2 Programs and processes

A *program* is its code; a *process* is an instance of a program, together with its state: its data. There may be several copies of the same program running simultaneously, several processes sharing the same code. Choosing to share code requires that a program does not modify it code. Different processes will require segregated memory for their data.

Code must be able to refer to data by some form of *relative addressing*, where a *base register* refers to the data area of the current process and a constant in the code represents an address calculated by adding that constant to the base register.

Multitasking (multiprogramming) allows a number of processes to be run interleaved with each other on a single processor. Parallel slackness makes more efficient use of resources, for example by occupying the processor with a compute-intensive task while it would otherwise be waiting for slow input or output.

2.1 Multitasking

On a single processor, the program of a process executes until either it starts some slow I/O (by a system call), or it is interrupted. If there were nothing else to do, the process would then have to wait until it could resume, but multitasking allows for other process to be ready and waiting to take up the slack. When the kernel has dealt with the call, it can store the identity of the process (the state of the processor registers), and revive a hibernating process.

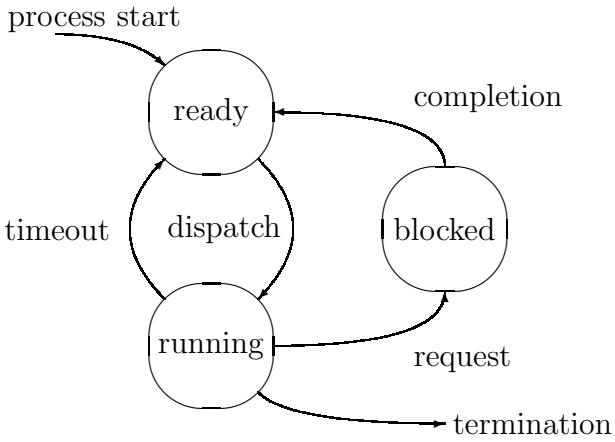
A process can be in one of three states:

running at most one process is current: its identity is in the processor registers;

blocked suspended and waiting for I/O: its identity may be stored in registers associated with the device; and

ready processes which are suspended but available to run if selected: their identity is stored in a data structure owned by the operating system.

The mechanism for multitasking allows processes to change state in these ways:



The step of *process creation* involves allocating spaces to be occupied by a process, loading a program into its code space, and setting up the descriptor of a ready process. In UNIX this is factored into two distinct activities:

```
int p = fork();
if (p == 0) {                    /* I am the child       */
    execve(command, path, args); /*   run 'command'      */
} else {                         /* I am the parent of p */
    waitpid(p, &status, 0);      /*   wait for p to exit */
}
```

The call to *fork*() returns twice: once in the calling (parent) process, and once in a brand new (child) process which is an almost identical copy of its parent. In the parent it returns the identity of the child; in the child it returns zero. The running processes in UNIX are usually kept in an array, and the process identity is an index into that array.

As a result of the subsequent call to *execve* or one of the many variants of *exec* the operating system overwrites the program running in the child process with the new program.

2.2 Scheduling

In contrast to the *mechanism*, the *policy* selects which ready process (of potentially many) is selected to become the running process. The mechanism makes things safe, the policy makes things fair. A *pre-emptive* policy will interrupt running processes that might otherwise not surrender control, so that even compute-intensive processes can be run whilst simultaneously guaranteeing a maximum period when other processes are not allowed to run.

The policy will choose next process to be run with the intention of optimising some aspect of performance. This might be throughput, or fairness, or (interactive) response time, or (batch) turnaround time. Particularly important might be graceful degradation under conditions of overload.

The policy might depend on parameters set when a process is created, such as a user-assigned priority, a class of process (batch/on-line/real-time), declared resource requirements (such as expected run-time). It might equally depend on measured history, such as whether the process is I/O bound or CPU bound (measured by frequency of I/O requests), resources consumed so far (such as accumulated run time), or the extent of delay so far.

Obvious policies include first-come first served, shortest job first, shortest expected remaining time, highest response ratio next, round robin. Real time systems in which there is a deadline by which a task *must* be completed will generally schedule processes by earliest deadline first.

2.3 Threads

Processes will generally be well insulated from each other, for example with distinct address spaces so that they cannot (inadvertently) alter each other's variables. Threads, in contrast, although they have their own process registers and stack and so on will share address spaces and are open to (presumably deliberately) interfering with each other's variables.

Threads have the advantage of cheaper context switching, and cheaper communications with each other (through shared variables) at the expense of considerably less protection from each other. The advantage is clearest in those operating systems (and machine architectures) which made process context switching unduly expensive.

