Concurrency and Process Synchronisation

Comp 305 lecture 3

Lecture Plan

- Critical Section
- Semaphores (revision)
- Classical Synchronisation Problems
- Higher Level Synchronisation Structures
  - Critical Regions
  - Monitors
  - Condition Variables

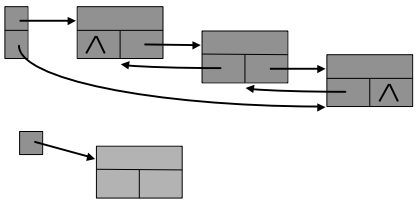
Concurrency and Process Synchronisation

- Concurrency is the actual or apparent simultaneous execution of processes.
- Concurrency on single processor systems is effected by timeslicing the CPU.
- Cooperative processes running concurrently often need to access shared data.

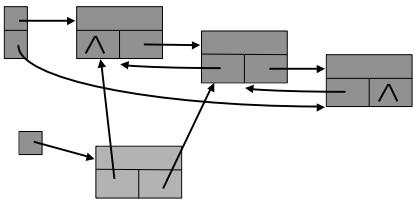
Concurrency and Process Synchronisation

- **Problem**: two or more concurrent processes which share data in an uncontrolled way, can cause that data to become inconsistent between the processes and ultimately incorrect when the shared data is updated.
- **Solution**: control or synchronise access to shared data to ensure it conforms to some sequential ordering of process execution.

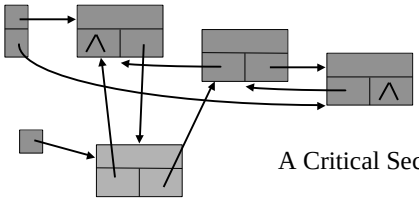
Concurrency Problem - Example



Concurrency Problem - Example



Concurrency Problem - Example



Critical-Sections

- We can identify in each cooperating process, code where shared data such as variables and files are altered, these sections of code are termed critical sections.
- If we ensure that no two cooperating processes are executing in their critical sections concurrently, then we can guarantee the consistency of the shared data.

The Critical Section Problem

- To design a protocol that allows processes to arbitrate the use of their respective critical sections.
- Any solution must satisfy:
  - mutual exclusion
  - progress
  - bounded waiting

Critical Section Terminology

- Entry section: used to negotiate entry into the critical section
- Exit section: advises that the process has finished executing in its critical section.

```
while(true){
  Entry Section
    critical section
  Exit Section
    remainder section
}
```


Two-Process Solution

- Let us first consider a solution involving only two processes. The obvious approach is to let each process take turns, that is, to strictly alternate.

```
while(true){
    while (turn != i);
    // Critical Section
    turn = j;
    // remainder
}
```


Two-Process Solution

- However, this solution does not satisfy the **progress** requirement.
- If p_i wants to enter the critical section, and the turn is j, then it cannot, even if p_j is only executing in the remainder.

```
while(true){
    while (turn != i);
    // Critical Section
    turn = j;
    // remainder
}
```

Two Process Solution, Part II.

- An alternative to strict alternation is to keep more state, and record which processes want to enter their critical section.

```
bool[] flag = new bool[2];

while(true){
  flag[i] = true;
  while(flag[j]);
  // critical section
  flag[i] = false;
  // remainder section
}
```


Two-Process Solution, Part II

- However, we still haven't solved the progress problem.

If processes i and j both execute the entry assignment in sequence...

We get deadlock, with both processes stuck in their while loops.

```
bool[] flag = new bool[2];

while(true){
  flag[i] = true;
  while(flag[j]);
  // critical section
  flag[i] = false;
  // remainder section
}
```


Correct Two-Process Solution

- If both of the ideas in part I and II are combined, then we can satisfy all three requirements.
- Essentially we use the boolean array to record which processes (i & j) wish to enter their critical sections.
- And resolve potential deadlock with the alternation from the first attempted solution.

Peterson's Solution

```
bool[] flag = new bool[2];
int turn;

while(true){
   flag[i] = true;
   turn = j;
   while(flag[j] && turn == j);
   //critical section
   flag[i] = false;
   //remainder
}
```


Multi-Processes

- Now we have the correct solution to the Critical Section problem for 2 processes we can consider synchronising N processes.

