Introduction:

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. The recent proliferation of clusters is a prime example of this type of system.

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.

### Key Terms:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic Operation</td>
<td>A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.</td>
</tr>
<tr>
<td>Critical Section</td>
<td>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.</td>
</tr>
<tr>
<td>Deadlock</td>
<td>A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.</td>
</tr>
<tr>
<td>Livelock</td>
<td>A situation in which two or more processes continuously change their states in response to changes in the other process without doing any useful work.</td>
</tr>
<tr>
<td>Mutual Exclusion</td>
<td>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.</td>
</tr>
<tr>
<td>Race Condition</td>
<td>A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.</td>
</tr>
<tr>
<td>Starvation</td>
<td>A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.</td>
</tr>
</tbody>
</table>
Principles of Concurrency:

In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution (Figure 2.12a). Even though actual parallel processing is not achieved, and even though there is a certain amount of overhead involved in switching back and forth between processes, interleaved execution provides major benefits in processing efficiency and in program structuring. In a multiple-processor system, it is possible not only to interleave the execution of multiple processes but also to overlap them (Figure 2.12b).

At first glance, it may seem that interleaving and overlapping represent fundamentally different modes of execution and present different problems. In fact, both techniques can be viewed as examples of concurrent processing, and both present the same problems. In the case of a uniprocessor, the problems stem from a basic characteristic of multiprogramming systems: The relative speed of execution of processes cannot be predicted. It depends on the activities of other processes, the way in which the OS handles interrupts, and the scheduling policies of the OS. The following difficulties arise:

1. **The sharing of global resources is fraught with peril.** For example, if two processes both make use of the same global variable and both perform reads and writes on that variable, then the order in which the various reads and writes are executed is critical. An example of this problem is shown in the following subsection.

2. **It is difficult for the OS to manage the allocation of resources optimally.** For example, process A may request use of, and be granted control of, a particular I/O channel and then be suspended before using that channel. It may be undesirable for the OS simply to lock the channel and prevent its use by other processes; indeed this may lead to a deadlock condition.

3. **It becomes very difficult to locate a programming error because results are typically not deterministic and reproducible.**

Consider the following procedure:

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

This procedure shows the essential elements of a program that will provide a character echo procedure; input is obtained from a keyboard one keystroke at a time. Each input character is stored in variable chin. It is then transferred to variable chout and sent to the display. Any program can call this procedure repeatedly to accept user input and display it on the user’s screen.

Consider the following sequence:

1. Process P1 invokes the echo procedure and is interrupted immediately after getchar returns its value and stores it in chin. At this point, the most recently entered character, x, is stored in variable chin.
2. Process P2 is activated and invokes the echo procedure, which runs to conclusion, inputting and then displaying a single character, y, on the screen.

3. Process P1 is resumed. By this time, the value x has been overwritten in chin and therefore lost. Instead, chin contains y, which is transferred to chout and displayed.

Thus, the first character is lost and the second character is displayed twice. The essence of this problem is the shared global variable, chin. Multiple processes have access to this variable. If one process updates the global variable and then is interrupted, another process may alter the variable before the first process can use its value. Suppose, however, that we permit only one process at a time to be in that procedure. Then the foregoing sequence would result in the following:

1. Process P1 invokes the echo procedure and is interrupted immediately after the conclusion of the input function. At this point, the most recently entered character, x, is stored in variable chin.

2. Process P2 is activated and invokes the echo procedure. However, because P1 is still inside the echo procedure, although currently suspended, P2 is blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the echo procedure.

3. At some later time, process P1 is resumed and completes execution of echo. The proper character, x, is displayed.

4. When P1 exits echo, this removes the block on P2. When P2 is later resumed, the echo procedure is successfully invoked.

