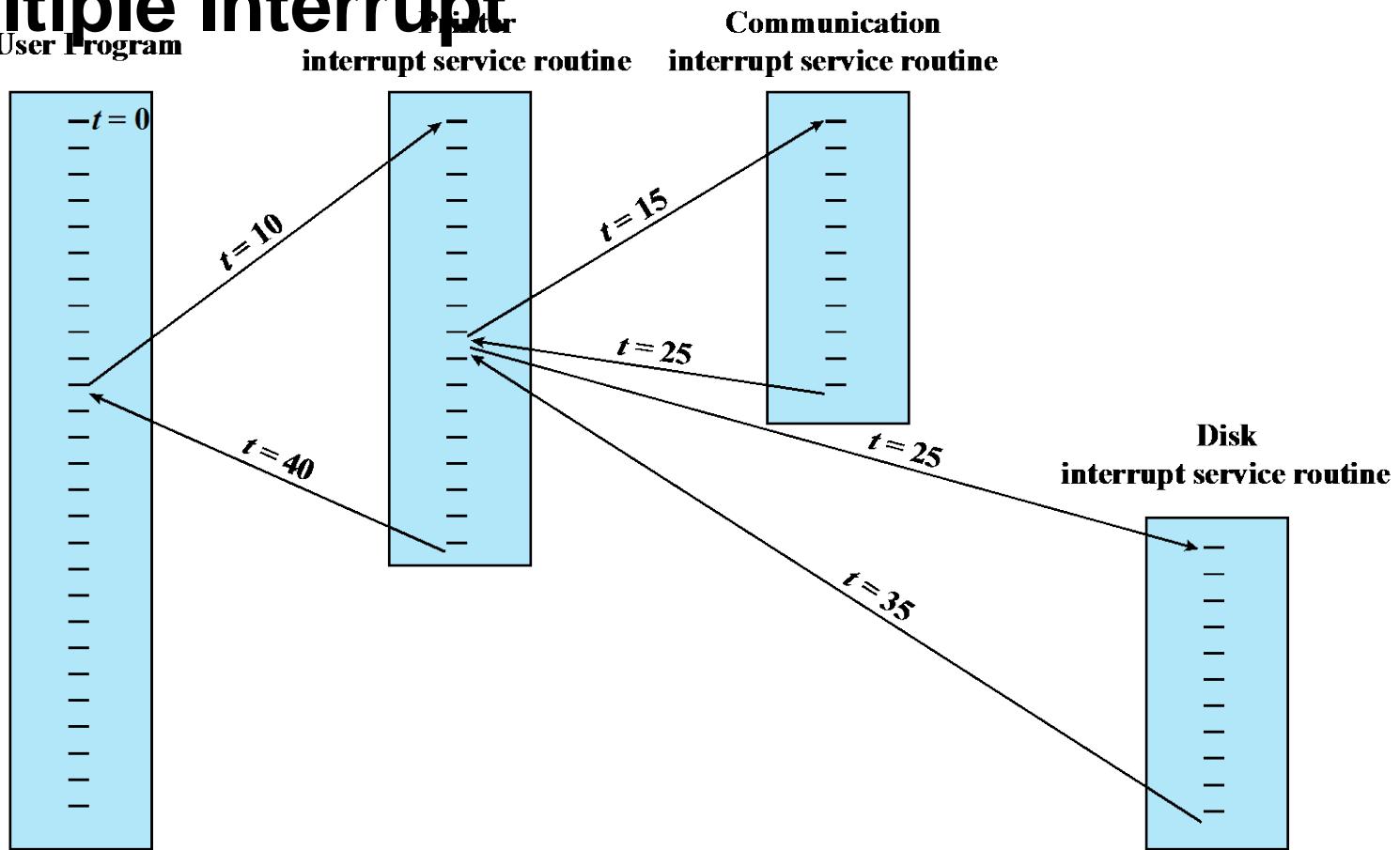


Figure 1.7 Instruction Cycle with Interrupts



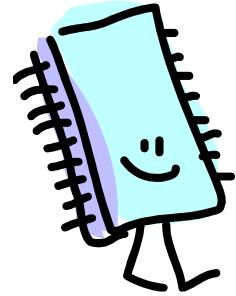
Multiple Interrupt

User Program



Memory Hierarchy

- Major constraints in memory
 - amount
 - speed
 - expense
- Memory must be able to keep up with the processor
- Cost of memory must be reasonable in relationship to the other components



The Memory Hierarchy

- Going down the hierarchy:
 - decreasing cost per bit
 - increasing capacity
 - increasing access time
 - decreasing frequency of access to the memory by the processor

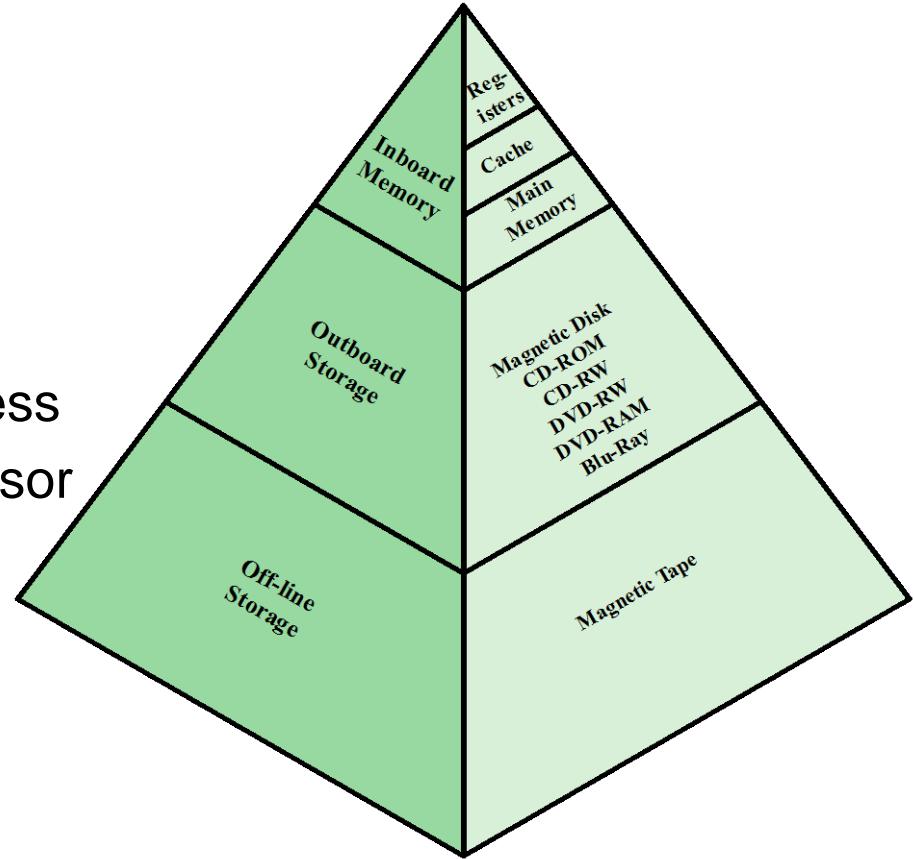


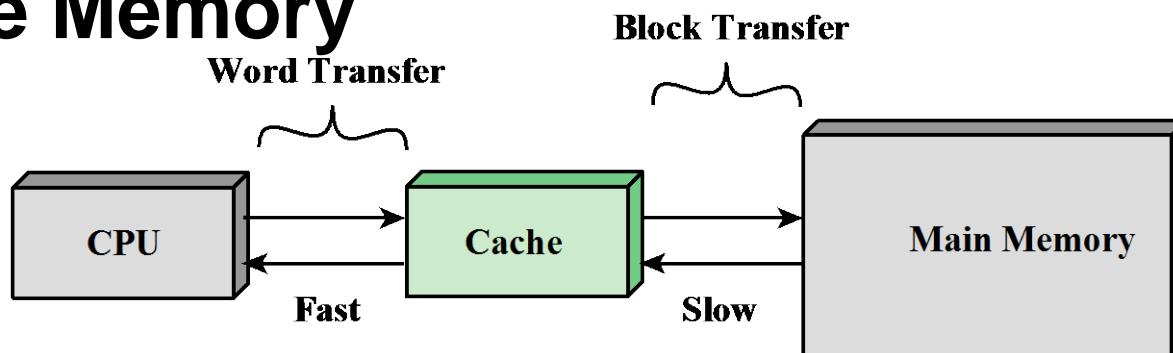
Figure 1.14 The Memory Hierarchy

Principle of Locality

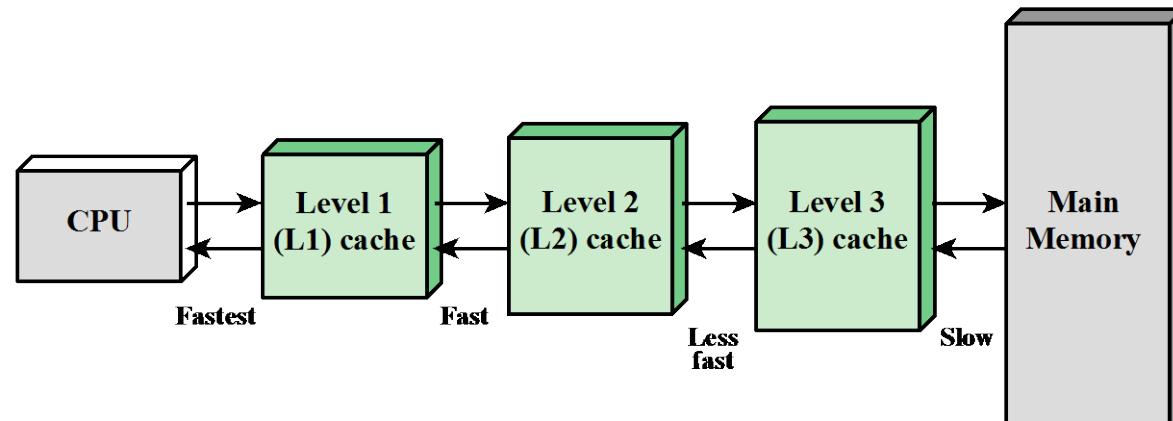
20

- Memory references by the processor tend to cluster
- Data is organized so that the percentage of accesses to each successively lower level is substantially less than that of the level above
- Can be applied across more than two levels of memory

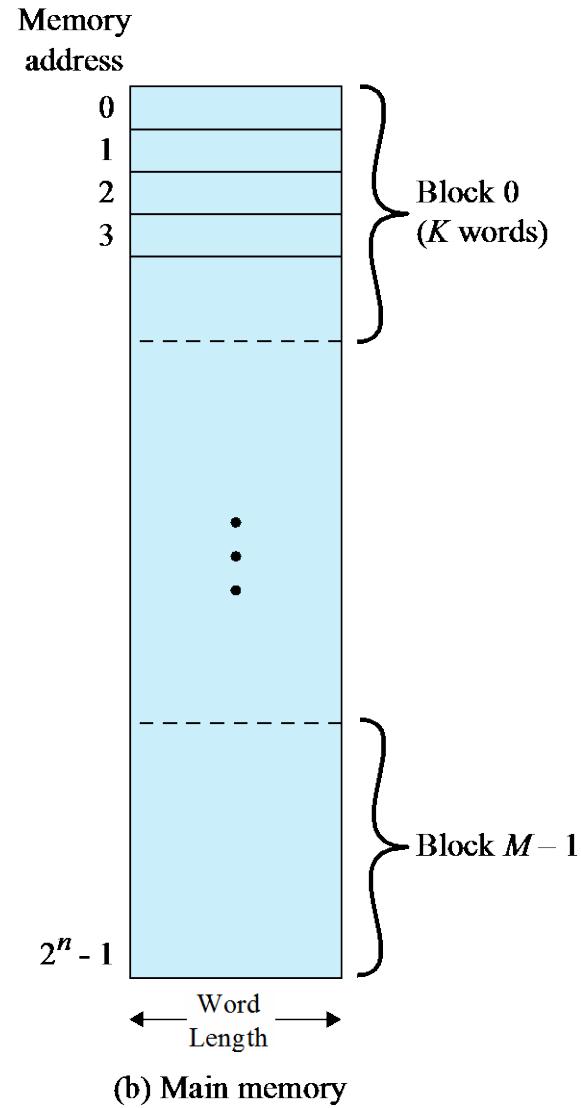
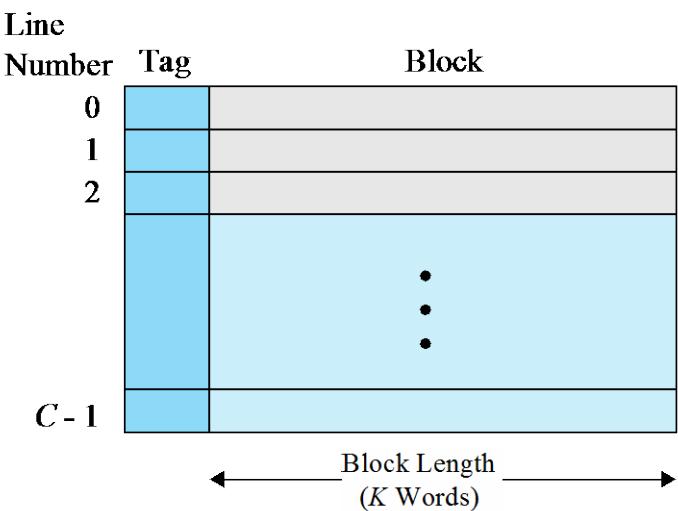
Cache Memory

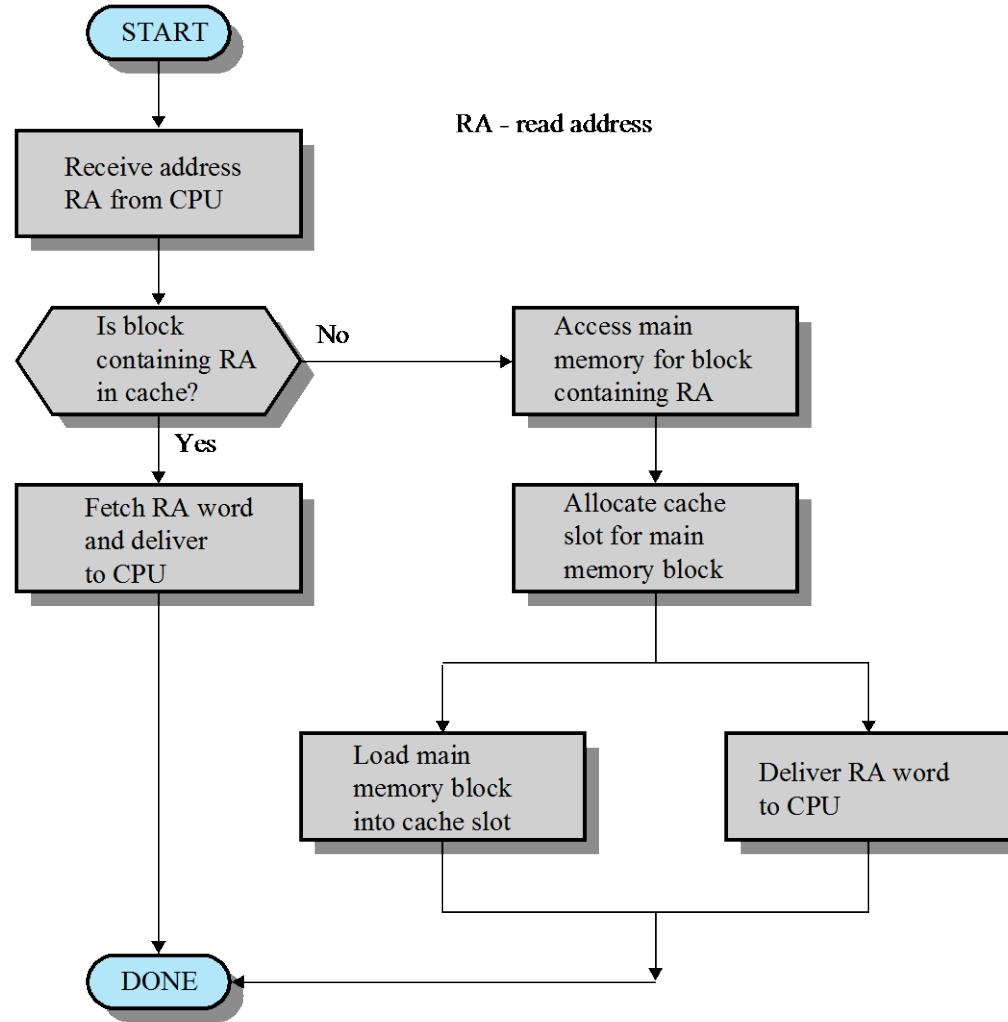


(a) Single cache



(b) Three-level cache organization



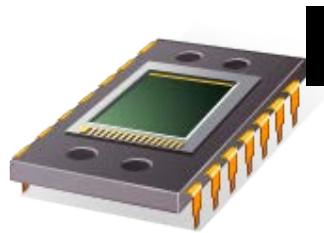
**Figure 1.18 Cache Read Operation**

I/O Techniques

- When the processor encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module
- Three techniques are possible for I/O operations:
 - Programmed I/O
 - Interrupt-Driven I/O
 - Direct Memory Access (DMA)

SMP VS Multiple Core

25



- SMP: A stand-alone computer system with the following characteristics with two or more similar processors of comparable capability
- Combines two or more processors (cores) on a single piece of silicon (die)

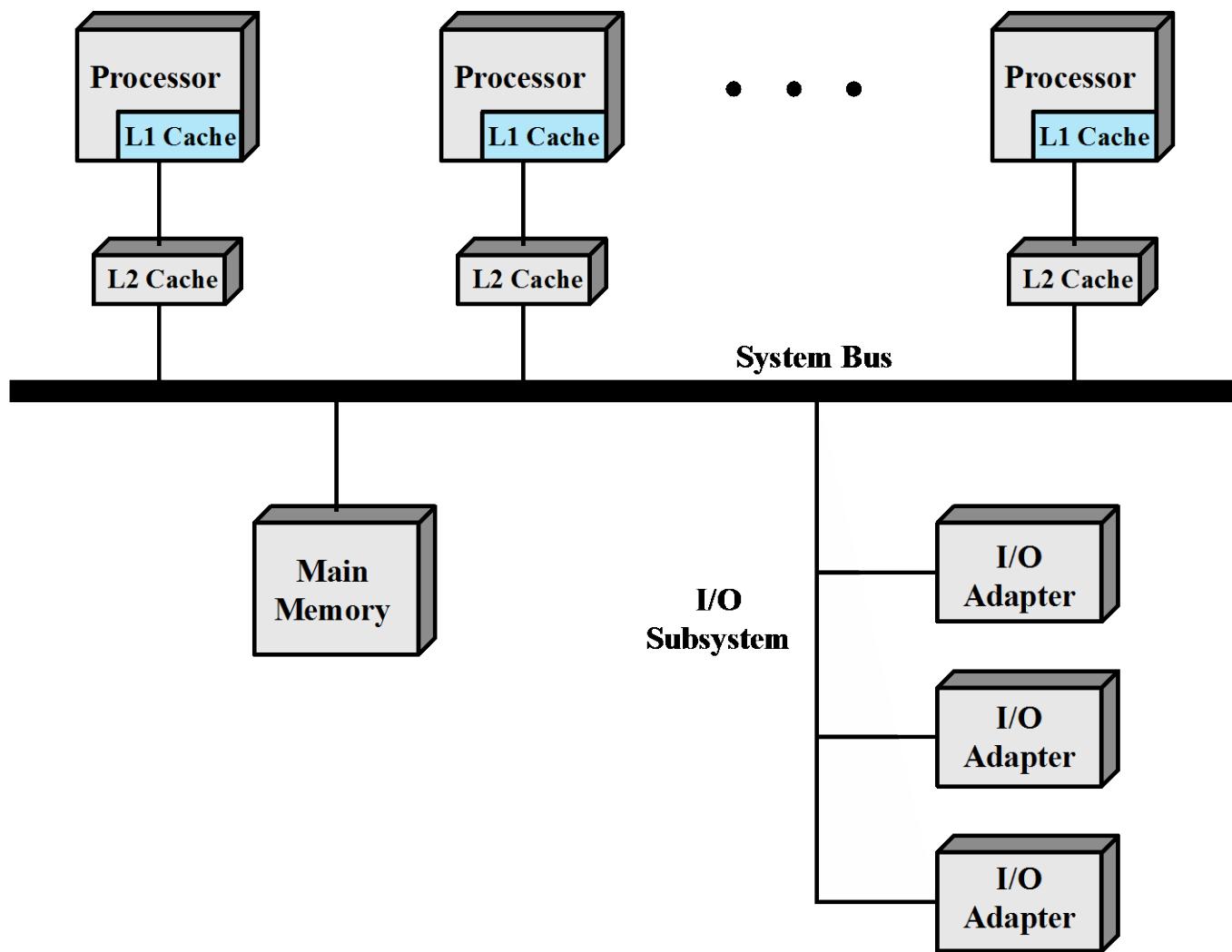
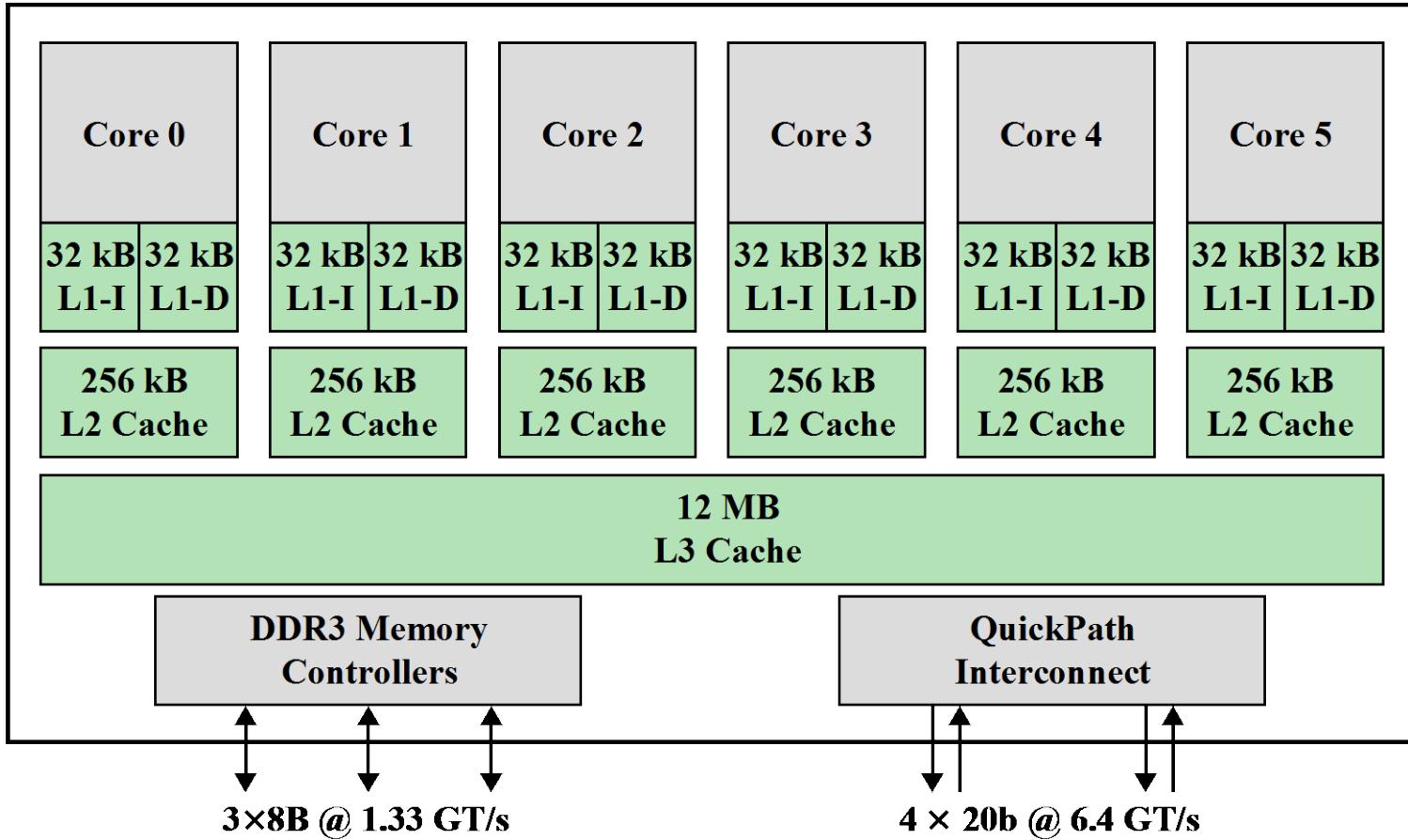


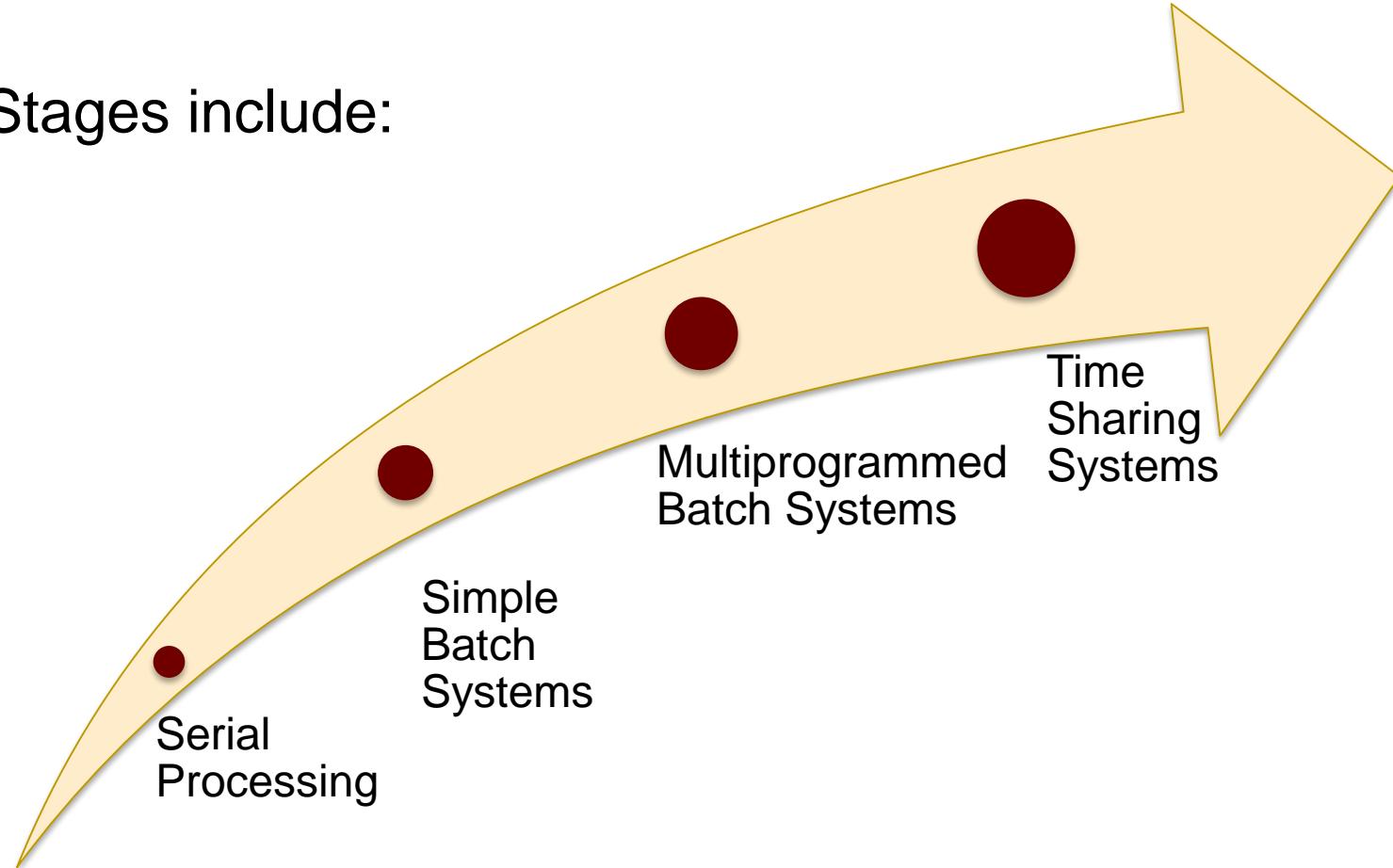
Figure 1.19 Symmetric Multiprocessor Organization



Evolution of Operating Systems

28

- Stages include:

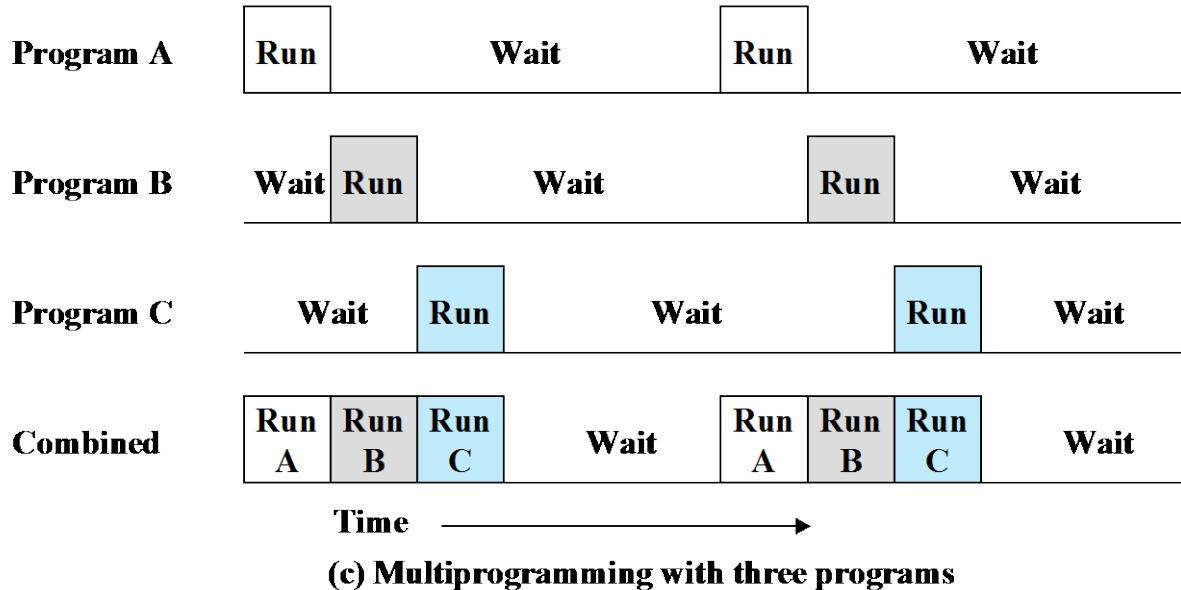


Modes of Operation

- **User Mode**
 - user program executes in user mode
 - certain areas of memory are protected from user access
 - certain instructions may not be executed
- **Kernel Mode**
 - monitor executes in kernel mode
 - privileged instructions may be executed
 - protected areas of memory may be accessed

Multiprogramming

30



- Multiprogramming
 - also known as multitasking
 - memory is expanded to hold three, four, or more programs and switch among all of them

Compatible Time-Sharing Systems (CTSS)

31

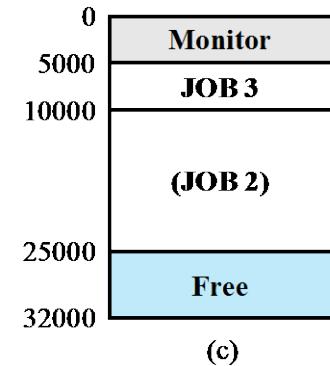
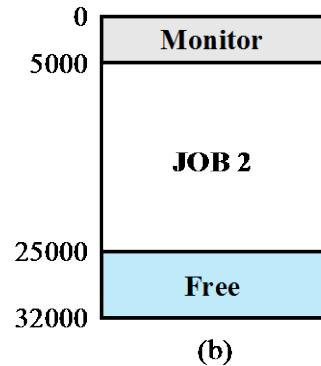
- One of the first time-sharing operating systems (at MIT by Project MAC)
- Ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that. To simplify both the monitor and memory management a program was always loaded to start at the location of the 5000th word
- System clock generates interrupts at a rate of approximately one every 0.2 seconds. At each interrupt OS regained control and could assign processor to another user (**Time Slicing**)
- At regular time intervals the current user would be preempted and another user loaded in. Old user programs and data were written out to disk. Old user program code and data were restored in main memory when that program was next given a turn.

Compatible Time-Sharing Systems (CTSS)

32

Memory Requirements

- JOB1: 15000
- JOB2: 20000
- JOB3: 5000
- JOB4: 10000



Loading order

J1 > J2 > J3 > J1 > J4 > J2

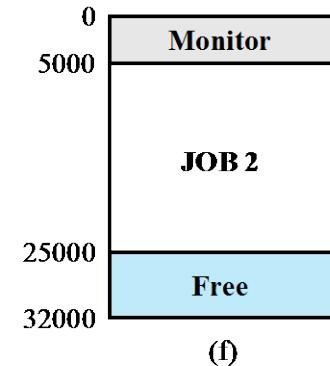
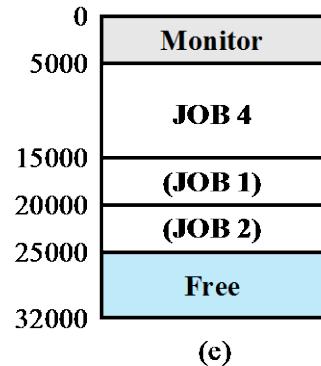
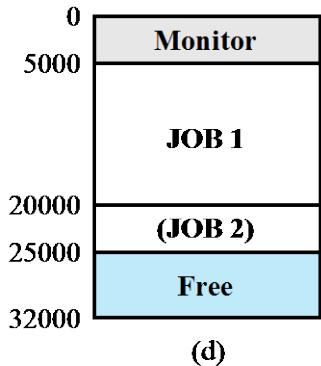


Figure 2.7 CTSS Operation

Fault Tolerance

33

- Refers to the ability of a system or component to continue normal operation despite the presence of hardware or software faults
- Typically involves some degree of redundancy
- Intended to increase the reliability of a system
 - typically comes with a cost in financial terms or performance
- The extent adoption of fault tolerance measures must be determined by how critical the resource is

Operating System Mechanisms

34

- A number of techniques can be incorporated into OS software to support fault tolerance:
 - Process isolation
 - Concurrency control
 - Virtual machines
 - Checkpoints and rollbacks

Processes and Threads

WEEK 2

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

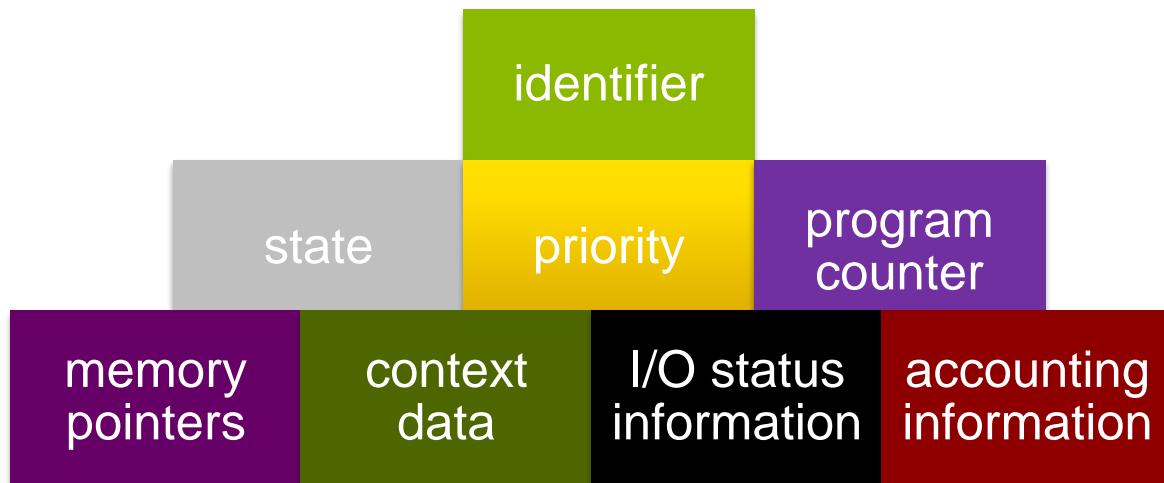
What is a process?

- 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 resources

Process Elements

37

- Two essential elements of a process are:
 - Program code
 - which may be shared with other processes that are executing the same program
 - A set of data associated with that code



Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

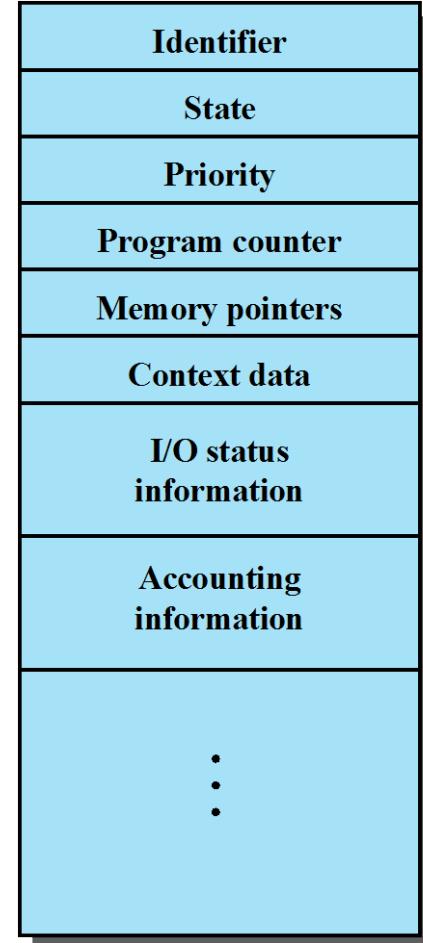


Figure 3.1 Simplified Process Control Block

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.

Typical Elements of a Process Control Block (Cnt..)

40

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

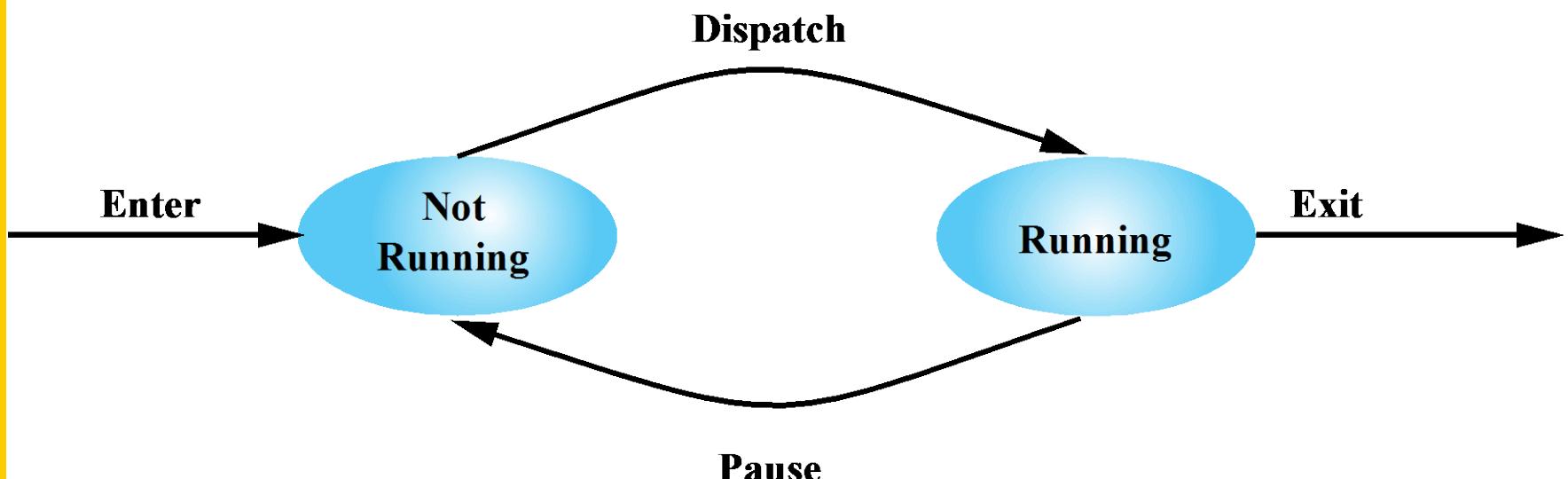
Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

Process States

- Not all processes are running at the same time.
 - Consider single (uni)-processor:
 - Only one process is “running” at any instant.
- Process in two “states”
 - Running:
 - May have any/all registers modified by CPU.
 - Process is actually doing something.
 - Not-running
 - “idle” - freeze and store for later
 - Program code is not executed

Two-State Process Model

42



(a) State transition diagram

Five-State Process Model

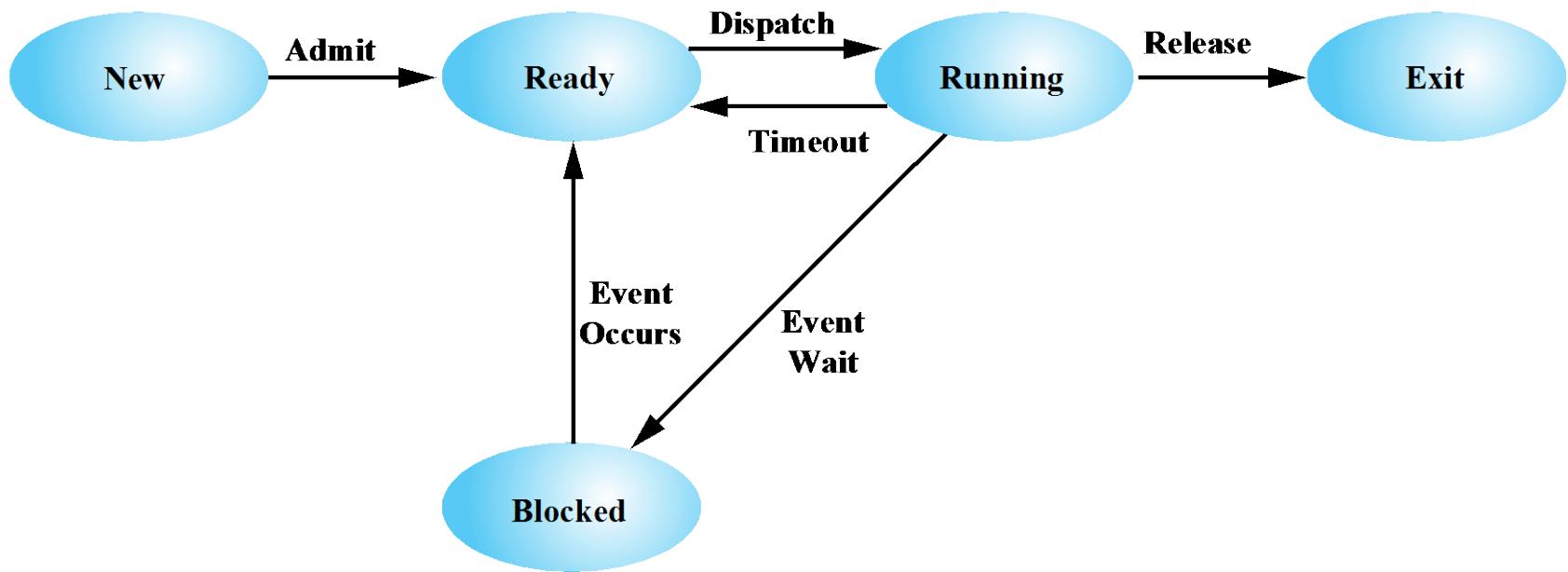
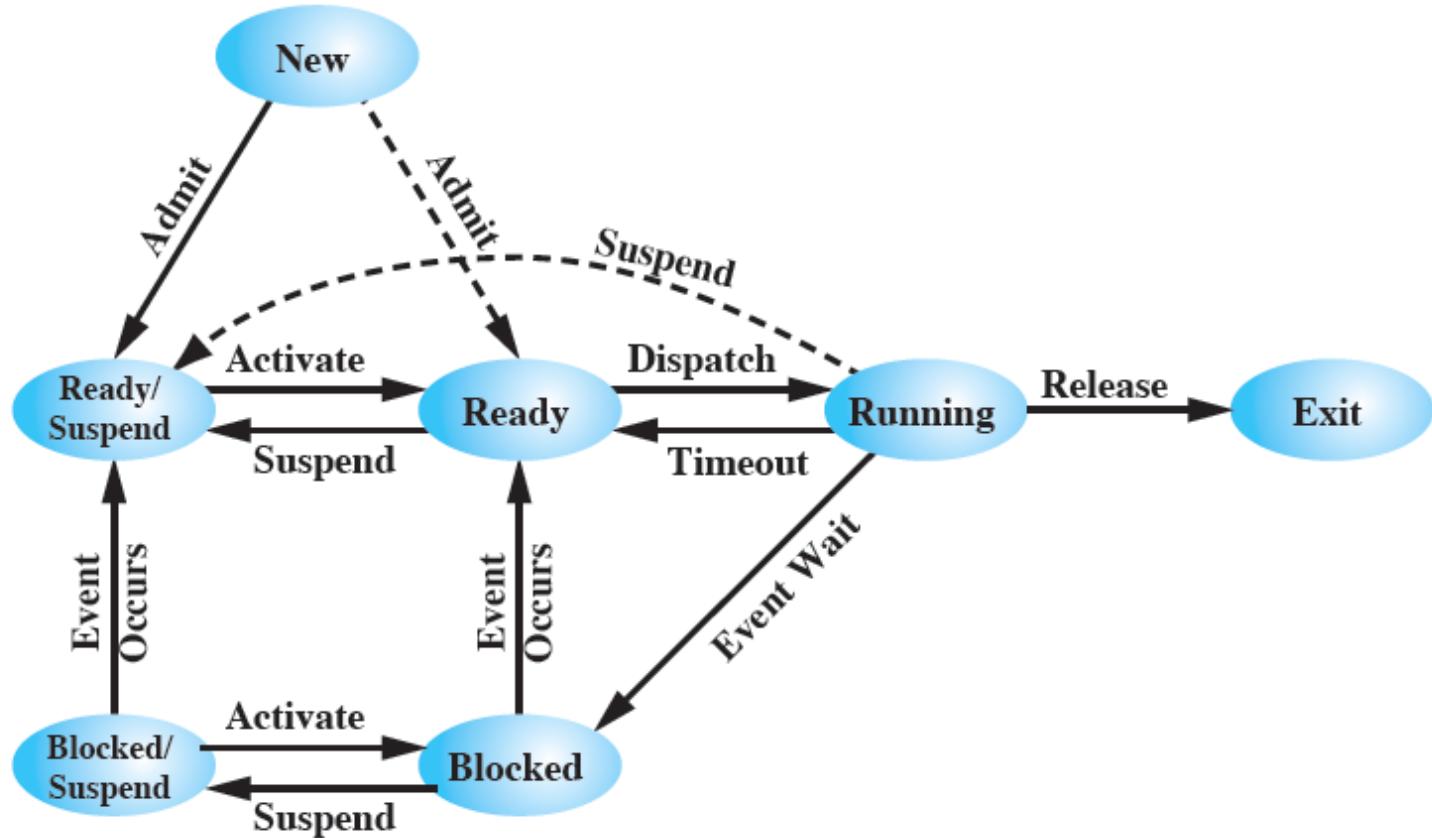


Figure 3.6 Five-State Process Model

Seven-State Process Model



OS Control Structures

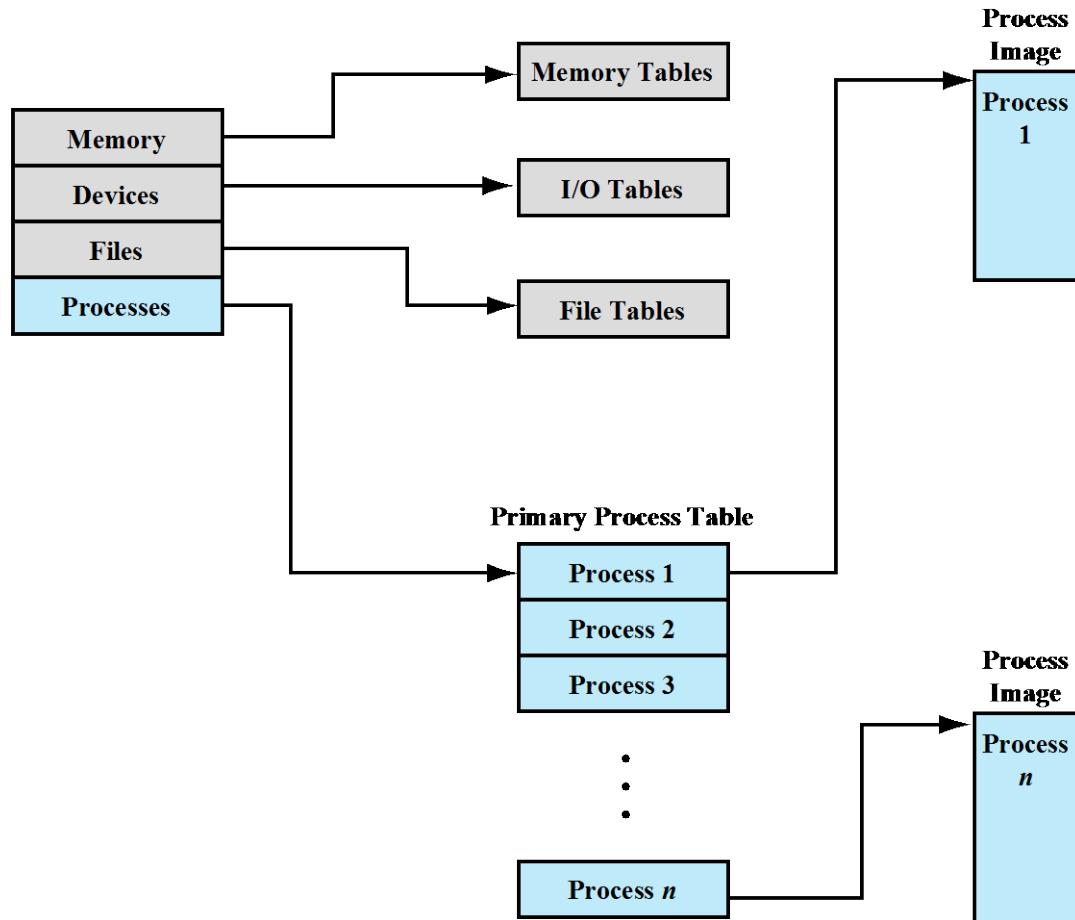


Figure 3.11 General Structure of Operating System Control Tables

Typical Elements of a Process Image

User Data

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

User Program

The program to be executed.

