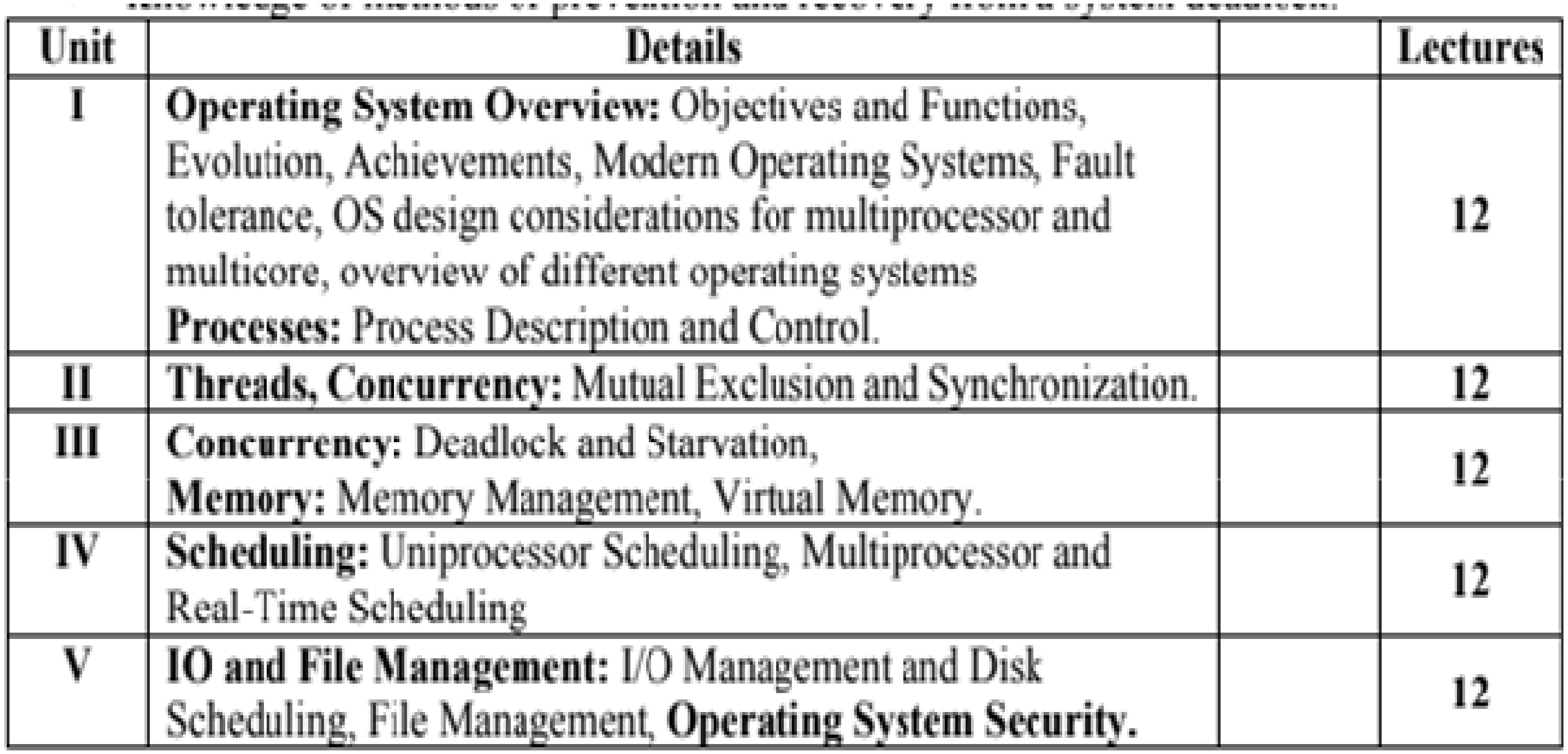


Unit	Details		Lectures
I	**Operating System Overview:** Objectives and Functions, Evolution, Achievements, Modern Operating Systems, Fault tolerance, OS design considerations for multiprocessor and multicore, overview of different operating systems **Processes:** Process Description and Control.		12
II	**Threads, Concurrency:** Mutual Exclusion and Synchronization.		12
III	**Concurrency:** Deadlock and Starvation, **Memory:** Memory Management, Virtual Memory.		12
