College of Natural and Computational Science

Department of Computer Science

Operating System Assignment

Title: Classical Problems of Process Synchronization

Name: Biruktawit Nibret

Id: UGR/9355/12

Section: one

Submitted to: Mr. Ashenafi

Submission date: April 15, 2022

Table of Contents

Introduction

In the Operating System, there are a number of processes present in a particular state. At the same time, we have a limited amount of resources present, so those resources need to be shared among various processes. But you should make sure that no two processes are using the same resource at the same time because this may lead to data inconsistency. So, synchronization of process should be there in the Operating System for cooperative processes which are processes that share resources between each other. [16]

Process Synchronization is mainly needed in a multi-process system when multiple processes are running together and more than one process try to gain access to the same shared resource or any data at the same time. It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources. It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes. [17]

In order to synchronize the processes, there are various synchronization mechanisms like semaphores and monitors.

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations wait ( ) and signal ( ) that are used for process synchronization. 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. [15]

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

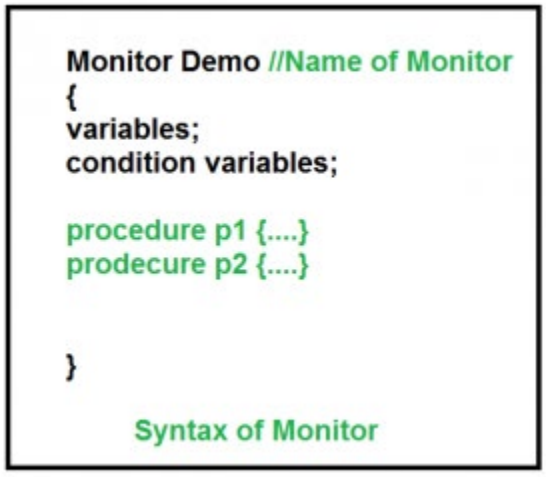
```
signal (S){
        S++;
}
```

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Counting semaphores are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented. The binary semaphores

are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores. [15]

Semaphores have both advantages and disadvantages. Semaphores allow only one process into the critical section. They are implemented in the machine independent code of the microkernel. But they are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks. They may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later. [15]

The monitor is the other way to achieve process synchronization. It is supported by programming languages to achieve mutual exclusion between processes. It is the collection of condition variables and procedures combined together in a special kind of module or a package. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor. Only one process at a time can execute code inside monitors. [14]

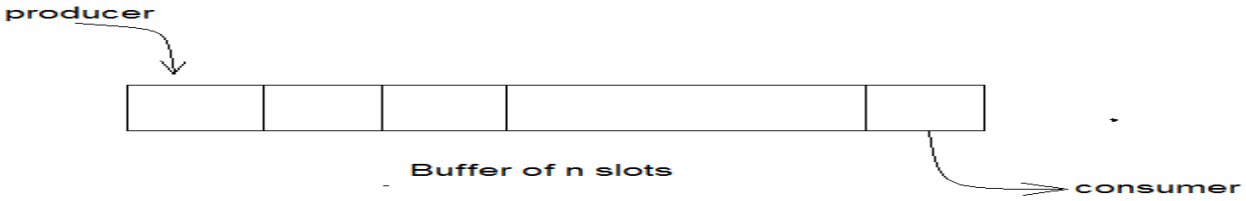


Two different operations are performed on the condition variables of the monitor: wait and signal. For any condition variable x, x.wait() performs wait operation on any condition variable to suspend it. The suspended processes are placed in block queue of that condition variable. While x.signal() performs signal operation on condition variable, one of the blocked processes is given chance. Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore. But they have to be implemented as part of the programming language. The compiler must generate code for them. This gives the

compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. [14]

Bounded Buffer (Producer-Consumer) Problem

The bounded-buffer problem is a classic example of concurrent access to a shared resource, buffer. A bounded buffer lets multiple producers and multiple consumers share a single buffer. Producers write data to the buffer and consumers read data from the buffer. [1] There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer. [2]



Things to take care:

- ✓ Producer inserts to empty slot. It must be blocked if the buffer is full. [1]
- ✓ Consumer removes from filled slot. It must be blocked if the buffer is empty. [1]

Problem:

The two processes should not insert and remove data to and from the buffer simultaneously.

