Mutex – Peterson's algorithm

Peterson came up with a much simpler and more elegant (and generalizable) algorithm.

```
flag[i] = true;
turn = î;
while ( flag[î] && turn == î ) { }
/* critical section */
flag[i] = false;
```

Compulsory Exercise: show that this works. (Use textbooks if necessary.)

Monitors

Because solutions using semaphores have `wait` and `signal` separated in the code, they are hard to understand and check.

A *monitor* is an 'object' which provides some *methods*, all protected by a blocking mutex lock, so only one process can be 'in the monitor' at a time. Monitor local variables are only accessible from monitor methods.

Monitor methods may call:

- `cwait(c)` where *c* is a *condition variable* confined to the monitor: the process is suspended, and the monitor released for another process.
- `csignal(c)`: some process suspended on *c* is released and takes the monitor.

Unlike semaphores, `csignal` does nothing if no process is waiting.

What's the point? The monitor enforces mutex; and all the synchronization is inside the monitor methods, where it's easier to find and check.

This version of monitors has some drawbacks; there are refinements which work better.

Deadlock

We have already seen deadlock. In general, *deadlock* is the *permanent* blocking of two (or more) processes in a situation where each holds a resource the other needs, but will not release it until after obtaining the other's resource:

Process P

```
acquire(A);
acquire(B);
release(A);
release(B);
```

Process Q

```
acquire(B);
acquire(A);
release(B);
release(A);
```

Some example situations are:

- `A` is a disk file, `B` is a tape drive.
- `A` is an I/O port, `B` is a memory page.

Another instance of deadlock is message passing where two processes are each waiting for the other to send a message.

Preventing Deadlock

Deadlock requires three facts about system policy to be true:

- resources are held by only one process at a time
- a resource can be held while waiting for another
- processes do not unwillingly lose resources

If any of these does not hold, deadlock does not happen. If they are true, deadlock may happen if

- a circular dependency arises between resource requests

The first three can to some extent be prevented from holding, but not practically so. However, the fourth can be prevented by ordering resources, and requiring processes to acquire resources in increasing order.

Avoiding Deadlock

A more refined approach is to deny resource requests that might lead to deadlock. This requires processes to declare in advance the maximum resource they might need. Then when a process does request a resource, analyse whether granting the request might result in deadlock.

How do we do the analysis? If we grant the request, is there sufficient resource to allow one process to run to completion? And when it finishes (and releases its resources), can we run another? And so on. If not, we should deny (block) the original request.

Suggested Reading: Look up *banker's algorithm*. (E.g. Stallings 6.3).

Deadlock Detection

Even if we don't use deadlock avoidance, similar techniques can be used to detect whether deadlock *currently exists*. What can we do then?

- kill all deadlocked processes (!)
- selectively kill deadlocked processes
- forcibly remove resources from some processes (what does the process do?)
- if checkpoint-restart is available, roll back to pre-deadlock point, and hope it doesn't happen next time (!)