# *PROBLEMS OF CONCURRENT PROCESSING*

Apart from interprocess communication ( which we assume to be expected ), processes can interact with each other in many ways. We have seen how the hardware, in conjunction with the operating system, prevents unwanted interference in principle. All requests for resources, including more memory, printers, files etc. are mediated by the operating system. This makes it impossible under most circumstances for one process to use a resource which is currently allocated to another process.

We have also seen, though, that in other situations processes want to share a resource; sharing memory is a prime example. In this case the processes themselves must ensure that the shared resource is used in a sensible manner, without data from one process getting jumbled with data from another process. To coordinate their actions, the processes must communicate with each other. Well, that's all right – they can communicate through the shared resource. Of course, to achieve reliable communication, the processes must ensure that the shared resource is used in a sensible manner ..... We have been here before. It is clear that they need help from the operating system. But how exactly does it work ?

These two problems are essentially the same. A solution to the first, restricting access to a resource, would enable processes sharing a resource to coordinate their activity on the shared resource. In effect, access to the shared resource is voluntarily restricted in the interests of reliable communication.

## MUTUAL EXCLUSION

In the chapter *PROCESSES IN COMPUTERS*, we noticed that we would require some means of ensuring that resources accessible to more than one process can be temporarily reserved for a single process so that it can perform some operation without any danger of the resource being changed while the operation was in progress.

To give a simple illustration, suppose there is a shared memory buffer which receives characters from one process which manages an input device, and from which characters are taken by a second process which must interpret them. Every time an end-of-line character is received, the interpreting process is activated, and the input process begins again at the beginning of the buffer. It is clearly necessary to ensure that the input process cannot use the buffer while the interpreting process is reading it, for if that were not so the input process could overwrite the characters in the buffer before the interpreter had used them. ( It is tempting to rely on the slowness of typing and the high speed of the interpreting process to make the system work without special provision for safety – which is all very well, until someone redirects the input and takes it from a disc file. It never works. ) Similarly, we don't really want the interpreter process to use the contents of the buffer until the input process has reached the end of the line.

This sort of relationship between processes which share resources is very common, and we must be able to restrict access to the shared entity to one process for an arbitrary period of time. This restriction of access is known as *mutual exclusion*. Only one process at a time is allowed to use the resource ( so it's also called *exclusive access* ) .

### Locks.

We will start by looking at what happens within the kernel itself when a resource is about to be allocated to a process. Something like the following code might be used internally by the kernel to ensure restricted access to a resource :

```
if this resource is in use then
     put the requesting process to sleep;{ it will eventually wake
up here }
mark the resource as being used; { preventing other processes from
getting the resource }
```

This code manages transitions between the *runnable* and *waiting* states in the process state diagram. It provides ( or attempts to provide ) the mutual exclusion which we required, and is known as a *lock*. In the simplest case the resource is marked as in use by setting a specific value in a variable, known as a *lock variable*, corresponding to the resource. ( Other housekeeping operations are also likely to be necessary, but here we shall restrict our discussion to the communication issues. ) Every resource which is protected by this method must have a lock variable ( which might be a single lock bit ). We see that restricting access to any resource, hardware or software, can be done the same way. Here's the same code presented rather more formally in terms of a variable `X.lock` which acts as the lock variable for a resource X. We require code for two operations, which we shall call **lock** and **unlock**.

```
lock( X ) :
     if X.lock
     then begin
          link the PCB to a queue of processes waiting for X;
          suspend the current process;
          end
     else X.lock := true;

unlock( X ) :
     if there is a process waiting for X
     then make it runnable
     else X.lock := false;
```

Though there's a little more to say about it yet, it's clear that this code is doing what we need in order to enforce the mutual exclusion we require. Once a process has the resource, no other process can take it away.

The code outlined above means well – but, as we know from our daily life, meaning well is not always enough. In practice, we have to guard against possible ill effects on our good intentions from external sources, and it is the same with the operating system. The code is, by definition, running in a multiprogramming system, for if there were no other process active, we wouldn't need to worry about cooperation. There are therefore other things going on, some of which are likely to involve interrupts. Interrupts might suspend this process, and, as we have seen, it might not be resumed immediately – so circumstances might change without the process being aware of it. This can cause trouble.

Let us assume that the resource required is not currently being used when the process ( call it A ) makes its system call to request the resource. The system starts work on the code, and first finds that the resource is not in use, so doesn't suspend the process. Just before it gets as far as marking the resource as being used, an interrupt occurs. This interrupt does whatever it has to do, but then starts a different process – say, B.

