Final Exam Review

Fall 2017

Outline

- Chapter 0 --- Basic Linux Commands and C Programming
- Chapter 1 --- Operating System Overview
- Chapter 2 --- Linux and Process Overview
- Chapter 3 --- Processes concepts and implementation
- Chapter 4 --- Threads and System Calls
- Chapter 5 --- Concurrency
- Chapter 6 --- Deadlock
- Chapter 7 --- Memory Management
- Chapter 8 --- Virtual Memory and Virtual Page Allocation

Chapter 0

Basic Linux Commands and C Programming

Basic Linux Commands

- cat
- ls
- gcc
- make
- chmod
- scp
- tar
- ssh
- scp

Universality of I/O

`$ ./copy test test.old`	*Copy a regular file*
`$ ./copy a.txt /dev/tty`	*Copy a regular file to this terminal*
`$ ./copy /dev/tty b.txt`	*Copy input from this terminal to a regular file*
`$ ./copy /dev/pts/16 /dev/tty`	*Copy input from another terminal*

FILES in C

- #include <stdio.h>
- FILE *: pointer to a file object
  - The FILE object is a special data structure that keeps tract of the current file information
- Special files
  - **stdin**: Standard input
  - **stdout**: standard output
  - **stderr**: Standard error

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

FILES in C

- #include <stdio.h>
- FILE *: pointer to a file object
  - The FILE object is a special data structure that keeps tract of the current file information
- Special files
  - **stdin**: Standard input
  - **stdout**: standard output
  - **stderr**: Standard error

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

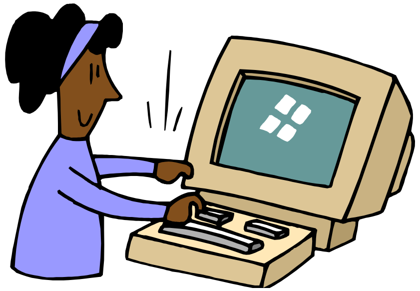
Chapter 1

Operating Systems Overview

Basic Elements

Processor

I/O Modules



Main Memory

System Bus

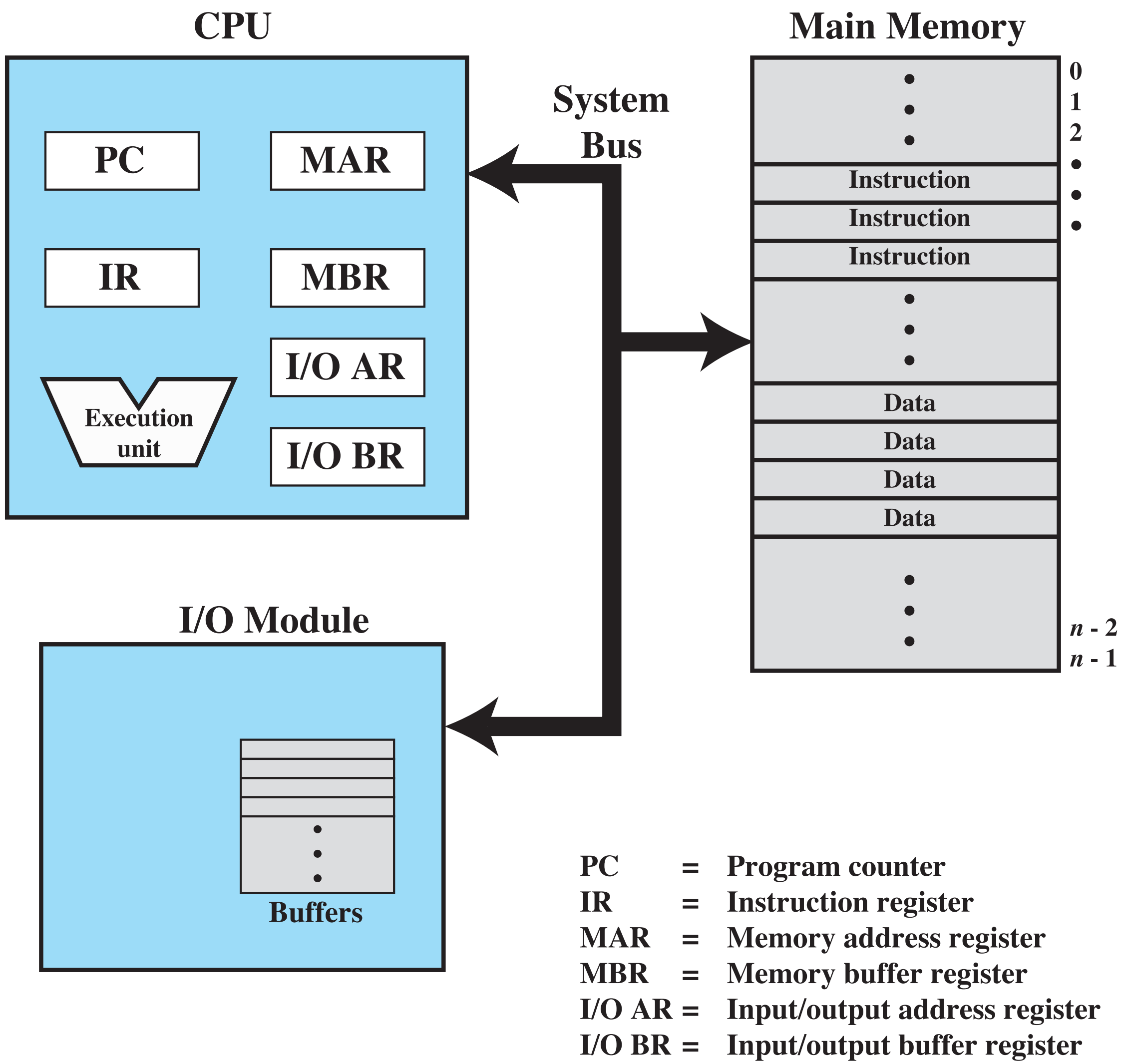


Figure 1.1 Computer Components: Top-Level View

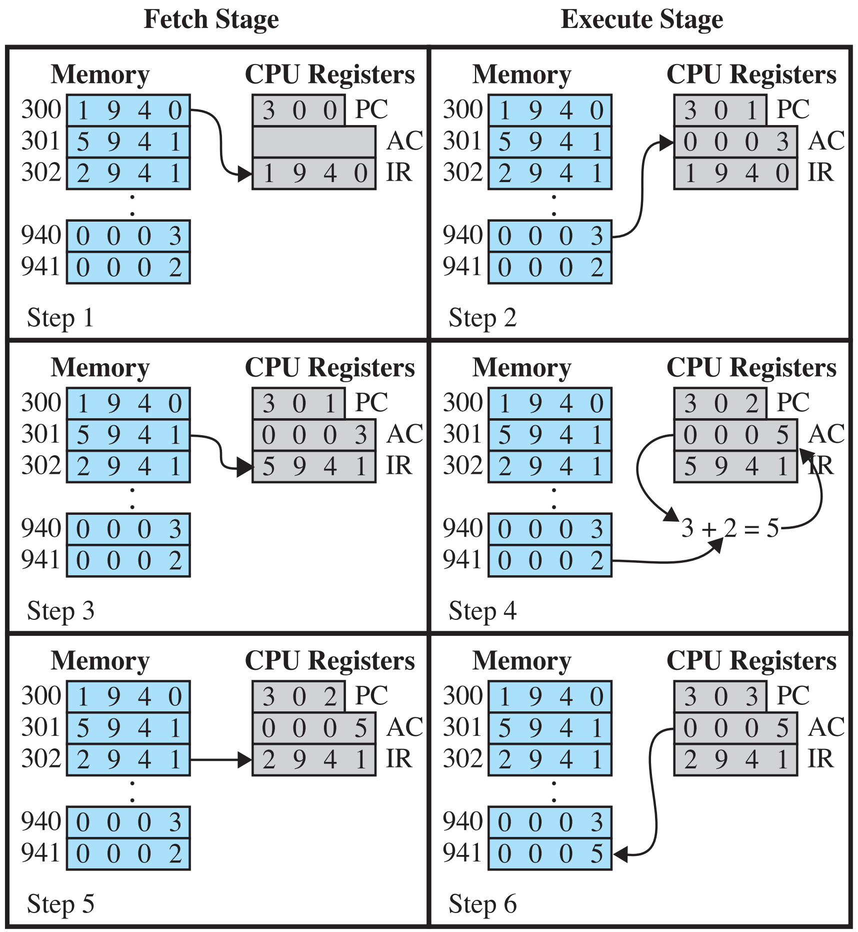


Figure 1.4 Example of Program Execution (contents of memory and registers in hexadecimal)

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

Table 1.1 Classes of Interrupts

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

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

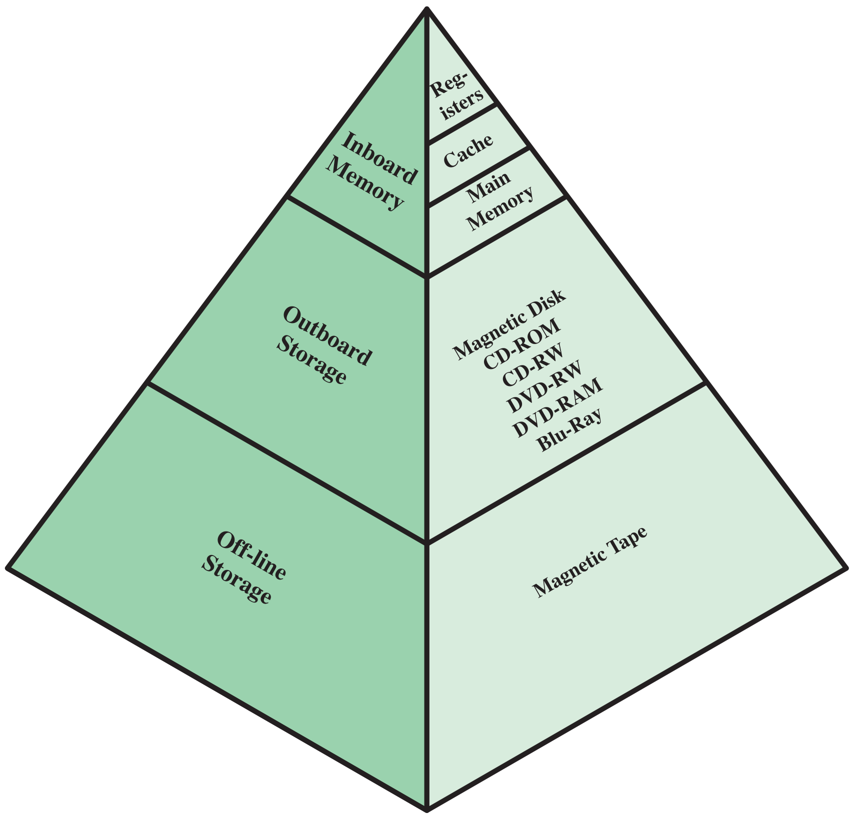


Figure 1.14 The Memory Hierarchy

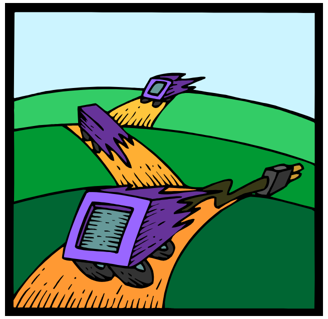
Principle of Locality

- 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

Chapter 2

Linux and Process Overview

Process



- Fundamental to the structure of operating systems

A *process* can be defined as:

