**Synchronization**

Synchronization is ensuring **cooperation amongst threads**. We don’t want multiple processes doing the same tasks. We also want to make sure we don’t have issues with threads interfering with each other. At the simplest level, synchronization is providing waits at the appropriate time.

**Critical Section Problem**

One solution we have to this issue is by implementing **critical sections**. In one process, only one thread is allowed to work in a critical section at a time. Meaning, we make a chunk of code special and say when it executes, no other thread can execute its special code. This prevents memory from being overwritten at the wrong time. This also prevents the duplication of execution.

In order to solve this problem, we must satisfy the following: **Mutual Exclusion** (if one process is in its critical section, no other processes can be in theirs), **Progress** (if multiple processes are ready to go to their critical section, only the ones with nothing else to do can go first), **Bounded Waiting** (there is a limit on how long a process must wait before its their turn to go into the critical section. Once they are waiting, there is a limit on how long they have to wait).

There are two ways for kernels to handle critical sections: **Preemptive Kernels** and **nonpreemptive kernels**. Preemptive kernels allow for the kernel to interrupt a process even when it is in a critical section. It may swap that process for another one, so there still is only one process executing its critical section but we’re chaining which one is. Non-preemptive kernels do not allow the process in its critical section to be interrupted. This ensures that the process finishes its critical section before the next process can enter its.

While nonpreemptive kernels are simpler, they allow a process to run for an arbitrary amount of time which can be too long. Preemptive kernels solve that problem by limiting how long a process can run its critical section but are much more complicated. Preemptive is more common.

**Peterson’s Solution**

Peterson’s solution is a software based solution to the critical section problem. In this case, we have an int that tells which process’s turn it is to enter their critical section. Along with this, there is also an array of booleans that tell whether each process is ready to enter its critical section. A process must have it be their turn as well as be ready to enter the critical section for them to being executing the critical section.

This is not guaranteed to work on a modern system.

**Locks**

One solution to synchronization is a lock. A lock is one thread saying “I’m doing this, no one else is allowed to.”

We always get a lock before accessing shared data. If we always make sure that we get the lock first, we will never have two threads accessing the shared memory at the same time. We also need to make sure we release the lock after we’re done so the next thread can go ahead.

We really want to avoid wasting CPU cycles by having threads waiting on a lock. We will have to schedule our locks too so they have their own wait queue.

Acquiring and releasing the lock are atomic instructions meaning they cannot be interrupted. This adds some safety to the lock process.