1 Process Coordination

Process coordination consists of *synchronization* and *mutual exclusion*, which were discussed earlier. We will now study different techniques for process coordination. We first make a few definitions related to process coordination.

Critical Section

Instructions that must be executed while another activity is excluded will be called a **critical section**. For instance in an earlier example we saw that the statement:

```
A[n++] = item
```

could be executed by only one of the processes *AddChar1* and *AddChar2*. This statement is a critical section.

Starvation and Deadlock

Consider the following situation: Process *p1* is in a critical section *c1* waiting for process *p2* to get out of a critical section *c2*. Process *p2*, meanwhile, is waiting for process *p1* to get out of *c1*. Both *p1* and *p2* are stuck: each is waiting to get into a critical section being executed by the other. This sort of circular waiting is called a **deadlock**. In Europe, it is known by the more striking name of "deadly embrace".

Another problem related to process coordination is **starvation**. This problem arises when one or more processes waiting for a critical section are *never* allowed to enter the region.

Different process coordination techniques are often judged on the basis of their ability to prevent these two problems.

Other Requirements

There are many other metrics of measuring the effectiveness of a process coordination technique:

Concurrency: How much concurrency does it allow, while meeting the process coordination constraints?

Functionality: What are the range of procecss coordination constraints it support? Does it support both mutual exclusion and synchronization? What kind of synchronization constraints does it support?

Progammabilty: What is the programming cost of using the approach? How easy is it to make mistakes when using it ?

Efficiency: What are the space and time costs? Does it result in frequent context switches? Worse, does it do polling?

Multiprocessor systems: Does it work on multiprocessor systems?

As we will see below, some of these requirements conflict with each other.

1.1 Disabling Context Switching

Needed by operating system to implement its functions (for instance to implement *resume* and other system calls).

In Xinu it consists of disabling interrupts and not making any call that may result in rescheduling.

Major drawbacks of this technique as a general tool for mutual exclusion:

Allows only one activity on a single computer. Thus it stops all processes when one of them executes the critical section.

Disables clock and other device interrupts. Thus lengthy computations in a critical section are not desirable.

Technique fails on a multiprocessor system.

1.2 Busy Waiting

Another possible technique for mutual exclusion is to use **busy waiting**. We can introduce a Boolean variable called *mutex* that is set to true when an activity is in a critical section, to false otherwise. A process executes the following code before it enters a critical section:

```
while  mutex do
    do nothing
end;
mutex <- true;
```

and the following code when it exits the region:

```
mutex <- false
```

Unfortunately, this code is wrong. If two processes start executing the while loop at the same time, they might both get past the loop and enter their regions. The problem is that the checking and setting of the Boolean are done in two statements. A process that has tested the Boolean variable may be rescheduled before it sets the variable.

Let us assume that the ‘test and set’ can be done in one instruction, so that the code looks like:

```
while  (TestAndSet(mutex)) do
    do nothing
end;
```

where TestAndSet looks like:

```
atomic function TestAndSet (var Lock: Boolean): Boolean;
begin
    TestAndSet <- Lock;
    Lock <- true
end TestAndSet;
```

The solution is not good since the CPU may spend a lot of time executing the while loop which does no useful work. In particular, if the ‘waiting’ process is a high priority process waiting for a lower priority process to set the Boolean variable to false, the CPU will spend *all* its time executing the loop. However several computers offer such an instruction as the low-level alternative to disabling interrupts in a multiprocessor system.

1.3 Semaphores

An abstract entity (not provided by hardware)

Named by a unique semaphore id.

Associated with four operations, which are described below.

Consists of a tuple (count, queue), where count is an integer and queue is a list of processes. The following invariant is true for semaphores: a nonnegative count means that the queue is empty; a semaphore count of negative *n* means that the queue contains *n* waiting processes.

The four operations are:

Wait: decrements count, and enqueues process if count is negative.

Signal: increments count and make a process ready if waiting.

