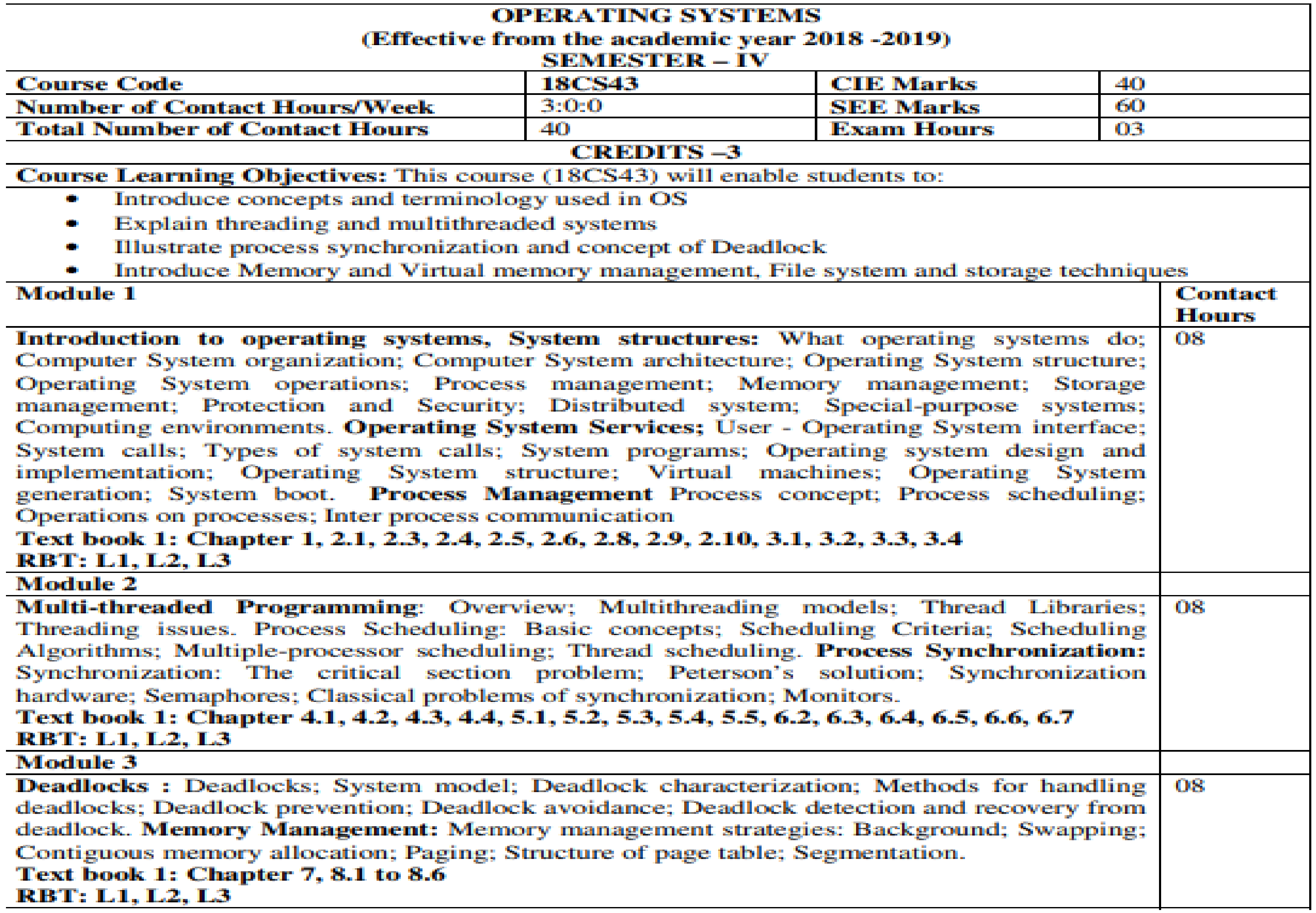
Operating System
(18CS43)

Syllabus



OPERATING SYSTEMS
(Effective from the academic year 2018 -2019)
SEMESTER – IV

Course Code	**18CS43**	**CIE Marks**	40
Number of Contact Hours/Week	3:0:0	**SEE Marks**	60
Total Number of Contact Hours	40	**Exam Hours**	03

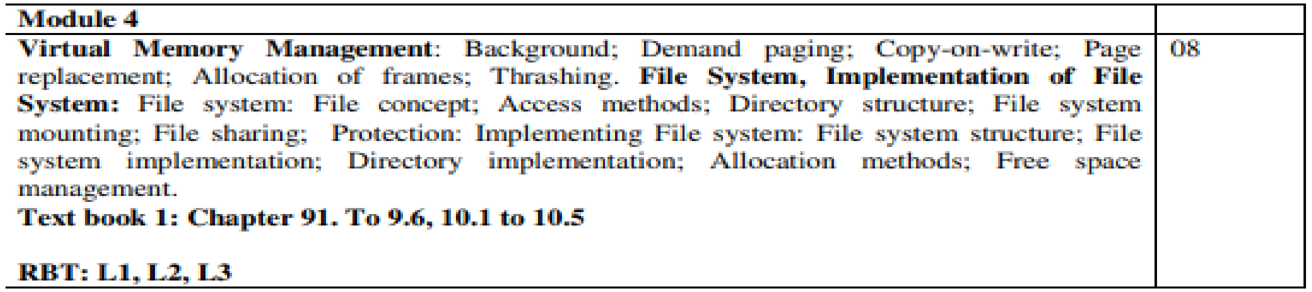
CREDITS –3

Course Learning Objectives: This course (18CS43) will enable students to:

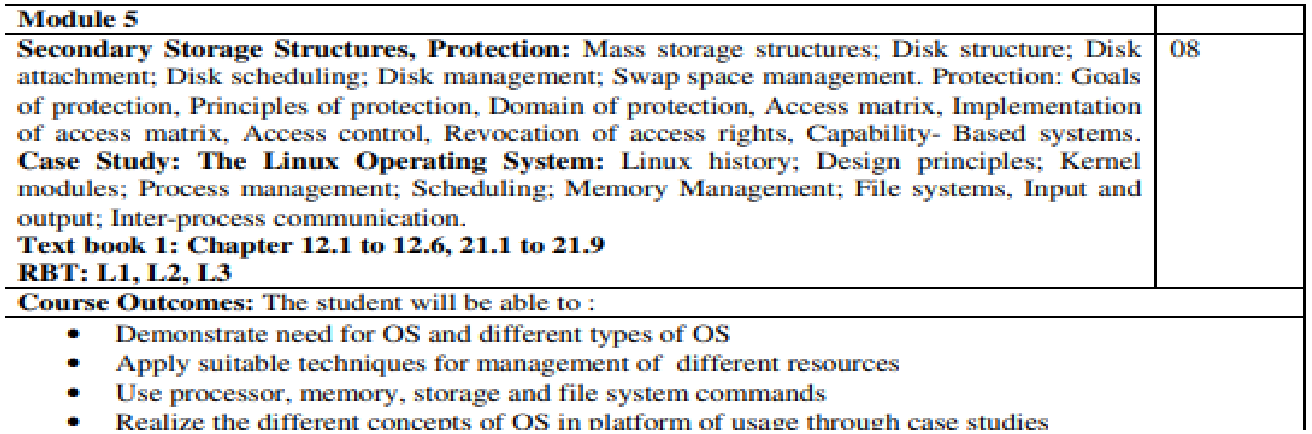
- Introduce concepts and terminology used in OS
- Explain threading and multithreaded systems
- Illustrate process synchronization and concept of Deadlock
- Introduce Memory and Virtual memory management, File system and storage techniques

Module 1	**Contact Hours**
Introduction to operating systems, System structures: What operating systems do; Computer System organization; Computer System architecture; Operating System structure; Operating System operations; Process management; Memory management; Storage management; Protection and Security; Distributed system; Special-purpose systems; Computing environments. **Operating System Services;** User - Operating System interface; System calls; Types of system calls; System programs; Operating system design and implementation; Operating System structure; Virtual machines; Operating System generation; System boot. **Process Management** Process concept; Process scheduling; Operations on processes; Inter process communication **Text book 1: Chapter 1, 2.1, 2.3, 2.4, 2.5, 2.6, 2.8, 2.9, 2.10, 3.1, 3.2, 3.3, 3.4** **RBT: L1, L2, L3**	08
Module 2	
Multi-threaded Programming: Overview; Multithreading models; Thread Libraries; Threading issues. Process Scheduling: Basic concepts; Scheduling Criteria; Scheduling Algorithms; Multiple-processor scheduling; Thread scheduling. **Process Synchronization:** Synchronization: The critical section problem; Peterson's solution; Synchronization hardware; Semaphores; Classical problems of synchronization; Monitors. **Text book 1: Chapter 4.1, 4.2, 4.3, 4.4, 5.1, 5.2, 5.3, 5.4, 5.5, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7** **RBT: L1, L2, L3**	08
Module 3	
Deadlocks : Deadlocks; System model; Deadlock characterization; Methods for handling deadlocks; Deadlock prevention; Deadlock avoidance; Deadlock detection and recovery from deadlock. **Memory Management:** Memory management strategies: Background; Swapping; Contiguous memory allocation; Paging; Structure of page table; Segmentation. **Text book 1: Chapter 7, 8.1 to 8.6** **RBT: L1, L2, L3**	08

Syllabus



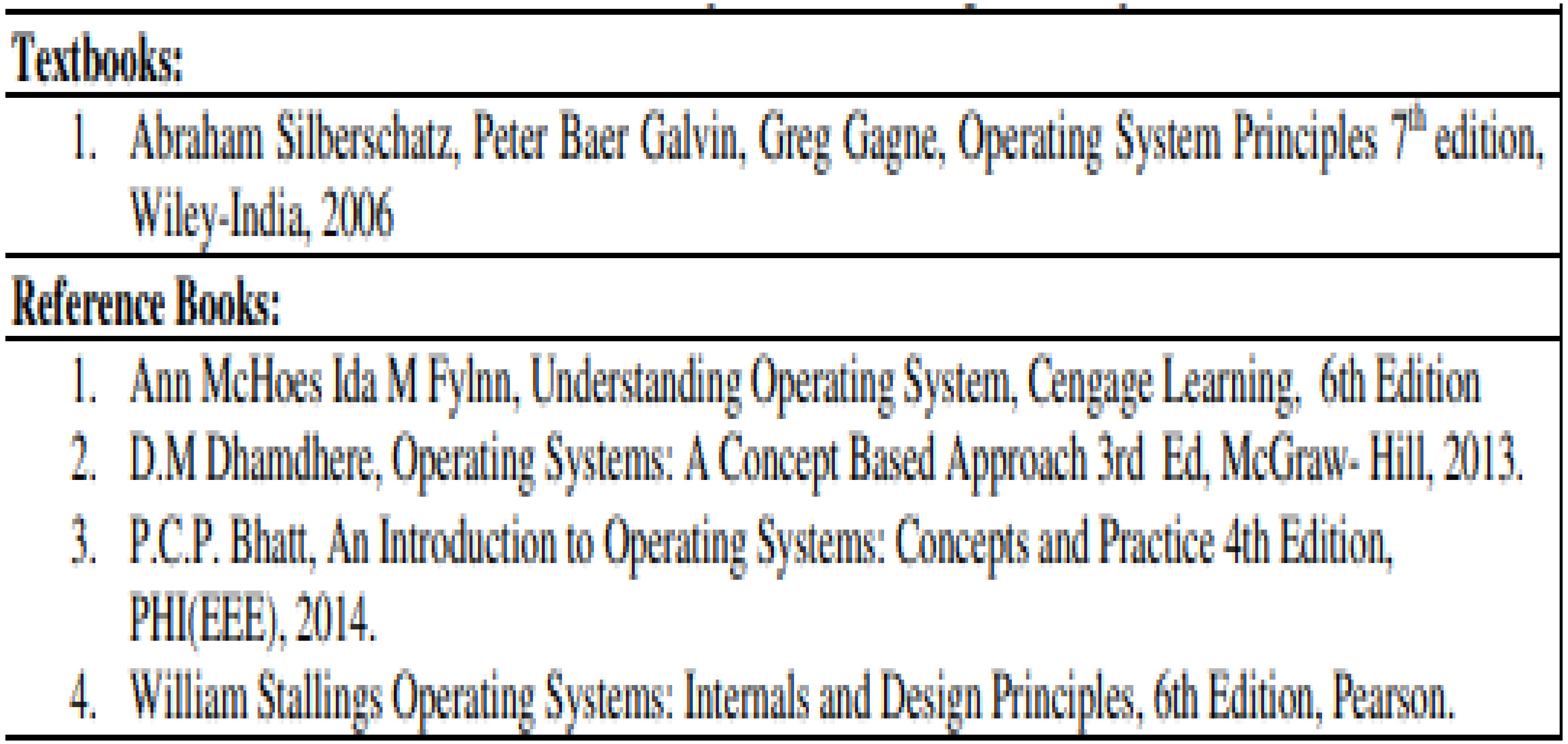
Module 4	
Virtual Memory Management: Background; Demand paging; Copy-on-write; Page replacement; Allocation of frames; Thrashing. **File System, Implementation of File System:** File system: File concept; Access methods; Directory structure; File system mounting; File sharing; Protection: Implementing File system: File system structure; File system implementation; Directory implementation; Allocation methods; Free space management. **Text book 1: Chapter 91. To 9.6, 10.1 to 10.5** **RBT: L1, L2, L3**	08