IV	**Scheduling:** Uniprocessor Scheduling, Multiprocessor and Real-Time Scheduling		12
V	**IO and File Management:** I/O Management and Disk Scheduling, File Management, **Operating System Security.**		12

Thread or **lightweight process** created by process to execute multiple activities that are going on at once. Treads are mini processes or subprocesses working to contribute the main process.

Multithreading

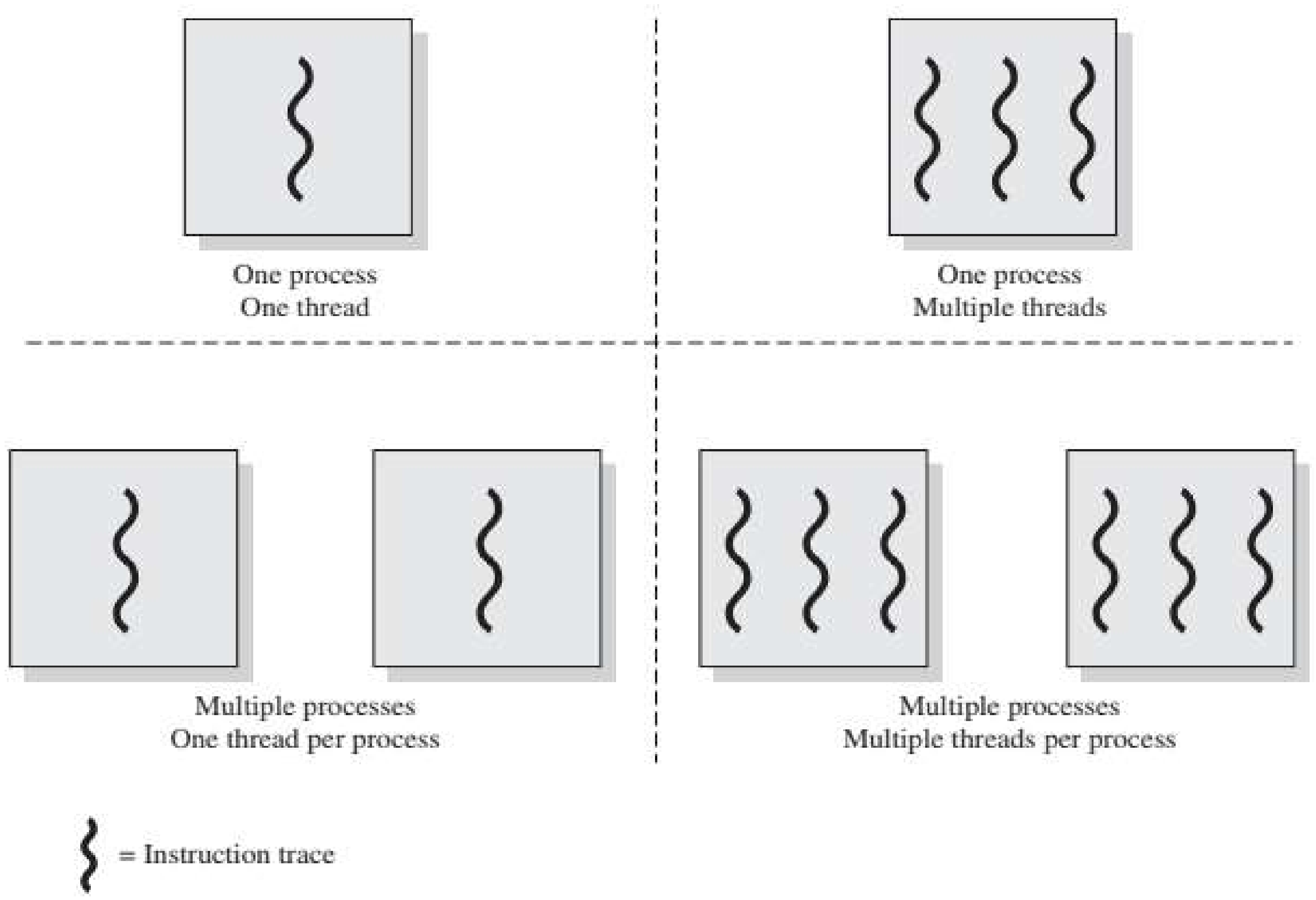
Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.

