Lecture 7: Synchronisation

Adam Hawley

February 5, 2019

Contents

1 Coordination

The OS must support multiple applications/processes (threads working together). This is formally known as **Inter-Process Communication** (IPC). It is achieved through two main concepts: shared memory and message passing.

Shared memory requires synchronisation to ensure data is consistent. For example, if a data structure is being updated then a reader will read a consistent view of that data (i.e. doesn't read partial updates).

Some programming lanuages make synchronisation primitives available to programmers. Languages rely on atomic operations when accessing shared data. However, atomic operations at the language level are often implemented at multiple instruction at the CPU level and interrupts can occur between instructions.

1.1 Concurrent Access to Shared Data

Race conditions are when many processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last. This can mean that data may become inconsistent. Therefore, the ability to execute an instruction, or a number of instructions atomically is crucial for the correct operation of synchronisation primitives provided by programming languages.

1.2 Critical Section Problem

Every process which could have a race condition has a **critical section**. To overcome the problem, one general approach is to only allow one process into the critical section at a time.

One general protocol for solving the problem is outlined as follows:

- Each process must ask permission to enter the critical section in what is known as **entry section** code.
- It then executes in the critical section.
- Once it finishes executing in the critical section it enters the **exit section** code.
- The process then enters the **remainder section** code.

```
do {
    entrySection
        criticalSection
    exitSection
        remainderSection
};
```

1.3 Solutions to the Critical Section Problem

1.3.1 Requirements to Fulfill

Solutions to the critical section problem must satisfy the following three requirements:

Mutual Exclusion If process $\mathbf{P_i}$ is executing in its critical section, then no other processes can be executing in their critical sectinos.

Progress Processes waiting to enter a critical section cannot block processes inside; only those processes that are not executing in their remainder sections can participate in making the decision as to which process will enter its critical section next.

Bounded Waiting A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

1.3.2 Disabling Interrupts

One way to satisfy these requirements is to disable interrupts when a process is in the entry section and reenable them when it is in the exit secton. However this is very impractical if the critical section code takes a long time to execute as it becomes impossible to preempt. This means that any part of a preemptive scheduling algorithm becomes redundant.

1.3.3 Peterson's Algorithm

```
int turn; // indicates whose turn it is to enter the critical section if both are read
bool flag[2]; // indicates if a process is ready to enter the critical section
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        criticalSection
    flag[i] = false;
        remainderSection
};
```

This algorithm assumes read and store operations are atomic (in checking the conditions of the while loop). It also loops while waiting to enter the critical section (also called **spin lock**) - busy waiting, wastes CPU. This implementation works for two processes only, even though it is extendable.

1.3.4 Mutual Exclusion - Hardware Support

Modern CPUs provide special instructions to support exclusive access to shared variables. e.g test-and-set instruction:

```
bool test_and_set (bool *target)
    {
        bool rv = *target;
        *target = TRUE;
        return rv;
    }
```

With this we can use the instruction to enforce mutual exclusion. We have a shared boolean variable `lock`, initialised to FALSE. Then each process wishing to execute critical section code does the following:

```
do {
    while (test_and_set(&lock)); //do nothing
        criticalSection;
    lock = false;
        remainderSection;
}
```

The process keeps looping unless `lock == false`. This still uses a spin lock which is still wasteful. Actual implementations put the process in a waiting queue.

2 Semaphores

A semaphore is a programming mechanism used to achieve synchronisation and mutual exclusion. They are based on mutual exclusion services provided by the OS. They ensure atomic execution and rely on hardware support (e.g the `test_and_set` instruction). Semaphores are accessed through system calls. They are integer variables used as a *flag* and the atomic code that increments or decrements it. There are two types of semaphore:

Binary Semaphores (mutex) Integer values can range only between 0 and 1.

Counting Semaphores:: Integer value can range over an unrestriceted domain.

2.1 Semaphore Operations

Semaphores have two indivisible operations: **wait** (**P**) and **signal** (**V**).

2.1.1 `wait`

1. Decrement semaphore
2. If semaphore value is (or became) negative, block the process and add it to a waiting queue.

2.1.2 `signal`

1. Increment semaphore
2. If semaphore is less than or equal to zero, process waiting at the head of the queue is awakened.
3. If semaphore is greater than zero, it means that no process is waiting, no immediate action needed.