This example shows that it is necessary to protect shared global variables (and other shared global resources) and that the only way to do that is to control the code that accesses the variable. If we impose the discipline that only one process at a time may enter echo and that once in echo the procedure must run to completion before it is available for another process, then the type of error just discussed will not occur. How that discipline may be imposed is a major topic of this chapter. This problem was stated with the assumption that there was a single-processor, multiprogramming OS. The example demonstrates that the problems of concurrency occur even when there is a single processor. In a multiprocessor system, the same problems of protected shared resources arise, and the same solution works. First, suppose that there is no mechanism for controlling access to the shared global variable:

1. Processes P1 and P2 are both executing, each on a separate processor. Both processes invoke the echo procedure.

2. The following events occur; events on the same line take place in parallel.

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>• chin = getchar();</td>
<td>• chin = getchar();</td>
</tr>
<tr>
<td>• chout = chin;</td>
<td>• chout = chin;</td>
</tr>
<tr>
<td>• putchar(chout);</td>
<td>• putchar(chout);</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
The result is that the character input to P1 is lost before being displayed, and the character input to P2 is displayed by both P1 and P2. Again, let us add the capability of enforcing the discipline that only one process at a time may be in echo. Then the following sequence occurs:

1. Processes P1 and P2 are both executing, each on a separate processor. P1 invokes the echo procedure.
2. While P1 is inside the echo procedure, P2 invokes echo. Because P1 is still inside the echo procedure (whether P1 is suspended or executing), P2 is blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the echo procedure.
3. At a later time, process P1 completes execution of echo, exits that procedure, and continues executing. Immediately upon the exit of P1 from echo, P2 is resumed and begins executing echo.

**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.

**Example 1**:

Two Processes P1 and P2 shared a global variable ‘a’. 
P1 updates ‘a’ to the value 1 and P2 update ‘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’.

**Example 2**:

Two Processes P3 and P4 shared a global variable ‘b’ and ‘c’.
Initial value B = 1 and c = 2.
P3 execute the assignment b = b + c.
P4 execute the assignment c = b + c.
If P3 executes its assignment statement first, 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.

**Process Interaction**:

We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other’s existence. Table 5.2 lists three possible degrees of awareness plus the consequences of each:
Table: 5.2 Process Interactions.

<table>
<thead>
<tr>
<th>Degree of Awareness</th>
<th>Relationship</th>
<th>Influence That One Process Has on the Other</th>
<th>Potential Control Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes unaware of each other</td>
<td>Competition</td>
<td>• Results of one process independent of the action of others</td>
<td>• Mutual Exclusion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Timing of process may be affected</td>
<td>• Deadlock</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Starvation</td>
</tr>
<tr>
<td>Processes indirectly aware of each other (e.g., shared object)</td>
<td>Cooperation by sharing</td>
<td>• Results of one process may depend on information obtained from others</td>
<td>• Mutual Exclusion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Timing of process may be affected</td>
<td>• Deadlock</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Starvation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Data coherence</td>
</tr>
<tr>
<td>Processes directly aware of each other (have communication Primitives available to them)</td>
<td>Cooperation by communication</td>
<td>• Results of one process may depend on information obtained from others</td>
<td>• Deadlock</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Timing of process may be affected</td>
<td>• Starvation</td>
</tr>
</tbody>
</table>

- **Processes unaware of each other**: These are independent processes that are not intended to work together. The best example of this situation is the multiprogramming of multiple independent processes. These can either be batch jobs or interactive sessions or a mixture. Although the processes are not working together, the OS needs to be concerned about competition for resources. For example, two independent applications may both want to access the same disk or file or printer. The OS must regulate these accesses.

- **Processes indirectly aware of each other**: These are processes that are not necessarily aware of each other by their respective process IDs but that share access to some object, such as an I/O buffer. Such processes exhibit cooperation in sharing the common object.

- **Processes directly aware of each other**: These are processes that are able to communicate with each other by process ID and that are designed to work jointly on some activity. Again, such processes exhibit cooperation.

---

**Competition among Processes for Resources**

Concurrent processes come into conflict with each other when they are competing for the use of the same resource. In its pure form, we can describe the situation as follows. Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of other processes, and each is to be unaffected by the execution of the other processes. It follows from this that each process should leave the state of any resource that it uses unaffected. Examples of resources include I/O devices,
memory, processor time, and the clock. There is no exchange of information between the competing processes. However, the execution of one process may affect the behavior of competing processes. In particular, if two processes both wish access to a single resource, then one process will be allocated that resource by the OS, and the other will have to wait. Therefore, the process that is denied access will be slowed down. In an extreme case, the blocked process may never get access to the resource and hence will never terminate successfully.