- a program in execution
- an instance of a running program
- the entity that can be assigned to, and executed on, a processor
- a unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

LINUX Overview

- Started out as a UNIX variant for the IBM PC
- Linus Torvalds, a Finnish student of computer science, wrote the initial version
- Linux was first posted on the Internet in 1991
- Today it is a full-featured UNIX system that runs on several platforms
- Is free and the source code is available
- Key to success has been the availability of free software packages
- Highly modular and easily configured

Modular Monolithic Kernel

- Includes virtually all of the OS functionality in one large block of code that runs as a single process with a single address space
- All the functional components of the kernel have access to all of its internal data structures and routines
- Linux is structured as a collection of modules

Loadable Modules

- Relatively independent blocks
- A module is an object file whose code can be linked to and unlinked from the kernel at runtime
- A module is executed in kernel mode on behalf of the current process
- Have two important characteristics:
  - dynamic linking
  - stackable modules

Chapter 3

Process Concepts and Implementation

Process Elements

- Two essential elements of a process are:

Program code

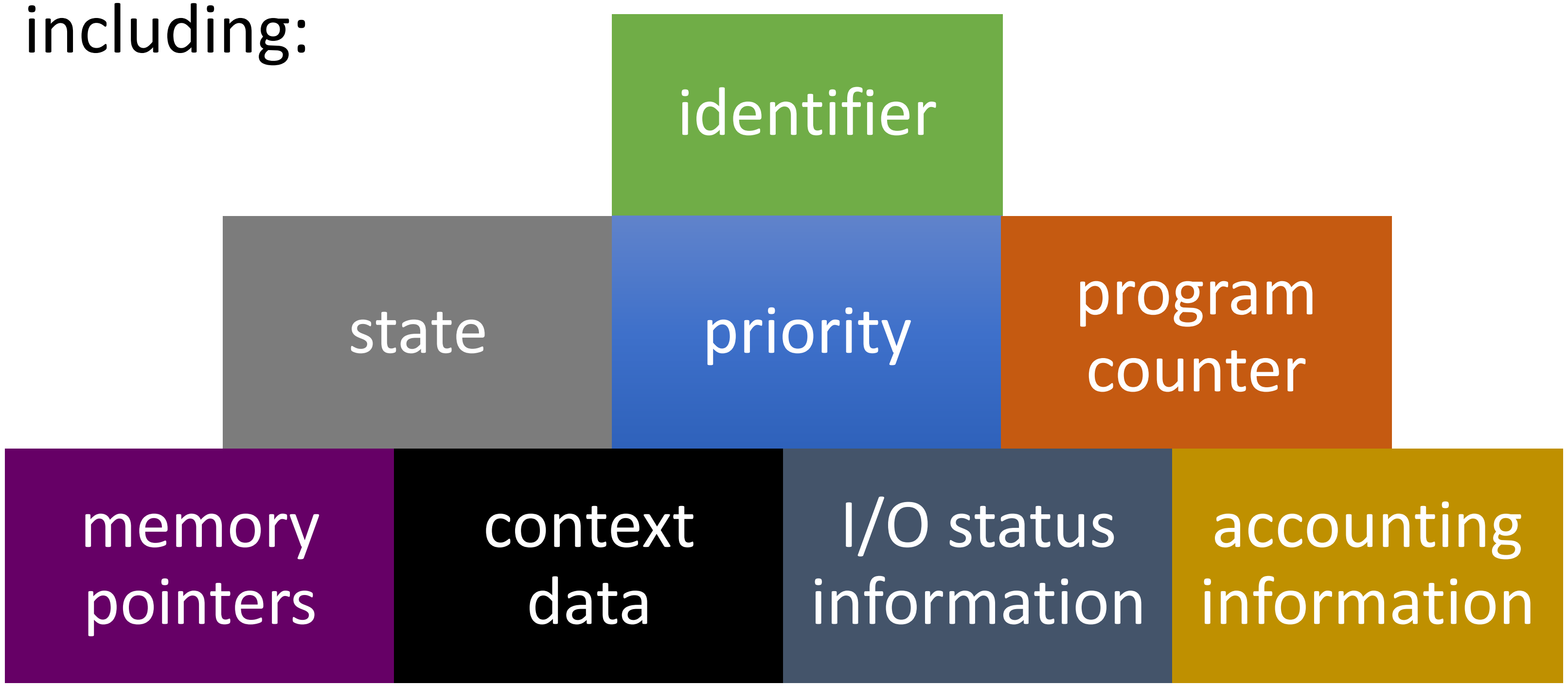
- which may be shared with other processes that are executing the same program

A set of data associated with that code

- when the processor begins to execute the program code, we refer to this executing entity as a ***process***

Process Elements

- While the program is executing, this process can be uniquely characterized by a number of elements, including:



Five-State Process Model

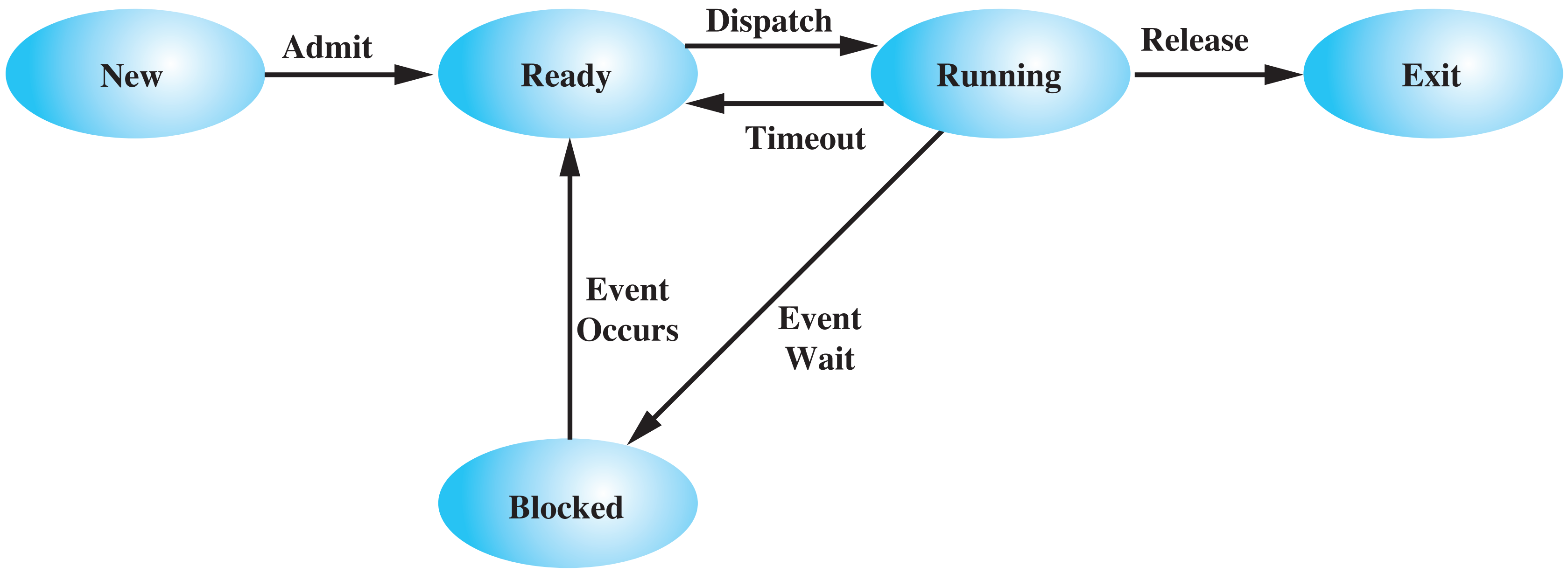


Figure 3.6 Five-State Process Model

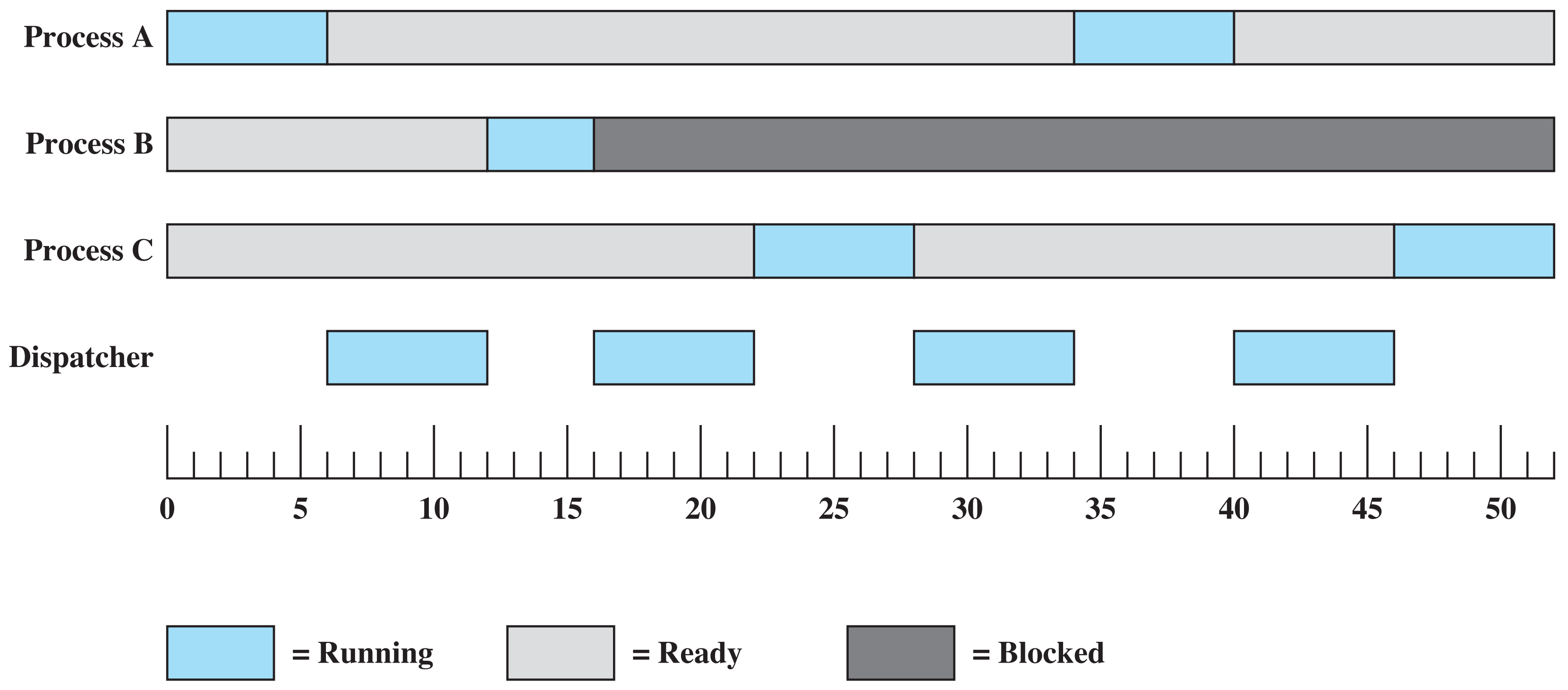
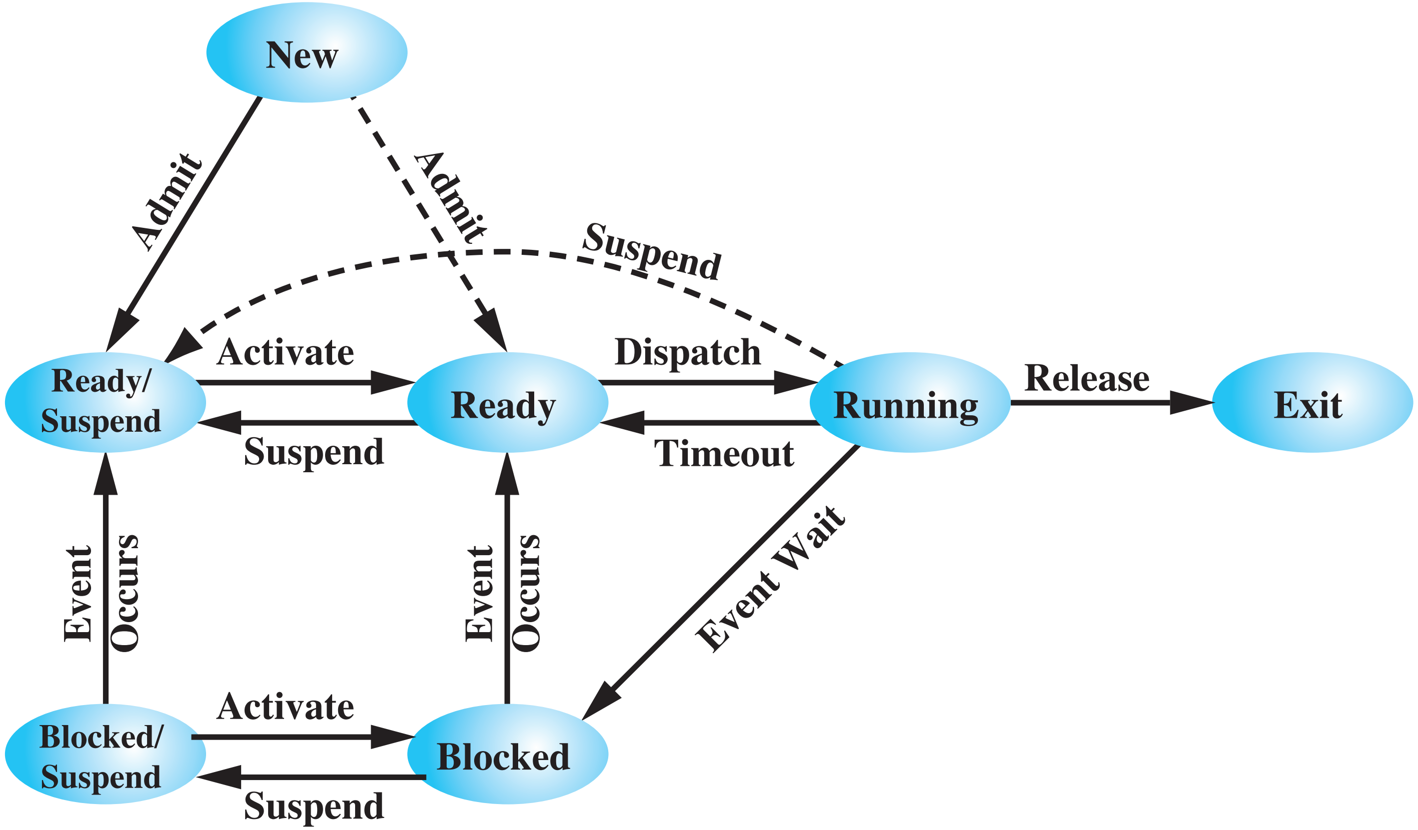


Figure 3.7 Process States for Trace of Figure 3.4



(b) With Two Suspend States

Figure 3.9 Process State Transition Diagram with Suspend States

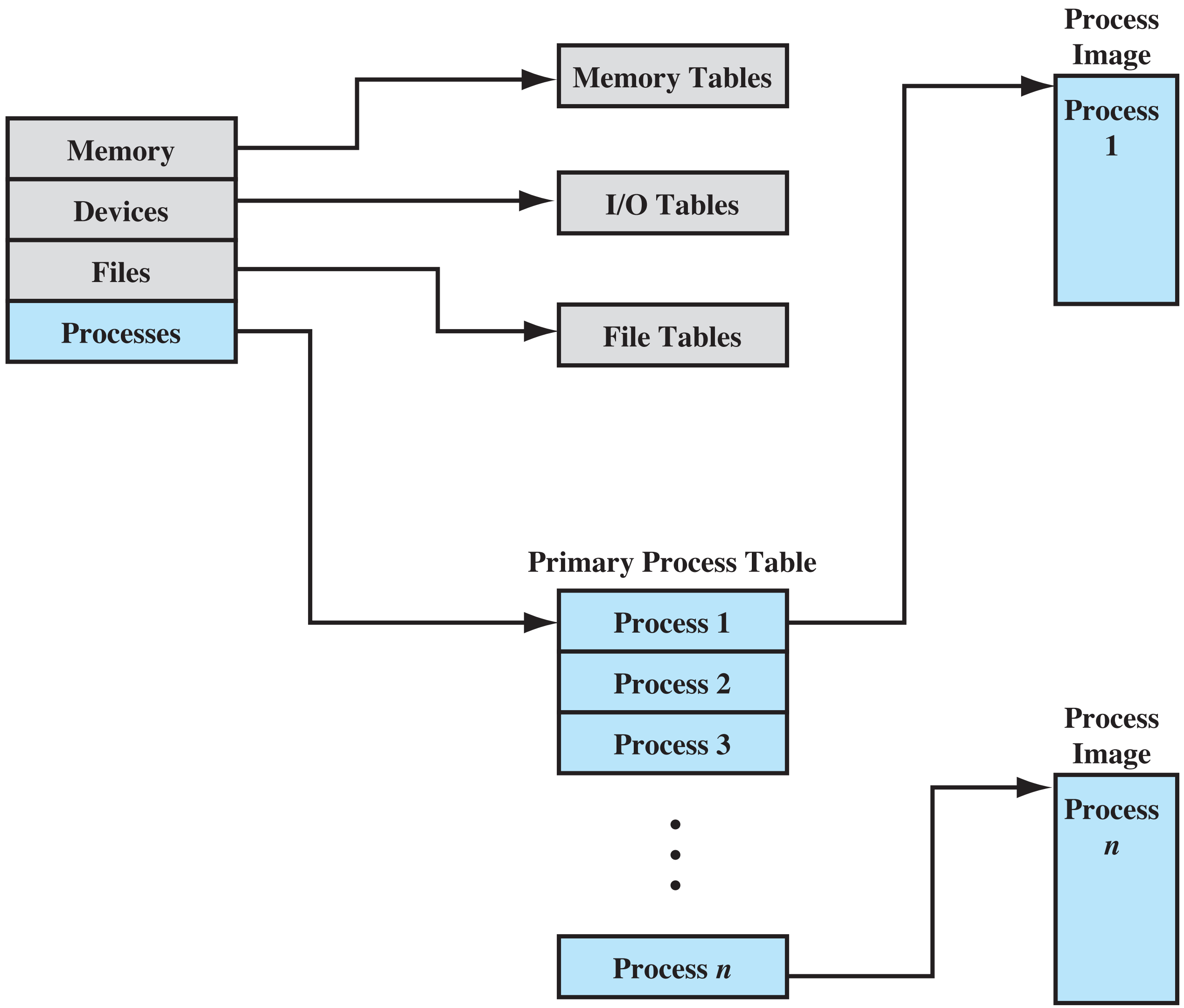


Figure 3.11 General Structure of Operating System Control Tables

Modes of Execution

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

Process Creation

- Once the OS decides to create a new process it:

1. assigns a unique process identifier to the new process
2. allocates space for the process
3. initializes the process control block
4. sets the appropriate linkages
5. creates or expands other data structures

System Interrupts

Interrupt

- Due to some sort of event that is external to and independent of the currently running process
  - clock interrupt
  - I/O interrupt
  - memory fault
- Time slice
  - the maximum amount of time that a process can execute before being interrupted

Trap

- An error or exception condition generated within the currently running process
- OS determines if the condition is fatal
  - moved to the Exit state and a process switch occurs
  - action will depend on the nature of the error

Change of Process State

- The steps in a full process switch are:

1. save the context of the processor
2. update the process control block of the process currently in the Running state
3. move the process control block of this process to the appropriate queue
4. select another process for execution
5. update the process control block of the process selected
6. update memory management data structures
7. restore the context of the processor to that which existed at the time the selected process was last switched out

If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment

Chapter 4

Threads and System Calls

Processes and Threads

Resource Ownership

Process includes a virtual address space to hold the process image

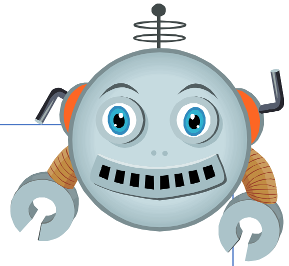
- the OS performs a protection function to prevent unwanted interference between processes with respect to resources

Scheduling/Execution

Follows an execution path that may be interleaved with other processes

- a process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the OS

One or More Threads in a Process



Each thread has:

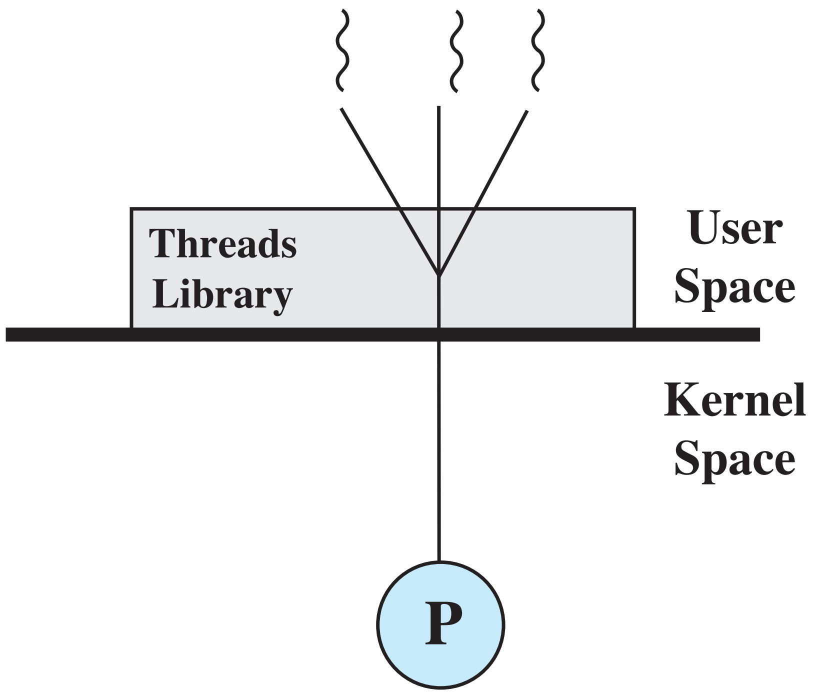
- an execution state (Running, Ready, etc.)
- saved thread context when not running
- an execution stack
- some per-thread static storage for local variables
- access to the memory and resources of its process (all threads of a process share this)

Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs

User-Level Threads (ULTs)

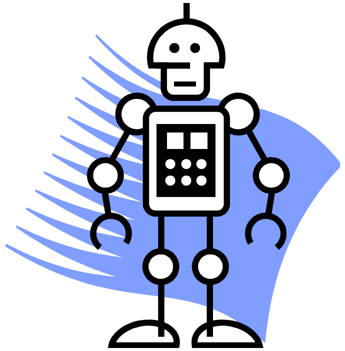
- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

Advantages of ULTs

- Thread switching does not require kernel mode privileges
- Scheduling can be application specific
- ULTs can run on any OS



Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing

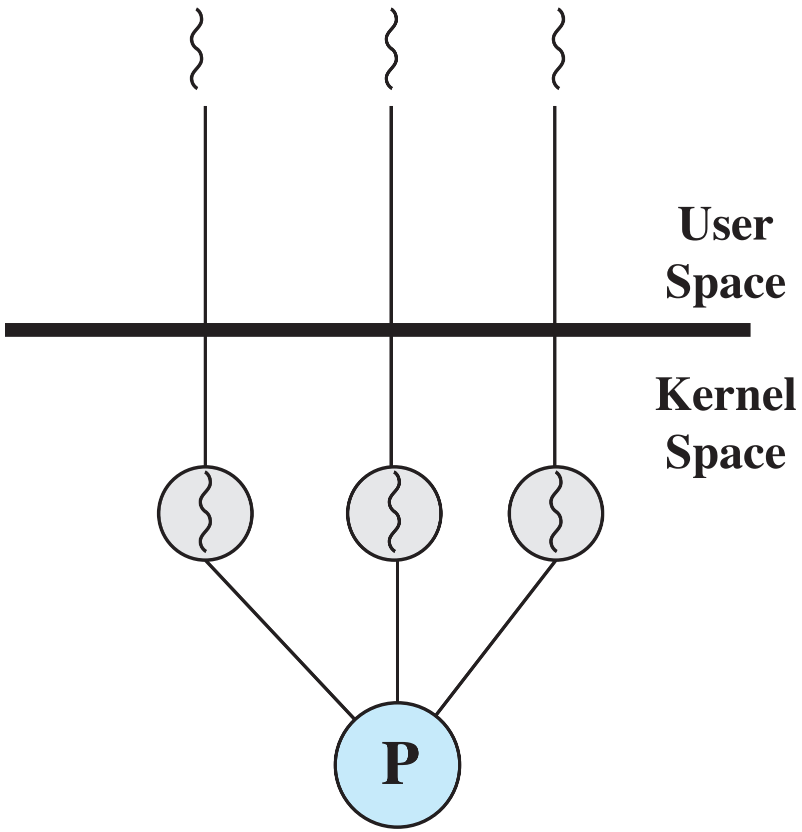
Overcoming ULT Disadvantages

Jacketing

- converts a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

Kernel-Level Threads (KLTs)

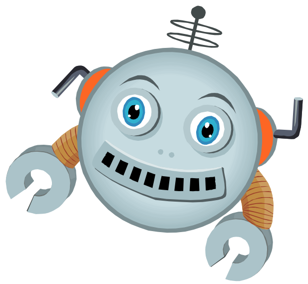


(b) Pure kernel-level

- Thread management is done by the kernel
  - no thread management is done by the application
    - Windows is an example of this approach

Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded



Disadvantage of KLTs

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

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies (μs)

Applications That Benefit

- Multithreaded native applications
  - characterized by having a small number of highly threaded processes
- Multiprocess applications
  - characterized by the presence of many single-threaded processes
- Java applications
- Multiinstance applications
  - multiple instances of the application in parallel