The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a **single-threaded approach**.

In a **multithreaded environment**, a process is defined as the unit of resource allocation and a unit of protection.

The following are associated with processes:

- A virtual address space that holds the process image
- Protected access to processors, other processes (for interprocess communication), files, and I/O resources (devices and channels)


One process
One thread
One process
Multiple threads
Multiple processes
One thread per process
Multiple processes
Multiple threads per process
= Instruction trace

All of the threads of a process share the state and resources of that process. They reside in the same address space and have access to the same data.
When one thread alters an item of data in memory, other threads see the results if and when they access that item. If one thread opens a file with read privileges, other threads in the same process can also read from that file.

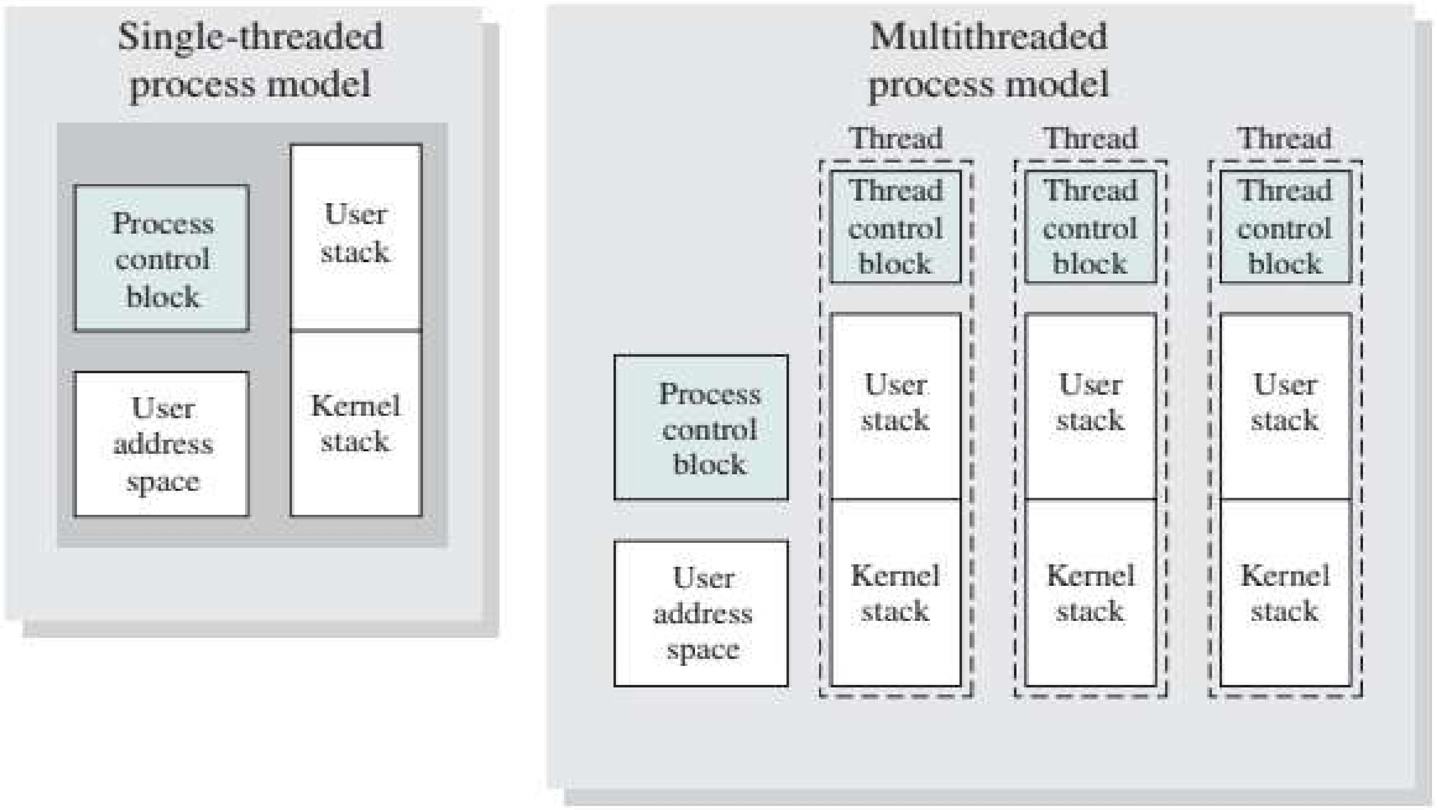


Figure 4.2 Single-Threaded and Multithreaded Process Models

Within a process, there may be one or more threads, each with the following:

• A thread execution state (Running, Ready, etc.)

• A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process

• An execution stack

• Some per-thread static storage for local variables

• Access to the memory and resources of its process, shared with all other threads in that process

The **key benefits of threads** derive from the performance implications:

1. It takes far less time to create a new thread in an existing process than to create a brand-new process.
2. It takes less time to terminate a thread than a process.
3. It takes less time to switch between two threads within the same process than to switch between processes.
4. Threads enhance efficiency in communication between different executing programs. In most operating systems, communication between independent processes requires the intervention of the kernel to provide protection and the mechanisms needed for communication. However, because threads within the same process share memory and files, they can communicate with each other without invoking the kernel.

Thread Functionality
Like processes, threads have execution states and may synchronize with one another.
THREAD STATES