B also requests the same resource by similarly making a system call. The system starts work on the code, and first finds that the resource is not in use, so doesn't suspend B. This time, though, no interrupt occurs, so the request succeeds, the resource is marked as being used, and B carries on, confident that all is well for it to use the resource. At some later time, while B is still using the resource, some other pattern of interrupts starts A again, and it carries on, as it should, from exactly where it left off. Because it has already checked that the resource is not being used, it also returns successfully from the system call believing that it can safely proceed to use the resource. In pictures :

| Process A | System, working for A | System, working for B | Process B |
|---|---|---|---|
| `lock( X ) :` | | | |
| | `if X.lock` | | |
| | `then` ( not locked, so doesn't happen ) | | |

| >>>>>> Context switch >>>>>> | | | |
|---|---|---|---|
| | | | lock( X ) : |
| | | if X.lock | |
| | | then ( not locked, so doesn't happen ) | |
| | | else<br>X.lock := true; | |
| | | | continues ... |
| <<<<<< Context switch <<<<<< | | | |
| | else<br>X.lock := true; | | |
| continues ... | | | |

In other words the system did not ensure exclusive access to the resource.

The problem is the possibility of a process switch between testing to see if the resource is in use and marking the resource as used. There are many solutions to this problem. The simplest is to notice that the criminal in the story is the interrupt, so if we prevent interrupts the problem is solved. Here's a revised version of the **lock** code ( **unlock** works, so doesn't need revision ) :

```
lock( X ) :
     disable interrupts;
     if X.lock
     then begin
          link the PCB to a queue of processes waiting for X;
          suspend the current process;
                         { – and don't forget to turn on the
interrupts again when starting a new process ! }
          end
     else X.lock := true;
     permit interrupts;
```

This works provided that the story is sufficiently realistic – which it is, provided that there is only one processor in the system. And that, in turn, works because the hardware absolutely and immutably guarantees mutual exclusion so far as access to the processor is concerned. We are relying on one sort of mutual exclusion to implement another.

But that no longer works if there are several processors, each of which can address the lock variable. ( You have to be quite careful in this section not to confuse *processes* – the operating system abstraction, with *processors* – the hardware entities which do the work. The first sentence of this paragraph was referring to multiple processors. ) If that's so, disabling interrupts does not stop a process currently running on another processor from reading the variable.

Even in single-processor systems it is not generally a good idea to prevent interrupts if it can be avoided. The longer the time for which interrupts are switched off, the worse the service for any operation which needs interrupts, and the greater the chance that important events might be missed. There is also a greater chance that the interrupts might not get switched on again, which can be catastrophic. Some other method is needed.

The accept-interrupts-but-don't-switch-immediately solution.

One widely used solution to this problem is to postpone context switching. Whenever the system is in supervisor mode no forced context switches are allowed. The diagram shows that this will certainly work for the example given. This is the solution used by Unix. A clock interrupt can be used to mark the time at which the current process has used up its allotted ration of time in execution; if such an interrupt occurs while the processor is in system mode, the system sets a flag to record the occurrence and then continues with the

supervisor call. When the system mode operation is complete and it is time to return to user mode the flag is checked; if it is set, the current process is returned to a waiting state and the next process resumed.

Notice that interrupts are not disabled. In some cases the computer can be spending quite a lot of time in kernel mode and many sources of interrupts have to be serviced as soon as possible after generating the interrupt.

In this way our operating system can ensure mutual exclusion over resources using code very similar to that outlined above.

This solution is still only good for a single processor system. There are in fact purely software solutions to this problem even for multiprocessor systems. They are all based on the fact that only one processor can access the contents of a particular memory location at a time – once again, we are using mutual exclusion enforced by the hardware to implement mutual exclusion more generally.

Test-and-set instructions.

Most processors provide indivisible instructions, i.e. instructions which are guaranteed to run to completion without being broken into either by an interrupt or by a memory access from another processor. All we need to make our lock work is an instruction like this which allows the testing of a variable and changing it to a given value ensuring no other access to the variable until the changed value has been successfully stored.

There are a variety of such instructions; we will look at one example, commonly called the test-and-set instruction.

It works like this :

```
testandset( a, b )
```

The value of b is copied into a and b ( whatever its previous value ) is set to some standard value, which we shall call *true*. Using this instruction we can implement secure mutual exclusion.

If we want to use resource X and the lock variable for this resource is known as `X.lock` :

```
repeat
     testandset( busy, X.lock )
until not busy;
```

If you are unaccustomed to concurrent processing you should wonder how the Boolean variable `busy` can ever become *false* if the first time through the loop it is *true*, since **`testandset( )`** always makes `X.lock` *true*.

The answer is that if the value of `busy` is *true* the first time through the loop, another process has exclusive access to the resource X. In this case there will come a time when it releases its hold on the resource by setting `X.lock` to *false* – so the full code for our operation must be something like this :

```
repeat
     testandset( busy, X.lock )
until not busy;

...... Do things with X ....

X.lock := false;
```

This solution definitely works, but it is still not the solution we want to use because it causes unfriendly behaviour.

First of all, any process requesting an unavailable resource will "busy wait" : the process keeps executing the repeat loop until the other process releases the resource, or until it is displaced from the processor by an interrupt. In a machine with a single processor, this is almost always a complete waste of time, since the process which currently has control of the resource cannot be running simultaneously. If no other interrupt can happen, it stops the system completely. In a multiprocessor the situation is not so bad, for other processes can continue using other processors, but it is still something we want to avoid.

Locks which work on the principle of busy waiting are known as *spin locks*.