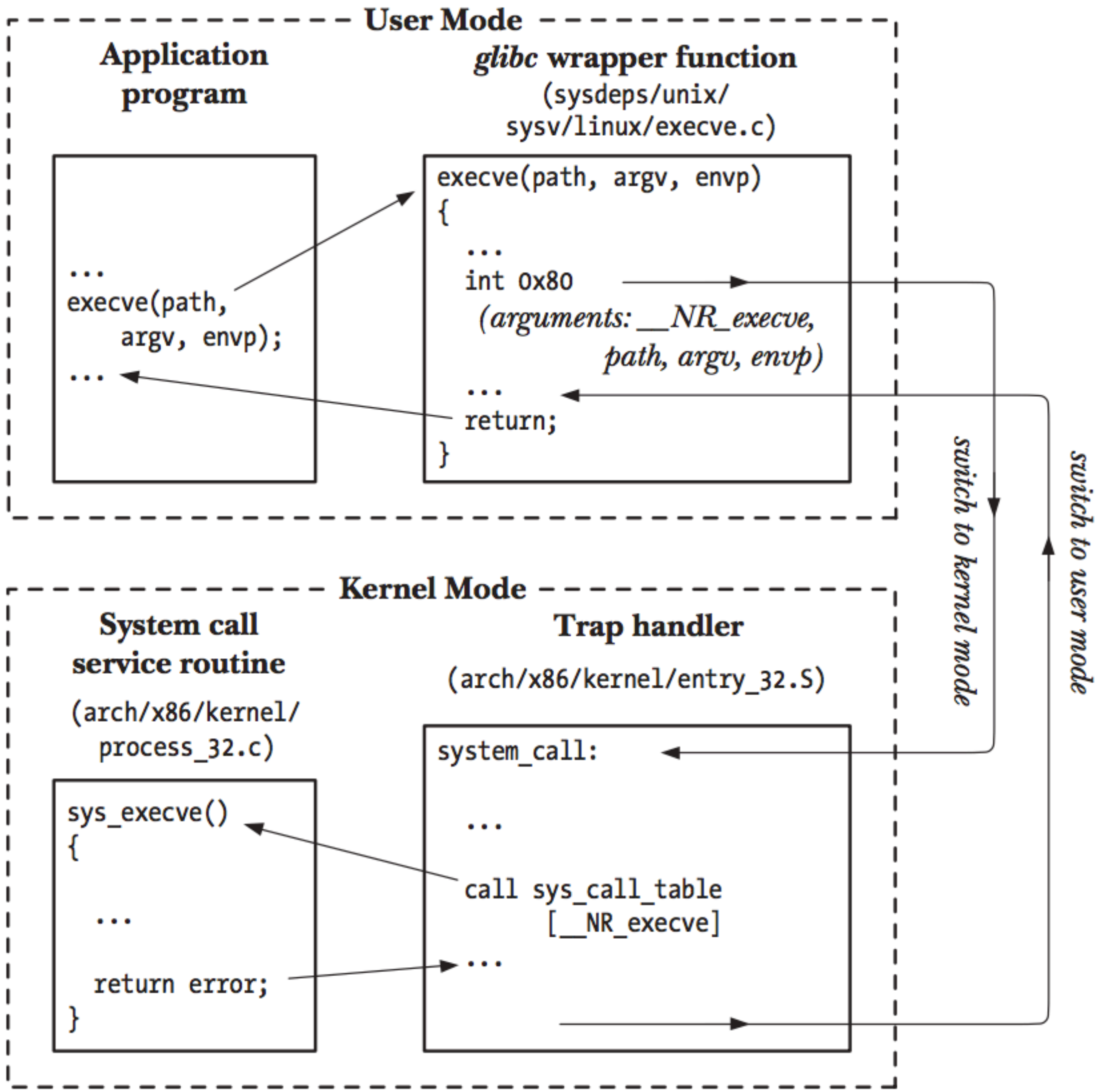


Figure 3-1: Steps in the execution of a system call

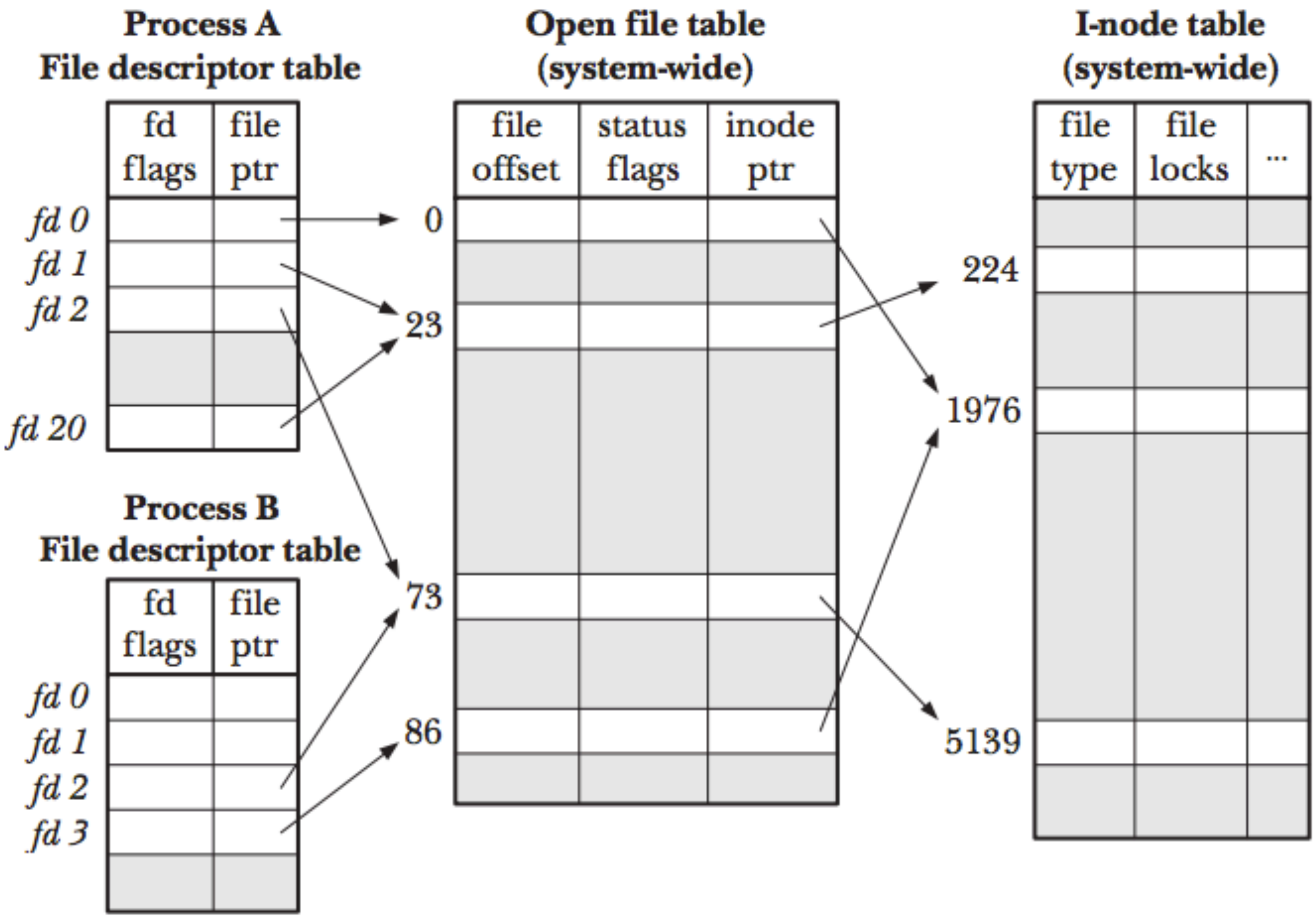


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

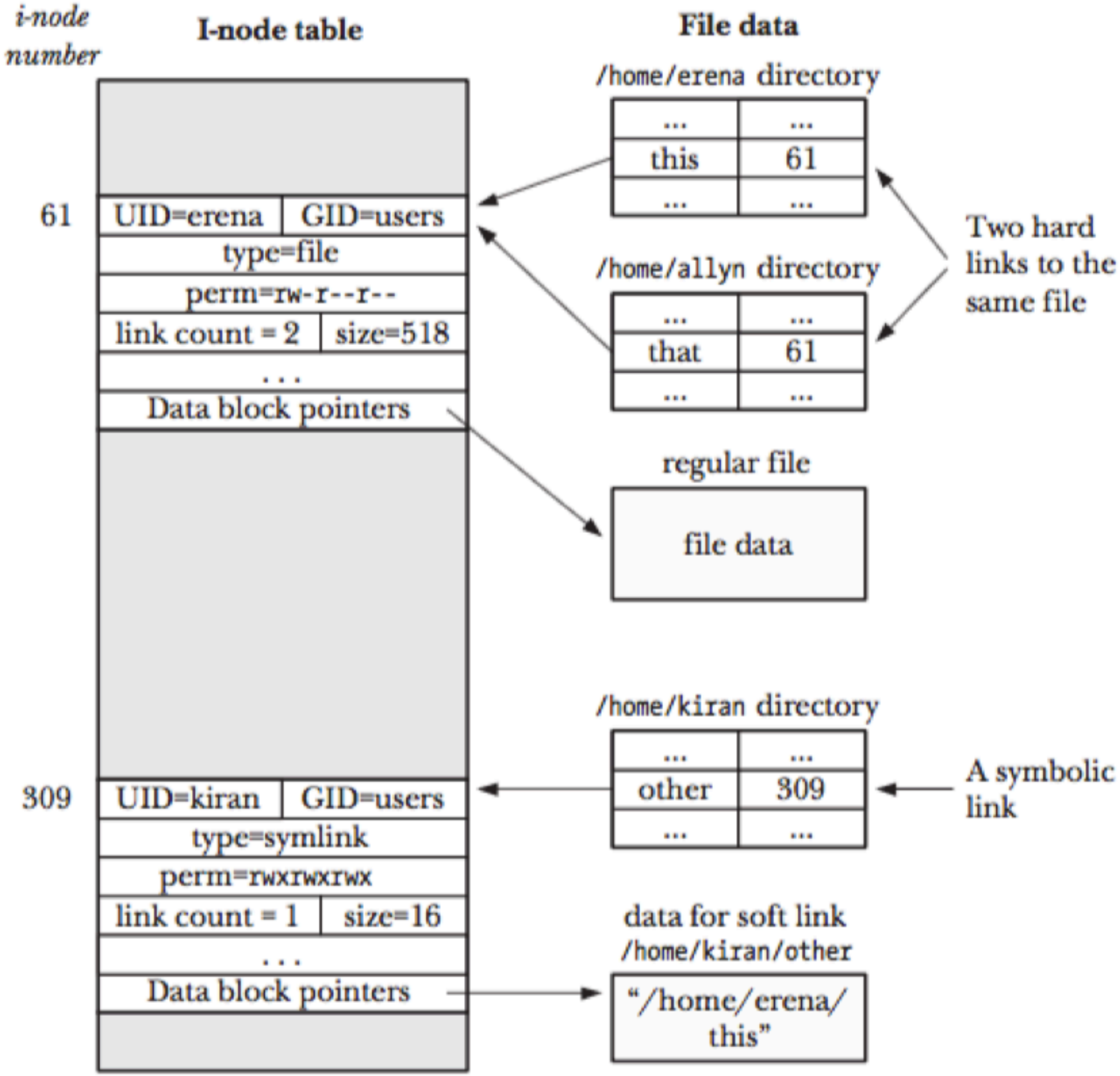


Figure 18-2: Representation of hard and symbolic links

fork

```
#include <unistd.h>

pid_t fork(void);
```

In parent: returns process ID of child on success, or −1 on error;
in successfully created child: always returns 0

```
pid_t childPid;              /* Used in parent after successful fork()
                                to record PID of child */
switch (childPid = fork()) {
case -1:                     /* fork() failed */
    /* Handle error */

case 0:                      /* Child of successful fork() comes here */
    /* Perform actions specific to child */

default:                     /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```

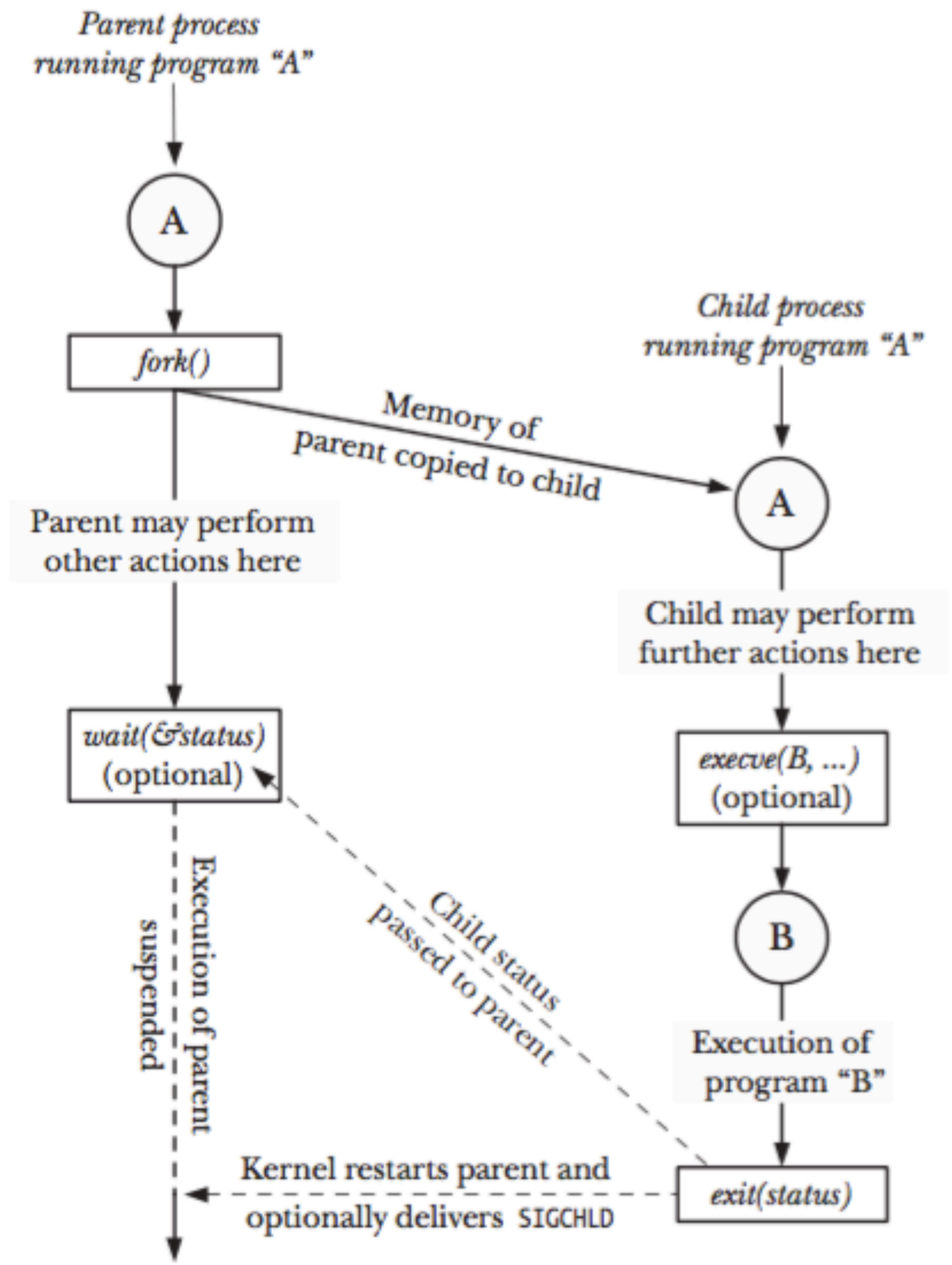


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

Copy on write memory

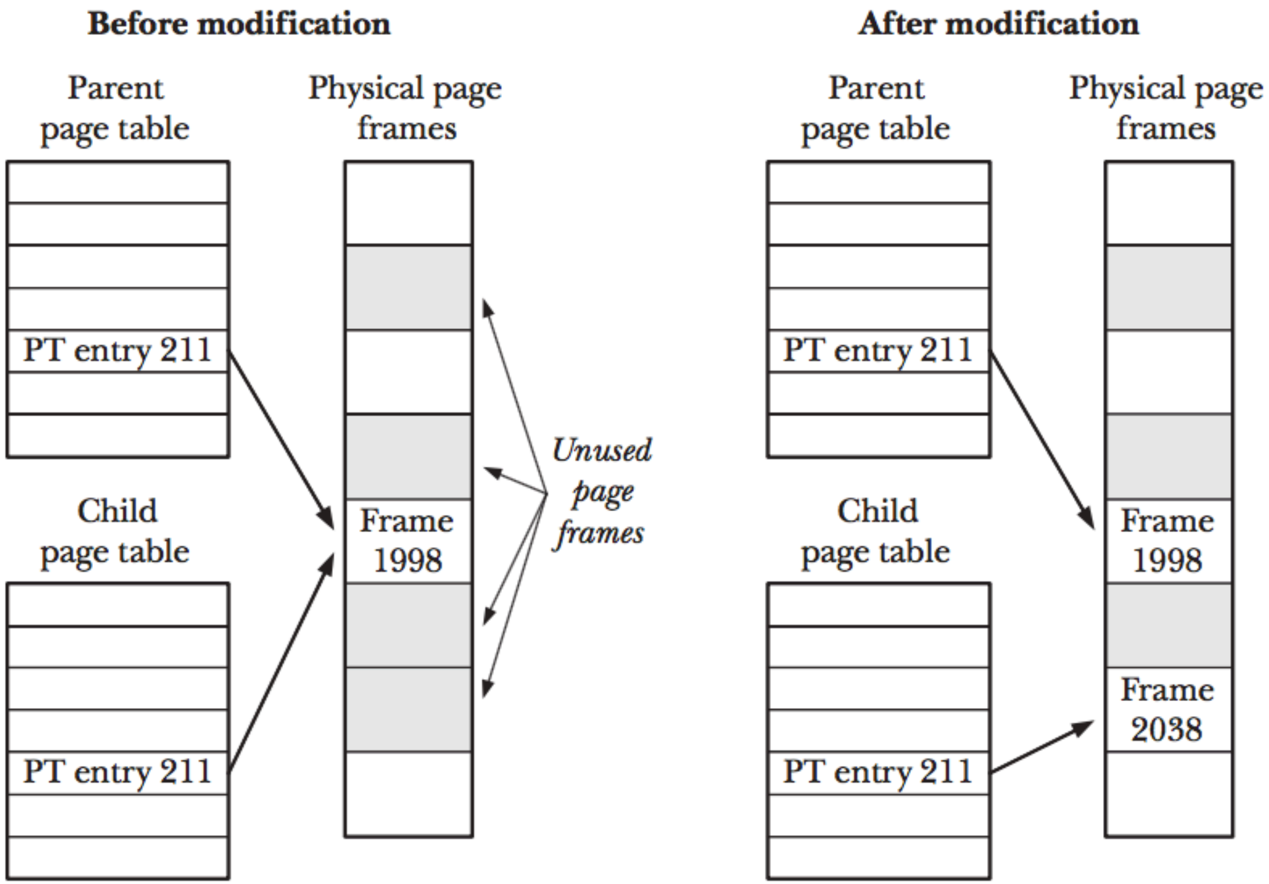


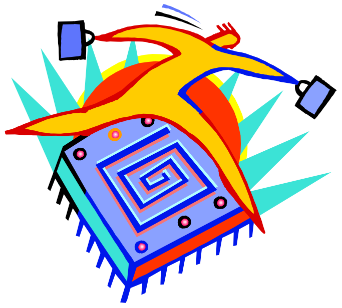
Figure 24-3: Page tables before and after modification of a shared copy-on-write page

Chapter 5

Concurrency

Principles of Concurrency

- Interleaving and overlapping
  - can be viewed as examples of concurrent processing
  - both present the same problems
- Uniprocessor – the relative speed of execution of processes cannot be predicted
  - depends on activities of other processes
  - the way the OS handles interrupts
  - scheduling policies of the OS



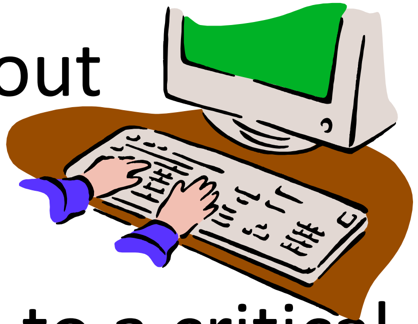
Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

Race Condition

- 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

Requirements for Mutual Exclusion



- Must be enforced
- A process that halts must do so without interfering with other processes
- No deadlock or starvation
- A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Mutual Exclusion: Hardware Support

- **Interrupt Disabling**
  - uniprocessor system
  - disabling interrupts guarantees mutual exclusion

- **Disadvantages:**
  - the efficiency of execution could be noticeably degraded
  - this approach will not work in a multiprocessor architecture

Mutual Exclusion: Hardware Support

- 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