Module 5	
Secondary Storage Structures, Protection: Mass storage structures; Disk structure; Disk attachment; Disk scheduling; Disk management; Swap space management. Protection: Goals of protection, Principles of protection, Domain of protection, Access matrix, Implementation of access matrix, Access control, Revocation of access rights, Capability- Based systems. **Case Study: The Linux Operating System:** Linux history; Design principles; Kernel modules; Process management; Scheduling; Memory Management; File systems, Input and output; Inter-process communication. **Text book 1: Chapter 12.1 to 12.6, 21.1 to 21.9** **RBT: L1, L2, L3**	08
Course Outcomes: The student will be able to :	

- Demonstrate need for OS and different types of OS
- Apply suitable techniques for management of different resources
- Use processor, memory, storage and file system commands
- Realize the different concepts of OS in platform of usage through case studies

Book



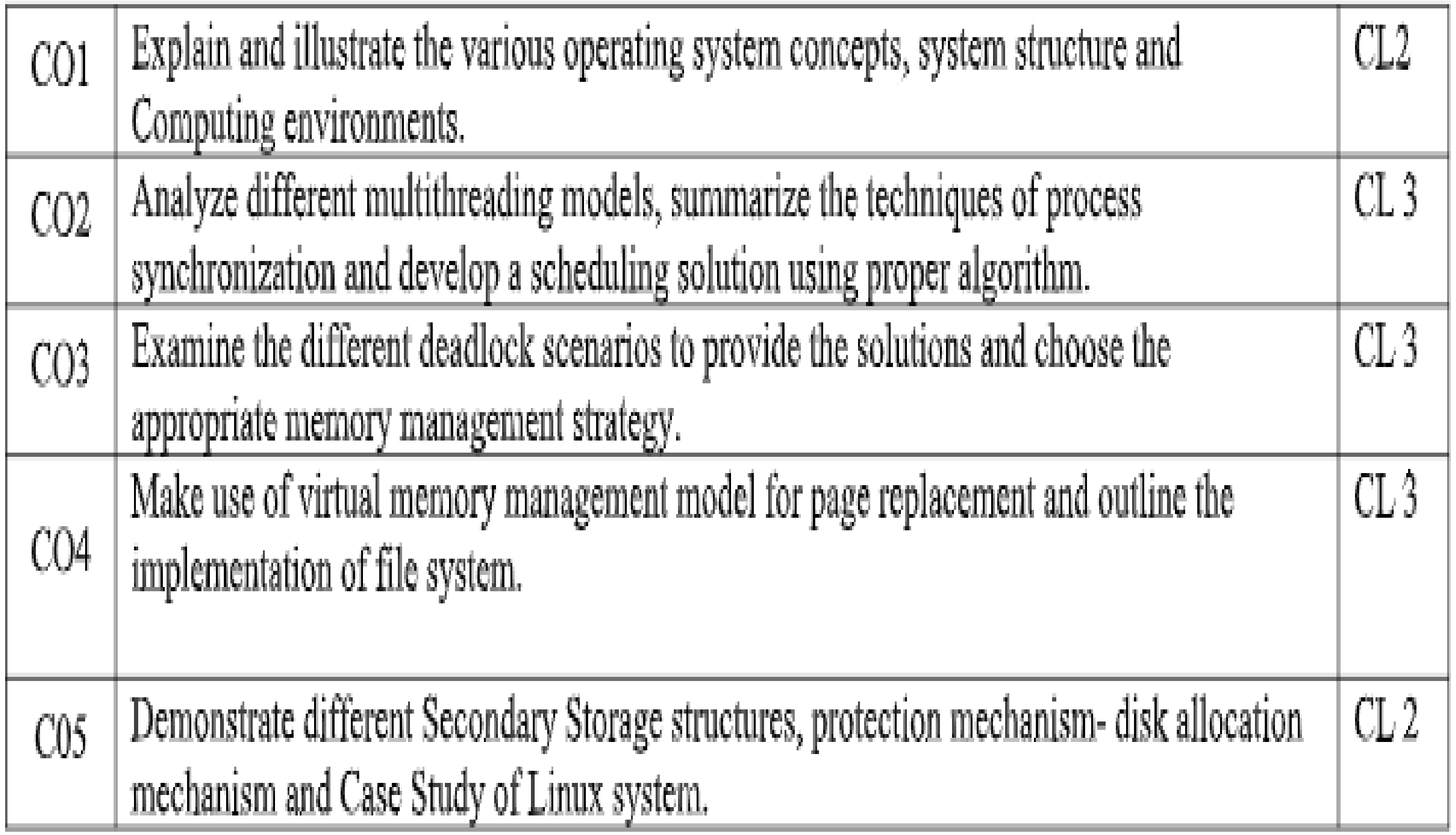
Textbooks:

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating System Principles 7th edition, Wiley-India, 2006

Reference Books:

1. Ann McHoes Ida M Fylnn, Understanding Operating System, Cengage Learning, 6th Edition
2. D.M Dhamdhere, Operating Systems: A Concept Based Approach 3rd Ed, McGraw- Hill, 2013.
3. P.C.P. Bhatt, An Introduction to Operating Systems: Concepts and Practice 4th Edition, PHI(EEE), 2014.
4. William Stallings Operating Systems: Internals and Design Principles, 6th Edition, Pearson.

Course Outcome



CO1	Explain and illustrate the various operating system concepts, system structure and Computing environments.	CL2
CO2	Analyze different multithreading models, summarize the techniques of process synchronization and develop a scheduling solution using proper algorithm.	CL 3
CO3	Examine the different deadlock scenarios to provide the solutions and choose the appropriate memory management strategy.	CL 3
CO4	Make use of virtual memory management model for page replacement and outline the implementation of file system.	CL 3
C05	Demonstrate different Secondary Storage structures, protection mechanism- disk allocation mechanism and Case Study of Linux system.	CL 2

The textbook

- **Operating System Concepts**, 7th Edition Abraham Silberschatz, Yale University Peter Baer Galvin, Corporate Technologies Greg Gagne, Westminster College
  ISBN: 0-471-69466-5

  http://he-cda.wiley.com/WileyCDA/HigherEdTitle/productCd-0471694665.html

MODULE II
Multi-threaded Programming

Overview of chapter 4 and 5

- **Multithreading models**
  - Thread Libraries
  - Threading issues
- **Process Scheduling**
  - Basic concepts;
  - Scheduling Criteria;
  - Scheduling Algorithms;
  - Multiple-processor scheduling;
  - Thread scheduling.

Overview

- A thread is a basic unit of CPU utilization;
  - it comprises a **thread ID, a program counter, a register set, and a stack**.
  - It shares with other threads belonging to the same process its **code section, data section**, and other operating-system resources, such as **open files and signals.**
- If a process has multiple threads of control, it can perform more than one task at a time. Figure 4.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

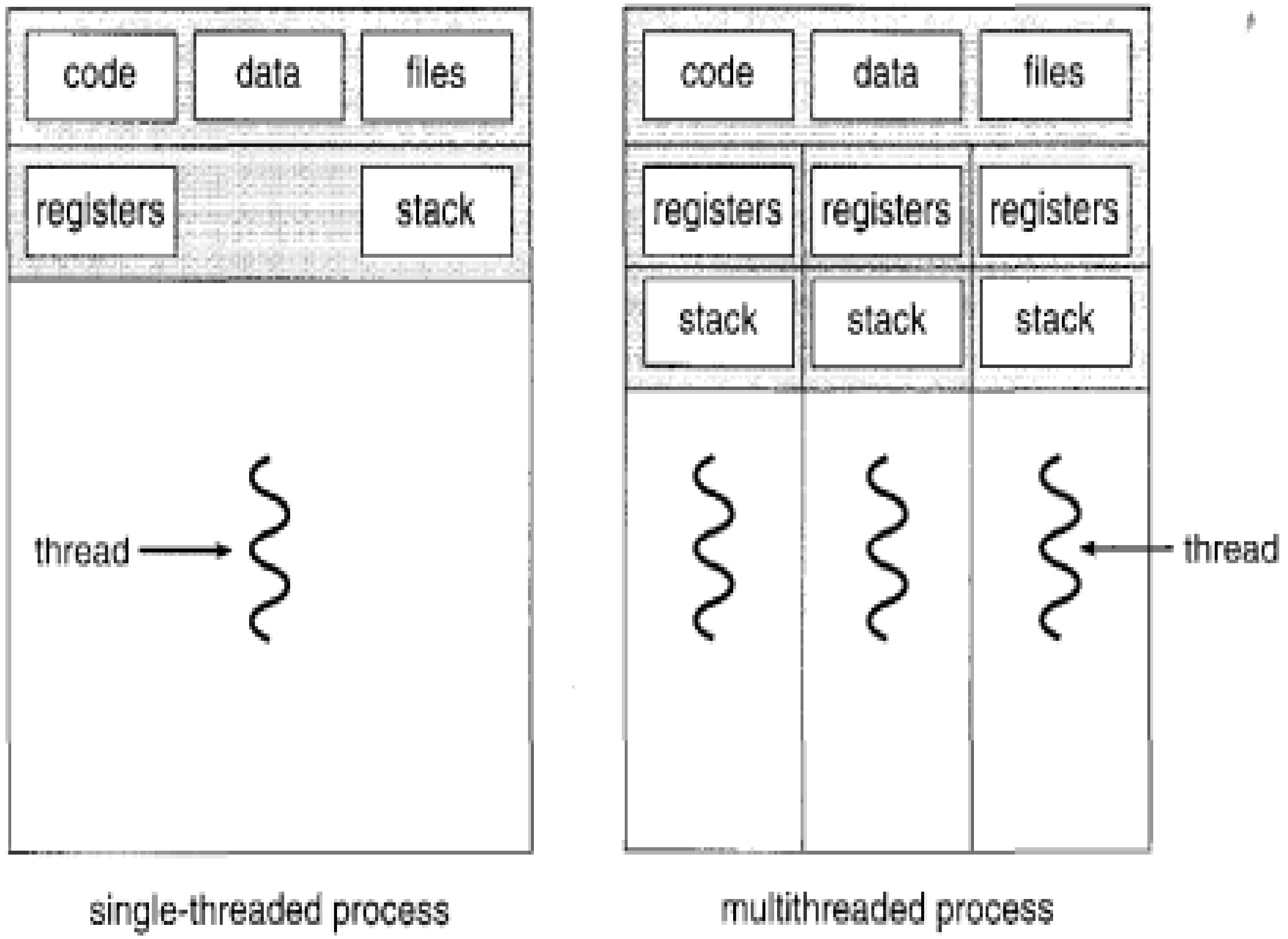


Figure 4.1 Single-threaded and multithreaded processes.