The other major problem with this solution is that there is no way of being fair to processes requesting the resource. Whichever process happens to run straight after the resource is released will pick it up, regardless of the fact that another process might have been waiting a long time for the resource.

To solve these problems we must be able to suspend a process if the resource it requires is not immediately available, and wake it up when it is deemed fair that it should receive the resource. We will postpone a closer look at this until we have studied semaphores.

## Semaphores.

Sometimes we don't require a specific resource, but are happy to have any device which will do the job we want done – for example, if you want to print something it might not matter which printer you use. In this case it does not make sense to insist that a process wait for a particular printer. By allocating to the process the first acceptable printer which becomes available we will improve the service to the user. Compare queueing in a bank : before the days of the single-queue-to-multiple-tellers all the other queues always moved faster than the queue which you joined. We hasten to add that, apart from this rather subjective evidence, there are good sound reasons from queueing theory to believe that the best overall service is indeed given by multiple servers with a single queue.

This was the original motivation behind *semaphores* introduced by Edsger Dijkstra[EXE5] in 1965. Dijkstra defined a semaphore as an integer counter which is acted on by the indivisible operations P and V, and an initialization. The indivisibility means that only one process can use the semaphore at a time.

Take S as our example semaphore. So far as counting goes, it behaves just like an integer variable, but it has the peculiar characteristic that it never becomes negative. V is an increment operation :

```
V( S ) :
    S := S + 1;
```

P is a decrement operator which always reduces the value of the counter by 1. What if the counter is zero when P is applied ? If the counter must not become negative, but P must always reduce its value, then P will have to wait until the current value of S is at least 1.

```
P( S ) :
    wait until S > 0;
    S := S – 1;
```

This is all very well, but what does it have to do with resource allocation and mutual exclusion ? That's easy : think of the integer counter as recording the number of printers ( or whatever ) that are available. ( That should make it clear why the counter can't be negative. ) Whenever a printer is allocated to a process, the count is decremented; and whenever a process finishes with a printer and returns it to the operating system, the count is incremented. That makes it clear what the initial value of the counter must be : it must be set to the number of resources which that semaphore is controlling. Clearly, if the value is to retain this significance, we can't just execute P and V indiscriminately; instead,

every execution of P ( corresponding to the allocation of a resource ) must normally be followed eventually by a single execution of V ( when the resource is released ). The normal pattern is therefore :

```
P( Printers );

..... do things with the allocated printer.

V( Printers );
```

Apart from that, P and V should be left strictly alone, except in two special circumstances, both directly related to the counting function. It is sensible to use V when a resource is added to the system, either when the system is started or at any subsequent time. Using V once during each addition makes sure that the counter ends up with the correct value. Similarly, it is reasonable to use P when a resource is removed from the system ( for maintenance or repair, for example ).

You might notice that the little code fragment above is quite like another little code fragment which we presented while discussing locks. That isn't accidental; consider a semaphore controlling a resource of just one item. Such a semaphore can only take on values of 0 and 1 ( so it is called a *binary semaphore* ) and it is exactly equivalent to a lock. Non-binary semaphores are sometimes called counting semaphores or general semaphores to distinguish them, but as there's no real distinction that isn't particularly helpful.

*There is no theoretical difference between a lock and a semaphore, but there's a practical difference : it's easier to implement test-and-set as an atomic hardware instruction than increment-and-return-the-value. That's because the new value to be stored in the variable for a lock is independent of the original value, while for a semaphore there is ( depending on the sort of semaphore you want ) an arithmetic or logical function to perform. Particularly with small processors, it's therefore often easier to implement locks than counting semaphores.*

The theoretical identity depends on both locks and semaphores being correctly implemented; there is no guarantee that an incorrectly implemented lock will be equivalent to a semaphore, binary or not ! In practice, the main difficulty in managing a correct implementation of a semaphore is just the same as that for a lock : the P and V operations must – by definition – be indivisible, which leads us straight back to the discussion about indivisible locks. This time, there is no easy answer corresponding to test-and-set, because the values of the semaphore variable are not restricted to *true* and *false*, so if we get the implementation wrong the result might well be that the semaphore variable ends up with an impossible value, implying that there are too many or too few resources.

And this is just a bit embarrassing, because the obvious answer is to use a lock to implement the semaphore. For example :

```
P( S ) :
     mine := false;
     repeat
          lock( S );
          if S > 0
          then begin
               S := S – 1;
               mine := true;
               end;
          unlock( S );
     until mine;
```

That will work – provided that we can implement the lock properly. Everything depends on locks; once again, we're unable to implement mutual exclusion unless we have some already.

You might reasonably wonder why the two semaphore operations were given such helpful names as P and V. The reason is that Edsger Dijkstra is Dutch, so, sensibly enough, used Dutch words in his original definition. We do not speak Dutch, but, according to one source, V stood for "verhogen" – to increment – and P stood for "proberen" – to test. Without in any way disparaging the Dutch language, we find these difficult to remember, and you will recall that we believe that instructions should be easy to remember and understand. In the interests of consistency, then, ( and for the sake of remembering which one is which ) we shall henceforth call them *signal* ( as in "signal that a resource is available" ) and *wait* ( as in "wait, if necessary, until a resource is available" ). Though not the only alternative names ( some texts call them *up* and *down* for reasons which might – or might not – become apparent later ), *wait* and *signal* are perhaps the most common.

