IDATT2202 OBL2-OS

Nicolai H. Brand

October 2022

1 Synchronisation

1.1

a) One process may want to use the output generated from another process as its input.

b) In Unix-like operating systems this is done through a "pipe" which has a read end and a write end. The process generating output would attach to the write end of the pipe, while the process using that output would attach to the read end of that pipe.

c) In the aforementioned example, the second process would only start once the write end of the pipe is closed. If one is dealing with a continuous flow of data the write end of pipe would never close, and the reading process would not make any progress.

A more general issue with inter-process communication is the problem of reading outdated data. For example, one program might write some data to a file. Another process might read that data and later process it. While the second process is processing the data, the first process might update the data in the file which would make the data being processed outdated.

1.2

The critical region is the section of a program that accesses shared data. In this section, a thread will usually hold an exclusive lock over the shared data it is accessing. Most implementations temporarily disable interrupts while executing inside the critical section.

1.3

While polling, a process/thread will constantly check whether it is eligible to enter the critical section. This is often an unnecessary waste of resources. A better option in some circumstances is blocking using wait and signals. The process/thread will wait until it receives a signal that it is eligible to enter the critical section.

1.4

A race condition is when the behaviour of a program depends on the order of which threads are executed in. Since the order of execution for threads appear "random", one cannot make any assumptions on which thread will execute before another. An example would be if two threads depend on the same shared data structure, but that data structure could be updated by the other. For example, say thread A wants to calculate y = 1 + x, while thread B wants to calculate x = x + 1. The result of y in thread A is dependent on whether or not thread B was executed or not.

1.5

A spinlock is similar to polling. It attempts to acquire a lock by constantly checking (spinning) the condition variable in a loop. Spinlocks are used when it is expected that waiting time to acquire the lock is short. It is used in the Linux kernel.

1.6

- Lock contention
- Shared data may be in the wrong cache. This will lead to shared data being pinged back and forth between the local caches.
- False sharing.

MCS attempts to solve the problem of lock contention. It is optimised for the case where you have many waiting threads for a lock. It assigns each waiting thread separate memory where it can spin.

RCU is optimized for the case where you have many waiting threads, but most of this only want to read the shared data, and not write to it. It depends on atomic read-modify-write instructions.

2

2.1

Starvation is when a process is denied the resources it needs to progress. A deadlock is when two or more threads are waiting for each other, and no progress can be made. For example, thread A may wait for thread B to progress, but thread B is waiting for thread A to progress which leads to neither making any progress. A deadlock is therefore a form of starvation.

2.2

- Limited access to resources
- No preemption
- Multiple independent requests
- Circular chain of requests

Limited access to resources is an inherent property in an operating system as the resources will be limited by hardware. Multiple independent requests is also an inherent property of any operating system as multiple concurrent programs are running at the same time.

2.3

The operating system keeps track of resource allocations given to threads and processes. The OS can then detect when multiple threads enter what is called a ”resource cycle”. It can also detect when a process allocates an extreme amount of resouces. In either of these cases the OS can kill the bad behaving processes.

3

3.1

a) FIFO is optimal when all tasks need the processor. This is because FIFO introduces no overhead with preemption.

b) MFQ uses multiple layers of FIFO queues with different precedence. Once a process is out of the FIFO queue, it executes until its finished or until it has used the quantum time (the max time allotted for a process before preemption). If a process was preempted before it was finished, it will be move down a layer of precedence. Once a process is at the base level queue it circulates in round robin fashion until it completes.

On a multi-core system, MFQ may have issues with lock contention and pinging.

3.2

a)
- Cache coherence: Since each processor has a local cache, one processor may need to fetch new data as the current local cache has been invalidated my a processor running another task.
- Limitied cache reuse: A consequence of poor cache coherence. The data in the local caches in each processor is likely invalidated before it can be used.

- Lock contention: As mentioned earlier, MFQ may have issues with lock contention for example when tasks require waiting for I/O.

b) Work-stealing is when a one processor or processor core ”steals” a thread from the queue of a another processor or processor core.