**Chapter 05: Process Synchronization**

Table of Contents

[Race Condition 4](#_Toc83033021)

[Critical Section Problem 5](#_Toc83033022)

[Peterson’s Solution 7](#_Toc83033023)

[Hardware Solutions 10](#_Toc83033024)

[Test and Set Approach 11](#_Toc83033025)

[Compare and Swap Approach 12](#_Toc83033026)

[Complete Solution with Test and Set 12](#_Toc83033027)

[Mutex Locks 14](#_Toc83033028)

[Semaphore 16](#_Toc83033029)

[Binary Semaphores 17](#_Toc83033030)

[Counting Semaphore 17](#_Toc83033031)

[Semaphore Usage 17](#_Toc83033032)

[Deadlock and Starvation 18](#_Toc83033033)

[Classic Problems of Synchronization 19](#_Toc83033034)

[Bounded-Buffer Problem 19](#_Toc83033035)

[Readers-Writers Problem 20](#_Toc83033036)

[Dining-Philosophers Problem 23](#_Toc83033037)

**Concurrent access** to shared data may result in data **inconsistencies**. Maintaining consistency requires mechanisms that ensure the orderly execution of cooperating processes.

Consider a **consumer-producer** problem. We have a **producer** who adds items to a **buffer** when a new item is produced. It also increments the total **count** of items. If the buffer is full, which can be verified using the variable, the producer will not produce any more items.

while (true)  
{  
 */\* produce an item and put in nextProduced \*/* while (count == BUFFER\_SIZE); *// do nothing* buffer[in] = nextProduced;  
 in = (in + 1) % BUFFER\_SIZE;  
 count++;  
}

C++

A **consumer** on the other hand, removes items from the buffer and decrements the variable. If the buffer becomes empty, it stops trying to consume items.

while(true)  
{  
 while (count == 0); *// do nothing* nextConsumed = buffer[out];  
 out = (out + 1) % BUFFER\_SIZE;  
 count--;  
 */\* consume the item in nextConsumed \*/*}

C++

Our main concern here is going to be with the variable. When a producer produces an item, it is **incremented** and when a consumer consumes an item, it is **decremented**.

## Race Condition

Say we are using two registers, and , to keep track of the variable. The **producer** uses locally while the consumer uses locally. Since these are different processes, they cannot directly be using . They are having to perform read and write operations to their own local registers.

Say, initially, , and we then have the following series of events:

|  |  |  |  |
| --- | --- | --- | --- |
| Event |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Think of this as a production and a consumption occurring at roughly the same moment. The result is that the copy of the variable in each register could not be synchronized. Essentially, the **read** operation for the consumer took place before the **write** operation for the producer, which resulted in the consumer not having the correct value.

A situation where several processes access and manipulate the same data concurrently and the outcome of the executions in dependent on the **order** in which the access takes place is called the **race condition**.

To prevent this issue from occurring, we must ensure that only **one process** can be manipulating data at one moment of time. To do this, we must **synchronize** the processes in some way.

## Critical Section Problem

In a system with processes, each process has a segment of code in which that process is changing **common variables**, such as updating a table, writing to a file, etc. This section of code is called the **critical section**.

To avoid a situation like the **race condition**, we must ensure that no two processes are executing in their critical sections at the same time. We must have a **protocol** to allow the processes to cooperate.

Each process must request **permission** to enter its critical section. The section of code implementing this request is called the **entry section**. Similarly, we may have an **exit section** which occurs after the critical section. The rest of the code is called the **remainder section**.

do  
{  
 */\* entry section \*/  
 /\* critical section \*/  
 /\* exit section \*/  
 /\* remainder section \*/*} while (true);

C++

Addressing this problem requires that we satisfy three requirements:

1. **Mutual Exclusion** – If one process is executing in its critical section, no other processes can be.
2. **Progress** – If none of the processes are executing in their critical sections and some of them wish to enter their critical sections, then one of them will.
3. **Bounded Waiting** – After a process makes a request to enter its critical section, there is a bound on the number of times that other processes are allowed to enter their critical sections, before that request is granted. Essentially, if we have processes, then a single process will have to wait for at most other processes before its request is granted. No process will be allowed to execute twice before all the other processes have even executed once.

## Peterson’s Solution

**Peterson’s Solution** is a software-based solution to the critical section problem. Sadly, it only works with **two processes**, which means it is not scalable.

For this solution, we will be assuming that the and instructions are **atomic**, meaning they cannot be interrupted.

The two data items that will be shared by the two processes are:

1. An integer, , used to keep track of who’s turn it is to enter their critical section next.
2. An array of Boolean flags, , used to indicate if each of the processes is ready to enter its critical section.