Create: generates a new semaphore with a count.

Delete: destroys an existing semaphore.

The last two operations are non-standard and allow semaphores to be created and destroyed dynamically.

The description of semaphores given above is too implementation-oriented in that it is possible to create an implementation that does not, for instance, increments rather than decrements a count in wait. Here is a more general description of the semaphore semantics.

A semaphore is associated with an initial count, a FIFO process-queue, and the wait and signal operations. It keeps track of how many times wait and signal are called.

The signal operation deques the process, if any, at the head of the queue.

The wait operation puts the process calling it in the queue if:

```
The number of previous signals + the initial count < the number of prervious waits
```

1.3.1 Implementation

The implementation of semaphores in Xinu consists of the following:

Semaphore lists and *wait* state.

Semaphore Table

The routines *wait*, *signal*, *create*, and *delete*.

Semaphore Lists and Wait state

A process waiting on a semaphore is put into the **wait** state and the **semaphore list** associated with the semaphore. This list implements the queue associated with the semaphore.

Semaphore Table

The table contains an entry for each semaphore which in turn contains:

integer count of the semaphore

head and tail of list associated with the semaphore.

A semaphore is identified by the index of its semaphore table entry.

The Routine Wait

Decrements count of the semaphore.

If count is negative puts process in the **wait** state and the list associated with the semaphore. Also calls *reschedule*.

The Routine Signal

If count is negative puts process at head of semaphore list in ready list, and reschedules.

Increments the count of the semaphore (before calling reschedule)

The Routine Create

Takes as argument the initial count of the semaphore.

Return error if count is negative. (why?)

Create a table entry for the semaphore by finding an unused one and initializing it. Each table entry, at startup time, is associated with a semaphore list. So the latter does not have to be created.

Returns index of the semaphore.

The Routine Delete

Frees table entry.

Puts waiting processes in ready list.

Calls reschedule.

1.4 Monitors

Provided by Programming Languages such as Concurrent Pascal, Modula, Modula-2, Mesa, and Java.

Like a Module: Declares data (access to which is to be mutually excluded), procedures that manipulate these data, and an initialization code. The procedures can be **exported** to other modules which may **import** them.

Unlike a Module in following respect: Can declare certain procedures as **entry procedures**, which have the property that no matter how many processes are running, only one process is allowed to execute an entry procedure at a time. Thus if a monitor declares two entry procedures *e1* and *e2*, then while *e1* is being executed by a process, another process cannot call *e1* or *e2*. A process that invokes an entry procure while the monitor is 'busy' (that is some other process is executing an entry procedure in it) is put on an **entry** queue.

List Problem

The following program shows the use of monitors to implement the ‘list’ problem, explained earlier, involving two processes that add items to a list;

```
monitor List;

export AddChar;

var
    /* shared data */
    n: integer;
    a: array 0..1 of char;

/* an entry procedure */
entry procedure AddChar [item: char];
begin
    a[n++] <- item;
end;

/* initialization code */
begin
    n <- 0;
end;
```

Monitors provide a ‘higher-level’ solution to the mutual exclusion problem than semaphores:

We need not worry that some other piece of code accesses the shared data and thus must be made a critical region.

It is possible to determine the criticial regions by looking only at the headers of the monitor procedures.

A process does not have to remember the final *signal* operation.

Bounded Buffer Problem

Monitors, as described so far, provide a solution to the mutual exclusion problem, but not the synchronization problem. To accommodate a solution to the latter, they are associated with **conditions** associated with the **wait** and **signal** operations. A call to *wait* places the process that calls it in a **condition queue** associated with the signal. It is removed from this queue when some other process does a *signal* on the condition. Before calling wait, a process must restore the invariant of the monitor as the call frees the monitor for other processes to enter.