Stack

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

Process Control Block

Data needed by the OS to control the process (see Table 3.5).

Modes of Execution

47

User Mode



- less-privileged mode
- user programs typically execute in this mode

System Mode



- more-privileged mode
- also referred to as control mode or kernel mode
- kernel of the operating system

BUT how does the processor know which mode?

Process Switching

48

Mechanisms for Interrupting the Execution of a Process

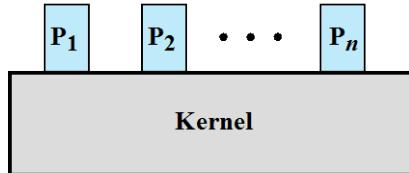
Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

Mode Switching

- **If no interrupts are pending the processor:**
 - proceeds to the fetch stage and fetches the next instruction of the current program in the current process
- **If an interrupt is pending the processor:**
 - sets the program counter to the starting address of an interrupt handler program
 - switches from user mode to kernel mode so that the interrupt processing code may include privileged instructions

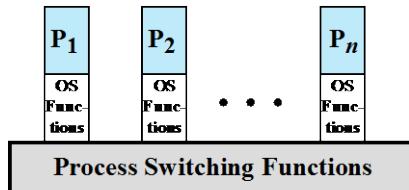
Execution of the Operating System

Non-process Kernel



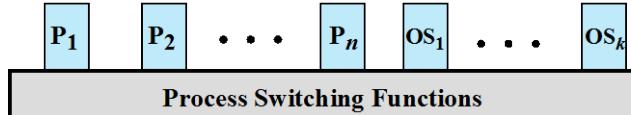
(a) Separate kernel

Execution with User Processes



(b) OS functions execute within user processes

Process based OS



(c) OS functions execute as separate processes



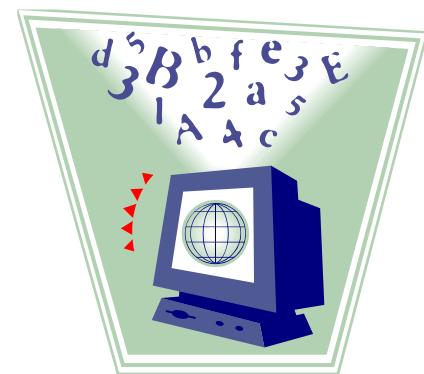
Process VS Threads

Processes

- The unit or resource allocation and a unit of protection
- A virtual address space that holds the process image
- Protected access to:
 - processors, files, I/O and other processes

Threads

- One or more threads in a 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)



Single VS Multi-Threaded Approaches

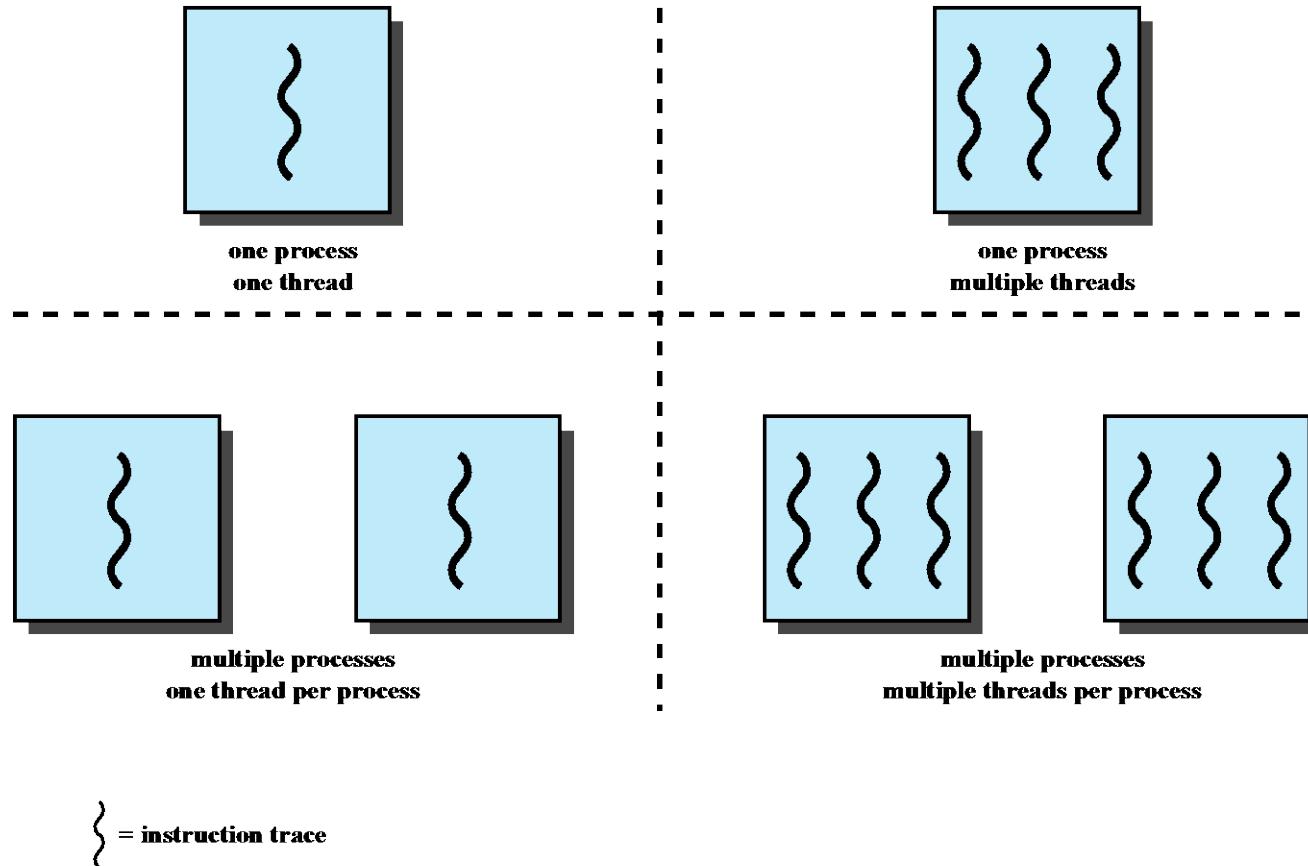


Figure 4.1 Threads and Processes

Key Benefits of Threads

- Takes less time to create a new thread than a process
 - In some systems 10 times faster
- Less time to terminate a thread than a process
- Switching between two threads (within the same process) takes less time than switching between processes
- Threads enhance efficiency in communication between programs

Thread Use in a Single-Processor System

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



Threads VS Process Management

54

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
- Process level action affect all of the threads in a process
 - Suspending a process involves suspending all threads of the process
 - Termination of a process terminates all threads within the process



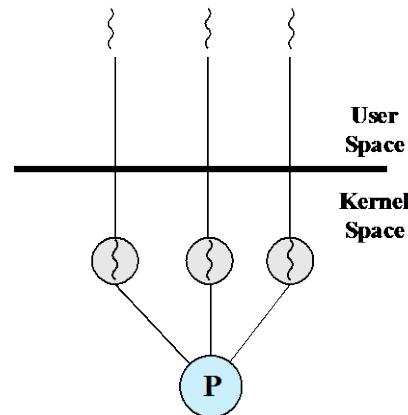
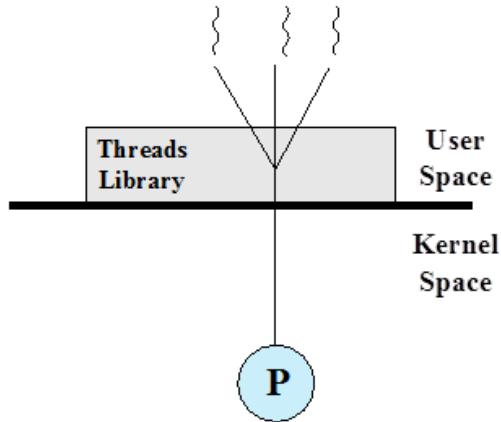
Thread Execution States

55

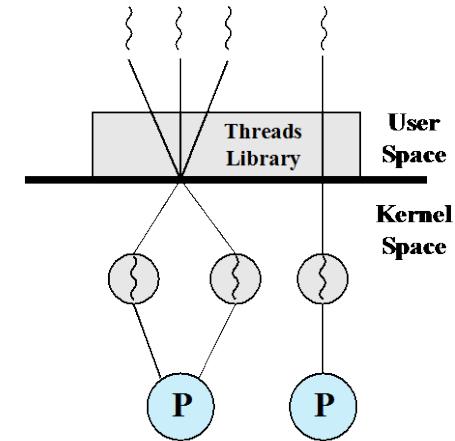
- The key states for a thread are:
 - Running
 - Ready
 - Blocked
- Thread operations associated with a change in thread state are:
 - Spawn
 - Block
 - Unblock
 - Finish
- Does blocking of a thread block a process?

Types of Threads

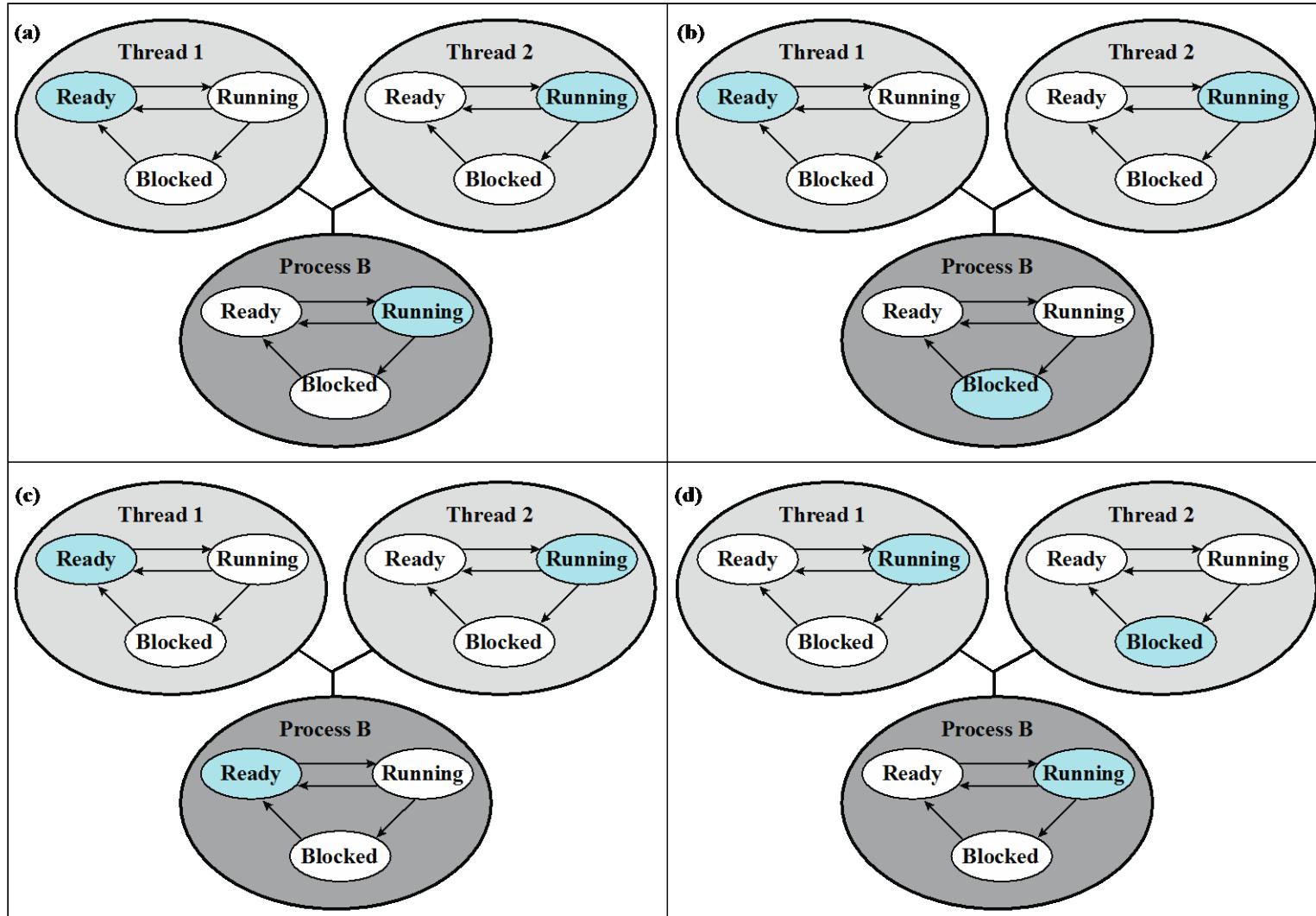
- User Level Thread (ULT)
- Kernel level Thread (KLT)
- Combined Approach



(b) Pure kernel-level



(c) Combined



Colored state
is current state

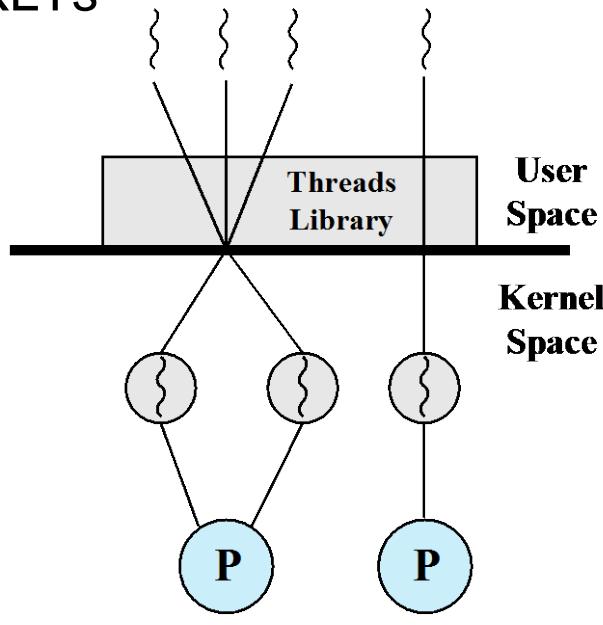
Figure 4.6 Examples of the Relationships Between User -Level Thread States and Process States

29/10/2019

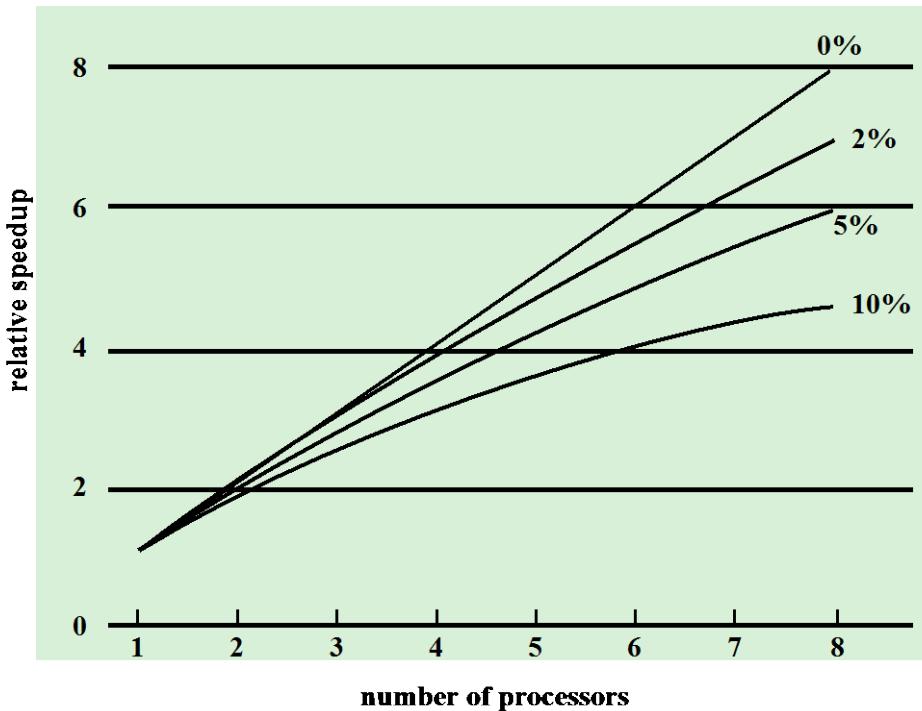
COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

Combined Approaches (ULT/KLT)

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Multiple ULTs from an application is mapped onto some (fewer or smaller) number of KLTs
- Solaris is an example



Multicore and Multithreading Performance

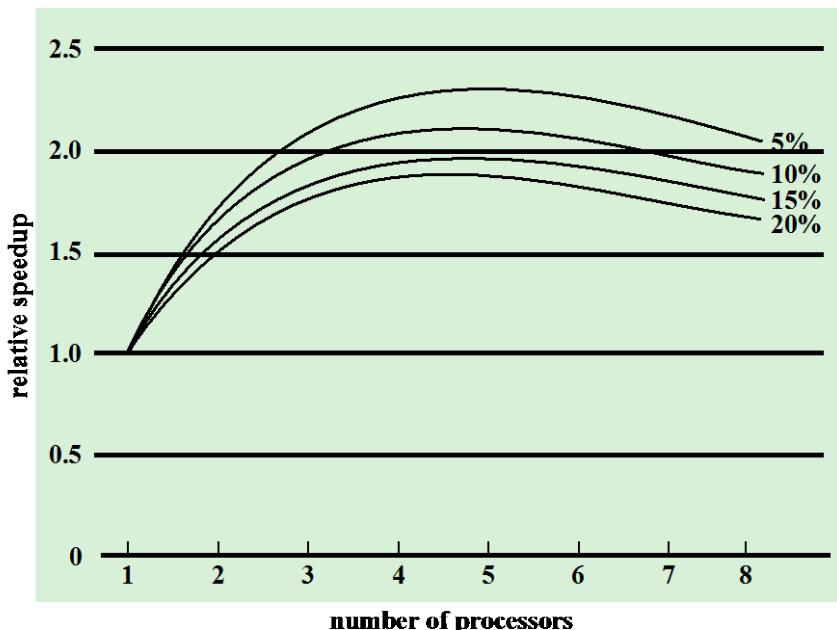


(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

Amdahl's Law:

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

F: fraction of code that is infinitely parallelizable
 (1-f): fraction of code that is inherently serial
 N: number of parallel processors



(b) Speedup with overheads

Scheduling
WEEK 3

What is Scheduling?

- On a multi-programmed system
 - We may have more than one *Ready* process
- On a batch system
 - We may have many jobs waiting to be run
- On a multi-user system
 - We may have many users concurrently using the system
- The *scheduler* decides who to run next.
 - The process of choosing is called ***scheduling***.

Types of scheduling

There are four kinds of scheduling in an OS:

1. Long term scheduling

- The decision to add to the pool of processes to be executed

2. Medium term scheduling

- The decision to add to the number of processes that are partially or fully in main memory

3. Short term scheduling

- The decision as to which available process will be executed by the processor

4. I/O scheduling

- The decision as to which process's pending I/O request shall be handled by an available I/O device.

Processor Scheduling

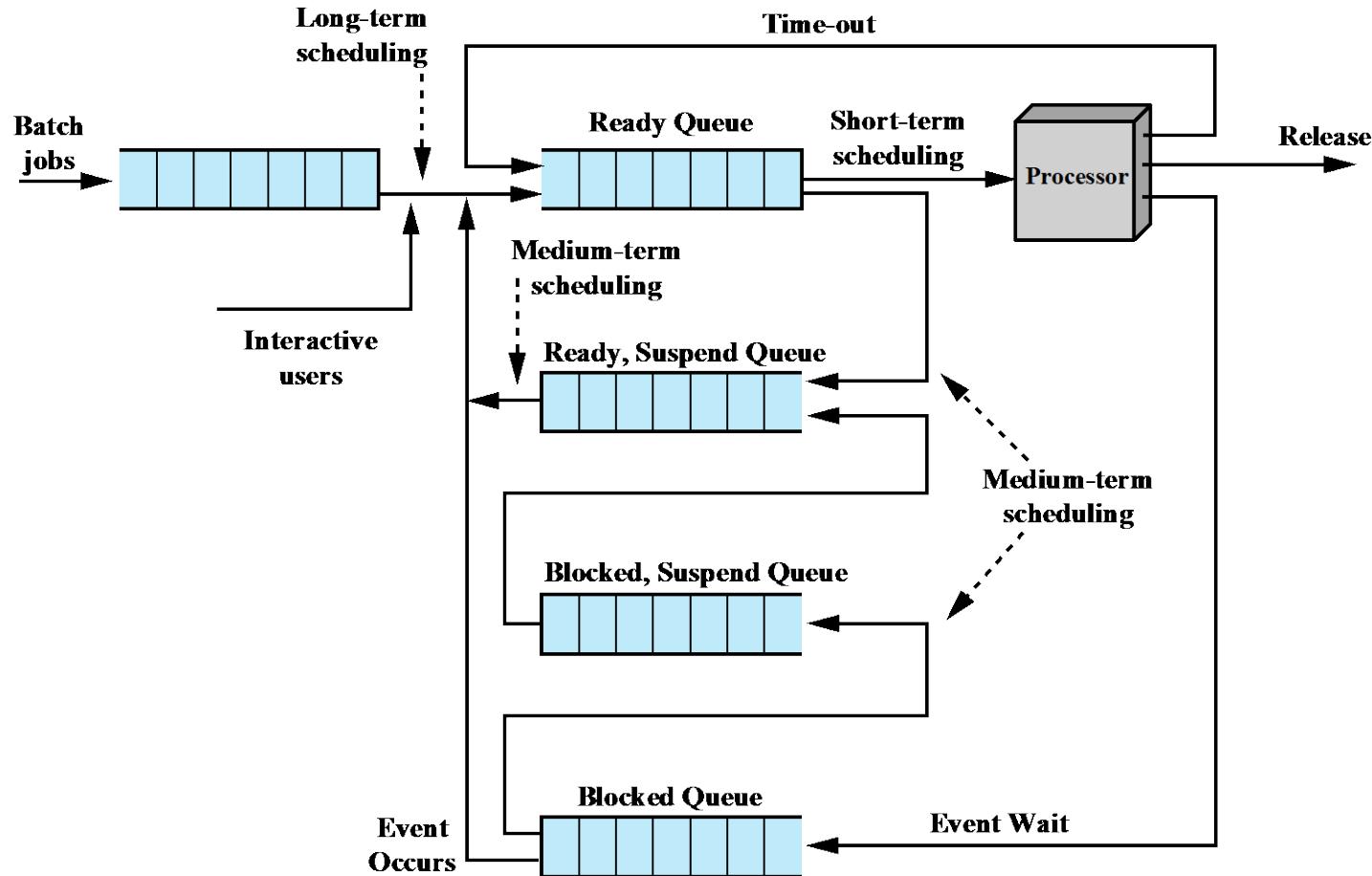


Figure 9.3 Queuing Diagram for Scheduling

Short Term Scheduling Criteria

64

- Main objective is to allocate processor time to optimize certain aspects of system behavior
- A set of criteria is needed to evaluate the scheduling policy
- **User-oriented criteria**
 - relate to the behaviour of the system as perceived by the individual user or process (such as response time in an interactive system)
 - important on virtually all systems
 - **Focus:** Service provided to the user
- **System-oriented criteria**
 - focus in on effective and efficient utilization of the processor (rate at which processes are completed)
 - generally of minor importance on single-user systems
 - **Focus:** System performance

Scheduling Policies

65

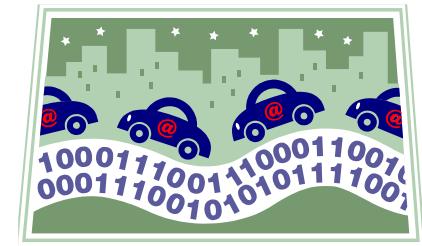
There are many kinds of short term scheduling algorithms.

Schedulers have two main parameters:

- **Selection function:**
- **Decision mode:**

Decision mode: Nonpreemptive vs. Preemptive

- **Nonpreemptive**
 - Once a process is in the running state, it will continue until it terminates or blocks itself for I/O
 - **Preemptive**
 - Currently running process may be interrupted and moved to ready state by the OS
 - Preemption may occur when new process arrives, on an interrupt, or periodically



Scheduling Algorithms

67

- **First Come First Served (FCFS)**
- **Round Robin (RR)**
- **Shortest Process Next (SPN)**
- **Shortest Remaining Time (SRT)**
- **Highest Response Ration Next (HRRN)**
- **Feedback (FB)**
- **Priority Scheduling**
- **Fair Share Scheduling**

Performance Indices

- **Arrival Time (T_a):** Time when process enter the system (short time scheduling)
- **Service Time (T_s):** Total execution time required by the process
- **Turnaround Time (T_r):** Total time that the items spends in the system
- **Normalized Turnaround Time:** T_r/T_s : Relative delay experienced by a process. Minimum value is 1.0; Increasing values correspond to decreasing level of service.

First-Come-First-Served (FCFS)

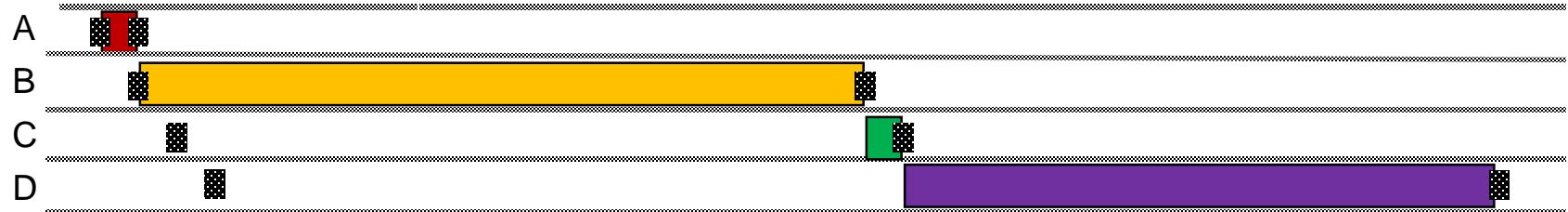
- Simplest scheduling policy
- Also known as first-in-first-out (FIFO) or a strict queuing scheme
- When the current process ceases to execute, the process that has been in the Ready queue the longest is selected
- Non-preemptive policy
- It may be implemented by using the ready queue as a FIFO queue.
- Performs much better for long processes than short ones
- Tends to favor processor-bound processes over I/O-bound processes



FCFS Performance

- FCFS is simple but performs badly if the job mix contains jobs of widely different characteristics, for example:

process	A	B	C	D
arrival time	0	1	2	3
service time	1	100	1	100
start time	0	1	101	102
finish time	1	101	102	202
turnaround time Tq	1	100	100	199
Tq/Ts	1.0	1.0	100	2.0



- Here the long jobs (B, and D) get a reasonable turnaround time, but the short job (C) has an unreasonable turnaround time.

FCFS Performance

71

- FCFS also has the **disadvantage** that a CPU bound job can monopolise the processor, leaving I/O devices idle.
 - Suppose there is one processor bound job and many I/O bound jobs
 - When CPU bound job runs all I/O bound jobs wait and I/O devices are idle
 - When the CPU bound job finishes, the I/O bound jobs quickly become blocked waiting and the CPU becomes idle.
 - May result in inefficient use of both the processor and the I/O devices
- FCFS is not an attractive scheduling policy on its own
- FCFS is best used in **combination** with another policy

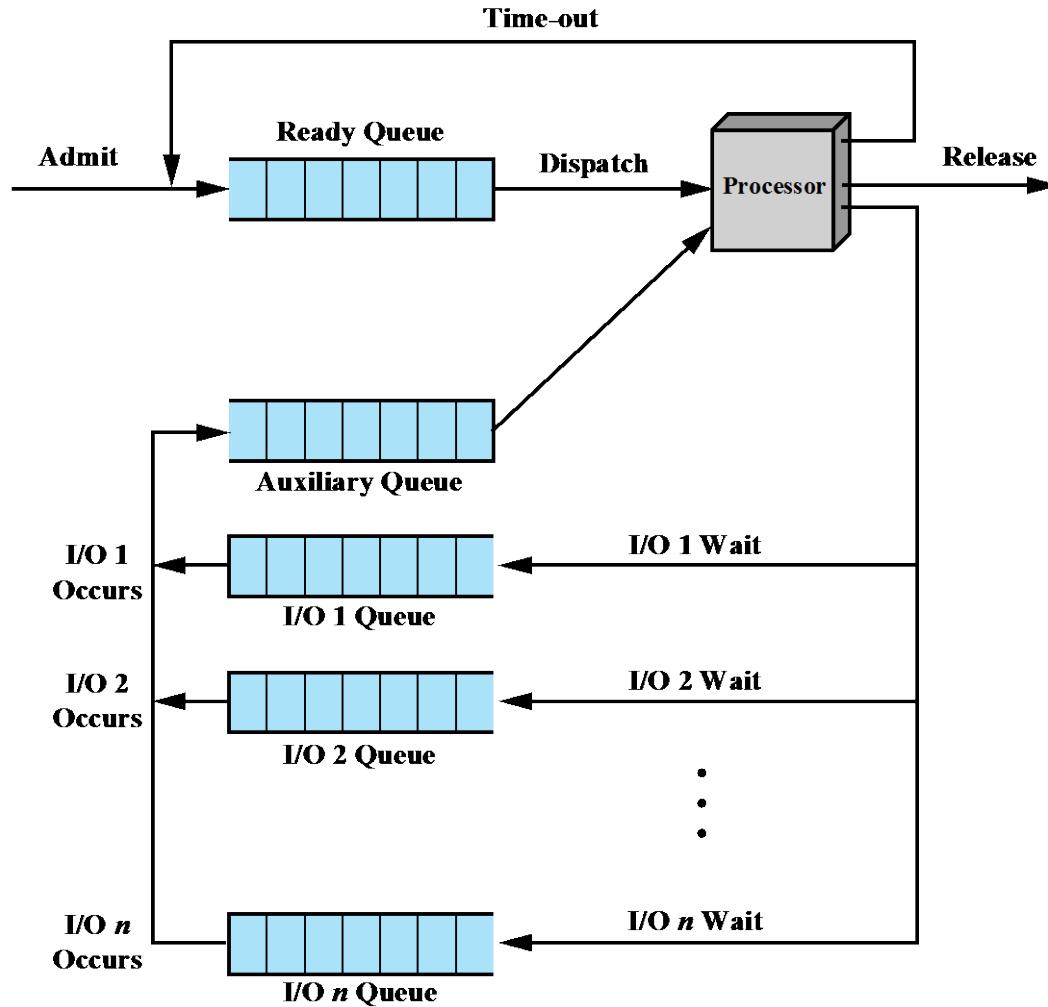
Round Robin

72

- Stop monopolization of length job by preemption
- In **Round Robin scheduling**, (RR) or **time slicing**, a clock regularly interrupts and pre-empts the running process.
- The next process to run is chosen by an FCFS strategy.
- The main parameter is the **length of the time slice**. A short time slice ensures that brief processes move through the system rapidly, but means many interrupts.
- What happens when time slice becomes infinite?



Virtual Round Robin (VRR)



Virtual Round Robin (VRR)

- **Virtual Round Robin (VRR)** avoids unfairness of RR.
- New processes arrive & join Ready queue - FCFS.
- Timed-out running processes rejoin Ready queue.
- I/O blocked processes join appropriate I/O queue.
 - (So far this is standard)
- NEW FEATURE: add FCFS *Auxiliary* queue.
- Processes released from I/O wait queue join *Auxiliary* queue.
- Scheduler dispatches processes from *Auxiliary* queue before processes from Ready queue. But these processes only run with up to the time remaining from their last basic time quantum (before they were blocked).

- The **Shortest Process Next** (SPN) policy always chooses the process which has the shortest expected running time.
- This policy is **non-pre-emptive**.
- The process with the shortest expected processing time is selected next (Shortest Job First – SJF)
- A short process will jump to the head of the queue
- Possibility of starvation for longer processes

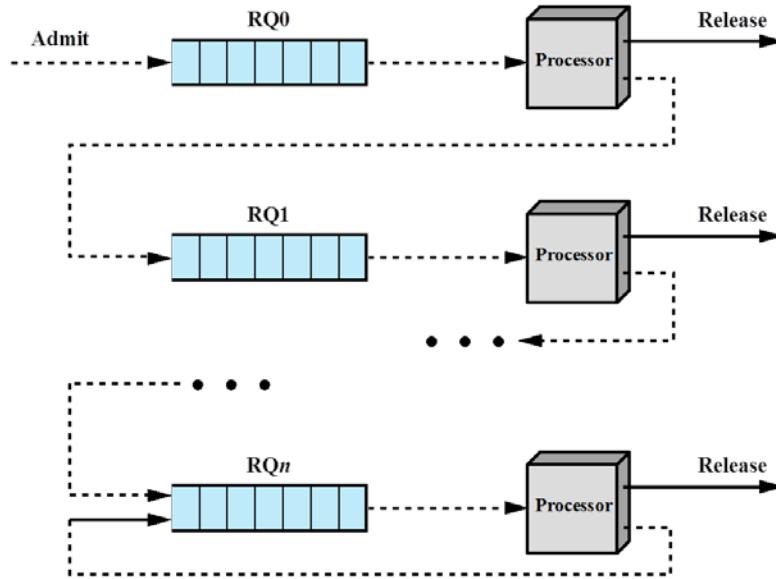
- The **Shortest Remaining Time** (SRT) scheduler chooses a process with the shortest expected remaining time.
- This is a kind of **preemptive version of SPN**.
- When a new process becomes available, the OS may **preempt** a running process if the expected time to completion for the currently running process is longer than the expected run time of the new process.
- As with SPN:
 - the OS must estimate runtimes
 - long processes may be starved

- The **Highest Response Ratio Next (HRRN)** policy chooses the next process to run to be the one with the highest value of RR, defined below.
- The policy is **non-pre-emptive**.
- Define $RR = \frac{w+s}{s}$ where
 w = time spent waiting, and
 s = expected service time.
- Thus, at the completion of a job, $RR = T_r/T_S$, which is the value we are trying to minimise.

- SPN, SRT, HRRN can not be used if no indication of the relative length of the processes are available
- We can prefer shorter jobs by penalizing the jobs that have been running for long
- Here we are focusing
 - NOT on the time remaining
 - on the time already executed

Feedback

- Preemptive scheduling
- For the **FB** policy, we use a set of queues, one for each priority level.

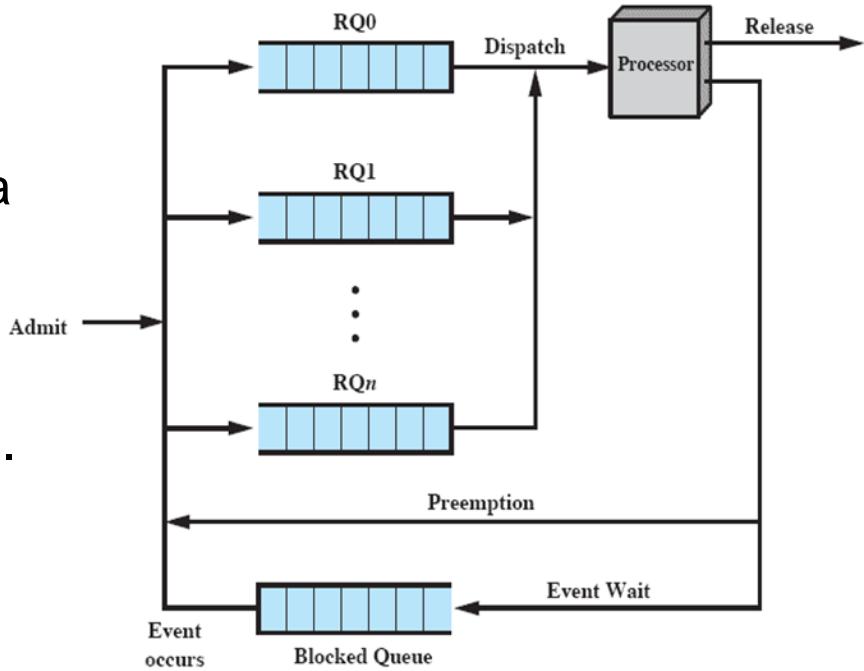


- When a process enters the system
 - it enters the **high priority queue** (*priority i*, with $i=0$).
 - It receives a time slice of service
- When it is pre-empted or blocked,
 - it returns to a lower queue priority queue (*priority i+1*).
- Within each queue, an FCFS mechanism is used; except for the lowest priority queue, which is round-robin.

Priority Scheduling

Priorities can be managed with a queue for each priority level:

- If queue of priority $0..m$ are empty and queue $m+1$ is non-empty, then a process is chosen from queue $m+1$.
- This procedure can be used in conjunction with a separate scheduling algorithm for each queue.
- Note that lower priority processes may be **starved**; this can be alleviated by changing the priority of a process with its execution history (DYNAMIC PRIORITY).



Priority Boosting

- Major problem in priority scheduling is **starvation**
- A steady stream of higher-priority process can leave some low-priority process waiting indefinitely

RUMOR: When they shut down the IBM7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.

- A solution is **aging**
 - Gradually increase the priority of processes that wait in the system for a long time

Fair-Share Scheduling

- Scheduling decisions based on the process sets
- Each user is assigned a share of the processor
- Scheduling is done for a set of **fair share** groups
 - Allocate a fraction of processor resource to each group
- Objective is to monitor usage to give fewer resources to users who have had more than their fair share and more to those who have had less than their fair share



Time	Process A			Process B			Process C		
	Priority	Process CPU count	Group CPU count	Priority	Process CPU count	Group CPU count	Priority	Process CPU count	Group CPU count
0	60 1 2 • • 60	0 1 2 • • 60	0 0 0	60 1 2 • • 60	0 1 2 • • 60	0 0 0	60 1 2 • • 60	0 0 0	0 1 2 • • 60
1	90 60	30 30	30	60 1 2 • • 60	0 1 2 • • 60	0 0 0	60 1 2 • • 60	0 0 0	0 1 2 • • 60
2	74 16 17 • • 75	15 16 17 • • 75	15 16 17 • • 75	90 30 30	30 30	30	75 60 60	0 0 30	0 1 2 • • 60
3	96 74 16 17 • • 75	37 15 16 17 • • 75	37 15 16 17 • • 75	74 15 16 17 • • 75	15 16 17 • • 75	15 16 17 • • 75	67 60 60 0 1 2 15 16 17 • • • 75	0 60 75	15 16 17 • • 75
4	78 19 20 • • 78	18 19 20 • • 78	18 19 20 • • 78	81 7 37	7 37	37	93 90 90	30 30 37	37 37 37
5	98 70 3	39 3 18	39 3 18	70 3 18	3 18	18	76 76 76	15 15 18	18 18 18

Group 1 Group 2

Colored rectangle represents executing process



Real-time System Scheduling and Multiprocessor Scheduling

WEEK 4

Multiprocessor Scheduling

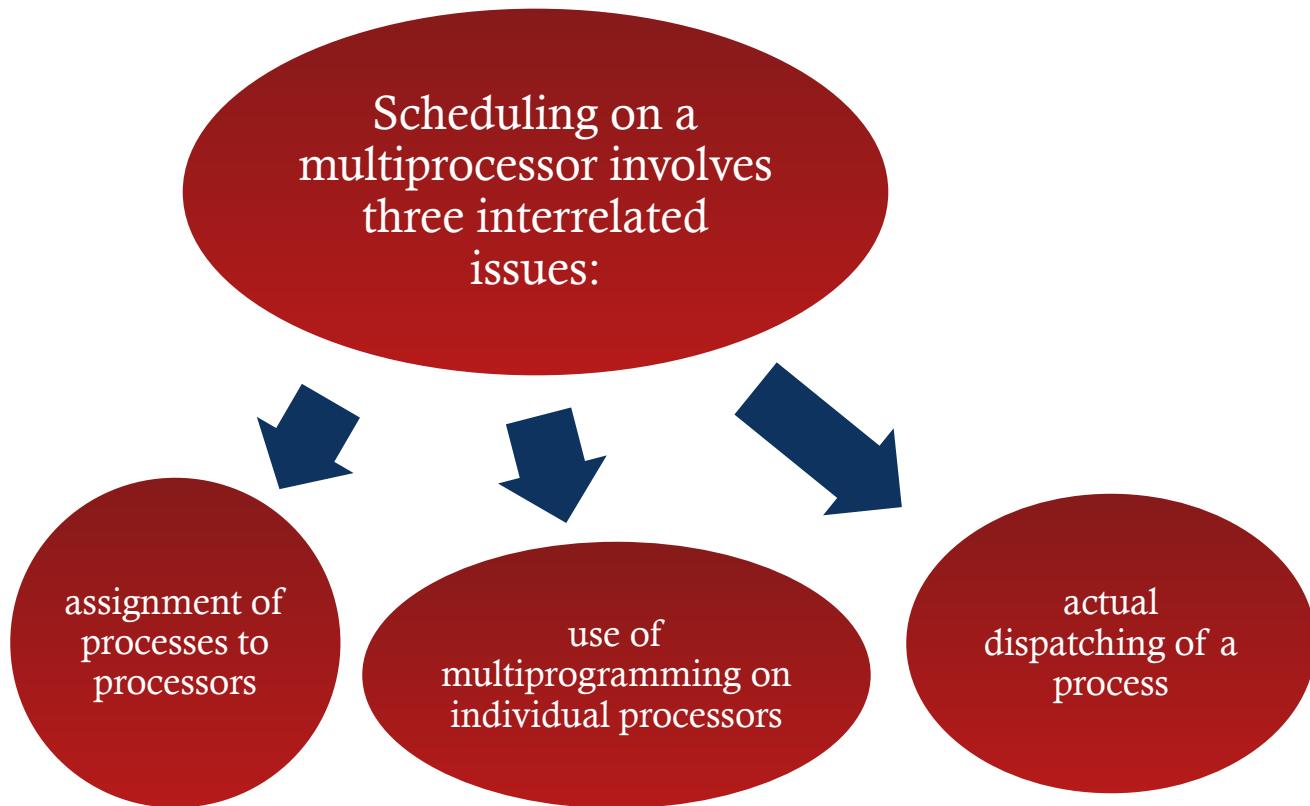
- There are many kinds of multiprocessors:
 - **Loosely coupled multiprocessor:** consists of a collection of relatively autonomous systems, each with its own main memory and I/O channels.
 - Example: clusters, network of workstations,,
 - **Functionally specialised processors:** these are slave processors which provide services for a general purpose processor.
 - Examples: I/O processor, PIPADS image processor linked to i486 master, array processor
 - **Tightly coupled multiprocessing:** a set of processors which share an operating system, and often share a large amount of their memory. Examples: Vax dual processor, Sun10/54, Intel Hypercube, Maspar, Connection machines.

Synchronization Granularity

- Parallelism can be categorised on granularity (frequency of synchronisation):
 - **Independent parallelism**
 - **Coarse and very coarse grained parallelism**
 - **Medium grained parallelism**
 - **Fine grained parallelism**

Design Issues

87



The approach taken will depend on the degree of granularity of applications and the number of processors available



Design Issues

- Assignment of processes to CPUs for uniform multiprocessor
 - *Static*: a process is assigned to a processor for the total life of the process. A short-term queue is maintained for each processor
 - a simple approach, with very little scheduling overhead.
 - the main disadvantage is that one processor may be idle while another has a long queue.
 - *Dynamic*: a process may change processors during its lifetime. A single global queue is maintained - jobs scheduled to any available processor.
 - more efficient than static assignment
 - if shared-memory, context info available to all processors, cost of scheduling is independent of processor identity

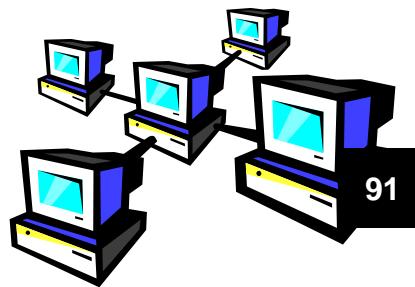


Design Issues

- Means of assigning processes to processors
 - **Master-slave architecture**: the OS runs on a particular processor and schedules user tasks on other processors. Resource control is simple.
 - very simple and requires little enhancement to a uniprocessor multiprogramming operating system.
 - the master can be performance bottleneck and failure of the master is disastrous.
 - **Peer architecture**: the OS can run on any processor, and effectively each processor schedules itself.
 - complicates the OS
 - conflict resolution (two processors want the same job or resource) becomes complex.

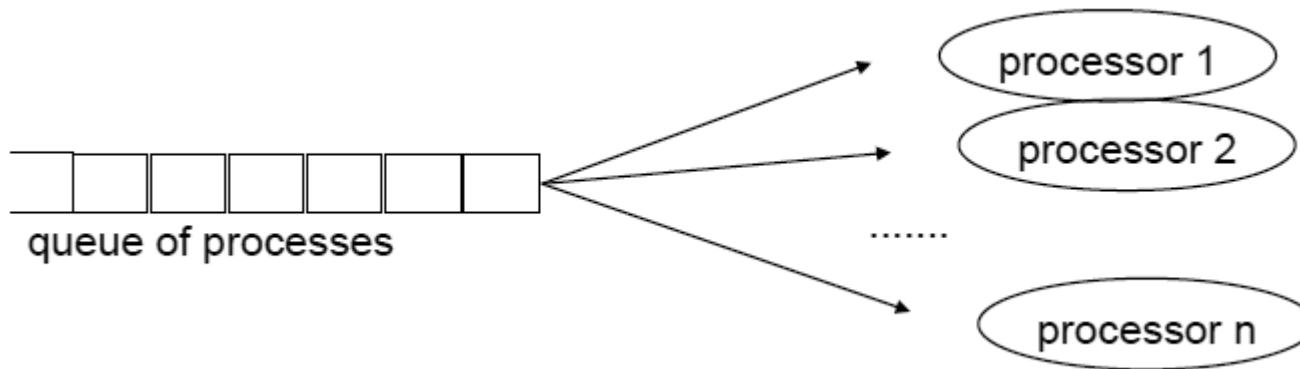
Design Issues

- **Multiprogramming on individual processors**
 - for **coarse grained parallelism**, multiprogramming is necessary for each processor, in the same way as for a uniprocessor - to get higher usage and better performance.
 - however if there are many processors (medium- or fine-grained parallelism) then it may not be necessary to keep all processors busy all the time.
 - Better throughput may be achieved if some processors are sometimes idle. WHY?
 - Thus multiprogramming on individual processors may not be necessary. We're looking for best average performance of applications.
- **Process Dispatching**
 - the complex selection procedure involved in uniprocessors may not be necessary and could even be detrimental; a simple FIFO strategy may be best - less overhead.
 - the issues can be quite different for scheduling **threads**.

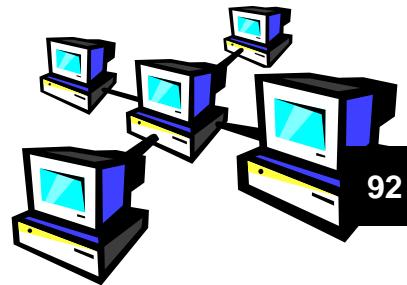


Process Scheduling

- Generally processes are *not* dedicated to processors.
- Processes can be scheduled from a single queue to multiple processors. Each processor chooses its next process from the queue.
- Or there is a multiple queue priority scheme.

