COS Questions – Lecture 3

Operating System Concepts (Tenth Edition)

Synchronization

6.1 What is a race condition? Give an example of a situation where one can occur.

A race condition occurs when two or more processes/threads simultaneously attempt to access shared memory, such as a global variable, resulting in an undefined change. An example could be two threads attempting to increment the same global counter.

6.2 What is a critical section?

A critical section is a section in code in which a process may **singly** be accessing or manipulating data that is shared with the other processes, meaning no other process is allowed to execute in its critical section simultaneously.

6.3 What is atomic access? What are atomic operations and atomic variables?

Generally, atomic can be understood as "one at a time", meaning that atomic access only allows for a single process to perform an operation on a shared variable – namely an atomic operation. An atomic variable is then a variable for which the order of concurrent access is well defined.

6.4 What is the difference between a binary semaphore and counting semaphore?

A binary semaphore can only have a value of 0 or 1, whereas a counting semaphore is essentially an integer that can be incremented and decremented.

6.5 What is the difference between a mutex and semaphore?

Mutex and Semaphore both provide synchronization. A Mutex is different than a semaphore as it is a locking mechanism while a semaphore is a signaling mechanism. A binary semaphore can be used as a Mutex but a Mutex can never be used as a semaphore.

A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that called the wait function (source).

6.6 What is a conditional variable? How can it be used together with a mutex?

Condition variables allow us to synchronize threads via notifications. For example, given a producer and consumer thread; if the consumer wants to know whether there is data available, it must continuously acquire the mutex, check the queue, and release the mutex – or process any data if it's there – if not, then repeat with some sleep in between the checks.

This is highly inefficient. Instead, a conditional variable can be used, where the producer thread can simply notify the consumer thread, when the data is ready. For example (source):

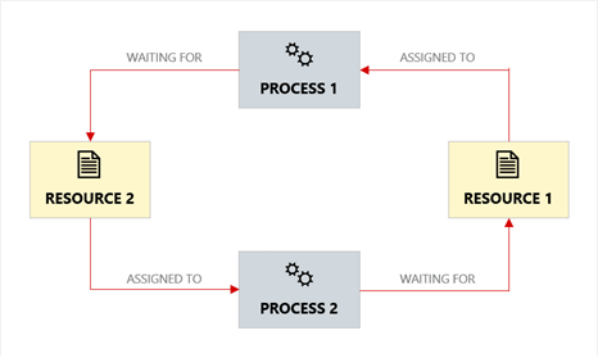
```
condition_variable cond;
mutex mtx;
bool data_ready = false;

void producer() {
    unique_lock<mutex> lock(mtx); // protect data_ready
    data_ready = true;
    cond.notify_one();
}

void consumer() {
    unique_lock<mutex> lock(mtx); // protect data_ready
    while(!data_ready) {
        cond.wait(lock);
    }
    // process data ..
}
```

Here, the `wait()` on the condition variable puts the thread to sleep and the mutex is released. When a thread is awoken by a `notify_one()`, it must re-acquire the lock before the function `wait()` returns to the user code.

In practice, one should be vary of spurious wakeup, where the consumer thread might awaken and continue without a lock – this can be remedied by adding a predicate to the `wait()` method (source).

6.7 What is a deadlock?

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.



6.8 Write a C++ program that creates two threads, each of which atomically increment the same (global) variable every 100 ms. Print the variable each time it is incremented. How might the print log look if no synchronization is used?

Solution posted on It'sLearning (`atomic-increment.zip`).