OBL3-OS

August 12, 2020

This is a mandatory assignment. Use resources from the course to answer the following questions. **Take care to follow the numbering structure of the assignment in your submission**. Some questions may require a little bit of web searching. Some questions require you to have access to a Linux machine, for example running natively or virtually on your own PC, or by connecting to `gremlin.stud.iie.ntnu.no` over SSH (Secure Shell). Working in groups is **permitted**, but submissions must be **individual**.

1 Synchronisation

1. The principle of process isolation in an operating system means that processes must not have access to the address spaces of other processes or the kernel. However, processes also need to communicate.

   (a) Give an example of such communication.

   Solution: We can, for example, divide an application into several modular parts, each in charge of handling its own tasks. With modular programs, each part can be used in a more flexible way or shared with/used by other programs (e.g., consider a web server application that contains code to do database operations. This application can be divided into a web server and a database server which communicate through network sockets).

   (b) How does this communication work?

   Solution: Pipes, network sockets, UNIX sockets, message passing, signals, are some ways we can achieve interprocess communication, shared memory. Most of these are available through system calls, so the kernel can be used for synchronisation.

   (c) What problems can result from inter-process communication?

   Solution: If we use shared memory without kernel involvement for synchronisation, we can have problems like race conditions while accessing the shared memory with multiple processes simultaneously. If we're communicating via a bounded buffer (FIFO), then we could read from an empty buffer or write to a full buffer. This should be properly handled via synchronisation (through the kernel) with use of shared memory.

2. What is a critical region? Can a process be interrupted while in a critical region? Explain.

   Solution: A critical region is a section of executable code that only one process or thread has access to at any given time, and is protected by a lock. A critical region normally protects some shared variable in memory, so it should only be occuppied for as short a time as possible.

3. Explain the difference between busy waiting (polling) versus blocking (wait/signal) in the context of a process trying to get access to a critical section.

> **Solution:** Busy waiting or polling means that the process or thread checks the value of a lock variable constantly, and in doing so uses CPU cycles. Blocking with wait and signal is a more efficient way of waiting for a lock to become free, because the process or thread is sent to the CPU queue until it receives a signal that the lock is free. When the signal is received, the scheduler can give the process or thread the CPU again, and it can enter the lock and continue into the critical region.

4. What is a race condition? Give a real-world example.

> **Solution:** A race condition is the unintended result of two or more processes or threads accessing shared memory where the result depends on the order of accesses. The milk example of the textbook is an example of a race condition where one flatmate is currently buying milk, and the other comes home to see that there is no milk, so goes to the store to buy milk at the same time, resulting in more milk than necessary. Race conditions can be solved with properly implemented synchronisation.

5. What is a spin-lock, and why and where is it used?

> **Solution:** A spinlock is a lock that is used in cases where the waiting time for a lock is assumed to be short. These types of locks are mainly used in the kernel to control access to data shared between different kernel threads.

6. List the issues involved with thread synchronisation in multi-core architectures. Two lock algorithms are MCS and RCU (read-copy-update). Describe the problems they attempt to address. What hardware mechanism lies at the heart of each?

> **Solution:**
>
> - If many threads are trying to enter a critical section, there will be high lock contention
> - The lock, a variable in shared memory, will tend to ping between the cores running the threads that are requesting the lock, causing slowdown
> - MCS is a type of spinlock optimised for when there are many waiting threads. RCU is optimised for when there are many readers, and writes happen occasionally. Overhead for readers is decreased at the cost of overhead for writers. Both MCS and RCU rely on atomic read-modify-write instructions such as *compare-and-swap* or *test-and-set*.

2 Deadlocks

1. What is the difference between resource starvation and a deadlock?

Solution: Both deadlock and starvation are liveness concerns. Starvation: a thread fails to make progress for an indefinite period of time. Deadlock: a group of threads are blocked in a resource cycle which cannot progress. A deadlock implies starvation, but starvation does not imply a deadlock situation.

2. What are the four necessary conditions for a deadlock? Which of these are inherent properties of an operating system?

Solution:

- Limited resources
- Non-preemptive (kernel cannot take resources away from a process/thread)
- Hold and wait (gain a resource and hold it while asking for another resource)
- Circular waiting (each thread involved in the deadlock is waiting on the resource of another)

3. How does an operating system detect a deadlock state? What information does it have available to make this assessment?

Solution: The OS knows which processes/threads have own which resources, and it knows which resources are being requested (via system calls). It also holds thread state information, so it can determine if there is a resource cycle in any given set of threads and use this to detect a deadlock (the other three requirements for a deadlock are already met, generally speaking, for any modern OS).

3 Scheduling

1. Uniprocessor scheduling

   (a) When is first-in-first-out (FIFO) scheduling optimal in terms of average response time? Why?

   Solution: FIFO is optimal when tasks are equal in size. Overhead is minimised because tasks run to completion once started, so there is only context switching when a task finishes.

   (b) Describe how Multilevel feedback queues (MFQ) combines first-in-first-out, shortest job first, and round robin scheduling in an attempt at a fair and efficient scheduler. What (if any) are its shortcomings?

   Solution:

   - New tasks are given high priority, and if they are shorter than the CPU time quantum they will finish (shortest-job-first). Time time quantum and priority queues comes from round robin, each priority queue treats each task fairly in turn. Minimising scheduling overhead by reducing preemptions as longer tasks move down to lower priorities with longer time quantums (first-in-first-out).

- MFQ is quite good, but if we consider a multiprocessor, there are issues with the memory hierarchy and lock contention (see next question).

2. Multi-core scheduling

   (a) Similar to thread synchronisation, a uniprocessor scheduler running on a multi-core system can be very inefficient. Explain why (there are three main reasons). Use MFQ as an example.

   Solution:

   (b) MFQ will be subject to lock contention depending on how much work each task does before waiting for I/O.

   (c) Cache coherence overhead - each processor will have to fetch the state of the MFQ, and possibly modify it, so that the other cores will have to evict the old data from their caches.

   (d) Limited cache reuse by threads that have switched processor cores.

   (e) Explain the concept of work-stealing.

   Solution: Cores that have empty task queues can "steal" tasks from other cores that have too many. This is a way of balancing workload across multiple cores.