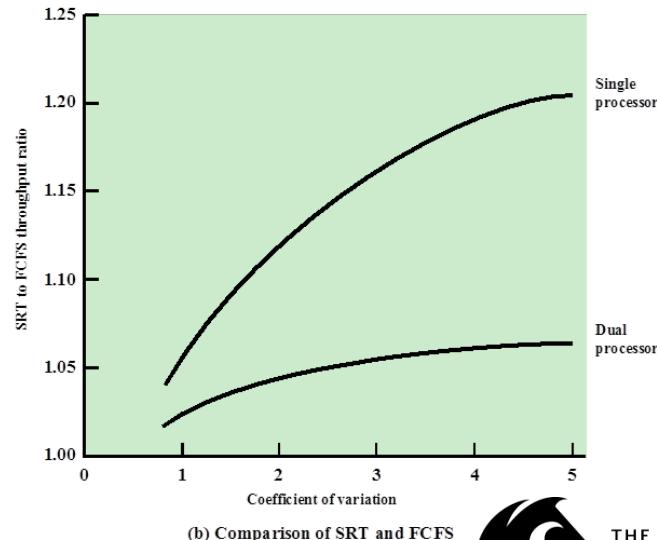
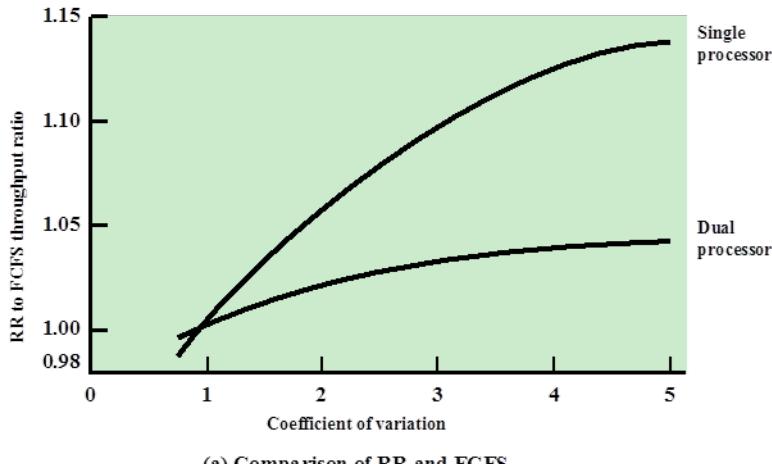


Process Scheduling



92

- Experience has shown that for **processes** (tasks are relatively independent), a simple FCFS policy for scheduling is best.
 - If the job mix is highly variable, then Round Robin is better than FCFS, both on a single processor and a dual processor.
 - However, the improvement of RR over FCFS is very small for the dual processor, and becomes even smaller for more than two processors.



Thread Scheduling

- Threads are different from processes in ways that are important for scheduling:
 - the overhead of thread switching is smaller than for process switching
 - threads share resources (including memory) and there is some principle of locality.
- Threads are important for exploiting medium grained parallelism. Thread scheduling is important for exploiting true parallelism as much as possible.
 - Threads simultaneously running on separate processors give big performance gains.
 - Thread scheduling models significantly impact performance.
- Experience with thread scheduling is increasing; research is still active and ideas are broad.

Approaches to Thread Scheduling

processes are not assigned to a particular processor

Load Sharing/ Self Sharing

a set of related threads scheduled to run on a set of processors at the same time, on a one-to-one basis

Gang Scheduling

Four approaches for multiprocessor thread scheduling and processor assignment are:

provides implicit scheduling defined by the assignment of threads to processors

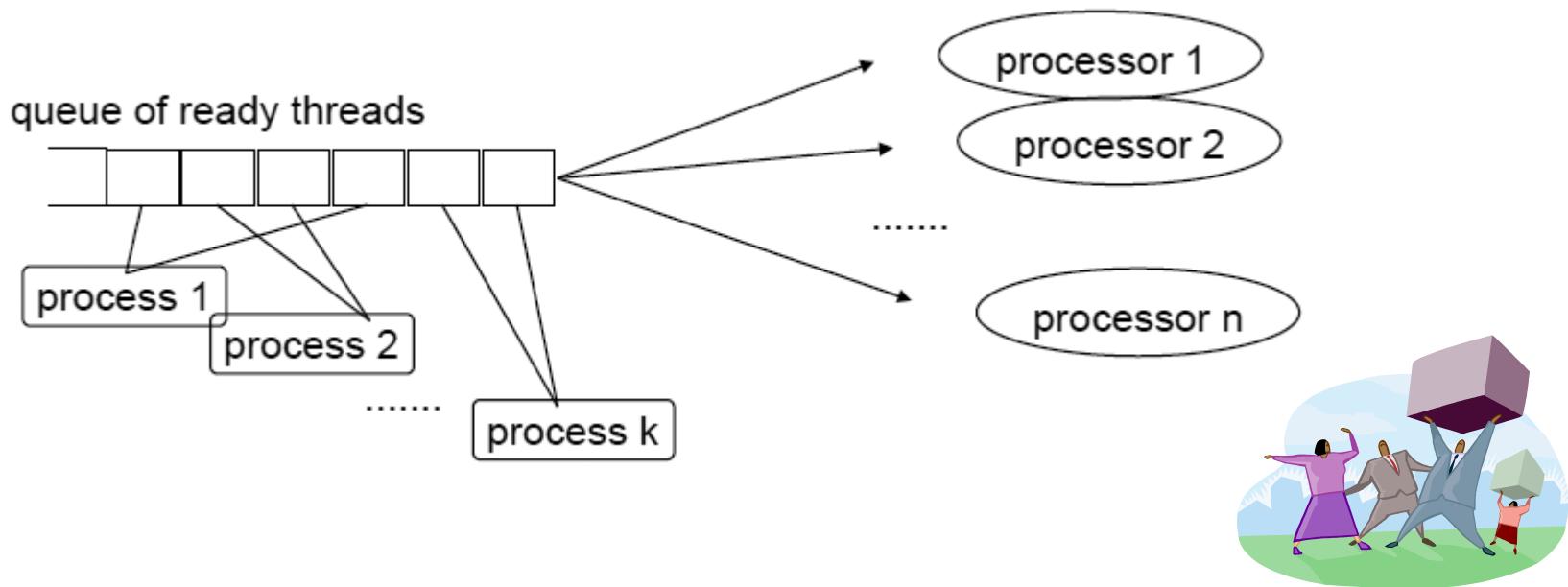
Dedicated Processor Assignment

the number of threads in a process can be altered during the course of execution

Dynamic Scheduling

Load Sharing/Self Scheduling

- For **Self Scheduling**, processes are not dedicated to any particular processor. A global queue of ready threads is maintained, and an idle processor selects a thread from the queue.



Gang Scheduling

- In **Gang Scheduling** (*group scheduling, co-scheduling*), a set of related threads are scheduled on a set of processors at the same time. Some rationale behind gang scheduling is:
 - if closely related threads execute in parallel, then synchronisation blocking will be reduced, and less process switching will occur.
 - scheduling overhead is reduced by making a single decision for a group of threads.
- The set or **gang** of threads can be defined by the programmer, or it can be automatically defined (for example, all threads of a particular process can be a gang).

Processor Allocation in Gang Scheduling

- **Uniform Scheduling:** Each application gets $1/M$ of the available time in N processor.
- **Weighted:** The amount of N processor time an application gets is weighted by the number of threads in that application

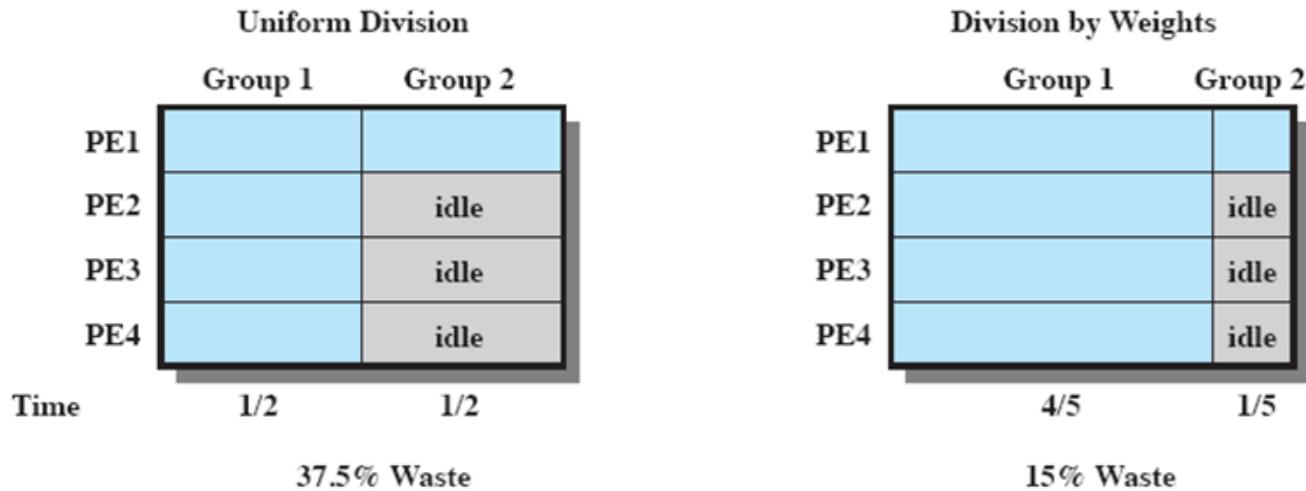


Figure 10.3 Example of Scheduling Groups with Four and One Threads [FEIT90b]

Dedicated Processor Assignment

98

- In **Dedicated Processor Assignment**, a group of processors is assigned to a job for the complete duration of the job.
- An extreme form of gang scheduling
- Each thread is assigned to a processor and this processor remains dedicated to that thread until the job completes.
- This approach results in idle processors (there is no multiprogramming).
- Defense of this strategy:
 - in a highly parallel system, with tens or hundreds of processors, processor utilization is no longer so important as a metric for effectiveness or performance
 - the total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program

Dynamic Scheduling

- For some applications it is possible to provide language and system tools that permit the number of threads in the process to be altered dynamically
 - this would allow the operating system to adjust the load to improve utilization
- Both the operating system and the application are involved in making scheduling decisions
- The scheduling responsibility of the operating system is primarily limited to processor allocation
 - For example, the OS can partition the processors among the jobs, and each job partitions its threads among its processors.
- Not suitable for all applications.

Real Time Scheduling

- Correctness of the system may depend not only on the logical result of the computation but also ***on the time*** when these results are produced
- e.g.
 - Tasks attempt to control events or to react to events that take place in the outside world
 - These external events occur in *real time* and processing must be able to keep up
 - Processing must happen in a timely fashion
 - Neither too late, nor too early

Hard-Real time system

- Requirements:
 - **Must always meet all deadlines** (time guarantees)
 - You have to guarantee that in any situation these applications are done in time,
 - otherwise it will cause unacceptable damage or a fatal error to the system
- Examples:
 1. If the landing of a fly-by-wire jet cannot react to sudden side-winds within some milliseconds, an accident might occur.
 2. An airbag system or the ABS has to react within milliseconds

Soft-Real Time Systems

- Requirements:
 - **Must mostly meet all deadlines**
 - An associated deadline that is desirable but not mandatory
 - it still makes sense to schedule and complete the task even if it has passed its deadline
- Examples:
 1. Multimedia: 100 frames per day might be dropped (late)
 2. Car navigation: 5 late announcements per week are acceptable
 3. Washing machine: washing 10 sec over time might occur once in 10 runs, 50 sec once in 100 runs.

Properties of Real-Time Tasks

103

- To schedule a real time task, its properties must be known
- The most relevant properties are
 - Arrival time (or release time)
 - Maximum execution time (service time)
 - Deadline
 - Starting deadline
 - Completion deadline

Categories of Real Time Tasks

- **Periodic**
 - Each task is repeated at a regular interval
 - Max execution time is the same each period
 - Arrival time is usually the start of the period
 - Deadline is usually the end
- **Aperiodic (sporadic)**
 - Each task can arrive at any time

Characteristics of Real-Time System

105

Real-time operating systems have requirements in five general areas:

- Determinism
- Responsiveness
- User Control
- Reliability
- Fail-safe operation

Real-time Scheduling Approaches

106

- **Static table-driven scheduling**
 - Given a set of tasks and their properties, a schedule (table) is precomputed offline.
 - Used for periodic task set
 - Requires entire schedule to be recomputed if we need to change the task set
- **Static priority-driven scheduling**
 - Given a set of tasks and their properties, each task is assigned a fixed priority
 - A preemptive priority-driven scheduler used in conjunction with the assigned priorities
 - Used for periodic task sets

Real-time Scheduling Approaches

107

- **Dynamic planning-based scheduling**
 - Task arrives prior to execution
 - The scheduler determines whether the new task can be admitted
 - Can all other admitted tasks and the new task meet their deadlines?
 - If no, reject the new task
 - Can handle both periodic and aperiodic tasks

Scheduling in Real-Time Systems

- We will only consider periodic systems
- Schedulable real-time system
 - Given
 - n periodic events
 - event i occurs within period T_i and requires C_i seconds
 - $U_i = \frac{C_i}{T_i}$ is called processor utilization
 - Then the load can only be handled if

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq 1$$

Execution Profile of Two Periodic Tasks

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	25	50
B(2)	50	25	100
•	•	•	•
•	•	•	•
•	•	•	•

Real-time Scheduling Algorithms

110

- **Earliest Deadline First Scheduling**
 - The task with the earliest deadline is chosen next
 - The deadline could be
 - Completion deadline
 - Starting deadline

Real-time Scheduling Algorithms

- **Rate Monotonic Scheduling**
 - Static Priority-driven scheduling
 - Priorities are assigned based on the period of each task
 - The shorter the period, the higher the priority
 - Task P has a period of T then rate of the task P is $1/T$
 - If C is the execution time for task P then
 - $U = \frac{C}{T}$ is called processor utilization for task P
 - For RMS the following inequality holds

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

n	$n(2^{1/n} - 1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
•	•
•	•
•	•
∞	$\ln 2 \approx 0.693$

RMS vs EDS

- For RMS: $\frac{c_1}{T_1} + \frac{c_2}{T_2} + \dots + \frac{c_n}{T_n} \leq n(2^{\frac{1}{n}} - 1)$ For EDS: $\frac{c_1}{T_1} + \frac{c_2}{T_2} + \dots + \frac{c_n}{T_n} \leq 1$
 - EDS can achieve greater overall processor utilization
- Still RMS is widely adopted for use in industrial applications:
 - The performance difference is small in practice. The upper bound for RMS is a conservative one and in practice upto 90% utilization is often achieved.
 - Most hard-RT system also have soft-RT components which are not used with RMS scheduling of hard-RT tasks
 - Stability is easier to achieve with RMS.

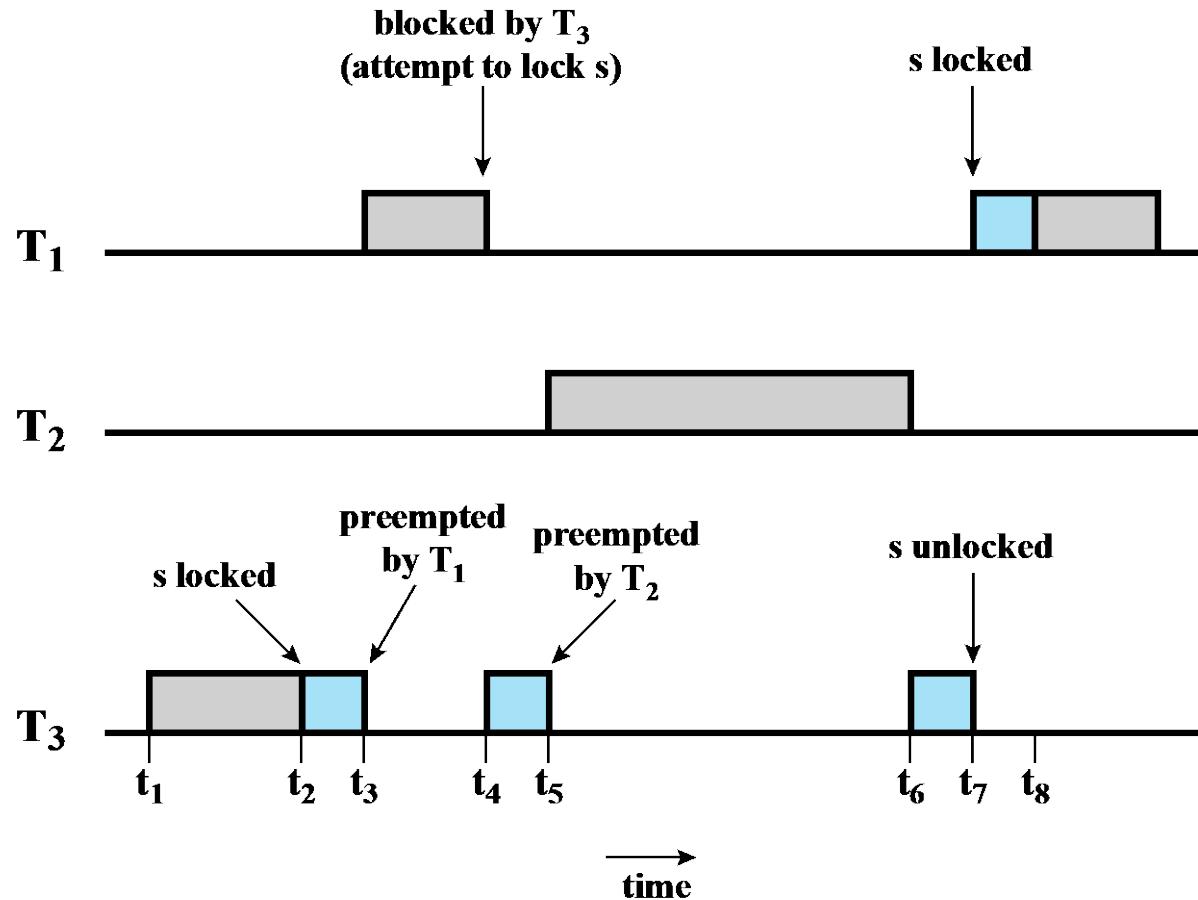
Priority Inversion

- Can occur in any priority-based preemptive scheduling scheme
- Particularly relevant in the context of real-time scheduling
- Best-known instance involved the Mars Pathfinder mission
- Occurs when circumstances within the system force a higher priority task to wait for a lower priority task

Unbounded Priority Inversion

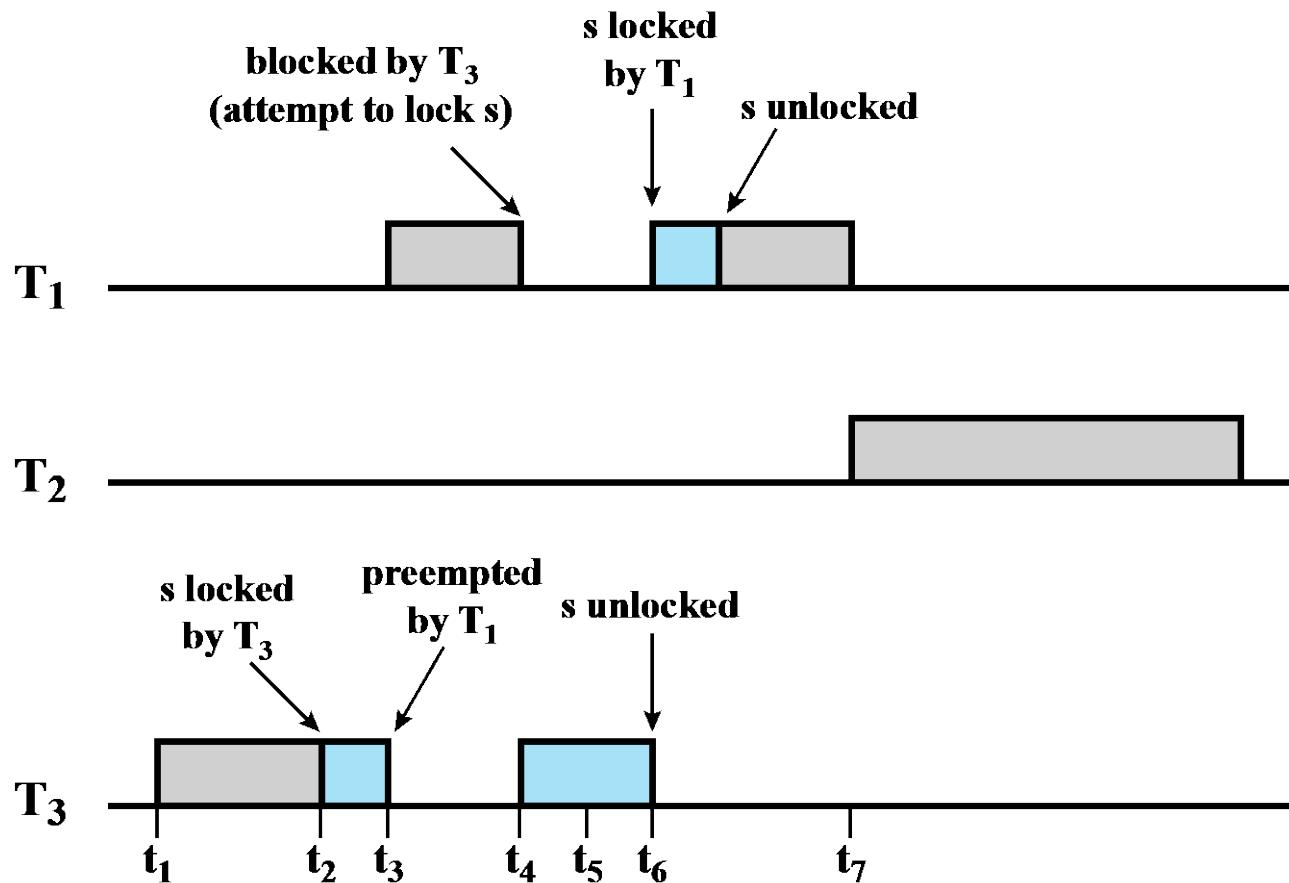
- the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks

Unbounded Priority Inversion



(a) Unbounded priority inversion

Priority Inheritance



Priority Ceiling

- A priority is associated with each resource.
 - The assigned priority is one level higher than the priority of its highest-priority user.
- Scheduler dynamically assigns the resource's priority to any process that access the resource
 - After the process is done with the resource the process priority returns to normal

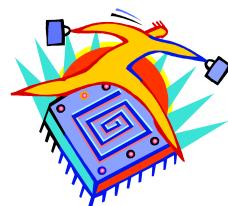
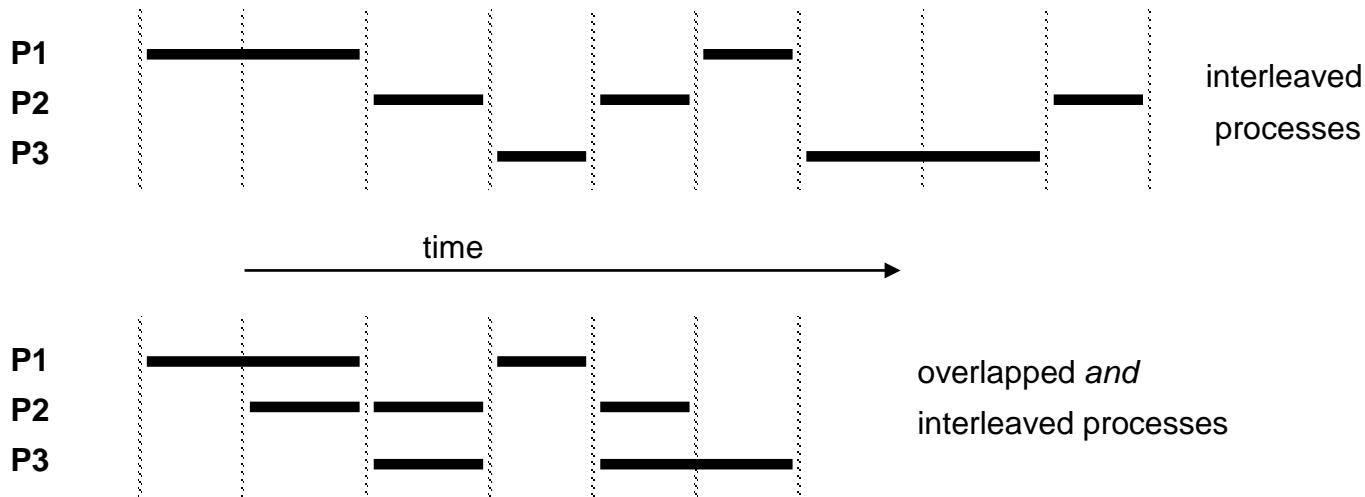
Concurrency: Mutual Exclusion and Synchronization

WEEK 5

Concurrency

118

- **Concurrency** is the concept of more than one process (or thread) operating at the same time.
- **Two kinds** of concurrency:



- The basic characteristic of multiprogramming concurrency is that the relative execution speed of processes cannot be predicted. Other processes affect it.
- This implies some non-determinism: a program may have different results when run at different times.

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

Concurrency: key terms

- **Atomic Operation** A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
- **Critical Section** A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
- **Deadlock** A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

Concurrency: key terms

120

- **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.

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

Problems from shared resource: example

P1

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

P2

P1

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

chin=x

P2

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

chin=y

P1

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

chin=y



Race conditions

122

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
 - the “loser” of the race is the process that updates last and will determine the final value of the variable
- Example 1: P1 and P2 shares a variable ‘a’
 - P1: $a=1$
 - P2: $a=2$
- Example 2: P3 and P4 shares variable ‘b’ and ‘c’
 - Initial values: ‘b=1’ and ‘c=2’
 - P3: $b=b+c$
 - P4: $c=b+c$



Operating system concerns

- Design and management issues raised by the existence of concurrency:
 - The OS must:
 - be able to keep track of various processes
 - allocate and de-allocate resources for each active process
 - protect the data and physical resources of each process against interference by other processes
 - ensure that the processes and outputs are independent of the processing speed



Process interaction

124

- Processes may interact in three ways:
 - **unaware** of each other but in **competition** for resources,
 - **indirectly aware** of each other by **sharing** some data (e.g. sharing an I/O buffer),
 - **directly aware** of each other by **communicating** for cooperation.
- We now look at each kind of interaction in turn.

Mutual Exclusion Mechanism

```

PROCESS 1 */
void P1
{
    while (true) {
        /* preceding code */;
        entercritical (Ra);
        /* critical section */;
        exitcritical (Ra);
        /* following code */;
    }
}

/* PROCESS 2 */
void P2
{
    while (true) {
        /* preceding code */;
        entercritical (Ra);
        /* critical section */;
        exitcritical (Ra);
        /* following code */;
    }
}

...
/* PROCESS n */
void Pn
{
    while (true) {
        /* preceding code */;
        entercritical (Ra);
        /* critical section */;
        exitcritical (Ra);
        /* following code */;
    }
}

```

- Mechanisms for mutual exclusion can be provided by two abstract functions
 - `entercritical(R)` and
 - `exitcritical(R)`.
- These functions may be implemented by the processes themselves, hardware, or the OS; they are used by the processes.
- The important questions are:
 - **how are they implemented?**
 - **how are they used?**

MUTUAL EXCLUSION APPROACHES

126

- **Responsibility** for mutual exclusion **may** be vested in:
 - **The processes**
 - The software approach
 - **The hardware**
 - Use special purpose atomic (uninterruptable) instructions
 - **The operating system**
 - Use semaphores
 - Use messages
- In any case, we can assume that the hardware provides elementary mutual exclusion for memory accesses.
 - Simultaneous access to the same location in main memory is prevented by the hardware.
 - Requests are serialised and granted in an unspecified order.

Hardware support

127

- **Interrupt disabling**
 - One can ensure mutual exclusion on a **uniprocessor** by **disabling interrupts** during a critical section.
 - This ensures mutual exclusion, but degrades performance because it limits the ability of the OS to interleave programs.
 - Interrupt disabling does not work on a multiprocessor.

```
While (true){  
    /* disable interrupts */;  
    /* critical section */;  
    /* enable interrupts */;  
    /* reminder */;  
}
```

Hardware support

- **Special machine instructions**
 - Remember that access to a memory location excludes any other access to that same location.
 - We can build general mutual exclusion methods by adding new **atomic instructions**.
 - **Atomic instructions cannot be interrupted.**

Mutual Exclusion: Hardware Support

129

- Compare&Swap Instruction

- also called a “compare and exchange instruction”
- a **compare** is made between a memory value and a test value
- if the values are the same a **swap** occurs
- carried out atomically

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



Compare and swap instruction

```
/* program mutual exclusion */
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));
}
```

Mutual Exclusion: Hardware Support

131

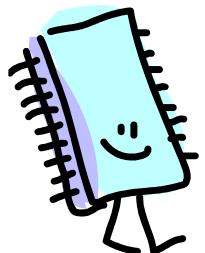
- Exchange Instruction

- Exchanges the contents of a register with that of a memory location
- carried out atomically

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

Exchange instruction

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





Hardware methods

- But there are some serious disadvantages:
 - **Busy-waiting** is employed; that is, while a process is waiting for access, it consumes processor time.
 - **Starvation** is possible: when a process leaves a critical section and more than one process is waiting, the selection of the next process is arbitrary and may indefinitely deny access to some process.
 - **Deadlock** is possible. Suppose that P2 has high priority and P1 has low priority. Suppose that P1 goes critical. Suppose that P1 is interrupted and gives control to P2, then P2 attempts to access the resource controlled by P1. Then P2 will wait for the resource, but P1 will never relinquish it because P1 is never dispatched (since P2 has high priority).

Mutual Exclusion: Software Approach

134

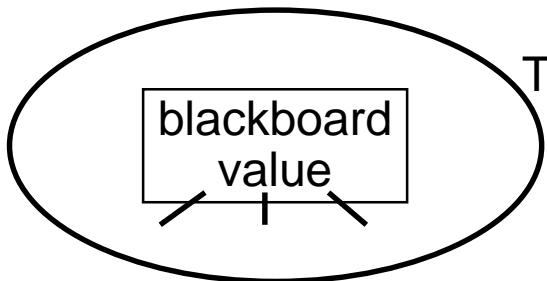
- Software approaches can be implemented for concurrent processes
 - Executing on single processor or multiple processor
 - Assumes elementary mutual exclusion at memory access level
 - No other support in hardware, OS or programming language is assumed



Mutual Exclusion: Software Approach

135

- In any case, we can assume that the hardware provides elementary mutual exclusion for memory accesses.
 - Simultaneous access to the same location in main memory is prevented by the hardware.
 - Requests are serialised and granted in an unspecified order.

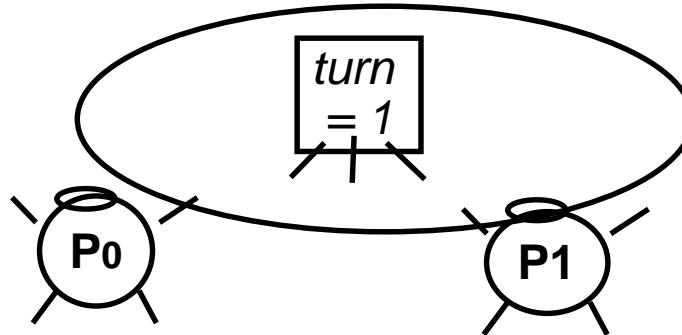


The elementary hardware protection is modelled by the **protocol igloo**: the entrance and interior of the igloo are small enough that at most **one person can enter** or **be in** the igloo at a time.

The igloo contains a **blackboard** for writing a message (usually a single value).

Software approach: Attempt 1

136



- A process P_i that wants to execute its critical section first enters the igloo and checks the blackboard.
- If process P_i finds the value i , then it leaves the igloo and goes critical.
- If not, then P_i repeats the entry and check until the blackboard shows i .
- When it finishes the critical section, it returns to the igloo and writes the number of the other process.
- **Mutual exclusion** is guaranteed.

```
global var turn:0..1;
```

```
process Pi
```

```
begin
```

```
  while turn !=i do <nothing>;  
  <critical section>  
  turn := 1-i
```

```
end.
```

Turn = 0

```
process Po
```

```
begin
```

```
  while turn !=0  
    do <nothing>;  
    <critical section>  
  turn := 1-0
```

```
end.
```

```
process P1
```

```
begin
```

```
  while turn !=1  
    do <nothing>;  
    <critical section>  
    <critical section>  
  turn := 1-1
```

```
end.
```

Software approach: Attempt 1

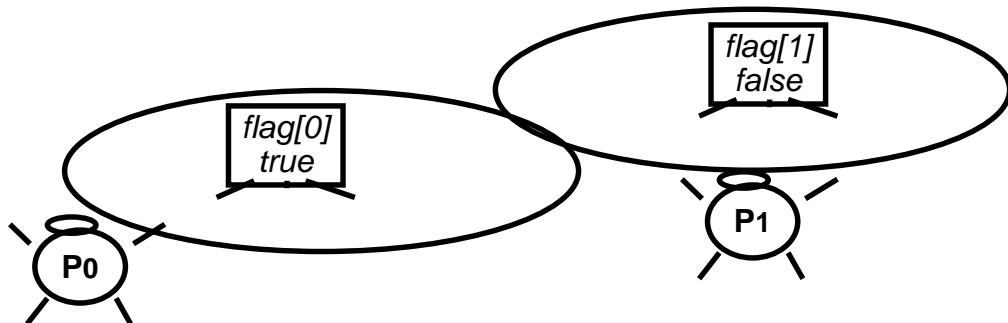
138

Problems:

- The procedure is “**busy-waiting**”, that is, the thwarted process can do nothing productive while it waits, yet it **keeps the CPU active** by continually checking.
- Pace of execution is **limited to the slower process**, because the executions must strictly alternate.
- **It does NOT prevent starvation**, because one process may fail, permanently blocking the other.

Software approach: Attempt 2

139



```
var flag : array[0..1] of Boolean;  
  
process Pi  
  while flag[1-i] do <nothing>;  
  flag[i]:=true;  
  <critical section>  
  flag[i]:=false;
```

- Each process should have its own key to its critical section, rather than rely on others as in last attempt.
- Process P_i is in its critical section if and only if we have $\text{flag}[i] = \text{true}$.
- Each process can **check the other** blackboard, but it can **update only its own** blackboard.

```
var flag : array[0..1] of Boolean;
```

```
process Pi
```

```
  while flag[1-i] do <nothing>;  
    flag[i]:=true;  
    <critical section>  
    flag[i]:=false;
```

140

```
process Po  
  while flag[1] do  
    <nothing>;  
    flag[0]:=true;  
    <critical section>  
    <critical section>  
    flag[0]:=false;
```

```
process P1  
  while flag[0] do  
    <nothing>;  
    flag[1]:=true;  
    <critical section>  
    <critical section>  
    flag[1]:=false;
```

$flag[0] = true$

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

$flag[1] = false$



Software approach: Attempt 3

- Attempt 2 failed because one process could change its state (flag)
 - after the other process has checked it,
 - before it enters its critical section.
- We can fix this by changing state **before** checking the other's flag.

```
var flag : array[0..1] of Boolean;  
  
process Pi  
  flag[i]:=true;  
  while flag[1-i] do <nothing>  
    <critical section>  
    flag[i]:=false;
```

- This enforces mutual exclusion.

```
var flag : array[0..1] of Boolean;  
  
process Pi  
  flag[i]:=true;  
  while flag[1-i] do <nothing>  
    <critical section>  
    flag[i]:=false;
```

process P₀
~~flag[0].=true;~~
while flag[1] **do**
 <nothing>
<critical section>
<critical section>
flag[0]:=false;

process P₁
~~flag[1].=true;~~
while flag[0] **do**
 <nothing>
<critical section>
<critical section>
flag[1]:=false;

flag[0] = true

flag[1] = true

Software approach: Attempt 4

- Attempt 3 fails because **a process sets its state without knowing the state of the other process.**
- Thus we change the procedure so that **each process gives the other some time to jump ahead of it**, i.e. it “backs off”, or defers to the other process.

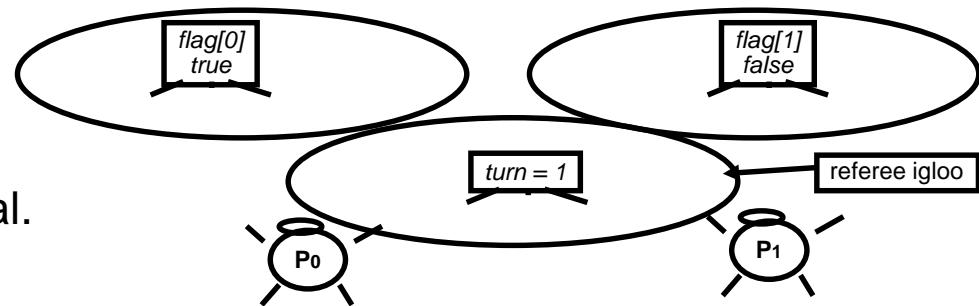
```
flag[i]:=true
while flag[1-i] do
  begin
    flag[i]:=false;
    <delay for a short time>
    flag[i]:=true
  end
  <critical section>
  flag[i]:=false
```

- **Mutual exclusion** is enforced.

Correct S/W approach: Dekker's Algorithm

144

We give the processes **the right to insist** that they can enter their critical sections. The processes can take **turns** to insist.



- Suppose that P0 wants to go critical.
 - it sets its own flag to true.
 - it checks the flag of P1:
 - if P1's flag is false, then P0 goes critical.
 - if P1's flag is true, then it checks the referee
 - if turn = 0, then P0 has the right to insist on taking a turn, and it checks P1's flag again.
 - if turn = 1, then P0 sets its flag to false, waits for its turn, resets its flag and goes critical.
 - after finishing its critical section, P0 sets turn = 1 and its flag to false transferring the right to insist to P1.

Dekker's Algorithm

```
var flag: array[0..1] of Boolean;  
turn:0..1;
```

```
process P0  
begin  
repeat  
    flag[0]:=true;  
    while flag[1] do  
        if turn=1 then  
            begin  
                flag[0]:=false;  
                while turn=1 do <nothing>  
                flag[0]:=true  
            end;  
            <critical section>  
            turn:=1;  
            flag[0]:=false;  
            <remainder of the process>  
        forever  
end;
```

```
main program  
begin  
    flag[0]:=false;  
    flag[1]:=false;  
    turn:=1;  
    parbegin  
        P0;  
        P1  
    parend  
end
```

```
process P1  
begin  
repeat  
    flag[1]:=true;  
    while flag[0] do  
        if turn=0 then  
            begin  
                flag[1]:=false;  
                while turn=0 do <nothing>  
                flag[1]:=true  
            end;  
            <critical section>  
            turn:=0;  
            flag[1]:=false;  
            <remainder of the process>  
        forever  
end;
```

Peterson's Algorithm

146

- Mutual exclusion is enforced:
- Deadlock is prevented:

```
var flag: array[0..1] of Boolean;  
turn:0..1;
```

```
procedure P0;  
begin  
repeat  
  flag[0]:=true;  
  turn:=1;  
  while flag[1] and turn=1 do <nothing>  
  <critical section>  
  flag[0]:=false;  
  <remainder>  
forever  
end
```

```
procedure P1;  
begin  
repeat  
  flag[1]:=true;  
  turn:=0;  
  while flag[0] and turn=0 do <nothing>  
  <critical section>  
  flag[1]:=false;  
  <remainder>  
forever  
end
```

Peterson's Algorithm

- **Mutual exclusion is enforced:** Suppose that P0 wants to enter its critical section. First, it sets flag[0] to true; then it is impossible for P1 to enter its critical section. If P1 is already in its critical section, then flag[1] must be true, and so P0 cannot get to its critical section.
- **Deadlock is prevented:** Suppose that both P0 and P1 are blocked; this can only occur indefinitely if both are in the **while** loops. But the conditions on these **while** loops are inconsistent: it is not possible to have turn=1 and turn=0 at the same time.

OS and PL approaches to concurrency

148

- Operating systems and programming languages offer several mechanisms to support concurrency
 - Semaphore
 - Monitor
 - Message Passing



Semaphores

149

- A set of processes can cooperate using simple signals called **semaphores**.
 - A process **transmits** a signal via a semaphore **s** by executing an atomic procedure **semSignal(s)**.
 - A process **receives** a signal via a semaphore **s** using an atomic procedure **semWait(s)**; if the corresponding signal has not yet been sent, then the process is blocked until transmission takes place.

A Definition of 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 */;
    }
}
```

- semWait/semSingal are atomic

A Definition of Binary Semaphore Primitives

151

```
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 */;
    }
}
```

Mutual Exclusion Using Semaphores

152

```
/* 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));
}
```

Producer/Consumer Problem

General Statement:

one or more producers are generating data and placing these in a buffer

a single consumer is taking items out of the buffer one at a time

only one producer or consumer may access the buffer at any one time

The Problem:

ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

154

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

155

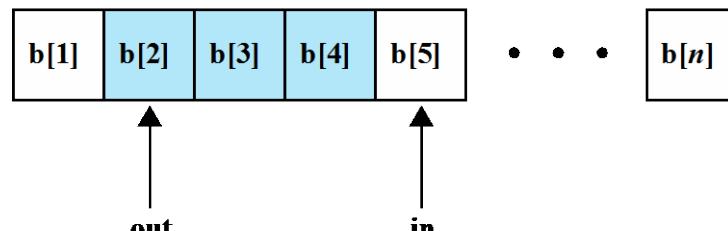
```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

29/10/2019

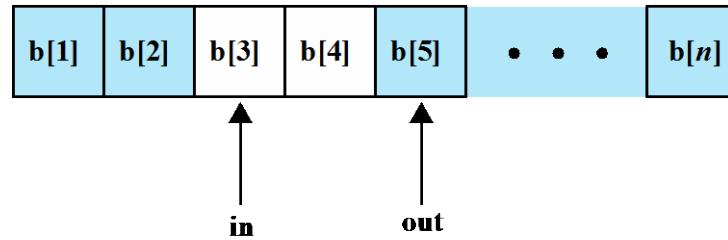
```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Producer/Consumer problem with finite buffer

156



(a)



(b)

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Two Possible Implementations of Semaphores

158

```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0)
    */
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else
        allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(b) Interrupts

Monitor

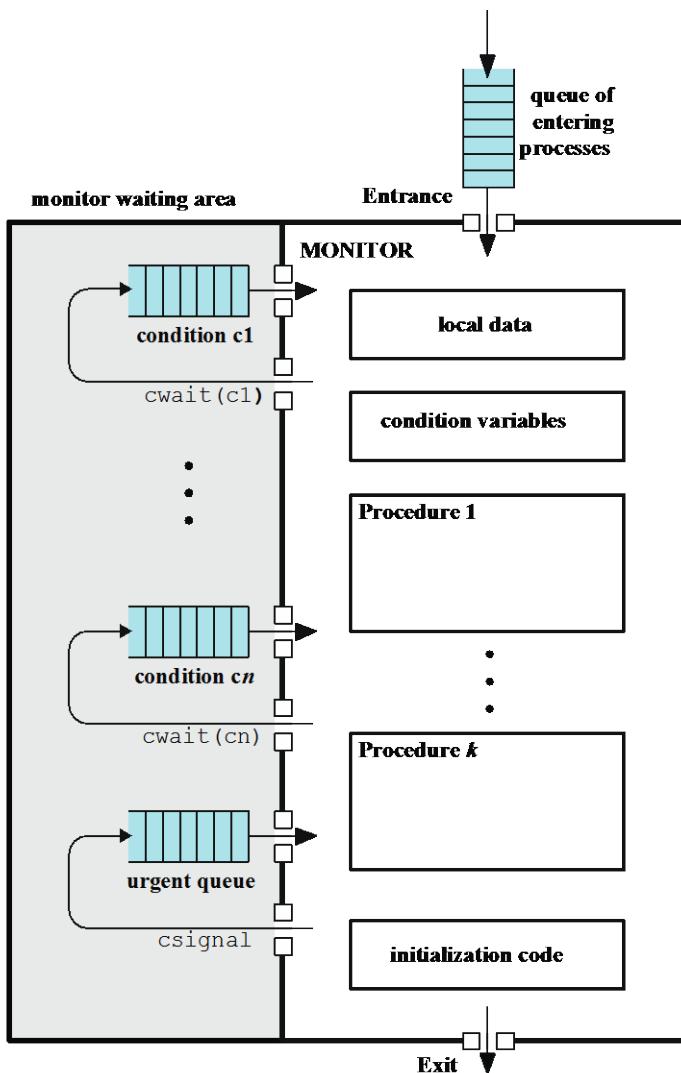
159

Monitor is a software module consisting of one or more procedures, an initialization sequence, and local data

Local data variables are accessible only by the monitor's procedures and not by any external procedure

Process enters monitor by invoking one of its procedures

Only one process may be executing in the monitor at a time



A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

```

/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];
int nextin, nextout;                                /* space for N items */
int count;                                         /* buffer pointers */
int cond notfull, notempty;                         /* number of items in buffer */
                                                /* condition variables for synchronization */

void append (char x)
{
    if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                  /* resume any waiting consumer */
}
void take (char x)
{
    if (count == 0) cwait(notempty);    /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    csignal(notfull);                /* one fewer item in buffer */
                                      /* resume any waiting producer */
}
                                                /* monitor body */
{
    nextin = 0; nextout = 0; count = 0;           /* buffer initially empty */
}

void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}
void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}

```

Monitor VS Semaphore

- Monitors
 - Mutual exclusion: enforced by the construct (monitor) itself
 - Synchronization: Programmers responsibility - to signal (cwait/csignal) appropriately
 - Synchronization is confined to monitor
 - Easy to verify and debug
- Semaphores
 - It is programmers responsibility to
 - Enforce mutual exclusion
 - Synchronization
 - Programming/debugging is difficult



Mesa versus Hoare Monitors

- What should happen when csignal() is called?
 - No waiting threads => the signaler continues and the signal is effectively lost (unlike what happens with semaphores).
 - If there is a waiting thread, one of the threads starts executing, others must wait
- **Mesa-style:** (Nachos, Java, and most real operating systems)
 - The thread that signals keeps the lock (and thus the processor).
 - The waiting thread waits for the lock.
- **Hoare-style:** (most textbooks)
 - The thread that signals gives up the lock and the waiting thread gets the lock.
 - When the thread that was waiting and is now executing exits or waits again, it releases the lock back to the signalling thread.