Now we return to the problem of busy waiting. In Dijkstra's original description of semaphores there was no mention of what happens at the word "wait" in the P ( wait ) operation. What can happen ? So far, all we've seen is the busy wait, but we didn't like that much. The obvious alternative is perhaps to suspend the process for a while, then let it have another try, and keep going until it's lucky. That's an improvement so far as processor use is concerned, but not the way we'd like to run our operating system.

For such reasons, it has become standard to associate a queue with each semaphore. This is a much more disciplined way of controlling the resource allocation, and we have much more control over the rules – so, depending on how we organise the queue, we can offer a first-come-first-served service ( simple queue ), or take account of priorities ( with a priority queue ).

And so signal and wait become :

```
signal( S ) :
     if anything waiting on S then
          start the first process on the S queue
     else
          S := S + 1;

wait( S ) :
     if S < 1 then
          put this process on the S queue { this means the process
is stopped }
     else
          S := S – 1;
```

In this implementation the value of S always tells us how many of the resources are currently available. Also note how much more work the signal operation has to do compared to the simple V operation of Dijkstra.

An even more useful implementation is :

```
signal( S ) :
     S := S + 1;
     if S < 1 then
          start the first process on the S queue;

wait( S ) :
     S := S – 1;
     if S < 0 then
          put this process on the S queue;
```

Under this definition the semaphore S indicates two different things. If S is negative it means that there are abs( S ) processes waiting in the queue. If S is not negative it means that there are S of the resources available and that no processes are waiting in the queue.

The only reason semaphores work is that both semaphore operations are indivisible or atomic i.e. from the programmer's point of view a signal or a wait runs to completion before any other process can access the semaphore. It is possible to provide atomic semaphore instructions in hardware as with the testandset instructions, however it is easy to construct secure signal and wait routines by making entry to these mutually exclusive via the testandset lock we saw earlier.

Apart from the mutual exclusion aspect of semaphores they also provide a convenient method for *synchronizing* processes, as we mentioned when discussing interprocess communication. In this case one of the processes can signal the other which is waiting.

As an example of this we will look at the classic producer/consumer relationship. In which one process produces results which the other consumes or uses. ( Mutual exclusion is still here. The resource is the buffer which carries the result from the producer to the consumer. )

```
program producerconsumerrelationship;
var
     numberdeposited : semaphore;
     numberreceived : semaphore;
     numberbuffer : integer;

procedure producerprocess;
var
     nextresult : integer;
begin
     while true do begin
          calculate( nextresult );
          wait( numberreceived );
          numberbuffer := nextresult;
          signal( numberdeposited )
     end
end;

procedure consumerprocess;
var
     nextresult : integer;

begin
     while true do begin
          wait( numberdeposited );
          nextresult := numberbuffer;
          signal( numberreceived );
          use( nextresult )
     end
end;

begin
     semaphoreinitialize( numberdeposited, 0 );
     semaphoreinitialize( numberreceived, 1 );
     cobegin
          producerprocess;
          consumerprocess
     coend
end.
```

The cobegin ... coend pseudo-Pascal keywords indicate that all statements between them can theoretically run in parallel. In a multiprocessor both the producerprocess and

consumerprocess could actually run simultaneously, with the synchronization being handled by the semaphores.

One nice thing about this implementation of the producer/consumer problem is that it is possible to have several producer and consumer processes without changing our code ( except by adding more calls inside the cobegin ... coend ). This might be desirable, for example, if the consumption of the next result takes a lot longer than the production. One producer could keep many consumers happy.

In case you didn't notice this is a user level example. The operating system would provide the semaphore routines as system calls. Semaphores can also be used within the kernel to provide the mutual exclusion that operating system resources require.

Flashback to busy-waiting.

At the end of our discussion on the test-and-set instruction, we promised that the busy-waiting problems would be cleared up. This is it.

All is well if we use a simple lock to guarantee the indivisibility of the semaphore operations. At first sight, this is an outrageous statement : we've seen that the same problems turn up in both cases, so how can one cure the other ? But contemplate these two points : first, busy-waiting is only a problem when the resource concerned isn't available ( obvious ); and, second, access to semaphores is almost always possible, because – unlike the resources which they protect – they are only actively in use momentarily during the allocation and release operations. There is still no universally excellent solution – but there are satisfactory solutions for both cases :

- If there is only one processor, then we can implement the lock by preventing context switches while the semaphore operation is in progress – that is, by ensuring exclusive access to the processor.

- If there are several processors, then we cannot prevent multiple access by that method unless we also stop the other processors, which is bad in principle, and gets worse as the number of processors increases ( more processes mean both more interrupts and more disruption per interrupt ). A solution which only affects the requesting process is therefore necessary, so we have to fall back on a spin lock. But this is no longer such a dreadful thing; it doesn't stop all processing, because the other processors carry on, and we know that it won't be necessary for very long.

