## Overview

- We examined some of the problems that can arise when we move to a multithreaded programming model
- Threads executing concurrently in the same address space can interrupt each other at inopportune moments
- If more than one thread is concurrently executing in a critical section (where shared data is manipulated) then bad things can happen (data integrity cannot be ensured)
- We now look at how we might prevent this from happening by implementing mutual exclusion across critical sections

## Mutual Exclusion in Software

Turn-taking

- Will the following enforce mutual exclusion?

```
Thread A                    Thread B
while (turn == B) {}        while (turn == A) {}
/* CS */                    /* CS */
turn = B;                   turn = A;
```

- What are the advantages and drawbacks to this approach to implementing mutual exclusion?

## Mutual Exclusion in Hardware

Interrupt inhibition

- A thread either removes itself from the CPU or is forcefully preempted by the operating system
- It is the scheduler that decides when to preempt a running thread and it is only invoked on a clock interrupt
- Does that give you an idea how to stop others interrupting a thread while it is executing in a critical section?
- What are the advantages and drawbacks to this approach to implementing mutual exclusion?

## Mutual Exclusion in Hardware

Exchange

- There is an x86 assembly instruction `XCHG` that atomically swaps the contents of two locations in memory
- With `XCHG` we can implement mutual exclusion:

```
/* CS entry */              /* CS exit */
Local = 1;                  Global = 0;
REPEAT
  XCHG(Local, Global);
UNTIL Local == 0
```

- Each thread has its own `Local` while `Global` is shared
- What are the advantages and drawbacks to this approach to implementing mutual exclusion?

## Summary

| Approach | Scalability | Busy waiting | Privileges |
|---|---|---|---|
| Turn-taking | - | - | + |
| Disabling interrupts | + | + | - |
| XCHG | + | - | + |

- Turn-taking is out because of scalability issues
- Disabling interrupts is out because it is a privileged instruction and our solution must work with userland threads
- That leaves us with `XCHG` as a possibility...

## Busy Waiting

- The problem with `XCHG` arises when the critical section is a long one: chances are high a thread will be switched out while executing inside it
- Another thread trying to enter that critical section will busy wait, wasting its timeslice spinning in a loop
- If the CS were short (20-30 machine instructions) the problem would disappear since the chance of a thread being switched out while inside would be negligible
- What are we going to do about those long critical sections?
- Maybe by exploiting the atomicity of `XCHG` we can build more sophisticated synchronisation mechanisms that can cope with longer critical sections

## The Semaphore

- First general solution to the problem, proposed by Dijkstra
- A semaphore is a non-negative integer which apart from initialisation can be acted upon only by two atomic, uninterruptible operations
- We can WAIT on a semaphore
- We can SIGNAL a semaphore
- What are the effects of executing a WAIT or SIGNAL on a semaphore? Well, that depends...

## WAIT

- A thread that WAITs on a positive-valued semaphore simply returns after decrementing its value
- Not the same as assignment i.e. not equivalent to $s = s - 1$ because the atomicity of the decrement is ensured and the problem of "lost updates" does not arise
- Using assignment to decrement the semaphore its final value could vary as assignment compiles to 3 instructions (copy to register, decrement, copy to memory)
- A thread that WAITs on a zero-valued semaphore removes itself from the CPU and sleeps on a queue associated with that semaphore

## SIGNAL

- Increments the value of a semaphore in one atomic, indivisible operation
- However, if there is a thread waiting on that semaphore then SIGNAL has a different effect
- The semaphore's value does not change but the waiting thread is woken, marked runnable and moved to the run queue; when next it runs the woken thread will continue at the instruction following the WAIT that put it to sleep
- If several threads are waiting on the semaphore then the decision on which one to wake depends on the scheduling policy

## Mutual Exclusion with a Semaphore

- A single semaphore, initialised to 1 can be used for mutual exclusion (also called a mutex)
- Here the mutex is called `Guard`:

```
WAIT(Guard)
/* ... CS ... */
SIGNAL(Guard)
```

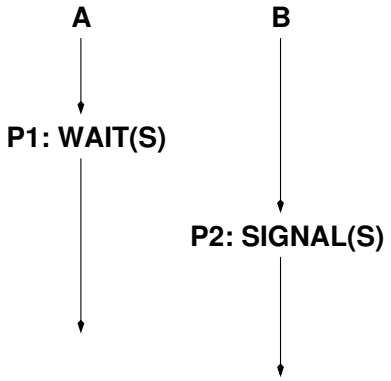
- An arriving thread WAITs on the semaphore; if no one is in the CS (the semaphore's value is 1) the WAIT succeeds and the thread enters the CS
- If another thread is already in the CS the semaphore will be zero and the arriving thread will sleep until the mutex holder exits the CS and SIGNALs the semaphore

## Synchronisation with a Semaphore

- Thread A must not pass point P1 until thread B has reached point P2 in its code
- One semaphore initialised to 0
- Thread A WAITs immediately before P1
- Thread B SIGNALs immediately after P2

Two scenarios arise. . .

## Synchronisation with a Semaphore



## Producer and Consumer with Semaphores

- As mentioned previously, there is a whole class of problems that involve a producer thread placing data in a buffer and a consumer thread taking it out
- We have the following constraints:
  1. A consumer cannot take data from an empty buffer
  2. A producer cannot insert data into a full buffer
  3. The buffer (shared data) must be protected from concurrent access by the two threads

## Producer and Consumer with Semaphores

- We need three semaphores to solve the problem
- `Guard`, initialised to 1, enforces mutual exclusion across the CS
- `Slot`, initialised to the number of slots in the buffer, is waited on by the producer and signalled by the consumer
- `Data`, initialised to 0, is waited on by the consumer and signalled by the producer
- Two pointers control where data is inserted and removed, `NextIn` and `NextOut`
- Note the order in which semaphores are waited upon and signalled; what could happen if we switched it?

## Producer and Consumer with Semaphores

Producer

```
Produce an item
WAIT(Slot)
WAIT(Guard)
Insert data at NextIn
NextIn++
SIGNAL(Guard)
SIGNAL(Data)
```

Consumer

```
WAIT(Data)
WAIT(Guard)
Get data at NextOut
NextOut++
SIGNAL(Guard)
SIGNAL(Slot)
```

## Semaphore Implementation

- Implementing a semaphore is not trivial: the WAIT and SIGNAL operations must be atomic and there are scheduler interactions to be handled
- WAIT must be atomic or two threads might simultaneously acquire a mutex and so must SIGNAL or only one thread might be woken when two resources were freed
- How can we get this atomicity?
- Because they are so short we can now use a hardware spinlock (e.g. one based on `XCHG`) across the critical sections of WAIT and SIGNAL
- We are building higher level synchronisation primitives by exploiting the lower level ones provided by the hardware

## Semaphore Implementation

- We have busy waiting but it has been moved from the entry point to our application's critical section to the entry point to WAIT and SIGNAL's critical sections
- As long as the semaphore's critical section is short enough, chances are a thread will acquire and release the spinlock before its timeslice expires meaning the spinlock is almost always available and little or no busy waiting occurs
- It is unlikely a thread will sleep while holding a spinlock
- Compare this to the situation where we wrap a spinlock around an application's critical section (containing thousands of instructions)

## Semaphore Implementation