Concurrent Programming Part II
Building Concurrent Programs and
the Design of Concurrent Languages

Lecturer: Dr Rowan Davies
Consultation time: Tuesday 3-5 in CS&SE 2.16

Labs: start next week (week 8) for this half
- 2 weeks of non-assessed introductory labs
- Mini-project: worth 20%, due week 13, working in pairs

Tutorials: none

Topics covered: (approximately)
- Threads, Synchronization and Monitors
- Java Concurrency
- Distributed Programming: Java RMI and CORBA
- Distributed snapshots and termination
- Fault tolerance (if time permits)

Introducing Concurrency into a Language

- Suppose we have a sequential programming language.
- How can we support concurrency without radical changes?
  - The standard approach is to introduce multiple threads of execution.
  - Each thread is sequential, and runs as per usual in the absence of interaction.
- How are threads created?
  - Generally a thread can create another, but we’ll start with a fixed number of threads.
- How do threads interact?
  - Interaction may occur through messages or shared variables. We’ll focus on shared variables.
- How do threads synchronize?
  - Synchronization between threads is critical in most concurrent programs for mutual exclusion, etc.

Preliminaries

- Up to now this unit has focused on describing and analyzing concurrency precisely.
- Now we will focus on how to build concurrent programs and the language features that support concurrency.
- In the labs we will use Java.
- In the lectures we will see both Java and Ada-ish pseudo-code based on the recommended text by Ben-Ari.
- You will not need to learn to read and write proper Ada code, only pseudo-code that is reasonably precise.

Monitors

- Semaphores are good synchronizing primitives since they don’t waste CPU time through busy waiting - threads can be suspended and resumed appropriately by the implementation.
- 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.
- Java uses a variant of monitors that is object-oriented and has some other important differences, as we’ll see in the next lecture topic.

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.

```plaintext
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) Procedure Take(I: out Integer)
begin begin
  if (Count = N) then if (Count = 0) then Wait(Not_Full);end if; Wait(Not_Empty);end if;
  B(In_Ptr) := I; I:= B(Out_Ptr);
  In_Ptr := (In_Ptr+1) mod N; Out_Ptr:=(Out_Ptr+1)mod N;
  Signal(Not_Empty); Signal(Not_Full);
end Append; end Take;
end Producer_Consumer_Monitor;
```

Producer-Consumer Problem

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

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.

Emulation of Semaphores by Monitors

**Monitors** and **Semaphores** are of similar power. If a system does not provide support for one primitive, it is possible to implement that primitive through the other. We will show emulation in both directions.

Emulation of semaphores by monitors

```pascal
monitor Semaphore_Emulation is
  S: Integer := S0;
  Not_Zero : Condition;
Procedure Semaphore_Wait is
begin
  if S=0 then Wait(Not_Zero); end if;
  S := S-1;
end Semaphore_Wait;
Procedure Semaphore_Signal is
begin
  S := S+1;
  Signal(Not_Zero);
end Semaphore_Signal;
end monitor
```

**Theorem.** The semaphore invariants hold:

\[ S \geq 0 \quad (1) \]

\[ S = S0 - \#\text{waits} + \#\text{signals} - 1 \quad (2) \]

**Proof:** Remember that every execution of a monitor procedure is an atomic instruction. It is possible that variables may temporarily have values which falsify the invariants, but these temporary values are invisible to other processes.

(1) is true initially. The only transition that can falsify the formula is an execution of **Semaphore_Wait** when S = 0. In that case, **Wait(Not_Zero)** will suspend the process. It will be awakened only by a **Signal(Not_Zero)**. But the value of S is increased just before the signal is issued. So, (1) is an invariant.

If a **Wait** is completed, the value of S decreases by 1 and if a **Signal** is called, the value of S increases by 1. So, (2) is an invariant. Remember that, if a process is awakened, it is scheduled immediately and hence the statement \( S := S-1 \) is executed immediately.

This proof assumes a blocked-queue semaphore since the processes suspended on the **Wait** are kept in a FIFO queue.
Emulation of Monitors by Semaphores

- To emulate monitors using semaphores, we need a semaphore \( S \) to ensure mutual exclusion on the monitor procedures.
- We need one semaphore for each condition variable to ensure mutual exclusion. For a condition variable \( C \), we will call this semaphore as \( \text{CSemaphore} \).
- The semaphores are assumed to be blocked-queue, otherwise we have to explicitly implement a queue to maintain the suspended processes on a \textit{Wait}.
- The meaning of \textit{Signal}(C) for a condition variable depends on whether the queue for \( C \) is empty or not. So, we have to keep a count of the number of processes suspended on a condition variable \( C \). We will call this as \( \text{C_Count} \) for a condition variable \( C \).

Each procedure in the monitor will have \textit{Wait}(S) as its first instruction and \textit{Signal}(S) as its last instruction. Remember that, these are the \textit{Wait} and \textit{Signal} operations on the semaphore \( S \).

- Each occurrence of \textit{Wait}(C) is translated to:
  
  \[
  \text{C_Count} := \text{C_Count} + 1; \\
  \text{Signal}(\text{CSemaphore}); \\
  \text{Wait}(\text{CSemaphore}); \\
  \text{Wait}(S); \\
  \text{C_Count} := \text{C_Count} - 1;
  \]

- Initially, \( \text{C_Count} \) is incremented by 1. Then the monitor is released through \textit{Signal}(S). After that, the process waits on \( \text{CSemaphore} \) if there is some other process currently accessing \( C \). Then an \textit{Wait}(S) is executed to get hold of the monitor and finally, \( \text{C_Count} \) is decremented since \( \text{CSemaphore} \) has been signalled already.

- The definition of \textit{Wait}(C) requires that the mutual exclusion on the monitor be released by \textit{Signal}(S). This must be done before waiting for the condition, otherwise there may be deadlock.

---

Emulation of Monitors by Semaphores

- Each occurrence of \textit{Signal}(C) is translated to:
  
  \[
  \text{if C_Count} > 0 \text{ then} \\
  \text{Signal(}\text{CSemaphore}); \\
  \text{end if}; \\
  \text{Signal}(S);
  \]

- If \( \text{C_Count} > 0 \), then there are processes waiting in the queue for \( C \). Hence, \( \text{CSemaphore} \) should be signalled so that a waiting process can wake up and decrement \( \text{C_Count} \).
- Also, remember that \textit{Signal}(C) should be the last instruction of a monitor procedure.
- These two translations of \textit{Wait}(C) and \textit{Signal}(C) are almost correct. However, there is still a chance of the following interleaving of instructions:
  
  a third process may come in between the execution of \textit{Wait(}\text{CSemaphore}) and \textit{Wait}(S). This third process could execute \textit{Wait}(S) before the awakened process and violate the immediate resumption condition.

- The solution is not to release the mutual exclusion on the monitor when signalling, but to let it remain in force until the awakened process itself releases the mutual exclusion.
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

- 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

- 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

- 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

- 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

- 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

- 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.