Lamport's Algorithm

- This algorithm is often called the bakery algorithm as it imitates the procedure of taking a ticket, a traditional means of imposing order in busy bakeries etc.
- The idea is simple, on entering the shop, each customer is issued a number.
- The customer with the lowest number is served next.

Lamport's Algorithm

- However, there is no guarantee that two customers did not receive the same number.
- In case of a tie, the customer with the lowest name is served first.

Lamport's Algorithm

```
int[] number = new int[N];          // shared data, init to 0
bool[] choosing = new bool[N];      // shared data, init false

    choosing[i] = true;
    number[i] = max(number)+1;
    choosing[i] = false;

    for(int j = 0; j < N; j++){
        while (choosing[j]);        // wait til j has finished choosing number
        while (number[j] != 0 && ((number[j],j) < (number[i],i)));
    }
    // critical section

    number[i] = 0;
    }
```


Hardware Solutions

- One observation from the previous section is that these software solutions to synchronisation are rather complicated and cumbersome.
- Fortunately many systems provide hardware instructions or system-calls to make the programming task easier and more efficient.

Atomic Instructions

- The main problem with software solutions is the lack of atomicy.
- Seemingly atomic high level programming structures are translated into a series of machine instructions, between which the CPU can be surrendered to another process.
- The solution is to provide special synchronisation instructions that operate in one indivisible unit.

Test and Set

- Test and Set takes one boolean argument and returns the original value.

Assume all parameters are Var (by reference)

```
bool testAndSet(bool lock){
    bool tmp = lock;
    lock = true;
    return tmp;
}
```


Test and Set

```
initialise lock = false;

while(testAndSet(lock));
// critical section
lock = false;
```

test	Set	Action
F	T	proceed
T	T	wait

Swap

- Swap is an atomic instruction that simply exchanges two values.

```
void swap(bool a, bool b){
        bool tmp = a;
        a = b;
        b = tmp;
}
```


Swap

- Swap is used a bit differently to testAndSet

```
key = true;
do{
    swap(lock,key);
}while(key);
```


Bounded Waiting

- TestAndSet and swap both satisfy the requirements of progress and mutual exclusion.
- Bounded waiting is not satisfied, but is simple to add an additional datastructure to ensure FIFO type behaviour.

Semaphores (Dijkstra - 1965)

- Semaphores introduce a cleaner interface to process synchronisation.

```
void P(int S){
   while(S ==0);
   S--;
}
```

```
void V(int S){
        S++;
}
```


Semaphores

- Semaphores are a more general synchronisation tool than first appears.
- If we need to establish a fixed order on two processes p_1 & p_2 , We set S = 0.
- If the order is p_1 -> p_2, we simply make p_2 issue a P(S), and it then must wait until p_1 issues a V(S);

Counting Semaphores

- The general semaphore can also be used to allow N processes into a critical section, simply by initialising S to N. Thus the first N processes each decrement S, until S == 0, at which time new processes must wait.
- This is ideal for controlling access to a limited number of resources.

Binary Semaphores

- A binary semaphore is a specialisation of the counting semaphore, with values of: S = {0,1}, and S is initially == 1.
- This is now used to control exclusive entry into critical sections.
- Of course we are at the mercy of balanced P(S) and V(S) commands.

Spinlocks (Review)

- All of the solutions to the synchronisation problem shown so far have involved busy waiting, and are termed spinlocks.
- The Nachos version of Semaphores discussed in the tutorials overcomes busy-waiting by blocking.
- Otherwise they are identical.

Blocking Semaphores

```
void P(int S){
  while(S==0){
      waiting->enQueue(currentThread);
      currentThread->sleep();
  }
  S--;
}
```


Blocking Semaphores

```
void V(int S){
  readyQueue->enQueue(wait->deQueue());
  S++;
}
```


Examples (Readers and Writers)

- Readers cause no change in state to the shared data (a file for example)
- Writers change the state of the object, reading and writing.
- Multiple readers are not a synchronisation problem, as no state is effected.
- Multiple writers or writers and readers are a recipe for disaster.