As with processes, the key states for a thread are **Running, Ready, and Blocked**. Generally, it does not make sense to associate suspend states with threads because such states are process-level concepts. In particular, if a process is swapped out, all of its threads are necessarily swapped out because they all share the address space of the process.

The blocking of a thread do not results in the blocking of the entire process.

THREAD SYNCHRONIZATION

All of the threads of a process share the same address space and other resources, such as open files. Any alteration of a resource by one thread affects the environment of the other threads in the same process. It is therefore necessary to synchronize the activities of the various threads so that they do not interfere with each other or corrupt data structures.

SUMMARY

Some operating systems distinguish the concepts of process and thread, the former related to resource ownership and the latter related to program execution. This approach may lead to improved efficiency and coding convenience.

In a multithreaded system, multiple concurrent threads may be defined within a single process. This may be done using either user-level threads or kernel-level threads.

User-level threads are unknown to the OS and are created and managed by a threads library that runs in the user space of a process. User-level threads are very efficient because a mode switch is not required to switch from one thread to another. However, only a single user-level thread within a process can execute at a time, and if one thread blocks, the entire process is blocked.

SUMMARY

Kernel-level threads are threads within a process that are maintained by the kernel. Because they are recognized by the kernel, multiple threads within the same process can execute in parallel on a multiprocessor and the blocking of a thread does not block the entire process. However, a mode switch is required to switch from one thread to another.

[Two mode switches (user to kernel; kernel back to user).]

The central themes of operating system design are all concerned with the management of processes and threads:

- **Multiprogramming:** The management of multiple processes within a uniprocessor system
- **Multiprocessing:** The management of multiple processes within a multiprocessor
- **Distributed processing:** The management of multiple processes executing on multiple, distributed computer systems.

