50004 - Operating Systems - Lecture 5

Oliver Killane

02/11/21

Lecture Recording

Lecture recording is available here

Process Synchronisation

The key concepts to consider in process synchronisation are:

- **Critical Sections**
  Section of code where processes access a shared resource
- **Mutual Exclusion**
  Multiple threads/processes cannot execute in a critical section simultaneously.
- **Atomic Operations**
  Operations that can occur without interruption or interleve by other threads, e.g reading and writing in 1 instruction.
- **Race Conditions**
  Where the behaviour of the program is dependent on the timing and interleving of threads.
- **Deadlock**
  Where the completion of tasks are mutually dependent, there is some loop in the dependency graph meaning a task's completion cannot occur before its completion.
- **Starvation**
- **Synchronisation Mechanisms**
  Locks, Semaphores and Monitors for example.

  Primitive that allow programmers to enfoce ordering of tasks, and exclusion. They are required at entry to and exit from a critical section.

Requirements of Mutual Exclusion

- No two processes can simultaneously be in a critical section.
- No process running outside the critical section can prevent another from entering the critical section.
- When no process is in the critical region, any process requesting permission to enter must be immediately admitted.
- No process requiring access to a critical section can be delayed indefinitely.
- No assumptions are made about the relative speed of processes.

Disabling Interrupts

```c
void Extract(int acc_no , int sum)
{
    CLI();
    int B = Acc[acc_no];
    Acc[acc_no] = B - sum;
    STI();
}
```

- Works only on single-core Systems (other core may access shared in memory).
- May never release the CPU if code buggy.
- Interrupts could be missed while interrupts disabled.
- Only possible in kernel code which has the ability to disable interrupts.

Software Solution - Strict Alternative

```c
while (true) {
    while (turn != 0)
        /* loop */ ;
    critical_section();
    turn = 1;
    noncritical_section0();
}
```

```c
while (true) {
    while (turn != 1)
        /* loop */ ;
    critical_section();
    turn = 0;
    noncritical_section0();
}
```

Issues:

- Cannot run 0 or 1 twice in a row without the other working.
- Busy waiting of thread on turn variable.
- If the noncritical section takes a long time, the other thread is blocked as it must wait for turn to be set. (Thread outside critical region prevent critical region access).
- Turn must be volatile to prevent compiler optimisations causing issues.

Busy Waiting

Continuously checking a value to determine if a thread can progress. It is a waste of CPU time and should only be used if the wait is short.

However as a result of actively polling the a value, it can stop waiting very quickly. Spinlocks used in OS kernels take advantage of this.

Peterson's Solution

```
int turn = 0;
int interested[2] = {0, 0};

// thread is 0 or 1
void enter_critical(int thread) {
    int other = 1 - thread;
    interested[thread] = 1;
    turn = other;
    while (turn == other && interested[other])
        /* loop */ ;
}

void leave_critical int thread) {
    interested[thread] = 0;
}
```

```
enter_critical(1);
critical_section();
leave_critical(1);
```

```
enter_critical(0);
critical_section();
leave_critical(0);
```

This lock busy waits while the the other is interested and it is the other's turn.

Atomic Operations

```
void Extract(int acc_no , int sum)
{
    int B = Acc[acc_no];
    Acc[acc_no] = B - sum;
}
```

Not atomic, other extract could be interleaved.

$\Rightarrow$

```
void Extract(int acc_no , int sum)
{
    Acc[acc_no] -= sum;
}
```

Not atomic, still requires multiple instructions, between which another process could be interleaved.

Locks

```
void Extract(int acc_no , int sum)
{
    lock(L);
    int B = Acc[acc_no];
    Acc[acc_no] = B - sum;
    unlock(L);
}
```

```
void lock(int *L)
{
    while (L != 0)
        /* wait */;
    *L = 1;
}
```

```
void unlock(int *L)
{
    *L = 0;
}
```

Here the implementation of the lock is flawed as we cannot atomically check the value of the lock, and set its value.

A working implementation is:

```
void lock(int *L)
{
```

```
    while(TSL(L) != 0)
        /* wait */ ;
}
```

TSL (test-and-set-lock) atomically sets the given memory location to 1 and returns the old value. Locks of this type are called **spin locks**.

Spin Locks

Spin locks should only be used when the expected wait time is short (otherwise wasteful). However it may run into the priority inversion problem.

Priority Inversion

When a lower priority thread is scheduled instead of a higher priority thread.

Threads $H > M > L$.
L acquires a lock, however is descheduled. H attempts tpo acquire the lock, but is blocked. The lock is not released as L needs to run in order to release it. M is higher priority than L, and runs instead.