Readers and Writers

- If multiple readers are permitted, then there are further issues about when a writer can access the object.
- We must wait for all readers to complete reading before issuing the lock to the writer.
  - but we can either, let no new readers start,
  - or make the writer wait until there are no readers waiting.
- Both solutions may result in starvation

Readers and Writers

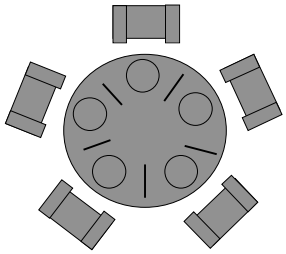
```
P(wrt);
    // writing
V(wrt);
```

```
P(mutex);
    readers++;
    if(readers == 1) P(wrt);
V(mutex)

// reading

P(mutex)
    readers--;
    if(!readers) V(wrt)
V(mutex)
```


Applet Animation

http://www.mcs.vuw.ac.nz/~kris/305/RW/RW.html

Dining-Philosophers



Dining Philosophers

- The simple solution to the philosophers problem is to represent each chopstick by a semaphore.
- A chopstick is picked up with:
  P(chopstick[pos]),
  and set down with:
  V(chopstick[pos]).

Simple Solution

```
while(true){
   P(chopstick[left]);
   P(chopstick[right]);
       // eat
   V(chopstick[left]);
   V(chopstick[right]);
       // think
}
```


Deadlock

- While the simple solution to the dining philosophers problem prevents neighbours from eating simultaneously, there is a problem if the philosophers all simultaneously pick up their left chopstick.
- If all left chopsticks are held, no philosopher can grab their right chopstick (anothers left) and will therefore wait forever. This is *deadlock.*

Applet Animation

http://www.mcs.vuw.ac.nz/~kris/305/DP/DP.html

Deadlock Prevention

- How can we stop the dining philosophers from becoming deadlock?
- Limit the number of philosophers to 4.
- Pick up the chopsticks only if both are available (in a critical section) .
- Alternate the order that chopsticks are picked up in depending on whether the philosopher is seated at an odd or even seat.

The Problem with Semaphores

- Semaphores are very susceptible to programming errors:
  - The reversal of a P() and V() pair will cause the mutual exclusion to be violated,
  - and the omission of either a P() or the V() will potentially cause deadlock
- These problems are difficult to debug and potentially difficult to reproduce.

Higher Level Synchronisation

- To cope with this a number of higher level programming language constructs exist:
  - Condition Variables
  - Monitors
  - Critical Regions
- These constructs lead not only to safer, but also to neater and more understandable code.

Higher Level Synchronisation

- Lets consider a good solution to the producer-consumer using counting semaphores to allocate the buffers...

mutex = 1, empty =N and full = 0

```
bool put(int X){
    P(empty);
    P(mutex);
    buf[in] = X;
    in = (in++) % N;
    V(mutex);
    V(full);
    return true;
}
```

```
bool get(int X){
    P(full);
    P(mutex);
    buf[out] = X;
    out = (out++) % N;
    V(mutex);
    V(empty);
    return true;
}
```


Condition Variables

- The use of 3 separate semaphores suggest the need for a better solution.
- The general problem is the need to wait for a condition to become true within a critical section.
- We can define a condition-variable, on which processes are queued, waiting on the condition variable.

Condition Variables

- All operations on a condition variable must occur while the process performing the operation holds the lock or semaphore.
- A condition-variable has two operations:
  - wait(S) // place process on CV
  - signal(S) // wake a process on CV
- Nachos features an addition operation:
  - broadcast(S) // wake all processes on CV

Condition-Variables Example

```
Semaphore mutex;        // this is now a Semaphore class
Condition empty, full;  // Condition class

bool put(int X){
    mutex ->P();
    while(n=N) empty->wait(mutex);
    buf[in] = X;
    in = (in++) % N;
    n++;  // number of full buffers
    full->signal(mutex);
    mutex ->V();
    return true;
}

bool get(int X){
    mutex ->P();
    while(n=0) full->wait(mutex);
    X = buf[out];
    out = (out++) % N;
    n--;
    empty->signal(mutex);
    mutex ->V();
    return true;
}
```


Condition-Variables

- A couple of caveats:
  - the lock/semaphore is released in wait and the process is blocked in one atomic action.
  - the lock/semaphore passed to wait and signal *must* be the same.