Readers/Writers Problem

164

- A data area is shared among many processes
 - some processes only read the data area, (readers) and some only write to the data area (writers)
- Conditions that must be satisfied:
 1. any number of readers may simultaneously read the file
 2. only one writer at a time may write to the file
 3. if a writer is writing to the file, no reader may read it

Concurrency: Deadlock and Starvation

WEEK 6

Deadlock

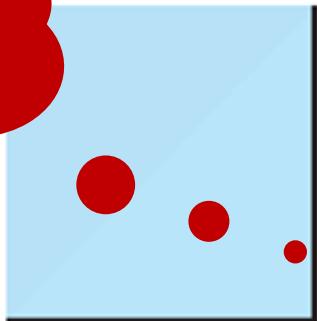
- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent – none of the events is ever triggered
- No efficient solution



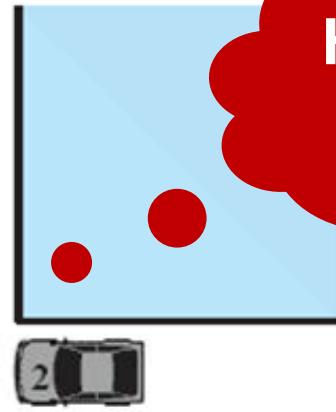
Actual Deadlock

167

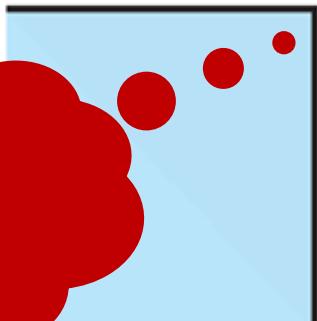
HALT until
D is free



HALT until
C is free



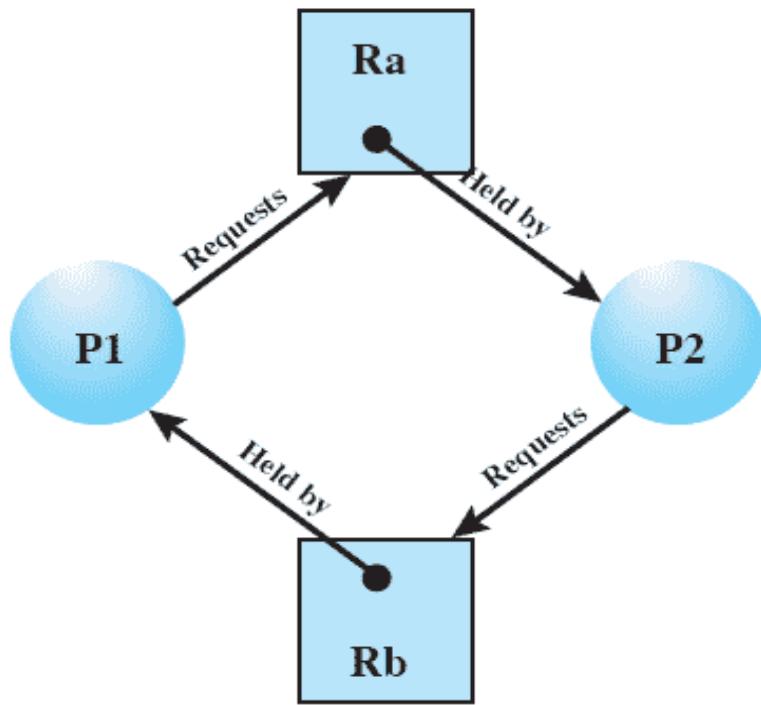
HALT until
A is free



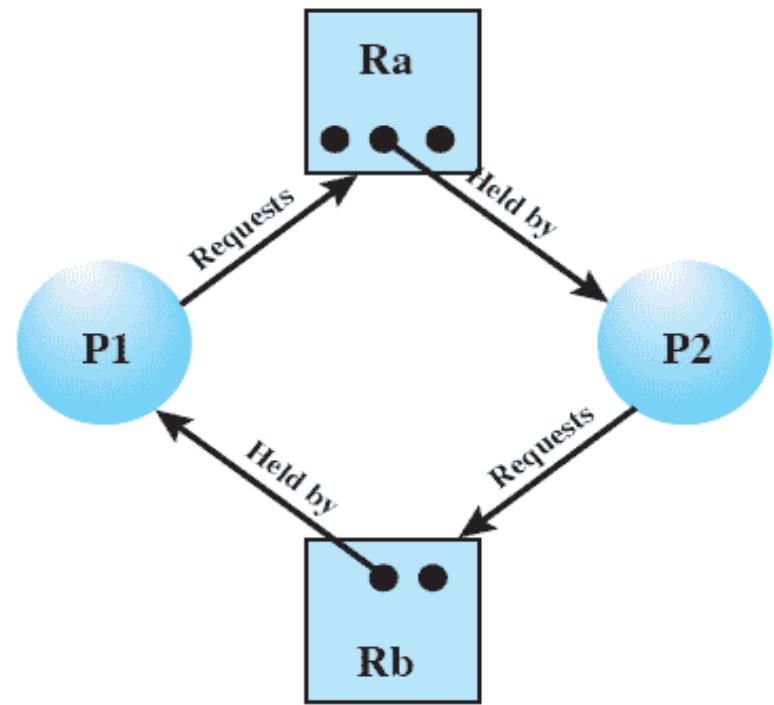
HALT until
B is free

Resource Allocation Graphs

168



(c) Circular wait



(d) No deadlock

Conditions for Deadlock

169

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 Pre-emption

- no resource can be forcibly removed from 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

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 Pre-emption

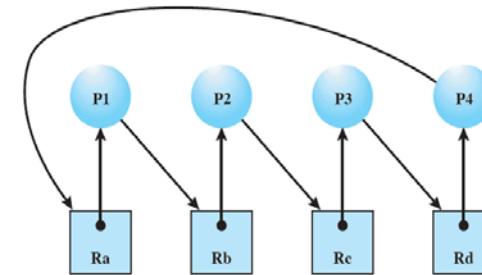
- no resource can be forcibly removed from a process holding it

- These conditions are quite desirable?
- The first three conditions are necessary but not sufficient for a deadlock to exist

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

Conditions for Deadlock



171

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
<ul style="list-style-type: none">only one process may use a resource at a time	<ul style="list-style-type: none">a process may hold allocated resources while awaiting assignment of others	<ul style="list-style-type: none">no resource can be forcibly removed from a process holding it	<ul style="list-style-type: none">a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

- “Circular Wait” is actually a potential consequence of the first three conditions
- The unresolvable circular wait is in fact the definition of deadlock

Dealing with deadlock

172

- **Prevent deadlock**
 - Adopt a policy that eliminates one of the conditions (conditions 1 through 4).
- **Avoid deadlock**
 - Make the appropriate dynamic choices based on the current state of resource allocation.
- **Detect** the presence of deadlock
 - When conditions 1 through 4 hold and take action to recover.

Deadlock condition prevention

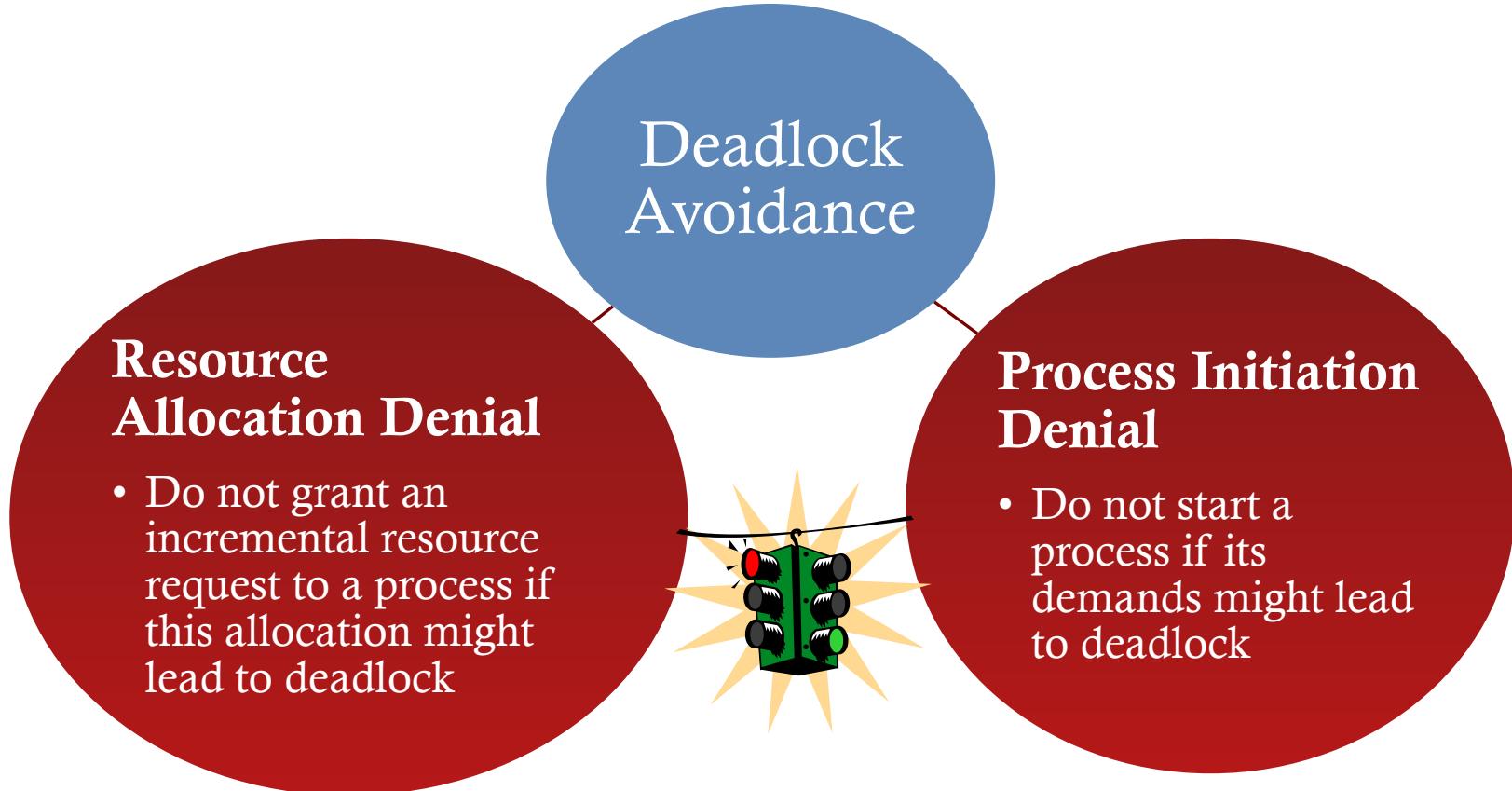
- **Mutual Exclusion:**
 - In general this can not be disallowed
 - If access to a resource requires mutual exclusion then it must be supported by the OS
- **Hold and Wait:**
 - Require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
 - Inefficient
 - Process may be held up for a long time
 - Resources underutilized

Deadlock condition prevention

- **No Preemption**
 - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
 - OR OS may preempt the second process and require it to release its resources
 - Practical when resources state can be easily saved and restored later
- **Circular Wait**
 - Define a linear ordering of resource types
 - If a process has been allocated resources of type R_i then it may subsequently request resource R_j if $i < j$.

Two approaches to deadlock avoidance

175



Resource Allocation Denial

176

- Referred to as the **banker's algorithm**
- Consider a system with a fixed number of processes and a fixed number of resources. At any time a process may have zero or more resources allocated to it.
- **State** of the system reflects the current allocation of resources to processes
 - Two vectors, Resource and Available, and
 - Two matrices, Claim and Allocation
- **Safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion).
- **Unsafe state** is a state that is not safe.

Example state

- State of a system consisting of four processes and three resources
- Allocations have been made to the four processes
- But is it safe?

	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

Deadlock Detection

178

- Deadlock prevention strategies are very conservative
 - Limit access to resources by imposing restrictions on processes
- Deadlock detection strategies do the opposite
 - Resource requests are granted whenever possible

Deadline detection algorithms

- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur.
- Checking at each resource request
 - Advantages:
 1. It leads to early detection
 2. The algorithm is relatively simple because it is based on incremental changes to the state of the system.
 - Disadvantage
 - Frequent checks consume considerable processor time.

Deadline detection algorithm example

180

	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

Available vector

Deadline detection algorithm example

	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

Deadlock: Involved process are P1 and P2

Deadlock recovery

- **Abort all deadlocked processes.**
 - One of the most common, if not the most common, solution adopted in operating systems.
- **Back up each deadlocked process to some previously defined checkpoint, and restart all processes.**
 - This requires that rollback and restart mechanisms be built in to the system.
 - The risk in this approach is that the original deadlock may recur but the nondeterminacy of concurrent processing may ensure that this does not happen.
- **Successively abort deadlocked processes until deadlock no longer exists.**
 - The order in which processes are selected for abortion should be on the basis of some criterion of minimum cost. After each abortion, the detection algorithm must be reinvoked to see whether deadlock still exists.
- **Successively preempt resources until deadlock no longer exists.**
 - As (3) above, a cost-based selection (*for example?*) should be used, and reinvocation of the detection algorithm is required after each preemption.
 - A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

Deadlock recovery

- **Possible Selection Criteria for aborting processes for deadlock recovery**
 - Choose the process
 - Least amount of time consumed so far
 - Lease amount of output produced so far
 - Most estimated time remaining
 - Least total resources allocated so far
 - Lowest priority

Dining Philosophers Problem

184

Think

...

Eat

...

Think

...

Eat

...

(and spaghetti is the best)

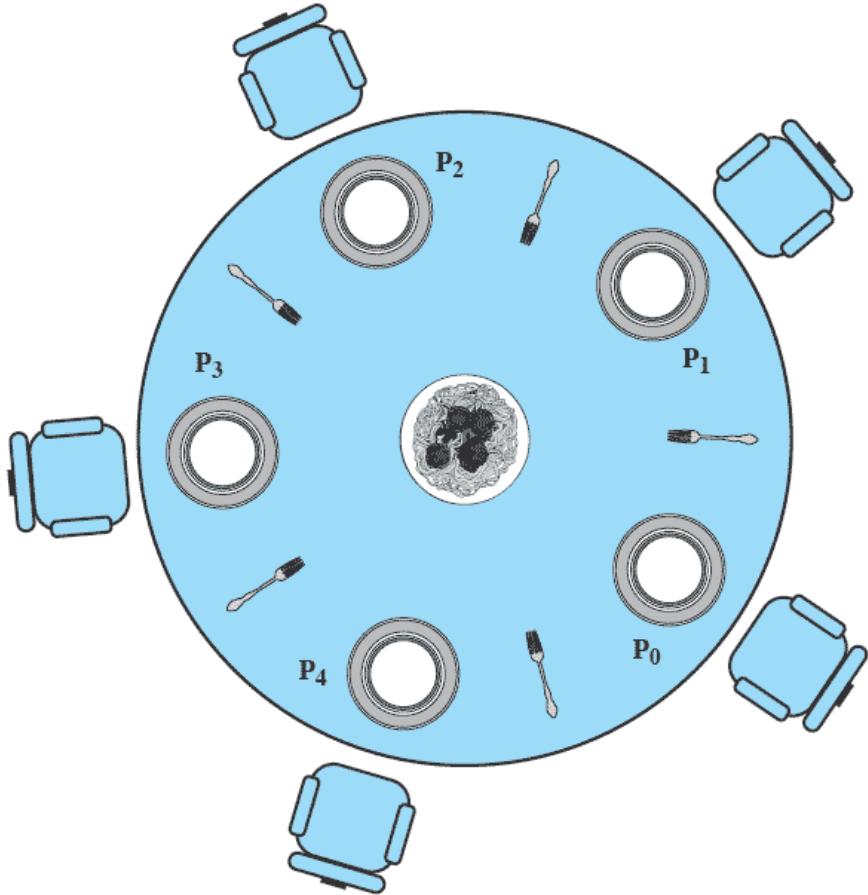


Figure 6.11 Dining Arrangement for Philosophers

Using semaphores (1)

```
/* 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

Using semaphores (2)

