MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Synchronisation (Part 2)

Daniel Playne
d.p.playne@massey.ac.nz

Race Condition

Race condition - when the overall behaviour of a multithreaded program depends on a particular sequence or timing of uncontrollable events.

Critical sections - a section of code where threads or processes access some shared resource or data.

Critical-Section Problem

The **critical-section problem** involves designing a protocol such that when multiple processes that share access to some resource can never execute their critical sections at the same time.

This involves each process performing some actions to request permission to enter the critical section and some actions to signal the other processes when they have left the critical section.

Critical-Section Problem

Requirements for a solution to the **critical-section** problem:

- **Mutual exclusion** - no two processes may be executing their critical sections at the same time.
- **Progress** - if no process is executing its critical section and some processes wish to enter their critical sections, only those processes participate in deciding which process should enter.
- **Bounded waiting** - there is an upper limit on the number of times other processes may enter their critical sections after a process has made a request to enter and before that request is granted.

Software solution: Peterson's Algorithm (two threads)

Bakery Algorithm

What about if there are more than two threads? Could think of multiple cars wanting to drive through an intersection.

Lamport's bakery algorithm is a method for providing mutual exclusion for multiple threads/processes that are attempting to access a shared resource.

Bakery Algorithm

The idea behind algorithm is that when each customer (process) enters the bakery (wants to enter a critical section) they must take a number from a ticket machine.

There is a globally visible sign showing the number of the customer currently being served (inside the critical section).

The others customers must wait until the sign shows their number at which point it is their turn to be served (enter the critical region).

Bakery Algorithm

This would be a simple solution if it wasn't for the race condition of taking a ticket from the ticket machine - in a computer system, two threads could both end up being assigned the same number.

The solution is to let two threads sometimes take the same number but the order they will be served depends on their thread id. The thread with the lowest id will get to enter before the thread with the higher id.

Bakery Algorithm

The bakery algorithm is as follows:

```
bool entering[N];
unsigned int number[N];
void lock(long id) {
   entering[id] = true;

   unsigned int max = 0;
   for(int i = 0; i < N; ++i) {
      max = (number[i] > max) ? number[i] : max;
   }

   number[id] = 1 + max;
   entering[id] = false;

   for(int j = 0; j < N; j++) {
      while(entering[j]) {}
      while((number[j] != 0) && ((number[j] < number[id]) ||
           ((number[j] == number[id]) && (j < id)))) {}
   }
}
```

Bakery Algorithm

Each thread will wait until any other thread that is currently entering has taken a number and until any thread with either a lower number or the same number and a lower id has already gone first.

Then the thread will enter the critical region. Unlocking is as simple as setting the thread's number back to 0.

```
void unlock(long id) {
    number[id] = 0;
}
```

Bakery Algorithm

In practice, we need to mark variables as `volatile` and use `__sync_synchronize()` to ensure memory transactions are made visible to the other cores in the correct order.

Synchronisation Hardware

Both Peterson's algorithm and Lamport's Bakery algorithm are software based solutions, as discussed they do have issues on modern computer architectures.

Another different mechanism for implementing locking (protecting critical sections with the use of locks) makes use of specially designed hardware instructions.

Synchronisation Hardware

The `test-and-set` or `TSL` (Test and Set Lock) instruction can read the value stored in a particular memory address and set the value to a non-zero value **atomically** - that is the instructions cannot be interrupted they must always be performed at the same time.

In addition to this, the instruction will lock the memory bus to prevent any other CPU from accessing memory until the instruction has been completed.

Synchronisation Hardware

The pseudocode for the `test-and-set` instruction, these operations are completed in a single, uninterruptible instruction and no other core can access memory in the middle of the instruction.

```
int test_and_set(int *lock) {
   int result = *lock;
   *lock = 1;
   return result;
}
```

Note: this is just pseudocode of what the instruction will do.

Synchronisation Hardware

This test-and-set instruction can be used to easily implement the lock() function for our critical-section programs.

```
void lock(int *lock_var) {
  while(test_and_set(lock_var) != 0) {
    // wait
  }
}
```

If *lock_var is 0 then it will be set to 1 and 0 will be returned and the thread will have locked the variable. If it was already 1 then the value will be unaffected and 1 returned, so the thread will have to try to lock again.

Synchronisation Hardware

The `unlock()` function for the critical-section programs is even easier to implement. The `*lock_var` can simply be set to 0.

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

On modern CPU architectures this actually requires slightly different instruction that sets `*lock_var` to 0 and ensures that all previous memory stores are globally visible.