Hyperthreading extends the idea of running several threads in a single address space by sharing as much as possible of the architecture of a machine between threads. Essentially only the registers are distinct, and two (or more) copies of the registers are used by as many threads, which execute instructions on the same processor. This provides parallel slackness inside the deepest levels of caching, which hides some of the (small) costs of missing even the fastest of caches in an architecture with several levels of caching.

Most of these descriptions are based on the assumption of a single processor, but the ideas are the same in multiple processor computers, or processors with multiple cores. The difference between those two is one of scale: how much of the memory system, how many caches (including virtual memory) are shared by processors. One significant difference between single processors and multiprocessor/multicore machines is the cost of implementing atomic operations, to which we will return.

Exercises

2.1 Suppose that a machine has to complete n jobs in sequence, and job i takes time t_i. Show that the total over all jobs of the time spent waiting for completion is minimised by running the jobs in ascending order of t_i.

2.2 Suppose that a machine has to complete n jobs in sequence, and job i which takes time t_i has to be completed by deadline d_i. Show that if this is possible at all, it is possible if the jobs are scheduled in order of increasing d_i.

2.4 Interprocess communication

The point of the process abstraction is to give the current program the impression that it has exclusive use of a machine. However, that protection from interference limits its usefulness to running entirely independent programs. In order for several processes to co-operate they need to be able to communicate, and such communication interferes with that interprocess protection.

The obvious way to communicate between processes is for one to alter the variables of another. However, shared mutable state is dangerous. It is no longer true that:

if $x = 0$ **then** $\{x = 0\}$ ... **end**

because some other process might alter x immediately after it is tested. Moreover it is no longer the case that $x := x + 1$ increments x: two assignments in different processes might interleave reading and writing of x so that the effect of two parallel assignments might be $x{:=}x{+}2$ or $x{:=}x{+}1$. Reading the value of a variable which does not fit into a single machine word is not guaranteed to return a consistent value which that variable ever had.

Assuming only atomic read and atomic write of small integers, there are some cunning algorithms which will allow two (but often no more than two) processes to co-operate, often waiting busily for each other to agree. They are generally difficult to understand and hard to scale. Some more significant operations on shared data need to be *atomic* in order to make progress. Significantly more powerful atomic primitives are necessary.

In single processor machines, it was usual to provide a test-and-set instruction which guaranteed that two memory cycles would be consecutive, with no intervening interference. With multiple processors and highly structured memories with distributed caches this sort of thing becomes too expensive to provide. Modern processors provide such things as conditional store instructions, which will write a location only if this write is the first since a read by the same processor. Given

The upper red semaphore is raised (safe to pass) and the yellow (distant) semaphore is at danger.

some such small atomic primitive, it is possible to build up higher level atomic actions which can be used by a programmer.

Historically, the first of these is the *semaphore*, invented by Dijkstra for the THE operating system.[1] Semaphores combine the testing and setting of a value with a queue of processes that have failed to pass a test, have been suspended, and need to be resumed when the test can be passed.

A semaphore is a non-negative integer variable with two atomic operations, *signal* and *wait*.[2] The *signal* operation increases the value of the semaphore, and the *wait* operation decreases it. Of course, if the semaphore cannot have a negative value *wait* on a semaphore already equal to zero cannot decrease it immediately. It is required to suspend the execution until a corresponding *signal* operation makes it possible. The suspended processes are kept in a queue associated with the semaphore.

Binary semaphores are restricted to the values zero and one, and provide a means to implement mutual exclusion between processes. Suppose some resource r is to be shared between several processes. Atomic access to r can be ensured by associating with it a binary semaphore, also shared by these processes. Suppose we call this $r.mutex$, and initialise it to one. If every process which uses r does so only after waiting on this semaphore and subsequently signals the semaphore

[1] Edsger W Dijkstra 1930–2002, in a paper *Co-operating sequential processes*. Technische Hogeschool Eindhoven, September 1965

[2] The word 'semaphore' is railway jargon for the traditional form of mechanical signalling device, colloquially a *signal*. The operations *wait* and *signal* were originally called P and V, for the Dutch *probeer te verlagen* (try to reduce) and *verhogen* (increase), but are also often known as *down* and *up*, *acquire* and *release*, *pend* and *post*, *procure* and *vacate*.

wait(*r.mutex*);
... code using r ...
signal(*r.mutex*);

then no other of these processes can be executing code which touches r while this process is doing so.

This generalises a little, to managing pools of resources. For example if we have n equivalent resources, such as printers, each to be allocated for exclusive use of one of a number of processes, this might be done with a semaphore initialised to n and an array of bits indicating whether a particular resource is allocated or not.