Fundamental to all of these areas, and fundamental to OS design, is concurrency.

Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files, and I/O access), synchronization of the activities of multiple processes, and allocation of processor time to processes.

Concurrency arises in three different contexts:

- **Multiple applications:** Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.
- **Structured applications:** As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.
- **Operating system structure:** The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.

Race Condition

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

Let us consider two simple examples.
suppose that **two processes, P1 and P2**, share the global variable **a**
At some point in its execution, **P1 updates a to the value 1**, and at some point in its execution, **P2 updates a to the value 2**.
Thus, the two tasks are in a race to write variable a . In this example, the “loser” of the race (the process that updates last) determines the final value of **a**

For our second example, consider **two process, P3 and P4**, that share global variables **b** and **c** , with initial values **b = 1** and **c = 2** .
At some point in its execution,
P3 executes the assignment b = b + c ,
and
at some point in its execution, P4 executes the assignment c = b + c .
Note that the two processes update different variables. However, the final values of the two variables depend on the order in which the two processes execute these two assignments.
If P3 executes its assignment statement first,
b=b+c //1+2=3=b
c=b+c //3+2=5=c
then the final values are b = 3 and c = 5 .
If P4 executes its assignment statement first, then the final values are b = 4 and c = 3 .

Operating System Concerns

Operating System Concerns What design and management issues are raised by the existence of concurrency? We can list the following concerns:

1. The OS must be able to **keep track** of the various processes. This is done with the use of **process control blocks.**
2. The OS must **allocate and deallocate** various **resources** for each active process. At times, multiple processes want access to the **same resource**.

These resources include

- **Processor time**: This is the scheduling function,
- **Memory**: Most operating systems use a virtual memory scheme.
- **Files**
- **I/O devices**

Operating System Concerns

3. The OS must **protect the data and physical resources** of each process against unintended interference by other processes.

Process Interaction

Processes **unaware** of each other: such processes exhibit **competition**

Processes **aware** of each other: such processes exhibit **cooperation**

Potential **Control Problems**

- **Mutual exclusion**
- **Deadlock**
- **Starvation**

Processes **aware** of each other also has **Data coherence along with the above control problems.**

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.

Deadlock A situation in which two or more processes are unable to proceed because each one is waiting for one of the others to do something.

Starvation A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Requirements for Mutual Exclusion

Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. **Mutual exclusion must be enforced:** Only one process at a time is allowed into its critical section, among all processes that have **critical sections** for the same resource or shared object.

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.

2. A process that halts in its **non critical section** must do so without interfering with other processes.

Requirements for Mutual Exclusion

3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

Approaches for implementing Mutual Exclusion

One approach is to **leave the responsibility with the processes** that wish to execute concurrently. Processes, whether they are system programs or application programs, would be required to coordinate with one another to enforce mutual exclusion, with no support from the programming language or the OS. But **this approach is prone to high processing overhead and bugs.**

A second approach involves the use of special-purpose machine instructions. These have the advantage of reducing overhead.

A third approach is to provide some level of support within the OS or a programming language.

MUTUAL EXCLUSION: HARDWARE SUPPORT
Several interesting hardware approaches to mutual exclusion.
Interrupt Disabling for uniprocessor systems only
Special Machine Instructions for multiprocessor system

SEMAPHORES
The OS and programming language **mechanisms** that are used to provide concurrency.

Common Concurrency Mechanisms

Semaphore

An **integer value** used for signaling among processes. Only **three operations** may be performed on a semaphore, all of which are atomic: **initialize**, **decrement, and increment**. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore.

Binary Semaphore

A semaphore that takes on only the values 0 and 1.