Motivation and benefits

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads: Update display, Fetch data, Spell checking and Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

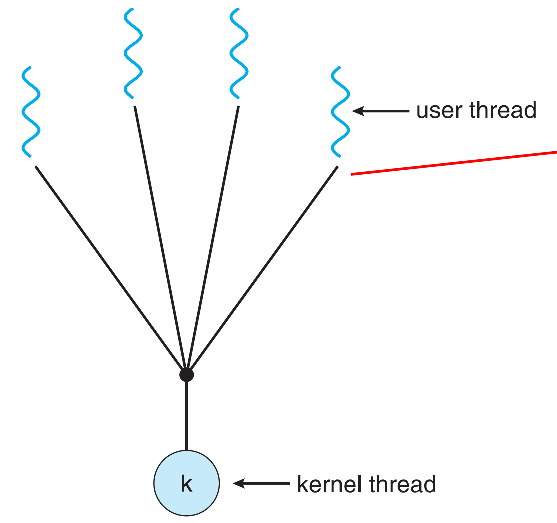
Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multithreading Models

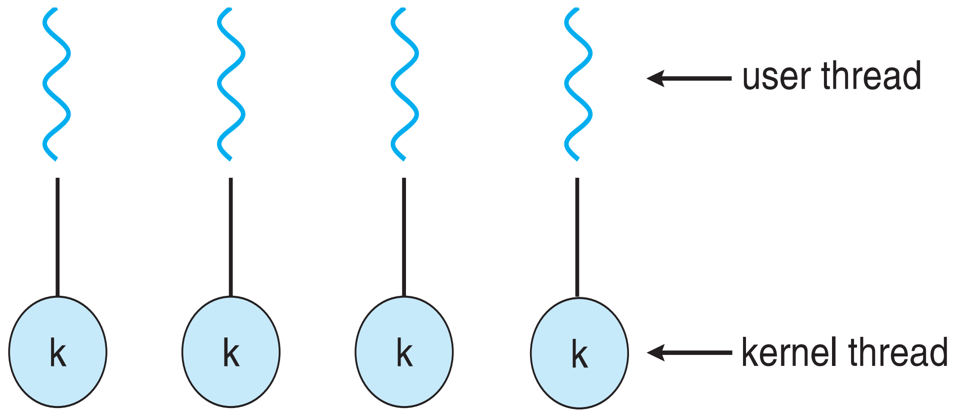
- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
- **User threads** are supported above the kernel and are managed without kernel support, whereas kernel threads are **supported and managed directly by the operating system**.
- Virtually all contemporary operating systems-including Windows XP, Linux, Mac as x, Solaris, and Tru64 UNIX (formerly Digital UNIX)-support kernel threads.
- Three models
  - **Many to one**
  - **One to one**
  - **Many to many**

Multithreading Models



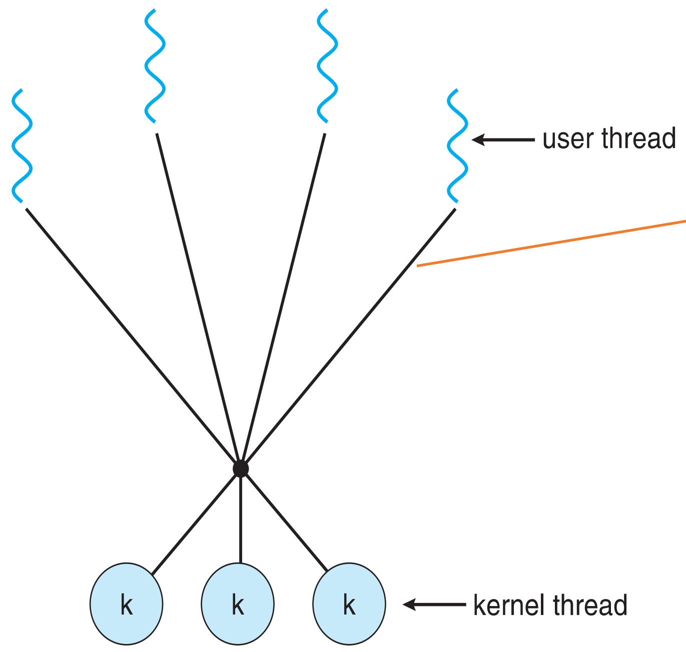
Many-to-One

Thread management is done by the thread library in user space, so it is efficient.Ex:Green Threads & GNU portabe threads



One-to-One

it allows multiple threads to run in parallel on multiprocessors.
Linux, along with the family of Windows operating systems-including Windows 95, 98, NT, 2000, and implement the one-to-one model.



Many-to-Many

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.

Multithreading Models

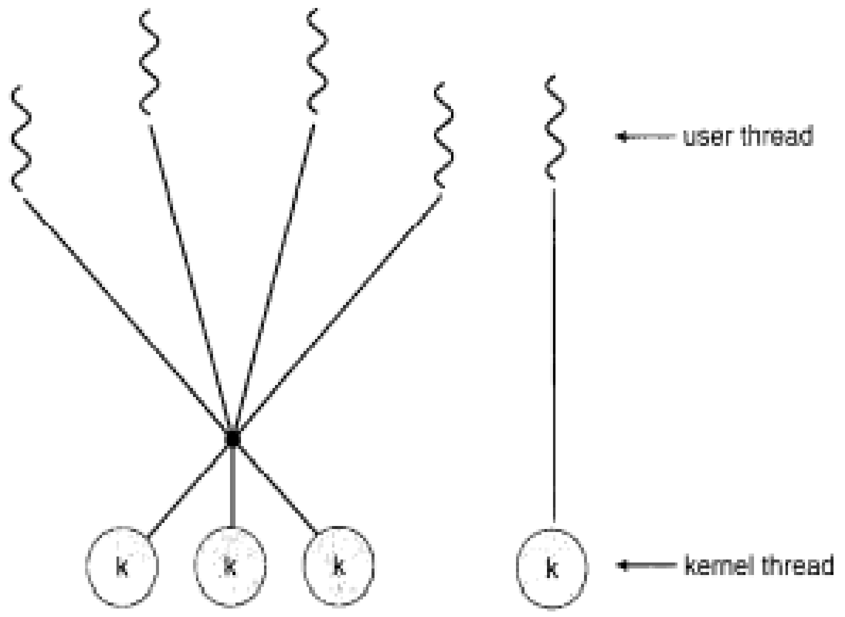


Figure 4.5 Two-level model.

One popular variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation, sometimes referred to as the two-level model (Figure 4.5), is supported by operating systems such as IRIX, HP-UX, and Tru64 UNIX.

Thread Libraries

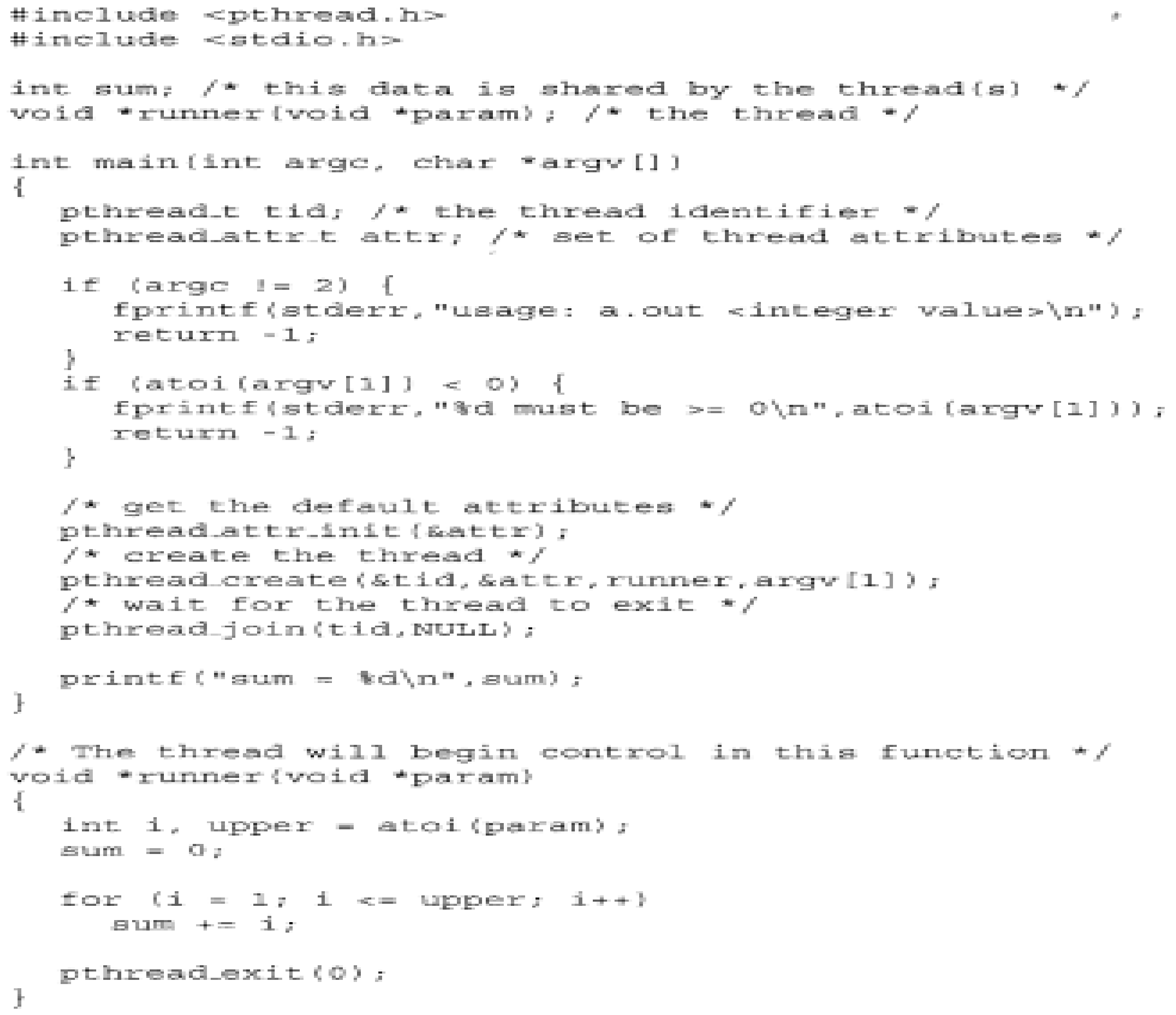
- A thread library provides the **programmer an API for creating and managing threads.**
- There are two primary ways of implementing a thread library.
  - The first approach is to provide a **library entirely in user space with no kernel support.** All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.
  - The second approach is to implement a **kernel-level library** supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.
- Three main thread libraries are in use today: (1) POSIX Pthreads, (2) Win32, and (3) Java.

Thread Libraries

- **Pthreads**, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library.
- **The Win32** thread library is a kernel-level library available on Windows systems.
- **The Java thread API** allows thread creation and management directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is typically implemented using a thread library available on the host system.