```
/* 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));
}
```

Memory Management

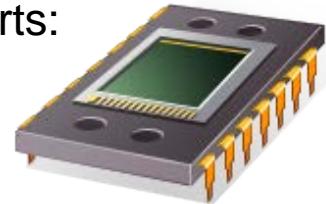
WEEK 7 + 8

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

Memory management

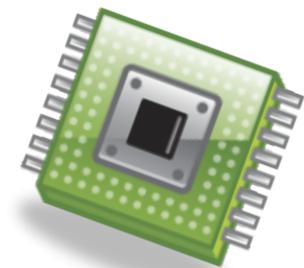
- In a uni-programming system, main memory is divided into two parts:
 - one part for the operating system (resident monitor, kernel)
 - one part for the program currently being executed.
- In a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes.
 - The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.
- Effective memory management is vital in a multiprogramming system.
 - If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle.
 - Thus memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.



Memory management terms

189

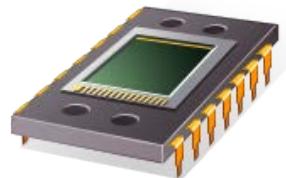
Frame	A fixed-length block of main memory.
Page	A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.
Segment	A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).



Memory management requirements

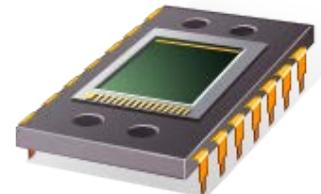
190

- Memory management is intended to satisfy the following requirements:
 1. Relocation
 2. Protection
 3. Sharing
 4. Logical organization
 5. Physical organization



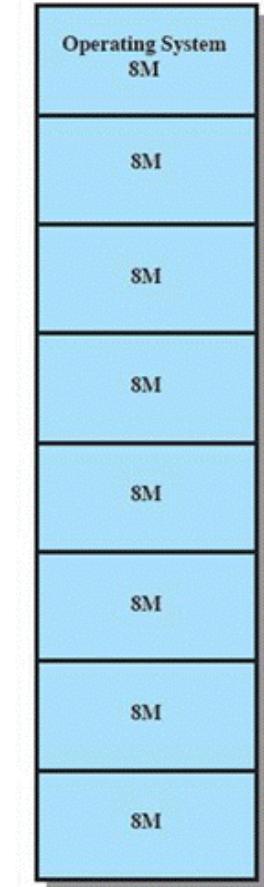
Memory partitioning

- Memory management brings processes into main memory for execution by the processor
 - Involves virtual memory
 - Based on segmentation and paging
- Partitioning
 - Used in several variations in some now-obsolete operating systems
 - Does not involve virtual memory



Fixed partitioning

- Equal-size partitions
 - 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 out a process if all partitions are full and no process is in the Ready or Running state



(a) Equal-size partitions

Fixed partitioning: Disadvantages

- A program may be too big to fit in a partition
 - Program needs to be designed with the use of overlays
- Main memory utilization is inefficient
 - Any program, regardless of size, occupies an entire partition
 - ***Internal fragmentation***
 - Wasted space due to the block of data loaded being smaller than the partition



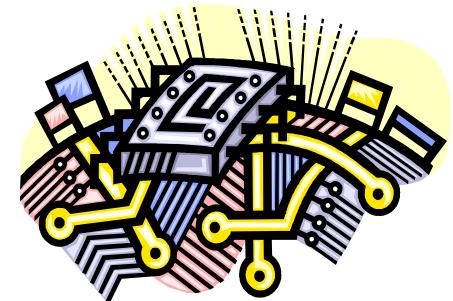
Equal VS unequal-size partitions

- **Equal Sized Partitions:**
 - Simple and require minimal OS software and processing overhead
- **Unequal-Sized Partitions:**
 - **Advantages:**
 - Provides a degree of flexibility
 - **Disadvantages:**
 - The number of partitions specified at system generation time limits the number of active processes in the system
 - Small jobs will not utilize partition space efficiently

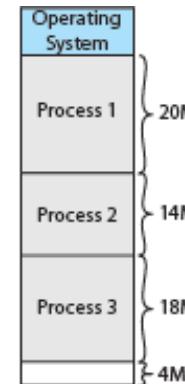
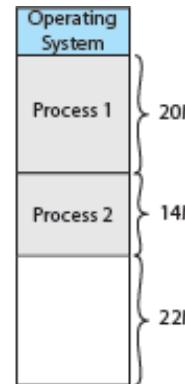
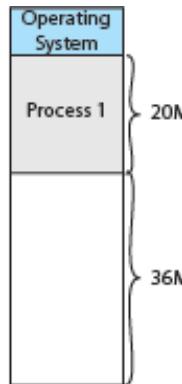


Dynamic partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as it requires
- This technique was used by IBM's mainframe operating system, OS/MVT



Effect of dynamic partitioning

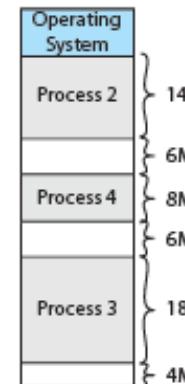
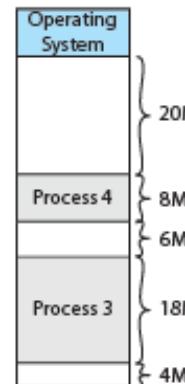
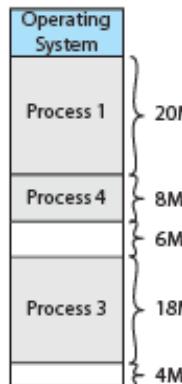
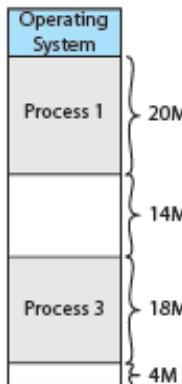


(a)

(b)

(c)

(d)



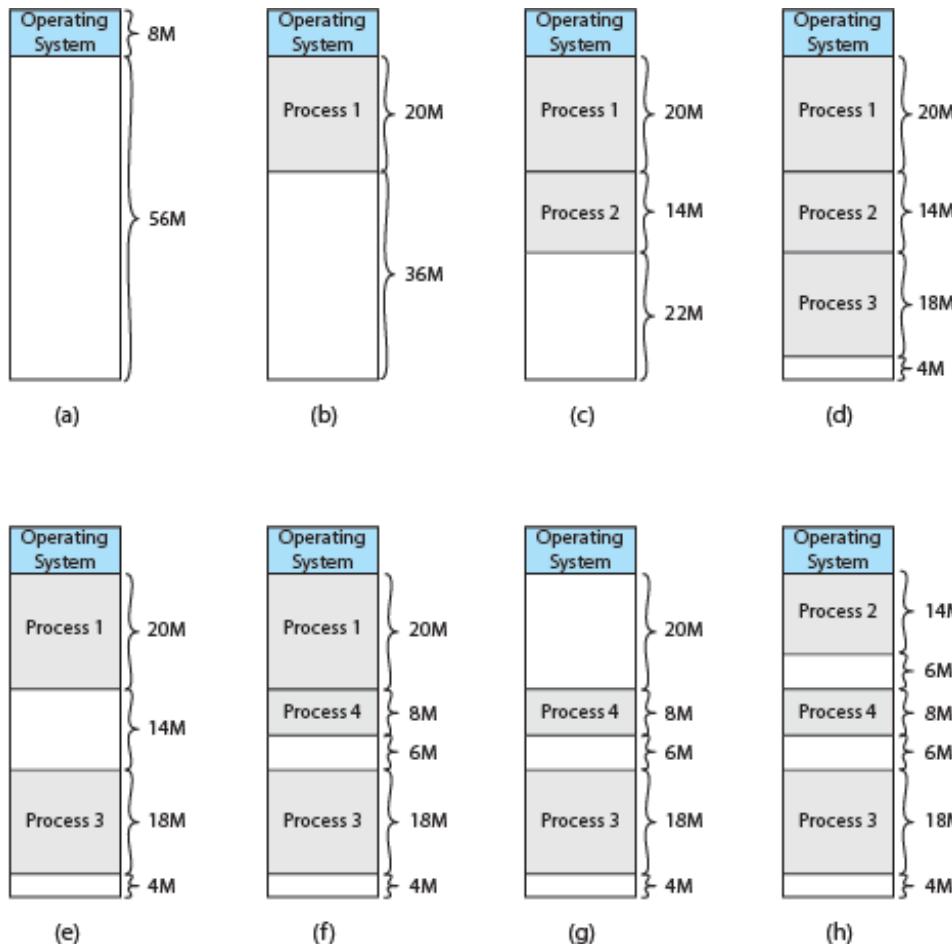
(e)

(f)

(g)

(h)

Effect of dynamic partitioning



external fragmentation:
memory that is external to all partitions becomes increasingly fragmented

Dynamic partitioning

External Fragmentation

- memory becomes more and more fragmented
- memory utilization declines

Compaction

- technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous
- free memory is together in one block
- time consuming and wastes CPU time

Placement algorithms

Best-fit

- Chooses the block that is closest in size to the request

First-fit

- Begins to scan memory from the beginning and chooses the first available block that is large enough

Next-fit

- Begins to scan memory from the location of the last placement and chooses the next available block that is large enough

Memory configuration example

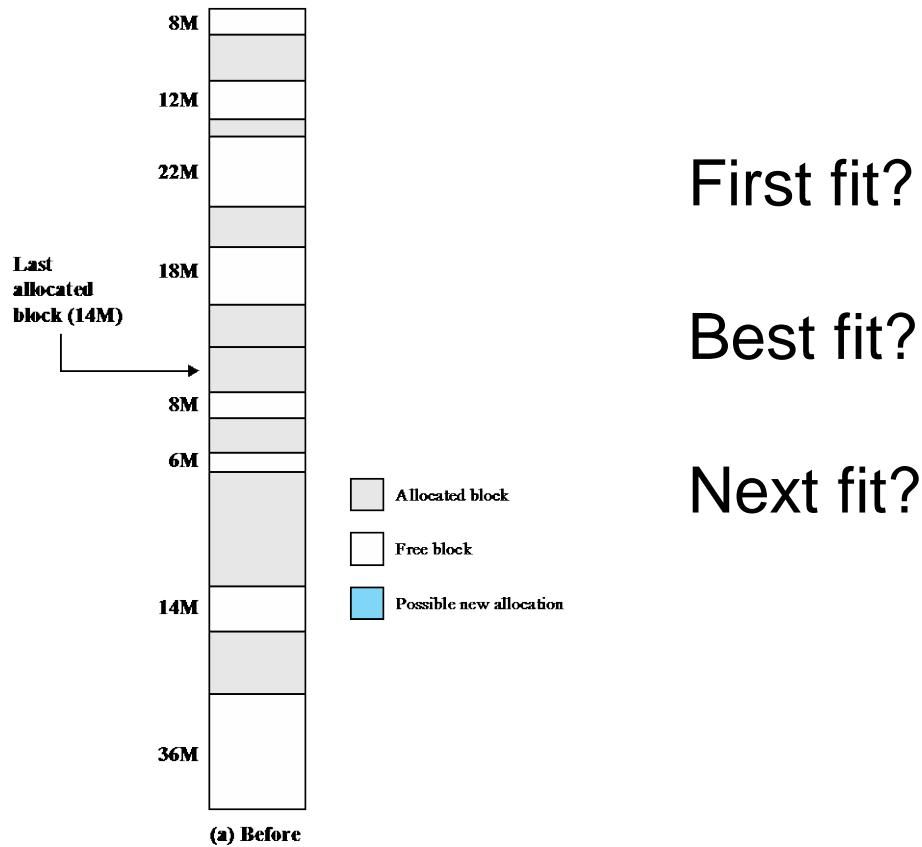


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Addresses

Logical

- reference to a memory location independent of the current assignment of data to memory

Relative

- address is expressed as a location relative to some known point

Physical or Absolute

- actual location in main memory

Relocation

- Programs that employ relative addresses in memory are loaded using **dynamic run-time loading**
- Typically, all of the memory references in the loaded process are relative to the origin of the program.
- Thus a hardware mechanism is needed for translating relative addresses to physical main memory addresses at the time of execution of the instruction that contains the reference.

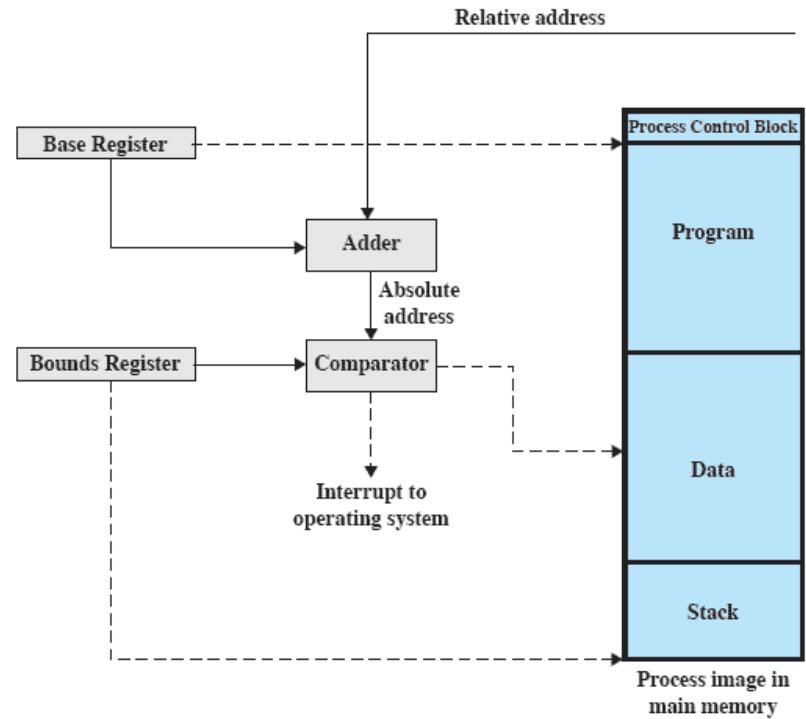


Figure 7.8 Hardware Support for Relocation

Paging

203

- Partition memory into equal fixed-size chunks that are relatively small
- Process is also divided into small fixed-size chunks of the same size

Pages

- chunks of a process

Frames

- available chunks of memory

Assigning process to free frames

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

(a) Fifteen Available Frames

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

(b) Load Process A

Main memory
0 A.0
1 A.1
2 A.2
3 A.3
4 B.0
5 B.1
6 B.2
7
8
9
10
11
12
13
14

(c) Load Process B

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

(d) Load Process C

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

(e) Swap out B

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

(f) Load Process D

Page table

205

- Maintained by operating system for each process
- Contains the frame location for each page in the process
- Processor must know how to access for the current process
- Used by processor to produce a physical address



Data structures

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	

(f) Load Process D

0	0
1	1
2	2
3	3

Process A
page table

0	—
1	—
2	—

Process B
page table

0	7
1	8
2	9
3	10

Process C
page table

0	4
1	5
2	6
3	11
4	12

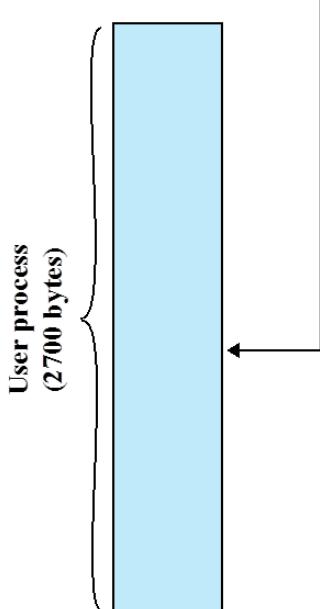
Process D
page table

13
14

Free frame
list

Logical addresses

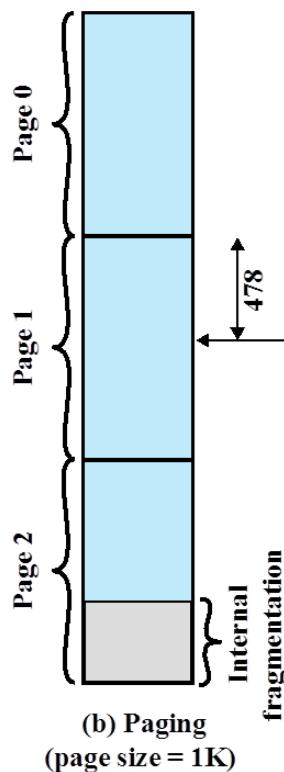
Relative address = 1502
`0000010111011110`



(a) Partitioning

Logical address =
 Page# = 1, Offset = 478

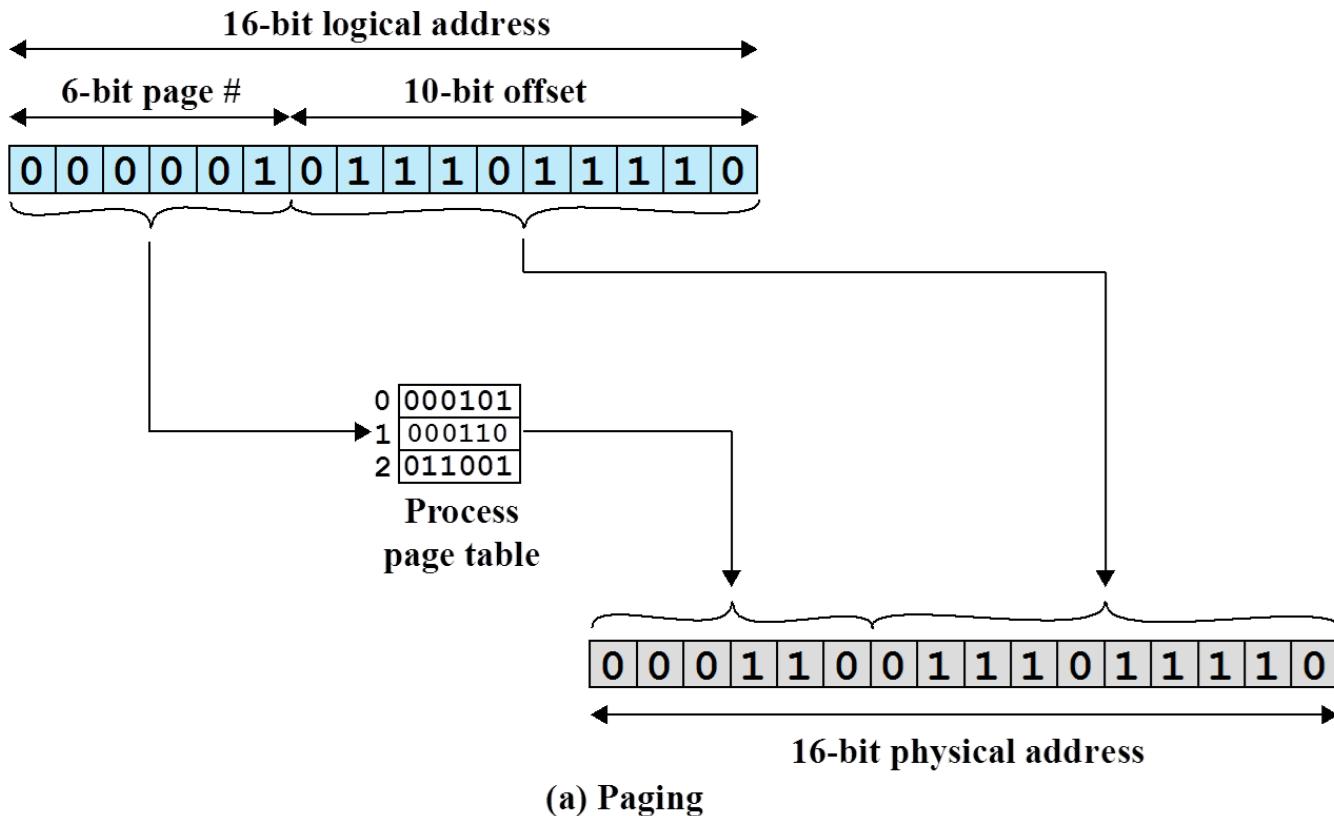
`0000010111011110`



(b) Paging
 (page size = 1K)

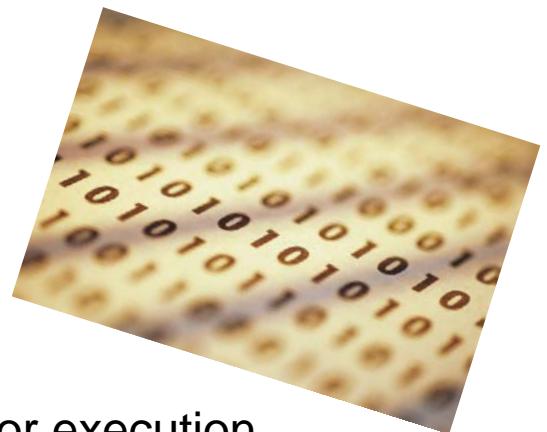
- Consider page/frame size a power of 2
- 16-bit addresses are used, and the page size is 1K (1,024 bytes)
 - 1111101000
- The relative address 1502, in binary form, is **0000010111011110**.
- With a page size of 1K, an offset field of 10 bits is needed, leaving 6 bits for the page number.
 - Thus a program can consist of a maximum of $2^6 = 64$ pages of 1K bytes each.
- Relative address 1502 corresponds to an offset of 478 (0111011110) on page 1 (000001), which yields the same 16-bit number, 0000010111011110.

Logical to physical address translation: Paging



Segmentation

- A program can be subdivided into segments
 - **May vary in length**
 - There is a maximum length
- Addressing consists of two parts:
 - Segment number
 - An offset
- Similar to dynamic partitioning
 - Requires all segments to be in main memory for execution
 - No internal fragmentation, but suffers from external fragmentation
 - External fragmentation should be less
 - Contrast: a program may occupy more than one partition which need not to be contiguous...



Logical addresses: Revisited

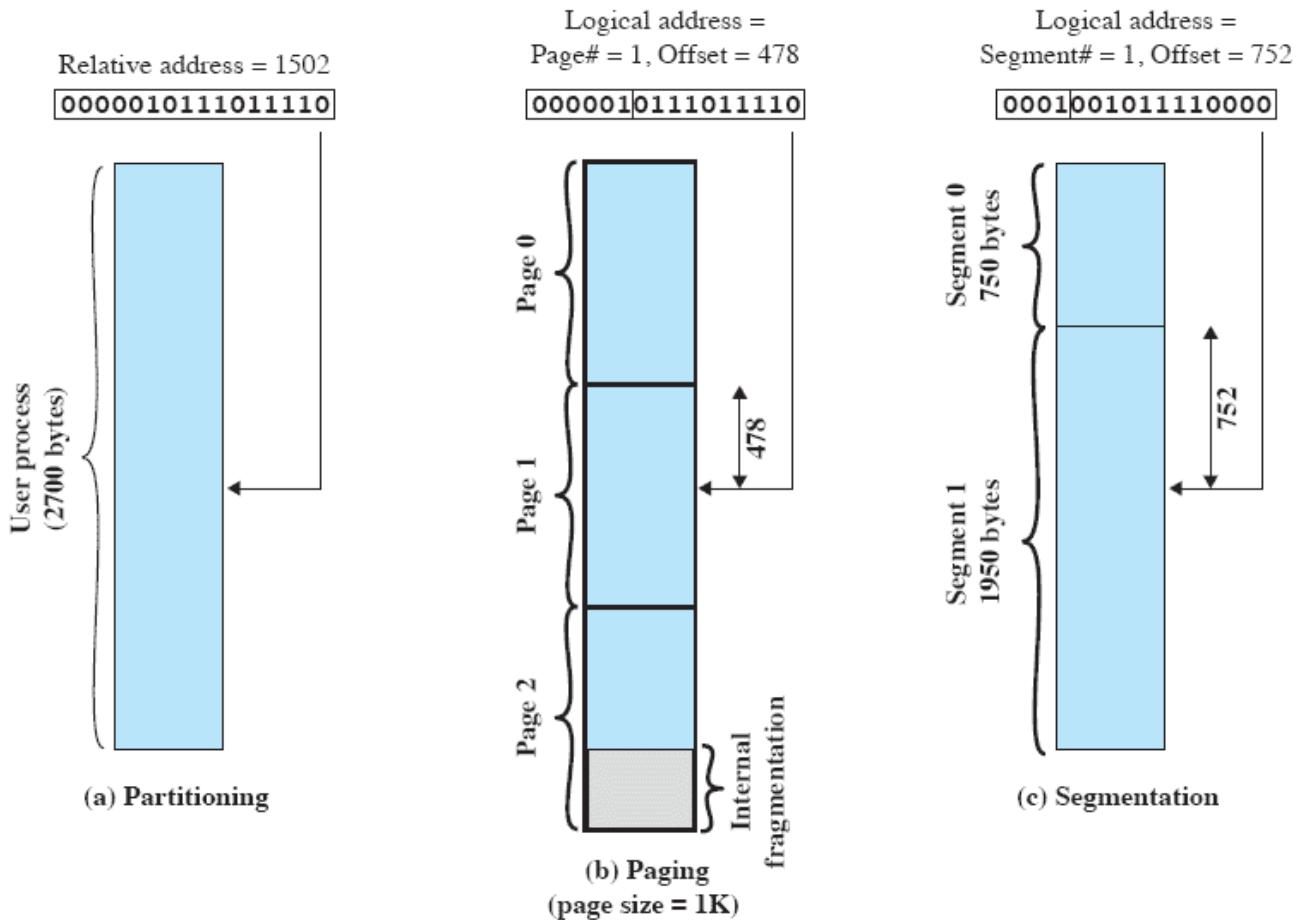
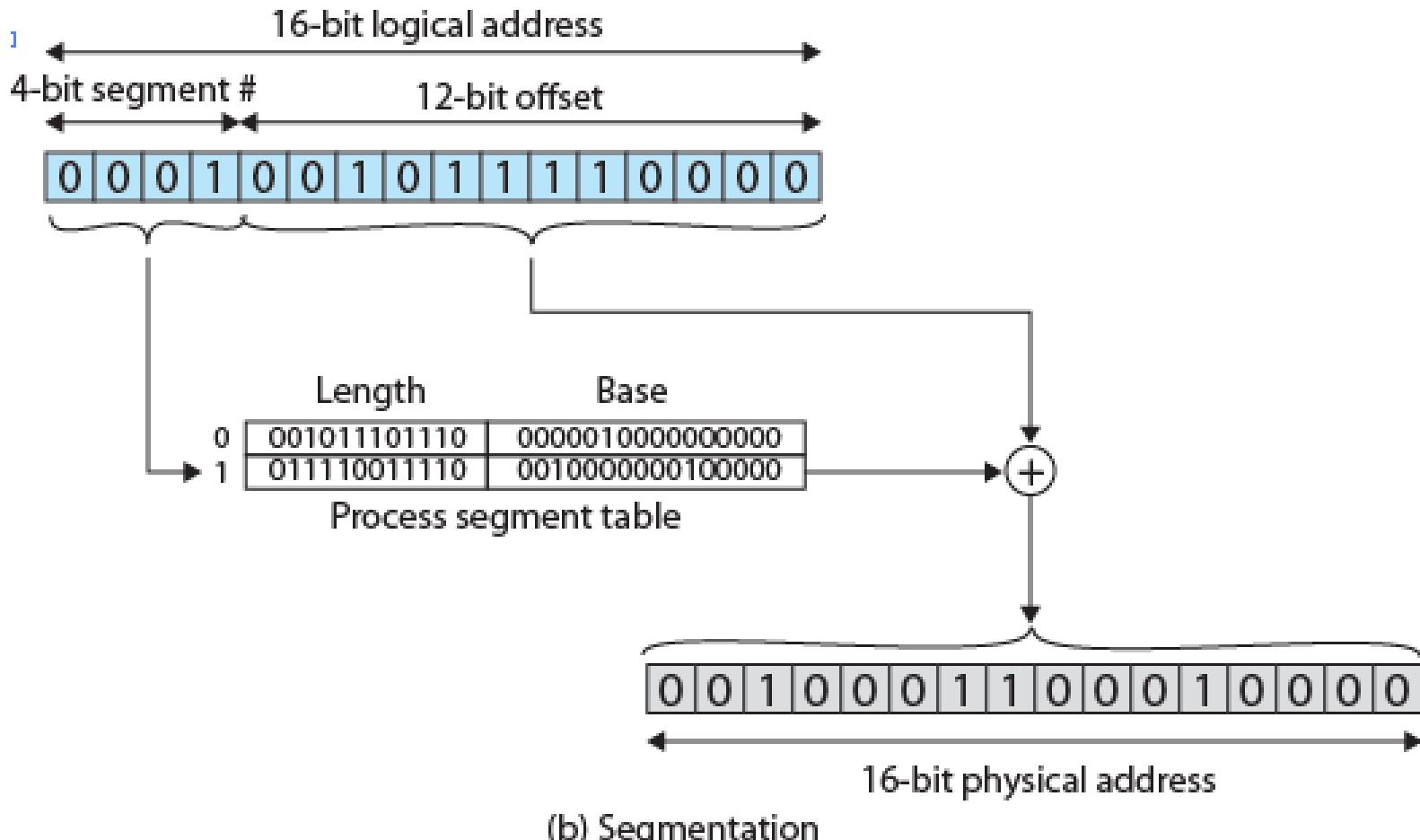


Figure 7.11 Logical Addresses

Logical to physical address translation: Segmentation



Virtual memory

212

- A **virtual memory system** ensures that the currently executing parts of a process are in memory, when required. It is possible that the parts which are not currently executing or accessed are on disk (secondary storage).
- Virtual memory gives the machine a **very large** (virtual) address space

Virtual memory: Advantages

- **Advantages** of virtual memory include:
 1. **More processes** may be in memory at once.
 - The scheduler has a larger choice of processes which are ready to run, and the probability of idleness is decreased.
 2. **More memory**: it is possible for a process to be larger than main memory.
 - The programmer need not be concerned with the details of address translation: the *virtual memory* appears the same as *real memory*.
- There is some overhead required for maintaining virtual memory
 - Every time a page is swapped in or out, the process may be suspended and the page or segment tables need to be updated
- However, virtual memory has **proved to be a big gain in efficiency**.
 - The user is freed from the tight constraints of main memory.
 - Multiprogramming becomes more effective.

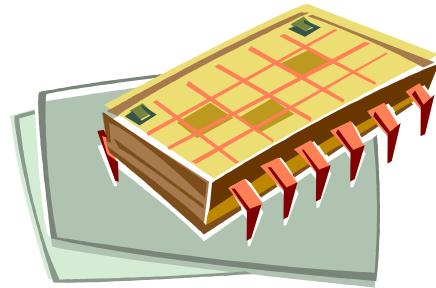


Real and virtual memory

214

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

Thrashing

215

A state in which the system spends most of its time swapping process pieces rather than executing instructions

To avoid this, the operating system tries to guess, based on recent history, which pieces are least likely to be used in the near future

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
- Avoids thrashing



Support needed for virtual memory

217

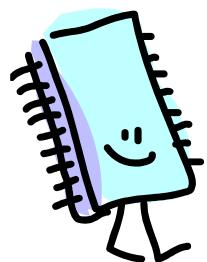
For virtual memory to be practical and effective:

- Hardware must support paging and segmentation
- Operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory

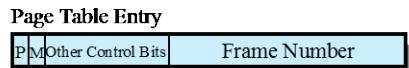
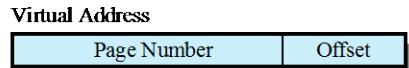
Paging

218

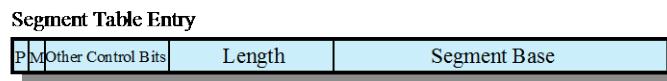
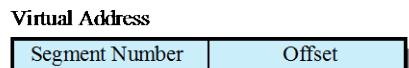
- The term *virtual memory* is usually associated with systems that employ paging
- Use of paging to achieve virtual memory was first reported for the Atlas computer
- Each process has its own page table
 - each page table entry (PTE) contains the frame number of the corresponding page in main memory



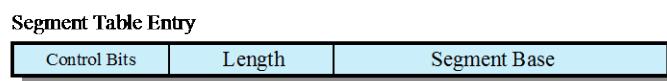
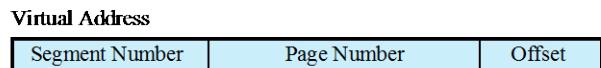
Memory management formats



(a) Paging only



(b) Segmentation only

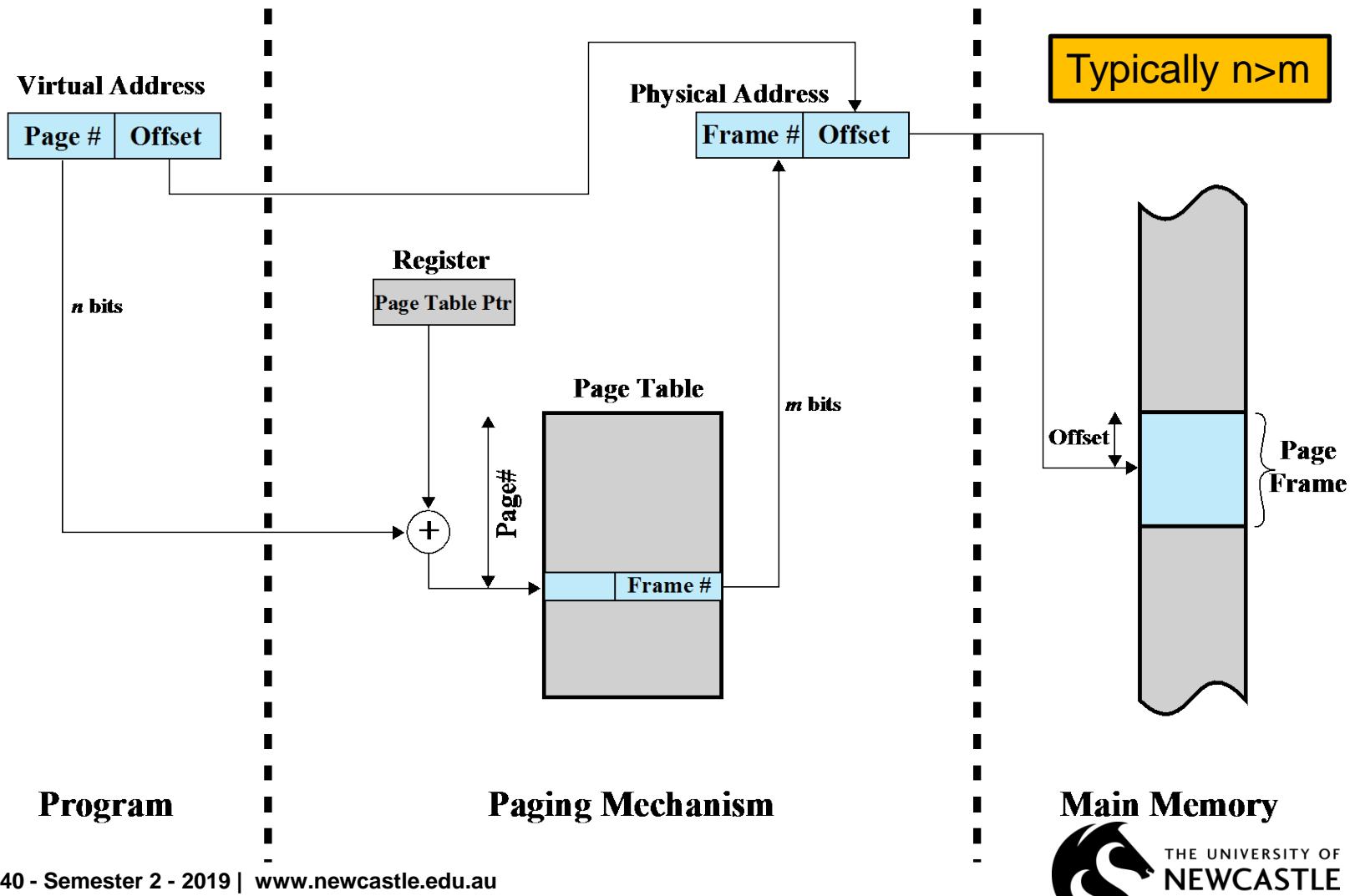


P = present bit
M = Modified bit

(c) Combined segmentation and paging

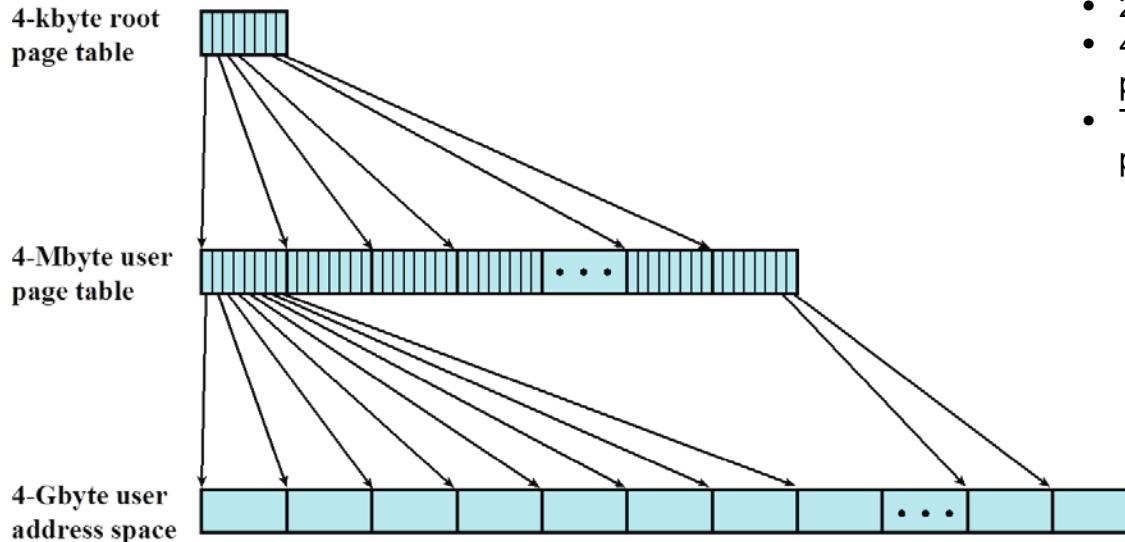
Address translation

220



Two level hierarchical page table

- A process can occupy huge amount of virtual memory
 - In VAX each process can be upto $2^{31} = 2$ Gbytes of virtual memory
- If $2^9=512$ byte size page is used then 2^{22} page table entries!
- Page tables are subject to paging..
 - 32 bit address
 - 2^{20} pages of size 4-Kbyte (2^{12})
 - 2^{20} entries in the page table
 - 4-byte/ entry means the size of the page table is 4 Mbyte (2^{22})
 - The page table itself needs 2^{10} pages (4-Kbte page size)



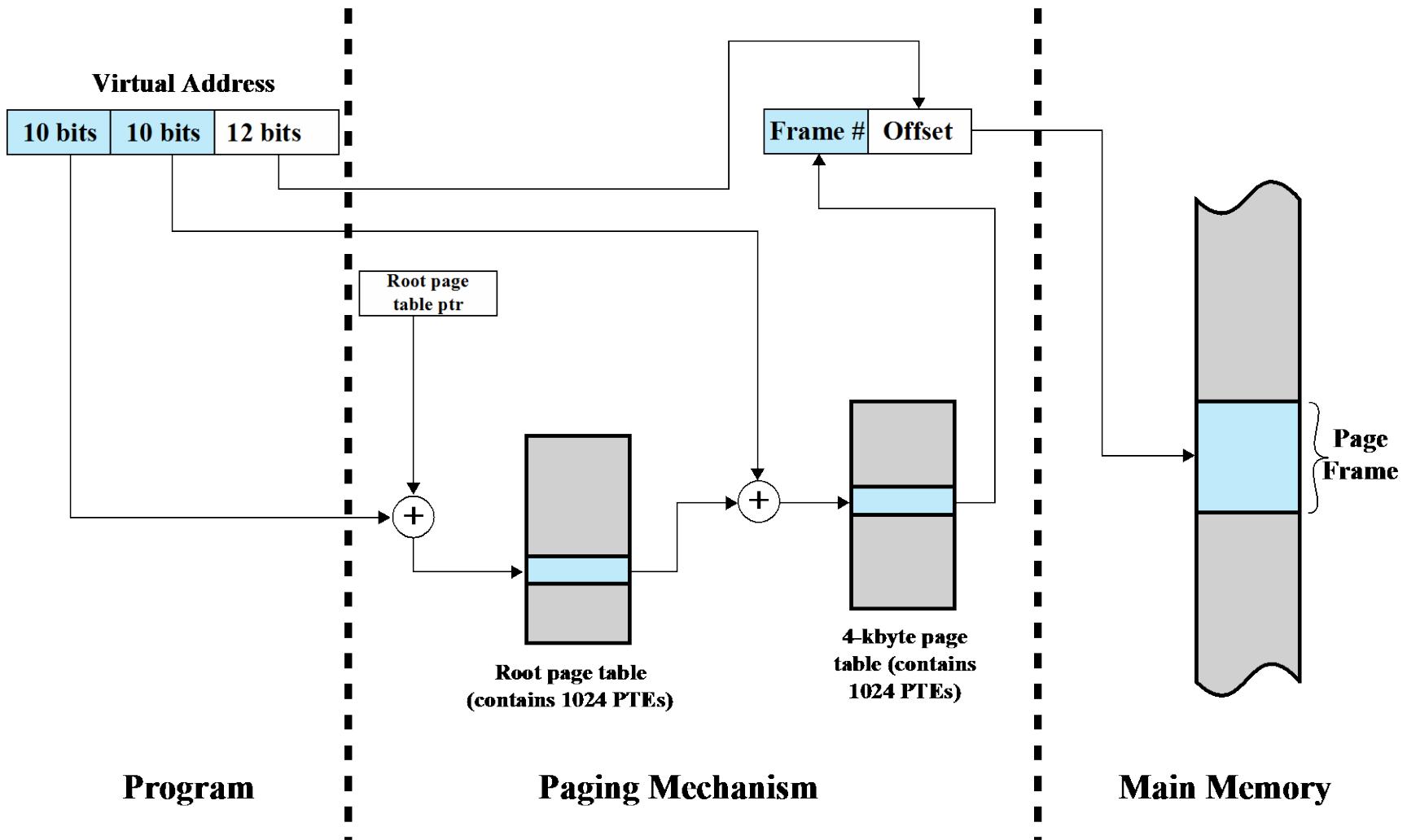
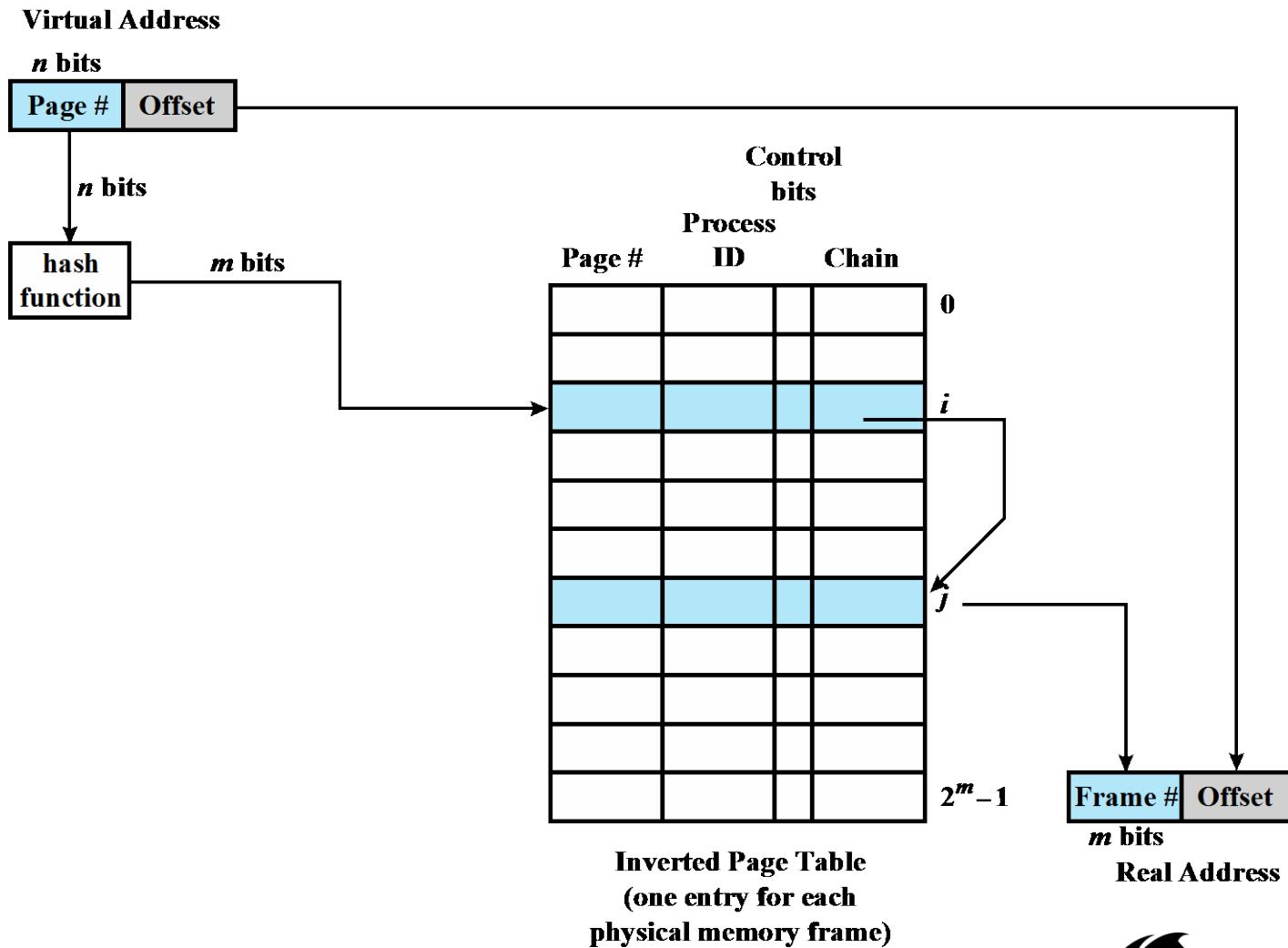


Figure 8.4 Address Translation in a Two-Level Paging System

Inverted page table



Translation lookaside buffer (TLB)

224

- Each virtual memory reference can cause two physical memory accesses:
 - One to fetch the page table entry
 - One to fetch the data
- To overcome the effect of doubling the memory access time, most virtual memory schemes make use of a special high-speed cache called a *translation lookaside* buffer for page table entries
 - This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used.

Use of a TLB

225

- Given a virtual address, the processor will first examine the TLB.
- If the desired page table entry is present (**TLB hit**), then the frame number is retrieved and the real address is formed.
- If the desired page table entry is not found (**TLB miss**), then the processor uses the page number to index the process page table and examine the corresponding page table entry.
 - If the “present bit” is set, then the page is in main memory, and the processor can retrieve the frame number from the page table entry to form the real address.
 - The processor also updates the TLB to include this new page table entry.
 - If the “present” bit is not set, then the desired page is not in main memory and a memory access fault, called a **page fault**, is issued. At this point, we leave the realm of hardware and invoke the operating system, which loads the needed page and updates the page table.

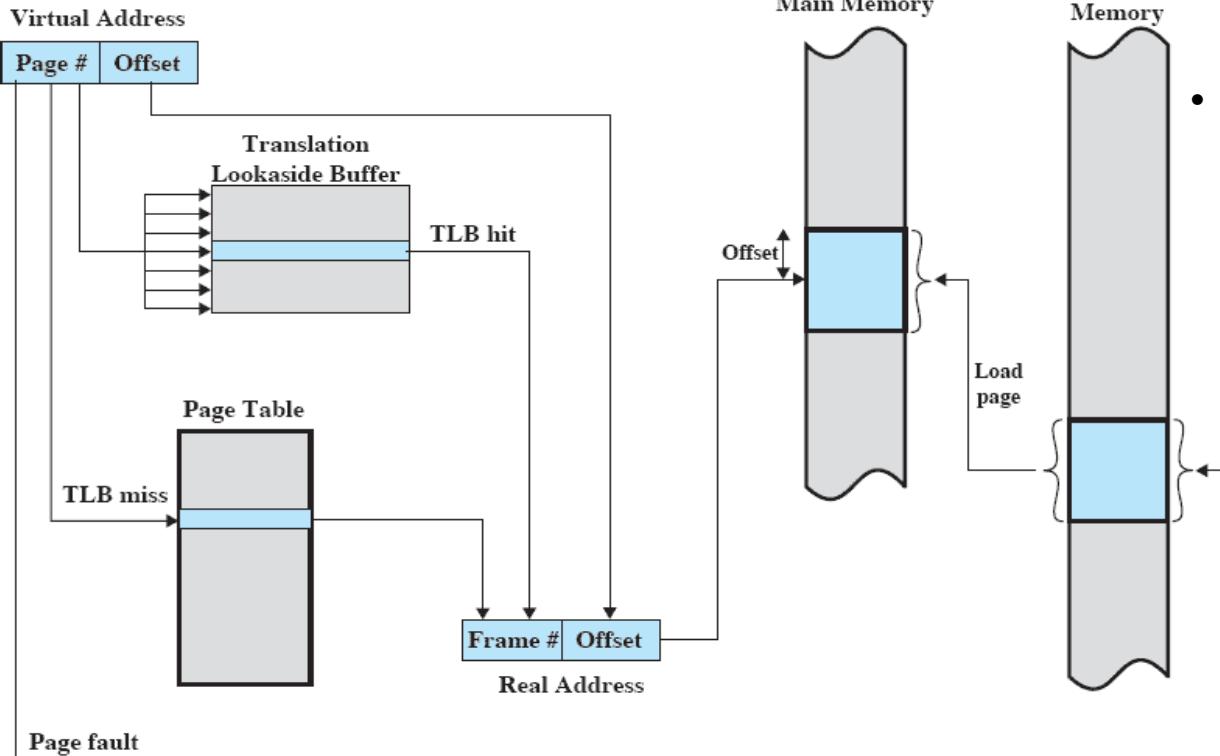
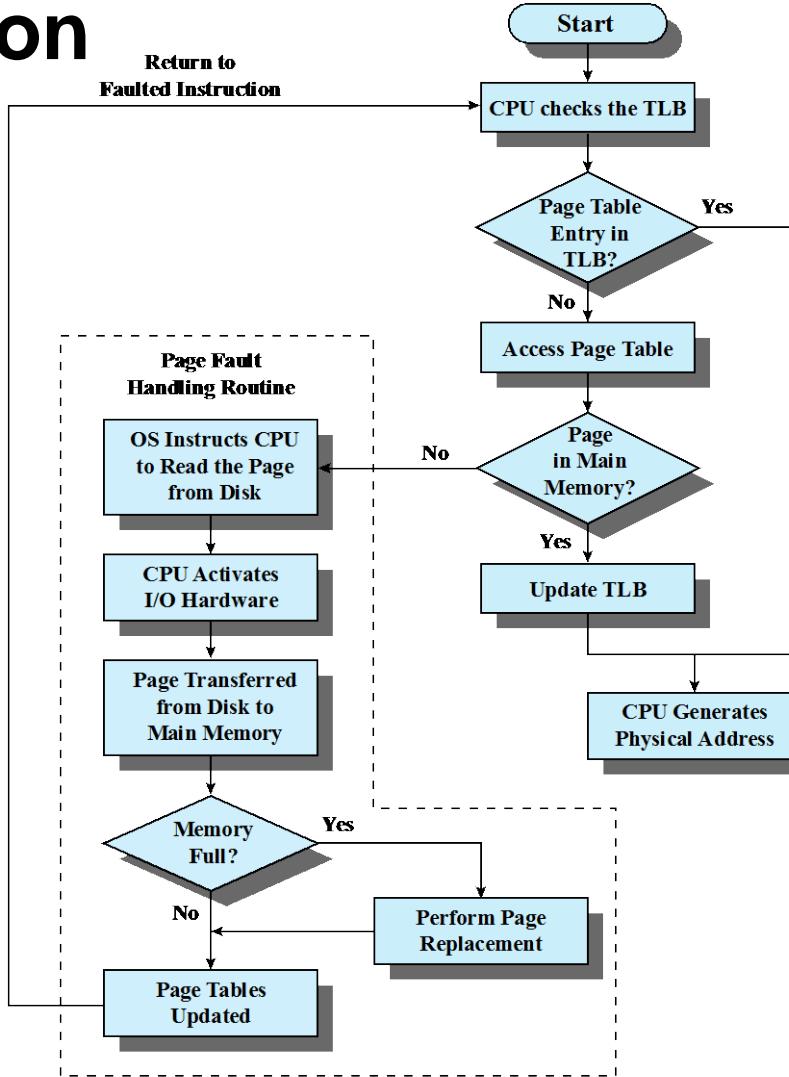


Figure 8.7 Use of a Translation Lookaside Buffer

TLB operation



Direct vs. associative lookup

Virtual Address

Page # Offset

5	502
---	-----

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

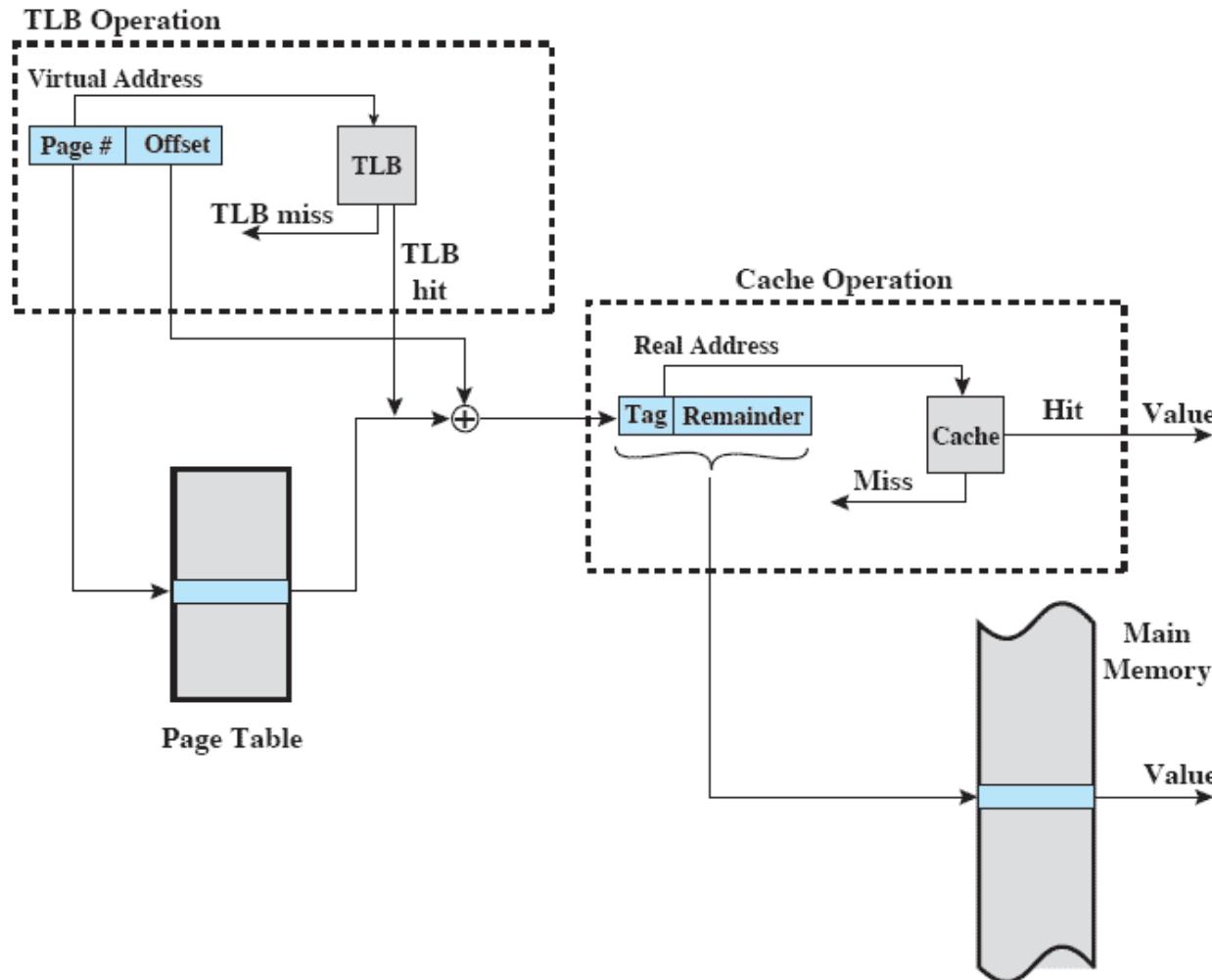
</div

TLB and memory cache

228

- The virtual memory mechanism must interact with the cache system
 - i.e. Main memory cache
- A virtual address will generally be in the form of a page number, offset.
- First, the memory system consults the TLB to see if the matching page table entry is present.
 - If it is, the real (physical) address is generated by combining the frame number with the offset.
 - If not, the entry is accessed from a page table.
- Once the real address is generated, which is in the form of a tag 2 and a remainder, the cache is consulted to see if the block containing that word is present. If so, it is returned to the CPU. If not, the word is retrieved from main memory.

TLB and cache



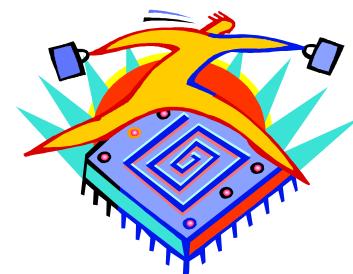
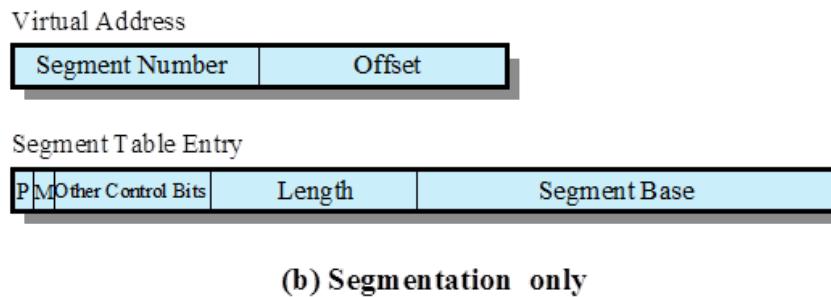
Segmentation

230

- Segmentation differs from paging in two aspects:
 1. The segments are of **variable length**, pages are fixed size.
 2. Segmentation may be under **programmer control**, paging is invisible to the programmer.
- These two differences lead to some **advantages** of segmented over non-segmented memory are:
 - Data structures of **unpredictable size** can be handled.
 - **Program structure** can be organised around segments, e.g., simplifying separate compilation.
 - Segments can be **shared** among processes.
 - Programs can be organised into segments of the **same protection level**. This is more powerful, and easier to control than with paging.
- Structures to control segmentation are very similar to those controlling paging:
- A **segment table** is like a page table with an extra field for the size of the segment

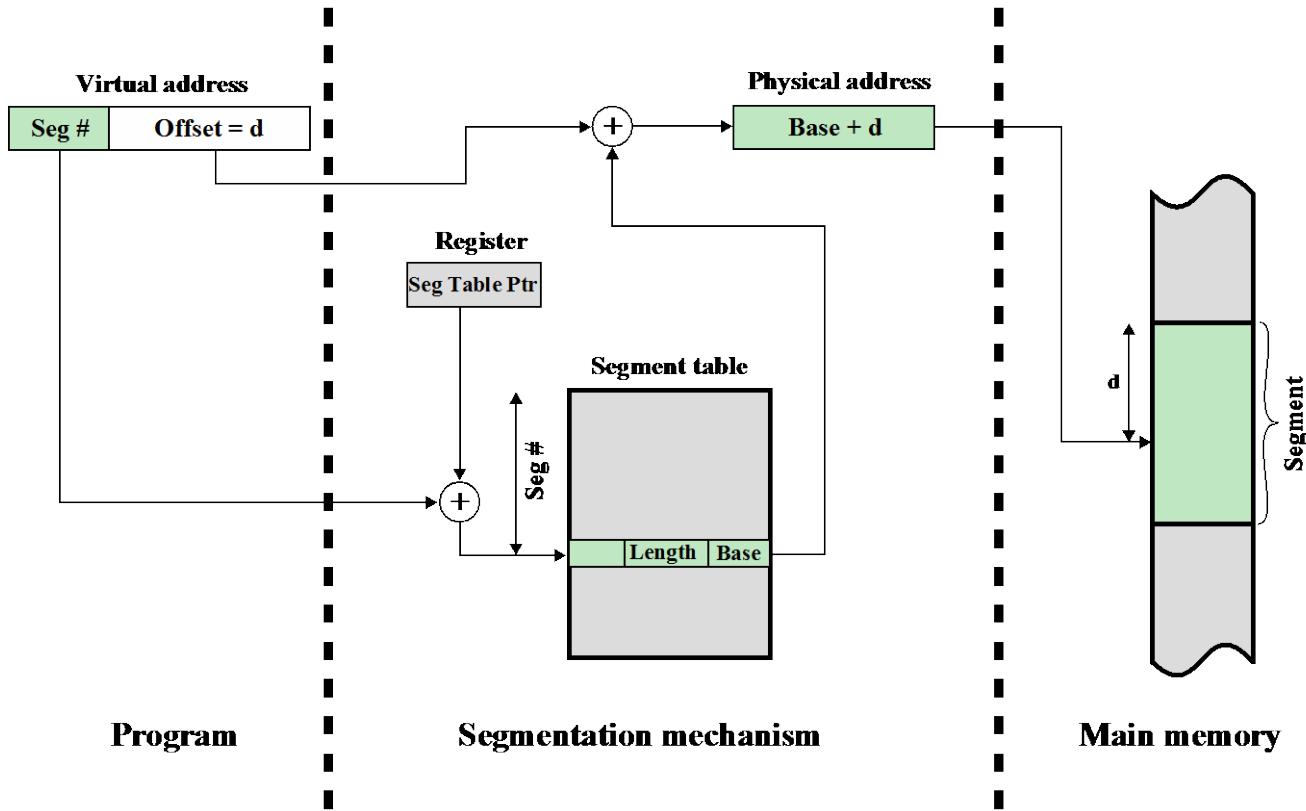
Segment organisation

- Each segment table entry contains the starting address of the corresponding segment in main memory and the length of the segment
- A bit is needed to determine if the 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



Address translation

232



The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of segment number and offset, into a physical address, using a segment table.

Combining paging and segmentation

In a combined paging/segmentation system a user's address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame

Segmentation is visible to the programmer

Paging is transparent to the programmer

Virtual Address

Segment Number	Page Number	Offset
----------------	-------------	--------

Segment Table Entry

Control Bits	Length	Segment Base
--------------	--------	--------------

Page Table Entry

P M Other Control Bits	Frame Number
------------------------------	--------------

P= present bit

M = Modified bit



THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Address translation

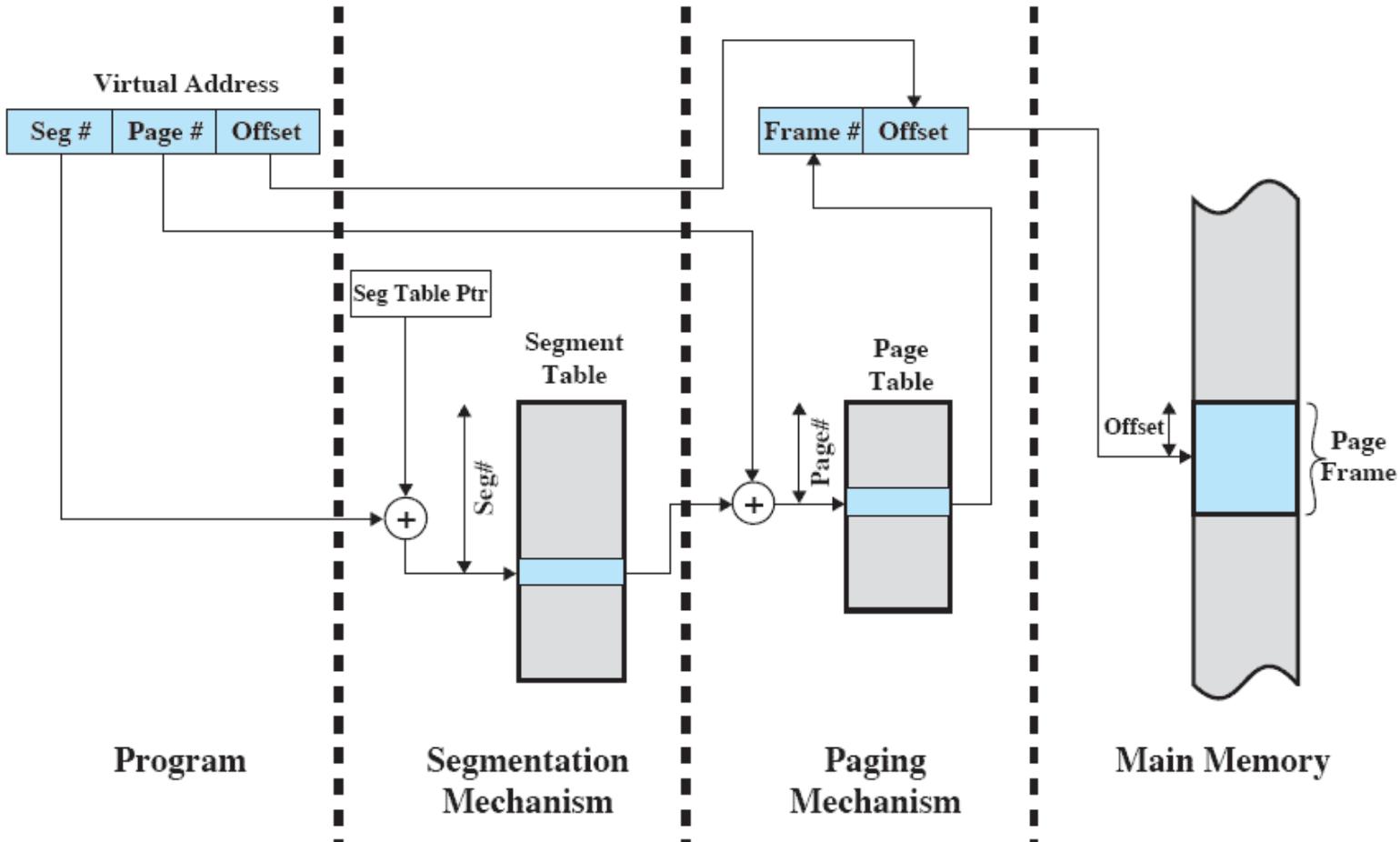


Figure 8.13 Address Translation in a Segmentation/Paging System

Combining segmentation and paging

235

Virtual Address

Segment Number	Page Number	Offset
----------------	-------------	--------

Segment Table Entry

Control Bits	Length	Segment Base
--------------	--------	--------------

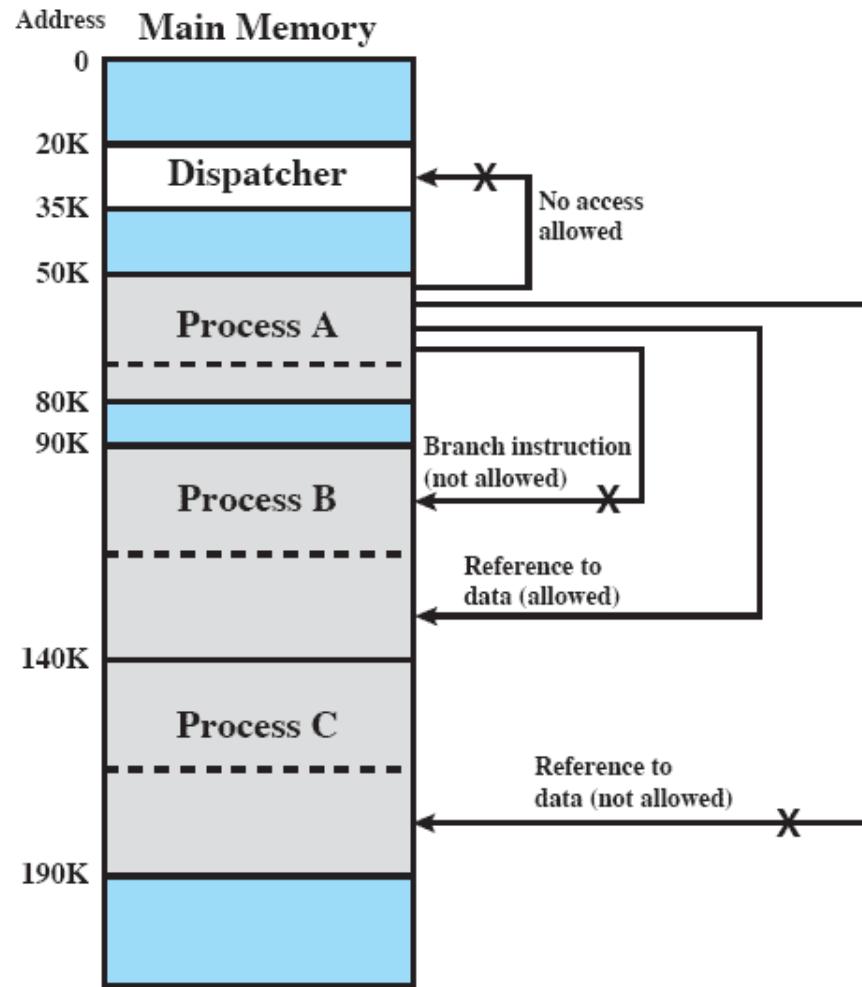
Page Table Entry

P	M	Other Control Bits	Frame Number
---	---	--------------------	--------------

P= present bit
M = Modified bit

(c) Combined segmentation and paging

Protection relationships



OS Policies for virtual memory

- Key issue is performance
 - Minimize page faults

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

- A **good OS** should provide tools for **monitoring** and **tuning** such parameters.
- NOTE: we will mainly discuss paging rather than segmentation.

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

Fetch policy

- The **fetch policy** determines when a page should be brought into main memory.
 - **Demand paging** brings in a page only when a memory reference to that page is made.
 - This policy can cause a large number of page faults before a process "settles down" to a "steady state".
- **Pre-paging** brings in pages *likely to be referenced*, such as those adjacent to a referenced page.
 - When pages are stored contiguously on disk, the overhead in bringing in extra pages is small.
 - Pre-paging can be triggered by page faults.
- For most purposes, **demand paging** seems to be best.

Placement policy

- **Placement policy** determines **where** a new page/segment is to be placed.
 - For paging systems, the location of a page does not effect performance in any way.
 - For a segmented memory, the issues are the same as discussed under "simple segmentation"
 - **best-fit:** choose the block with minimum wasted space.
 - **first-fit:** choose the first block (from the top of memory) into which the process will fit.
 - **next-fit:** choose the first block (from last point) into which the process will fit

Replacement policy

- Deals with the selection of a page in main memory to be replaced when a new page must be brought in
 - Which one to replace?
 - Objective is that the page that is removed be the page least likely to be referenced in the near future
- Because of the principle of locality often there is a high correlation between recent referencing history and near-future referencing patterns.
- The more elaborate the replacement policy the greater the hardware and software overhead to implement it

Frame locking

- When a frame is locked the page currently stored in that frame may not be replaced
 - Kernel of the OS as well as key control structures are held in locked frames
 - I/O buffers and time-critical areas may be locked into main memory frames
 - Locking is achieved by associating a lock bit with each frame



Replacement policies

242

- **Optimal**
 - Choose the page for which the next reference is furthest in the future.
 - it can be shown that this generates the least number of page faults.
 - it is not practical because of the overhead involved in the lookahead.
 - it is a standard by which other methods can be judged: how close is your method to optimal?
- **Least Recently Used (LRU)**
 - Choose the page which has not been referenced for the longest time.
 - The locality principle suggests that this is a good policy.
 - It can be proven to be almost as good as optimal.
 - There are many implementation difficulties: we have to tag each page with the time of the last reference.

Replacement policies

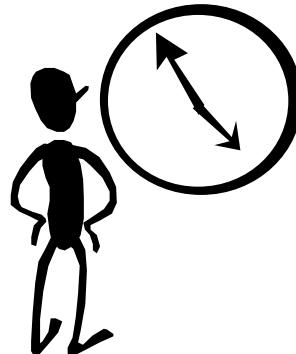
243

- **First In First Out (FIFO)**
 - simple to implement: one pointer per page, arranged in a FIFO queue.
 - removes the page which has been in memory the longest.
 - does not recognise that some pages are referenced more frequently than others.
 - performs badly.

Replacement Policies

244

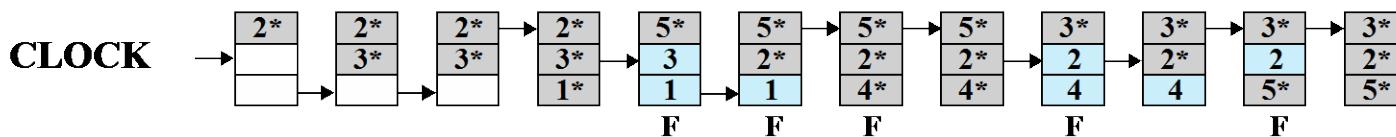
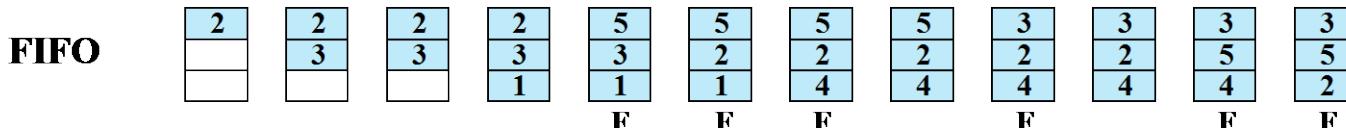
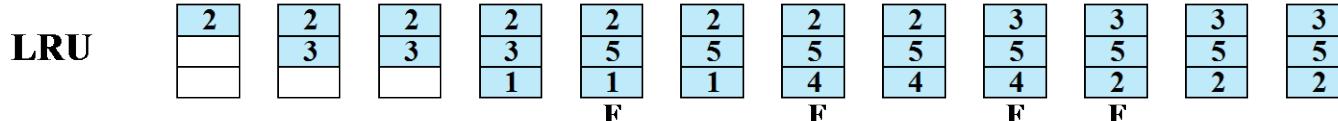
