Inter-Process Communication and Synchronisation

Cooperating processes

Up to now, we have assumed that processes are independent. However, this is not always the case. Typing `prog1|prog2` to the Unix shell causes two processes to be created, one for `prog1` and one for `prog2`. The processes need to communicate via the pipe – data must be passed from one to the other, and they must also synchronise – the `prog2` process cannot proceed until the `prog1` process passes it some data.

On a uniprocessor machine, cooperating processes will be active in the machine at the same time, but obviously only one can be running at any time – their execution will be interleaved, in chunks terminated by context switches. Exactly when the context switches occur depends on the exact pattern of interrupts. This will be different on every run of the same processes, and we cannot say anything about the exact pattern in which two cooperating processes will interleave, or about their relative execution speeds.

On a multiprocessor system, more than one process can actually be running at once (one on each processor), and two cooperating processes may therefore both be running at the same instant. It is still impossible to know which of the two processes will run faster though, and the relative speeds will still vary from run to run.

Therefore any synchronisation required between the two processes must be done explicitly. Whether running in a multi-tasking environment on a single processor, or on separate processors, the processes are effectively executing concurrently, but at varying and unpredictable rates.

Process communication via shared memory

A simple way for data to be communicated between processes is via variables in an area of memory accessible to all the processes concerned. However, synchronisation mechanisms are normally also required. As an example, consider the pair of concurrent processes in the diagram opposite.

Both increment a shared variable, `count`, to give an overall count of some interesting event which can happen in either process. Suppose `count` is currently 200, and process 1 encounters an interesting event, and begins to increment it, executing the instruction `lw $8,($10)`, and loading 200 into register `$8`. Process 1 can be interrupted at any time – suppose an interrupt occurs immediately following this instruction, and that process 2 gets the next timeslice, and while running, also encounters an interesting event, and increments `count` to 201. Some time later process 1 is dispatched again, its processor registers are restored, including 200 in `$8`, and it executes `addiu $8,$8,1`, followed by `sw $8,($10)`, placing 201 in `count`. An error has occurred - two interesting events should have been logged, one in each process, taking `count` to 202, not 201.
The error occurred because process 2 updated the shared variable while process 1 was in the middle of updating it. To prevent these errors, we must ensure that one process cannot change the variable while the other is updating it – *ie* during the bracketed sequence of instructions above, known as the *critical section*. One way to achieve this is to disable context switches for the duration of the critical section, but this is unsatisfactory in a preemptive multi-tasking environment, as there would be nothing to stop a program, once it entered a critical section, from looping within it, and thus locking out all other processes. A better solution is to allow context switching (after an interrupt) at any point in the user’s program, but to use an interlock to prevent more than one of the two (or more) processes which share a data area from being in their critical sections at the same time.

A neat mechanism which does this is based on an abstract data type called a *semaphore*. 

```
PROCESS 1

la    $10, count
lw    $8, ($10)
addiu $8, $8, 1
sw    $8, ($10)

SECTION

MAY CAUSE ERRORS

PROCESS 2

la    $10, count
lw    $8, ($10)
addiu $8, $8, 1
sw    $8, ($10)

CRITICAL SECTION

MAY CAUSE ERRORS
```
Semaphores

The simplest form of semaphore, the binary semaphore, is a binary value shared between two or more cooperating processes, which can be initialised to 0 or 1, and can then be operated upon by any of the processes which share it, using operations known as wait (sometimes called P) and signal (sometimes called V).

The two operations on a semaphore variable sem are defined by:

**wait(sem)** Wait until sem==1, then set sem=0 and proceed

**signal(sem)** Set sem=1

The two processes above may now be prevented from simultaneously entering their critical sections by giving them a shared semaphore sem1, and placing wait(sem1) immediately before each critical section, and signal(sem1) immediately afterwards:

```
PROCESS 1
         |         |
         |         |
la      $10,count
wait(sem1)
lw     $8,($10)
addiu $8,$8,1
sw     $8,($10)
signal(sem1)
```

```
PROCESS 2
         |         |
         |         |
la      $10,count
wait(sem1)
lw     $8,($10)
addiu $8,$8,1
sw     $8,($10)
signal(sem1)
```

The semaphore sem1 is initialised to 1. When the first process executes the wait, it will set sem1 to 0, and enter its critical section. If it is now interrupted, the other process may run quite happily, but will be prevented from entering its critical section by the wait operation, as sem1 is 0. When the first process leaves its critical section, it sets sem1 back to 1, and the other process will now be free to enter its critical section.

The semaphore initialisation and the wait and signal operations are implemented for user programs by system calls, and the semaphore variable itself is stored in the system area of memory. When a process makes a wait system call on a 1 valued semaphore, the semaphore is set to 0 and the system call returns to the calling process, so execution
continues. If a \texttt{wait} call is made on a 0 valued sempahore, the calling process is put into the \textit{blocked} state – it cannot continue, and a context switch occurs.

When a process makes a \texttt{signal} system call on a sempahore, if there is no process blocked waiting for that semaphore, the semaphore is set to 1 and the system call returns. If there is a waiting process, that process is switched to the \textit{ready} state, and the \texttt{signal} call returns with the semaphore remaining at 0.

This mechanism enforces \textit{mutual exclusion} on the shared variable \texttt{count} – mutual exclusion is a requirement on many other shared resources in a computer system, \textit{eg} I/O devices. Semaphores may also be used to enforce other types of synchronisation between concurrent processes, such as that required between a process producing data and a process consuming it (\textit{eg} the \texttt{prog1|prog2} example above).

\textbf{Inter-process communication by message passing}

So far we have considered inter-process communication using shared memory, and synchronisation using semaphores. However, it is not always feasible for processes to communicate via shared memory – in particular, some types of multiprocessor have separate memories for each processor – the processors are only connected by wires along which messages can be sent. Two cooperating processes running on different processors within the machine can now only communicate by sending each other messages.

To support this we can introduce operating system calls \texttt{send(channel,message)} which sends a message along a specific channel interconnecting two processes, and \texttt{receive(channel,message)} which is the corresponding receive call. \texttt{send} and \texttt{receive} can achieve synchronisation between processes as well as communication – \texttt{receive} may be likened to \texttt{wait} on a semaphore, and \texttt{send} corresponds to \texttt{signal}. If one of the two communicating processes reaches its \texttt{receive} call before the other process reaches the corresponding \texttt{send} call, the receiving process will block until the message arrives.