Thread Libraries Continued

- **Pthreads** refers to the POSIX standard (JEEE 1003.1c) defining an API for thread creation and synchronization.
- The C program shown in Figure 4.6 demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread.



```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

Chapter 5

Overview

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Thread Scheduling

Basic Concepts

- In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled.
- The objective of **multiprogramming is to have some process running at all times**, to maximize CPU utilization.
- The idea is relatively simple. A process is executed until' it must wait, typically for the completion of some I/O request.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

CPU-I/O Burst Cycle

- The success of CPU scheduling depends on an observed property of processes: Process execution consists of a cycle of **CPU execution** and **I/O wait**. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1)

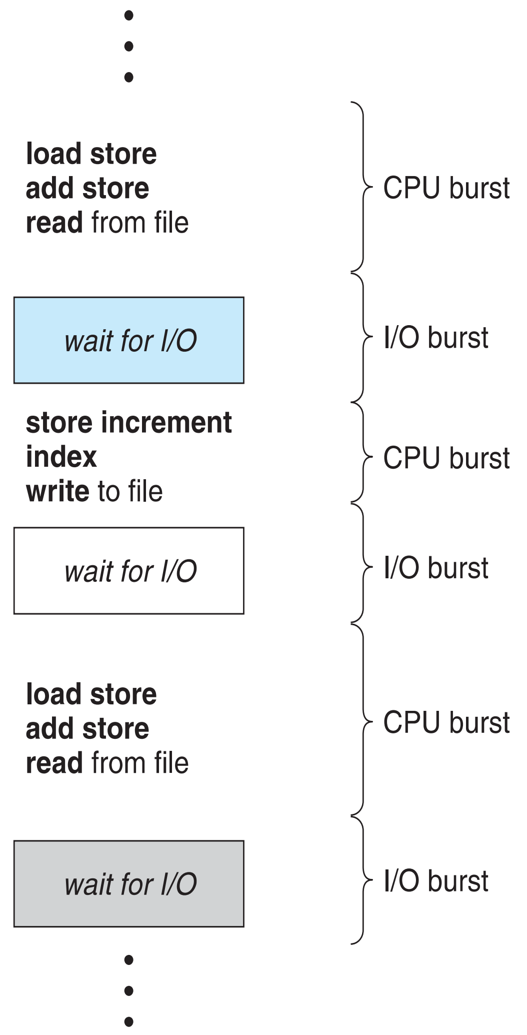


Figure 5.1 Alternating sequence of CPU and I/O bursts

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities
  - **In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running.

- **Burst time**:Time required by a process for CPU Execution
- **Arrival Time**: Time at which the process arrives in the ready queue.
- **Gantt(Generalized activity Normalization time table) chart** is type of chart which shows activities or process or task performed against time

Scheduling Criteria

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput.** If the CPU is busy executing processes, then work is being done. One measure of work is the **number of processes that are completed per time unit**, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround time**. **The interval from the time of submission of a process to the time of completion is the turnaround time**. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

CPU Scheduling Algorithms

- First-Come, First Serve (FCFS or FIFO) (non-preemptive)
- Priority –
  - Shortest Job First (SJF; non-preemptive) or
  - Shortest Remaining Time First (SRTF; preemptive)- **Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling**.
  - Priority scheduling
- Round Robin (preemptive)

Chapter 6-Process Synchronization

Overview

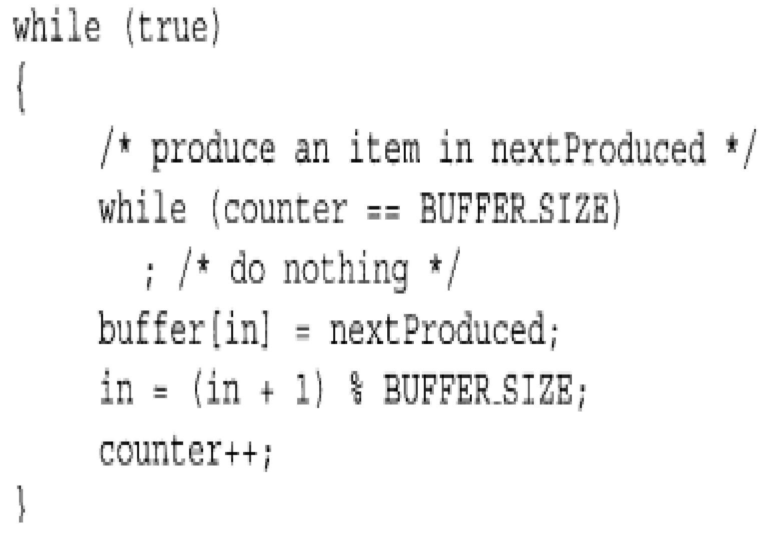
- Synchronization: The critical section problem;
- Peterson’s solution;
- Synchronization hardware;
- Semaphores;
- Classical problems of synchronization; Monitors.

Background

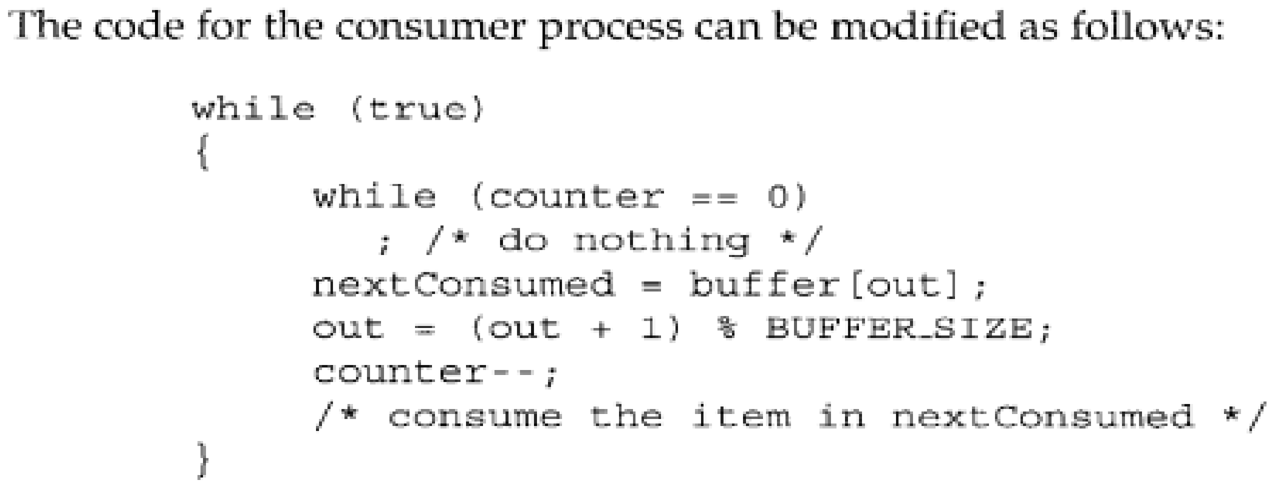
We discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

The code for the producer process

can be modified as follows:



```
while (true)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
      ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```



The code for the consumer process can be modified as follows:

```
while (true)
{
      while (counter == 0)
         ; /* do nothing */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /* consume the item in nextConsumed */
}
```

- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

The Critical-Section Problem

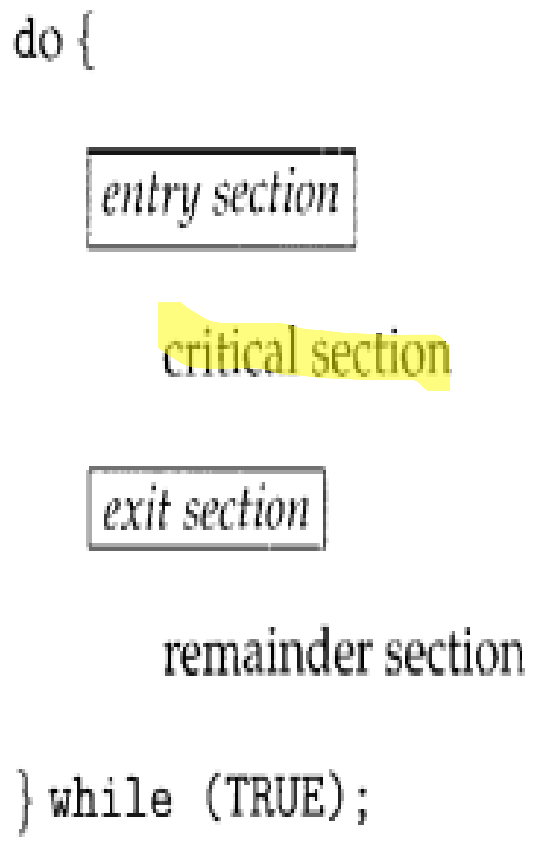
Consider a system consisting of n processes {Po, PI, ..., Pn-1}.

- Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
- The critical-section problem is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**

The Critical-Section Problem Continued

The critical section may be followed by an **exit section**.

The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.



```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

Figure 6.1 General structure of a typical process P_i.

The Critical-Section Problem Continued

Solutions to the critical section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. When no process is executing in its critical section, and there exists a process that wishes to enter its critical section, it should not have to wait indefinitely to enter it.
3. **Bounded waiting.** There must be a bound on the number of times a process is allowed to execute in its critical section, after another process has requested to enter its critical section and before that request is accepted.

Two general approaches are used to handle critical sections in operating systems:

(1) preemptive kernels and (2) nonpreemptive kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU

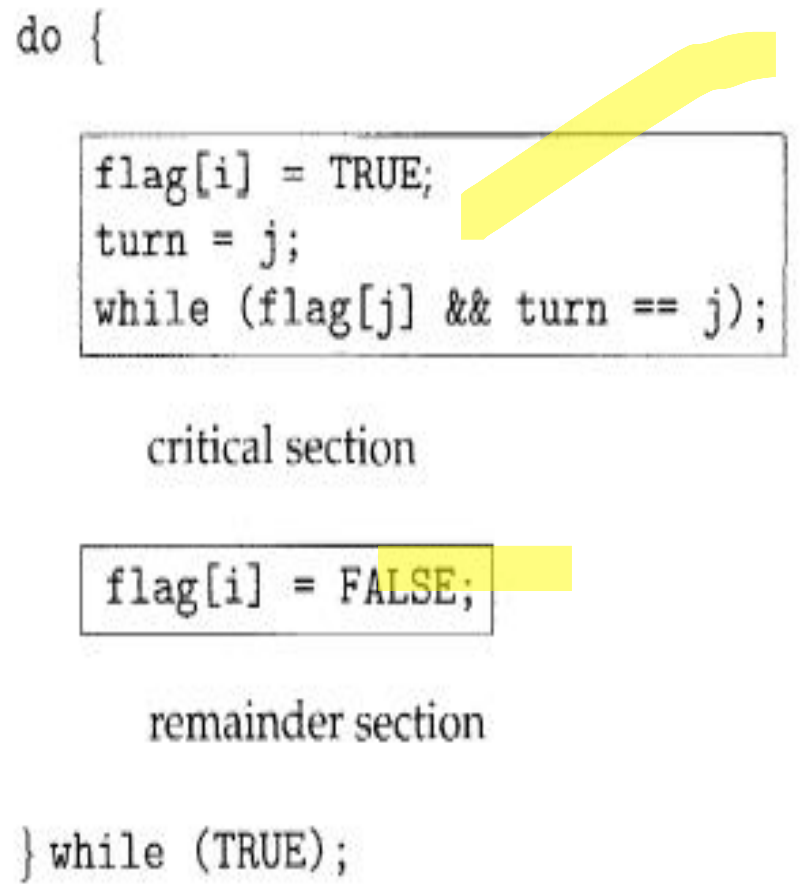
Windows XP and Windows 2000 are nonpreemptive kernels, as is the traditional UNIX kernel. Prior to Linux 2.6, the Linux kernel was

Peterson's Solution

- Peterson's solution:**a classic software-based solution to the critical-section problem**
- It provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress/ and bounded waiting requirements.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered Po and P1. For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1 - i.
- Peterson's solution requires two data items to be shared between the two processes:
  - int turn;
  - boolean flag[2];

Peterson's Solution

- The variable **turn** indicates whose turn it is to enter its critical section.
- The **flag array** is used to indicate if a process is ready to enter its critical section.
- To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so



```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure 6.2 The structure of process P_i in Peterson's solution.

Peterson's Solution Continued

- We now prove that this solution is correct. We need to show that:

1. **Mutual exclusion is preserved. 2. The progress requirement is satisfied. 3. The bounded-waiting requirement is met**.