Solution:

There is a solution using three semaphores: mutex, empty and full. [2]

- ✓ **mutex,** a binary semaphore which is used to acquire and release the lock.
- ✓ **empty**, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- ✓ **full,** a counting semaphore whose initial value is 0 since initially no slot is full.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer. [2]

The Producer Operation:

```
do {
        wait (empty);          // wait until empty >0 and decrement empty
        wait (mutex);          //acquire lock
        // insert to the buffer
        signal (mutex);        //release lock
        signal (full);         //increment full
}
```

Looking at the above code for a producer, we can see that a producer first waits until there is at least one empty slot. Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots. Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation. After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer. [2]

The Consumer Operation:

```
do {
        wait (full);           //wait for full>0 and decrements full
        wait (mutex);          //acquire lock
        //remove data from buffer
        signal (mutex);                //release lock
        signal (empty);                //increment empty
}
```

The consumer waits until there is at least one full slot in the buffer. Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation. After that, the consumer acquires lock on the buffer. Following that, the consumer completes the removal operation so that the data from one of the full slots is removed. Then, the consumer releases the lock. Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty. [2]

So, the bounded buffer problem is solved using semaphores by letting only one of the processes which acquire the lock to use the buffer.

Reader-Writer Problem

In an Operating System, we deal with various processes and these processes may use files that are present in the system. Basically, we perform two operations on a file i.e. read and write. All these processes can perform these two operations. [6]

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers which only want to read the data from the object and some of the processes are writers which want to write into the object. The readers-writers problem is used to manage synchronization so that there are no problems with the object data. [3]

But the problem that arises when: [6]

- ✓ If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state. Only one process should be allowed to change the value of the data present in the file at a particular instant of time.
- ✓ Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read because the process writing on the file will change the value of the file, but the process reading that file will read the old value present in the file. So, this should be avoided.

Problem parameters: [4]

- ✓ One set of data is shared among a number of processes.
- ✓ Once a writer is ready, it performs its write. Only one writer may write at a time.
- ✓ If a process is writing, no other process can read it.
- ✓ If at least one reader is reading, no other process can write.
- ✓ Readers may not write but only read.

Solution:

The following is the proposed solution: [6]

- ✓ If a process is performing some write operation, then no other process should be allowed to perform the read or the write operation.
- ✓ If a process is performing some read operation only, then another process that is demanding for reading operation should be allowed to read the file and get into the critical section because the read operation doesn't change anything in the file. So, more than one reads are allowed. But if a process is reading a file and another process is demanding for the write operation, then it should not be allowed.

Here, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource. [5]

Two semaphores: mutex and wrt and one integer variable: readcnt are used to solve this problem. [6]

- ✓ **wrt:** This semaphore is used to achieve the mutual exclusion property. It is used by the process that is writing in the file and it ensures that no other process should enter the critical section at that instant of time. Initially, it will be set to "1".
- ✓ **mutex:** This semaphore is used to achieve mutual exclusion during changing the variable that is storing the count of the processes that are reading a particular file. Initially, it will be set to "1".
- ✓ **readcnt:** will have the count of the processes that are reading a particular file. It is initially initialized to 0.

