Operating systems task 3

Ingebrigt Hovind

synchronization, deadlocks & scheduling

Synchronisation

1.

   A. Such communication may be in order to share application specific logic. For example, one thread in a word processor might be trying to close the entire application, while another thread may be trying to autosave the changes made. The application does not want to quit without finishing the save, so the application might implement a shared object which tells the thread whether or not the application is ready to be closed.

   B. This communication can take place through the use of shared objects. This may be through using a synchronization variable in order to share state between two threads.

   C. The problems that can result from inter-process communication are dependent on the checks on the threads implemented by the programmer. If these checks are implemented poorly they may lead to a desired action never taking place, or that the desired action takes place too many times (lack of liveness or lack of safety).

2. A critical region is an area in code or a variable that is shared between threads in a multi-threaded program. A process can be interrupted while in a critical section, but this should be avoided in order to protect the locks data structures

3.

   Blocking is done through condition variables. This is more efficient than polling. A condition variable has three methods; wait() which releases the suspends the thread that called it and releases the lock. When the thread returns to execution the lock is re-acquired and execution of the thread proceeds. Signal() is called by the thread which has the lock before it releases the lock. The lock is given to the first thread in the queue.

   Broadcast() wakes up every thread that is waiting on the condition variable and the thread that first manages to acquire the lock resumes execution.

   Polling is to repeatedly check the shared state of the lock to see if it has changed. this approach is inefficient: the waiting thread continually loops, or busy-waits, consuming processor cycles without making useful progress. Worse, busy-waiting can delay the scheduling of the thread that has the lock, increasing the time that it has to hold the lock. But for short waits it is more efficient than de-scheduling the thread as you have to do with a condition variable.

4. A race condition is when a program has threads whose interleavings affect each other and which therefore has an effect on the execution of the program.

   A real world example may be roommates taking care of a cat together. If the communication between them is not sufficient they may end up feeding the cat twice as often, or perhaps not at all. Here the behaviour of one roommate (a thread) depends on whether or not the other has fed the cat (the execution of the other thread), making this a race condition.

5. A spin lock is when a thread waiting to acquire a lock is stuck in a loop, doing nothing but continuously checking whether or not the lock can be acquired yet. Spin locks are used to ensure that no thread can enter a critical section while another thread has acquired a lock and if the average wait time is low, otherwise the thread will call wait().

6. The problem with multi-threaded programs on multi-cored architecture is that operations by a thread on one processor are interleaved with operations by other threads on other processors.

   Every entry in a processor cache has a state, either exclusive or read-only. If any other processors have a cached copy of the data, it must be read-only everywhere. To modify a shared memory location, the processor must have an exclusive copy of the data; no other cache is allowed to have a copy. Otherwise, one processor could read an out-of-date value for some location that another processor has already updated. To read or write a location that is stored exclusive in some other cache, the processor needs to fetch the latest value from that cache.

   MCS locks are an implementation of the lock abstraction that works best when there are a significant number of waiting threads, i.e the lock is the bottleneck. It uses compare and swap, which tests the value of a memory location and swaps in a new value if the old value has not changed. This allows the lock to be passed more efficiently from one thread to another.

   RCU is an implementation of a reader writer lock optimized for the case when there are many more readers than writers. Overhead for readers is reduced at the cost of increased overhead for writers. With many reader of a critical section the synchronization object can become a bottleneck because reading locks can only be acquired by one thread at a time. Multiple concurrent read-only versions of the same critical section to be in progress at the same time as the update. When the critical section is updated, this change is published with a single, atomic memory write. When updates are published there may still be threads reading the old version of the critical section, when all the threads that started before the grace period are finished with reading the critical section, the grace periods ends and the multiple states that were maintained during this period is unified into a single state with the updated values.

Deadlocks

1. A deadlock is when two separate threads have both set locks, and each of them are trying to acquire the other thread’s lock. Neither thread can release their own lock because they cannot progress, so they have to wait indefinitely. Resource starvation is when a thread fails to make progress for an infinite amount of time. This means that a deadlock is a type of resource starvation, but it is not the only type of starvation. For example a writing thread might starve if reading threads are arriving frequently enough to give the writing thread no

chance to acquire a writing lock. This example is not a deadlock, as there is no wait while holding, which is required for deadlocks.

2. The four necessary conditions are:

    1. **Bounded resources**

        No system can have an infinite amount of resources, so this is an inherent property of every single operating system.

    2. **No preemption**

        When a thread acquires a resource, this resource cannot be released until the thread itself chooses to release it. A thread can be forced to give up a lock if a deadlock is detected (e.g. forcing a thread to stop execution), so this is not inherent

    3. **Wait while holding**

        Not inherent in an operating system due to being able to design a program where threads have to atomically acquire both locks when it needs to acquire two locks.

    4. **Circular waiting**

        This is not inherent, as this has to do with the design of an individual program, if the program never requires more than one thread to acquire more than one lock, then deadlocks cannot happen.

3. The operating system can generate a graph with resources and threads being nodes. The thread-nodes are connected by either a thread acquiring a resource or waiting for a resource. If this graph has a loop, there is a deadlock.

Scheduling

1. Uniprocessor scheduling

    A. FIFO is optimal when there is a large amount of tasks, each task is close to the same size and ideally switching between tasks only when one completes. FIFO optimizes this situation because it is the simplest scheduling algorithm and therefore minimizes overhead and allows the most amount of time to be spent on performing the tasks

    B.

    MFQ Combines the low overhead from first-in-first-out and it runs the short tasks quickly like SJF. It avoid resource starvation by letting every task make progress like round robin. A MFQ is a jack of all trades, but master of none, which can be viewed as a weakness in certain situations when the need for one aspect far overshadows the need for the others.

2. Multi-core scheduling

    A. A uniprocessor scheduler running on a multi-core system is inefficient primarily because of:

Contention for the MFQ lock.

If the queue is centralized and has a lock in order to ensure that only one processor can modify the data structure at a time, then the lock can become a bottleneck depending on how much work each processor can do before it needs to acquire the lock. This problem gets worse with more processor if there is still only one lock.

Cache Coherence Overhead.

Every time a processor wants to interact with the MFQ it needs to first acquire the lock and then fetch the state of the MFQ from the cache of the processor that held it last. As the multiple processors have different caches this is much slower than reading from a local cache, which also causes the MFQ-lock to be held for longer.

Limited Cache Reuse.

Another problem due to the processors not sharing a cache. If a thread is scheduled on different processors each time it is scheduled, it will be unable to reuse any data that remains in the cache from the last time it was scheduled.

B. Work stealing is a work-conserving scheduling policy where each tasks on a processor can create new work which is then placed in the given processors work queue, when a processor finishes its own tasks it can “steal” work from other processors’ queues, which leads to the scheduling policy being work-conserving, as the processor should never run out of work as long as there are tasks that need to be done.