To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] == true. These two observations imply that Po and PI could not have successfully executed their while statements at about the same time, since the value of turn can be either a 0 or 1 but cannot be both.

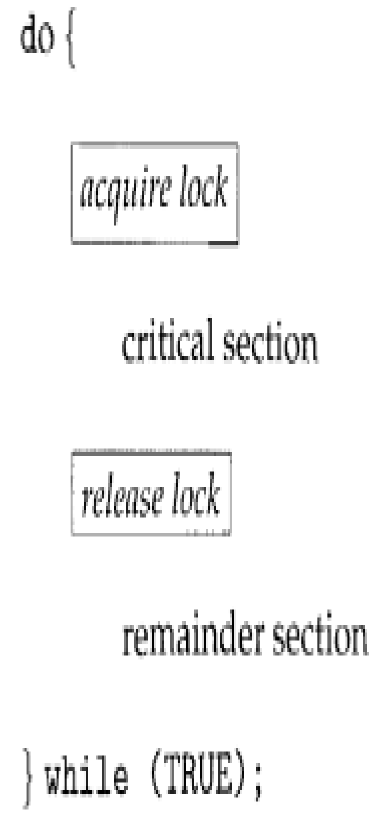
To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] == true and turn =:::::: j; this loop is the only one possible. If Pj is not ready to enter the critical section, then flag [j] == false, and Pi can enter its critical section. If Pj has set flag [j] to true and is also executing in its while statement, then either turn ::::::::::: i or turn == j. If turn =:::::: i, then Pi will enter the critical section. If turn == j, then Pj will enter the critical section.

Synchronization Hardware(Lock)

In general, we can state that any solution to the critical-section problem requires a simple tool- **a lock**.

Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

Several more solutions to the critical-section problem using techniques ranging from hardware to software based APIs available to application programmers. All these solutions are based on the premise of locking;

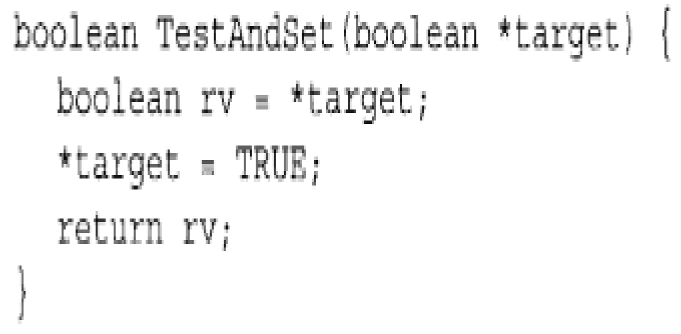


```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE);
```

Figure 6.3 Solution to the critical-section problem using locks.

Synchronization Hardware(Test and Set)

- We present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem
  - The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.
  - Unfortunately, this solution is not as feasible in a multiprocessor environment
- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically-that is, as one uninterruptible unit.



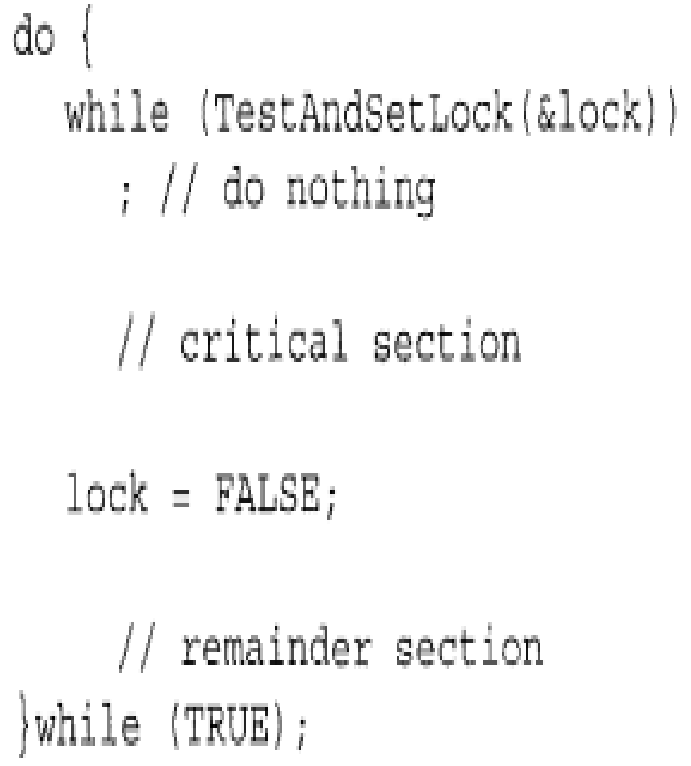
```
boolean TestAndSet(boolean *target) {
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
```

Figure 6.4 The definition of the TestAndSet() instruction.

The TestAndSet () instruction can be defined as shown in Figure 6.4. The important characteristic is that this instruction is executed atomically(non-interruptible).

Synchronization Hardware(Mutual Exclusion with test and set)

- Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process Pi is shown in Figure 6.5.



```
do {
  while (TestAndSetLock(&lock))
    ; // do nothing

    // critical section

  lock = FALSE;

    // remainder section
}while (TRUE);
```

Figure 6.5 Mutual-exclusion implementation with TestAndSet().

Synchronization Hardware(Mutual Excusion with Swap())

- The Swap () instruction, in contrast to the TestAndSet () instruction, operates on the contents of two words; it is defined as shown in Figure 6.6. Like the TestAndSet () instruction, it is executed atomically. If the machine supports the Swap () instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process Pi is shown in Figure 6.7. Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement

```
void Swap(boolean *a, boolean *b) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

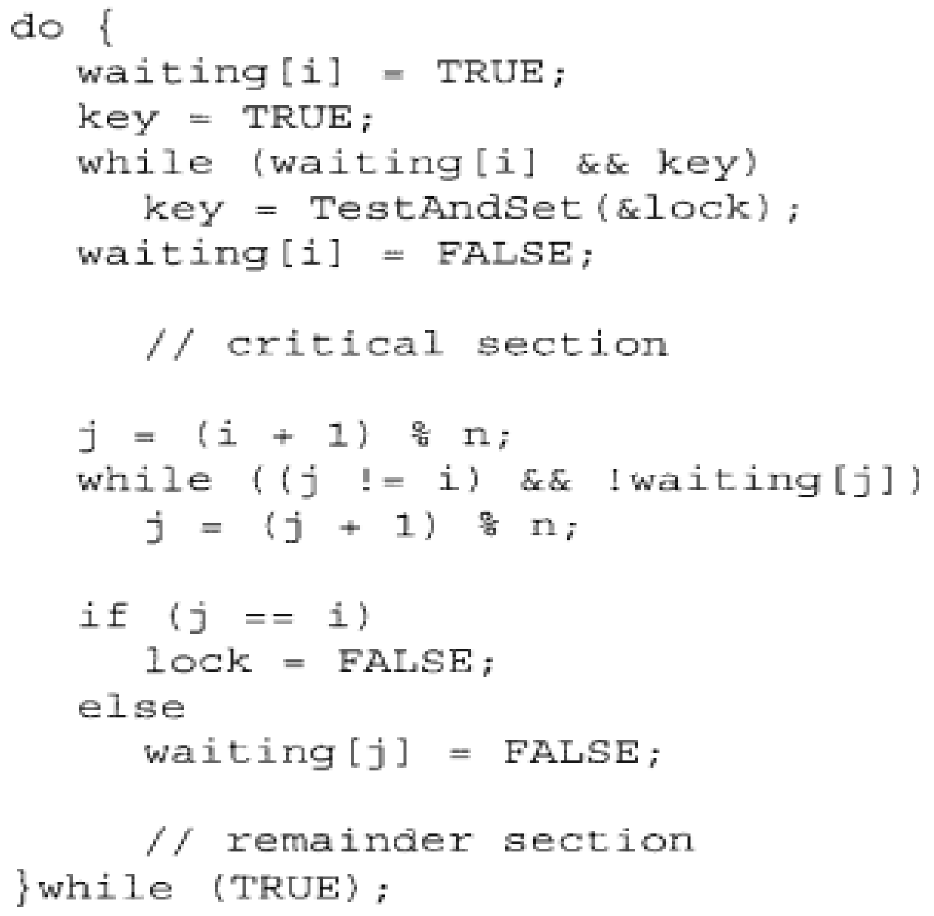
Figure 6.6 The definition of the Swap() instruction.

```
do {
  key = TRUE;
  while (key == TRUE)
    Swap(&lock, &key);

    // critical section

  lock = FALSE;

    // remainder section
}while (TRUE);
```

Figure 6.7 Mutual-exclusion implementation with the Swap() instruction.

Synchronization Hardware(Bounded-waiting mutual exclusion with TestAndSet())

- In Figure 6.8, we present another algorithm using the TestAndSet() instruction that satisfies all the critical-section requirements. The common data structures are boolean waiting[n] ; boolean lock;



```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

Figure 6.8 Bounded-waiting mutual exclusion with TestAndSet().

Mutex Locks

- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

```
•   acquire() {
        while (!available)
            ; /* busy wait */
        available = false;
    }
•   release() {
        available = true;
    }
•   do {
     acquire lock
        critical section
     release lock
       remainder section
  } while (true);
```

Semaphores

The various hardware-based solutions to the critical-section problem (using the TestAndSet ()and Swap () instructions) presented in Section 6.4 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.

- A semaphore S is an integer variable that, **apart from initialization**, is accessed only through two standard atomic operations:

  wait() and signal ().

- The wait () operation was originally **termed P** (from the Dutch proberen, "to test"); signal()was originally **called V** (from verhagen, "to increment").

- **Definition of the `wait()` operation**

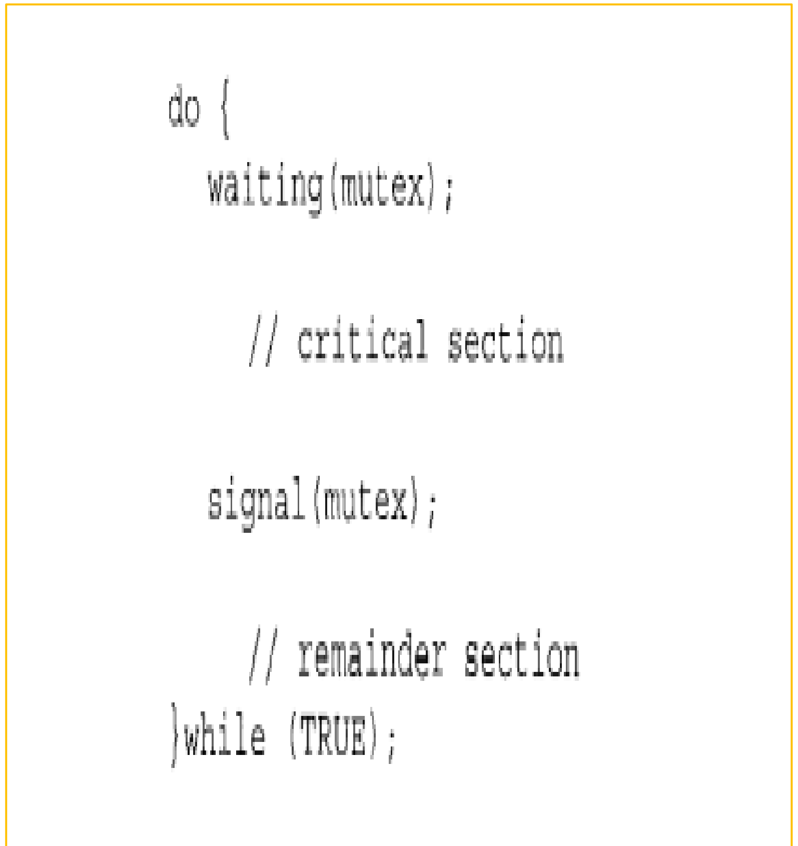
```
wait(S) {
     while (S <= 0)
          ; // busy wait
     S--;
}
```

- **Definition of the `signal()` operation**

```
signal(S) {
     S++;
}
```

Semaphore Usage

- Types of Semaphore
  - Counting Semaphore
  - Binary Semaphore(Mutex Locks)
- **Counting Semaphore**:
  - The value of a counting semaphore can range over an unrestricted domain.
  - Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available
- Binary Semaphore(Mutex Locks):
  - We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, mutex, initialized to 1. Each process Pi is organized as shown in Figure 6.9.

