MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Synchronisation (Part 3)

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

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 L) {
   while(test_and_set(L)!=0) {
      // busy-waiting
   }
}
```

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

Mutex Locks

A different approach would be to put the thread to sleep (into the waiting state) if it doesn't get the lock. It will wait until it is woken up by another thread.

```
void lock(unsigned int L) {
    while(test_and_set(L)!=0) {
        block(L);
    }
}
```

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

The `wakeup(L)` function will wake up a thread currently waiting on the lock.

Mutex Locks

Unfortunately this has a race condition, where is it?

```
void lock(unsigned int L) {
    while(test_and_set(L)!=0) {
        block(L);
    }
}
```

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

Mutex Locks

One solution would be to have a **wakeup bit** that it set if a process has a *wakeup* stored. If a process tries to go to sleep when the wakeup bit is set, it will simply continue instead of sleeping.

This approach could help for two processes, but what about if there are three or four? One bit will no longer suffice, instead we could count the number of wakeups that have been stored.

Semaphores

This approach is another tool that can be used to provide more sophisticated synchronisation between processes and 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++;
}
```

Semaphore Implementation

The `block()` and `wakeup()` functions have been replaced with `wait()` and `signal()` that will keep track of the signals that have been stored.

Any number of process can call `wait()` and `signal()` on the semaphore.

However, there is still the question of how to implement them.

Semaphore Implementation

The previous (very rough) definition of a semaphore would suffer from the same problem as the mutex locks we have discussed - **busy waiting**.

When a process calls `wait` and discovers that the semaphore value is `<= 0` then the semaphore will be stuck in busy waiting until another process calls `signal`.

Semaphores

We could rewrite our semaphores to use `block()` and `wakeup()`.

```
void wait(S) {
   S -= 1;
   if(S < 0) {
      block(S);
   }
}
```

```
void signal(S) {
   S += 1;
   if(S <= 0) {
      wakeup(S);
   }
}
```

These functions must be atomic - performed indivisibly.

Semaphores

Single-processor machines would sometimes implement this by disabling interrupts while calling the semaphore functions, however on multi-processor machines this approach requires interrupts to be disabled on all the cores and impacts performance.

Instead we could use our previous definition of mutexes to provide mutual exclusion. We'll still have busy-waiting but we know that the semaphore functions are very short so in this case our spinlocks won't have to wait long.

Semaphores

A first attempt might look like this:

```
void wait(S) {
  while(test_and_set(L)!=0){}
  S -= 1;
  if(S < 0) {
    block(S);
  }
  L = 0;
}
```

```
void signal(S) {
  while(test_and_set(L)!=0){}
  S += 1;
  if(S <= 0) {
    wakeup(S);
  }
  L = 0;
}
```

But if the `wait` has to block, it will never release the lock.

Semaphores

We could try to correct it with:

```
void wait(S) {
  while(test_and_set(L)!=0){}
  S -= 1;
  if(S < 0) {
    L = 0;
    block(S);
  } else {
    L = 0;
  }
}
```

```
void signal(S) {
  while(test_and_set(L)!=0){}
  S += 1;
  if(S <= 0) {
    wakeup(S);
  }
  L = 0;
}
```

But now we have a race-condition again - what to do?

Semaphore Implementation

The approach used to avoid busy-waiting is to maintain a list of processes that are currently waiting for a semaphore.

Whenever a process has to block, it will add itself to the list of processes waiting on the semaphore and switch to a waiting state.

A waiting process can be restarted when another process calls `signal` which will remove the process from the waiting list and wake it up - a **wakeup bit** is used to ensure that the waiting process doesn't lose its wakeup.

Semaphore Implementation

A conceptual implementation of these semaphores would look like this:

```
void wait(S) {
  S -= 1;
  if(S < 0) {
    add_to_list(P, S);
    block(P);
  }
}
```

```
void signal(S) {
  S += 1;
  if(S <= 0) {
    P = remove_from_list(S);
    wakeup(P);
  }
}
```

`block()` will suspend the process that calls it and `wakeup(P)` will cause process P to resume execution.

Semaphore Implementation

This relies on some carefully configured behaviour in `block` and `wakeup`.

`block` will put the process into a waiting state, unless a bit has been set that marks the process as awake.

`wakeup` will set the bit of the process to awake and and try to move it into the ready state.

Semaphore Implementation

This must be very carefully implemented and tends to get tied into other aspects of the operating system management.

See:
`https://elixir.bootlin.com/linux/latest/source/kernel/locking/semaphore.c#L205`
for details on how Linux implements semaphores.