In the case of competing processes three control problems must be faced. First is the need for **mutual exclusion**. Suppose two or more processes require access to a single non-sharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. We will refer to such a resource as a **CRITICAL RESOURCE** and the portion of the program that uses it a **CRITICAL SECTION** of the program. **It is important that only one program at a time be allowed in its critical section.** We cannot simply rely on the OS to understand and enforce this restriction because the detailed requirements may not be obvious. In the case of the printer, for example, we want any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.

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

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

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

**Figure 5.1**
Illustration of Mutual Exclusion

The enforcement of mutual exclusion creates two additional control problems. One is that of **deadlock**. For example, consider two processes, P1 and P2, and two resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation: the OS assigns R1 to P2, and R2
to P1. Each process is waiting for one of the two resources. Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. **The two processes are deadlocked.**

A final control problem is **starvation**. Suppose that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the OS grants access to P3 and that P1 again requires access before P3 completes its critical section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

There are \( n \) processes to be executed concurrently. **Each process includes**
1. A critical section that operates on some resource \( Ra \), and
2. Additional code preceding and following the critical section that does not involve access to \( Ra \).

Because all processes access the same resource \( Ra \), it is desired that only one process at a time be in its critical section. To enforce mutual exclusion, two functions are provided: **ENTERCRITICAL** and **EXITCRITICAL**. Each function takes as an argument the name of the resource that is the subject of competition. Any process that attempts to enter its critical section while another process is in its critical section, for the same resource, is made to wait.

** Cooperation among Processes by Sharing**

The case of cooperation by sharing covers processes that interact with other processes without being explicitly aware of them. For example, multiple processes may have access to shared variables or to shared files or databases. Processes may use and update the shared data without reference to other processes but know that other processes may have access to the same data. Thus the processes must cooperate to ensure that the data they share are properly managed. The control mechanisms must ensure the integrity of the shared data. Because data are held on resources (devices, memory), the control problems of mutual exclusion, deadlock, and starvation are again present. The only difference is that data items may be accessed in two different modes, reading and writing, and only writing operations must be mutually exclusive. However, over and above these problems, a new requirement is introduced: that of data coherence. As a simple example, consider a bookkeeping application in which various data items may be updated. Suppose two items of data \( a \) and \( b \) are to be maintained in the relationship \( a=b \). That is, any program that updates one value must also update the other to maintain the relationship.

Now consider the following two processes:

P1:
- \( a = a + 1; \)
- \( b = b + 1; \)

P2:
- \( b = 2 * b; \)
- \( a = 2 * a; \)
If the state is initially consistent, each process taken separately will leave the shared data in a consistent state. Now consider the following concurrent execution sequence, in which the two processes respect mutual exclusion on each individual data item (a and b):

\[
\begin{align*}
    a &= a + 1; \\
    b &= 2 \times b; \\
    b &= b + 1; \\
    a &= 2 \times a;
\end{align*}
\]

At the end of this execution sequence, the condition \(a = b\) no longer holds. For example, if we start with \(a = b = 1\), at the end of this execution sequence we have \(a=4\) and \(b=3\). The problem can be avoided by declaring the entire sequence in each process to be a critical section. Thus we see that the concept of critical section is important in the case of cooperation by sharing. The same abstract functions of `entercritical` and `exitcritical` discussed earlier (Figure 5.1) can be used here. In this case, the argument for the functions could be a variable, a file, or any other shared object. Furthermore, if critical sections are used to provide data integrity, then there may be no specific resource or variable that can be identified as an argument. In that case, we can think of the argument as being an identifier that is shared among concurrent processes to identify critical sections that must be mutually exclusive.

**Cooperation among Processes by Communication**

In the first two cases that we have discussed, each process has its own isolated environment that does not include the other processes. The interactions among processes are indirect.

- **In both cases, there is a sharing.**
  1. In the case of competition, they are sharing resources without being aware of the other processes.
  2. In the second case, they are sharing values, and although each process is not explicitly aware of the other processes, it is aware of the need to maintain data integrity.

When processes cooperate by communication, however, the various processes participate in a common effort that links all of the processes. The communication provides a way to synchronize, or coordinate, the various activities.

Typically, communication can be characterized as consisting of messages of some sort. Primitives for sending and receiving messages may be provided as part of the programming language or provided by the OS kernel.

Because nothing is shared between processes in the act of passing messages, mutual exclusion is not a control requirement for this sort of cooperation. However, the problems of deadlock and starvation are still present.