Semaphore

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

1) May be initialized to a nonnegative integer value
2) The semWait operation decrements the value
3) The semSignal operation increments the value

Strong/Weak Semaphores

- A queue is used to hold processes waiting on the semaphore

Strong Semaphores

- the process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

- the order in which processes are removed from the queue is not specified

Monitor Characteristics

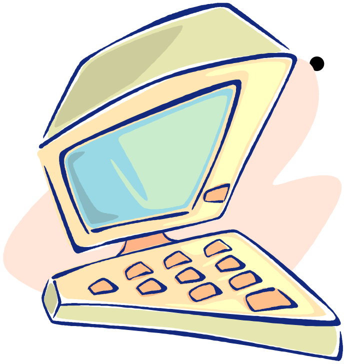
Local data variables are accessible only by the monitor’s procedures and not by any external procedure

Process enters monitor by invoking one of its procedures

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

Synchronization

- Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
  - Condition variables are operated on by two functions:
    - cwait(c): suspend execution of the calling process on condition c
    - csignal(c): resume execution of some process blocked after a cwait on the same condition



Message Passing

- When processes interact with one another two fundamental requirements must be satisfied:

synchronization	communication
• to enforce mutual exclusion	• to exchange information

- Message Passing is one approach to providing both of these functions
  - works with distributed systems *and* shared memory multiprocessor and uniprocessor systems

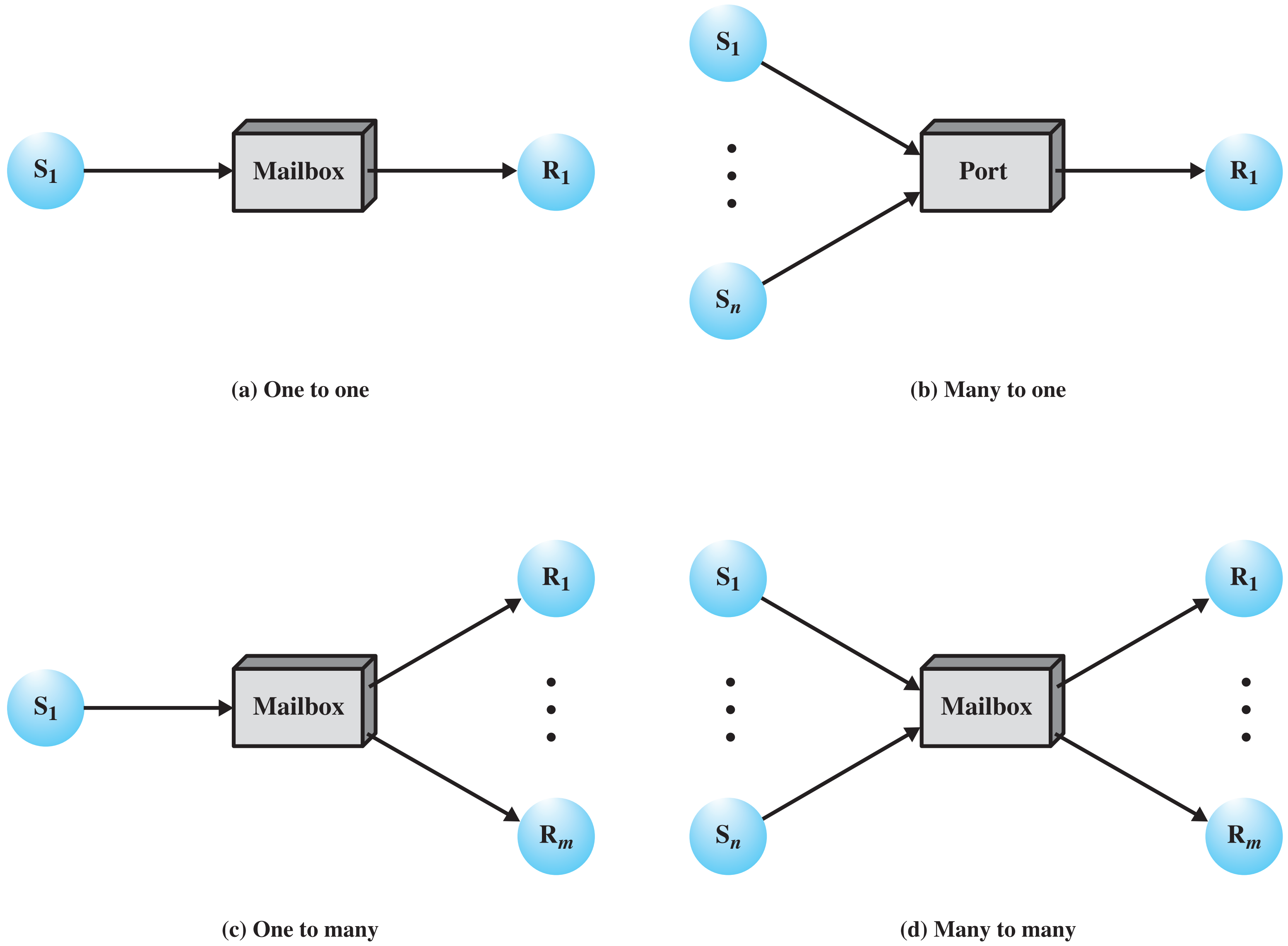


Figure 5.18 Indirect Process Communication

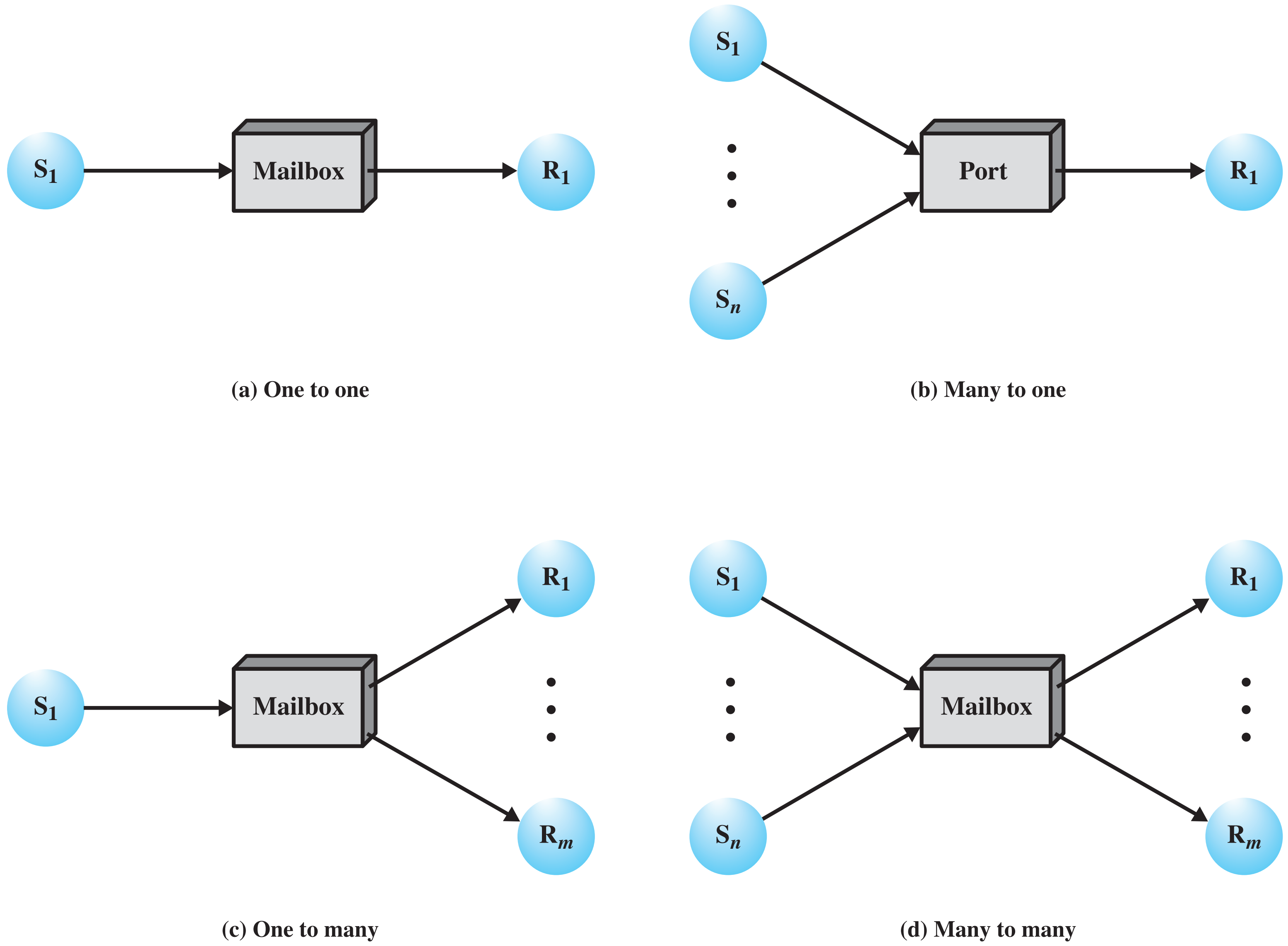


Figure 5.18 Indirect Process Communication

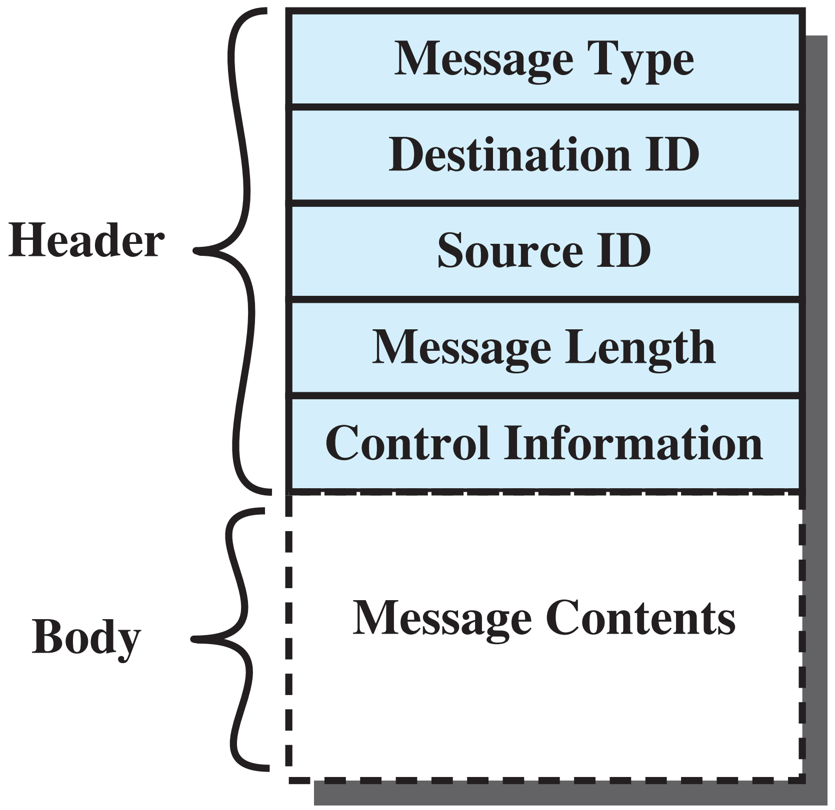


Figure 5.19 General Message Format

Chapter 6

Deadlock

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
- No efficient solution

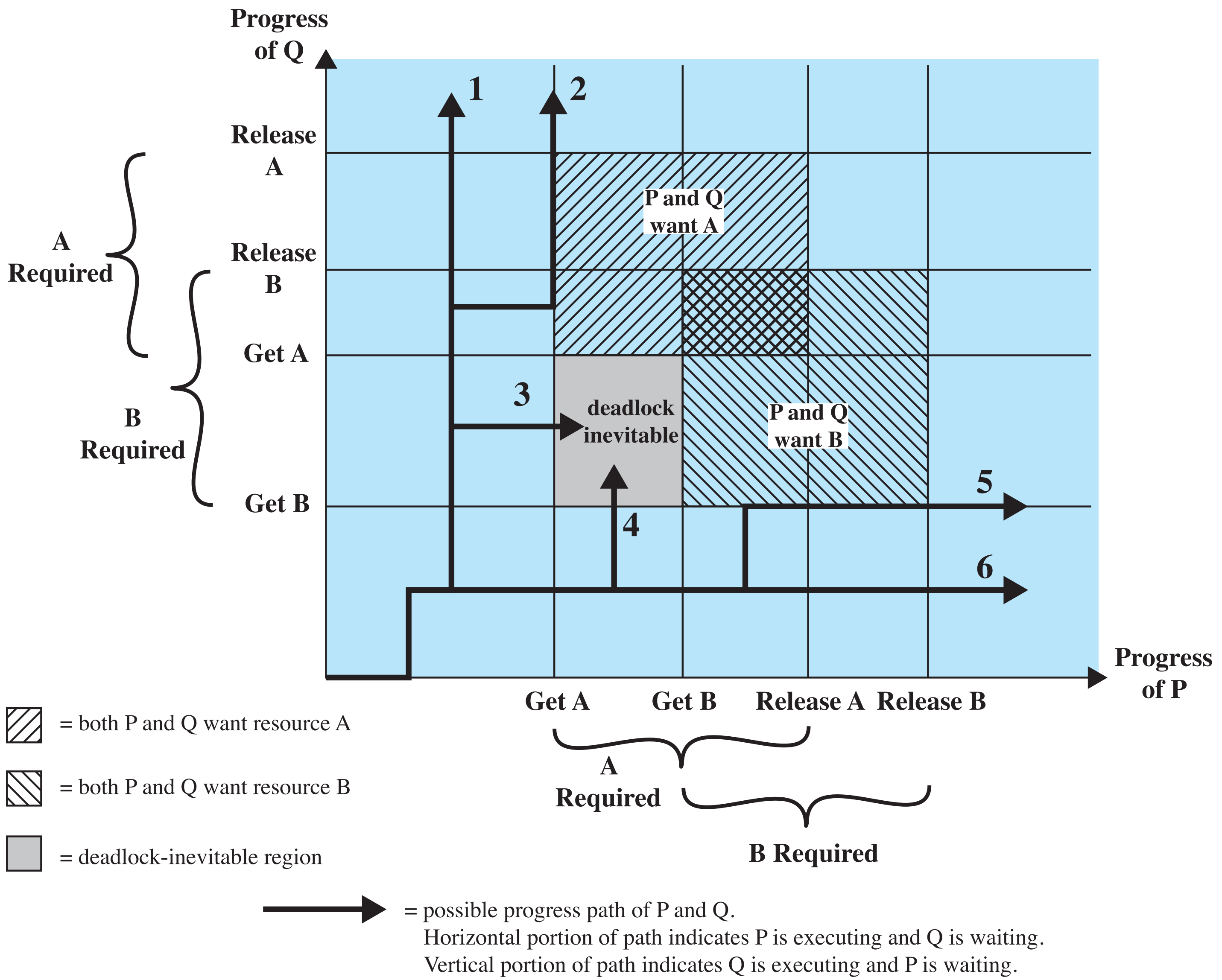


Figure 6.2 Example of Deadlock

Resource Categories

Reusable

- can be safely used by only one process at a time and is not depleted by that use
  - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- one that can be created (produced) and destroyed (consumed)
  - interrupts, signals, messages, and information
  - in I/O buffers

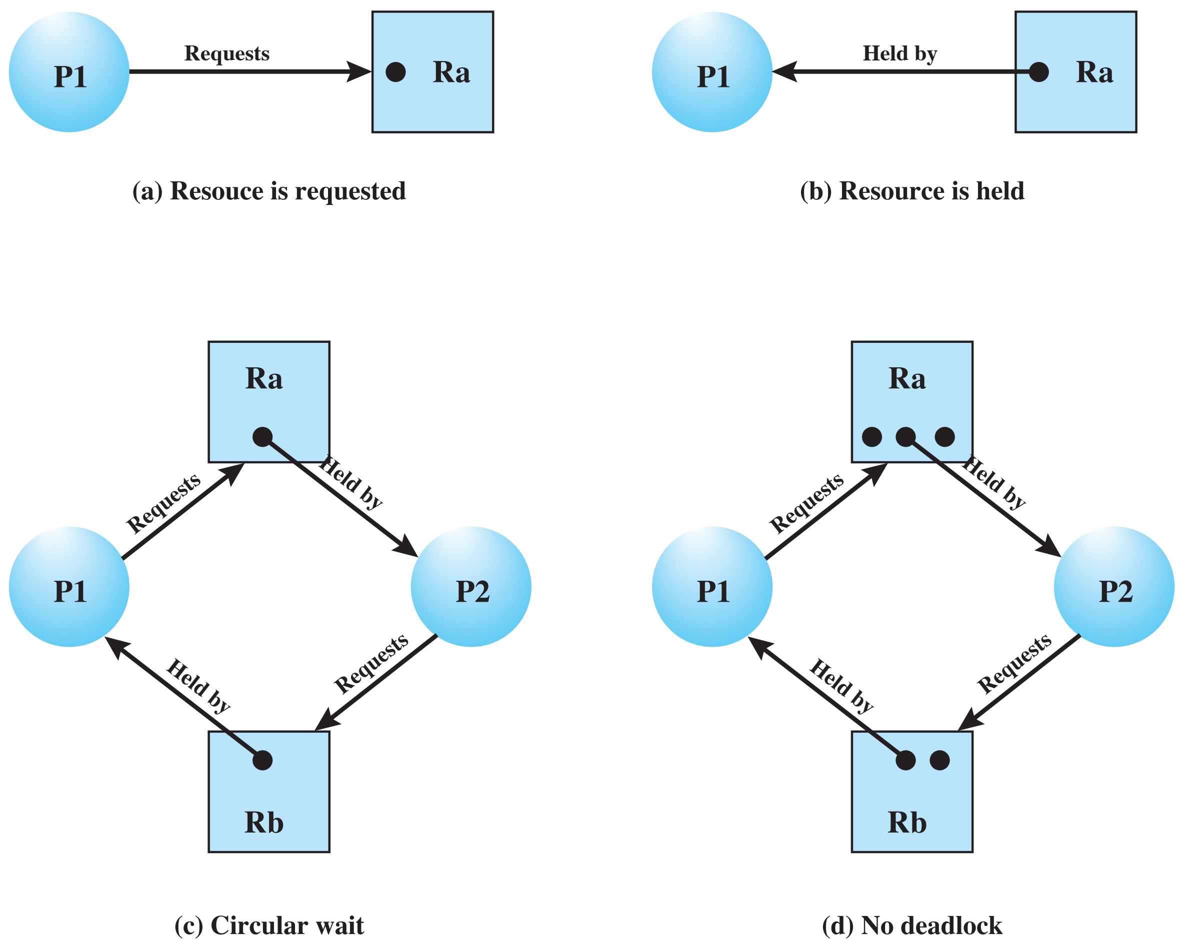


Figure 6.5 Examples of Resource Allocation Graphs

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it	• a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Banker's Algo