Semaphore Implementation

Another advantage of this implementation is that the queue of waiting processes can be implemented as a FIFO queue.

This will help to satisfy the requirement of bounded-waiting, each process only has to wait for the processes that joined the queue ahead of it to complete.

Semaphore Implementation

Summary of semaphores

- Avoid busy-waiting, waiting processes will sleep.
- Keep a count of the number of waiting processes/stored wakeups.
- `wait()` and `signal()` can be made atomic with hardware synchronisation.
- Implementation is easy to get wrong.
- FIFO queue can guarantee bounded-waiting.

POSIX Semaphores

POSIX provides a simple API for semaphores.

`sem_t`	semaphore type
`sem_init`	initialise a semaphore
`sem_wait`	call wait on a semaphore
`sem_post`	signal a semaphore

The `sem_init` function can specify the value to initialise a semaphore to as well as a flag that sets whether the semaphore is shared between threads of a process or between processes.

Windows Semaphores

Windows also provides a similar API.

`HANDLE`	semaphore type
`CreateSemaphore`	initialise a semaphore
`WaitForSigleObject`	call wait on a semaphore
`ReleaseSemaphore`	signal a semaphore

Producer Consumer - Semaphores

Semaphores can be used to very easily implement the producer-consumer model. A very simple method would be to have a shared variable that the producer will write to and the consumer will read from.

The producer will wait on a semaphore called `empty` before it writes to the shared variable and will then signal another semaphore `full`.

The consumer will do the opposite and wait on the semaphore `full`, read from the shared variable and then signal `empty`.

Producer Consumer - Semaphores

```c
// Shared buffer
sem_t full, empty;
unsigned int buffer;

// Producer
void* producer(void *arg) {
    while(true) {
        // Produce an item
        unsigned int item = produce_item();

        // Wait for an empty buffer
        sem_wait(&empty);

        // Insert item
        buffer = item;

        // Signal item ready
        sem_post(&full);
    }
    return NULL;
}
```

Producer Consumer - Semaphores

```
// Shared buffer
sem_t full, empty;
unsigned int buffer;

// Consumer
void* consumer(void *arg) {
    while(true) {
        // Wait for an item
        sem_wait(&full);

        // Get item
        unsigned int item = buffer;

        // Signal buffer empty
        sem_post(&empty);

        // Consume item
        consume_item(item);
    }
    return NULL;
}
```

Producer Consumer - Semaphores

For these semaphores to function as intended we must think about how they should be initialised.

At the start of the program, the buffer can be considered empty. The first time the producer waits for the empty semaphore it should immediately succeed.

However, the first time the consumer calls wait it should not succeed until the producer has put the first item into the buffer.

```
// Initialise Semaphores
sem_init(&full, 0, 0);
sem_init(&empty, 0, 1);
```

Producer Consumer - Semaphores

This solution will probably end up with the threads having to sleep and wakeup a lot as there is only space for one item in the buffer.

A nicer solution would be to create a larger buffer that the threads can interact with.

Once again, semaphores give us an easy way to do this by keeping track of the number of items and available space in the buffer.

Producer Consumer - Semaphores

```
sem_t full, empty;
unsigned int buffer[BUFFER_SIZE];
unsigned int in = 0, out = 0;

// Producer
void* producer(void *arg) {
   while(true) {
      // Produce an item
      unsigned int item = produce_item();

      // Wait for an empty buffer
      sem_wait(&empty);

      // Insert item
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;

      // Signal item ready
      sem_post(&full);
   }
   return NULL;
}
```

Producer Consumer - Semaphores

```c
sem_t full, empty;
unsigned int buffer[BUFFER_SIZE];
unsigned int in = 0, out = 0;

// Consumer
void* consumer(void *arg) {
   while(true) {
      // Wait for an item
      sem_wait(&full);

      // Get item
      unsigned int item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

      // Signal buffer empty
      sem_post(&empty);

      // Consume item
      consume_item(item);
   }
   return NULL;
}
```

Producer Consumer - Semaphores

In this case the values of two semaphores represent the number of available empty slots in the buffer and the number of items in the buffer.

Initially the buffer is empty so there are no items in the buffer and all the slots are empty.

```
// Initialise Semaphores
sem_init(&full, 0, 0);
sem_init(&empty, 0, BUFFER_SIZE);
```

Summary

- `block()`, `wakeup()`
- Semaphores
- Semaphore Implementation
- POSIX & Windows Semaphores
- Producer-Consumer with Semaphores