Chapter 6: Synchronization

Introduction:

A **cooperating process** is one that can affect or be affected by other processes executing in the system. These processes can be either directly share a logical address space or be allowed to share data only through shared memory or message passing.

Concurrent access to shared data may result in data inconsistency and there are various mechanisms to ensure the orderly execution of cooperating processes, so that data consistency is maintained.

6.1 | Background:

What we already know: Processes can execute concurrently or in parallel. A CPU scheduler rapidly switches between processes to provide concurrent execution.

Concurrent: A process can be interrupted at any point in its' instruction stream and a processing core may be assigned to execute instructions of another process.

Parallel: Two instruction streams execute simultaneously on separate processing cores.

This chapter will explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

We would arrive at an incorrect state because we allowed both processes to manipulate the variable count concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**

How do we protect against the race condition? We need to ensure that only one process at a time can be manipulating the variables. We need the processes to be synchronized in some way.

6.2 | The Critical-Section Problem:

The critical-section problem is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data.

Critical section: Segment of code in which the process may access and update the data that is shared with another process.

Each process must request permission to enter its critical section, a **entry section** followed by an **exit section**, the rest of the code is in the **remainder section**.

The solution must satisfy the following three requirements:

1. Mutual Exclusion: If one process is executing in its critical section then no other process can execute in their critical sections.
2. Progress: If no process is executing in its critical section, and a process wants to start entering their critical section. Only processes not executing in their remainder sections can participate in deciding which will enter its critical section next.

3. Bounded Waiting: There exists a limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

What kind of hiccups are we talking about if we don't meet the requirements?

If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes if the processes are creating child processes using fork().

What is the simplest solution?

Run in a single-core environment that could prevent interrupts while a shared variable was modified. Sequence of instructions would be maintained without preemption. No other instruction would run.. bada-bing bada-boom, no unexpected modifications to the shared variable.

Why is the simplest solution unrealistic?

Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors.

Two general approaches are used to handle critical sections:

A. **Preemptive Kernels:** Allows a process to be preempted while it is running in kernel mode. Race condition might still exist, why? Because it is possible for two kernel-mode processes to run simultaneously on different CPU cores.
B. **Non-preemptive Kernels:** Does not allow a process running in kernel mode to be preempted. This process will run till it exits kernel mode, blocks, or voluntarily yields control of CPU. This one is free from race conditions, duh.

Why even bother considering preemptive kernels? Because a preemptive kernel is more responsive, there is less risk that a process will run for a ridiculously long time and it is better for real-time processes.

6.3 | Peterson's Solution:

Now, a classic software-based solution to the critical-section problem. This solution takes on addressing the requirements of mutual exclusion, progress, and bounded waiting.

The basic idea is two processes that alternate execution between their critical sections and remainder sections.

1. Mutual exclusion is preserved. How? P_i enters its critical section only if flag[j] == false or turn ==i. If both processes can execute their critical sections at the same time, then flag[0] == flag[1] == true, and the value of true can be either 0 or 1 but cannot be both.
2. The progress requirement is satisfied. How? P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn ==j. However, once P_j exits its critical section, it will reset flag[j] to false, allowing P_i to enter its critical section.

3. The bounded-waiting requirement is met. Since P_i does not change the value of the variable turn while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

Can you explain the purpose of the bounded-waiting requirement and how the Peterson's solution tackles it?

Why is the Peterson's solution not guaranteed to work on modern computer architectures?

To improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies. For a multithreaded application with shared data, the reordering of instructions may render inconsistent or unexpected results. It is possible that both threads may be active in their critical sections at the same time,