	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

Figure 6.7 Determination of a Safe State

Deadlock Strategies

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

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

Request matrix Q

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

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Figure 6.10 Example for Deadlock Detection

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Pipes

Messages

Shared memory

Semaphores

Signals

Chapter 7

Memory Management

Memory Management Requirements

- Memory management is intended to satisfy the following requirements:
    - Relocation
    - Protection
    - Sharing
    - Logical organization
    - Physical organization

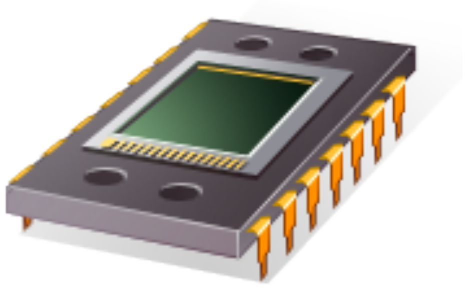


Table 7.2

Memory Management Techniques

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

(Table is on page 315 in textbook)

Placement Algorithms

Best-fit	First-fit	Next-fit
• chooses the block that is closest in size to the request	• begins to scan memory from the beginning and chooses the first available block that is large enough	• begins to scan memory from the location of the last placement and chooses the next available block that is large enough

Buddy System

- Comprised of fixed and dynamic partitioning schemes
- Space available for allocation is treated as a single block
- Memory blocks are available of size 2^K *words,* $L \le K \le U$, where
  - 2^L = smallest size block that is allocated
  - 2^U = largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation

Paging

- 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	Frames
• chunks of a process	• available chunks of memory

Page Table

- 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

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
- Eliminates internal fragmentation

Segmentation

- Usually visible
- Provided as a convenience for organizing programs and data
- Typically the programmer will assign programs and data to different segments
- For purposes of modular programming the program or data may be further broken down into multiple segments
  - the principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation

Address Translation