**As an example of deadlock**, two processes may be blocked, each waiting for a communication from the other. **As an example of starvation**, consider three processes, P1, P2, and P3 that exhibit the following behavior. P1 is repeatedly attempting to communicate with either P2 or P3, and P2 and P3 are both attempting to communicate with P1. A sequence could arise in which P1 and P2 exchange information repeatedly, while P3 is
blocked waiting for a communication from P1. There is no deadlock, because P1 remains active, but P3 is starved.

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

**Mutual Exclusion: Hardware Support**

A number of software algorithms for enforcing mutual exclusion have been developed. The software approach is likely to have high processing overhead and the risk of logical errors is significant. However, a study of these algorithms illustrates many of the basic concepts and potential problems in developing concurrent programs. For the interested reader, Appendix A includes a discussion of software approaches. In this section, we look at several interesting hardware approaches to mutual exclusion.

**Interrupt Disabling**:

In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved. Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts. A process can then enforce mutual exclusion in the following way:

```c
while (true)
{
    /* disable interrupts */;
    /* CRITICAL SECTION */;
    /* enable interrupts */;
    /* remainder */;
}
```

*Because the critical section cannot be interrupted, mutual exclusion is guaranteed.*

The price of this approach, however, is high. The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes. A second problem is that this approach will not work in a multiprocessor architecture. When the
computer includes more than one processor, it is possible (and typical) for more than one process to be executing at a time. In this case, disabled interrupts do not guarantee mutual exclusion.

**Special Machine Instructions**

In a multiprocessor configuration, several processors share access to a common main memory. In this case, there is not a master/slave relationship; rather the processors behave independently in a peer relationship. There is no interrupt mechanism between processors on which mutual exclusion can be based. At the hardware level, as was mentioned, access to a memory location excludes any other access to that same location. With this as a foundation, processor designers have proposed several machine instructions that carry out two actions atomically, such as reading and writing or reading and testing, of a single memory location with one instruction fetch cycle. During execution of the instruction, access to the memory location is blocked for any other instruction referencing that location. In this section, we look at two of the most commonly implemented instructions.

1. **Compare & Swap Instruction**

   The compare & swap instruction, also called a compare and exchange instruction, can be defined as follows:

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

   This version of the instruction checks a memory location (*word) against a test value (testval). If the memory location's current value is testval, it is replaced with newval; otherwise it is left unchanged. The old memory value is always returned; thus, the memory location has been updated if the returned value is the same as the test value. This atomic instruction therefore has two parts: A compare is made between a memory value and a test value; if the values differ a swap occurs. The entire compare & swap function is carried out atomically; that is, it is not subject to interruption.

   Another version of this instruction returns a Boolean value: true if the swap occurred; false otherwise. Figure 5.2a shows a mutual exclusion protocol based on the use of this instruction. A shared variable bolt is initialized to 0. The only process that may enter its critical section is one that finds bolt equal to 0. All other processes at enter their critical section go into a busy waiting mode. The term busy waiting, or spin waiting, refers to a technique in which a process can do nothing until it gets permission to enter its critical section but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance. When a process leaves its critical section, it resets bolt to 0; at this point one and only one of the waiting processes is granted access to its critical section. The choice of process depends on which process happens to execute the compare & swap instruction next.
/* program mutual exclusion */

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

void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ... ,P(n));
}

(a) Compare & Swap Instruction

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

void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}

(b) Exchange Instruction

Figure 5.2 Hardware Support for Mutual Exclusion

2. Exchange Instruction

The exchange instruction can be defined as follows:

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

The instruction exchanges the contents of a register with that of a memory location. Both the Intel IA-32 architecture (Pentium) and the IA-64 architecture (Itanium) contain an XCHG instruction. Figure 5.2b shows a mutual exclusion protocol based on the use of an exchange instruction. A shared variable bolt is initialized to 0. Each process uses a local variable key that is initialized to 1. The only process that may enter its critical section is one that finds bolt equal to 0. It excludes all other processes from the critical section by setting bolt to 1. When a process leaves its critical section, it resets bolt to 0, allowing another process to gain access to its critical section.
The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals. For signaling, special variables called semaphores are used. To transmit a signal via semaphore s, a process executes the primitive \texttt{semSignal(s)}. To receive a signal via semaphore s, a process executes the primitive \texttt{semWait(s)}; if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

- 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 \texttt{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 \texttt{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.