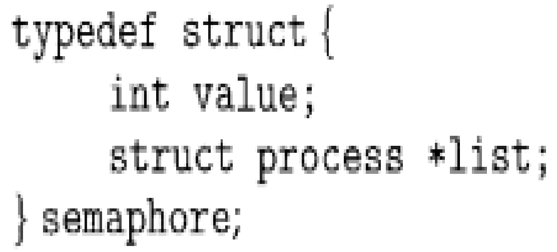


```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Figure 6.9 Mutual-exclusion implementation with semaphores.

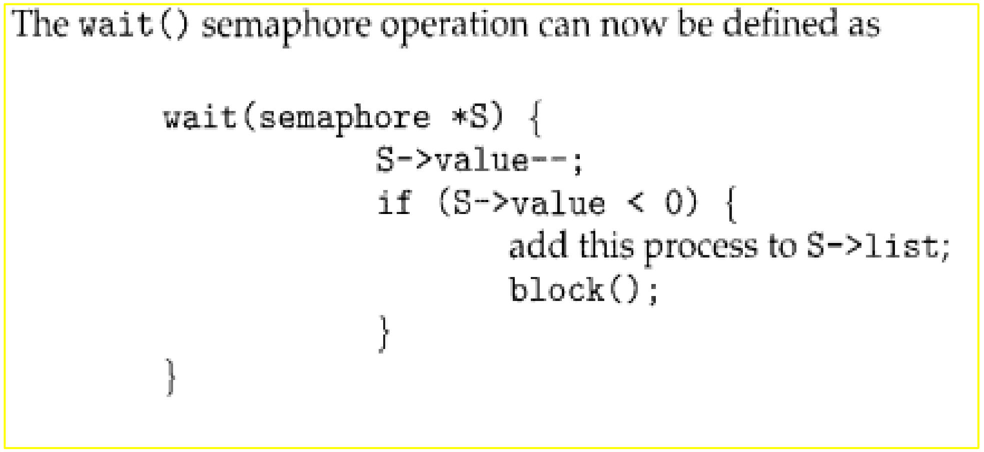
Semaphore Implementation

- To implement semaphores under this definition, we define a semaphore as a "C" struct:



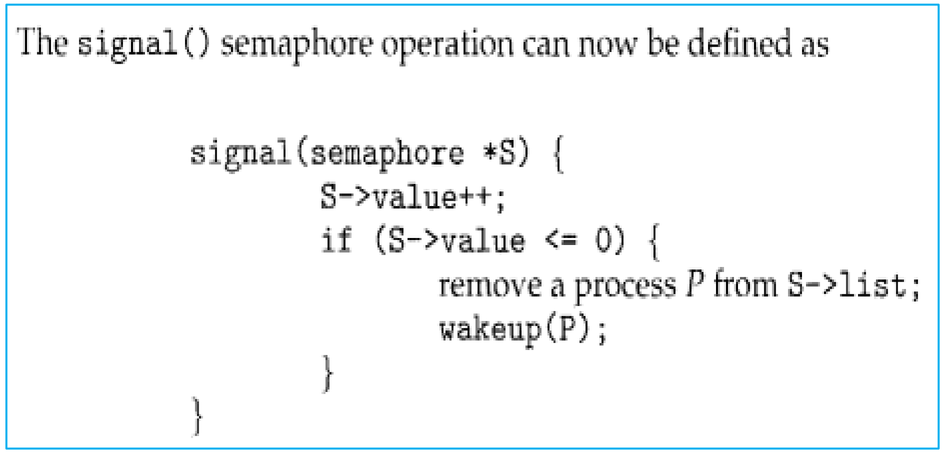
```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- Each semaphore has an **integer value** and a **list of processes** list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process



The wait() semaphore operation can now be defined as

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

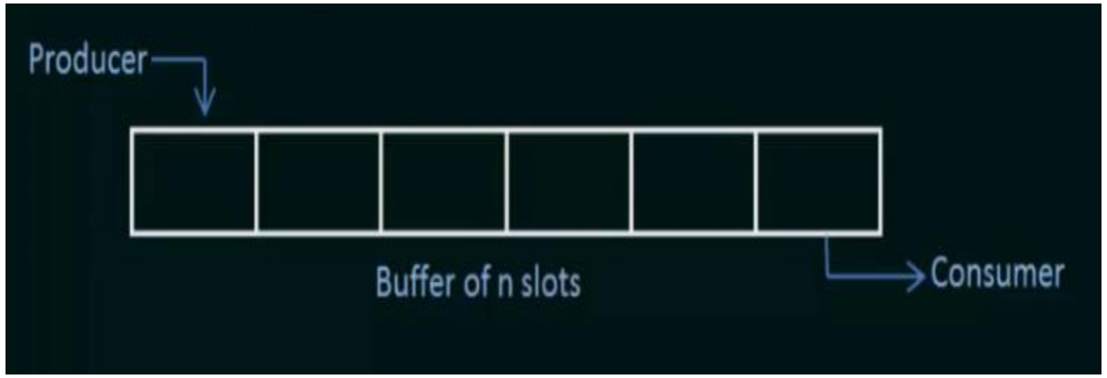


The signal() semaphore operation can now be defined as

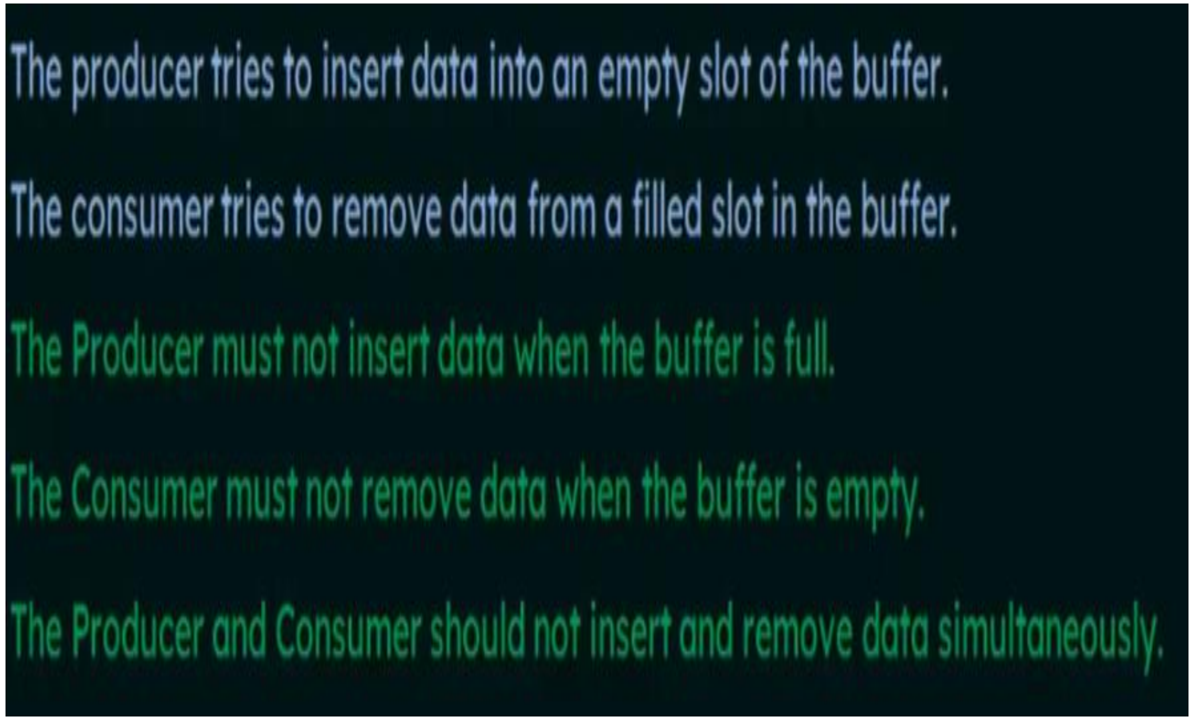
```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Classic Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
    1. Bounded-Buffer Problem
    2. Readers and Writers Problem
    3. Dining-Philosophers Problem
- Bounded-Buffer Problem(Producer Consumer Problem)
    - We assume that the pool consists of n buffers, each capable of holding one item



Problem faced in the Producer-Consumer



The producer tries to insert data into an empty slot of the buffer.

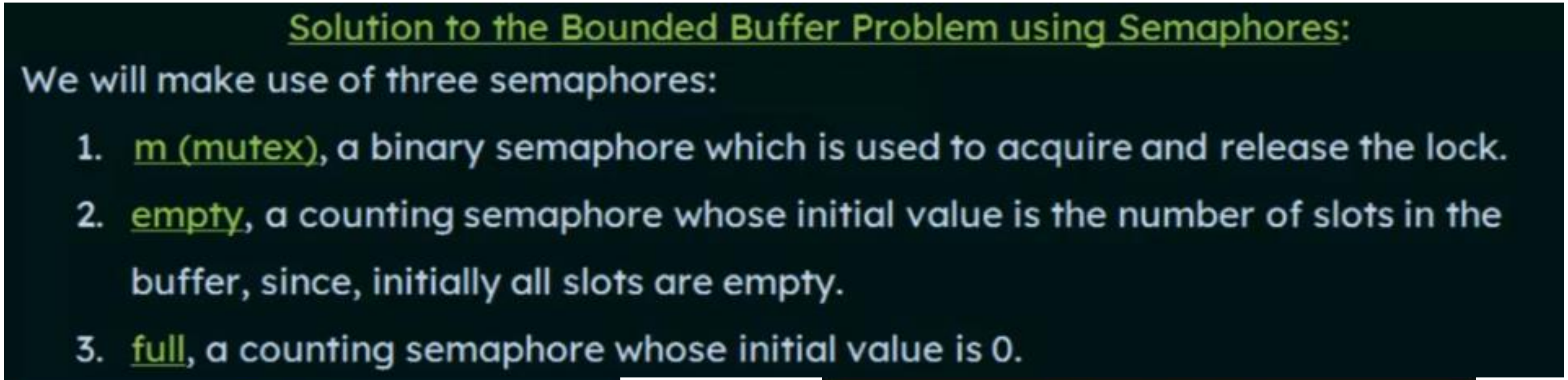
The consumer tries to remove data from a filled slot in the buffer.

The Producer must not insert data when the buffer is full.

The Consumer must not remove data when the buffer is empty.

The Producer and Consumer should not insert and remove data simultaneously.

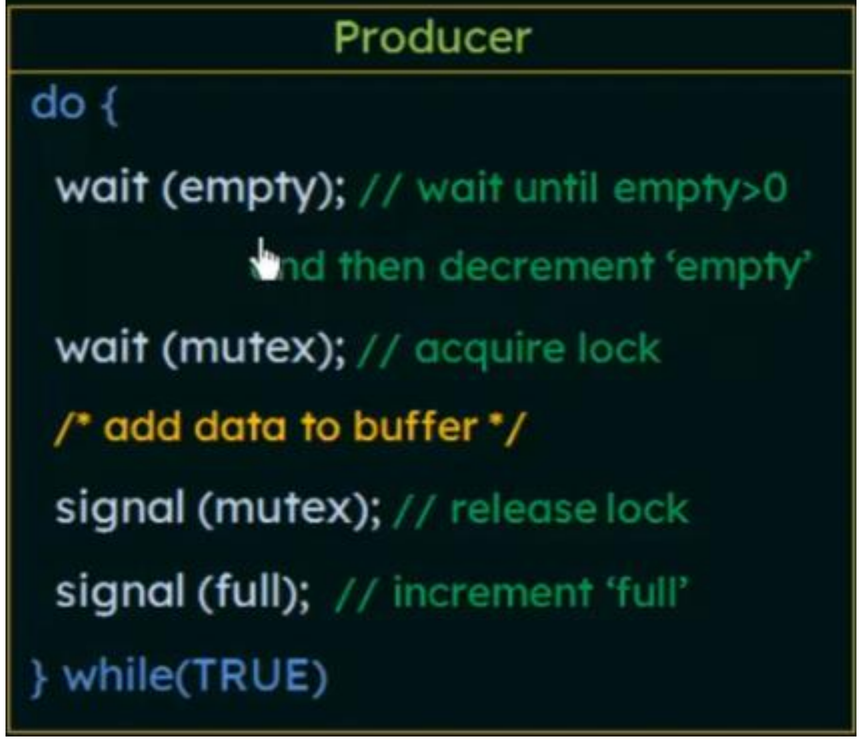
Bounded-Buffer Problem(Producer Consumer Problem)



Solution to the Bounded Buffer Problem using Semaphores:

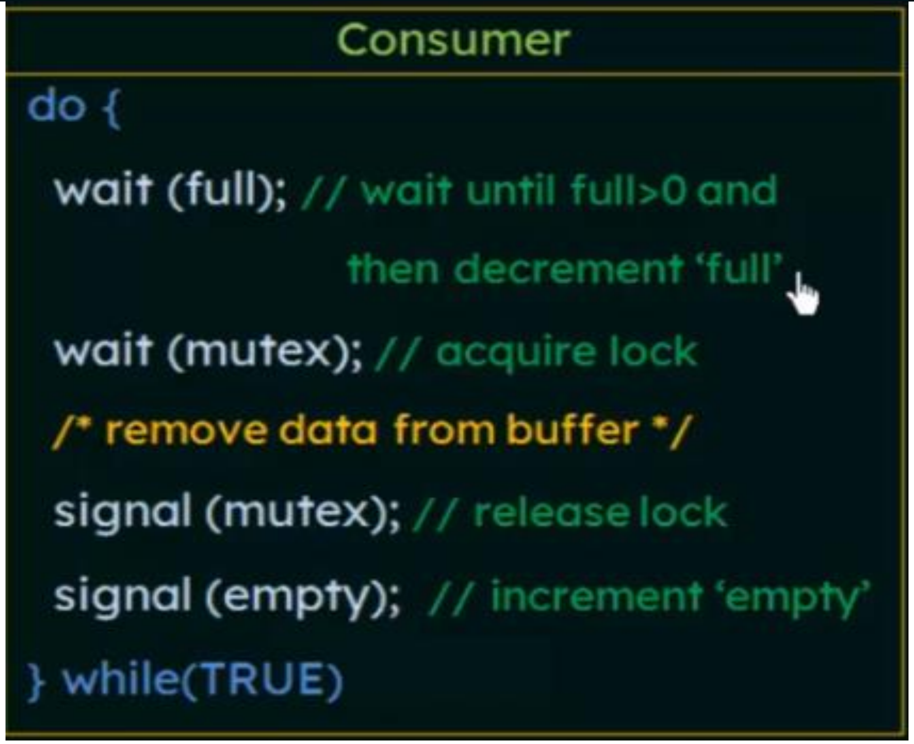
We will make use of three semaphores:

1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.



Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full); // increment 'full'
} while(TRUE)
```