These resources could just be storage: for example the locations in an array being used to implement a buffer. A reader and a writer might use an array a of n variables as a circular buffer, with w as a write pointer and r as a read pointer, and c as a count of the number of elements present.

var a : **array** n;
var $w = 0$, $r = 0$, $c = 0$; — invariant: $(r + c)$ **mod** $n = w$

proc *write*(**val** x) — precondition: $c < n$
begin $a[w] := x$; $w := (w + 1)$ **mod** n; $c := c + 1$ **end**;

proc *read*(**var** x) — precondition: $0 < c$
begin $x := a[r]$; $r := (r + 1)$ **mod** n; $c := c - 1$ **end**

Of course, *write* should only be called when $c < n$, and *read* should only be called when $c > 0$. It would be better to use a pair of semaphores *full* and *empty*, with the value of *full* being c, and that of *empty* being $n - c$.

var a : **array** n;
sema *full* $= 0$, *empty* $= n$; — invariant: *full* + *empty* $= n$
var $w = 0$, $r = 0$, $c = 0$; — invariant: $(r + c)$ **mod** $n = w$

proc *write*(**val** x)
begin *wait*(*empty*);
$a[w] := x$; $w := (w + 1)$ **mod** n; $c := c + 1$;
signal(*full*)
end;

proc *read*(**var** x)
begin *wait*(*full*);
$x := a[r]$; $r := (r + 1)$ **mod** n; $c := c - 1$;
signal(*empty*)
end

We can now remove the variable c, which is equal to *full* and this pair of processes can now safely be used by a single writing process and a single reading process. This is because r and w are not shared, and the two semaphores guarantee that any particular element of a is only accessed by one of the two processes at any one time. The semaphores manage the handing over of rights of access from one process to the other.

However if two writers (or two readers) try to use the queue concurrently, they will interfere. These problems of mutual exclusion can be solved by another pair of semaphores.

```
var a : array n;
sema readable = 1, writable = 1;
sema full = 0, empty = n; — invariant: full + empty = n
var w = 0, r = 0; — invariant: (r + full) mod n = w

proc write(val x)
begin wait(empty);
            wait(writable);
                 a[w] := x; w := (w + 1) mod n;
            signal(writable);
      signal(full)
end;

proc read(var x)
begin wait(full);
            wait(readable);
                 x := a[r]; r := (r + 1) mod n;
            signal(readable);
      signal(empty)
end
```

The order of the *wait*(*empty*) and the *wait*(*writable*) turns out not to matter in this code, but supposeing we had chosen to achieve mutual exclusion by having a single semaphore *mutex* guaranteeing at most one of a reader and a writer at any one time, then

```
var a : array n;
sema mutex = 1;
sema full = 0, empty = n; — invariant: full + empty = n
var w = 0, r = 0; — invariant: (r + full) mod n = w
```

```
proc write(val x)
begin wait(empty);
          wait(mutex);
              a[w] := x; w := (w + 1) mod n;
          signal(mutex);
      signal(full)
end;

proc read(var x)
begin wait(full);
          wait(mutex);
              x := a[r]; r := (r + 1) mod n;
          signal(mutex);
      signal(empty)
end
```

would work, but

```
proc write(val x)
begin wait(mutex);
          wait(empty);
              a[w] := x; w := (w + 1) mod n;
          signal(full);
      signal(mutex)
end;

proc read(var x)
begin wait(mutex);
          wait(full);
              x := a[r]; r := (r + 1) mod n;
          signal(empty);
      signal(mutex)
end
```

would not. (It can lead to a deadlock: exercise 2.4.)

Semaphores are the *goto* of concurrent programming: whilst small examples are straightforward, it becomes more difficult when trying to write something larger. Proper operation relies on all the pieces of code that are involved obeying what can become an intricate protocol of signalling and waiting. Although these examples involve locally matched brackets of calls to *wait* and *signal*, in general this is not the case. It is easy to make small mistakes in the design which can lead to processes being left waiting on a semaphore which will never be signalled. Worse, it is remarkably easy to write code which is superficially convincing but has subtle failures of atomicity.

More structured synchronisation primitives can in turn be constructed from semaphores, forcing them to be used in more disciplined ways. Tony Hoare first published the idea of a *monitor* which is essentially a module only one of whose interface procedures can be being called at any given time. Similar ideas appear in more recent programming languages, for example the synchronised classes in Java.