The Writer Process:

```
do {
        wait (wrt);
      // performs the write and leaves critical section
      signal (wrt);
} while (true);
```

The **wait (wrt)** function is called so that it achieves the mutual exclusion. The wait () function will reduce **wrt** value to "0" and this will block other processes to enter into the critical section. The write operation will be carried and finally, the **signal (wrt)** function will be called and the value of **wrt** will be again set to "1" and now other processes will be allowed to enter into the critical section. [6]

The Reader Process:

```
do {
     wait (mutex);
      readcnt++;
      if (readcnt==1)
            wait (wrt);
      signal (mutex);
      // current reader performs reading here
      wait (mutex);   // a reader wants to leave
      readcnt--;
      // that is, no reader is left in the critical section,
      if (readcnt == 0)
          signal (wrt);        // writers can enter
      signal (mutex); // reader leaves
} while (true);
```

We are using the **mutex** variable to change something in the **readcnt** variable. This is done because if some process is changing something in the **readcnt** variable, then no other process should be allowed to use that variable. So, to achieve mutual exclusion, we are using the mutex

variable. Initially, we are calling the **wait (mutex)** function and this will reduce the value of the mutex by one. After that, the **readcnt** value will be increased by one. If the **readcnt** variable is equal to "1" i.e. the reader process is the first process, in this case, no other process demanding for write operation will be allowed to enter into the critical section. So, the **wait (wrt)** will be called and the value of the writer variable will be decreased to "0" and no other process demanding for write operation will be allowed. After changing the **readcnt** variable, the value of the **mutex** variable will be increased by one, so that other processes should be allowed to change the value of the **readcnt** value. The read operation by various processes will be continued and after that when the read operation is done, then again we have to change the count the value of the **readcnt** and decrease the value by one. If the **readcnt** becomes "0", then we have to increase the value of **wrt** variable by one by calling the **signal (wrt)** function. This is done because if the **readcnt** is "0" then other writer processes should be allowed to enter into the critical section to write the data in the file. [6]

So, the reader-writer problem is solved by giving readers priority over writers using semaphores.

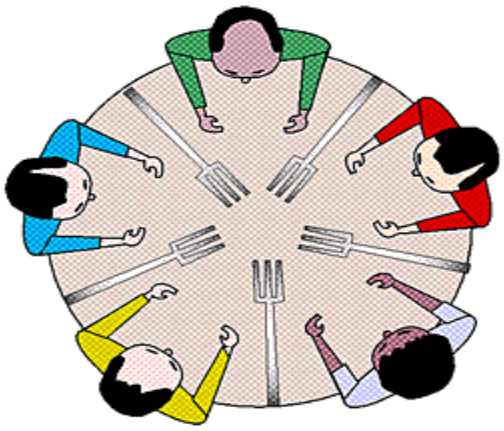
Dining Philosophers Problem

The dining philosopher's problem is the classical problem of synchronization which says that five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available. If both immediate left and right chopsticks of the philosopher are not available, then the philosopher puts down their (either left or right) chopstick and starts thinking again. [7]

Problem:

Since the chopsticks are limited, no two adjacent philosophers eat at the same time or in other words no two philosophers can access the same chopstick simultaneously. This problem resembles a resource allocation problem in operating system where the philosophers are the

processes and the chopsticks are the resources. Thus, solving this problem means giving answer to “How resources are utilized when shared?”



Solution:

1. Using semaphores:

This problem can be solved using a binary semaphore for each chopstick (array of semaphores). Then only one philosopher can use it at a time. The wait () operation is used to grab the chopstick and signal () operation is used to release it. The array is initially initialized to 1 since all chopsticks are free.

Philosopher Function:

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

Let value of i = 0 initially. Suppose Philosopher P0 wants to eat, it will enter in Philosopher function, and execute **Wait (chopstick[i]);** by doing this it holds **chopstick [0]** and reduces the semaphore to 0**.** Then it execute **Wait (chopstick [(i+1) % 5]);** by doing this it holds **chopstick [1]** (since i =0, therefore (0 + 1) % 5 = 1) and reduces the semaphore 0. Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher function, and execute **Wait(chopstick[i]);** by doing this it will try to hold **chopstick [1]** but will not be able to do that**,** since the value of semaphore has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick [1] whereas if Philosopher P2 wants to eat, it will enter in Philosopher function, and execute **Wait(chopstick[i]);** by doing this it holds **chopstick [2]** and reduces semaphore C2 to 0, after that, it executes **Wait(chopstick[(i+1) % 5]);** by doing this it holds **chopstick [3]** (since i =2, therefore (2 + 1) % 5 = 3) and reduces the semaphore to 0. [7]

