Synchronisation for Processes in OS

The Critical Section Problem

Concurrent access to shared data results in data inconsistency

Maintaining consistency requires checks to ensure proper cooperation and execution

Problem: Consumer and Producer share finite buffer
Producer produces items and puts into buffer
Consumer takes out items
Situations when producer waiting with full buffer and consumers waiting with empty buffer

Producer

```
while(true)
{
        // produce item and put in nextProduced
        while(count == BUFFER_SIZE); //wait at full buffer
        buffer[in] = nextProduced; // store new item
        in = (in + 1) % BUFFER_SIZE; // increment new pointer
        count++; // increment counter
}
```

Consumer

```
while(true)
{
        while(count == 0); //do nothing
        nextConsumerd = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        //consumer item in nextConsumed
}
```

Race condition

In the code above, `count++` and `count--` can lead to inconsistencies since producer and consumer race to update the value.

Synchronisation primitives for critical sections

Solutions for concurrent modification of data in *critical sections* require
Mutual exclusion - If process Pi is in critical section (cs) with variables that can be modified, no other process can be in the cs
Progress - No process outside should (remainder section, rs), should block another waiting to enter
Bounded waiting - Limits exist on number of processes entering cs before the

current one can enter (aka entry should be fair for all processes)
Should also not matter how fast or slow each process is

Peterson's Solution

gfg

Two process solution assuming CPU's LOAD and STORE instructions are atomic (complete fully and cannot be stopped part way)

Processes share variables
`int turn;` - indicates whose turn in cs
`bool wants_in[2];` - indicate if process wants to enter cs

```
do
{
        wants_in[i] = TRUE; // I want access...
        turn = j; // but, please, you go first

        while (wants_in[j] && turn == j); // if you are waiting and it is your turn, I will wait.

        //[critical section]

        wants_in[i] = FALSE; // I no longer want access

        //[remainder section]
} while (TRUE);
```

When both processes interested, achieve fairness through `turn`, causing alternate access. If no turn variable, processes race to enter no process currently in cs

Points to note
-What if we want to support >2 processes?
-If Pj goes to after critical section but before switching turns, Pi is waiting and wastes time/process power

Synchronisation hardware

Many systems provide HW support for cs

Uniprocessors disable interrupts
-Currently running code executes without preemption
-Generally inefficient on multiprocessors
-Delay in one processor telling others to disable their interrupts
Modern machines provide special atomic instructions
TestAndSet - Tests mem address (ie read) and set
Swap - Swap contents of two mem addresses
Used to implement simple locks for mutual exclusion

General lock pattern

```
do
{
```

```
        [acquire lock]
                [cs]
        [release lock]
                [rs]
}while(true);
```

TestAndSet

```
bool TestAndSet(bool* target)
{
        bool org = *target; //Store original value
        *target = true; //set variable to true
        return org; //return original value
}
```

Overall, sets a variable to true and returns original value
Useful as it allows us to guarantee that only our thread changed the changed value to true.

Solution with TestAndSet allows us to have a `lock` variable and if we change the value of `lock`, then we know only our process can enter since only we changed the value

Inefficient Spinning

Consider

```
do
{
        while (TestAndSet(&lock)) ; // wait until we successfully change lock from false to true
        [critical section]
        lock = FALSE; // Release lock [remainder section]
} while (true);
```

Guarantees mutual exclusion at high CPU usage until can enter cs

Rather than having a process spin (*spinlock*), implement a sleep/wake mechanism where waiting processes sleep and when process finishes cs, wakes other processes up to enter

Solution to this problem is implemented in OSs and they release the lock *during* the `sleep()` call so with a guarantee that it will not be interrupted
Lock is reacquired when woken up again just before we return from a `sleep()` call. This is implemented as a *sleeping lock* called a *sempahore*

Semaphores

-Simplifies synchronisation
-Does not require busy waiting
-Guaranteed **bounded waiting** and **progress**

Consists of
-Semaphore type S, which records waiting processes and an int
-Two atomic operations to modify S, wait() and signal()

Works like:
-Semaphore initialised with count = max processes allowed in cs
-When wait() called, if count == 0 then add to list of sleepers and blocks otherwise decrements count and enters cs
-When process exits cs, calls signal() which increments count and wakesup process head of list if one exists
The FIFO list allows for *bounded waiting*

In this process, the int is the *count* for the number of processes in the cs at a time. The max number can even be 1 (Binary semaphore) for complete mutual exclusion (only allowing one at a time). Once *count* reaches 0, no more processes allowed so sleeps

Critical section with semaphore

```
Semaphore mutex;
do
{
        wait(mutex); //blocks, kinda like if mutex_lock(mutex) where blocks if couldn't acquire the lock
        [critical section]
        signal(mutex); //frees mutex lock
        [remainder section]
}while(true);
```

Deadlock and Priority Inversion

Deadlock: Multiple processes wait indefinitely for event only one waiting process can cause

E.g.

Process 1	Process 2
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Priority Inversion: When lower priority process holds lock higher priority process needs
example

Semaphore examples

-Bounded Buffer Problem
-Reader/Writers Problem

Bounded Buffer

-A buffer holds N items, each item is considered a "slot"
-Semaphore *mutex* init to 1
-Sempahore *full_slots* init to 0
-Semaphore *empty_slots* init to N

Producer

```
while(true)
{
        wait(empty_slots); // Wait for, then claim empty slot
        wait(mutex);

        [critical section]

        signal(mutex);
        signal(full_slots); //Signal that another slot available
}
```

Consumer

```
while(true)
{
        wait(full_slots); //Wait for then claim a full slot
        wait(mutex);

        [critical section]

        signal(mutex);
        signal(empty_slots); //Signal that slot has become empty
}
```

Reader/Writer Problem

-Data set shared among processes with some only reading and some writing and reading
-Allow multiple readers at a time with no writers or only one writer at a time

-Semaphore *mutex* init to 1
-Semaphore *wrt* init to 1
-Integer *read_count* init to 0

Writer

```
while(true)
{
        wait(wrt);

        [critical section] //perform writing
```

```
        signal(wrt);
}
```

Reader

```
while(true)
{
        wait(mutex);

        read_count++;

        if(read_count == 1) //reading so lock out writers
        {
                wait(wrt)
        }
        signal(mutex); //release for other readers to enter

        [crtical section] //perform reading

        wait(mutex);
        read_count--; //decrement as we leave
        if(read_count == 0)
        {
                signal(wrt); // if last reader, allow writer to attempt to join
        }
        signal(mutex);
}
```

Linux kernel semaphores

`wait()` is `mutex_lock()`
`signal()` is `mutex_unlock()`

`down_read()` and `down_write` are read-write binary semaphores

counting semaphores also exist (although we didn't use them in assignments)

Summary

-Ensure that *cs* is executed in specific order
-Software solutions exist but very complex
-Need atomic ops (*TestAndSet*) supported by HW
-Synchronisation primitives (*semaphores* and *spinlocks*)
-Linux kernel implements these primitives