Mutex

Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined:

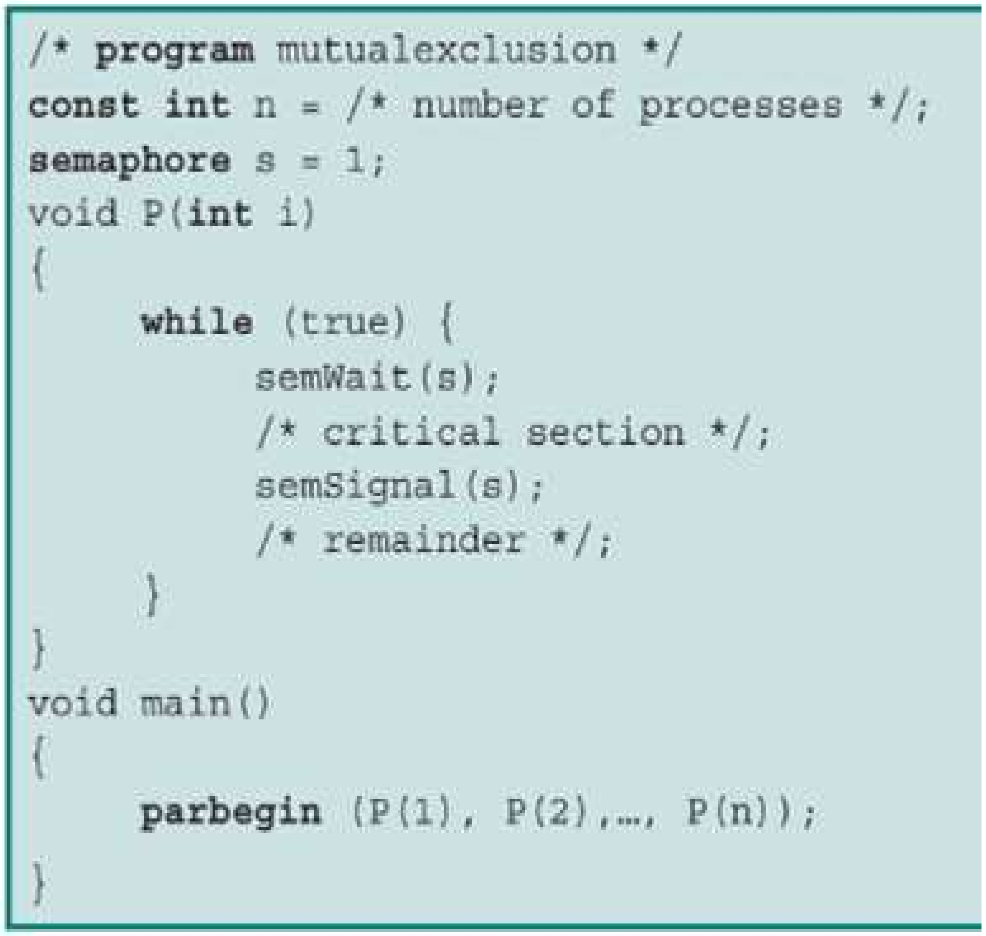
1. A semaphore may be **initialized** to a **nonnegative integer value**.
2. The **semWait** operation **decrements** the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.
3. The **semSignal** operation **increments** the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

Other than these three operations, there is no way to inspect or manipulate semaphores.

A binary semaphore may only take on the values 0 and 1 and can be defined by the following three operations:

1. A binary semaphore may be **initialized** to 0 or 1.
2. The **semWaitB** operation checks the semaphore value. If the value is zero, then the process executing the semWaitB is blocked. If the value is one, then the value is changed to zero and the process continues execution.
3. The **semSignalB** operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.

Mutual Exclusion Using Semaphores



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

Mutual Exclusion Using Semaphores

Figure shows a straightforward solution to the mutual exclusion problem using a semaphore **s.** Consider n processes, identified in the array **P (i)**, all of which need access to the same resource. Each process has a critical section used to access the resource.

In each process, a semWait(s) is executed just before its critical section. If the value of s becomes negative, the process is blocked.