We explain these operations as follows. To begin, the semaphore has a zero or positive value. \textbf{If the value is positive}, that value equals the number of processes that can issue a wait and immediately continue to execute. \textbf{If the value is zero}, either by initialization or because a number of processes equal to the initial semaphore value have issued a wait, the next process to issue a wait is blocked, and the semaphore value goes negative. Each subsequent wait drives the semaphore value further into minus territory. The negative value equals the number of processes waiting to be unblocked. Each \texttt{semSignal} unblocks one of the waiting processes when the semaphore value is negative.

- \textbf{Points out three interesting consequences of the semaphore definition:}
  1. In general, there is no way to know before process decrements a semaphore whether it will block or not.
  2. After process increments a semaphore and another process get woken up, both processes continue running concurrently. There is no way to know which process, if either will continue immediately on a uniprocessor system.
  3. When you signal a semaphore, you don’t necessarily know whether another process is waiting, so the number of unblocked processes may be zero or one.

Figure 5.3 suggests a more formal definition of the primitives for semaphores. The \texttt{semWait} and \texttt{semSignal} primitives are assumed to be atomic.
A more restricted version, known as the **binary semaphore**, is defined in Figure 5.4. 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 zero). 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.

In principle, it should be easier to implement the binary semaphore, and it can be shown that it has the same expressive power as the general semaphore. To contrast the two types of semaphores, the non binary semaphore is often referred to as either a **counting semaphore** or a **general semaphore**.

A concept related to the binary semaphore is the **mutex**. 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). In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.
struct binary semaphore
{
    enum {zero, one} value;
    queueType queue;
};

void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else
    {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else
    {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}

Figure 5.3: A DEFINITION OF BINARY SEMAPHORE PRIMITIVES.

For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore. The question arises of the order in which processes are removed from such a queue. The fairest removal policy is first-in-first-out (FIFO): The process that has been blocked the longest is released from the queue first; a semaphore whose definition includes this policy is called a strong semaphore. A semaphore that does not specify the order in which processes are removed from the queue is a weak semaphore.

Here processes A, B, and C depend on a result from process D. Initially (1), A is running; B, C, and D are ready; and the semaphore count is 1, indicating that one of D’s results is available. When A issues a semWait instruction on semaphore s, the semaphore decrements to 0, and A can continue to execute; subsequently it rejoins the ready queue. Then B runs (2), eventually issues a semWait instruction, and is blocked, allowing D to run (3). When D completes a new result, it issues a semSignal instruction, which allows B to move to the ready queue (4). D rejoins the ready queue and C begins to run (5) but is blocked when it issues a semWait instruction. Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution (6). When D has a result, it issues a semSignal, which transfers C to the ready queue. Later cycles of D will release A and B from the Blocked state.

Æ PRODUCER / CONSUMER PROBLEM Æ
We now examine one of the most common problems faced in concurrent processing: the producer/consumer problem. The general statement is this: there are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer. We will look at a number of solutions to this problem to illustrate both the power and the pitfalls of semaphores.

To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

**Producer:**
```c
while (true) {
    /* Produce item v */
    b[in] = v;
    in++;
}
```

**Consumer:**
```c
while (true) {
    while (in <= out) /* do nothing */
    w = b[out];
    out++;
    /* consume item w */
}
```

Figure 5.8 illustrates the structure of buffer b. The producer can generate items and store them in the buffer at its own pace. Each time, an index (in) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Thus the consumer makes sure that the producer has advanced beyond it (in > out) before proceeding.

This solution seems rather straightforward. The producer is free to add to the buffer at any time. It performs `semWaitB(s)` before appending and `semSignalB(s)` afterward to prevent the consumer or any other producer from accessing the buffer during the append operation. Also, while in the critical section, the producer increments the value of n. If n = 1, then the buffer was empty just prior to this append, so the producer performs `semSignalB(delay)` to alert the consumer of this fact. The consumer begins by waiting for the first item to be produced, using `semWaitB(delay)`. It then takes an item and decrements n in its critical section. If the producer is able to stay ahead of the consumer (a common situation), then the
consumer will rarely block on the semaphore delay because \( n \) will usually be positive. Hence both producer and consumer run smoothly.