In particular in cases where a resource is passed from process to process and eventually comes back by another route, a more natural synchronisation idea is the passing of a message. If the passing of a message is an atomic operation which involves a *rendezvous* between two processes, then anything which happened before that rendezvous in either processes is guaranteed to have happened strictly before anything that happened in the other. That guarantee can be used to manage interactions between processes. Buffered communication eliminates the rendezvous, and gives a weaker guarantee (that earlier events in the sender precede later ones in the receiver) but can also be used. If necessary, buffered messages can be used to arrange a rendezvous by returning acknowledgements.

Exercises

2.3 Write an outline (in some sort of pseudo-code) of procedures which can be called from several concurrent processes to allocate and release one of a pool of n equivalent resources as described in the notes.

> **sema** *nfree* = n;
> **bool** *free*[n]; — initially all *true*
> **proc** *allocate*(**var** i)
> — waits for a free resource and return its index in i
> **proc** *release*(i)
> — releases resource i, which it had previously been allocated

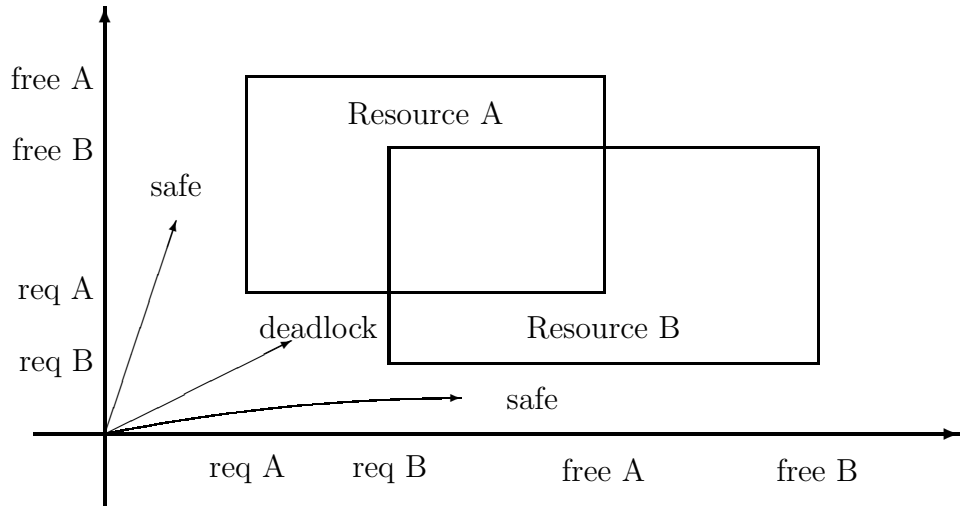
You will need another binary semaphore for a correct implementation.

2.4 Explain how a sequence of calls to incorrect code for the circular buffer (on page 16), using a single mutual exclusion semaphore, can lead to deadlock; an give an argument that the correct code using a single mutual exclusion semaphore cannot deadlock so long as there are some readers and some writers (and $n > 0$).

2.5 Deadlock

Whenever there is a need for synchronisation between concurrent agents the possibility exists that a number of agents may be prevented from making progress

because they cannot agree on that synchronisation. For example, if each of two processes requires simultaneous exclusive access to each of two resources, but the processes request the resources in opposite orders, it is possible that each process will pick up one resource and that both processes will then be trapped waiting for the other resource to become available. In this diagram the axes represent the passage of time in each process:



Possible trajectories are those that do not go left and do not go down; the boxes represent mutual exclusion on each of the resources, and the safe trajectories are those that can pass around the pair of boxes, but there is an unsafe area in the reflex corner in the bottom left. Once the system has entered this state, it is impossible to leave it, and these two processes become *deadlocked*.

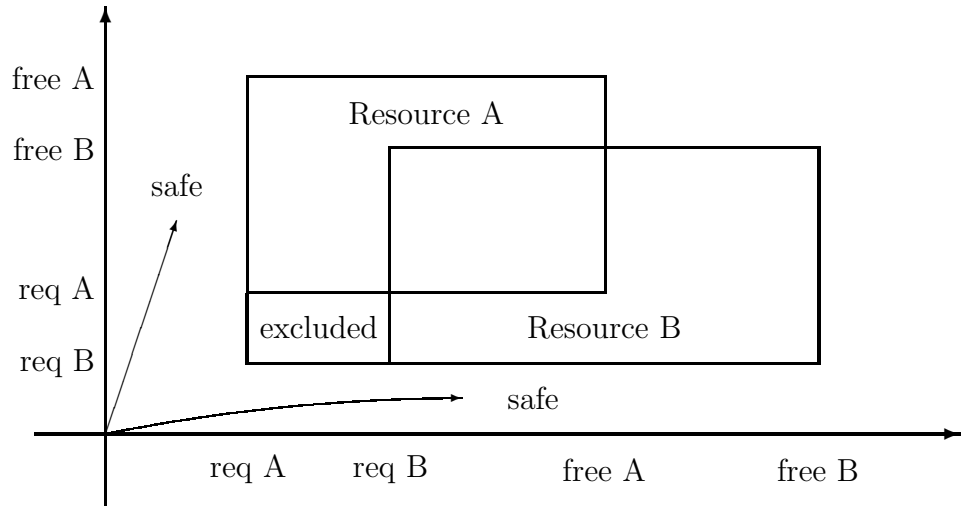
A straightforward solution to this particular instance of the problem would be to prioritise the resources, insisting that each process should acquire resource A before resource B. This sort of solution works in some cases, although it is both expensive because it causes too much synchronisation, and can destroy the symmetry of a problem. Asymmetric solutions to symmetric problems, as well as being bigger, may lead to unfair distribution of resources.