The drawback of the above solution

The drawback is that this solution can lead to a deadlock condition. A deadlock state of a system is a state in which no progress of system is possible. This situation happens if all the philosophers pick their left chopstick at the same time, which leads to the condition of deadlock and none of the philosophers can eat. [7]

To avoid deadlock, some of the solutions are: [7]

- ✓ Maximum number of philosophers on the table should not be more than four. In this case, one chopstick will be available for one of the philosophers so that he can eat and put down the chopstick when he finishes.
- ✓ A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick.
- ✓ Only in case if both the chopsticks (left and right) are available at the same time, only then a philosopher should be allowed to pick their chopsticks
- ✓ All the four starting philosophers should pick the left chopstick and then the right chopstick, whereas the last philosopher should pick the right chopstick and then the left

chopstick. This will force the last philosopher to hold his right chopstick first which is already held by the first philosopher and its value is set to 0 because of which the last philosopher will get trapped into an infinite loop and chopstick [4] remains vacant. Hence philosopher P3 has both left and right chopstick available, therefore it will start eating and will put down its both chopsticks once finishes and let others eat which removes the problem of deadlock.

2. Monitor based solution:

This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure: [8]

- ✓ THINKING**:** When philosopher doesn't want to gain access to either fork.
- ✓ HUNGRY**:** When philosopher wants to enter the critical section.
- ✓ EATING**:** When philosopher has got both the forks, i.e., he has entered the section.

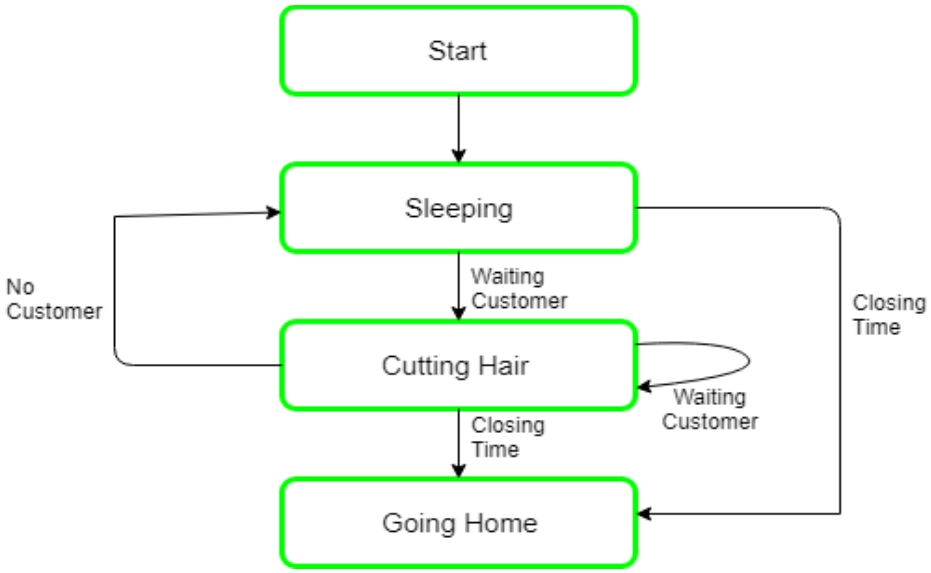
```
monitor dp{
        enum {T, H, E} state [5];
        condition self [5];
        void pickup (int i){
                state [i] = H;
                test (i);
                if (state [i]!=E)
                self [i].wait();
        }
        void putdown (int i){
                state [i] = T;
```

```
            test ((i+4)%5);
            test ((i+1)%5);
        }
        void test (int i) {
            if ((state[(i+4)%5]!=E) && (state [(i+1)%5]!=E) && (state[i] == H) {
            state [i] == E;
            self [i].signal();}
        }
}
```