Thus, a signal is like a semaphore in that it is associated with a queue in which processes may wait for signals. The difference is that it is a more lightweight mechanism not associated with a count variable. Any state associated with a condition must be explicitly managed by the monitor programmer. (Does it make sense to support conditions as an alternative to semaphores in a world without monitors?)

We can use a monitor to implement the **bounded buffer** problem stated as follows:
Any number of producer and consumer processes are running. A producer puts data in a buffer whose size is finite (hence a bounded buffer). A consumer removes data from this buffer. A producer blocks when the buffer is full and a consumer blocks when the buffer is empty.

The following is a monitor solution to this problem:

```
monitor BoundedBuffer;

export GetBuffer, PutBuffer;

const
    size = 10;
type
    Datum = ... /* the data type of the contents of the buffer */
var
    buffer: array 0..size-1 of Datum;
    count: 0..size; /* number of elements in buffer */
    nextIn, nextOut: 0..size-1; /* index of next datum to place in buffer or remove
*/
    nonEmpty, nonFull: condition;

entry procedure PutBuffer (what: Datum);
begin
    if count = size then
        wait nonFull;
    end;
    buffer [nextIn] <- what;
    nextIn <- (nextIn + 1) mod size;
    count <- count + 1;
    signal nonEmpty
end;

entry procedure GetBuffer(var result: Datum);
begin
    if count = 0 then
        wait nonEmpty
    end;
    result <- buffer[nextOut];
    nextOut <- (nextOut + 1) mod size;
    count <- count -1;
    signal nonFull;
end;

begin /* initialization code */
    count <- 0;
    nextIn <- 0;
    nextOut <- 0;
end}
```

A producer waits on the signal *nonFull* before adding new data, and signals *nonEmpty* when it successfully adds data. Conversely, a consumer waits on *nonEmpty* before consuming new data, and signals *nonFull* when it successfully adds new data.

Assume that a consumer is waiting on a condition, and a producer does a signal. When is the consumer allowed to execute? If immediately, then we may have two processes in the monitor at the same time, and our careful mutual exclusion is ruined. If later, then some other process may have taken the last datum, and the assumption made by the first consumer that an execution of the *wait nonEmpty* statement guarantees a non-empty buffer is wrong.

Not all definitions of monitors in the literature agree on the answers to these questions. The following is a common approach. We associate a monitor with an **urgent** queue. Thus a monitor is associated with an entry queue, a condition queue for each condition, and an urgent queue. Processes that are blocked are placed in these queues. Here are the rules that govern the placement and removal of processes from these queues:

New processes wait in the entry queue.

When a process exits a monitor (that is, finishes execution of an entry procedure), a process from the urgent queue is allowed to execute if the queue is not empty. Otherwise, a process from the entry queue is removed if the queue is not empty.

A process that does a *wait* enters the appropriate condition queue.

When a process executes *signal*, the signalled condition queue is inspected. If some process is waiting on the queue, the signaller enters the urgent queue and some waiting process is allowed to execute. If no process is waiting in that queue, then the signaller proceeds without leaving the monitor. The signal is ignored.

All queues are ordered first in, first out.

These rules ensure that a waiting consumer is unblocked immediately when a producer signals *NonEmpty*, and the producer is blocked in the urgent queue until the consumer has taken the datum. We have maintained the rule that at most one process occupies the monitor.

People who use monitors have noticed that the signal operation is almost always the last operation in an entry procedure. The above rules will often make the signaller wait in the urgent queue and then return to the monitor just to get out of it. Thus these rules, while they work, are often inefficient. Thus some people require that signal *must* be the last operation executed in an entry procedure.