The trick works because we now have a two-level system : the semaphore controls a long-term lock, but can itself be implemented with a short-term lock, for which we can use methods which are not acceptable for long-term control.

Event counters.

Another method for guaranteeing results with concurrent programs, this time without using mutual exclusion, is the event counter, developed by Reed and Kanodia[EXE6] in 1977. The operations which work on event counters are advance, read, and await. **Event counters always start at zero.**

```
advance( E ) :
     E := E + 1;
     if any processes awaiting the new value of E then
          start these processes;

read( E ) :
     return the value of E;

await( E, count ) :
     if E < count then
```

```
put this process to sleep, awaiting E = count;
```

Because of the fact that event counters don't require mutual exclusion in order to work you find that the processes themselves have to put a bit more effort in as can be seen in the following producer/consumer solution, modelled on the semaphore solution.

```
program producerconsumerrelationship;
var
     numberdeposited : eventcount;
     numberreceived : eventcount;
     numberbuffer : integer;

procedure producerprocess;
var
     i : integer;
     nextresult : integer;

begin
     i := 0;
     while true do begin
          calculate( nextresult );
          i := i + 1;
          await( numberreceived, i – 1 );
          numberbuffer := nextresult;
          advance( numberdeposited )
     end
end;

procedure consumerprocess;
var
     i : integer;
     nextresult : integer;

begin
     i := 0;
     while true do begin
          i := i + 1;
          await( numberdeposited, i );
          nextresult := numberbuffer;
          advance( numberreceived );
          use( nextresult )
     end
end;

begin
     cobegin
          producerprocess;
          consumerprocess
     coend
end.
```

Event counters alone are not powerful enough to provide generalised mutual exclusion to a resource. The difficulty lies in determining what number the counter should wait for. We saw a way around this in the producer/consumer problem. There is a way to make event counters work as providers of mutual exclusion by adding a new concept of tickets. This is similar to being given a ticket in a shop and waiting for your turn to come up.

Messages.

Semaphores and event counters are rather low level ways of providing synchronization. Even though it might not be obvious at first it is also possible to coordinate processes with messages. A simple send and receive protocol is adequate.

```
send( toprocess, message );
```

sends the message to the process toprocess. toprocess might be any process from a group of processes. In other words the receiving process doesn't have to be unique.

```
receive( fromprocess, messagebuffer );
```

waits for a message from fromprocess and stores the message in messagebuffer. fromprocess may be any process from a group of processes.

Using this system our producer/consumer solution becomes :

```
procedure producerprocess;
var
     nextresult : integer;
begin
     while true do begin
          calculate( nextresult );
          send( consumerprocess, nextresult );
     end
end;

procedure consumerprocess;
var
     nextresult : integer;
begin
     while true do begin
          receive( producerprocess, nextresult );
          use( nextresult )
     end
end;
```

followed by the same main program as before.

This example seems to show that message passing is simpler than the other methods we have seen. However you need to realize that the producer/consumer problem is concerned with passing information from one process to another – which is exactly what message passing deals with.

It is clear that messages must not be lost to make this method work. As we have seen either the send must block until the message has been received or the message system must buffer messages until they are received. This leads us to the conclusion that although programming a solution with message passing is simpler than other solutions, there is more going on behind the scenes.

Compared to semaphores and event counters the message method of providing mutual exclusion has advantages and disadvantages. The most important advantage of using messages is that they can be used easily on distributed systems ( i.e. over a network ). The major disadvantage is the speed ( or lack of it ) with which processes can communicate.

Being careful.

Before we carry on we need to talk about the difficulty of correct concurrent programming. The producer/consumer example is one of the simplest and yet even experienced programmers can get it wrong. Here is a semaphore solution to the problem which was published in a very popular textbook ( here altered slightly to fit our conventions ).

```
program producerconsumerrelationship;
var
     exclusiveaccess : semaphore;
     numberdeposited : semaphore;
     numberbuffer : integer;

procedure producerprocess;
var
     nextresult : integer;
begin
     while true do begin
          calculate( nextresult );
          wait( exclusiveaccess );
          numberbuffer := nextresult;
          signal( exclusiveaccess );
          signal( numberdeposited )
     end
end;

procedure consumerprocess;
var
     nextresult : integer;
begin
     while true do begin
          wait( numberdeposited );
          wait( exclusiveaccess );
          nextresult := numberbuffer;
          signal( exclusiveaccess );
          use( nextresult )
     end
end;

begin
     semaphoreinitialize( exclusiveaccess, 1 );
     semaphoreinitialize( numberdeposited, 0 );
     cobegin
          producerprocess;
          consumerprocess
     coend
end.
```

This solution is actually modelled on one in Dijkstra's original semaphore paper. However this implementation of it is incorrect. The author makes sure that the consumer process will always wait until a result has been produced, however there is nothing to prevent the producer process getting out of step. It only has to wait for exclusive access to the buffer. This means there is nothing to stop it going around and around several times if the consumer process is held up for any reason after releasing the buffer. If this happens previously calculated results will be overwritten before they are consumed.

