Chapter 6 – Synchronization Tools

As we know, processes can execute concurrently or in parallel. The CPU scheduler switches rapidly between processes to provide concurrent execution. It means that a process may be interrupted at any time, partially completing its execution while another process starts executing its instructions in the CPU.

A **cooperating process** is one that can affect or be affected by **other** processes executing in the system. They can directly share code and data (i.e. a logical address space) or interact with each other by shared memory or message passing. Concurrent accesses to the shared data may result in **data inconsistency**.

In this chapter we discuss various mechanisms for synchronizing the execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Contents

Chapter Objectives

- *Describe the critical-section problem and illustrate a race condition.*
- *Illustrate hardware solutions to the critical-section problem.*
- *Demonstrate how mutex locks, semaphores etc. be used to solve the critical section problem*

6.1 Background

In this section we explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

First, to illustrate the concept of cooperating processes, let's consider the **producer-consumer problem** (also called **bounded buffer problem**), which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler may produce object modules that are consumed by the loader. The **producer-consumer** problem also provides a useful metaphor for the **client-server** paradigm – the server is a producer and the clients are consumers.

One solution to the producer-consumer problem uses **shared memory**. To run concurrently, the producer and consumer processes must share a common **buffer** (usually a queue) that is accessible by both of the processes. A **producer** can repeatedly **generate** data and write it into the **buffer**. The consumer repeatedly **reads/uses** the data in the buffer and removes it afterwards. These activities must be **synchronized** so that the consumer does not try to consume before an item is produced.

There are **two** types of buffers:

- **The unbounded buffer** – There is **no practical limit** on the size of the buffer. The producer can always produce new items, and the consumer consumes them as they are produced. The consumer may have to wait when the buffer is empty (i.e., nothing produced by the producer).
- **The bounded buffer** – The buffer has a **fixed** size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Figure 1 *The buffer*

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Figure 2 *The Producer Process*

```
item next_consumed;

while (true) {
       while (in == out)
              ; /* do nothing */
       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;

       /* consume the item in next consumed */
}
```

Figure 3 *The Consumer Process*

Suppose we want to use an integer **counter** that keeps track of the number of items in the buffer. Initially counter is set to **0**. Each time a new item is added by the producer, the counter is incremented, and each time an item is consumed, the counter is decreased.

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

       while (counter == BUFFER_SIZE)
              ; /* do nothing */
       buffer[in] = next_produced;
       in = (in + 1) % BUFFER_SIZE;
       counter++;
}
```

Figure 4 *The Producer*

```
while (true) {
       while (counter == 0)
              ; /* do nothing */
       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;
       counter--;
       /* consume the item in next consumed */
}
```

Figure 5 *The Consumer*

When executed separately, the code works fine, but they may not function correctly when executed concurrently. Let's assume the **current value** of the **counter** is **5** and the producer and consumer processes concurrently execute the statements **"`counter++`"** and **"`counter--`"**. Following the execution of these two statements, the value of the variable counter can be 4, 5, or 6. We can see how.

The statement **"counter++"** may be implemented in the machine language as follows:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

where **register$_1$** is one of the local CPU registers.

Similarly, the statement **"counter--"** is implemented as follows:

```
Register2 = counter
register2 = register2 - 1
counter = register2
```

again **register$_2$** is one of the local CPU registers.

The concurrent execution of the statements **"counter++"** and **"counter--"** is equivalent to a sequential execution in which the lower-level statements shown above are interleaved in some arbitrary order (preserving the order of high-level statements), such as:

T_0:	producer executes	**register$_1$ = counter**	{register$_1$ = 5}
T_1:	producer executes	**register$_1$ = register$_1$ + 1**	{register$_1$ = 6}
T_2:	consumer executes	**register$_2$ = counter**	{register$_2$ = 5}
T_3:	consumer executes	**register$_2$ = register$_2$ – 1**	{register$_2$ = 4}
T_4:	producer executes	**counter = register$_1$**	{counter = 6 }
T_5:	consumer executes	**counter = register$_2$**	{counter = 4}

We can see that the end result is wrong, where the correct value of the counter should be 5. For some other ordering of the statements may result in counter = 6, which is also wrong.

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**.

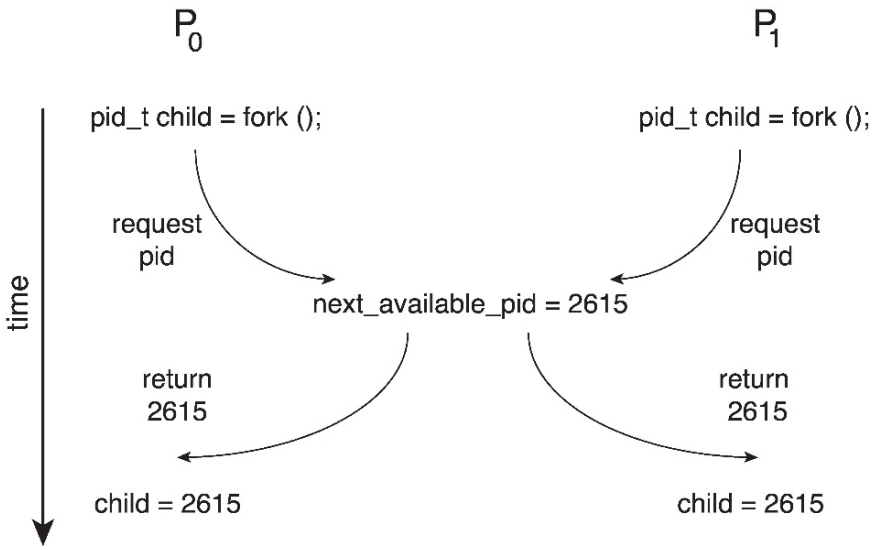

Figure 6 *Race condition when assigning a PID. Unless mutual exclusion**, same PID** can be assigned to different processes.*

To guard against the race condition above, we need to ensure that only **one** process at a time can be manipulating the variable **counter**. So we require that the processes be **synchronized** in some way.

6.2 The Critical-Section Problem

Consider a system consisting of **n** processes {P_0, P_1, P_2... P_n}. 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. When one process is executing its critical section, no other process is allowed to execute its critical section (i.e. no two processes can execute their critical section at the same time). The **critical-section problem** is to design a **protocol**, using which the processes can synchronize their activity and cooperatively share data. Each process must request permission to enter its critical section (this portion of the code is the **entry section**). The critical section is followed by an **exit section** and the rest of the code is **reminder section**.

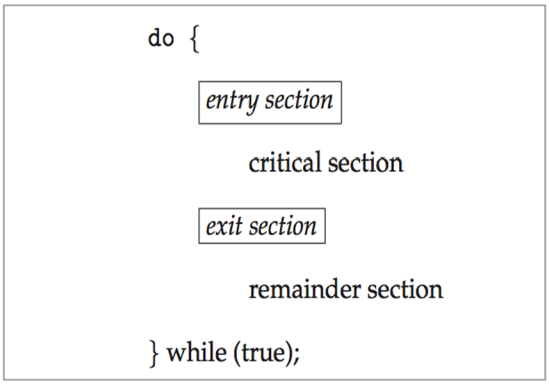

Figure 7 *General structure of a typical process.*

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

- **Mutual Exclusion** – If a process P_i is executing its critical section, then no other processes can be executing their critical sections. It ensures that only **one** process at a time is active in its critical section.
- **Progress** - If no process is currently executing its critical section, but some processes are waiting to enter their critical section, then a process must be selected for executing its critical section next. The selection can only be done by the processes that are not executing in their reminder sections. Progress ensures that programs will cooperatively determine what process will next enter its critical section.
- **Bounded Waiting –** A bound or limit must exist 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. It limits the waiting time of a process before it can enter its critical section.

6.3 Peterson's Solution

It is a classic software based solution to the critical section problem. Although is not guaranteed to work on modern computer architectures it (it assumes the basic machine-language instructions such as **load** and **store** are atomic, i.e., cannot be interrupted). But it provides a good algorithmic description of solving the problem. It also illustrates some complexities involved in designing software that addresses the three requirements of mutual exclusion, progress and bounded waiting.

Peterson's solution is restricted to **two processes** (**P_0** and **P_1**) that alternate execution between their critical sections and reminder sections. It requires the two processes to share two variables:

```
int turn;              //it indicates whose turn it is to enter the critical section.
boolean flag[2]        //it indicates if a process is ready to enter the critical section.
```

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Figure 8 *The structure of process P_i in Peterson's solution.*

`flag[i] = true` implies that process $\mathbf{P_i}$ is ready to execute its critical section.

To enter the critical section, process $\mathbf{P_i}$ first sets **`flag[i] = true`** and then sets **turn = j**, thereby asserting that if the **other** process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, **`turn`** will be set to both **`i`** and **`j`** roughly at the same time. Only one of these assignments will last; the other will occur, but will be overwritten immediately. The eventual value of **`turn`** determines which of the two processes is allowed to enter its critical section first.

By Peterson's solution **mutual exclusion** is preserved – $\mathbf{P_i}$ enters to its critical section only if either **`flag[j] = false`** or **`turn = i.`** From the structure above, we can prove that the **progress** and **bounded-waiting** requirements are also satisfied.

6.4 Hardware Support for Synchronization

The hardware support for the critical-section problem includes memory barriers, some hardware instructions and using atomic variables.

6.5 Mutex Locks

A **mutex lock** provides mutual exclusion to data modification. It has a **binary** value that indicates if the lock is available or not. Mutex lock requires that a process acquires a lock before entering a critical section and releases the lock on exiting the critical section.

6.6 Semaphores

Semaphores, like mutex locks, can be used to provide mutual exclusion and are much more efficient than some other methods of synchronization. A semaphore **S** is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **`wait()`** and **`signal()`**.

The definitions of **`wait()`** and **`signal()`** are as follows –

```
wait(S) {
   while (S<=0); //busy wait
   S--;
}
```

```
signal(S){
   S++;
}
```

The **`wait()`** operation decrements the value of its argument **S**, if it is positive. If **S** is negative or zero, then no operation is performed. The **`signal()`** operation increments the value of its argument **S**.

There are **two** main types of semaphores:

- **Counting Semaphores** – The value of the semaphore can range over an unrestricted value domain. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resources performs a **`wait()`** operation (thereby decrementing the count). When a process releases a resource, it performs a **`signal()`** operation (incrementing the count). When all the resources are used, the count becomes 0. Then processes that wish to use a resource, has to wait until the count > 0.
- **Binary Semaphores** – The value of a binary semaphores can be only 0 and 1, they behave like mutex locks. The **`wait()`**works when the semaphore is 1 and the **`signal()`** operation succeeds when semaphore is 0. On systems that do not provide mutex locks, binary semaphores can be used.

6.7 Monitors

A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.

Solutions to the critical-section problem may suffer from liveness problems, including deadlock. The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can only be performed by one of the waiting processes. This is called a deadlock. There are various mechanisms for dealing with deadlocks.