Critical Regions (Brinch-Hansen 1971)

- The fundamental ideas are that :
  - shared variables ***v*** are explicitly declared,
    VAR v: SHARED item;
  - the variable(s) may only be accessed within an explicit region ***s*** (effectively a critical section).

    REGION v DO s;

Critical Regions (Example)

```
VAR buf: SHARED record
    queue ARRAY [0..n-1] of item;
    in, out, count: 0..n;
end;

PROCEDURE in(X:item)
BEGIN
 REGION buf DO
    IF(count < n) THEN
        queue[in] := X;
        in := (in +1) MOD n;
        count := count +1;
END
```

← Critical Region

Conditional Critical Regions

- This example does not wait for the buffer to be empty or full. We need to consider entering the region based on a condition.
- A conditional critical region is simply a critical region, with the addition of a boolean test prior to entry of the region.

  REGION v WHEN b DO s;

Conditional Critical Regions (Example)

```
VAR buf: SHARED record
    queue ARRAY [0..n-1] of item;
    in, out, count: 0..n;
end;

PROCEDURE in(X:item)
BEGIN
 REGION buf WHEN (count < n) DO
    BEGIN
       queue[in] := X;
       in := (in +1) MOD n;
       count := count +1;
    END;
END;
```

Condition

Critical Region

Conditional Critical Regions -Variation

- There is also an interesting variation on the Conditional Critical Region:

```
REGION v DO
BEGIN
    S1;
    AWAIT b;
    S2;
END;
```

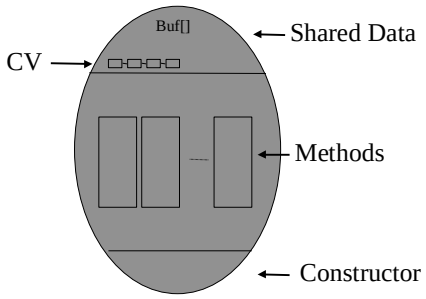

Monitors

- A monitor is equivalent to an object, with all state (data) private.
- Plus the addition of synchronisation, where invoking any method results in mutual exclusion over the entire object.

Condition Variables in Monitors

- In addition to basic mutual exclusion we need additional constructs to allow us to wait on conditions (akin to the condition variable).
- Condition x;
  - x->wait(); // suspend process
  - x->signal(); // wake exactly one process

The Monitor



Condition Variables in Monitors

- What happens when a signal is issued?
- The unblocked process will be placed on the ready queue and resume from the statement following the wait.
- This may violate mutual exclusion with both the signaller and signalled process executing in the monitor.

Condition Variable Semantics

- We have two choices:
  - The signalling process must wait until the signalled process has left the monitor, or waits on another condition.
  - The signalled process must wait for the signaller to leave the monitor or wait on another condition.
- A less powerful compromise is for the signaller to leave the monitor immediately.

Monitors (Example)

- See Overhead

A Real Example - Java

- Java offers monitor like synchronisation, with a couple of interesting differences.
- In place of the whole object being mutually exclusive, java allows individual methods within an object to form a monitor like structure.

Java Synchronisation

- Essentially the idea is to attach the *synchronized* keyword to individual methods.
- This means that at any time there is at most one thread executing in ANY of the *synchronized* methods within an object.
- Methods in the class that are not *synchronized* are completely free for all.

Java Synchronisation

- Java offers a condition variable equivalent for synchronised objects:
  - wait() // blocks the calling thread
  - notifyAll() // wakes *all* threads waiting on // this object.
- There is only one CV per synchronised object - but it is automatic.
- The notifying thread does not relinquish the lock on the monitor until it exits or waits.

Java Example

- See Overhead

Higher Level Synchronisation - Summary

- Just before we go, a couple of caveats:
  - Higher level structures do not and can not eliminate programming errors (especially logical errors).
  - Higher level structures are tools to aid the programmer in reducing logical and usage errors with Synchronisation primitives.