

Monitors

- *Semaphores* are good synchronizing primitives since they don't waste CPU time through busy waiting.
- However, the semaphore is still a low level primitive since it is unstructured. We consider the word *unstructured* in the context of developing large concurrent software.
- Since the correctness of a program which uses semaphores depends crucially on correct calls to **Wait** and **Signal**, the responsibility is completely on the programmer to issue these calls correctly.
- Suppose, a programmer forgets a single **Signal** call. The program may deadlock due to this single omission and since operations on semaphores are distributed all over the program, it will be difficult to isolate the reason for this deadlock.
- If the responsibility of ensuring mutual exclusion is concentrated in a few system level modules, it will be easier to debug and maintain concurrent programs.

Monitors

- Usually, several different *monitors* are provided in an operating systems for providing access to different shared devices or data structures.
- If two different processes call the same monitor, the underlying implementation ensures that these are processed sequentially to preserve mutual exclusion. However, if two or more processes each call different monitors, their execution can be interleaved.
- Another motivation behind implementing monitors is *encapsulation* of *data* and *procedures* related to a particular service in a single module. Users may not know about the details of how the implementation is done.

Producer-Consumer Problem

We will consider the *producer-consumer* problem through a monitor implementation. The following is the monitor implementation of various services, **Append** for the *producer* and **Take** for the *consumer*.

```
monitor Producer_Consumer_Monitor is
B: array (0..N-1) of Integer;
In_Ptr, Out_Ptr : Integer :=0;
Count : Integer := 0;
Not_Full, Not_Empty : Condition;

Procedure Append (I: in Integer)
begin
  if (count = N) then
    Wait(Not_Full);end if;
  B(In_Ptr) := I;
  In_Ptr := (In_Ptr+1) mod N;
  Signal(Not_Empty);
end Append;

Procedure Take(I: out Integer)
begin
  if (count = 0) then
    Wait(Not_Empty);end if;
  I:= B(Out_Ptr);
  Out_Ptr:=(Out_Ptr+1)mod N;
  Signal(Not_Full);
end Take;

end Producer_Consumer_Monitor;
```

Producer-Consumer Problem

```
task body Producer          task body Consumer
  I: Integer;              I: Integer;
begin loop                 begin loop
  Produce(I);              Take(I);
  Append(I);               Consume(I);
end loop;                  end loop;
end Producer;              end Consumer;
```

- Only one process is allowed to execute a monitor procedure at any time. In this case, the **Producer** may execute **Append** and the **Consumer** may execute **Take**, but not both at the same time. This ensures mutually exclusive access to the shared variables.
- This solution is more structured compared to **semaphores** due to two reasons. **i.** The data and procedures are encapsulated in a single module and **ii.** mutual exclusion is provided automatically and without user involvement in the code.
- The **Producer** and **Consumer** processes only see the abstract implementation of the procedures **Append** and **Take** and need not know the details of the implementations or data.

Condition Variables

Mutual exclusion is provided with the help of *condition variables*. A condition variable **C** has three operations defined upon it.

Wait(C) : The process that called the monitor procedure containing this statement is suspended on a FIFO queue associated with **C**. Once the process is suspended, the mutual exclusion on the monitor is released.

Signal(C) : If the queue for **C** is non-empty, then wake the process at the head of the queue.

Non_Empty(C) : A boolean function that returns **true** if the queue for **C** is non-empty.

Condition Variables

- The **Wait** operation allows a process to suspend itself.
- A *condition variable* is just a *signalling device*. Unlike general semaphores, it has no memory associated with it.
- It is the responsibility of the programmer to ensure that the condition actually occurs.
- **Wait(C)** releases the mutual exclusion on the monitor, allowing other processes to *acquire* or enter the monitor. In this way, other processes may be able to establish the condition of the **Wait**. In our case, if a consumer is waiting for an item, a producer may enter the monitor and produce that item.
- In the producer-consumer problem, we use two condition variables.

Not_Empty : Used by the consumer to suspend itself until the buffer is not empty.

Not_Full : Used by the producer to suspend itself until the buffer is not full.

Condition Variables

- **Signal(C)** only requires that the first process suspended on **C** should be awakened. It is not the responsibility of the signalling process to ensure that the awakened process is actually scheduled.
- A signal is issued when the signalling process determines that the condition required by a suspended process exists now. It is extremely important that an awakened process is scheduled for immediate execution after the signal.
- Consider the scenario when there are two consumer processes **C1** and **C2** and one producer process **P1**. **C1** is suspended on **Not_Empty**, **C2** has just called **Take** and **P1** is about to execute **Signal(Not_Empty)**.
- **P1** awakens **C1** and the buffer is not empty. If **C2** is allowed to enter the monitor, it will make the buffer empty and afterwards, **C1** will enter the monitor and will try to take an item from the empty buffer !
- It is the responsibility of the underlying implementation to ensure that an awakened process should execute immediately. This requires that an awakened process should be given higher priority.

Readers and Writers

- The Readers and Writers problem is almost similar to the mutual exclusion problem. In this problem, there are two distinct groups of processes, **Readers** and **Writers**.
- **Readers** are processes which are not required to exclude one another.
- **Writers** are processes which are required to exclude every other process, readers and writers alike.
- This problem is an abstraction of database operations. Many processes can read the information from a database concurrently, but the updating should be done in a mutually exclusive fashion.

```
task body Reader is
begin loop
  Start_Read;
  Read_Data;
  End_Read;
end loop;
end Reader;
```

```
task body writer is
begin loop
  Start_Write;
  Write_Data;
  End_Write;
end loop;
end Writer;
```

Readers and Writers

```
monitor Reader_Writer_Monitor is
  Readers : Integer := 0;
  Writing : Boolean := False;
  OK_to_Read, OK_to_Write : Condition;

  procedure Start_Read is
  begin
    if Writing or
       Non_Empty(OK_to_Write) then
      Wait(OK_to_Read);
    end if;
    Readers := Readers+1;
    Signal(OK_to_Read);
  end Start_Read;

  procedure End_Read is
  begin
    Readers := Readers-1;
    if Readers=0 then
      Signal(OK_to_Write);
    end if;
  end End_Read;

  procedure Start_Write is
  begin
    if Readers /=0 or Writing then
      Wait(OK_to_Write);
    end if;
    Writing := True;
  end Start_Write;

  procedure End_Write is
  begin
    Writing := False;
    if Non_Empty(OK_to_Read) then
      Signal(OK_to_Read);
    else
      Signal(OK_to_Write);
    end if;
  end End_Write;

end Reader_Writer_Monitor;
```

Readers and Writers

- The monitor has two *status variables* and two condition variables.
- The status variables are :
Readers : A counter of successful readers which have successfully passed **Start_Read** and are currently reading.
Writing : A boolean flag which is true when a process is writing.
- The condition variables are :
OK_to_Read to suspend readers.
OK_to_Write to suspend writers.
- A reader is suspended if some process is currently writing or if some process is waiting to write. This gives priority to a suspended writer over the readers.
- A writer is suspended if other readers are currently reading or some writer is writing.

Readers and Writers

- **End_Read** executes **Signal(OK_to_Write)** if there is no other reader.
- **End_Write** gives priority to suspended readers if any, otherwise, it signals suspended writers.
- The purpose of the **Signal(OK_to_Read)** in **Start_Read** is to allow other readers to start reading. So, the readers get access in a cascading fashion.
- If there are suspended writers, a new reader is required to wait until the termination of (at least) the first write.
If there are suspended readers, they will all be released before the next write.