Consumer

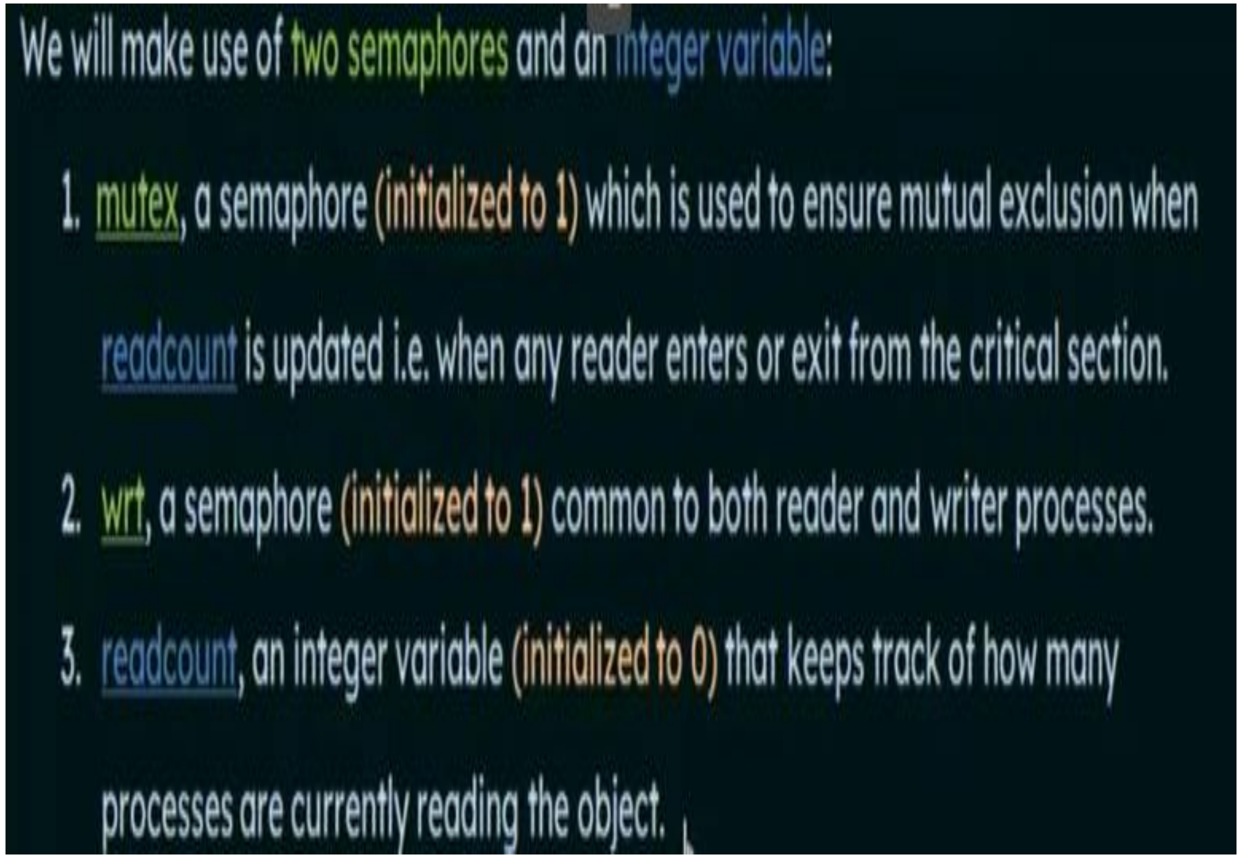
```
do {
  wait (full); // wait until full>0 and
                  then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty); // increment 'empty'
} while(TRUE)
```

The Readers-Writers Problem

- A database is to be shared among **several concurrent processes**. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as **readers and to the latter as writers**. Obviously, if two readers access the shared data simultaneously, no adverse affects will result.
- However, **if a writer and some other thread** (either a reader or a writer) access the database simultaneously, chaos may ensue.
- To ensure that these difficulties do not arise, **we require that the writers have exclusive access to the shared database**. This synchronization problem is referred to as the readers-writers problem.

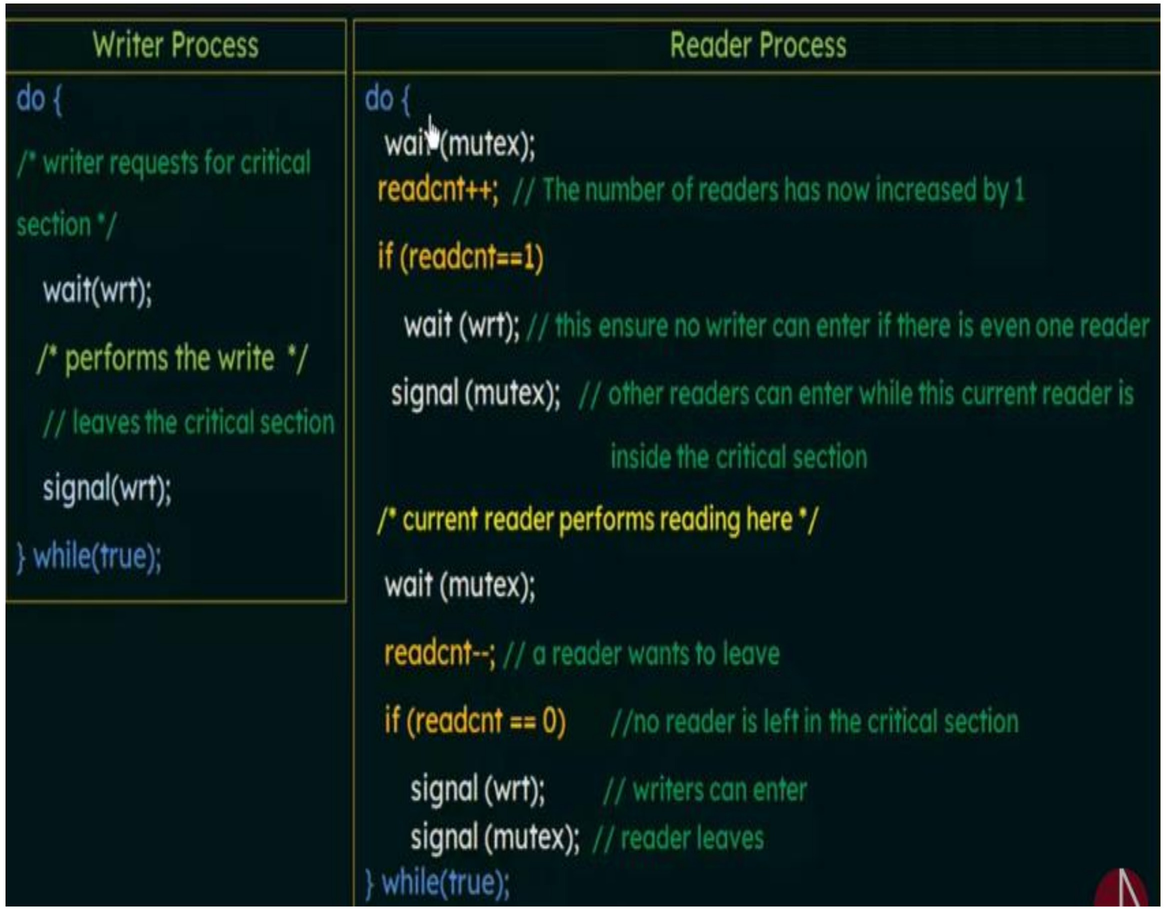
The Readers-Writers Problem Continued

Solution to Readers-Writers Problem



We will make use of two semaphores and an integer variable:

1. mutex, a semaphore (initialized to 1) which is used to ensure mutual exclusion when readcount is updated i.e. when any reader enters or exit from the critical section.
2. wrt, a semaphore (initialized to 1) common to both reader and writer processes.
3. readcount, an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.



