Assignment 3

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.**
    Shared memory
    b. **How does this communication work?**
    Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other.
    c. **What problems can result from inter-process communication?**
    For example, starvation, deadlocks, data inconsistency and shared buffer problem
2. **What is a critical region? Can a process be interrupted while in a critical region? Explain.**
    A critical region is a sequence of code that atomically accesses shared state. If some other process with a higher priority in a pre-emptive system doesn't need to run in critical section, i.e. doesn't need to acquire a lock which is held by a lower priority process, then it can pre-empt the lower priority process regardless of what it is executing. The lower priority process would then be scheduled for a new complete run. The process in the critical region cannot be migrated to another process while in a critical region.
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.**
    When a thread busy waits for access to a critical section, it spins in a loop until the critical section is available. This is bad if it takes long time because the thread continues to run until the critical section is made available. If the thread uses blocking instead, the thread can be paused, allowing other threads to do its work in the meantime.
4. **What is a race condition? Give a real-world example**
    A race condition is when the behaviour of a program relies on the interleaving of operations of different threads. For example, in a smart house with smart lightning, the state of the light with two connected switches depends on which of the switches was changed last.
5. **What is a spin-lock, and why and where is it used?**
    A spinlock is a lock where a thread waiting for a BUSY lock “spins” in a tight loop until some other thread makes it FREE. On a multi-core system, it would take much more time to putting threads to sleep and wake them up again, compared to let them “spin” waiting for a lock to be FREE.
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?**
    1. *Locking*. A lock implies mutual exclusion — only one thread at a time can hold the lock. As a result, access to a shared object can limit parallelism.
    2. *Communication of shared data*. Shared data protected by a lock will often need to be copied from one cache to another. Shared data is often in the cache of the processor that last held the lock, and it is needed in the cache of the processor

that is next to acquire the lock. Moving data can slow critical section performance significantly compared to a uniprocessor.

3. *False sharing*. A further complication is that the hardware keeps track of shared data at a fixed granularity, often in units of a cache entry of 32 or 64 bytes. This reduces hardware management overhead, but it can cause performance problems if multiple data structures with different sharing behaviour fit in the same cache entry.

MCS is an implementation of a spinlock optimized for the case when there are a significant number of waiting threads. *RCU* is an implementation of a reader/writer lock, optimized for the case when there are many more readers than writers. Both *MSC* and *RCU* uses threads as the underlying hardware mechanism.

2 Deadlocks

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

   Starvation occurs when multiple processes or threads competes for access to a shared resource and one process monopolise the resource so the others are denied access. Deadlocks can occur when two processes need multiple shared resources at the same time in order to continue. For example: thread A is waiting to receive data from thread B. Thread B is waiting to receive data from thread A. The two threads are in deadlock because they are both waiting for the other and not continuing to execute.

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

   1. *Bounded resources*. There are a finite number of threads that can simultaneously use a resource.
   2. *No pre-emption*. Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it.
   3. *Wait while holding*. A thread holds one resource while waiting for another. This condition is sometimes called multiple independent requests because it occurs when a thread first acquires one resource and then tries to acquire another.
   4. *Circular waiting*. There is a set of waiting threads such that each thread is waiting for a resource held by another.

   As devices do have a finite amount of resources, bounded resources are an inherent property of an operating system. The others can be handled by the operating system to prevent deadlocks.

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

   If there are several resources and only one thread can hold each resource at a time, then we can detect a deadlock by analysing a simple graph. In the graph, each thread and each resource are represented by a node. There is a deadlock if and only if there is a cycle in the graph.
   If there are multiple instances of some resources, then we represent a resource with *k* interchangeable instances as a node with *k* connection points. Now, a cycle is a necessary but not sufficient condition for deadlock.
   Another solution is a variation of Dijkstra's Banker's Algorithm. We can look at the current set of resources, granted requests and pending requests and ask whether it is

possible for the current set of requests to eventually be satisfied assuming no more requests come and all threads eventually complete. If so, there is no deadlock; otherwise, there is a deadlock.

3 Scheduling

1. **Uniprocessor scheduling**
    a. **When is first-in-first-out (FIFO) scheduling optimal in terms of average response time? Why?**
    When the tasks are equal in size, because switching tasks would just make all the tasks return later than they would if the got to run one-by-one.
    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?**
    MFQ has multiple Robin Round queues with different priorities and time slices. The higher the priority, the lower time slice. If the tasks use up its time slice, it's moved down a priority level. Additionally, it monitors every task to ensure that it's receiving its fair share of the resources. Based on the monitoring the tasks can be moved up or down in priority. One shortcoming can be that tasks in the lower priority queue can suffer from starvation if its priority isn't boosted regularly.
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.**
        - *Contention for the MFQ lock*. Depending on how much computation each thread does before blocking on I/O, the centralized lock may become a bottleneck, particularly as the number of processors increases.
        - *Cache Coherence Overhead*. Although only a modest number of instructions are needed for each visit to the MFQ, each processor will need to fetch the current state of the MFQ from the cache of the previous processor to hold the lock. On a single processor, the scheduling data structure is likely to be already loaded into the cache. On a multiprocessor, the data structure will be accessed and modified by different processors in turn, so the most recent version of the data is likely to be cached only by the processor that made the most recent update. Fetching data from a remote cache can take two to three orders of magnitude longer than accessing locally cached data. Since the cache miss delay occurs while holding the MFQ lock, the MFQ lock is held for longer periods and so can become even more of a bottleneck.
        - *Limited Cache Reuse*. If threads run on the first available processor, they are likely to be assigned to a different processor each time they are scheduled. This means that any data needed by the thread is unlikely to be cached on that processor. Of course, some of the thread's data will have been displaced from the cache during the time it was blocked, but on-chip caches are so large today that much of the thread's data will remain cached. Worse, the most recent version of the thread's data is likely to be in a remote cache, requiring even more of a slowdown as the remote data is fetched into the local cache.

b. **Explain the concept of work-stealing.**
In a work stealing scheduler, each processor in a computer system has a queue of work (tasks) to perform. Each task consists of a series of instructions, to be executed sequentially, but in the course of its execution, a task may also spawn new tasks that can be executed in parallel with its other work. These new tasks are initially put on the queue of the processor executing the task. When a processor runs out of work, it looks at the queues of other processors and "steals" their tasks. In effect, work stealing distributes the scheduling work over idle processors, and as long as all processors have work to do, no scheduling overhead occurs.