For Process 0,

do  
{  
 */\* entry section \*/* flag[0] = true; *// Process 0 is ready* turn = 1; *// it is Process 1’s turn next* while (flag[1] && turn != 0);  
 *// while Process 1 is executing and it is not Process 0’s turn, wait  
   
 /\* critical section \*/  
   
 /\* exit section \*/* flag[0] = false; *// Process 0 no longer wishes to execute  
   
 /\* remainder section \*/*} while (true)

C++

For Process 1,

do  
{  
 */\* entry section \*/* flag[1] = true; *// Process 1 is ready* turn = 0; *// it is Process 0's turn next* while (flag[0] && turn != 1);  
 *// while Process 0 is executing and it is not Process 1’s turn, wait  
   
 /\* critical section \*/  
   
 /\* exit section \*/* flag[1] = false; *// Process 1 no longer wishes to execute  
   
 /\* remainder section \*/*} while (true)

C++

A generalized solution for the two processes can be given by:

do  
{  
 flag[i] = true;  
 turn = j;  
 while (flag[j] && turn == j);  
 */\* critical section \*/* flag[i] = false;   
 */\* remainder section \*/*} while (true)

C++

Here, and for Process 0 and and for Process 1.

Since one process has to wait if the other is executing, this solution exhibits mutual exclusion.

Since a process that has requested permission is granted the permission as soon as the other process stops executing, this solution exhibits progress.

Since at most one process’s critical section is allowed before the requesting process is given permission, this solution exhibits bounded waiting.

All three of these can be seen in the first while loop.

## Hardware Solutions

The basic principle of the **hardware solutions** is that the shared variables are **locked** before a process enters its critical section.

do  
{  
 *// acquire lock  
 // critical section  
 // release lock  
 // remainder section*} while(true);

C++

This prevents the race condition from occurring.

Many systems provide hardware support for critical section code. Uniprocessors for example, can **disable interrupts** when modifying shared data. Since unexpected modifications are prevented, problem solved.

Not really. This solution is **not efficient** for multiprocessor systems. It is not scalable and is time consuming. Instead, modern machines provide special atomic hardware instructions.

### Test and Set Approach

In the **Test and Set Approach**, there is a shared Boolean variable, . This value is initialized to false. If a process finds that this value is false, it can proceed with its critical section. Before entering, it makes the value true, so that other processes cannot.

do  
{  
 while (testAndSet(&lock)); *// while locked, wait  
 // critical section* lock = false; *// unlock when done  
 // remainder section*} while (true);  
  
boolean testAndSet(boolean \*lock)  
{  
 boolean returnVar = \*lock; *// check if unlocked* \*lock = true; *// lock even if already locked* return returnVar;  
}

C++

This solution meets the mutual exclusion requirement as well as the progress requirement, but does not meet the **bounded waiting** requirement, since there is no guarantee that, with two processes, one process will not execute repeatedly. Thus, this is an incomplete solution.

### Compare and Swap Approach

The **Compare and Swap Approach** is almost the same as the Test and Set approach. The only difference is that the value is checked before changing it.

do  
{  
 while (compareAndSwap(&lock, 0, 1)); *// while locked, wait  
 // critical section* lock = 0; *// unlock when done  
 // remainder section*} while (true);  
  
int compareAndSwap(int \*lock, int expVal, int newVal)  
{  
 int returnVar = \*lock; *// check if unlocked* if (\*lock == expVal) \*lock = newVal; *// lock if unlocked* return returnVar;  
}

C++

This approach is also incomplete, in that it does not provide bounded waiting.

### Complete Solution with Test and Set

Previously, we saw that the **Test and Set Approach** was an incomplete solution to the Critical Section problem. However, that approach can be improved upon to create a complete solution.

In the complete solution, we have an additional Boolean variable, **waiting[n]** in addition to the Boolean lock variable. waiting[n] is an array that tracks whether the th process is waiting to enter its critical section or not.

do  
{  
 waiting[i] = true;  
 key = true;  
 while (waiting[i] && key) key = testAndSet(&lock)  
 waiting[i] = false;  
   
 *// critical section* j = (i + 1) % n  
 while ((j != i) && !waiting[j]) j = (j + 1) % n;  
 if (j == i) lock = false;  
 else waiting[j] = false;  
   
 *// remainder section*} while (true);

C++

The **entry section** of this code behaves similarly to the entry section of the original lock and set approach. If the **lock** is not set, then the key will become false and the critical section will be entered.