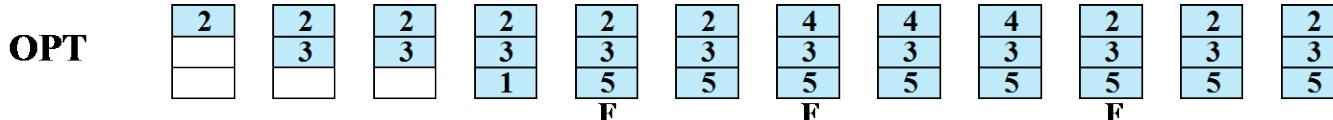
- **Clock policy**
 - Requires the association of an additional bit with each frame
 - referred to as the use bit
 - When a page is first loaded in memory or referenced, the use bit is set to 1
 - The set of frames is considered to be a circular buffer
 - Any frame with a use bit of 1 is passed over by the algorithm
 - Page frames visualized as laid out in a circle



Combined examples

Page address
stream

2 3 2 1 5 2 4 5 3 2 5 2



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

Resident set management

246

- 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 can reside in memory
 - Small number of pages loaded increases page faults
 - Beyond a certain size, further allocations of pages will not effect the page fault rate
 - Principle of locality

Resident set size

■ Fixed-allocation

- gives a process a fixed number of frames in main memory within which to execute
 - Based on process type and guidance from programmer/system admin
- when a page fault occurs, one of the pages of that process must be replaced

■ Variable-allocation

- allows the number of page frames allocated to a process to be varied over the lifetime of the process
 - Process with high page fault rate, more page is allocated
 - Process with exceptionally low page fault rate allocation is reduced
 - More powerful scheme but incurs more overhead

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
- **Local**
 - chooses only among the resident pages of the process that generated the page fault
- **Global**
 - considers all unlocked pages in main memory
- No convincing evidence that local policies perform better than global policies

Resident set management

	Local Replacement	Global Replacement
Fixed Allocation	<ul style="list-style-type: none"> • Number of frames allocated to a process is fixed. • Page to be replaced is chosen from among the frames allocated to that process. 	<ul style="list-style-type: none"> • Not possible.
Variable Allocation	<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • 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

- Concerned with determining when a modified page should be written out to secondary memory

Demand Cleaning

a page is written out to secondary memory only when it has been selected for replacement



Precleaning

allows the writing of pages in batches

Process suspension

- If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be swapped out

Six possibilities exist:

- Lowest-priority process
- Faulting process
- Last process activated
- Process with the smallest resident set
- Largest process
- Process with the largest remaining execution window

Disk and I/O Scheduling

WEEK 9

- The messiest aspect of OS design
- Wide variety of I/O devices
 - Wide variation in performance
- Difficult to provide a general, consistent solution

Difference in I/O devices

- Devices differ in a number of areas:

Data Rate

- there may be differences of magnitude between the data transfer rates

Application

- the use to which a device is put has an influence on the software

Complexity of Control

- the effect on the operating system is filtered by the complexity of the I/O module that controls the device

Unit of Transfer

- data may be transferred as a stream of bytes or characters or in larger blocks

Data Representation

- different data encoding schemes are used by different devices

Error Conditions

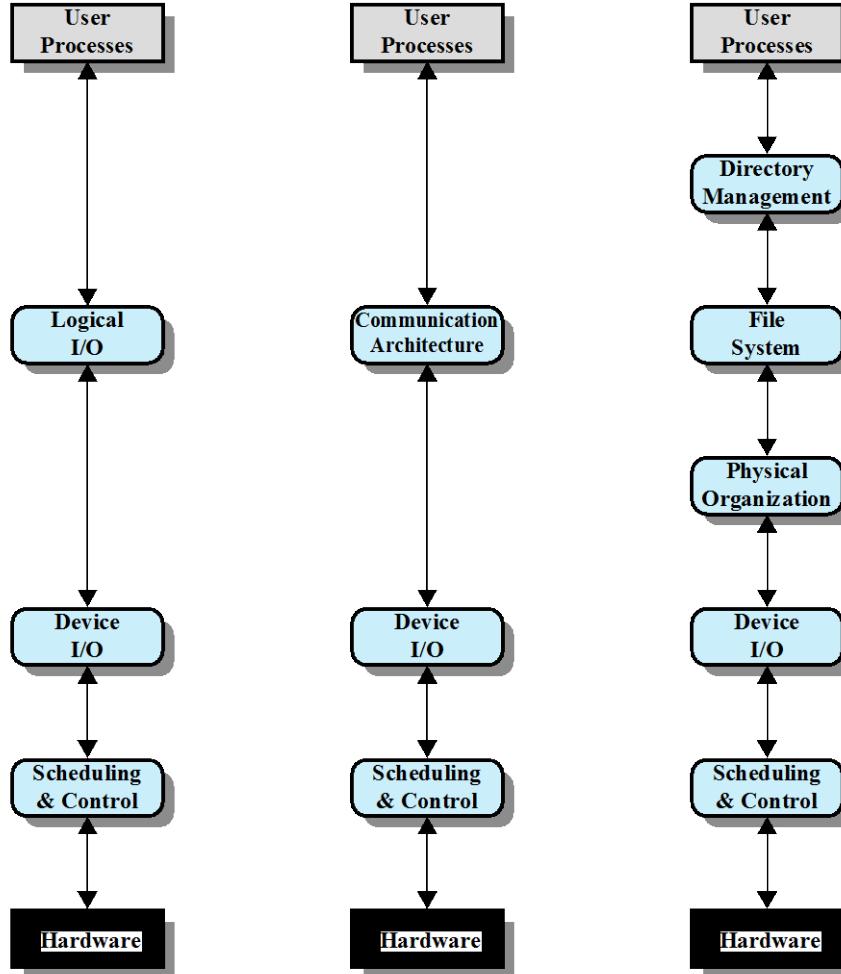
- the nature of errors, the way in which they are reported, their consequences, and the available range of responses differs from one device to another

Hierarchical design

255

- Functions of the operating system should be separated according to their complexity, their characteristic time scale, and their level of abstraction
- Leads to an organization of the operating system into a series of layers
- Each layer performs a related subset of the functions required of the operating system
- Layers should be defined so that changes in one layer do not require changes in other layers

A model of I/O organization



Organisation of the I/O function

- Three techniques for performing I/O are:
- **Programmed I/O**
 - The processor issues an I/O command on behalf of a process to an I/O module; that process then busy waits for the operation to be completed before proceeding
- **Interrupt-driven I/O**
 - the processor issues an I/O command on behalf of a process
 - If non-blocking – processor continues to execute instructions from the process that issued the I/O command
 - If blocking – the next instruction the processor executes is from the OS, which will put the current process in a blocked state and schedule another process
- **Direct Memory Access (DMA)**
 - A DMA module controls the exchange of data between main memory and an I/O module
 - The processor sends a request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

Techniques for performing I/O

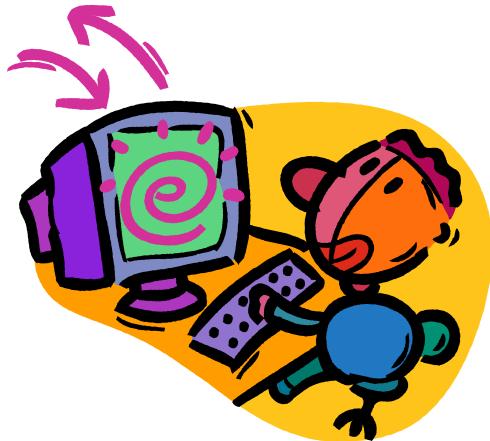


Table 11.1 I/O Techniques

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

- In most computer systems, DMA is the dominant form of transfer that must be supported by the operating system.

Direct memory access

- The DMA unit is capable of mimicking the processor and of taking over control of the system bus just like a processor.
 - It needs to do this to transfer data to/from memory over the system bus.
- When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending to the DMA module the following information:
 - Whether a read or write is requested, using the read or write control line between the processor and the DMA module
 - The address of the I/O device involved, communicated on the data lines
 - The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register
 - The number of words to be read or written, again communicated via the data lines and stored in the data count register
- The processor then continues with other work.
 - It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor.
 - When the transfer is complete, the DMA module sends an interrupt signal to the processor.
- The processor is involved only at the beginning and end of the transfer

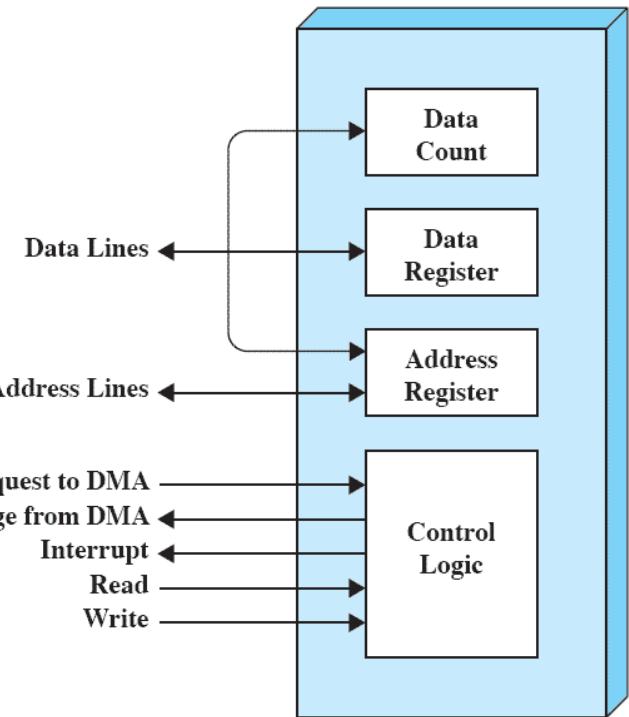
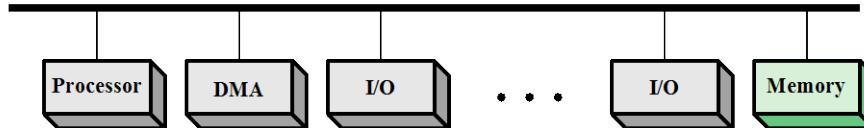
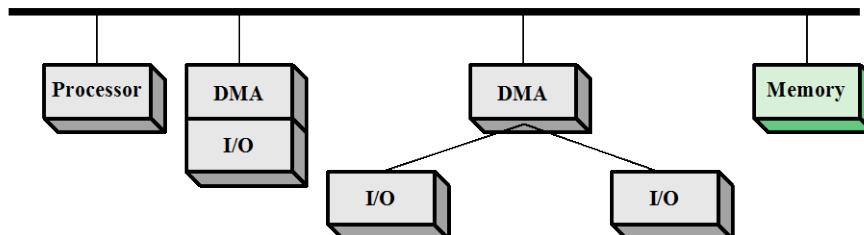


Figure 11.2 Typical DMA Block Diagram

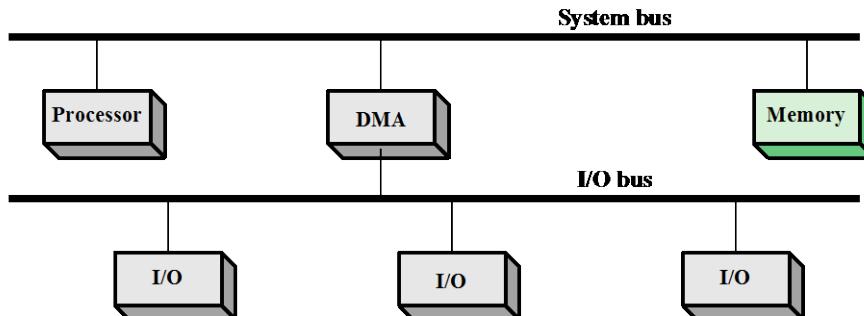
DMA Configurations



(a) Single-bus, detached DMA



(b) Single-bus, Integrated DMA-I/O



(c) I/O bus

I/O - Buffering

- Consider an I/O command to transfer 512 bytes from disk to virtual address 1000 to 1511 of the process
Read_Block[1000, disk]
- I/O and process swapping may interact, and possibly cause deadlock, due to the time delayed nature of I/O operations.
- It can be convenient to transfer input data in advance of an Input request, and to transfer output data some time after an Output request is made.
- Use buffering.

Buffering

- Perform input transfers in advance of requests being made and perform output transfers some time after the request is made

Block-oriented device

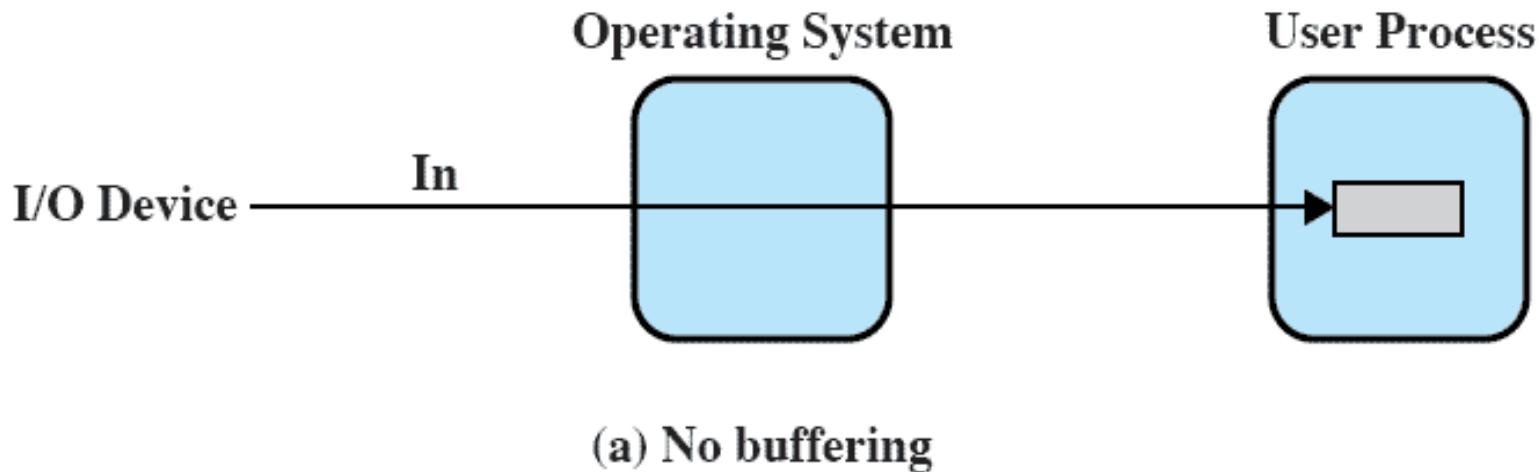
- stores information in blocks that are usually of fixed size
- transfers are made one block at a time
- possible to reference data by its block number
- disks and USB keys are examples

Stream-oriented device

- transfers data in and out as a stream of bytes
- no block structure
- terminals, printers, communications ports, and most other devices that are not secondary storage are examples

No buffer

- Without a buffer, the OS directly accesses the device when it needs



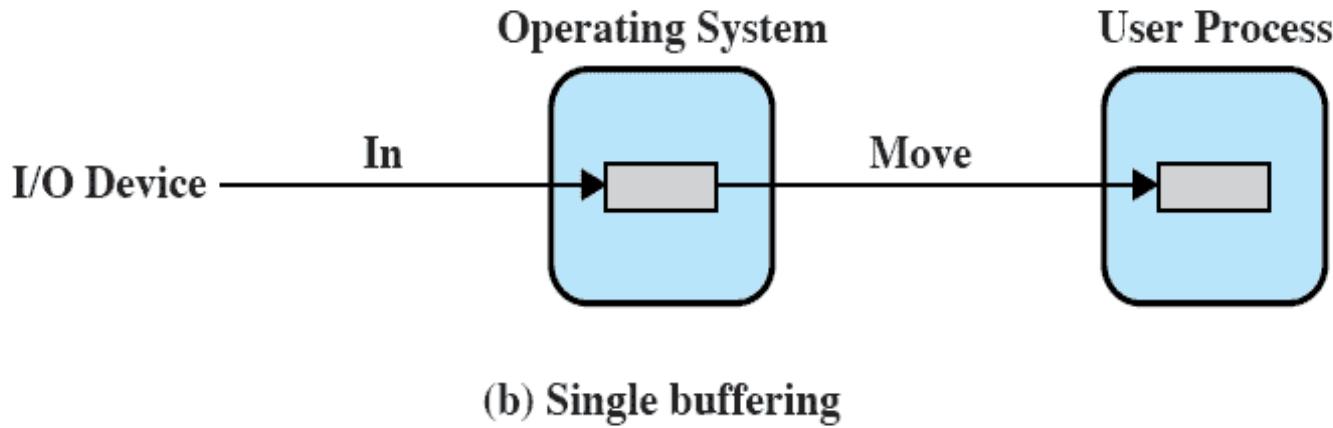
Suppose, **T**: time required to input one block

C: is the computation time between input requests

T+C: execution time per block

Single buffer

- Operating system assigns a buffer in the system portion of main memory for an I/O request



Suppose, **T**: time required to input one block

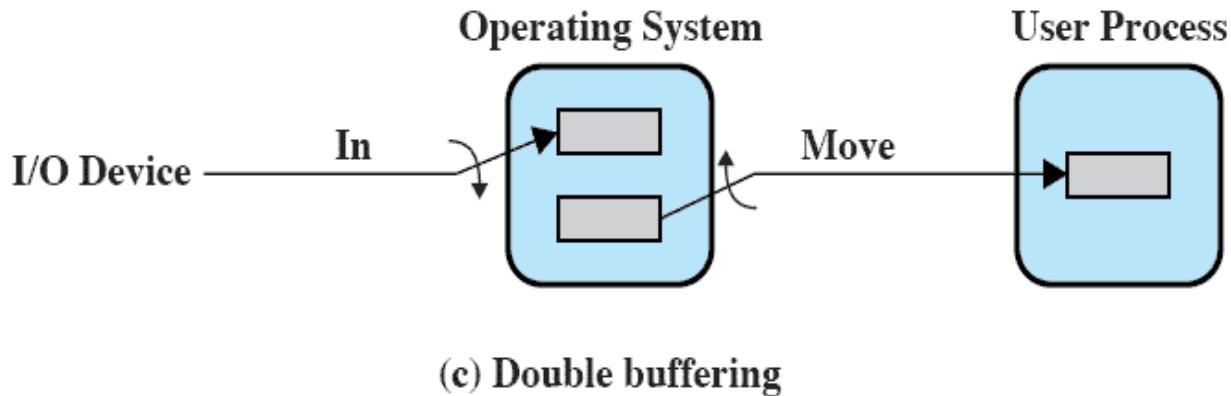
C: is the computation time between input requests

max[T,C] + M: execution time per block

M: time to move data from system buffer to user memory

Double buffering

- Use two system buffers instead of one
- A process can transfer data to or from one buffer while the operating system empties or fills the other buffer
- Also known as buffer swapping



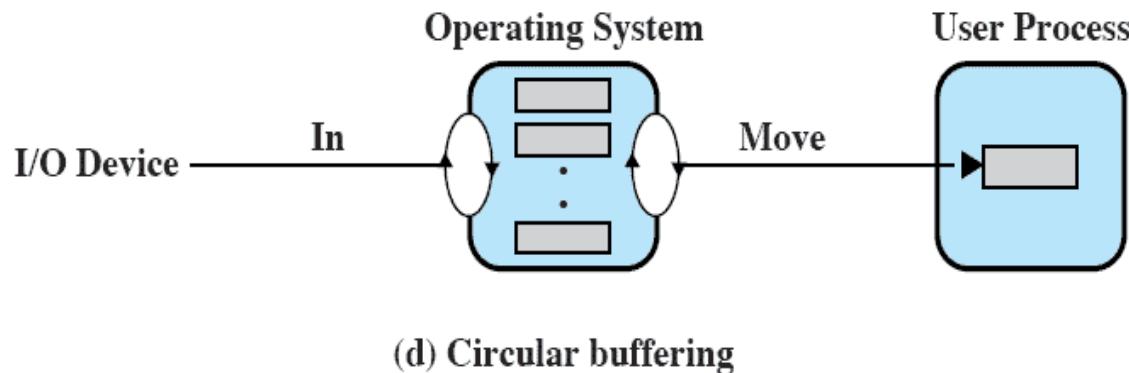
Suppose, **T**: time required to input one block

C: is the computation time between input requests

max[T,C]: execution time per block

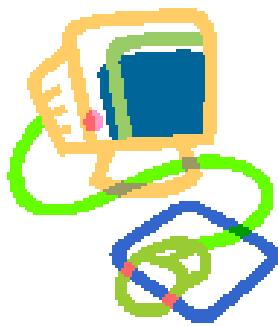
Circular buffer

- Two or more buffers are used
- Each individual buffer is one unit in a circular buffer
- Used when I/O operation must keep up with process
- Bounded-buffer producer/consumer model



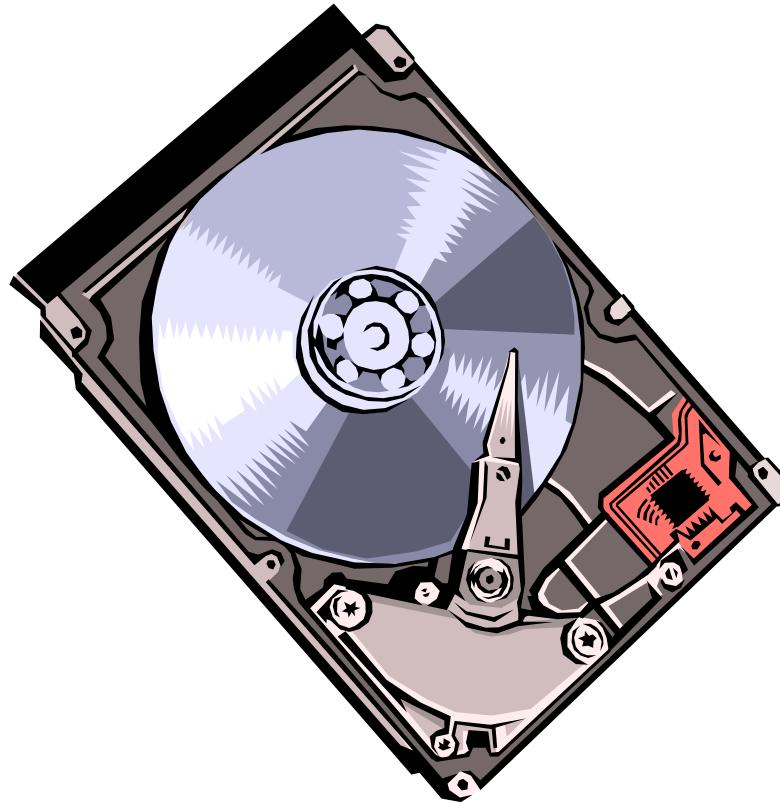
Utility of buffering

- Technique that smooths out peaks in I/O demand
 - with enough demand eventually all buffers become full and their advantage is lost
- When there is a variety of I/O and process activities to service, buffering can increase the efficiency of the OS and the performance of individual processes



Disk Scheduling

268

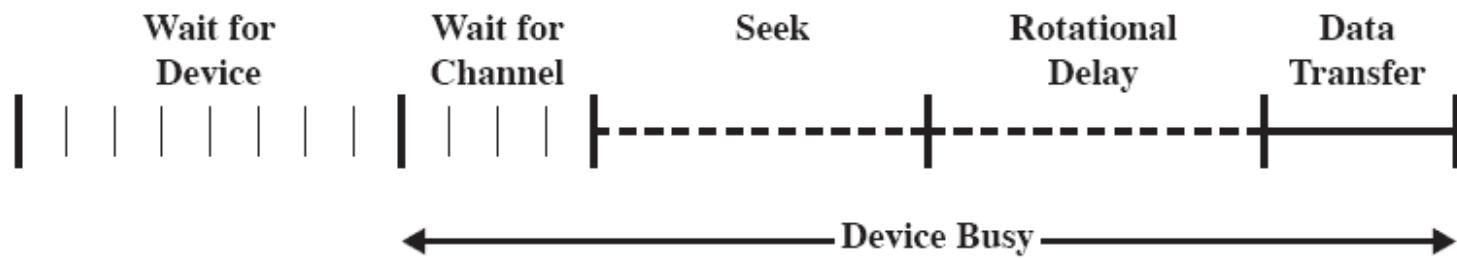


Positioning the Read/Write heads

- When the disk drive is operating, the disk is rotating at constant speed
- To read or write the head must be positioned at the desired track and at the beginning of the desired sector on that track
- Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system
- On a movable-head system the time it takes to position the head at the track is known as **seek time**
- The time it takes for the beginning of the sector to reach the head is known as **rotational delay**
- The sum of the seek time and the rotational delay equals the **access time**

Disk performance parameters

270



Disk scheduling algorithms

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	Average seek length	27.5	Average seek length	27.8	Average seek length	35.8

Disk scheduling algorithms

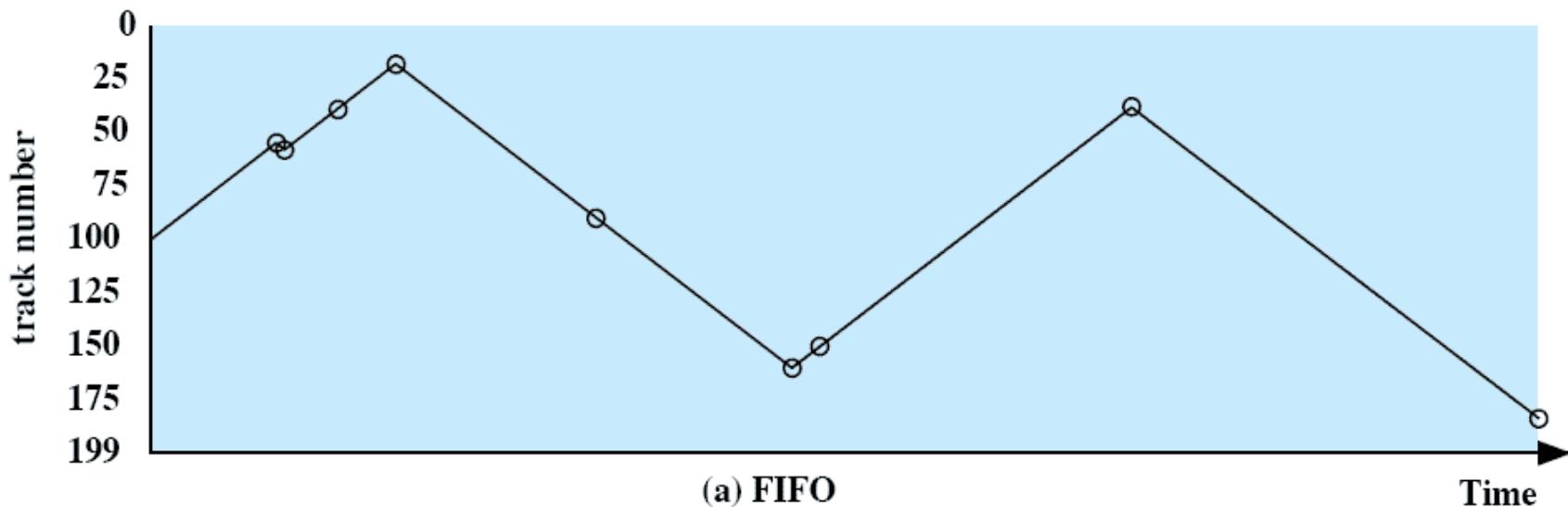
(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	Average seek length	27.5	Average seek length	27.8	Average seek length	35.8

First-In, First-Out (FIFO)

273

- Processes in sequential order
- Fair to all processes
- Approximates random scheduling in performance if there are many processes competing for the disk

55, 58, 39, 18, 90, 160,
150, 38, 184



Priority

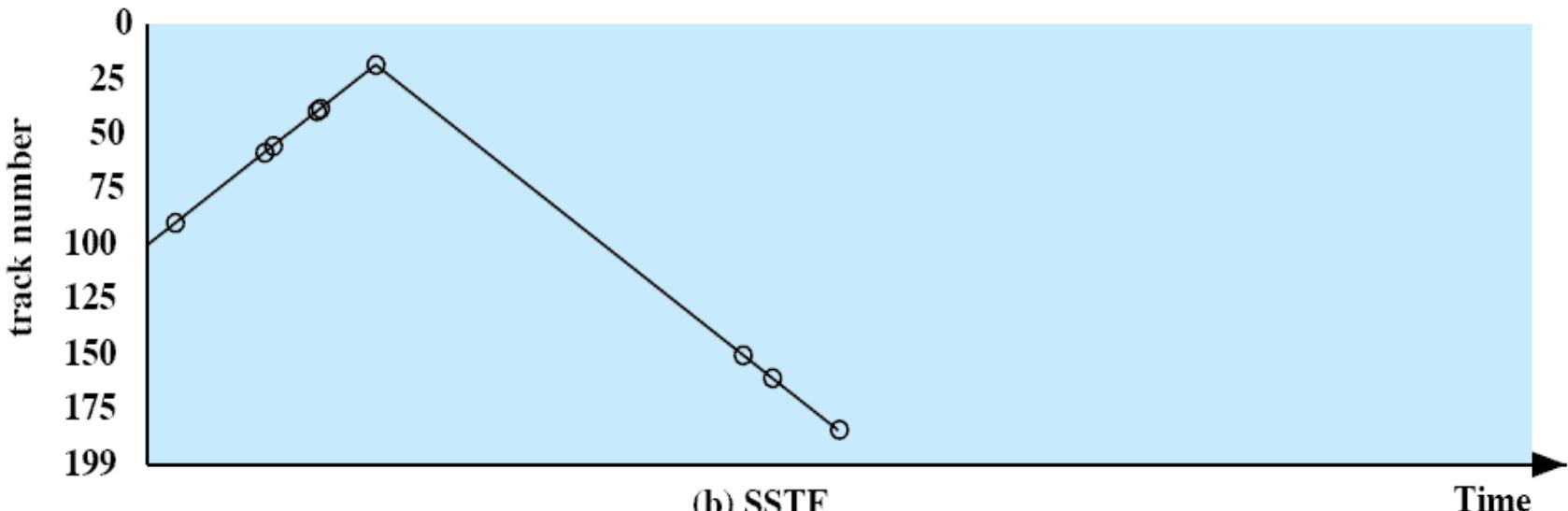
274

- Imposed outside of disk management to meet other objectives within OS
- Not intended to optimise disk use
- Example: interactive (and possibly short batch) jobs are given higher priority than longer batch or high computation jobs.
 - This flushes lots of short jobs through the system quickly -> good interactive response time. But long jobs are slow.

Shortest Service Time First (SSTF)

275

- Select the disk I/O request that requires the least movement of the disk arm from its current position
55, 58, 39, 18, 90, 160,
150, 38, 184
- Always choose the minimum seek time

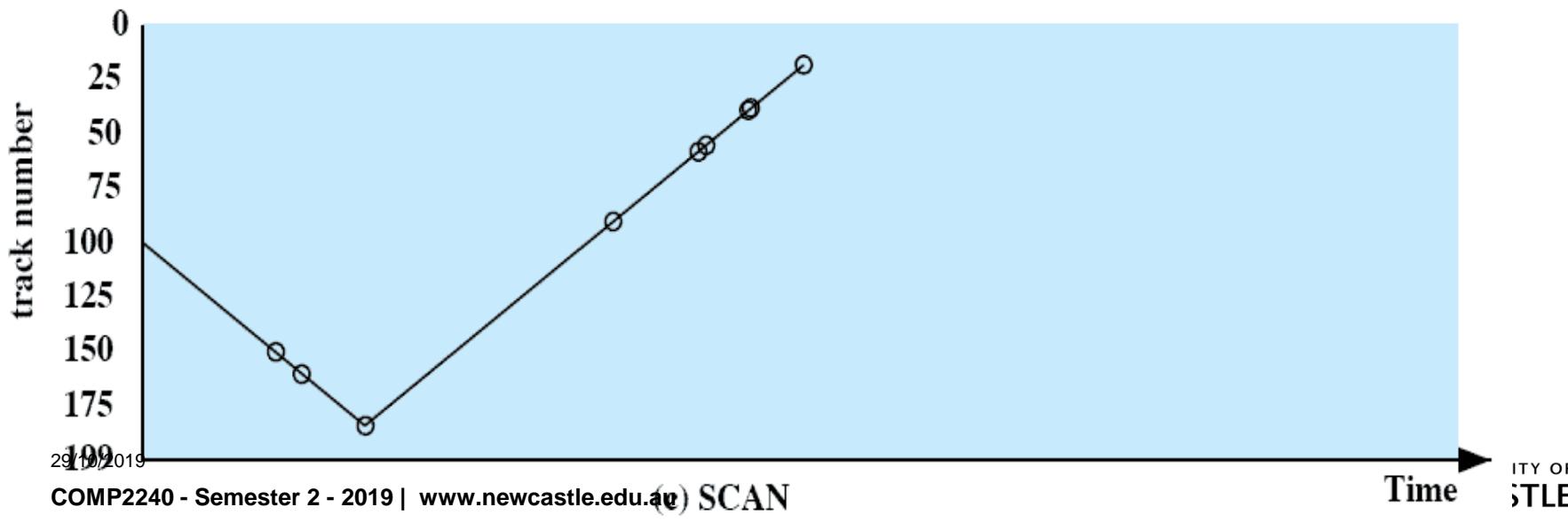


SCAN

276

- Also known as the elevator algorithm
- Arm moves in one direction only
 - Satisfies all outstanding requests until it reaches the last track in that direction then the direction is reversed
- Favours jobs whose requests are for tracks nearest to both innermost and outermost tracks

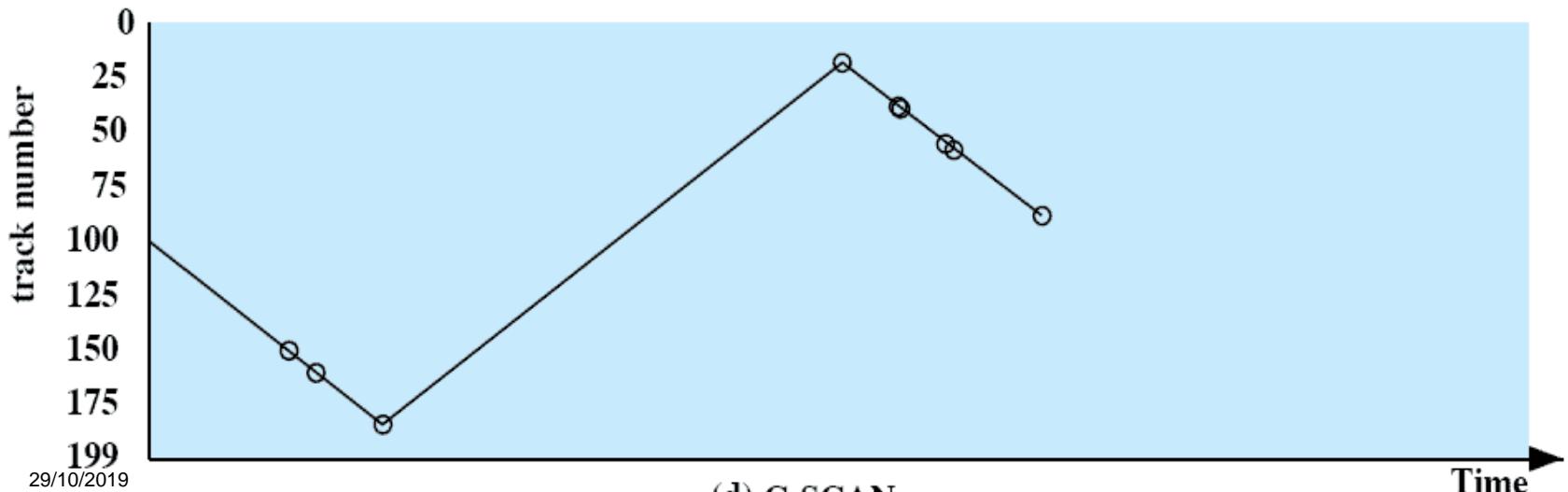
55, 58, 39, 18, 90, 160,
150, 38, 184



C-SCAN

277

- Circular SCAN
- Restricts scanning to one direction only
- When the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again



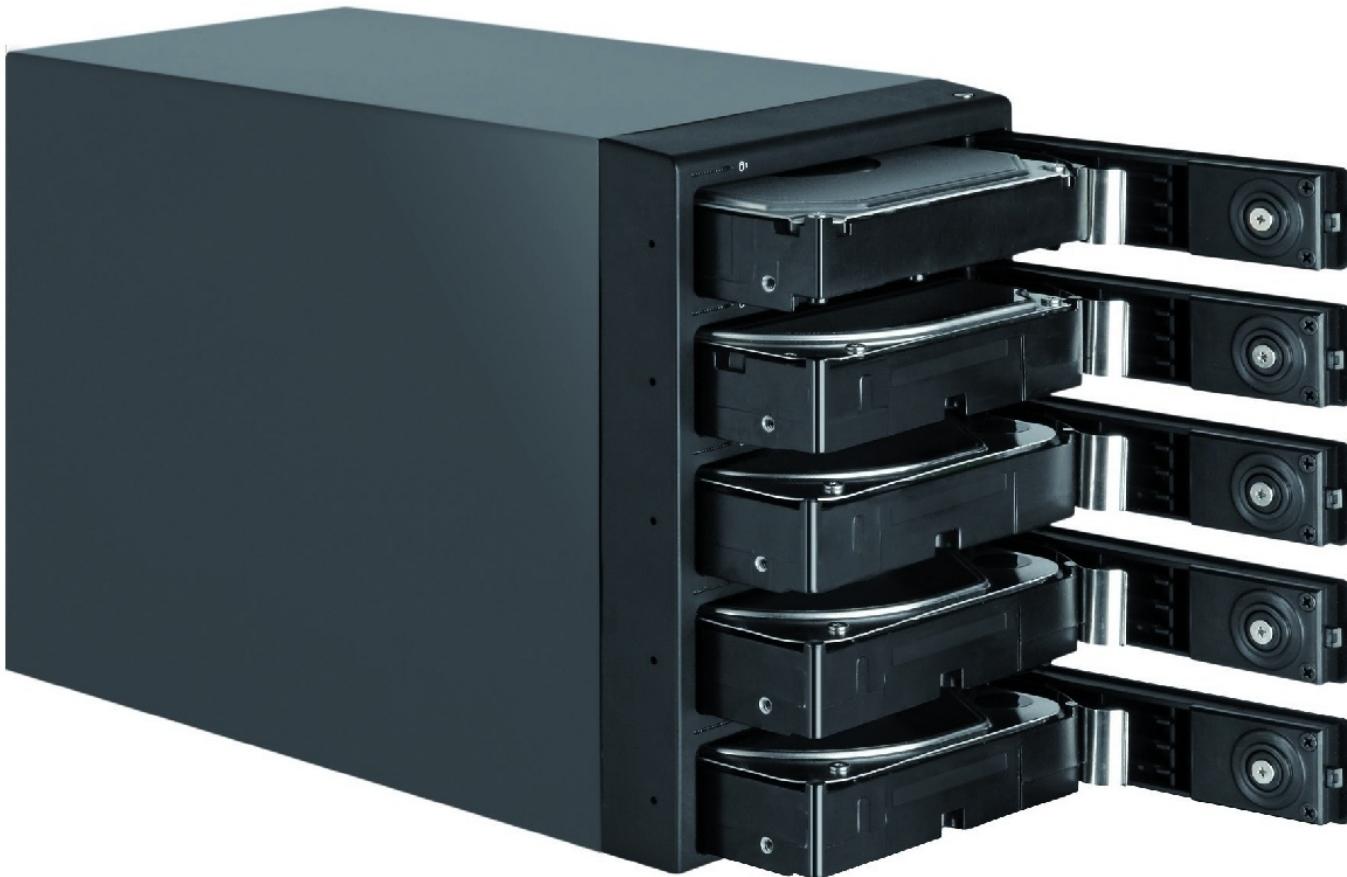
N-Step-SCAN

278

- Segments the disk request queue into sub-queues of length N
- Sub-queues are processed one at a time, using SCAN
- While a queue is being processed new requests must be added to some other queue
- If fewer than N requests are available at the end of a scan, all of them are processed with the next scan
- With large values of N , *the performance of N-Step-SCAN approaches that of SCAN*
 - with a value of $N = 1$, *the FIFO policy is adopted*

RAID

279



29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

RAID Characteristics

280

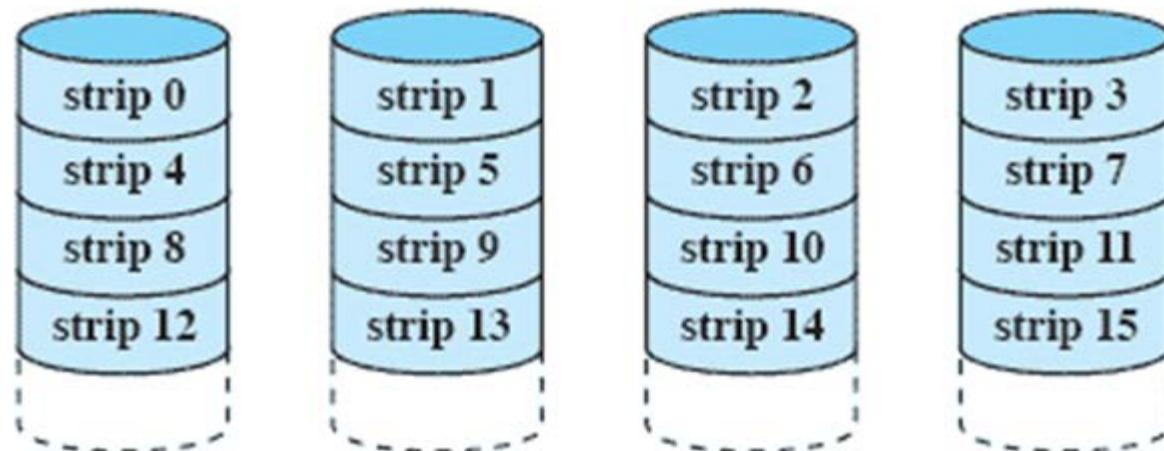
RAID design architectures share three characteristics:

1. RAID is a set of physical disk drives viewed by the operating system as a single logical drive
2. Data are distributed across the physical drives of an array in a scheme known as striping
3. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure
 - Not supported by RAID 0 and RAID 1

RAID Level 0

281

- Not a true RAID because it does not include redundancy to improve performance or provide data protection
- User and system data are distributed across all of the disks in the array
- Logical disk is divided into strips

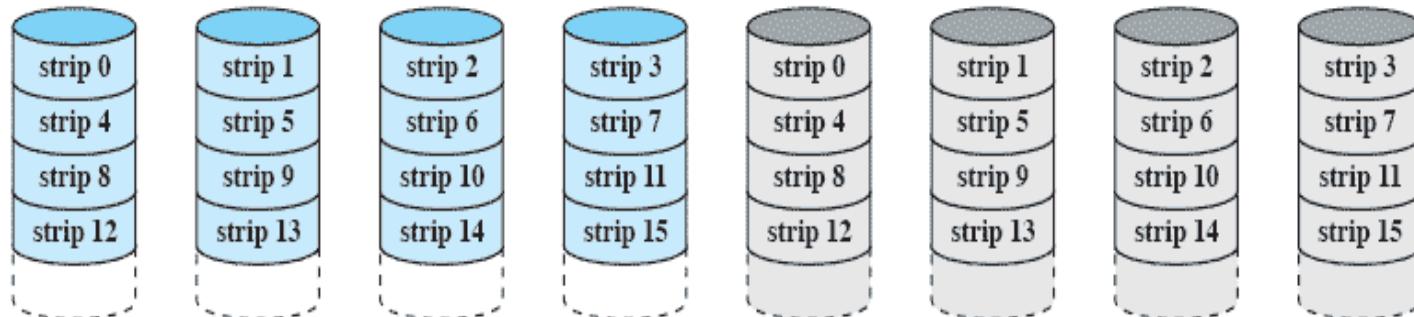


(a) RAID 0 (non-redundant)

RAID Level 1

282

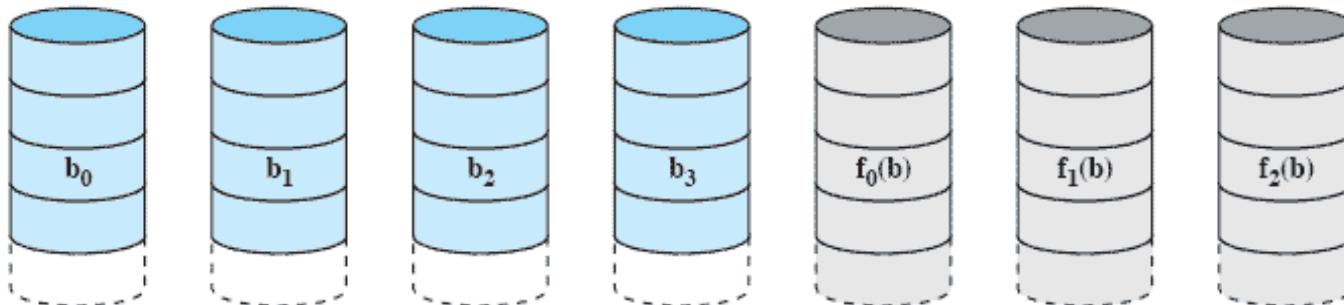
- Redundancy is achieved by the simple expedient of duplicating all the data
- There is no “write penalty”
- When a drive fails the data may still be accessed from the second drive
- Principal disadvantage is the cost



(b) RAID 1 (mirrored)

RAID Level 2

- Makes use of a parallel access technique
 - All member disks participate in the execution of every I/O request
- Data striping is used
 - Strips are very small
- Error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in the corresponding bit positions on multiple parity disks
 - Typically a Hamming code is used - correct single-bit errors and detect double-bit errors
- Effective choice in an environment in which many disk errors occur



(c) RAID 2 (redundancy through Hamming code)

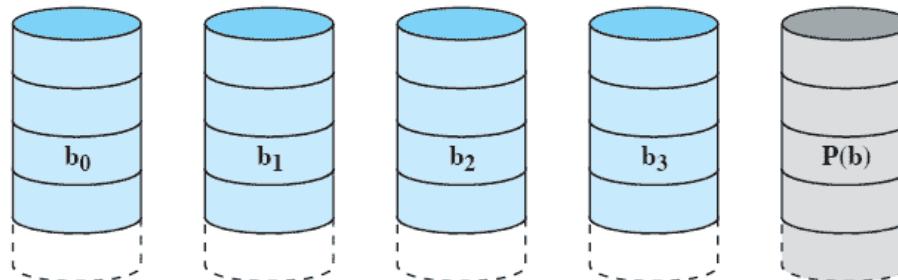
29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

RAID Level 3

284

- Requires only a single redundant disk, no matter how large the disk array
 - a simple parity bit is computed for the set of individual bits in the same position on all of the data disks
- Employs parallel access
 - With data distributed in small strips
- Can achieve very high data transfer rates

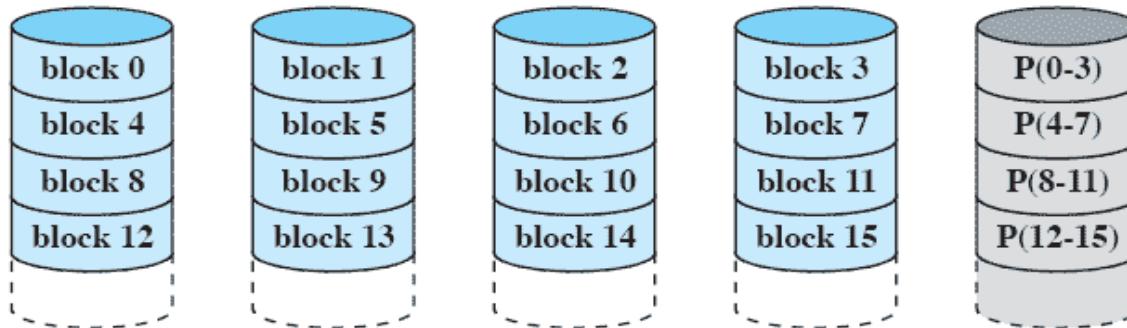


(d) RAID 3 (bit-interleaved parity)

RAID Level 4

285

- Makes use of an independent access technique
 - Each member disk operates independently, so that separate I/O requests can be satisfied in parallel
- A bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk
- Involves a write penalty when an I/O write request of small size is performed

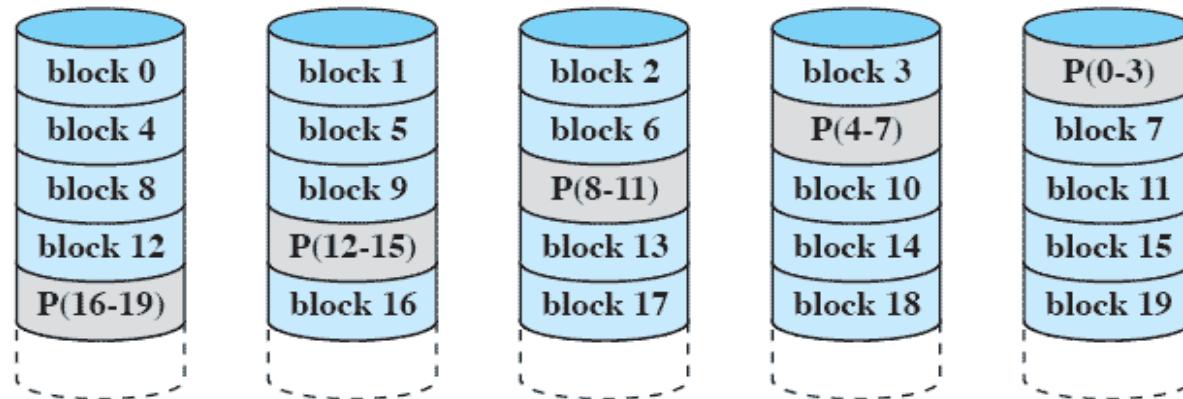


(e) RAID 4 (block-level parity)

RAID Level 5

286

- Similar to RAID-4 but distributes the parity bits across all disks
- Typical allocation is a round-robin scheme
- Has the characteristic that the loss of any one disk does not result in data loss

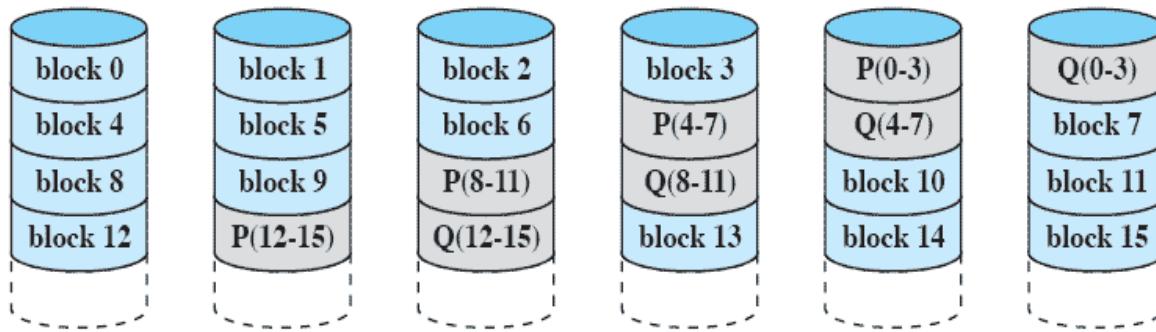


(f) RAID 5 (block-level distributed parity)

RAID Level 6

287

- Two different parity calculations are carried out and stored in separate blocks on different disks
- Provides extremely high data availability
- Incurs a substantial write penalty because each write affects two parity blocks



(g) RAID 6 (dual redundancy)

RAID Levels

288

Category	Level	Description	Disks required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

N = number of data disks; m proportional to $\log N$

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

File Management

WEEK 10

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

File Management

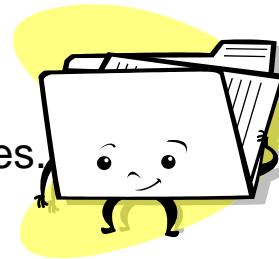
290

- The file is the central element in most applications
 - Input to the application is by means of a file
 - Virtually all applications, output is saved in a file for long term storage, later access by the user and use by other programs.
- Files also have a life outside of any individual application
 - Users wish to be able to access files, save them, and maintain the integrity of their contents.
- Virtually all operating systems provide file management systems.
 - Typically, a file management system consists of system utility programs that run as privileged applications.
 - At the least, a file management system needs special services from the operating system
 - At the most, the entire file management system is considered part of the operating system.

File system

291

- Provide a means to store data organized as files as well as a collection of functions that can be performed on files
- Typical operations include:
 - **Create:** A new file is defined and positioned within the structure of files.
 - **Delete:** A file is removed from the file structure and destroyed.
 - **Open:** An existing file is declared to be “opened” by a process, allowing the process to perform functions on the file.
 - **Close:** The file is closed with respect to a process, so that the process no longer may perform functions on the file, until the process opens the file again.
 - **Read:** A process reads all or a portion of the data in a file.
 - **Write:** A process updates a file, either by adding new data that expands the size of the file or by changing the values of existing data items in the file.
- Also maintains a set of attributes associated with the file
 - Owner, creation time, time last modified, access privileges etc.



File structure

292

Four terms are commonly used when discussing files:

Field

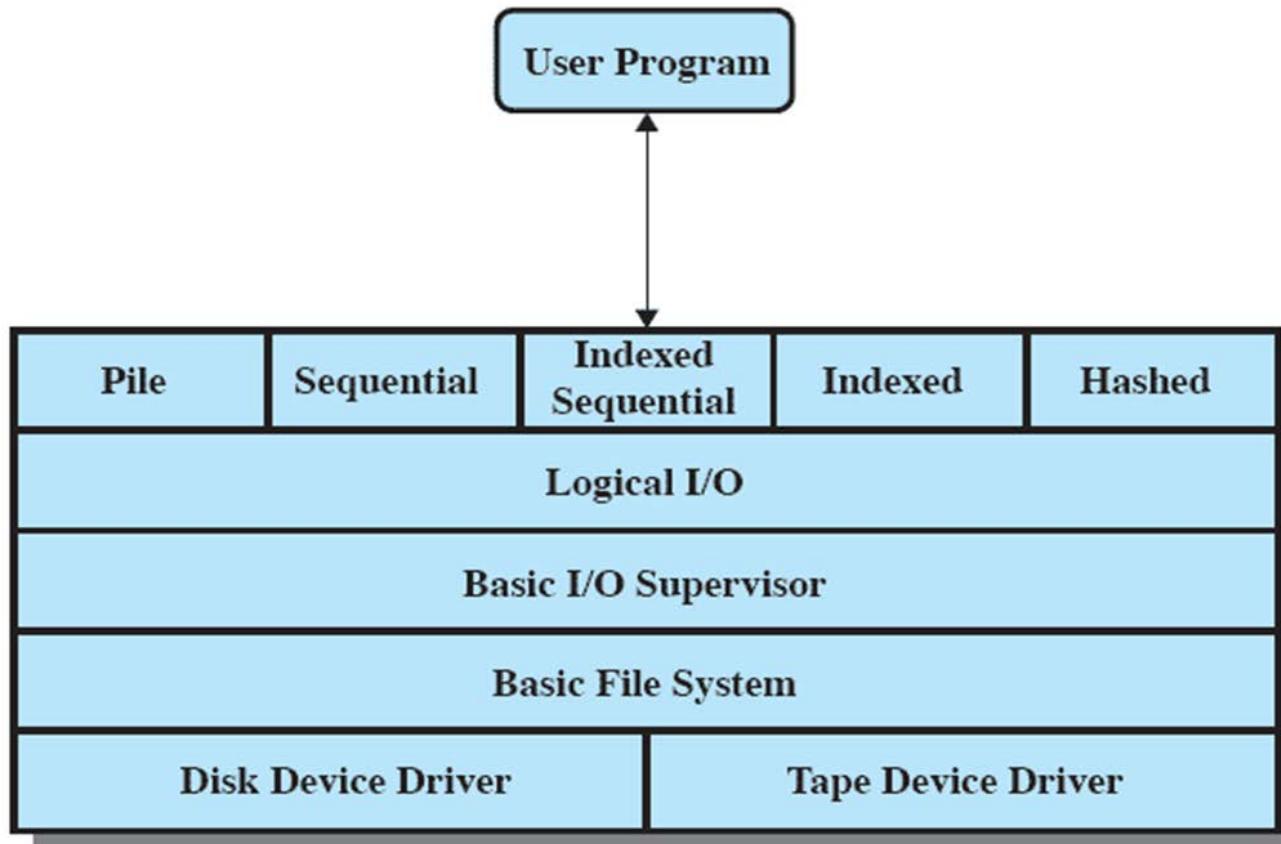
Record

File

Database

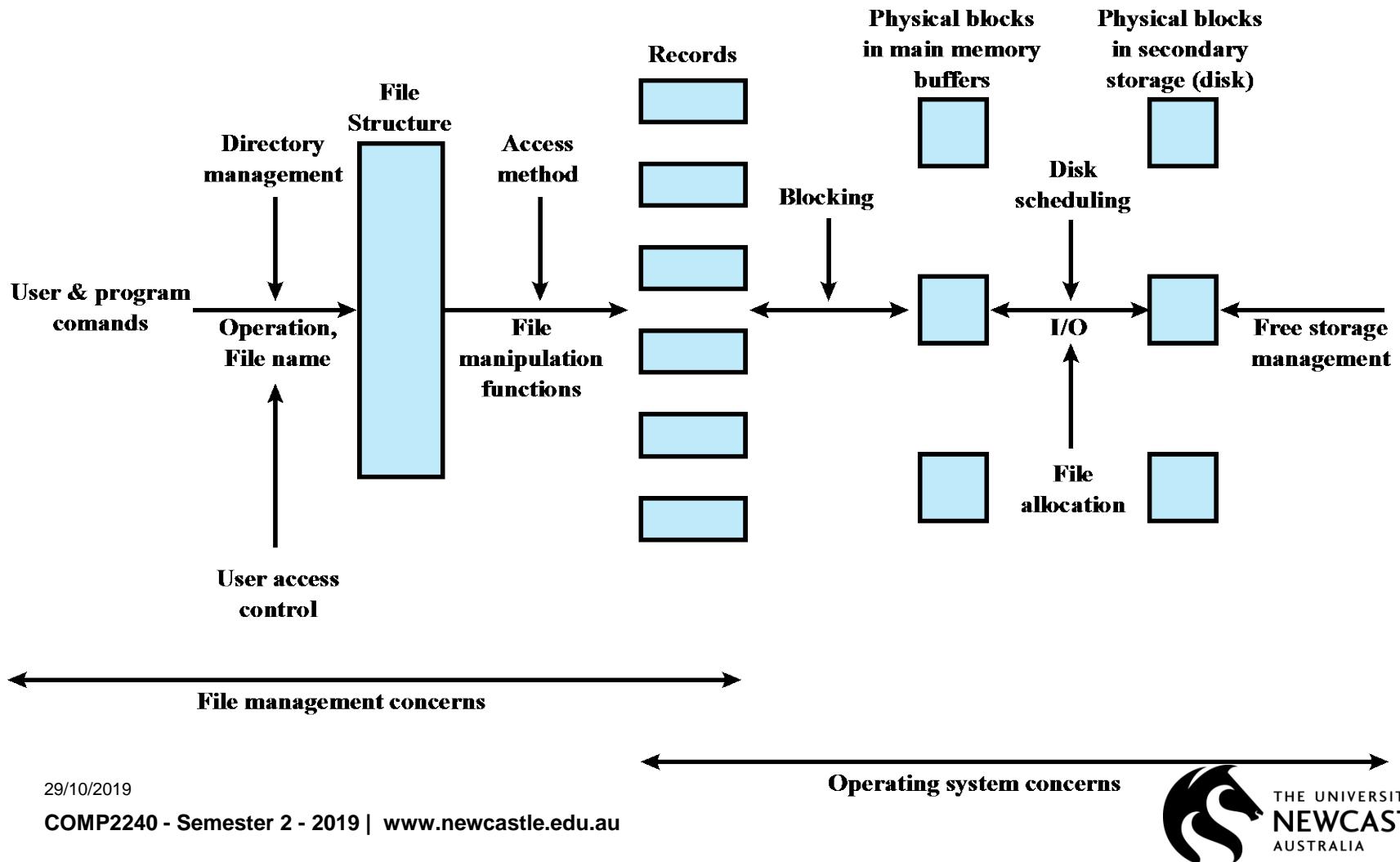
File system software architecture

293



Elements of file management

294



File organisation and access

295

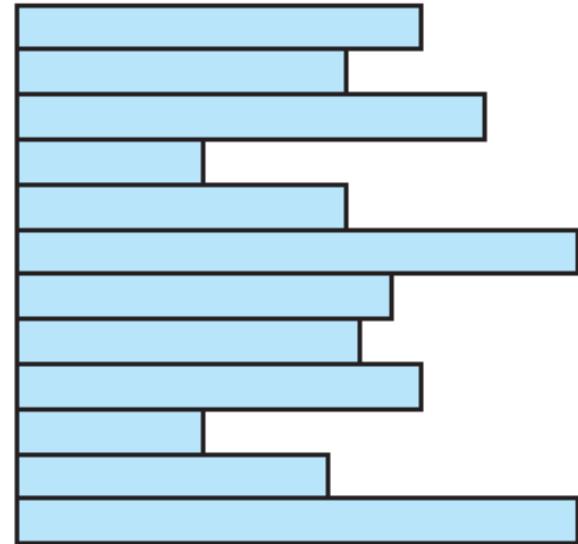
- File organization is the **logical structuring** of records as determined by the way in which they are accessed
- In choosing a file organization, several criteria are important:
 - short access time
 - ease of update
 - economy of storage
 - simple maintenance
 - reliability
- Priority of criteria depends on the application that will use the file



The Pile

296

- Least complicated form of file organization
- Data are collected in the order they arrive
- Each record consists of one burst of data
- Purpose is simply to accumulate the mass of data and save it
- Records may have different fields or differently ordered fields
 - each field should have a name and a value
 - field length and / delimiter should be used
- Record access is by exhaustive search



Variable-length records
Variable set of fields
Chronological order

(a) Pile File

Sequential file

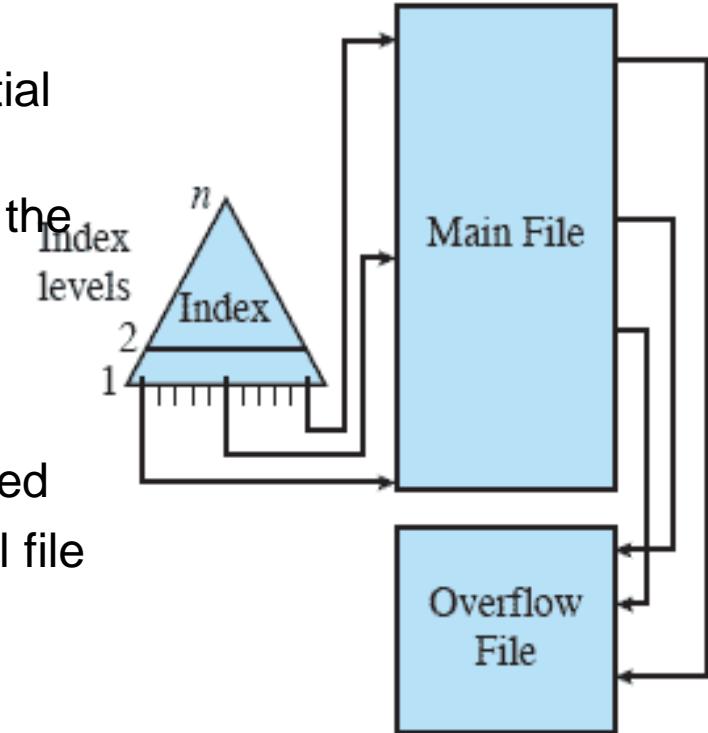
- Most common form of file structure
 - A fixed format is used for records
 - Key field **uniquely** identifies the record
 - Typically used in batch applications
 - Only organization that is easily stored on tape as well as disk
 - Suitable for batch applications
 - Poor performance for queries and/or update of individual records
 - Addition to the file is problematic
 - Use of linked list can be helpful

Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

(b) Sequential File

Indexed sequential file

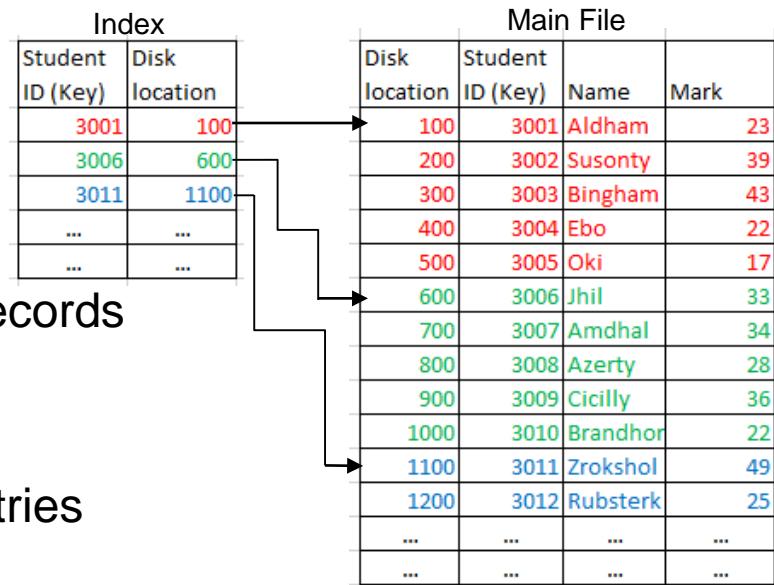
- Adds two additional features to the sequential file
 - An index to reach quickly the vicinity of the desired record
 - Adds an overflow file (similar to log file)
- Simplest case: a single level of indexing used
 - The index is itself is a simple sequential file
 - Two fields in record of the index:
 - (i) Key and (ii) pointer to main file



(c) Indexed Sequential File

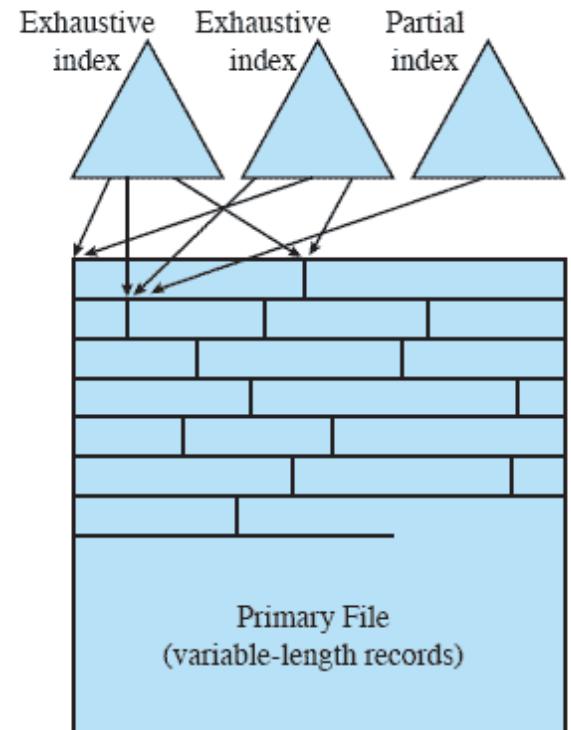
