Comp 204: Computer Systems and Their Implementation

Lecture 7: Synchronisation

Today

- Mutual Exclusion
- Synchronisation methods
  - Test-and-Set
  - Wait and Signal
  - Semaphores
- Classic synchronisation problems
  - The readers-writers problem

Problem

- Suppose we have an object (called ‘thing’) which has the following method:

```
public void inc() {
  count = count + 1;
}
```

- Count is private to ‘thing’, and is initially zero
- Two threads, T1 and T2, both execute the following:

```
thing.inc();
```

Indeterminacy

- Assume each thread executes following code:

```
LOAD count
ADD 1
STORE count
```

- Could execute as follows

```
          T1                          T2
0         LOAD
1         ADD                          ..
          ..                           ..
          ..                LOAD       0
          ..                ADD        1
1         STORE             STORE      1
                             ..
```

- This is known as a ‘race’ condition

Mutual Exclusion

- Indeterminacy arises because of possible simultaneous access to a shared resource
  - The variable ‘count’ in the example
- Solution is to allow only one thread to access ‘count’ at any one time; all others must be excluded
- To control access to such a shared resource we declare the section of code in which the thread/process accesses the resource to be the critical region/section
- We can then regulate access to the critical region
  - When one thread is executing in its critical region, no other thread/ process is allowed to execute in its critical region
  - This is known as mutual exclusion

Indeterminacy and Interrupts

- Uniprocessor environment
  - Indeterminacy can be avoided by prohibiting interrupts from occurring while a shared variable is being modified
    - Current sequence of instructions executed without unexpected modifications to the shared variable
- Multiprocessor environment
  - Prohibiting interrupts not feasible as it is time consuming and decreases system efficiency
  - Many machines provide special hardware instructions to deal with the problem

Synchronisation

- Problems such as indeterminacy require protocols that processes/threads can use to co-operate to perform their tasks effectively (process synchronisation)

- The success of any synchronisation mechanism depends upon OS's ability to make a resource unavailable to other processes/threads whilst it is being used by another: enforcing mutual exclusion
  - such resources can include data files, I/O devices, a storage location etc.

- A key part of synchronisation is ensuring that no job is left waiting indefinitely

Synchronisation

- Several mechanisms are available to provide co-operation and communication amongst processes/ threads

- A common theme runs through all such mechanisms: allowing one process/thread to finish work on a critical region of a program/resource before other processes/ threads have access to it

- Synchronisation solutions exist in the form of both hardware and software mechanisms

Locks and Keys

- Synchronisation can be implemented as a lock-and-key arrangement: before a process can access a critical region it is required to obtain the key
- Once the key is obtained all other processes are locked out until it finishes its work
- When process holding the key finishes its work, it unlocks entry to the critical region and releases the key so another process can obtain it and subsequently access the critical region
- There are two actions involved here:
  - 1) first see if the key is available
  - 2) if the key is available the process must obtain the lock to ensure the resource is unavailable to all other processes

Lock-and-Key

- For the lock and key mechanism to work both the previous actions must be performed in the same machine cycle
  - If not, it is possible that while the first process is ready to pick up the key, another could come along, find the key available and prepare to pick it up, causing each to block the other out

- There are several such locking mechanisms that have been developed for process synchronisation:
  - Test-and-Set
  - WAIT and SIGNAL
  - semaphores…

Test-and-Set

- Test-and-Set is a single indivisible machine instruction (known as TS) which tests to see if the key is available and if so, sets it to unavailable, all in one machine cycle
  - TS was developed by IBM for its multiprocessing computers
- Actual key is a single bit that contains 0 if it is free and 1 if it is busy
- TS can be viewed as a subprogram with the following properties:
  - It has one parameter: the storage location of the key
  - It returns one value: the condition of the key (busy/free)
  - It takes only one machine cycle

Test-and-Set

- Advantages:
  - Simple procedure to implement
  - Works well for a small number of processes
- Disadvantages:
  - When many processes are waiting to enter a critical region starvation could occur as processes gain access to the critical region in an arbitrary manner
    - Could be solved using a first-come, first-served policy
  - Waiting processes remain in unproductive, resource-consuming ‘wait loops’: this is known as busy waiting