The **exit section** is where things differ. We take another variable, j, and use it to check, one by one, whether any of the other processes have their **waiting** value set. If we find one, then j and i will have **different values**. In this scenario, we set the waiting value for the **other process** to false, which causes it to enter its critical section. If we do not find another process that is waiting, then j and i will have the **same value** again. In this scenario, we set the **lock** variable to false, so that the system is prepared to serve the next process that asks to enter its critical section.

We already had the **mutual exclusiveness** property due to the entry section. The exit section also gives us the **progress** property and the **bounded waiting** property, when the next waiting process is allowed to enter its critical section, thereby making this solution a complete one.

## Mutex Locks

The word **Mutex** stands for **Mutual Exclusion**. Mutex locks are one of the simplest software tools created to solve the critical section problem by protecting critical sections and preventing the race condition.

A process acquires the lock before entering its critical section and releases it after completing its critical section. A Boolean variable, available, indicates whether or not the lock is available. There are two primitives, acquire() and release(), used to acquire and release the lock respectively.

acquire()  
{  
 while(!available); *// busy; wait* available = false;  
}

release()  
{  
 available = true;  
}

C++

The calls to these primitives must be **atomic**.

The main disadvantage to this method is **busy waiting**. While one process is in its critical state, the others must continuously loop in the call to acquire(). This type of mutex lock is also called a **spin lock**.

In real multiprogramming situations, where a single CPU is being used by multiple processes, spin locks **waste CPU cycles**.

The main advantage, however, is that **no context switching** is needed. Thus, when locks are expected to be held for **short periods of time**, spin locks are useful. Spin locks are also used in multiprocessor systems, where one thread spins on one processor while another thread executes its critical section on another processor.

## Semaphore

Hardware based solutions to the critical section problem are difficult for application programmers to use. To overcome this difficulty, a synchronization tool, called a **semaphore**, can be used.

A semaphore is a **protected integer variable**, S, that can facilitate and restrict access to shared resources in a multi-processing environment. Since it is protected, it cannot be modified directly. Instead, we use two operations to modify it, wait(), which essentially requests for access to the shared resources, and signal(), which broadcasts a message to all the processes to notify them that the shared resources are free.

In the wait() operation, we essentially say this line:

while(S <= 0);

C++

In semaphores, this line is called the **no-operation**. We are putting the process into no-operation. If S is positive however, we decrement it and return the value.

In the signal() operation, if there are no processes waiting to use the shared resource, S is incremented and returned. Otherwise, one of the waiting processes is selected, woken and the signal is returned.

S.wait()  
{  
 while (S<=0); *//no-op* S--;  
}

S.signal()  
{  
 S++;  
}

C++

### Binary Semaphores

In **binary semaphores**, we only have one resource being shared. As such, the integer value can only be or . It must be initialized to . A binary semaphore is the same thing as a mutex lock.

### Counting Semaphore

If we have more than one resource, but the number is limited, the integer value could range over the domain. Such semaphores are called **counting semaphores**.

### Semaphore Usage

Suppose we have two processes, and , that are **simultaneously executing**, each with one statement, and respectively. We need to execute only when has finished execution.

To do this, we make and share a semaphore, synch, which we initialize to .

P1()  
{  
 S1;  
 signal(synch);  
}  
  
P2()  
{  
 wait(synch);  
 S2;  
}

C++

### Deadlock and Starvation

A **deadlock** occurs when one process has one resource and another process has another resource but both resources are required to continue execution. Both processes become stuck indefinitely.

P0  
{  
 wait(S); *//P0 takes up S* wait(Q); *//P1 has Q, so stuck   
 //some code* signal(S);  
 signal(Q);  
}  
  
P1  
{  
 wait(Q); *//P1 takes up Q* wait(S); *//P0 has S, so stuck  
 //some code* signal(Q);  
 signal(S);  
}

C++

**Starvation** occurs when a process is never removed from its waiting state.

## Classic Problems of Synchronization

We will now look into some classical synchronization problems that are used to test nearly every new proposed synchronization tool. We will be using semaphores to present the solutions, since that is the traditional way, but actual implementations could use mutex locks as well.

### Bounded-Buffer Problem

Consider that a producer and a consumer share the following variables:

int n;  
semaphore mutex = 1;  
semaphore empty = n;  
semaphore full = 0;

C++

We are assuming that there are buffers, each of which can hold one item. The mutex semaphore provides mutual exclusion for access to the buffers. The empty and full semaphores count the number of empty and full buffers.