Another alternative is to make the *signal* operation a *hint* to a waiting process; it causes execution of some process waiting on the condition to resume at some convenient future time. There is no guarantee that some other process will not enter the monitor before the waiting process. Under these semantics, the waiting process has to reestablish the condition for which it was waiting still holds when it wakes up. The proper pattern for usage is:

```
while not <ok to proceed> do
    wait c
```

instead of

```
if not <ok to proceed> then
    wait c
```

Thus in the bounded-buffer problem, the code

```
if count = 0 then
    wait nonEmpty
```

is replaced by

```
while count = 0  do
    wait nonEmpty
```

The last alternative results in an extra evaluation of the <ok to proceed> predicate after the wait. In return there are fewer process switches compared to the first alternative, and it is more flexible than the second alternative. Moreover, it allows dequeueing of all processes waiting on a signal. (why?). Thus a special operation **broadcast** can be defined on signals.

The queues maintained under the hint semantics are the same as above except that it is the signalled process that goes into the urgent queue rather than the signalling process.

In the Mesa solution, a separate module must be defined for each bounded buffer. Java overcomes this problem by allowing a single class to be defined for all instances of a synchronized resource.

In Java, a class may declare certain methods to be declared as entry procedures (by using the keyword `synchronized`). Each instance of a class with entry procedures behaves as a monitor. Java implements a simplified version of these hint-based monitor semantics. A Java object, is associated with a single condition variable and queue. It supports both signal (called `notify`) and broadcast (called `notifyAll`) operations. In either case, the notification is a hint, and the notified process is supposed to check for itself if its condition has been satisfied. (How would you implement the bounded buffer problem with these semantics?) To address deadlocks, Java also supports wait operations with timeouts.

Why did we not have analogous semantic problems with the semaphore signal operation?

Yet another issue in monitors is nested monitor calls - what if an entry procedure in one monitor calls an entry procedure in another monitor? If the first monitor remains occupied as the process enters the entry queue of the second monitor, deadlocks can occur. Therefore some designs require that the first monitor be freed for other processs to enter, but this approach requires the process to restore the invariant of the first monitor before entering the second monitor.

1.5 Synchronous Message Passing

Synchronous message passing offers another mechanism for process synchronization. To illustrate, a bounded buffer can be defined using message-passing primitives. The buffer is managed by a special "bounded buffer" process, with which producers and consumers communicate to produce and consume buffers, respectively:

```
loop
    select
        when count > 0 receive GetBuffer (...) ...
        when count < size receive PutBuffer (...) ...
    end
end loop.
```

Here we are assuming RPC, select, and guards. The process defines two input ports, one for producers and another for consumers, and uses select to receive both produce and consume requests. The variable `count` keeps the number of filled buffers. A consuming process is blocked if the buffer is empty and a producing process is blocked if it is full.

Notice a process executing such a loop statement is similar to a monitor. Each receive in an arm of a select corresponds to an entry procedure declaration in a monitor. A process services one receive at a time, just as a monitor executes one entry procedure at a time. The guards correspond to waits on conditions. The Lauer and Needham paper on duality of operatings system structures contains a more detailed discussion on this topic.

1.6 Path Expressions

While monitors can handle both mutual exclusion and synchronization, the mechanisms for supporting these two forms of process coordination have important differences. Mutual exclusion is specified by high-level declarations indicating which procedures are entry procedures and the code for supporting it is generated by the compiler. On the other hand, process synchronization is implemented by low-level code scattered throughout the monitors that is as low-level as the semaphore code. In particular, it is subject to the problem that a wait may not be associated with a matching signal.

Path expressions overcome this problem by allowing coordination constraints to be specified by high-level declarations that are defined in a single-place, provide a unified mechanism for supporting both mutual exclusion and synchronization, and are implemented by compiler-generated code. Like monitors, they extend the abstraction of a module - instead of defining entry procedures and conditions, the programmer associates a module with a path expression that defines the coordination constraints of the module.

A path expression is much like a regular expression - the "alphabet" consists of operation (procedures) defined by the associated module. The operators are the infix operator *','*, which says the system can select which of its two operands is (serially) executed first in the enclosing path expression, the infix operator *';'*, which says its left operand must be executed before its right operand, *N: (path_exp)* to specify upto N (global) concurrent activations of *path_exp*, and *[path_exp]* to specify an unbounded number of concurrent executions of *path_exp*. A path expression not enclosed by brackets or paretheses is implicitly enclosed by square brackets.