Wait and Signal

- Wait and Signal is a modification of TS
- Makes use of two mutually exclusive operations: WAIT and SIGNAL
- WAIT is activated when the process encounters a ‘busy’ condition code:
  - Sets the process to the blocked state
  - Links the blocked process to a queue of those waiting to enter the critical region
  - OS selects another process to execute

Wait and Signal

- SIGNAL is activated when the process exits the critical region and the condition code is set to ‘free’:
  - Checks the queue of processes waiting to enter the critical region and selects one, moving it to the ‘ready’ state
  - OS will eventually choose this process for execution
- Wait and Signal operations free processes from busy waiting and transfer control back to the OS to run other jobs while the waiting processes are idle

Semaphores

- A semaphore is an integer-valued variable that is used as a flag to signal when a resource is free and can be accessed

- Only two operations possible: P and V
  - also called *test* and *increment*, or *down* and *up*

```
P(S) {
   while (S<=0)
     ; //null
   S--;
}
```

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

Semaphores

- P and V are indivisible
  - When one thread/process modifies the semaphore, no other thread/process can modify that same semaphore
- They can be used to enforce mutual exclusion by enclosing critical regions

```
    T1
P(s);
  critical region
V(s);
```

```
   T2
P(s);
  critical region
V(s);
```

Example

- Java used not to support semaphores directly
  - Although they could be simulated
- In previous code, each thread would perform

```
P(s);
   thing.inc(); // critical region
V(s);
```

- The Java 5 API provides a counting semaphore

Semaphores

- A semaphore that can only take values 0 or 1 is a binary semaphore
  - unrestricted ones are counting semaphores
- When a process/task/thread is in its critical region, no others can enter theirs
  - hence, keep critical regions as small as possible
- Use of semaphores requires care

Question

- The value of a semaphore s is initially 1. What could happen in the following situation?

```
   T1                          T2
V(s);                       P(s);
  critical region             critical region
P(s);                       V(s);
```

a) Deadlock will ensue
b) T1 and T2 can both enter their critical regions simultaneously
c) Neither T1 nor T2 can enter its critical region
d) T1 can never enter its critical region, but T2 can enter its own
e) T1 can enter its critical region, but T2 can never enter its own

Answer: b
If T1 executes first, then it acquires the semaphore, which is immediately released by T2. Both then execute the critical region.
If T2 executes first, it releases a semaphore it does not have, which can be acquired by T1. Again, both can execute the critical region.

Classic Synchronisation Problems

- There are a number of famous problems that characterise the general issue of concurrency control
- These problems are used to test synchronisation schemes
- We will look at two such problems that involve synchronisation issues:
  - The Readers-Writers Problem
  - The Producer-Consumer Problem

The Readers-Writers Problem

- A problem in which several threads/processes have shared access to a file or database
- Readers: threads that only read the database
- Writers: threads that both read the database and update it (write)
- Two readers accessing the shared data simultaneously poses no problem
- But, if a writer and another thread (either a reader or a writer) access the shared data simultaneously, problems can arise

Example

- A real-world example of the readers-writers problem is an airline reservation system
- Readers: want to read flight information
- Writers: want to make flight reservations
- Potential problem: if readers and writers can access the shared data simultaneously then readers/writers may view flights as being available when they've actually just been booked
- Solution: enforce mutual exclusion, whilst ensuring the system is fair (avoids starvation)

Solution – First Attempts

- In the original statement of this problem, there were two solutions that both made use of semaphores
- Solution 1: give priority to readers over writers - readers are kept waiting only if a writer is modifying the data
  - Problem: results in starvation of writers if there is a continuous stream of readers
- Solution 2: give priority to writers over readers - as soon as a writer arrives, any readers that are reading are allowed to finish their task, but all additional readers are put on hold until the writer has finished its task
  - Problem: results in starvation of readers if there is a continuous stream of writers

Alternative Solution

- As neither of the previous two solutions are acceptable, another solution was later proposed that encompasses a combination priority policy and avoids starvation problems

- Solution:
  - when a writer has finished writing, any and all readers who are waiting are allowed to read
  - when this group of readers have finished reading, a writer on hold can begin to write
  - any new readers that arrive in the meantime are not allowed to start reading until the writer has finished