Deadlock requires four things: mutual exclusion; holding and waiting; no pre-emption; and a cycle of processes each waiting for the next. Insidiously, the possibility of deadlock or the guarantee of deadlock freedom is a property of a whole system. The relevant cycle of dependency may involve all of the parts of any decomposition.

The deadlock in this instance depends, for example on each process holding on to its first resource. One approach might be to detect the deadlock somehow, and let a process release its resource before trying again later. There are problems with this too: deadlock is in general hard to distinguish from a long wait, but worse the

solution may lead to repeated backing off and subsequent collision. Even if the processes are not starved by deadlock, starvation can still be caused by indefinitely many failed attempts to make progress. *Fair* solutions to conflicts like this are also expensive and difficult to implement in a distributed way.

The graphic in this instance suggests a way of preventing deadlock. All that is requires is to exclude any trajectory that enters the danger zone:



The *banker's algorithm* implements this exclusion by making each process declare what its maximum future resource requirements might be. Processes are only scheduled at times when the available resources are adequate to cover possible future requests from that process. In the small example, when a process requests its first resource it declares that it will need both A and B. If the one of these is already booked out, this process can make no further progress until that resource is returned.

Of course, the banker's ledger is now a centralised resource which must be accessed by all processes, and even this solution can lead to too much synchronisation.

Sometimes the best thing to do about the possibility of deadlock is to ignore it. Most operating systems, for example, are rife with ignored possible deadlock. In UNIX there is a table mapping process identifiers to their descriptors, and this table has a fixed size. If two or more processes between them require to create more children than will fit in the table, no serious attempt is made to prevent this. It just does not happen very often, so it is not worth worrying about.

Deadlock is a global property: a system is deadlocked when every component of a system is unable to make progress because it is waiting for some other component. In a deadlocked system, every component is waiting (for another). In general it cannot be detected by a local test, but it might be that it can be avoided by

local tactics which conspire to make a global strategy. In contrast to deadlock, a process can also be held up indefinitely in a busy system (sometimes called 'livelock') because resources are never allocated to that particular process. The system is making progress, but some particular component is starved because others are always consuming all of a resource. Livelock can only happen when there is an infinite amount of work being done by other components: there must (at some level of description) be an infinite stream of requests from elsewhere in the system.

Exercises

2.5 At traffic roundabouts in the UK and most of the civilised world, traffic already on the roundabout has priority over traffic joining the roundabout from a feeder road. In the absence of signs to the contrary in France the general rule of *priorité à droite* would give priority to traffic joining over that already on the roundabout. (*Place Charles de Gaulle* is notoriously like this.) When congested, each of these schemes can lead to starvation; explain how, and decide whether they are deadlock or livelock. Suggest ways of preventing starvation in each case.

2.6 The *dining philosophers* problem is a parable of Tony Hoare's based on a symmetric allocation problem posed by Edsger Dijkstra in 1965. For some large odd n, say five, each of n processes needs both of two resources at the same time, originally tape drives; process i needs resources i and $(i+1) \bmod n$.

In the parable the processes are called philosophers, and – between bouts of thinking – they sit at places at a round table to eat. They each require both of a pair of adjacent forks (which the share in an unhygenic way) to be able to eat an otherwise unmanageable shared bowl of spaghetti.

1. Show that deadlock is possible if all processes run the same determinstic program, requesting the resources one at a time.
2. Show that a system in which exactly one of the processes is left-handed (requesting its resources in the opposite order) is deadlock-free (for $n > 0$).
3. Show that there is a deadlock-free solution for *even* values of n if processes are alternately right-handed and left-handed.
4. Show that there is a deadlock-free solution even for odd n in which all of the processes are the same, but sharing a single shared counting semaphore initialised to $n - 1$.