Let us illustrate the semantics of path expressions using the bounded buffer example:

```
module BoundedBuffer;

export GetBuffer, PutBuffer;

 const
    size = 10;
 type
    Datum = ... /* the data type of the contents of the buffer */
 var
    buffer:  array 0..size-1  of Datum;
    nextIn, nextOut: 0..size-1; /* index of next datum to place in buffer or remove
*/

 procedure PutBuffer (what: Datum);
 begin
    buffer [nextIn] <- what;
    nextIn <- (nextIn + 1)  mod size;
 end;

 procedure GetBuffer( var result: Datum);
 begin
    result <- buffer[nextOut];
    nextOut <- (nextOut + 1)  mod size;
 end;
```

```
begin /* initialization code */
   nextIn <- 0;
   nextOut <- 0;
end;
```

This is the same implementation as the one we saw for the monitor case except that all code specifying coordination constraints has been removed. An interesting consequence is that the variable `count` shared by `GetBuffer` and `PutBuffer` has disappeared, allowing these two procedures to execute concurrently.

Let us try some path expressions to specify coordination constraints for this module. The three expressions

```
PutBuffer, GetBuffer
[PutBuffer, GetBuffer]
[PutBuffer], [GetBuffer]
```

all specify that an arbitrary number of activations of the two procedures can be executed concurrently. Thus they do not place any constraints on the module. The two expressions

```
PutBuffer; GetBuffer
[PutBuffer; GetBuffer]
```

on the other hand specify that each activation of `GetBuffer` be preceded by an execution of `PutBuffer` but there can be concurrent activations of this serial sequence. Thus, it ensures that the number of completed executions of `GetBuffer` never exceeds the number of completed executions of `PutBuffer.` It does not, however, prevent multiple executions of `GetBuffer` or `PutBuffer` to be active simultaneously. The expression

```
1 : (PutBuffer; GetBuffer)
```

ensures that there can only be one activation of this sequence at any one time. Thus, it ensures that the two procedures alternate (starting with `PutBuffer`) and implements a bounded buffer of size 1.

The expression

```
N: (PutBuffer; GetBuffer)
```

ensures that:

each activation of `GetBuffer` is preceded by an activation of `PutBuffer,`

the number of completed `PutBuffer` executions is never more than N greater than the number of completed `GetBuffer` operations.

Thus, it almost implements a bounded buffer of size N. The problem with it is that multiple instances of PutBuffer (GetBuffers) can execute concurrently, and thus does not exclud concurrent access to the variables manipulated by the procedure.

The expression

```
N: (1: (PutBuffer); 1 : (GetBuffer))
```

fixes this problem, ensuring that:

we can have at most one concurrent execution of `PutBuffer` at any one time;

we can have at most one concurrent execution of `GetBuffer` at any one time;

each activation of `GetBuffer` is preceded by an activation of `PutBuffer`,

the number of completed `PutBuffer` executions is never more than N greater than the number of completed `GetBuffer` operations.

Thus, it implements a bounded buffer of size N.

As the above example illustrates, N:(pathexp) specifies N total executions of pathexp, even if it is enclosed in another pathexp. In other words, it does not specify N executions in the enclosing path expression.

Path expression can also easily specify a multiple readers/single writer constraint:

```
1 : ([read], write)
```

This expression indicates that there can be an unbounded number of reads or a single write concurrently active at any time. The use of the selection operator ensures that the reads and the write mutually exclude each other but unlike the sequencing operator does not place an order on them.

While path expressions are high-level, they cannot specify all kinds of process coordination constraints. For instance, there is no way to specify a fair readers/writers constraint that ensures that a writer does not wait for a reader who came after him. In general, there is no way for specifying constraints that depend on information other than the history of operations invoked so far.