Indexed sequential file

- Greatly reduces the time required to access a single record
 - Consider a sequential file of 1million records
 - Use an index of 1000 entries
- Multiple levels of indexing can be used to provide greater efficiency in access
 - Consider a sequential file of 1million records
 - Use an lower level index of 10,000 entries
 - Use a higher level index of 100 entries



Indexed file

- Records are accessed only through their indexes
- Variable-length records can be employed
- Exhaustive index contains one entry for every record in the main file
- Partial index contains entries to records where the field of interest exists
- Used mostly in applications where timeliness of information is critical
 - Examples would be airline reservation systems and inventory control systems



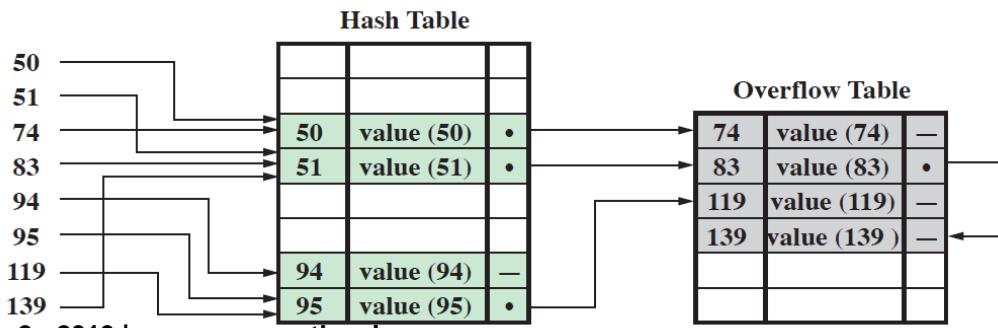
(d) Indexed File

Direct or hashed file

- Access directly any block of a known address
- Makes use of hashing on the key value
- Often used where:
 - very rapid access is required
 - fixed-length records are used
 - records are always accessed one at a time

Examples are:

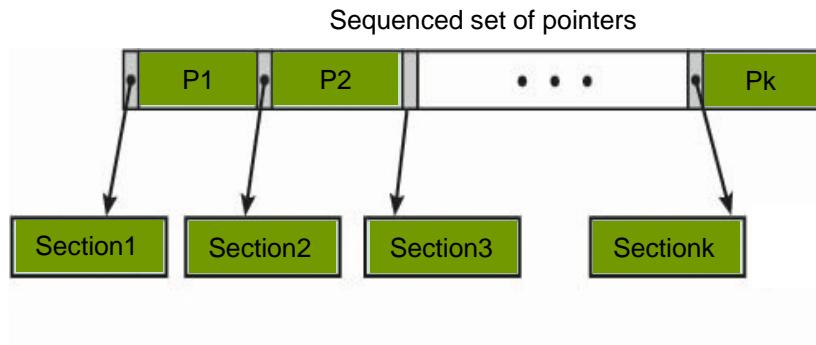
- directories
- pricing tables
- schedules
- name lists



Index structure

302

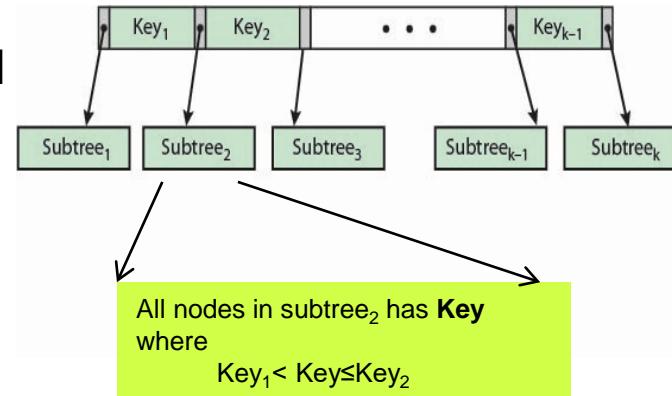
- There is common use of an index file to access individual records in a file or database.
 - For a large file or database, a single sequential file of indexes on the primary key does not provide for rapid access.
- To provide more efficient access, a structured index file is typically used.
 - The simplest such structure is a two-level organization in which the original file is broken into sections and the upper level consists of a sequenced set of pointers to the lower-level sections.
 - This structure can then be extended to more than two levels, resulting in a tree structure.



B-Tree

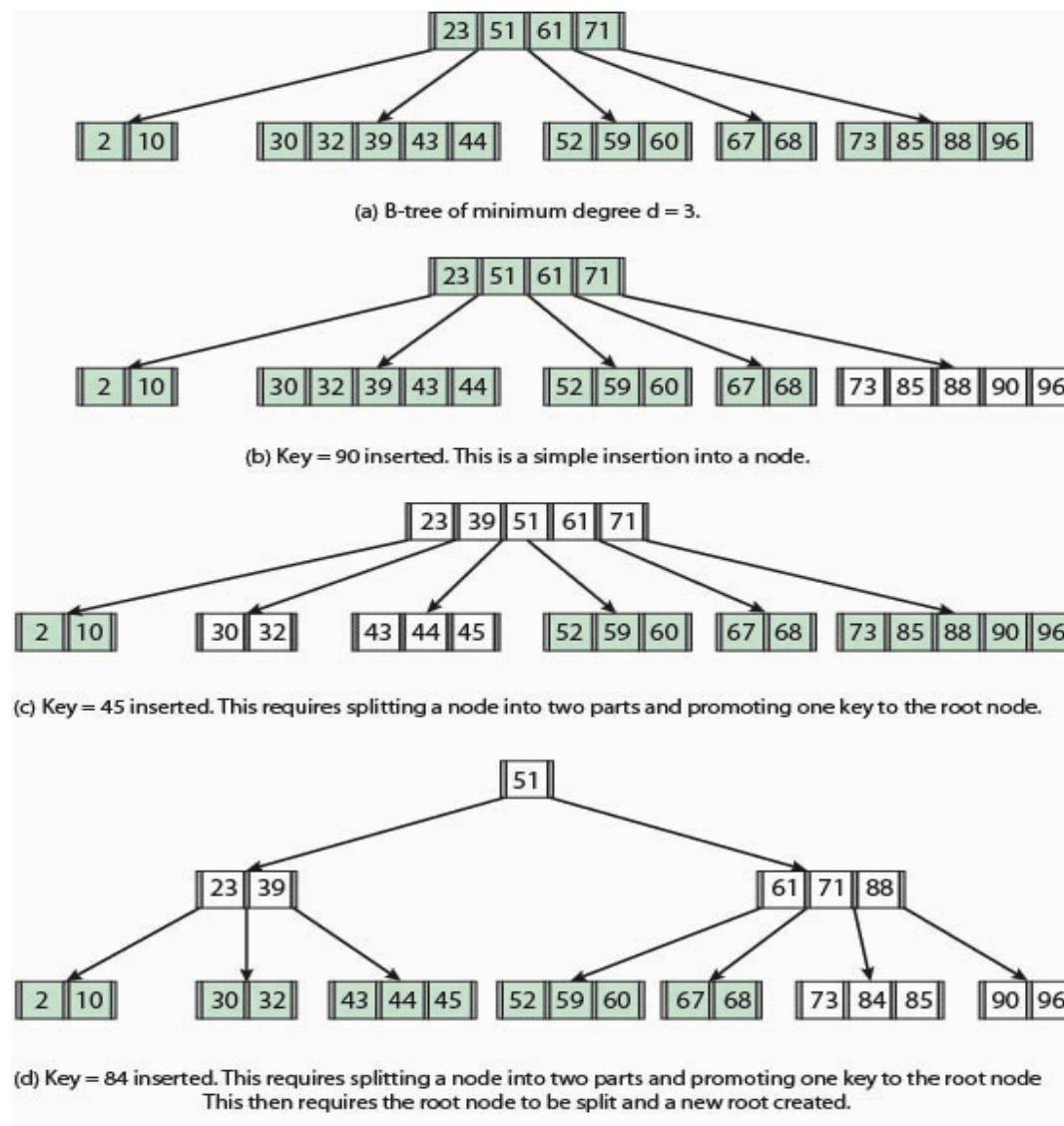
303

- A B-tree is a tree structure (no closed loops)
 - The tree consists of a number of nodes and leaves.
 - Each node contains at least one key which uniquely identifies a file record, and more than one pointer to child nodes or leaves.
 - Each node is limited to the same number of maximum keys.
 - The keys in a node are stored in nondecreasing order. Each node has one more pointer than keys.



Inserting nodes into a B-Tree

304



29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

Figure 12.5 Inserting Nodes into a B-tree

File directory information

Basic Information	
File Name	Name as chosen by creator (user or program). Must be unique within a specific directory.
File Type	For example: text, binary, load module, etc.
File Organization	For systems that support different organizations
Address Information	
Volume	Indicates device on which file is stored
Starting Address	Starting physical address on secondary storage (e.g., cylinder, track, and block number on disk)
Size Used	Current size of the file in bytes, words, or blocks
Size Allocated	The maximum size of the file
Access Control Information	
Owner	User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges.
Access Information	A simple version of this element would include the user's name and password for each authorized user.
Permitted Actions	Controls reading, writing, executing, transmitting over a network
Usage Information	
Date Created	When file was first placed in directory
Identity of Creator	Usually but not necessarily the current owner
Date Last Read Access	Date of the last time a record was read
Identity of Last Reader	User who did the reading
Date Last Modified	Date of the last update, insertion, or deletion
Identity of Last Modifier	User who did the modifying
Date of Last Backup	Date of the last time the file was backed up on another storage medium
Current Usage	Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk

Operations performed on a directory

306

- To understand the requirements for a file structure, it is helpful to consider the types of operations that may be performed on the directory



Search

Create files

Delete files

List directory

Update directory

- Use of a simple list for file names?

Two-level scheme

307

- There is one directory for each user and a master directory
- Master directory has an entry for each user directory providing address and access control information
- Each user directory is a simple list of the files of that user
- Names must be unique only within the collection of files of a single user
- File system can easily enforce access restriction on directories
- Does not help user in structuring collections of files

Tree-structured directory

- Master directory with user directories underneath it
- Each user directory may have subdirectories and files as entries
- Organization of directory:
 - Simplest approach is a sequential file
 - If directories may contain a very large number of entries then hashed structure may be preferred

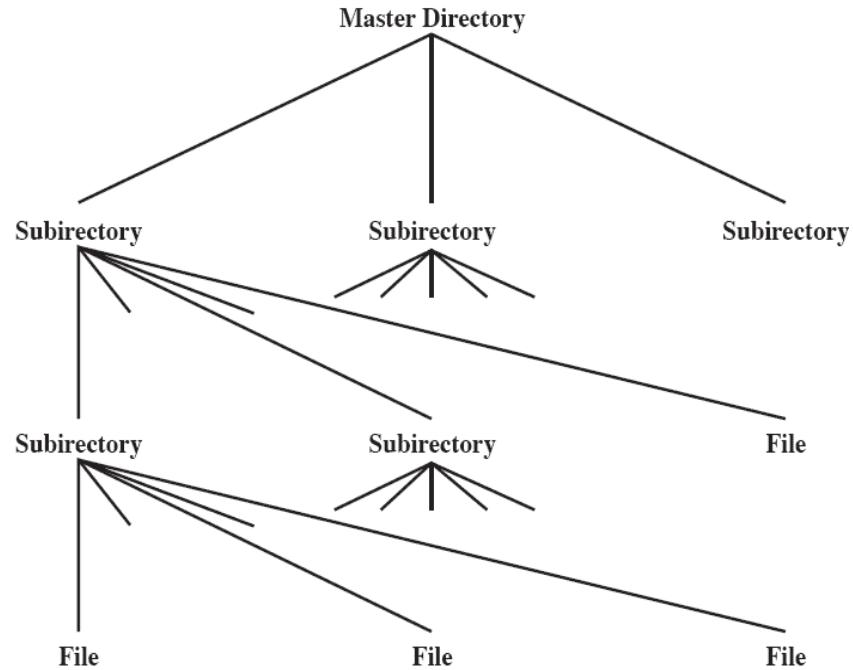


Figure 12.4 Tree-Structured Directory

Tree-structured directories

- Master directory with user directories underneath it
- Each user directory may have subdirectories and files as entries
- Organization of directory:
 - Simplest approach is a sequential file
 - If directories may contain a very large number of entries then hashed structure may be preferred

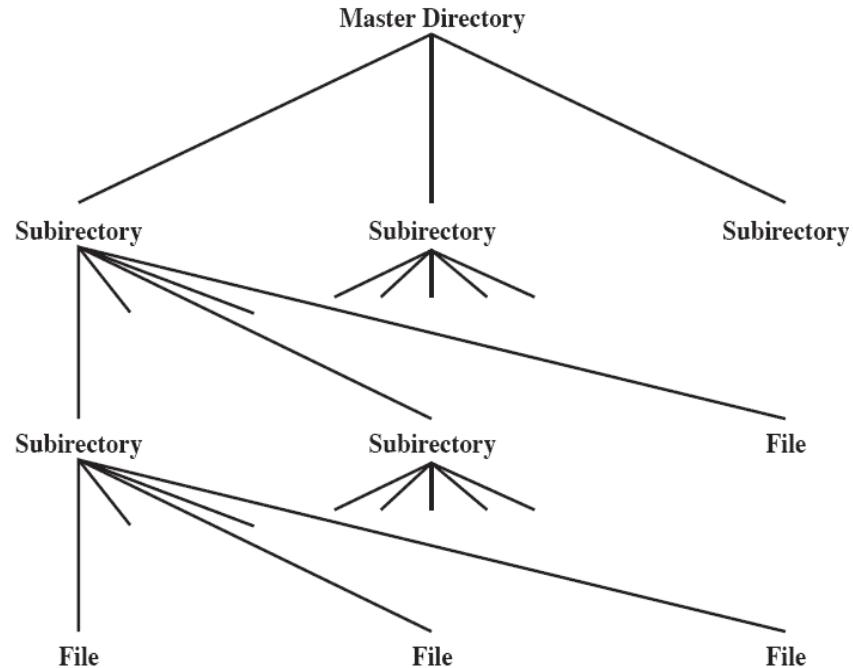
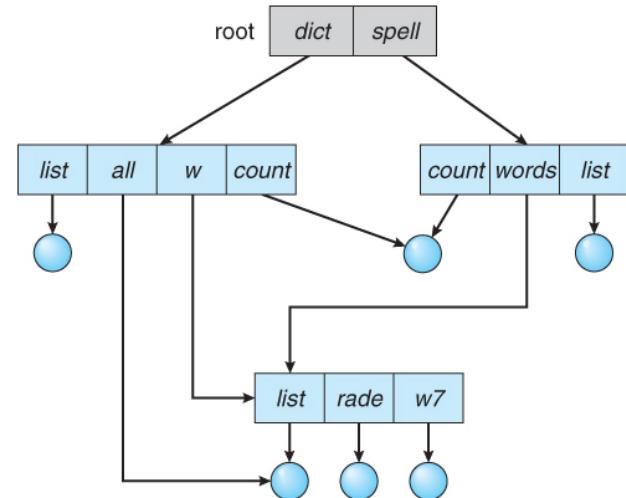


Figure 12.4 Tree-Structured Directory

Acyclic-Graph directories

- The same file/subdirectory may belong to different directories
- Useful for sharing
- Common way to implement using **link**
 - A link is effectively a pointer
 - May be implemented as a pathname
- More complexities involved in accessing, deleting, renaming files and other operations.



File sharing

311

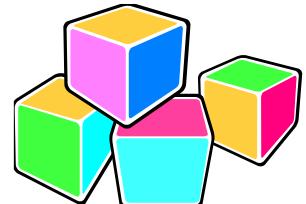
Two issues arise
when allowing files
to be shared among
a number of users:

access rights

management of
simultaneous
access

Record blocking

- Blocks are the unit of I/O with secondary storage
 - for I/O to be performed records must be organized as blocks
- Most systems use fixed length block
 - Simplifies IO and buffer allocation in main memory
- Larger blocks reduces I/O transfer time but needs larger buffer
- Given the size of a block, three methods of blocking can be used:



Fixed blocking

1) Fixed Blocking – fixed-length records are used, and an integral number of records are stored in a block

Internal fragmentation – unused space at the end of each block

Common mode for sequential files with fixed-length record.



Fixed Blocking



Data



Waste due to record fit to block size



Gaps due to hardware design



Waste due to block size constraint from fixed record size

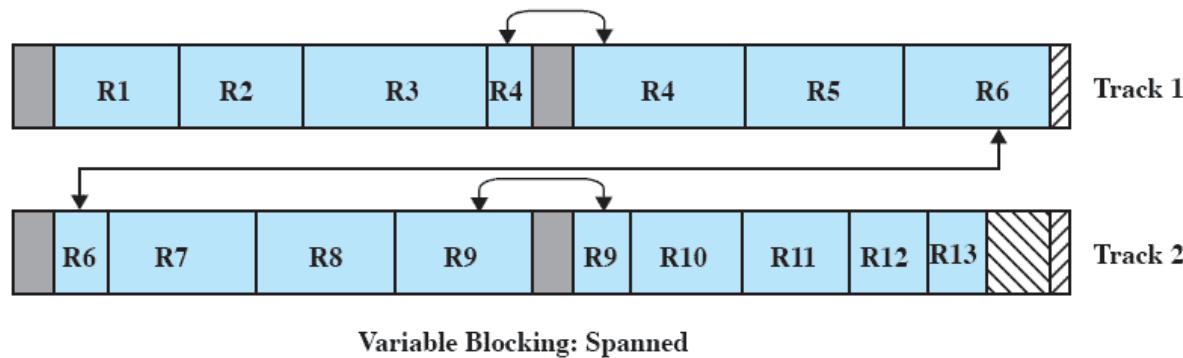


Waste due to block fit to track size

Variable blocking: spanned

2) Variable-Length Spanned Blocking – variable-length records are used and are packed into blocks with no unused space

Efficient of storage, not limited by the record size but difficult to implement



Data



Waste due to record fit to block size



Gaps due to hardware design



Waste due to block size constraint from fixed record size

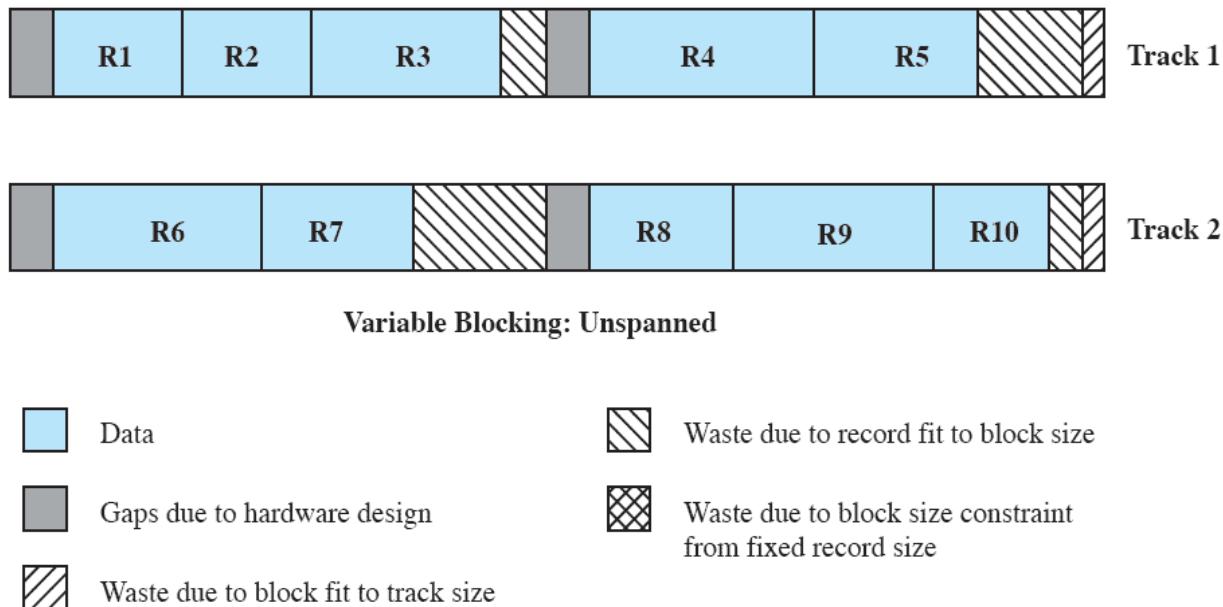


Waste due to block fit to track size

Variable blocking: unspanned

3) Variable-Length Unspanned Blocking – variable-length records are used, but spanning is not employed

Wastes space and limits record size to the size of a block



File allocation

316

- On secondary storage, a file consists of a collection of blocks
- The operating system or file management system is responsible for **allocating blocks to files**
- The approach taken for file allocation may influence the approach taken for **free space management**
- These two tasks are related.



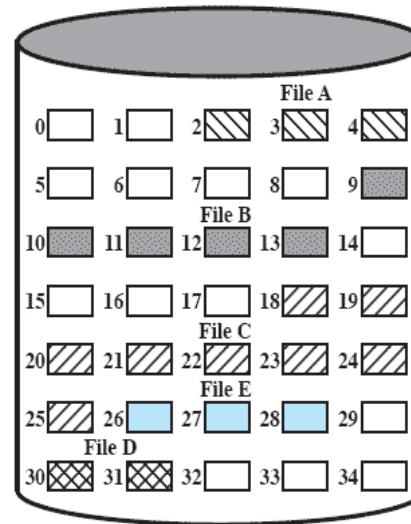
File allocation methods

317

	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

Contiguous file allocation

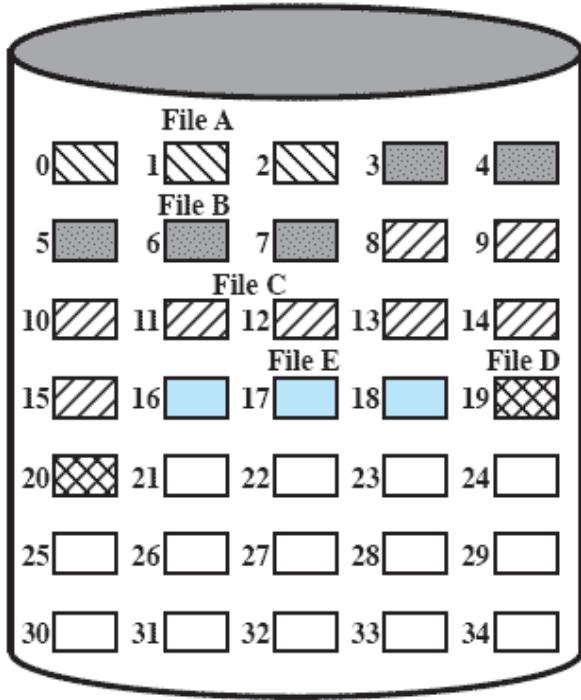
- A single contiguous set of blocks is allocated to a file at the time of file creation
- Preallocation strategy using variable-size portions
- A single entry for each file in file allocation table
- Is the best from the point of view of the individual sequential file
 - Multiple blocks can be read in at a time
 - Easy to retrieve a single block



File Allocation Table		
File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figure 12.7 Contiguous File Allocation

After compaction



File Allocation Table

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figure 12.8 Contiguous File Allocation (After Compaction)

Chained allocation

- Allocation is on an individual block basis
- Each block contains a pointer to the next block in the chain
- The file allocation table needs just a single entry for each file
- Pre-allocation is possible but more common to allocate blocks as needed
- No external fragmentation to worry about
- Best for sequential files to be processed sequentially
- No accommodation of the principle of locality

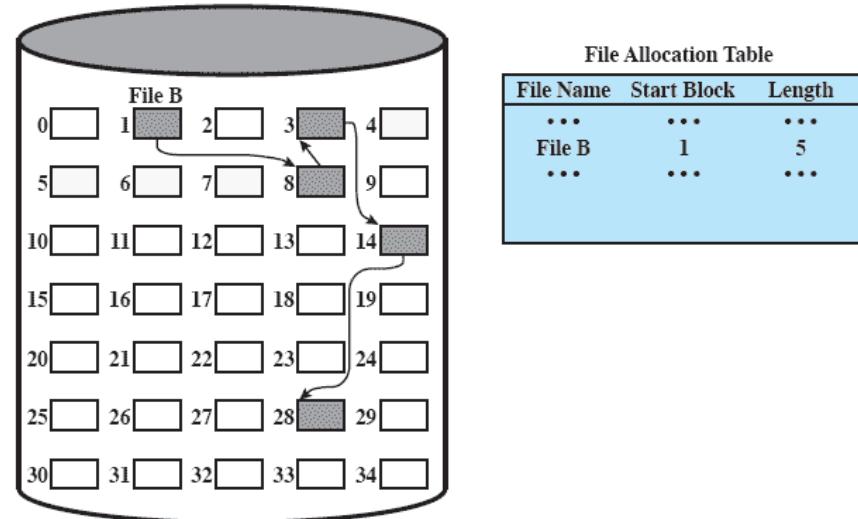


Figure 12.9 Chained Allocation

Chained allocation after consolidation

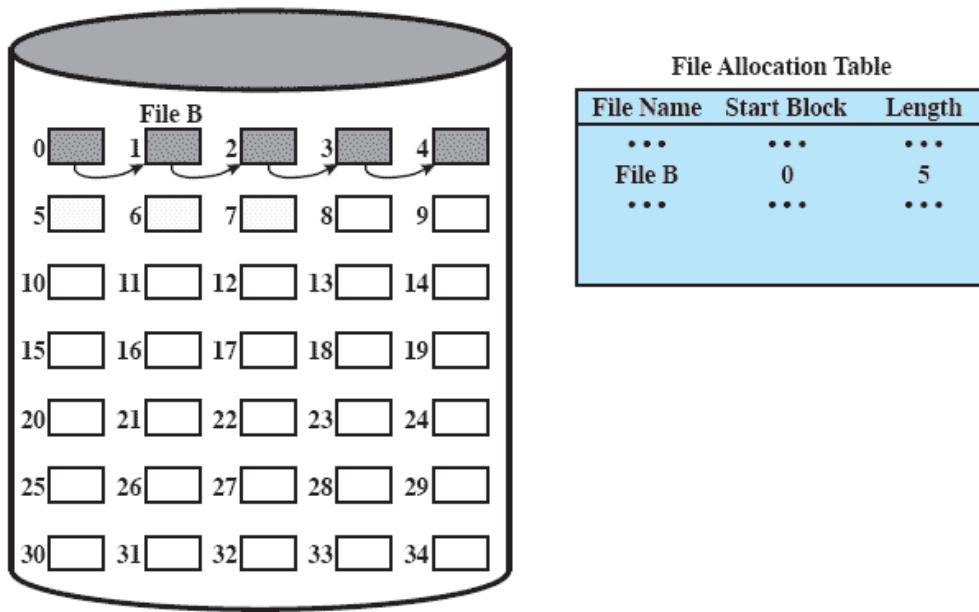


Figure 12.10 Chained Allocation (After Consolidation)

Indexed allocation

- Addresses many of the problems of the other two methods
- File allocation table contains a separate one-level index for each file.
 - Index has one entry for each portion allocated to the file
 - Typically index are not physically stored in the file allocation table rather file index is kept in a separate block.

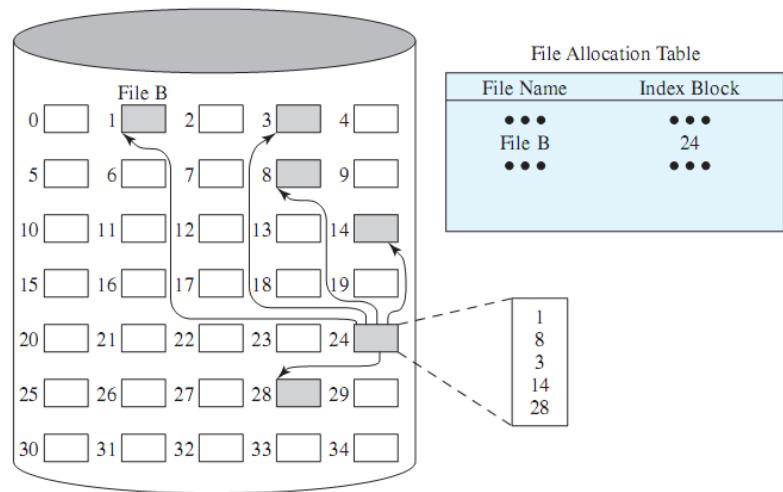


Figure 12.11 Indexed Allocation with Block Portions

Indexed allocation

Block Portion VS Variable length portion

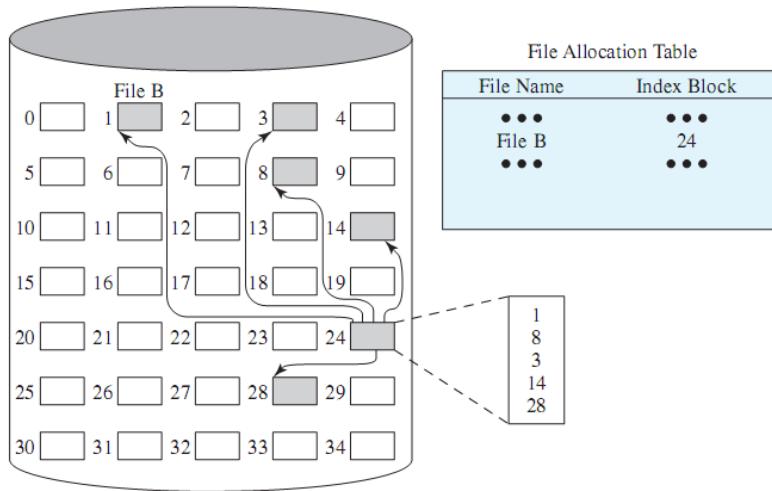


Figure 12.11 Indexed Allocation with Block Portions

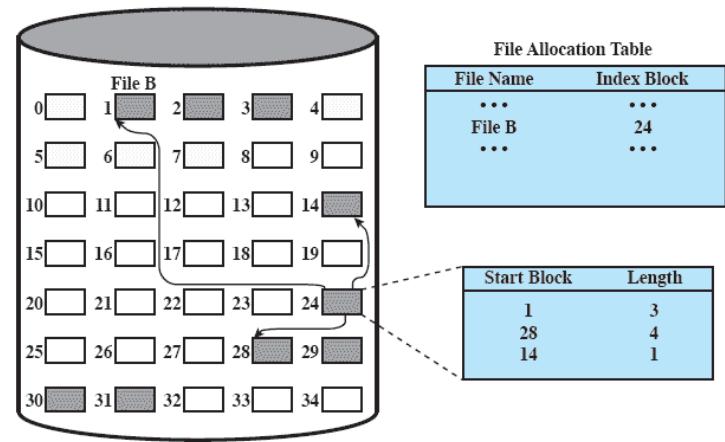


Figure 12.12 Indexed Allocation with Variable-Length Portions

Free Space Management

324

- Just as allocated space must be managed, so must the unallocated space
- To perform file allocation, it is necessary to know which blocks are available
- A ***disk allocation table*** is needed in addition to a file allocation table



Bit Tables

325

- This method uses a vector containing one bit for each block on the disk
- Each entry of a 0 corresponds to a free block, and each 1 corresponds to a block in use

Advantages:

- easy to find one or a contiguous group of free blocks
- works well with any file allocation method
- it is as small as possible

Chained Free Portions

326

- The free portions may be chained together by using a pointer and length value in each free portion
- Negligible space overhead because there is no need for a disk allocation table
- Suited to all file allocation methods

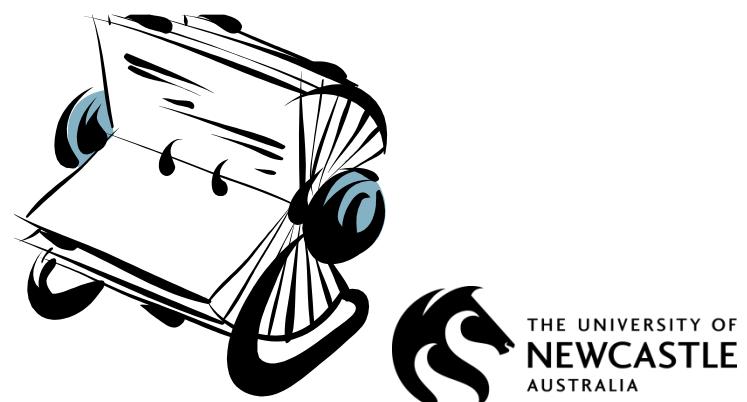
Disadvantages:

- every time you allocate a block you need to read the block first to recover the pointer to the new first free block before writing data to that block

Indexing

327

- Treats free space as a file and uses an index table as it would for file allocation
- For efficiency, the index should be on the basis of variable-size portions rather than blocks
- One entry in the table for every free portion on the disk
- This approach provides efficient support for all of the file allocation methods



Free Block List

Each block is assigned a number sequentially

the list of the numbers of all free blocks is maintained in a reserved portion of the disk

Depending on the size of the disk, either 24 or 32 bits will be needed to store a single block number

the size of the free block list is 24 or 32 times the size of the corresponding bit table and must be stored on disk

There are two effective techniques for storing a small part of the free block list in main memory:

the list can be treated as a push-down stack with the first few thousand elements of the stack kept in main memory

the list can be treated as a FIFO queue, with a few thousand entries from both the head and the tail of the queue in main memory

Security and Protection
WEEK 11

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au



Computer Security

- **Computer security: The protection afforded to an automated information system**
 - in order to attain the applicable objectives of preserving the integrity, availability, and confidentiality of information system resources
 - includes hardware, software, firmware, information/data, and telecommunications.





CIA Triad

- Security Objectives:
 - Confidentiality
 - a loss of confidentiality is the unauthorized disclosure of information
 - Integrity
 - a loss of integrity is the unauthorized modification or destruction of information
 - Availability
 - a loss of availability is the disruption of access to or use of information or an information system

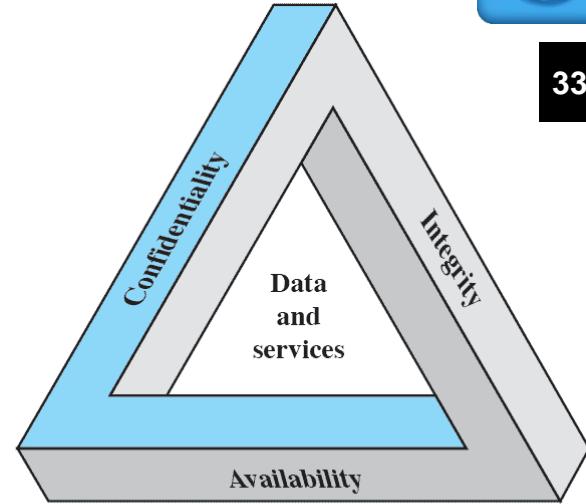


Figure 14.1 The Security Requirements Triad



332

Threat consequences

- Unauthorized Disclosure
- Deception
- Disruption
- Usurpation



Threats Areas and Examples

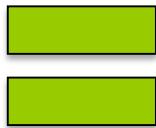
333

	Availability	Confidentiality	Integrity
Hardware	Equipment is stolen or disabled, thus denying service.		
Software	Programs are deleted, denying access to users.	An unauthorized copy of software is made.	A working program is modified, either to cause it to fail during execution or to cause it to do some unintended task.
Data	Files are deleted, denying access to users.	An unauthorized read of data is performed. An analysis of statistical data reveals underlying data.	Existing files are modified or new files are fabricated.
Communication Lines	Messages are destroyed or deleted. Communication lines or networks are rendered unavailable.	Messages are read. The traffic pattern of messages is observed.	Messages are modified, delayed, reordered, or duplicated. False messages are fabricated.

System Access Threats

334

System access threats fall into two general categories:



Intruders



Malicious software

Intruders

335

Masquerader

an individual who is not authorized to use the computer and who penetrates a system's access controls to exploit a legitimate user's account

Misfeasor

a legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges

Clandestine user

an individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection

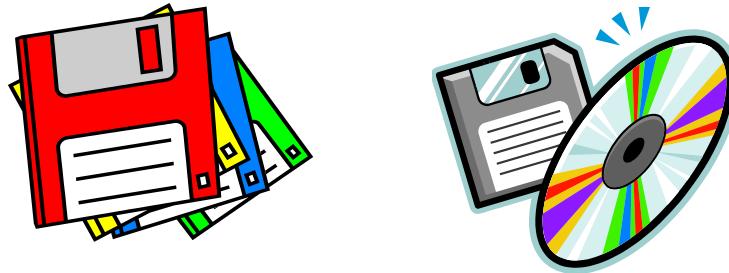
Intruders

336

- Intruders attacks range from the benign to the serious
- The objective of the intruder is to gain access to a system OR to increase the range of privileges accessible on a system
- Most attacks use system or software vulnerabilities that allow user to execute code that opens a backdoor into the system
- Alternatively intruders attempts to acquire information that should have been protected.

Malware

337



General term for any malicious software

Software designed to cause damage to or use up the resources of a target computer

Frequently concealed within or masquerades as legitimate software

In some cases it spreads itself to other computers via e-mail or infected discs

Terminology of Malicious Programs

338

Name	Description
Virus	Malware that, when executed, tries to replicate itself into other executable code; when it succeeds the code is said to be infected. When the infected code is executed, the virus also executes.
Worm	A computer program that can run independently and can propagate a complete working version of itself onto other hosts on a network.
Logic bomb	A program inserted into software by an intruder. A logic bomb lies dormant until a predefined condition is met; the program then triggers an unauthorized act.
Trojan horse	A computer program that appears to have a useful function, but also has a hidden and potentially malicious function that evades security mechanisms, sometimes by exploiting legitimate authorizations of a system entity that invokes the Trojan horse program.
Backdoor (trapdoor)	Any mechanisms that bypasses a normal security check; it may allow unauthorized access to functionality.
Mobile code	Software (e.g., script, macro, or other portable instruction) that can be shipped unchanged to a heterogeneous collection of platforms and execute with identical semantics.
Exploits	Code specific to a single vulnerability or set of vulnerabilities.
Downloaders	Program that installs other items on a machine that is under attack. Usually, a downloader is sent in an e-mail.
Auto-rooter	Malicious hacker tools used to break into new machines remotely.
Kit (virus generator)	Set of tools for generating new viruses automatically.
Spammer programs	Used to send large volumes of unwanted e-mail.
Flooders	Used to attack networked computer systems with a large volume of traffic to carry out a denial-of-service (DoS) attack.
Keyloggers	Captures keystrokes on a compromised system.
Rootkit	Set of hacker tools used after attacker has broken into a computer system and gained root-level access.
Zombie, bot	Program activated on an infected machine that is activated to launch attacks on other machines.
Spyware	Software that collects information from a computer and transmits it to another system.
Adware	Advertising that is integrated into software. It can result in pop-up ads or redirection of a browser to a commercial site.



Backdoor

339

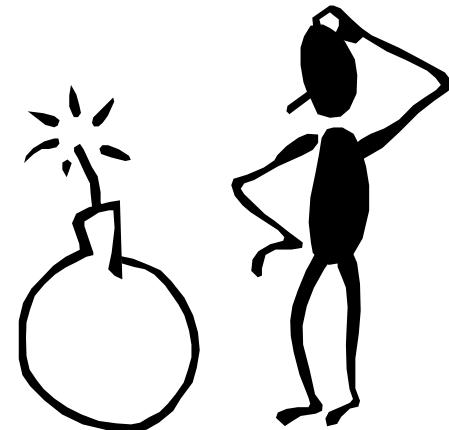


- Also known as a trapdoor
- A secret entry point into a program that allows someone to gain access without going through the usual security access procedures
- A ***maintenance hook*** is a backdoor that programmers use to debug and test programs
- Become threats when unscrupulous programmers use them to gain unauthorized access
- It is difficult to implement operating system controls for backdoors

Logic Bomb

340

- One of the oldest types of program threat
- Code embedded in some legitimate program that is set to “explode” when certain conditions are met
- Once triggered a bomb may alter or delete data or entire files, cause a machine halt, or do some other damage



Trojan Horse

- Useful, or apparently useful, program or command procedure that contains hidden code that, when invoked, performs some unwanted or harmful function
- Trojan horses fit into one of three models:
 - 1) continuing to perform the function of the original program and additionally performing a separate malicious activity
 - 2) continuing to perform the function of the original program but modifying the function to perform malicious activity or to disguise other malicious activity
 - 3) performing a malicious function that completely replaces the function of the original program



Viruses

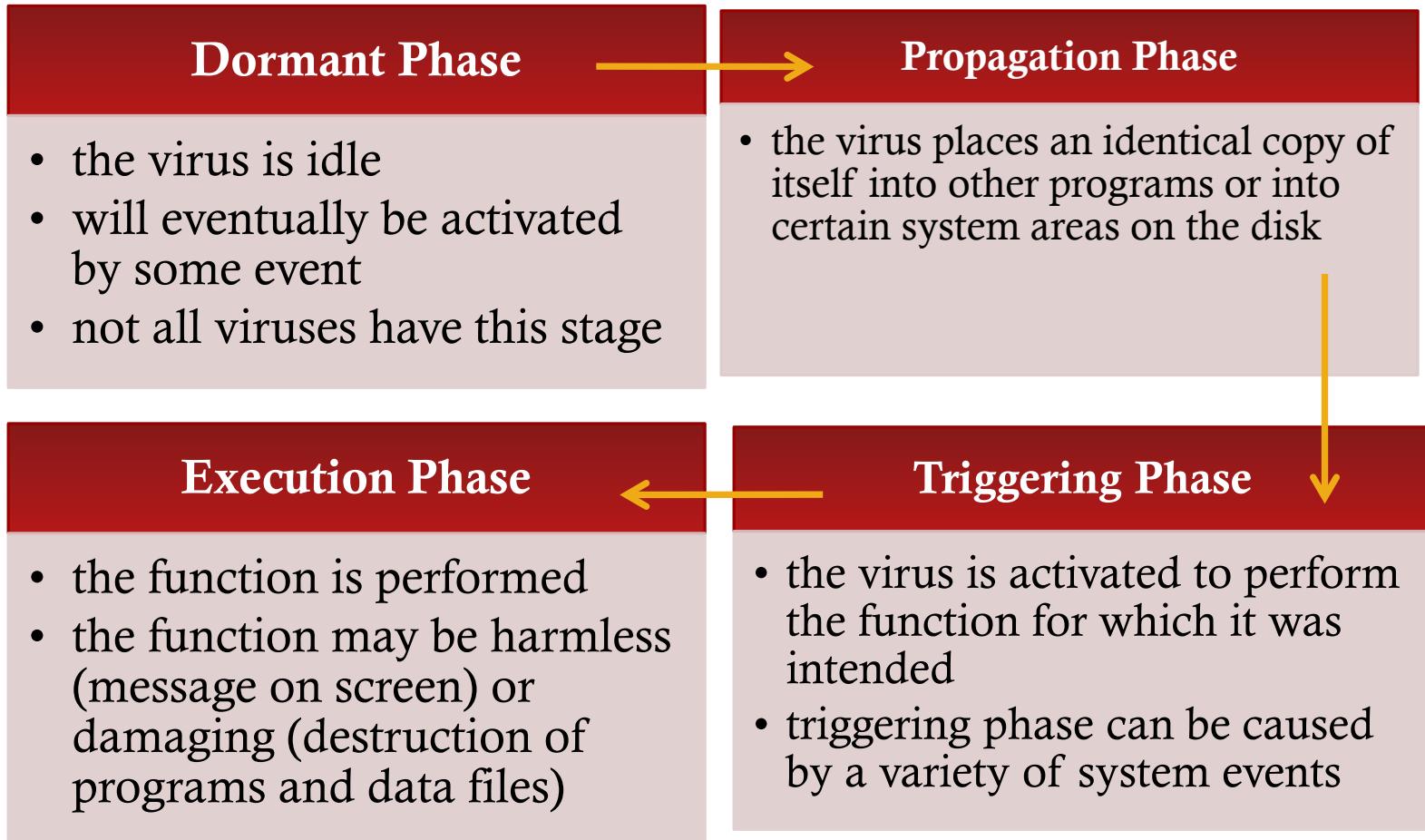
342

- Software that “infects” other programs by modifying them
 - carries instructional code to self duplicate
 - becomes embedded in a program on a computer
 - when the infected computer comes into contact with an uninfected piece of software, a fresh copy of the virus passes into the new program
 - infection can be spread by swapping disks from computer to computer or through a network
- A computer virus has **three parts**:
 - an infection mechanism
 - trigger
 - payload



Virus Phases

343



Simple Virus

- Virus can be prepended or postpended to an executable program
- Can also be embedded in some other fashion
- The key to its operation is that the infected program, when invoked, will first execute the virus code and then execute the original code of the program