Writer Process

```
do {
/* writer requests for critical
section */
   wait(wrt);
   /* performs the write */
   // leaves the critical section
   signal(wrt);
} while(true);
```

Reader Process

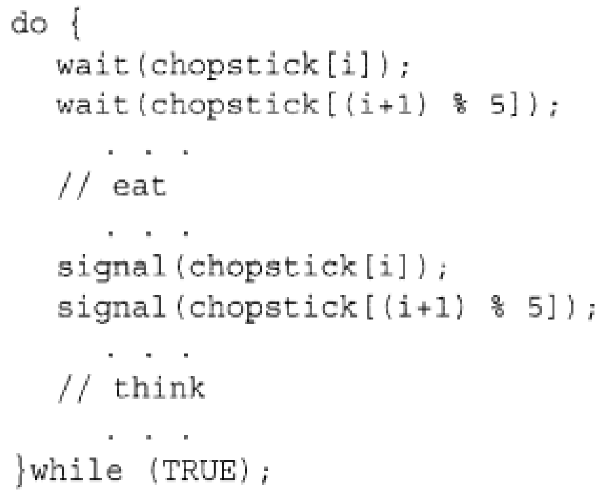
```
do {
  wait(mutex);
 readcnt++;  // The number of readers has now increased by 1
 if (readcnt==1)
   wait (wrt); // this ensure no writer can enter if there is even one reader
  signal (mutex);  // other readers can enter while this current reader is
                      inside the critical section
 /* current reader performs reading here */
  wait (mutex);
  readcnt--; // a reader wants to leave
  if (readcnt == 0)    //no reader is left in the critical section
    signal (wrt);     // writers can enter
    signal (mutex);  // reader leaves
} while(true);
```

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Deadlock handling

- Allow at most 4 philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
- Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick



```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
}while (TRUE);
```

Figure 6.15 The structure of philosopher *i*.

Dining-Philosophers Problem Continued

Problems with Semaphore

- Incorrect use of semaphore operations:
    - signal (mutex) …. wait (mutex)
    - wait (mutex) … wait (mutex)
    - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

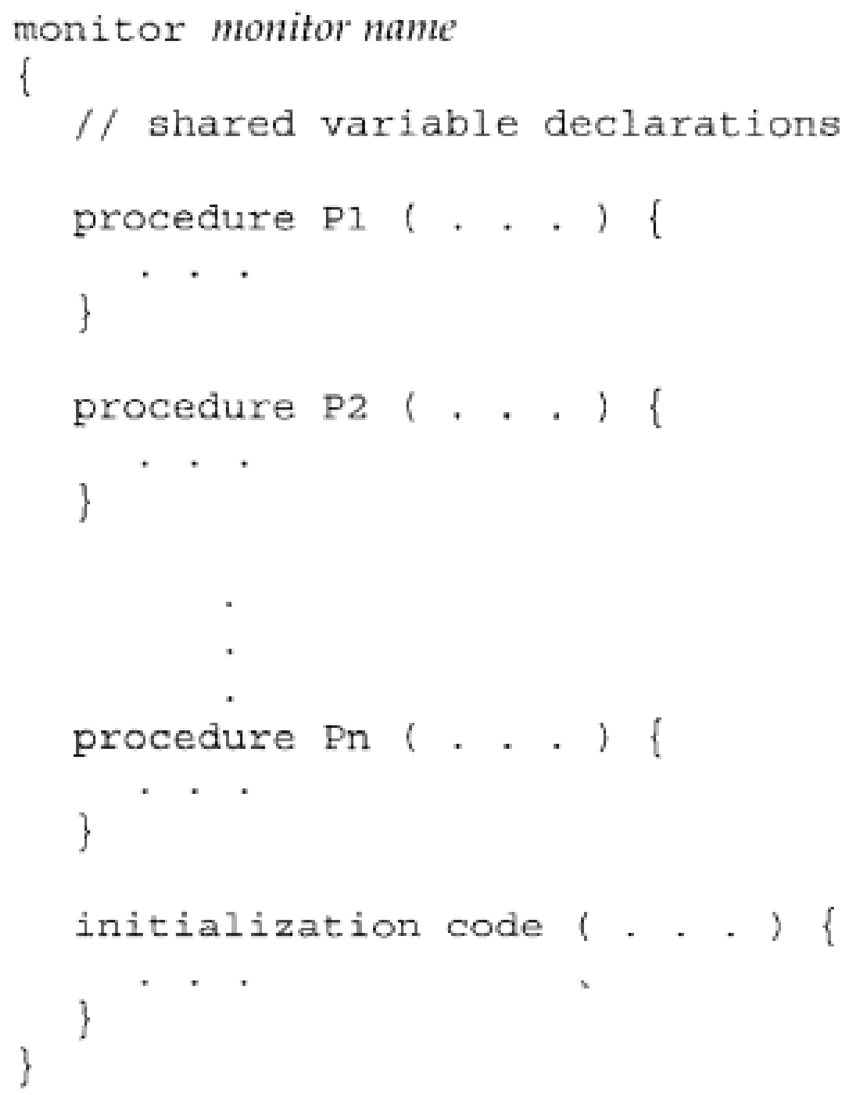
Monitors

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in **timing errors that are difficult to detect**, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.
- To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental **high-level synchronization construct-the monitor type**.
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure

Monitor Usage

- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes
- For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the **condition construct.** A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:
- condition x, y;

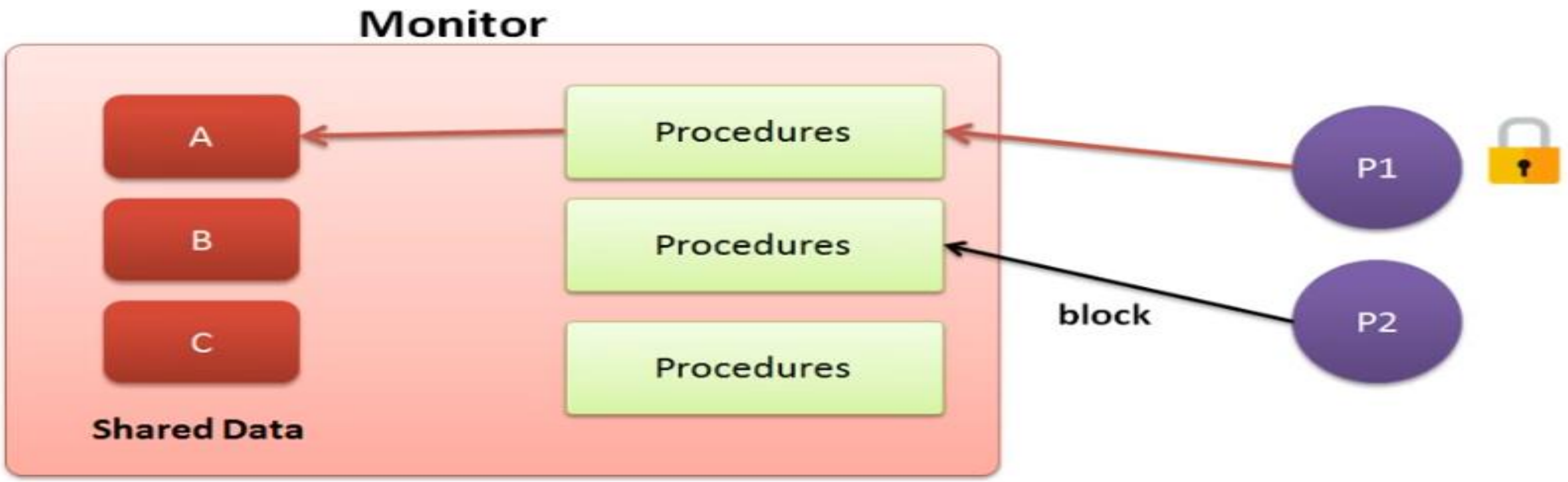
  x. wait(); x. signal();



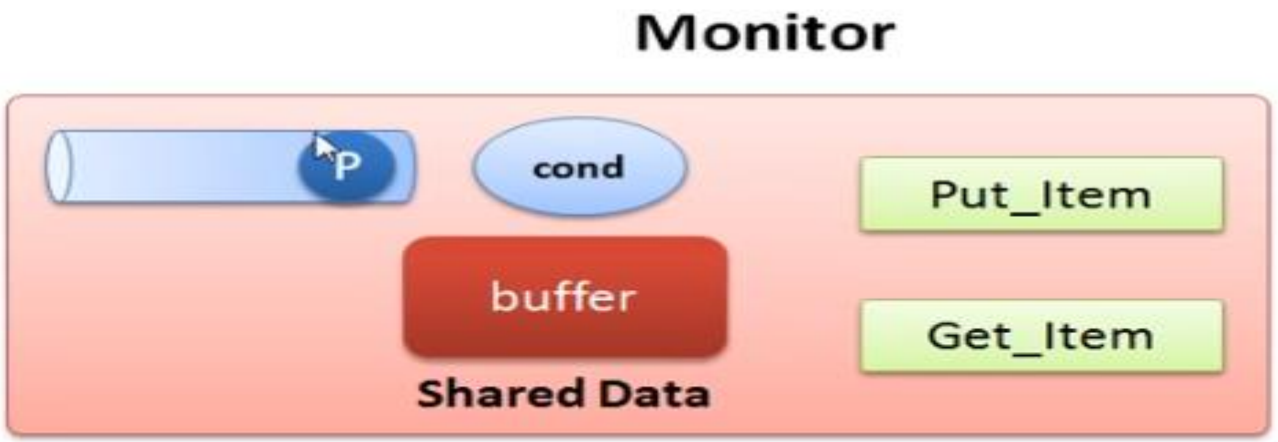
```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Figure 6.16 Syntax of a monitor.

Monitors



Only one thread can enter in monitor at any time.



Schematic View of Monitor

`x.wait()` – The process invoking this operation is suspended until another process invokes **`x.signal()`**

`x.signal()` – It resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect;

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

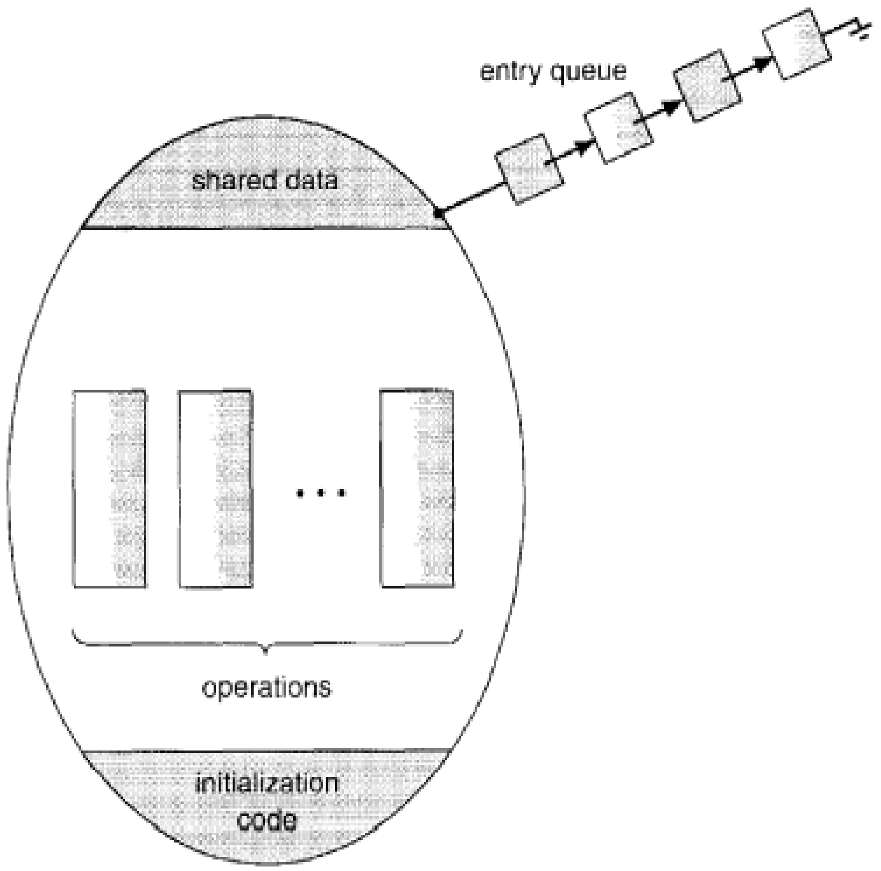


Figure 6.17 Schematic view of a monitor.