By the way, Dijkstra's solution didn't have this problem; his buffer was infinitely long and could accept results without overwriting.

So we see how easy it is to make a mistake when we try to coordinate activity between two or more processes. Since it is the operating system's job to make using the machine easier, wouldn't it be appropriate to provide assistance in this situation as well ?

Forgetting to unlock.

The most common danger is forgetting to unlock something which has been locked, thereby preventing any other process ever accessing that resource. This is especially easy to do when a section of code can be left in several ways.

Monitors.

A monitor is a mutual exclusion construct which can solve this problem along with some other security problems. Rather than giving each process the ability to control their own resource access ( which semaphores and event counters do ), a monitor is an external construct which completely controls use of a resource. There is no way to use a resource except by asking the monitor.

*The word "monitor" is thoroughly overloaded in the vocabulary of computing. There are at least four quite different meanings for the word. We have already mentioned one of them; when looking at the history of operating systems, we discussed monitor systems. The meaning in the context of this chapter is to do with concurrency. The appropriate meaning should always be obvious from the context – but take this as a warning to be careful !*

In a monitor not only are there shared resources, there are also shared routines to deal with those resources. In fact the only way the resources can be manipulated is with the routines provided by the monitor.

Not only does the monitor contain variables and procedures; it also restricts access to one process at a time. In other words, whenever a process wants to execute a shared routine the monitor checks to see if any other process is currently executing code inside the monitor. If there is such a process, the calling process has to wait until the process currently inside the monitor has left it. In fact a number of processes might be trying to access monitor routines at the same time. These processes will be stored in a queue awaiting service.

It is useful to think of the monitor as having one guarded entry point. This entry point queues processes and sends them to their required routines when it is their turn to enter the monitor.

Since only one process is allowed in the monitor at any time and the monitor is the only way to access the shared resources we are guaranteed mutual exclusion.

Sometimes it is necessary for a process to wait whilst still possessing mutual exclusion over a resource. This is a situation fraught with danger as we will see when discussing deadlock. In a monitor it does not make sense to stop all access to the monitor if the process currently running needs to wait for something, especially if that something can only be provided by another process running in the monitor. The solution is to have special variables known as condition variables.

There are wait and signal calls on condition variables just as we had with semaphores. The difference is that they are much simpler to implement, in that they are inside a protected area and there is no count associated with the variable.

A wait on a condition variable will stop the currently running process. In this case another process is then allowed to enter the monitor. The process will continue only after another process calls signal on the same condition variable. This means the associated resource is available. If no processes are waiting when there is a signal, nothing happens, and the signal is lost.

If more than one process is waiting on the same condition variable, the monitor must decide which one is to run.

There is a design problem which must be dealt with when talking about condition variables. When a signal indicates that a waiting process can proceed we will have two processes running in the monitor. One way of dealing with this is to hold all the signals until the currently running process either leaves the monitor or waits for some other condition. In this case the running process must ensure it does not change the condition

which it has signalled. A better solution halts the signalling process and restarts the signalled one. The simplest method is to force signalling processes to leave the monitor. In this way a signal is the last thing a process can do in the monitor.

The only way monitors can be used is if they are part of the language. We will see a monitor solution to a classic problem later.

The properties of monitors are those of their components and the interactions between them. Monitors encapsulate all or most of the items listed in this table :

| | |
|---|---|
| Resource : | the thing being protected or handed out. |
| Local data : | only accessible from inside the monitor. |
| Scheduler : | the monitor might have to make decisions about which process should enter next. The scheduler might ensure processes aren't postponed indefinitely. |
| Queues : | when a process is inside the monitor and has to wait for something it is put to sleep on one of these. There is a separate queue and associated condition variable for each reason that a process might wait whilst inside the monitor. When a process waiting on a condition variable is released it must run before a new process enters the monitor. |
| Procedures : | the routines which do all of the manipulation of the shared variables. |
| Initialization code : | for example, to set up the number of printers. |

Equivalence of solutions.

Now that we have seen a variety of solutions to the problem of providing mutual exclusion it is interesting to wonder which is the best solution. We have already seen some of the advantages semaphores and event counters have over messages and vice versa.

In one sense all of the methods we have looked at, simple locks, semaphores, event counters, messages and monitors, are equivalent. It is possible to implement any one of them using any other one. We will have a look at a couple of examples.

Semaphore implementation of a monitor.

As we have seen a semaphore is a relatively low level device to provide mutual exclusion. Using a semaphore to implement a monitor is almost trivial. The important thing about the monitor is that only one process is allowed to run inside it at a time. All we need to do is to place a semaphore at the entrance to the monitor. This semaphore is initialised to one, restricting the number of processes inside the monitor to one. Each process wanting to enter the monitor does a wait on this semaphore and most processes leaving the monitor signal it.