```
program V :=  
  
{goto main;  
 1234567;  
  
subroutine infect-executable :=  
  {loop:  
   file := get-random-executable-file;  
   if (first-line-of-file = 1234567)  
     then goto loop  
   else prepend V to file; }  
  
subroutine do-damage :=  
  {whatever damage is to be done}  
  
subroutine trigger-pulled :=  
  {return true if some condition holds}  
  
main:  main-program :=  
        {infect-executable;  
         if trigger-pulled then do-damage;  
         goto next; }  
  
next:  
 }
```

Compression Virus

- In this example the virus does nothing other than propagate
- One way to thwart detection is to compress the executable file so that both the infected and uninfected versions are of identical length

```
program CV :=
{goto main;
 01234567;

subroutine infect-executable :=
  {loop:
    file := get-random-executable-file;
    if (first-line-of-file = 01234567) then goto loop;
    (1)   compress file;
    (2)   prepend CV to file;
  }

main: main-program :=
  {if ask-permission then infect-executable;
  (3)   uncompress rest-of-file;
  (4)   run uncompressed file;}
}
```

Virus Concealment Strategy

- A virus **classification by concealment strategy** includes:

Encrypted virus

random encryption
key encrypts
remainder of virus

Stealth virus

hides itself from
detection of
antivirus software

Polymorphic virus

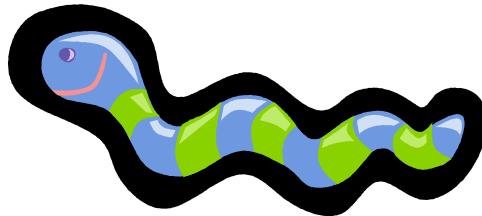
mutates with every
infection

Metamorphic virus

mutates with every
infection

mutation engine is the
portion of the virus that
is responsible for
generating keys and
performing
encryption/decryption

Worms



347

- A program that can replicate itself and send copies from computer to computer across network connections
- Upon arrival the worm may be activated to replicate and propagate again
- In addition to propagation the worm usually performs some unwanted function
- Actively seeks out more machines to infect and each machine that is infected serves as an automate launching pad for attacks on other machines

Worm Propagation

348

- To replicate itself a network worm uses some sort of network vehicle

Electronic mail facility

- a worm mails a copy of itself to other systems so that its code is run when the e-mail or an attachment is received or viewed

Remote execution capability

- a worm executes a copy of itself on another system either using an explicit remote execution facility or by exploiting a program flaw in a network service to subvert its operations

Remote log-in capability

- a worm logs on to a remote system as a user and then uses commands to copy itself from one system to the other

Uses of Bots

Distributed denial-of-service (DDoS) attacks

- causes a loss of service to users

Spamming

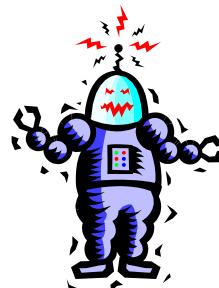
- sending massive amounts of bulk e-mail (spam)

Sniffing traffic

- a packet sniffer is used to retrieve sensitive information like user names and passwords

Keylogging

- captures keystrokes



Spreading new malware

- botnets are used to spread new bots

Installing advertisement add-ons and browser helper objects (BHOs)

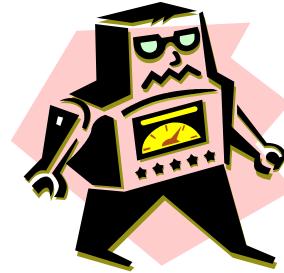
- set up a fake Web site and negotiate a deal with hosting companies that pay for clicks on ads

Attacking Internet Relay chat (IRC) chat networks

- victim is flooded with requests, bringing down the IRC network; similar to a DDoS attack

Manipulating online polls/games

- every bot has a distinct IP address so it appears to be a real person



Remote Control Facility

- Distinguishes a bot from a worm
 - a worm propagates and activates itself, whereas a bot is controlled from some central facility (at least initially)
- A typical means of implementing the remote control facility is on an IRC server
 - all bots join a specific channel on this server and treat incoming messages as commands
- More recent botnets tend to use covert communication channels via protocols such as HTTP
- Distributed control mechanisms are also used to avoid a single point of failure

Buffer Overflow Attacks

- Also known as a ***buffer overrun***
- Defined in the NIST (National Institute of Standards and Technology) *Glossary of Key Information Security Terms* as:

“A condition at an interface under which more input can be placed into a buffer or data-holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system”

- One of the most prevalent and dangerous types of security attacks

```

int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncpy(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}

```

(a) Basic buffer overflow C code

```

$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)

```

(b) Basic buffer overflow example runs

Memory Address	Before gets(str2)	After gets(str2)	Contains value of
.....	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
.....	
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
.....	
bffffbe8	08fcffbf	08fcffbf	old base ptr
.....	
bffffbe4	00000000	01000000	valid
.....	
bffffbe0	80640140 . d . @	00640140 . d . @	
.....	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
.....	
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
.....	
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
.....	
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
.....	

Figure 15.2 Basic Buffer Overflow Stack Values

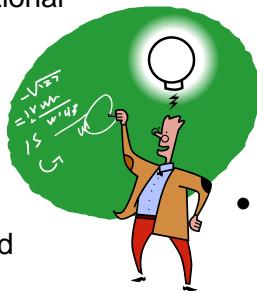
Exploiting Buffer Overflow

- To exploit any type of buffer overflow the attacker needs:
 - To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attackers control
 - To understand how that buffer will be stored in the processes memory, and hence the potential for corrupting adjacent memory locations and potentially altering the flow of execution of the program



Compile-time Techniques

- **Choice of programming language**
 - one possibility is to write the program using a modern high-level programming language that has a strong notion of variable type and what constitutes permissible operations on them
 - the flexibility and safety provided by these languages does come at a cost in resource use, both at compile time and also in additional code that must execute at runtime
- **Language extensions and use of safe libraries**
 - there have been a number of proposals to augment compilers to automatically insert range checks on pointer references
 - Libsafe is an example that implements the standard semantics but includes additional checks to ensure that the copy operations do not extend beyond the local variable space in the stack frame
- **Safe coding techniques**
 - programmers need to inspect the code and rewrite any unsafe coding constructs
 - an example is the OpenBSD project which produces a free, multiplatform 4.4BSD-based UNIX-like operating system
 - among other technology changes, programmers have undertaken an extensive audit of the existing code base, including the operating system, standard libraries, and common utilities
- **Stack protection mechanisms**
 - an effective method for protecting programs against classic stack overflow attacks is to instrument the function entry and exit code to set up and then check its stack frame for any evidence of corruption
 - Stackguard, one of the best-known protection mechanisms, is a GNU C Compiler Collection (GCC) compiler extension that inserts additional function entry and exit code



Runtime Techniques

- **Executable address space protection**
 - a possible defense is to block the execution of code on the stack, on the assumption that executable code should only be found elsewhere in the processes address space
 - extensions have been made available to Linux, BSD, and other UNIX-style systems to support the addition of the no-execute bit
- **Address space randomization**
 - a runtime technique that can be used to thwart attacks involves manipulation of the location of key data structures in the address space of a process
 - moving the stack memory region around by a megabyte or so has minimal impact on most programs but makes predicting the targeted buffer's address almost impossible
 - another technique is to use a security extension that randomizes the order of loading standard libraries by a program and their virtual memory address locations
- **Guard pages**
 - gaps are placed between the ranges of addresses used for each of the components of the address space
 - these gaps, or guard pages, are flagged in the MMU as illegal addresses and any attempt to access them results in the process being aborted
 - a further extension places guard pages between stack frames or between different allocations on the heap



Authentication

357

- In most computer security contexts, user authentication is the fundamental building block and the primary line of defense
- RFC 4949 defines user authentication as the process of verifying an identity claimed by or for a system entity
- An authentication process consists of two steps:
 - identification step
 - » presenting an identifier to the security system
 - verification step
 - » presenting or generating authentication information that corroborates the binding between the entity and the identifier



Means of Authentication

358

- **Something the individual knows**
 - examples include a password, a personal identification number (PIN), or answers to a prearranged set of questions
- **Something the individual possesses**
 - examples include electronic keycards, smart cards, and physical keys
 - referred to as a *token*
- **Something the individual is (static biometrics)**
 - examples include recognition by fingerprint, retina, and face
- **Something the individual does (dynamic biometrics)**
 - examples include recognition by voice pattern, handwriting characteristics, and typing rhythm

Password-based authentication

- A widely used line of defense against intruders is the password system
- The password serves to authenticate the ID of the individual logging on to the system
- The ID provides security by:

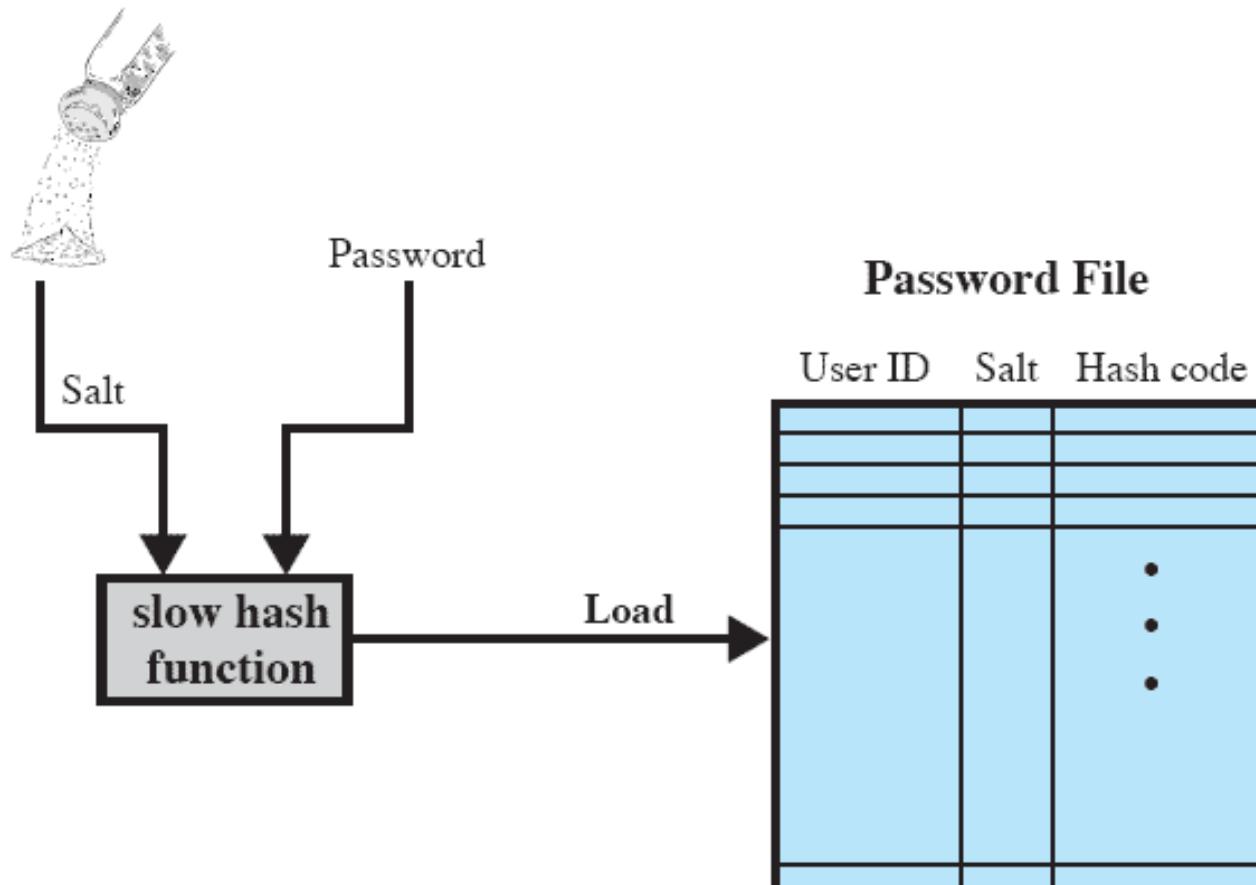
determining whether the user is authorized to gain access to a system

determining the privileges accorded to the user

discretionary access control

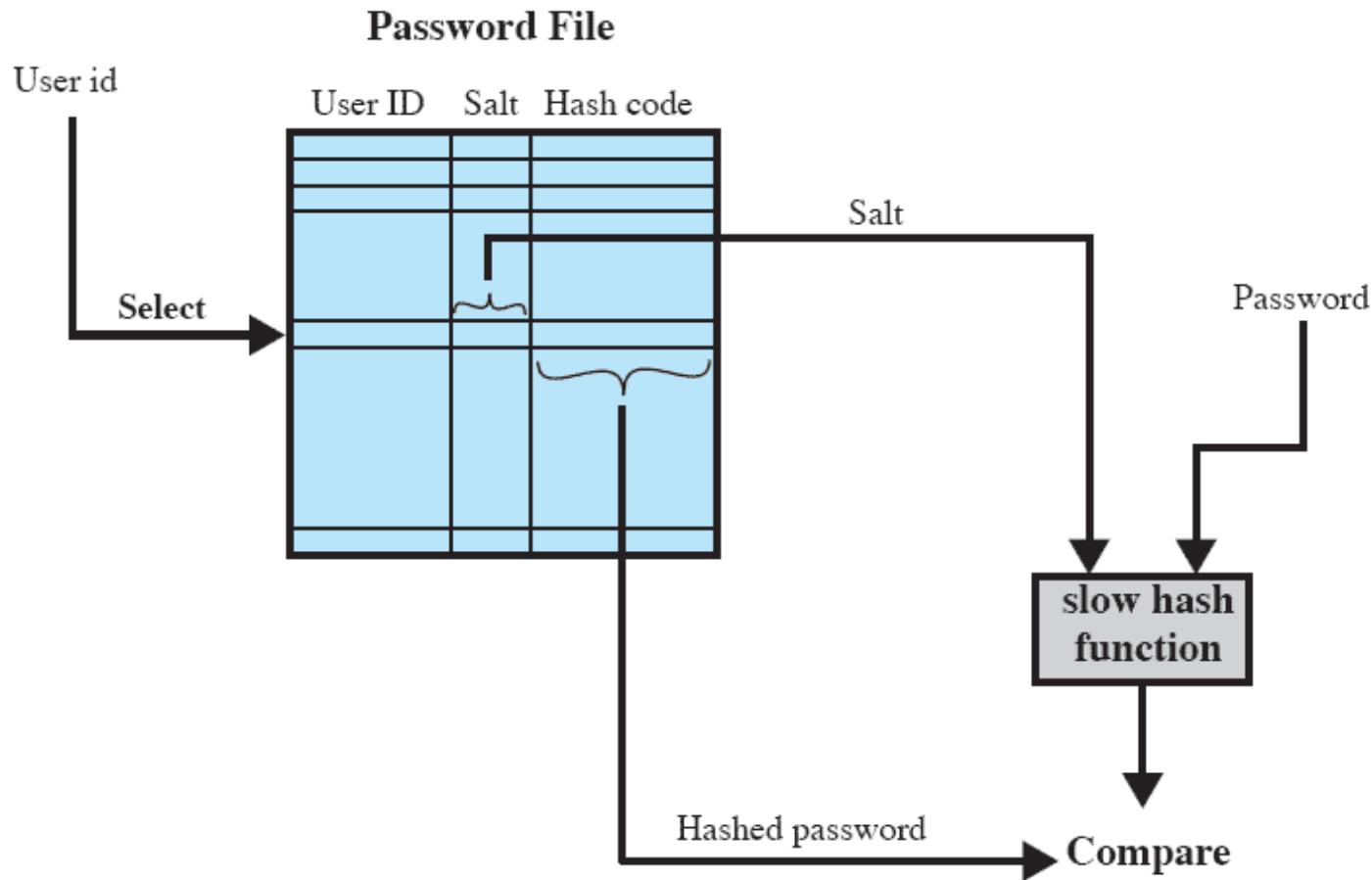
Hashed passwords/salt value

360



(a) Loading a new password

UNIX password scheme



(b) Verifying a password

- Serves three purposes:
 1. prevents duplicate passwords from being visible in the password file
 - even if two users choose the same password, the passwords will be assigned different salt values
 2. greatly increases the difficulty of offline dictionary attacks
 3. it becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them

Token-Based Authentication

363

- Objects that a user possesses for the purpose of user authentication are called tokens

Two types of tokens are:

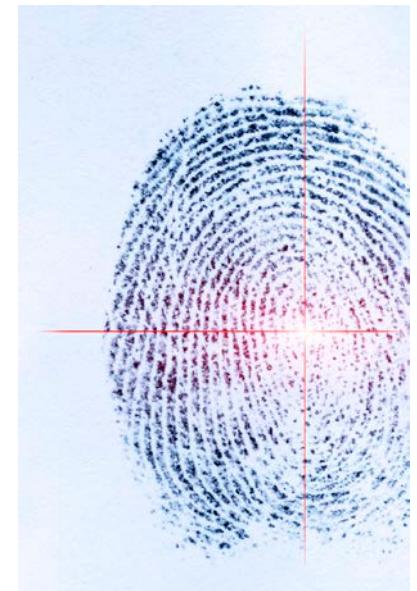
memory cards

smart cards

Static Biometric Authentication

364

- Attempts to authenticate an individual based on his or her unique physical characteristics
- Includes static characteristics such as:
 - fingerprints
 - hand geometry
 - facial characteristics
 - retinal and iris patterns
- Dynamic characteristics such as:
 - voiceprint
 - signature



Physical Characteristics

Facial characteristics

- most common means of human-to-human identification
- examples: eyes, eyebrows, nose, lips, and chin shape
- alternative approach is to use an infrared camera to produce a face thermogram

Fingerprints

- pattern of ridges and furrows on the surface of the fingertip
- unique across the entire human population

Hand geometry

- identify features of the hand, including shape, and lengths and widths of fingers

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

Retinal pattern

- the pattern formed by veins beneath the retinal surface is unique
- a retinal biometric system obtains a digital image of the retinal pattern by projecting a low-intensity beam of visual or infrared light into the eye

Iris

- the detailed structure of the iris is unique

Signature

- each individual has a unique style of handwriting

Voice

- voice patterns are closely tied to the physical and anatomical characteristics of the speaker

Access Control



366

- Dictates what types of access are permitted, under what circumstances, and by whom
- Access control is exercised by the OS, by the file system or at both levels
- Principles that have been typically applied are the same at the both levels

File System Access Control

367

- Identifies a user to the system
- Associated with each user there can be a profile that specifies permissible operations and file accesses
- The operating system can then enforce rules based on the user profile
- The database management system, however, must control access to specific records or even portions of records
- The database management system decision for access depends not only on the user's identity but also on the specific parts of the data being accessed and even on the information already divulged to the user

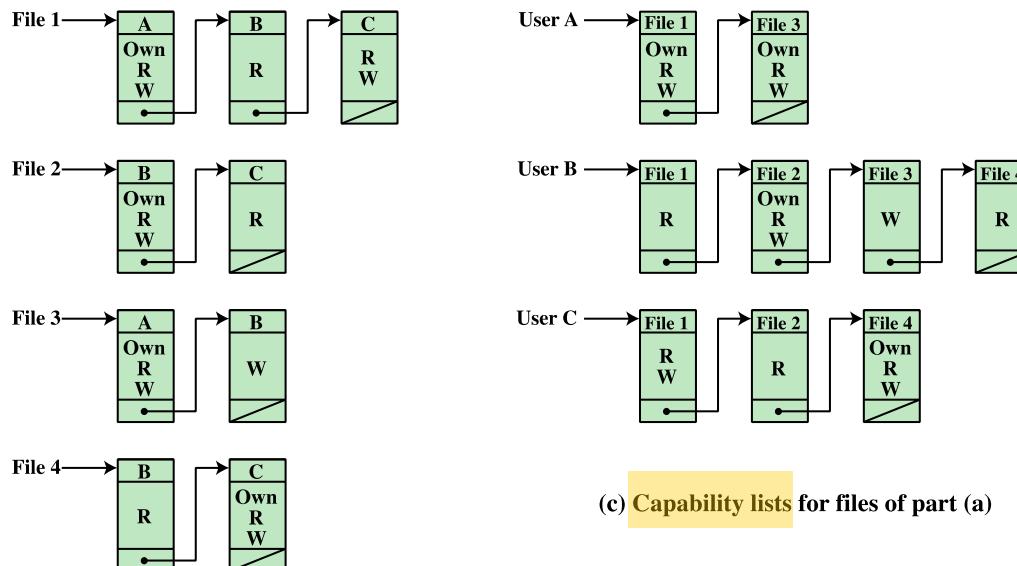
Access Matrix

Object

	File 1	File 2	File 3	File 4	Account 1	Account 2
User A	Own R W		Own R W		Inquiry Credit	
User B	R	Own R W	W	R	Inquiry Debit	Inquiry Credit
User C	R W	R		Own R W		Inquiry Debit

Access right

(a) Access matrix



(b) Access control lists for files of part (a)

(c) Capability lists for files of part (a)

Access Control Policies

369

- **Discretionary access control (DAC)**
 - controls access based on the identity of the requestor and on access rules stating what requestors are (or are not) allowed to do
- **Mandatory access control (MAC)**
 - controls access based on comparing security labels with security clearances
- **Role-based access control (RBAC)**
 - controls access based on the roles that users have within the system and on rules stating what accesses are allowed to users in given roles

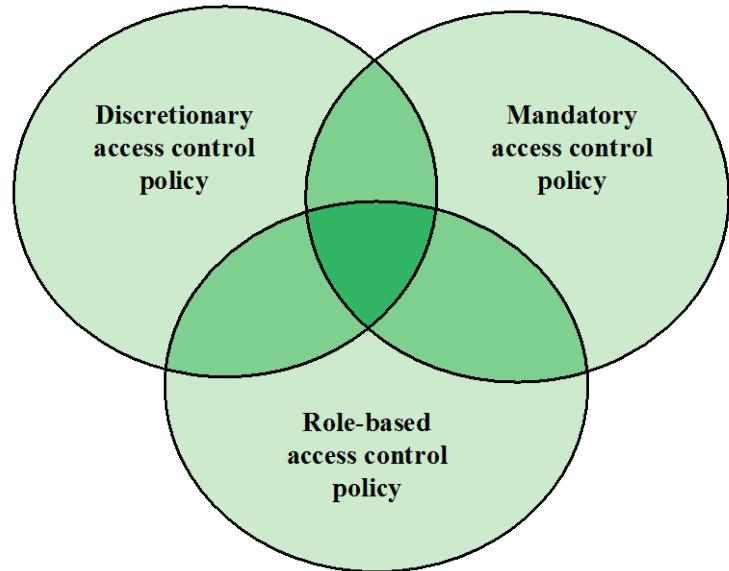


Figure 15.4 Access Control Policies

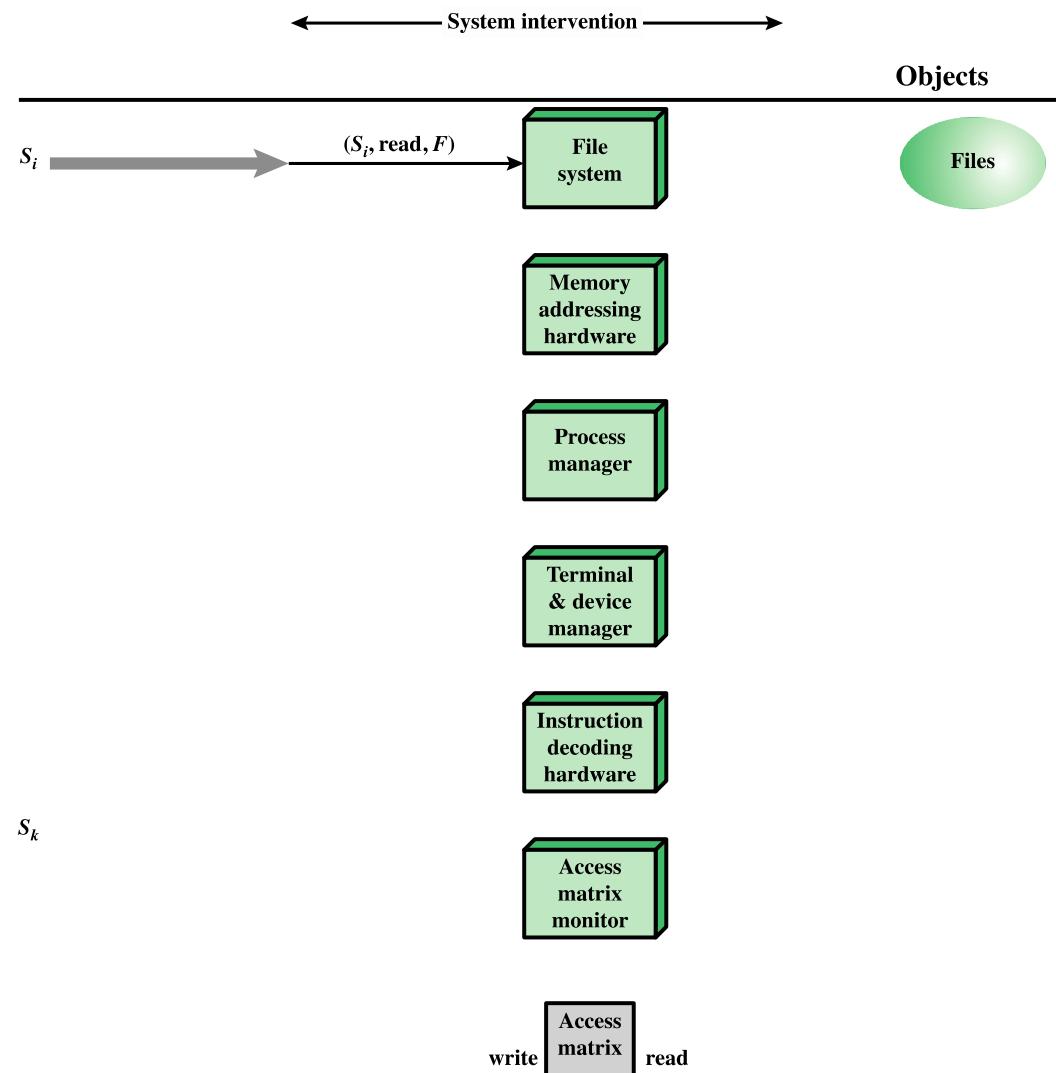
Extended Access Control Matrix for DAC

370

subjects			files		processes		disk drives		
SUBJECTS	S ₁	S ₂	S ₃	F ₁	F ₂	P ₁	P ₂	D ₁	D ₂
S ₁	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
S ₂		control		write *	execute			owner	seek *
S ₃			control		write	stop			

* - copy flag set

Figure 15.5 Extended Access Control Matrix



OBJECTS

SUBJECTS	subjects			files		processes		disk drives	
	S ₁	S ₂	S ₃	F ₁	F ₂	P ₁	P ₂	D ₁	D ₂
S ₁	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
S ₂		control		write *	execute			owner	seek *
S ₃			control		write	stop			

Rule	Command (by S _o)	Authorization	Operation
R1	transfer $\left\{ \begin{array}{l} \alpha^* \\ \alpha \end{array} \right\}$ to S, X	' α^* ' in A[S _o , X]	store $\left\{ \begin{array}{l} \alpha^* \\ \alpha \end{array} \right\}$ in A[S, X]
R2	grant $\left\{ \begin{array}{l} \alpha^* \\ \alpha \end{array} \right\}$ to S, X	'owner' in A[S _o , X]	store $\left\{ \begin{array}{l} \alpha^* \\ \alpha \end{array} \right\}$ in A[S, X]
R3	delete α from S, X	'control' in A[S _o , S] or 'owner' in A[S _o , X]	delete α from A[S, X]
R4	w \leftarrow read S, X	'control' in A[S _o , S] or 'owner' in A[S _o , X]	copy A[S, X] into w
R5	create object X	None	add column for X to A; store 'owner' in A[S _o , X]
R6	destroy object X	'owner' in A[S _o , X]	delete column for X from A
R7	create subject S	none	add row for S to A; execute create object S; store 'control' in A[S, S]
R8	destroy subject S	'owner' in A[S _o , S]	delete row for S from A; execute destroy object S

Role-based Access Control (RBAC)

373

- Based on the roles that user assume in a system rather than the user's identity
- Defines a role as a job function with an organization
- Access rights are assigned to roles and users are assigned to different roles (statically or dynamically)
- Relationship of users to roles (and roles to resources) is many to many

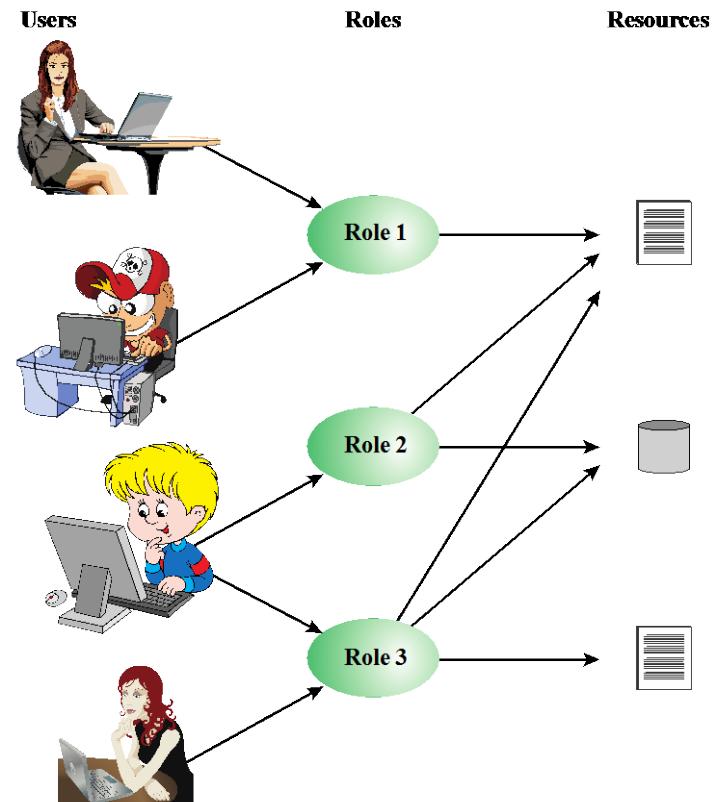


Figure 15.7 Users, Roles, and Resources

Access Matrices for RBAC

374

	R ₁	R ₂	• • •	R _n
U ₁	X			
U ₂	X			
U ₃		X		X
U ₄				X
U ₅				X
U ₆				X
•				
•				
•				
U _m	X			

Principle of least privilege

		OBJECTS								
		R ₁	R ₂	R _n	F ₁	F ₂	P ₁	P ₂	D ₁	D ₂
ROLES	R ₁	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
	R ₂		control		write *	execute			owner	seek *
	•									
	•									
	R _n			control		write	stop			

Figure 15.8 Access Control Matrix Representation of RBAC

29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

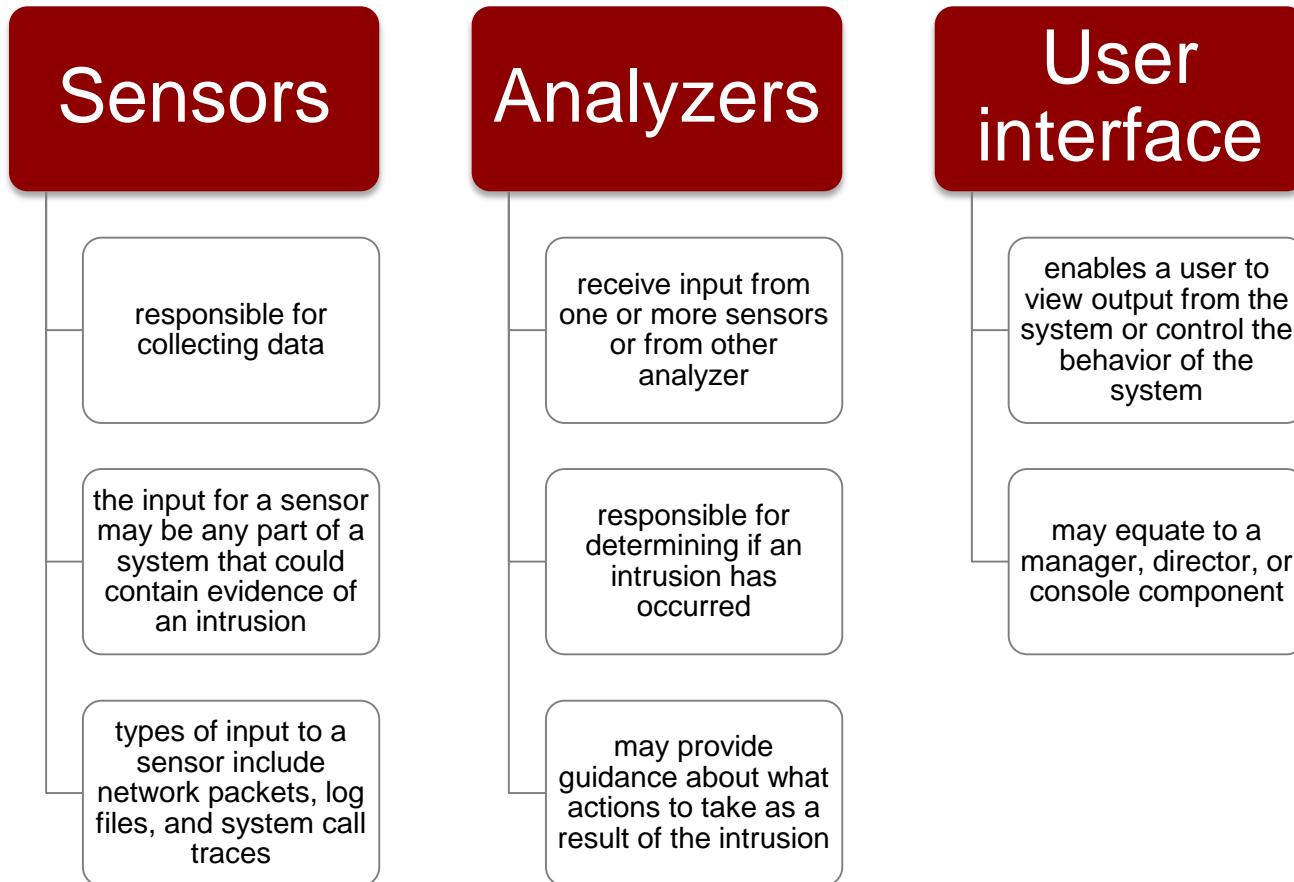
Intrusion Detection

375

- RFC 4949 (*Internet Security Glossary*) defines intrusion detection as a security service that monitors and analyzes system events for the purpose of finding, and providing real-time or near real-time warning of, attempts to access system resources in an unauthorized manner
- Intrusion detection systems (IDSs) can be classified as:
 - **host-based IDS**
 - » monitors the characteristics of a single host and the events occurring within that host for suspicious activity
 - **network-based IDS**
 - » monitors network traffic for particular network segments or devices and analyzes network, transport, and application protocols to identify suspicious activity

IDS Components

376



Detection-specific audit records: Example

377

- **Fields:** Subject, Action, Object, Exception-condition, Resource-usage, Time-stamp
- Suppose Smith issue the command

COPY GAME.EXE TO <Library>GAME.EXE

The following audit records may be generated

Smith	execute	<Library>COPY.EXE	0	CPU = 00002	11058721678
Smith	Read	<Smith>GAME.EXE	0	RECORDS = 0	11058721679
Smith	execute	<Library>COPY.EXE	write-viol	RECORDS = 0	11058721680

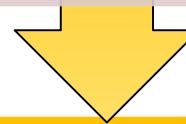
Antivirus Approaches

378

- Ideal solution to the threat of viruses is prevention, don't allow a virus onto the system in the first place!
- That goal is, in general, impossible to achieve, although prevention can reduce the number of successful viral attacks
- If detection succeeds but either identification or removal is not possible, then the alternative is to discard the infected program and reload a clean backup version

Detection

once the infection has occurred, determine that it has occurred and locate the virus



Identification

once detection has been achieved, identify the specific virus that has infected a program



Removal

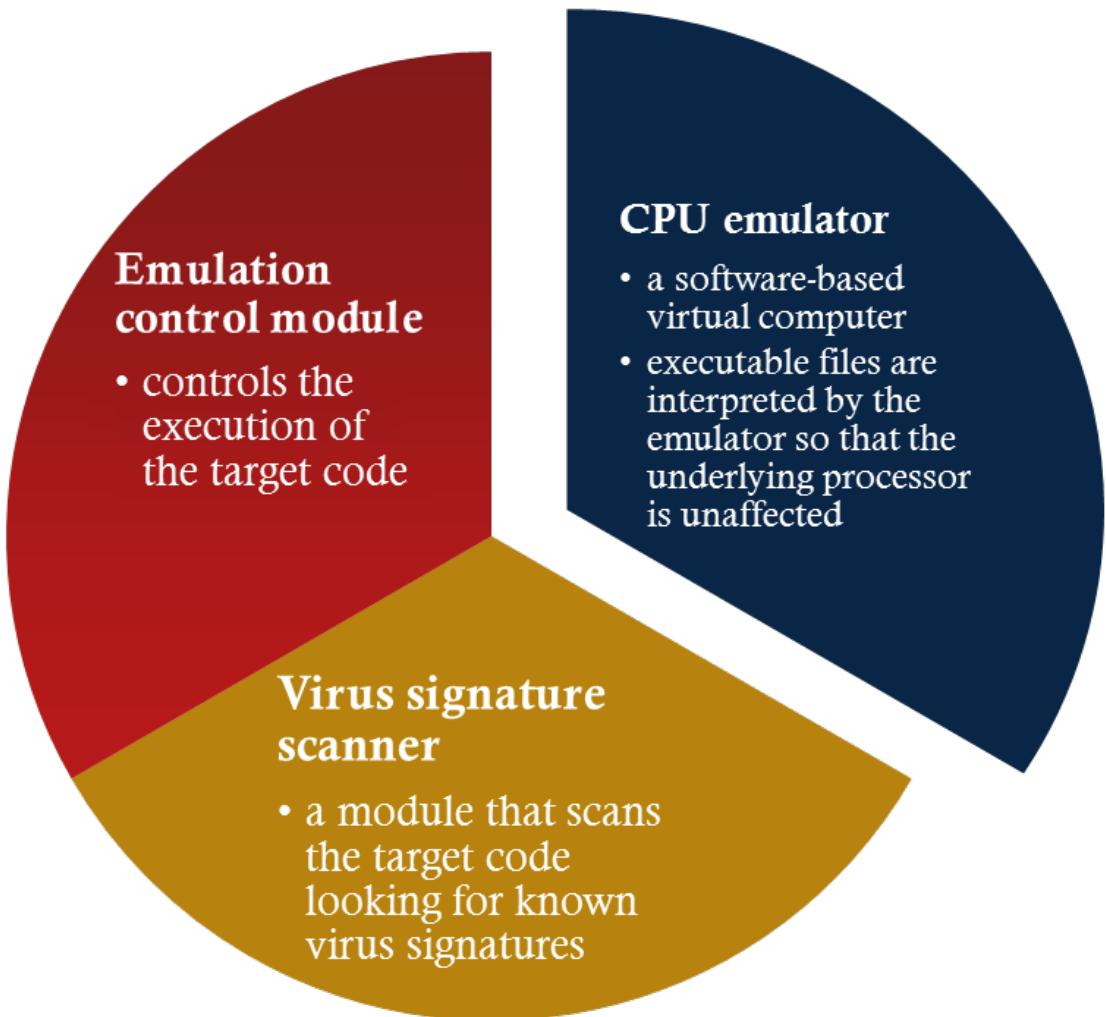
once the specific virus has been identified, remove all traces of the virus from the infected program and restore it to its original state

remove the virus from all infected systems so that the disease cannot spread further

GD Scanner

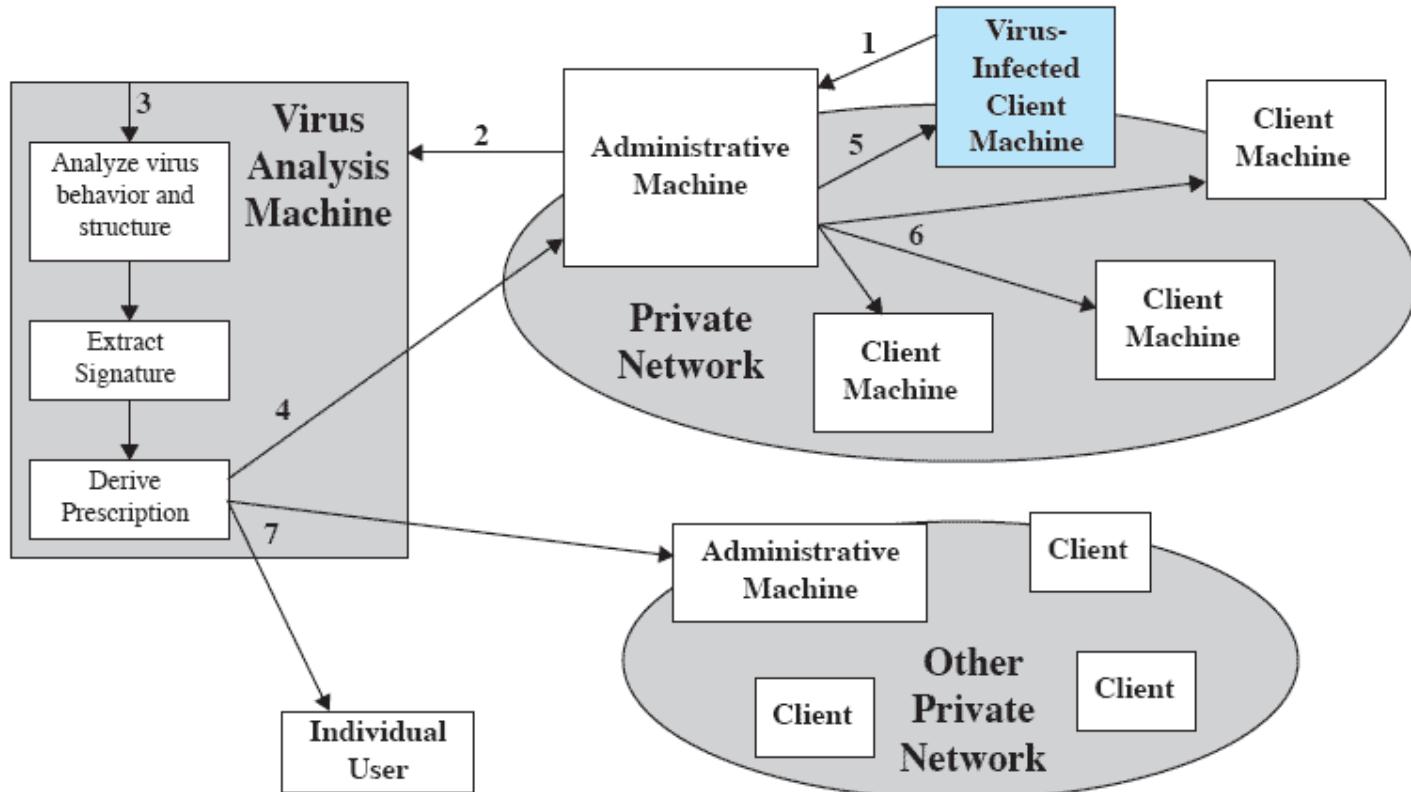
379

- GD scanner contains:
- Most difficult design issue with a GD scanner is to determine how long to run each interpretation



Digital Immune System

380



THANK YOU



29/10/2019

COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au