Synchronisation Hardware

Other atomic instructions that can be used to implement locks is the `XCHG` (exchange) instruction or the `compare-and-swap` instruction.

The `XCHG` instruction will swap the contents of two memory locations (again in a single, uninterruptible instruction):

```
void xchg(int *lock, int *var) {
   int tmp = *var;
   *var = *lock;
   *lock = tmp;
}
```

Synchronisation Hardware

The `compare-and-swap` instruction will write a new value into a memory location as long as it has a certain expected value:

```
int compare_and_swap(int *lock, int expected, int new_value) {
    int result = *lock;

    if(result == expected) {
        *lock = new_value;
    }
    return result;
}
```

Synchronisation Hardware

These can be used to implement the lock function as follows:

```
void lock(int *lock_var) {
    int key = 1;
    while(key != 0) {
        xchg(lock_var, &key);
    }
}

void lock(int *lock_var) {
    while(compare_and_swap(&lock, 0, 1) != 0) {
        // wait
    }
}
```

Modern CPUs will provide at least one of `test_and_set`, `xchg` and `compare_and_swap`.

Synchronisation Hardware

Hardware synchronisation can be easily used to solve the critical section problem (regardless of how many threads are trying to access their critical sections).

Mutex Locks

The solutions to the critical-section problem are somewhat inaccessible to most programmers, especially when a small mistake can be very hard to spot and may cause intermittent errors.

Instead operating systems will often provide tools to allow programmers to more easily solve the critical-section problem.

The simplest of these is called a **mutex lock**.

Mutex Locks

The term **mutex** stands for **mut**ual **ex**clusion, a mutex lock can be used to protect critical regions by preventing more than one thread holding the lock at the same time.

Mutex locks can be implemented using one of the previous hardware methods, but often suffer from the problem of **busy waiting** where the threads will continuously execute in a loop trying to get the lock (these are called **spinlocks**).

```
void lock(unsigned int lock_var) {
   while(test_and_set(lock_var) != 0) {
      // busy-waiting
   }
}
```

Mutex Locks

Busy-waiting can be a serious problem for multiprogramming systems as it can waste a lot of CPU time that another process could be using to actually do something productive.

Spinlocks do have an advantage in that they do not require a context switch while a process is waiting on a lock.

If a lock is only ever held for a very short period of time then spinlocks can still be useful. Time wasted spinning may be less than time wasting doing a context switch.

Semaphores

Another tool that can be used to provide more sophisticated synchronisation between threads is called a **semaphore**.

A **semaphore** `S` has an integer counter that (other than initialisation) is only ever modified by two **atomic** operations - `wait()` and `signal()`. All modifications of S must be performed **indivisibly**.

```
void wait(S) {
   while(S <= 0) {}
   S--;
}
```

```
void signal(S) {
   S++;
}
```

Semaphores

Semaphores can be separated into **binary semaphores** and **counting semaphores**.

A binary semaphore acts a lot like a mutex and can be used in place of mutexes to provide mutual exclusion.

A counting semaphore could be used to allow a number of threads to access a finite number of instances.

Semaphores

A semaphore can be used for mutual exclusion by initialising it with `S=1`.

The first thread to call `wait(S)` will immediately succeed and decrement `S` to `0`, any other threads that call `wait(S)` will be stuck waiting. After the thread has finished the critical section it can call `signal(S)` to increment `S` to `1` and another thread's `wait(S)` will then succeed.

```
void lock(S) {
    wait(S);
}
```

```
void unlock(S) {
    signal(S);
}
```

Semaphores

Semaphores can also be used to provide synchronisation between processes other than just mutual exclusion.

If two processes *A* and *B* are both running concurrently and we must ensure process *A* only performs a certain task T_1 after process *B* has first completed a different task T_2.

These two processes can be synchronised by using a semaphore and making process *A* wait before performing task T_1 until process *B* signals that it has completed task T_2.

Semaphores

The semaphore `S` is initialised to `0`. If process A reaches the `wait` before process B has called `signal` it will stop and wait. Once process B has completed task T_2 it will call `signal` and process A will wake up and continue with task T_1.

Process A

```
...
wait(S);

// perform task T1
...
```

Process B

```
...
// perform task T2

signal(S);
...
```

What happens if process B calls `signal` before process A calls `wait`?

Summary

- Lamport's Bakery Algorithm
- Synchronisation Hardware
- `test-and-set`, `compare-and-swap`, `exchg`
- Mutex Locks
- Semaphores