This allows philosopher i to delay when he is hungry but is unable to obtain the chopsticks needed. The distribution of the chopsticks is controlled by the monitor Dining Philosophers (dp). Each philosopher, before starting to eat, must invoke the operation pickup (). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown () operation. It is easy to show that this solution ensures that **no two neighbors** are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. [8] If philosophers take two forks at a time, there is a possibility of starvation. Philosophers P2 & P5 and P1 & P3 can alternate in a way that starves out philosopher P4. This possibility of starvation means that any solution to the problem must include some provision for preventing starvation. [9]

Sleeping Barber Problem

The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair. If there is no customer, then the barber sleeps in his own chair. When a customer arrives, he has to wake up the barber. If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty. [17]



Problem:

1. Deadlock:

A scenario in which customer end up waiting on barber or a barber waiting on the customer will result a deadlock. It is solved using three semaphores: customer, barber, mutex and integer variable: FreeSeats. [17]

- ✓ **customer**: count waiting customers (customer in the barber chair is not included because he is not waiting).
- ✓ **barber**: check the status of the barber whether it is idle (0) or working (1).
- ✓ **mutex**: allow customer to get exclusive access to number of free seats and allow them to increase or decrease.
- ✓ **FreeSeats**: count number of free seats in waiting room.

The implementation is:

```
Semaphore customers = 0;
Semaphore barber = 0;
mutex = 1;
int FreeSeats = N;
Barber {
        while(true) {
                wait (customers);
                wait (mutex);
```

```
                FreeSeats--;
                signal (barber);
                signal (mutex);
                /* barber is cutting hair.*/
        }
}
Customer {
        while(true) {
                wait (mutex);.
                if(FreeSeats > 0) {
                        FreeSeats++;
                        signal (Customers);
                        signal (Seats);
                        wait (Barber);
                        // customer is having hair cut
                } else {
                        signal (Seats);
                        // customer leaves
                }
        }
}
```

When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up. When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less than the number of chairs then he sits otherwise he leaves and releases the mutex. If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping. At this point, customer and barber are both awake and the barber is ready to give that person a haircut.

When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps. [17]

2. Starvation

Starvation may occur when customer wait for a long time if either customer does not follow order or barber calls out randomly. The solution is utilizing a queue where customers are added as they arrive.

Reference

1. http://www.it.uu.se/education/course/homepage/os/vt18/module-4/bounded-buffer/ Last visited: April 10, 2022.
2. https://www.studytonight.com/operating-system/bounded-buffer Last visited: April 10,2022
3. https://www.tutorialspoint.com/readers-writers-problem Last visited: April 10, 2022
4. https://www.geeksforgeeks.org/readers-writers-problem-set-1-introduction-and-readers-preference-solution/ Last visited: April 10, 2022
5. https://www.studytonight.com/operating-system/readers-writer-problem Last visited: April 10,2022
6. https://afteracademy.com/blog/the-reader-writer-problem-in-operating-system Last visited: April 10, 2022
7. https://www.javatpoint.com/os-dining-philosophers-problem Last visited: April 10, 2022
8. https://www.geeksforgeeks.org/dining-philosophers-solution-using-monitors/ Last visited: April 10, 2022
9. https://www.codingninjas.com/codestudio/library/dining-philosopher-solution-using-monitors Last visited: April 10, 2022
10. https://www.studytonight.com/operating-system/process-synchronization Last visited: April 10, 2022
11. https://afteracademy.com/blog/what-is-process-synchronization-in-operating-system Last visited: April 10, 2022
12. https://www.tutorialspoint.com/semaphores-in-operating-system Last visited: April 10, 2022
13. https://www.geeksforgeeks.org/monitors-in-process-synchronization/ Last visited: April 10, 2022
14. https://www.tutorialspoint.com/semaphores-in-operating-system Last visited: April 10, 2022
15. https://afteracademy.com/blog/what-is-process-synchronization-in-operating-system Last visited: April 10, 2022
16. https://www.studytonight.com/operating-system/process-synchronization Last visited: April 10, 2022
17. https://www.geeksforgeeks.org/sleeping-barber-problem-in-process-synchronization/ Last visited: April 10, 2022.