- Another consequence of unequal size segments is that there is no simple relationship between logical addresses and physical addresses
- The following steps are needed for address translation:

Extract the segment number as the leftmost *n* bits of the logical address

Use the segment number as an index into the process segment table to find the starting physical address of the segment

Compare the offset, expressed in the rightmost *m* bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid

The desired physical address is the sum of the starting physical address of the segment plus the offset

Address Translation

- Another consequence of unequal size segments is that there is no simple relationship between logical addresses and physical addresses
- The following steps are needed for address translation:

1. Extract the segment number as the leftmost *n* bits of the logical address
2. Use the segment number as an index into the process segment table to find the starting physical address of the segment
3. Compare the offset, expressed in the rightmost *m* bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid
4. The desired physical address is the sum of the starting physical address of the segment plus the offset

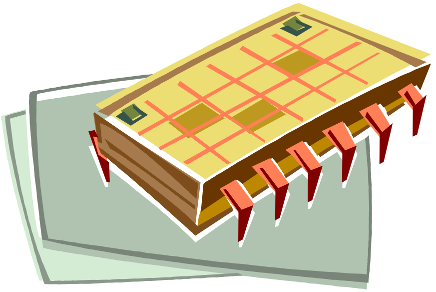
Chapter 8

Virtual Memory and Virtual Page Allocation

Real and Virtual Memory

Real memory

- main memory, the actual RAM



Virtual memory

- memory on disk
- allows for effective multiprogramming and relieves the user of tight constraints of main memory

Thrashing

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

Inverted Page Table

- Page number portion of a virtual address is mapped into a hash value
    - hash value points to inverted page table
- Fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported
- Structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number

Inverted Page Table

Each entry in the page table includes:

Page number

Process identifier

- the process that owns this page

Control bits

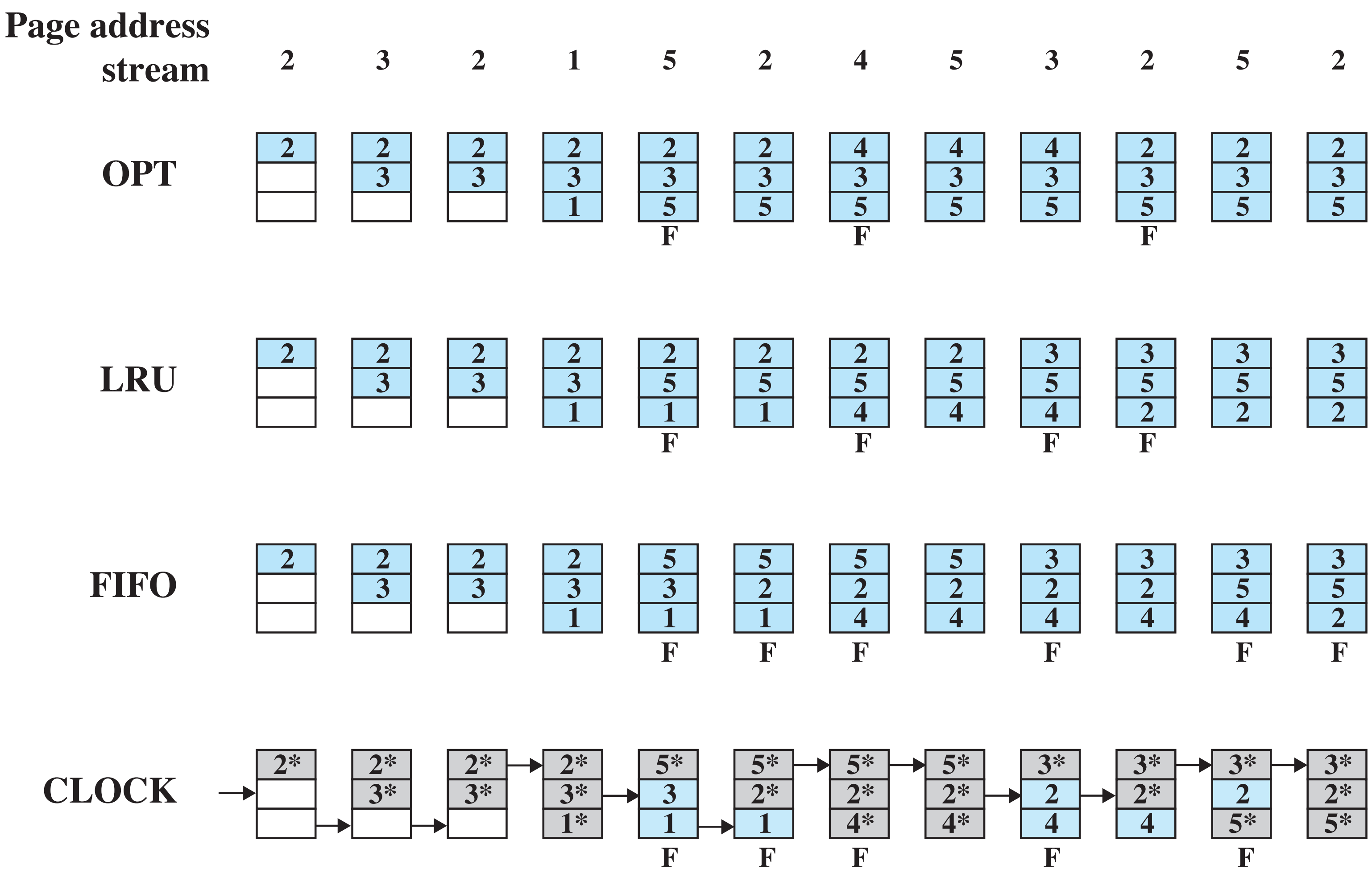
- includes flags and protection and locking information

Chain pointer

- the index value of the next entry in the chain

Translation Lookaside Buffer (TLB)

- 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***



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

Figure 8.14 Behavior of Four Page-Replacement Algorithms