CHAPTER 6

Synchronization Tools

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

CHAPTER OBJECTIVES

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

6.1 Background

We've already seen that processes can execute concurrently or in parallel. Section 3.2.2 introduced the role of process scheduling and described how the CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process. Additionally, Section 4.2 introduced parallel execution, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores. In this chapter, we explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

Let's consider an example of how this can happen. In Chapter 3, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer–consumer problem, which is a representative paradigm of many operating system functions. Specifically, in Section 3.5, we described how a bounded buffer could be used to enable processes to share memory.

We now return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at most BUFFER_SIZE − 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable, count, initialized to 0. count is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable count is currently 5 and that the producer and consumer processes concurrently execute the statements "count++" and "count--". Following the execution of these two statements, the value of the variable count may be 4, 5, or 6! The only correct result, though, is count == 5, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `count` may be incorrect as follows. Note that the statement "`count++`" may be implemented in machine language (on a typical machine) as follows:

$$\begin{aligned}&\mathit{register}_1 = \texttt{count}\\&\mathit{register}_1 = \mathit{register}_1 + 1\\&\texttt{count} = \mathit{register}_1\end{aligned}$$

where $\mathit{register}_1$ is one of the local CPU registers. Similarly, the statement "`count--`" is implemented as follows:

$$\begin{aligned}&\mathit{register}_2 = \texttt{count}\\&\mathit{register}_2 = \mathit{register}_2 - 1\\&\texttt{count} = \mathit{register}_2\end{aligned}$$

where again $\mathit{register}_2$ is one of the local CPU registers. Even though $\mathit{register}_1$ and $\mathit{register}_2$ may be the same physical register, remember that the contents of this register will be saved and restored by the interrupt handler (Section 1.2.3).

The concurrent execution of "`count++`" and "`count--`" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:

T_0:	*producer*	execute	$\mathit{register}_1 = \texttt{count}$	$\{\mathit{register}_1 = 5\}$
T_1:	*producer*	execute	$\mathit{register}_1 = \mathit{register}_1 + 1$	$\{\mathit{register}_1 = 6\}$
T_2:	*consumer*	execute	$\mathit{register}_2 = \texttt{count}$	$\{\mathit{register}_2 = 5\}$
T_3:	*consumer*	execute	$\mathit{register}_2 = \mathit{register}_2 - 1$	$\{\mathit{register}_2 = 4\}$
T_4:	*producer*	execute	$\texttt{count} = \mathit{register}_1$	$\{\mathit{count} = 6\}$
T_5:	*consumer*	execute	$\texttt{count} = \mathit{register}_2$	$\{\mathit{count} = 4\}$

Notice that we have arrived at the incorrect state "`count` == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5, we would arrive at the incorrect state "`count` == 6".

We would arrive at this 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**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable `count`. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, as we have emphasized in earlier chapters, the prominence of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads—which are quite possibly sharing data—are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one another. Because of the importance of this issue, we devote a major portion of this chapter to **process synchronization** and **coordination** among cooperating processes.

6.2 The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so-called critical-section problem. Consider a system consisting of n processes $\{P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The ***critical-section problem*** is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Figure 6.1 General structure of a typical process.

3. **Bounded waiting**. There exists a bound, or 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.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (***kernel code***) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Another example is illustrated in Figure 6.2. In this situation, two processes, P_0 and P_1, are creating child processes using the `fork()` system call. Recall from Section 3.3.1 that `fork()` returns the process identifier of the newly created process to the parent process. In this example, there is a race condition on the variable kernel variable `next_available_pid` which represents the value of the next available process identifier. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence

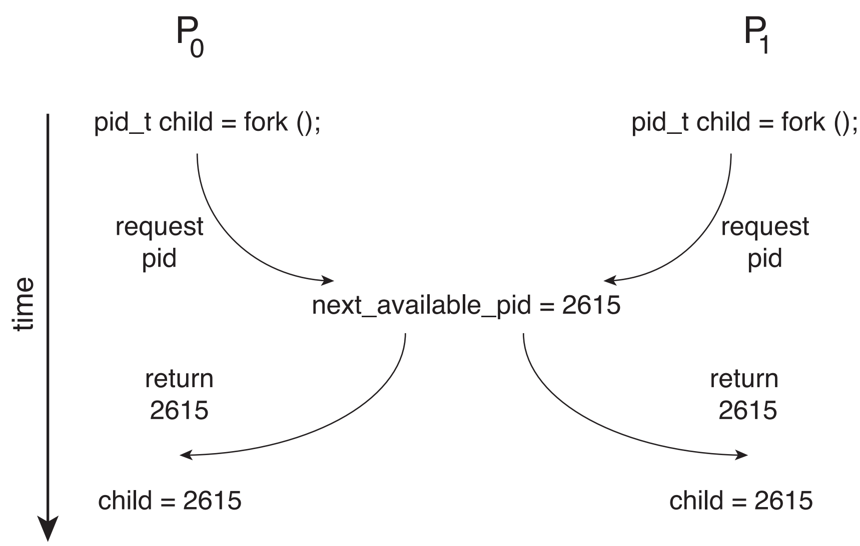

Figure 6.2 Race condition when assigning a pid.