There is a complication with condition variables. We decided that the simplest way of handling signals on condition variables was to ensure that the signalling process then left the monitor. Now it must leave the monitor in a different way from that outlined above. If it does a semaphore signal on the guarding semaphore another waiting process might run. We have to guarantee that any processes waiting on the condition variable run first. So the signal on the condition variable must check to see if any processes are waiting on this variable. If a process is waiting it must be restarted. One way to accomplish this is by associating a semaphore with each condition variable[EXE7].

Message implementation of a semaphore.

As our second example we will construct a lower level mechanism, a semaphore, with a more complicated one, messages.

The method is to have a controlling process which receives all of the wait and signal messages and sends messages informing processes when they can continue running.

A wait is a send to this process followed by a receive waiting for a reply.

When a wait message is sent to this controlling process it checks the value of what is effectively the semaphore count. If this indicates the resource is free this count is decremented and it sends a reply message straight away to the requesting process. If the resource is not available the reply is not sent until a corresponding signal is received. In the meantime the requesting process is blocked waiting for a reply.

Hopefully these two examples show the approach to take in trying to implement any of the mutual exclusion constructs in terms of any other one.

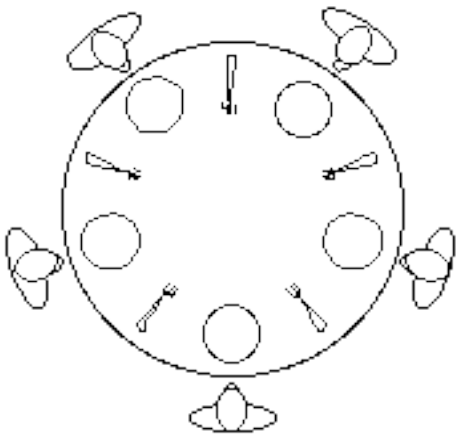
At a logical level these different constructs are equally powerful. At the practical level they all have their advantages and disadvantages. The most important being the trade-off between flexibility and safety. We have seen that monitors are safer in the sense that a programmer is less likely to make a mistake using them. This safety comes at the price of inflexibility. In certain situations ( SIGOS 1992 ) monitors can lead to far more context switching than a semaphore solution to the same coding problem.

## ANOTHER CLASSIC PROBLEM : THE DINING PHILOSOPHERS.

Another very well known problem, not because it occurs often but because it is so elegant, is the problem of the Dining Philosophers. This problem was developed by the father of the semaphore, Edsger Dijkstra.

- We have a round table around which are sitting five philosophers.
- In front of each philosopher is a plate with spaghetti.
- Each philosopher thinks for a while and then gets hungry and wants to eat for a while.

Each philosopher thinks and eats independently of the others. The times each one spends thinking and eating might vary.



The problem is that there are only five forks on the table, one between each pair of philosophers. To eat the spaghetti a philosopher must be holding the two forks immediately to the left and right. ( At this point someone usually comments that if that's the way philosophers go about hygiene, then it's no wonder that there are so few of them. Or that it's much easier to eat spaghetti with one fork anyway, so why don't we reformulate the problem with chopsticks ? These are both sound comments, but we are bound by the weight of tradition. )

Obviously not all the philosophers can eat at once. We must write a procedure for each philosopher in such a way that they can all carry on with their business with no philosophers starving to death whilst they meditate on the meaning of life or why people enrol in Computer Science courses. All the philosopher procedures are run concurrently.

Our first attempt using semaphores might be something like this :

```
procedure philosopher( name : integer );
```

```
begin
     while true do
     begin
          think;
          wait( name );  /* semaphore associated with the fork to
the right */
          wait( ( name+1 ) mod 5 ); /* the fork to the left */
          eat;
          signal( name );
          signal( ( name+1 ) mod 5 )
     end
end;
```

What happens if by coincidence all philosophers manage to get as far as getting the fork on their right ? Unfortunately we end up with five dead philosophers, they all starve.

This is an example of *deadlock* where a cycle of processes are each waiting on resources held by other processes in the cycle. We will talk about deadlock in detail later in the course.

To stop this problem we might try doing a wait on both forks at once.

```
     while true do
     begin
          think;
          simultaneous_wait( name, ( name+1 ) mod 5 );
          eat;
          simultaneous_signal( name, ( name+1 ) mod 5 )
     end;
```

Here we have introduced a new sort of mutual exclusion primitive, a simultaneous wait and signal. In this case the running process is suspended whenever either resource is unavailable. While it is suspended it doesn't hold either of the resources it is waiting for. It is only restarted when both resources are available for it.

Well, this certainly solves our deadlock problem. A philosopher never holds one fork at a time; always either two or none. This means that the philosopher will eventually finish eating and return both forks allowing those adjoining philosophers to try to grab two forks. The deadlock problem is solved.

Imagine what could happen to some unlucky philosopher though. Assume the philosophers are numbered from 1 to 5 around the table. Say it works out that whenever philosopher 1 stops eating philosopher 3 always starts and vice versa. Poor philosopher 2 will still starve. This is a case of indefinite postponement ( sometimes known as starvation ). Oh well, I suppose philosopher 2 just has to be philosophical about the situation.

It seems that every new mutual exclusion construct gets applied to the Dining Philosophers to show how elegant the new method is.