\[
\begin{align*}
/* & program producer\_consumer */
\int n; 
binary\_semaphore s = 1, delay = 0;
void producer()
\{}
\quad while (true)
\quad \quad \{ 
\quad \quad produce();
\quad \quad semWaitB(s); 
\quad \quad append();
\quad \quad n++;
\quad \quad if (n==1) semSignalB(delay);
\quad \quad semSignalB(s);
\quad \}
\}

void consumer()
\{}
\quad int m; /* a local variable */
\quad semWaitB(delay);
\quad while (true)
\quad \quad \{ 
\quad \quad semWaitB(s);
\quad \quad take();
\quad \quad n--;
\quad \quad m = n;
\quad \quad semSignalB(s);
\quad \quad consume();
\quad \quad if (m==0) semWaitB(delay);
\quad \}
\}

void main()
\{}
\quad n = 0;
\quad parbegin (producer, consumer);
\}
\]

Figure 5.10 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

Finally, let us add a new and realistic restriction to the producer/consumer problem: namely, that the buffer is bounded. The buffer is treated as a circular storage (Figure 5.12), and pointer values must be expressed modulo the size of the buffer.

\[
\begin{align*}
\text{producer:} & \quad \text{consumer:} \\
\quad \text{while (true)} \{} & \quad \text{while (true)} \{}
\quad \quad /* produce item v */ & \quad \quad /* do nothing */
\quad \quad \text{while } ((in + 1) \% n == out) & \quad \quad \text{while } (in == out)
\quad \quad /* do nothing */; & \quad \quad /* do nothing */;
\quad \quad b[in] = v; & \quad \quad w = b[out];
\quad \quad in = (in + 1) \% n; & \quad \quad out = (out + 1) \% n;
\quad \}
\]

Prepared By :- Ajay A. Ardeshana
Email :- ajay.24021985@gmail.com
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;

void producer()
{
    while (true)
    {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}

void consumer()
{
    while (true)
    {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}

void main()
{
    parbegin (producer, consumer);
}

Figure 5.11 A Correct Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

MONITORS

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.

The first two characteristics are reminiscent of those for objects in object-oriented software. Indeed, an object-oriented OS or programming language can readily implement a monitor as an object with special characteristics. By enforcing the discipline of one process at a time, the monitor is able to provide a mutual exclusion facility. The data variables in the monitor can be accessed by only one process at a time. Thus, a shared data structure can be protected by placing it in a monitor. If the data in a monitor represent some resource, then the monitor provides a mutual exclusion facility for accessing the resource.

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.

Figure 5.15 illustrates the structure of a monitor. Although a process can enter the monitor by invoking any of its procedures, we can think of the monitor as having a single entry point that is guarded so that only one process may be in the monitor at a time. Other processes that attempt to enter the monitor join a queue of processes blocked waiting for monitor availability. Once a process is in the monitor, it may temporarily block itself on condition x by issuing cwait(x); it is then placed in a queue of processes waiting to reenter the monitor when the condition changes, and resume execution at the point in its program following the cwait(x) call. If a process that is executing in the monitor detects a change in the condition variable x, it issues csignal(x), which alerts the corresponding condition queue that the condition has changed. As an example of the use of a monitor, let us return to the bounded-buffer producer/consumer problem. Figure 5.16 shows a solution using a monitor.
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N]; /* space for N items */
int nextin, nextout; /* buffer pointers */
int count; /* number of items in buffer */
cond notfull, notempty; /* condition variables for synchronization */
void append (char x) {
    if (count == N) cwait(notfull); /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal (notempty); /* resume any waiting consumer */
} 

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

/* monitor body */
nextin = 0; nextout = 0; count = 0; /* buffer initially empty */

void producer() {
    char x;
    while (true)
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. Message-passing systems come in many forms. In this section, we provide a general introduction that discusses features typically found in such systems. The actual function of message passing is normally provided in the form of a pair of primitives:

\[
\text{send (destination, message)} \\
\text{receive (source, message)}
\]

This is the minimum set of operations needed for processes to engage in message passing. A process sends information in the form of a message to another process designated by a destination. A process receives information by executing the receive primitive, indicating the source and the message.

- **Destination** is the address of the process where the message has to send.
- **Source** is the address of the process from where the message is coming.