On the **producer** end, we have the following code:

do  
{  
 */\* produce an item in next\_prodcued \*/* wait(empty);  
 wait(mutex);  
 */\* add next\_produced to the buffer \*/* signal(mutex);  
 signal(full);  
} while (true);

C++

On the **consumer** end, we have the following code:

do  
{  
 wait(full);  
 wait(mutex);  
 */\* remove an item from buffer to next\_consumed \*/* signal(mutex);  
 signal(empty);  
 */\* consume the item in next\_consumed \*/*} while (true);

C++

Essentially, the producer is creating full buffers for the consumer and the consumer is creating empty buffers for the producer. The **producer** waits if there are **no empty buffers** or if the shared memory is **locked**. Once it adds an item, it denotes an extra buffer as **full** and **releases** the shared memory. The consumer waits while there are **no full buffers** or if the shared memory is **locked**. Once it consumes an item, it denotes an extra buffer as **empty** and **releases** the shared memory.

### Readers-Writers Problem

Suppose several processes are sharing a **database**. Obviously, if two processes **read** from the database at the same time, there will be no issue, but if a process wants to **write** to a database, every other process must wait.

The readers-writers problem has many variations, all involving priorities. The simplest one, the **first readers-writers problem**, states that a reader cannot be kept waiting unless a writer has already obtained permission to use the shared resource. This just means that no reader should wait for another reader simply because a writer is waiting. This is opposed to the **second readers-writers problem**, which states that if a writer is waiting for access, no new readers may start reading, since that would increase the writer’s waiting time.

Solutions to either type of problem may lead to **starvation**. In the first case, writers may starve and in the second case readers may starve. Because of this, alternative variations of the problem have been proposed.

We will just be looking into the solution for the **first readers-writers problem**. The following data is shared:

semaphore rw\_mutex = 1;  
semaphore mutex = 1;  
int read\_count = 0;

C++

The ru\_mutex­ semaphore is the one that prevents writers from writing if there are readers reading and vice versa. The ­read\_count variable keeps track of whether there are readers reading. The mutex semaphore protects the read\_count variable from simultaneous access.

The code for the **writers** is given below:

do  
{  
 wait(rw\_mutex);  
 */\* writing is performed \*/* signal (rw\_mutex);  
} while(true);

C++

The code for the **readers** is given below:

do  
{  
 wait(mutex);  
 read\_count++;  
 if (read\_count == 1)  
 wait(rw\_mutex);  
 signal(mutex);  
 */\* reading is performed \*/* wait(mutex);  
 read\_count--;  
 if (read\_count == 0)  
 signal(rw\_mutex);  
 signal(mutex);  
} while(true);

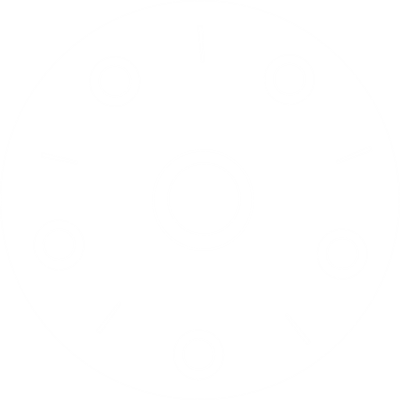
C++

Notice that if a writer is writing, only a **single reader** is waiting on the rw\_mutex lock while all the others are waiting on the mutex lock. Also notice that a writer releasing the rw\_mutex­ lock allows us to either give it to another writer or let the readers use it. The selection is made by the **scheduler**.

The two locks can also be generalized to a reader\_writer\_lock, which in turn has a **reader mode** and a **writer mode**.

### Dining-Philosophers Problem

Consider there are a few people sitting in a round table thinking. They each have food in front of them and a single chopstick. Since there is just one chopstick per person, if one of the people wish to eat, the person next to them will be unable to do so.



This seemingly absurd problem is actually an example of a large class of concurrency control problems.

One simple solution is to represent each chopstick as a semaphore.

do  
{  
 wait(chopstick[i]);  
 wait(chopstick[(i + 1) % 5]);  
 */\* eat for a while \*/* signal(chopstick[i]);  
 signal(chopstick[(i + 1) % 5]);  
 */\* think a while \*/*} while(true);

C++

However, this solution could create a **deadlock**. If all people grab their own chopstick at the same time, everyone will be stuck. This could be solved in several ways:

* Allow people to be sitting at the table at the same time.
* Allow a person to pick up their chopstick only if both are available. This itself must be done in a critical section.
* Use an asymmetric solution, where people sitting in odd-numbered seats take their left chopstick first and then the right one and people sitting in even-numbered seats take their right chopstick first and then the left one.

Note however, that even a deadlock free solution does not prevent **starvation**, so we will need to guard against that explicitly.