There is a nice Monte Carlo solution to this problem which entails two philosophers tossing a coin to solve contention for a fork. This always works in the average case but can still lead to indefinite postponement even though it is very unlikely.

Apart from the trivial solution of imposing mutual exclusion over any attempt to eat, allowing only one philosopher to eat at a time, most solutions to this problem entail keeping track of the states of the philosophers so that decisions can be made according to what surrounding philosophers are doing.

Here is our monitor solution. We hope this shows the power of monitors.

```
monitor ForkControl;
```

```
var
     forkAvail : array [ 0..4 ] of boolean;
     forkWait : array [ 0..4 ] of conditionVariable;

procedure GetBothForks( name : integer );
begin
     if not forkAvail[ name ] then
          Wait( forkWait[ name ] );
     forkAvail[ name ] := FALSE;

     if not forkAvail[ ( name + 1 ) mod 5 ] then
          Wait( forkWait[ ( name + 1 ) mod 5 ] );
     forkAvail[ ( name + 1 ) mod 5 ] := FALSE;
end;

procedure PutBackBothForks( name : integer );
begin
     forkAvail[ name ] := TRUE;
     forkAvail[ ( name + 1 ) mod 5 ] := TRUE;
     Signal( forkWait[ name ] );
     Signal( forkWait[ ( name + 1 ) mod 5 ] )
end;

begin
     for i := 0 to 4 do
          forkAvail[ i ] := TRUE
end. { monitor ForkControl }

Where each philosopher looks like

procedure Philosopher( name : integer );
begin
     while TRUE do
     begin
          Think;
          GetBothForks( name );
          Eat;
          PutBackBothForks( name )
     end
end;
```

Note on preemption ( What is this doing here ? ).

To preempt something generally means to get in before it and stop it from happening. When we talk of preemptive scheduling we mean that a currently running process can be stopped wherever it is in its code, and another process can be started. Usually preemptive scheduling is contrasted with non-preemptive scheduling ( obviously enough ) and cooperative scheduling.

Non-preemptive scheduling means that the process doesn't have to give up the processor, except when it wants to. It is tempting to think that this means that all processes would run to completion before another process can begin. In reality this is not the case. Processes don't always have all necessary resources. When a process cannot proceed until it has acquired some resource, typically input, there is no point in the process holding on to the processor and the request for the resource will lead to a context switch.

Cooperative scheduling is a specialised type of non-preemptive scheduling. With cooperative scheduling the processes are supposed to go out of their way to enable other processes to get a fair share of the processor. This means that someone, or something

( the compiler ) has inserted code into each process to request a context switch after every few ( hundred thousand ) instructions. Cooperative scheduling has usually been implemented on single user systems which were not intended to be multiprogramming when originally designed and the mechanisms for preempting processes do not exist. ( Several systems built on top of MS-DOS are good examples of this. The Macintosh dispatcher is also based on a cooperative scheme. )

Of course there is no guarantee that programmers will follow the rules of cooperative scheduling, this is why all multi-user operating systems have to use preemptive scheduling to ensure that users are treated fairly. On a single user system if one process is receiving more processor time than it should, this is mitigated by the fact that the work is being done for one person.

Whenever different solutions to the same problem are implemented in operating systems there are also hybrid solutions which try to get the best features of multiple solutions. As an example we have seen that Unix postpones context switches until the process is about to return to the user running state. This means that Unix doesn't implement strict preemptive process scheduling. There is a period of time that the process keeps running until it is ready to relinquish the processor. Of course the user level programmer has no say over the matter.

COMPARE :

Silberschatz and Galvin[INT4] : Chapter 6.

REFERENCES.

EXE5 : E.E. Dijkstra : "Cooperating sequential processes", in *Programming Languages* ( ed. F. Gennys, Academic Press, 1965 ).

EXE6 : D.P. Reed, R.K. Kanodia : "Synchronization with event counts and sequencers", *Proceedings of the sixth ACM Symposium on Operating Systems Principles,* 1977.

EXE7 : A.S. Tanenbaum : *Modern operating systems* ( Prentice-Hall, 1992 ).

---

QUESTIONS.

Consider the "disable interrupts" version of the lock( ) procedure. How can you guarantee that the interrupts will indeed be switched on when the new process starts ?

Modify the semaphore solution to the producer/consumer problem allowing a ring buffer for the output.

Earlier we said that the semaphore solution to the producer/consumer problems works unchanged with multiple consumers and producers. Does the event counter solution also scale correctly ? If not, explain what the difficulty is.

Modify the event counter solution to the producer/consumer problem allowing a ring buffer for the output.

How would the message passing producer/consumer problem be generalized to allow multiple producers and consumers ?

In some ways the traditional Unix kernel is like a monitor. Make a list of similarities and differences between the two.

Design the mutual exclusion primitives simultaneous_wait and simultaneous_signal mentioned above.

What unwanted limitations are included in our monitor solution to the Dining Philosopher's problem ?

Message passing can be equivalent in power to semaphores. How could you implement semaphores with an asynchronous message passing system similar to that shown above ?