If the value is 1, then it is decremented to 0 and the process immediately enters its critical section; because s is no longer positive, no other process will be able to enter its critical section.

Mutual Exclusion Using Semaphores

The semaphore is initialized to 1. Thus, the first process that executes a semWait will be able to enter the critical section immediately, setting the value of s to 0. Any other process attempting to enter the critical section will find it busy and will be blocked, setting the value of s to –1. Any number of processes may attempt entry; each such unsuccessful attempt results in a further decrement of the value of s . When the process that initially entered its critical section departs, s is incremented and one of the blocked processes (if any) is removed from the queue of blocked processes associated with the semaphore and put in a Ready state. When it is next scheduled by the OS, it may enter the critical section.

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 */;
       }
}
```

MONITORS

Semaphores provide a primitive yet powerful and flexible tool for enforcing mutual exclusion and for coordinating processes.

It may be **difficult to produce a correct program** using semaphores. The difficulty is that semWait and semSignal operations may be scattered throughout a program and it is **not easy to see the overall effect** of these operations on the semaphores they affect.

The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.

The monitor construct has been **implemented** in a number of programming languages, including **Concurrent Pascal**, **Pascal-Plus**, **Modula-2**, **Modula-3**, and **Java**. It has also been implemented as a program library.

This allows programmers to put a monitor lock on any object.

In particular, for something like a linked list, you may want to lock all linked lists with one lock, or have one lock for each list, or have one lock for each element of each list.

Monitor with Signal

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

The **chief characteristics** of a monitor are the following:

1. The local data variables are accessible only by the monitor’s procedures and not by any external procedure.
2. A process enters the monitor by invoking one of its procedures.
3. Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available.

A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor.

Condition variables are a special data type in monitors, which are operated on by two functions:

• **cwait(c) :** Suspend execution of the calling process on condition c . The monitor is now available for use by another process.

• **csignal(c) :** Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

Note that monitor wait and signal operations are different from those for the semaphore. If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.

MESSAGE PASSING

When processes interact with one another, two fundamental requirements must be satisfied: **synchronization and communication**. Processes need to be synchronized to enforce mutual exclusion; cooperating processes may need to exchange information. **One approach to providing both of these functions is message passing.**

Message passing has the further advantage that it lends itself to implementation in distributed systems as well as in shared-memory multiprocessor and uniprocessor systems.

Table 5.5 Design Characteristics of Message Systems for Interprocess Communication and Synchronization

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queueing Discipline
- FIFO
- Priority

READERS/WRITERS PROBLEM

In dealing with the design of synchronization and concurrency mechanisms, it is useful to be able to relate the problem at hand to known problems and to be able to test any solution in terms of its ability to solve these known problems.

The readers/writers problem is defined as follows: There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers.

READERS/WRITERS PROBLEM …cont.

There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

The conditions that must be satisfied are as follows:

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.

Thus, readers are processes that are not required to exclude one another and writers are processes that are required to exclude all other processes, readers and writers alike.

Summary

Concurrency is the ability of your program to execute multiple tasks at the same time, or in an overlapping manner.

Concurrency can be achieved by using multiple processors, cores, or threads, depending on the level of parallelism that you want to achieve.

Synchronization is the coordination of your concurrent tasks, to ensure that they do not interfere with each other, or access shared resources in an inconsistent way.

A system typically consists of **severa**l (perhaps hundreds or even thousands) of **threads running either concurrently or in paralle**l. Threads often share user data. Meanwhile, the operating system continuously updates various data structures to support multiple threads. A **race condition** exists **when access to shared data is not controlled**, possibly resulting in corrupt data values.

Process synchronization involves using tools that control access to shared data to avoid race conditions.Synchronization can be achieved by using various mechanisms such as mutex, semaphores, monitors, or message passing depending on the type and complexity of your concurrency model. These tools must be used carefully, as their incorrect use can result in poor system performance, including deadlock.