**Synchronization** :-

The communication of a message between two processes implies some level of synchronization between the two: the receiver cannot receive a message until it has been sent by another process. In addition, we need to specify what happens to a process after it issues a send or receive primitive. Consider the send primitive first.

**When a send primitive is executed in a process, there are two possibilities:**
1. The sending process is blocked until the message is received
2. The sending process is unblocked until the message is received.

**When a process issues a receive primitive, there are two possibilities:**
1. If a message has previously been sent, the message is received and execution continues.

2. If there is no waiting message, then either (a) the process is blocked until a message arrives, or (b) the process continues to execute, abandoning the attempt to receive.

Thus, both the sender and receiver can be blocking or non-blocking. Three combinations are common, although any particular system will usually have only one or two combinations implemented:

**Blocking send, blocking receive**: Both the sender and receiver are blocked until the message is delivered; this is sometimes referred to as a *rendezvous*. This combination allows for tight synchronization between processes.

**Non blocking send, blocking receive**: Although the sender may continue on, the receiver is blocked until the requested message arrives. This is probably the most useful combination. It allows a process to send one or more messages to a variety of destinations as quickly as possible. A process that must receive a message before it can do useful work needs to be blocked until such a message arrives. An example is a server process that exists to provide a service or resource to other processes.

**Non blocking send, non blocking receive**: Neither party is required to wait.

### Addressing:

Clearly, it is necessary to have a way of specifying in the send primitive which process is to receive the message. Similarly, most implementations allow a receiving process to indicate the source of a message to be received. The various schemes for specifying processes in send and receive primitives fall into two categories: direct addressing and indirect addressing.

With **direct addressing**, the send primitive includes a specific identifier of the destination process. The receive primitive can be handled in one of two ways. One possibility is to require that the process explicitly designate a sending process. Thus, the process must know ahead of time from which process a message is expected. This will often be effective for cooperating concurrent processes. In other cases, however, it is impossible to specify the anticipated source process. An example is a printer-server process, which will accept a print request message from any other process. For such applications, a more effective approach is the use of implicit addressing. In this case, the source parameter of the receive primitive possesses a value returned when the receive operation has been performed.

The other general approach is **indirect addressing**. In this case, messages are not sent directly from sender to receiver but rather are sent to a shared data structure consisting of queues that can temporarily hold messages. Such queues are generally referred to as *mailboxes*. Thus, for two processes to communicate, one process sends a message to the appropriate mailbox and the other process picks up the message from the mailbox.

Strength of the use of indirect addressing is that, by decoupling the sender and receiver, it allows for greater flexibility in the use of messages. The relationship
between senders and receivers can be one-to-one, many-to-one, one-to-many, or many-to-many (Figure 5.18).

![Diagram](image)

**Figure 5.18 Indirect Process Communication**

A **one-to-one** relationship allows a private communications link to be set up between two processes. This insulates their interaction from erroneous interference from other processes.

A **many-to-one** relationship is useful for client/server interaction; one process provides service to a number of other processes. In this case, the mailbox is often referred to as a port.

A **one-to-many** relationship allows for one sender and multiple receivers; it is useful for applications where a message or some information is to be broadcast to a set of processes.

A **many-to-many** relationship allows multiple server processes to provide concurrent service to multiple clients.

**Message Format:**

The format of the message depends on the objectives of the messaging facility and whether the facility runs on a single computer or on a distributed system. For some operating systems, designers have preferred short, fixed-length messages to minimize processing and storage overhead. If a large amount of data is to be passed, the data can be placed in a file and the message then simply references that file. A more flexible approach is to allow variable-length messages.
Figure 5.19 shows a typical message format for operating systems that support variable-length messages. The message is divided into two parts: a header, which contains information about the message, and a body, which contains the actual contents of the message. The header may contain an identification of the source and intended destination of the message, a length field, and a type field to discriminate among various types of messages. There may also be additional control information, such as a pointer field so that a linked list of messages can be created; a sequence number, to keep track of the number and order of messages passed between source and destination; and a priority field.

READERS/WRITER 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. In the literature, several problems have assumed importance and appear frequently, both because they are examples of common design problems and because of their educational value. One such problem is the producer/consumer problem, which has already been explored. In this section, we look at another classic problem: the readers/writers problem.

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. 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. Before proceeding, let us distinguish this problem from two others: the general mutual exclusion problem and the producer/consumer problem. In the readers/writers problem readers do not also write to the data area, nor do writers read the data area while writing. A more general case, which includes this case, is to allow any of the processes to read or write the data area. In that case, we can declare any portion of a process that accesses the data area to be a critical section and impose the general mutual exclusion solution. The reason for being concerned with the more restricted case is that more efficient solutions are possible for this case and that the less efficient solutions to the general problem are unacceptably slow.
For example, suppose that the shared area is a library catalog. Ordinary users of the library read the catalog to locate a book. One or more librarians are able to update the catalog. In the general solution, every access to the catalog would be treated as a critical section, and users would be forced to read the catalog one at a time. This would clearly impose intolerable delays. At the same time, it is important to prevent writers from interfering with each other and it is also required to prevent reading while writing is in progress to prevent the access of inconsistent information.

Can the producer/consumer problem be considered simply a special case of the readers/writers problem with a single writer (the producer) and a single reader (the consumer)? The answer is no. The producer is not just a writer. It must read queue pointers to determine where to write the next item, and it must determine if the buffer is full. Similarly, the consumer is not just a reader, because it must adjust the queue pointers to show that it has removed a unit from the buffer. We now examine two solutions to the problem.

**Readers Have Priority**

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

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphore: Readers Have Priority

Figure 5.22 is a solution using semaphores, showing one instance each of a reader and a writer; the solution does not change for multiple readers and writers. The writer process is simple. The semaphore wsem is used to enforce mutual exclusion. As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader
process also makes use of wsem to enforce mutual exclusion. However, to allow multiple readers, we require that, when there are no readers reading, the first reader that attempts to read should wait on wsem. When there is already at least one reader reading, subsequent readers need not wait before entering. The global variable readcount is used to keep track of the number of readers, and the semaphore x is used to assure that readcount is updated properly.

**Writers Have Priority :-**

In the previous solution, readers have priority. Once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader in the act of reading. Therefore, writers are subject to starvation.

Figure 5.23 shows a solution that guarantees that no new readers are allowed access to the data area once at least one writer has declared a desire to write. For writers, the following semaphores and variables are added to the ones already defined:

- A semaphore \( rsem \) that inhibits all readers while there is at least one writer desiring access to the data area
- A variable \( writecount \) that controls the setting of \( rsem \)
- A semaphore \( y \) that controls the updating of \( writecount \)

For readers, one additional semaphore is needed. A long queue must not be allowed to build up on \( rsem \); otherwise writers will not be able to jump the queue. Therefore, only one reader is allowed to queue on \( rsem \), with any additional readers queuing on semaphore \( z \), immediately before waiting on \( rsem \). Table 5.6 summarizes the possibilities.

An alternative solution, which gives writers priority and which is implemented using message passing, is shown in Figure 5.24. In this case, there is a controller process that has access to the shared data area. Other processes wishing to access the data area send a request message to the controller, are granted access with an “OK” reply message, and indicate completion of access with a “finished” message. The controller is equipped with three mailboxes, one for each type of message that it may receive.

The controller process services write request messages before read request messages to give writers priority. In addition, mutual exclusion must be enforced.
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1)
            semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true)
    {
        semWait (y);
        writecount++;
        if (writecount == 1)
            semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}

Figure 5.23 A Solution to the Readers/Writers Problem Using Semaphore: Writers Have Priority
void reader(int i)
{
    message rmsg;
    while (true)
    {
        rmsg = i;
        send (readrequest, rmsg);
        receive (mbox[i], rmsg);
        READUNIT();
        rmsg = i;
        send (finished, rmsg);
    }
}

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

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

Figure 5.24 A Solution to the Readers/Writers Problem Using Message Passing

To do this the variable count is used, which is initialized to some number greater than the maximum possible number of readers. In this example, we use a value of 100. The action of the controller can be summarized as follows:

- If count _ 0, then no writer is waiting and there may or may not be readers active. Service all “finished” messages first to clear active readers. Then service write requests and then read requests.
- If count _ 0, then the only request outstanding is a write request. Allow the writer to proceed and wait for a “finished” message.
- If count _ 0, then a writer has made a request and is being made to wait to clear all active readers. Therefore, only “finished” messages should be serviced.