Process Synchronization

Process Synchronization

- On the basis of synchronization, processes are categorized as one of the following two types:
- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.
- Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Process Synchronization

- Processes can execute concurrently
    - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer `counter` that keeps track of the number of full buffers. Initially, `counter` is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Race Condition

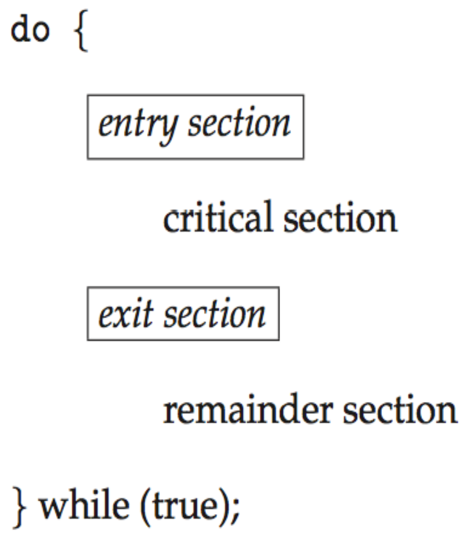
- When more than one processes are executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition.
- A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute.
- Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Critical Section Problem

- Critical Consider system of ***n*** processes {$\boldsymbol{p_0}$, $\boldsymbol{p_1}$, **…** $\boldsymbol{p_{n-1}}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**
- Section Problem

Critical Section

- General structure of process $\boldsymbol{P_i}$



```
do {
     entry section
          critical section
     exit section
          remainder section
} while (true);
```

Solution to Critical-Section Problem

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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** - A bound 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

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Peterson’s Solution

- Peterson’s Solution is a classical software based solution to the critical section problem.
- In Peterson’s solution, we have two shared variables:

  (a) Boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section

  (b) int turn : The process whose turn is to enter the critical section.

```
do {

      flag[i] = TRUE ;
      turn = j ;
      while (flag[j]  &&  turn == j) ;

          critial section

      flag[i] = FALSE ;

          remainder section

   }  while (TRUE) ;
```

Peterson’s Solution

- Peterson’s Solution preserves all three conditions :
- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance. Disadvantages of Peterson’s Solution
  - It involves Busy waiting
  - It is limited to 2 processes.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
 } while (TRUE);
```

TestAndSet

- TestAndSet is a hardware solution to the synchronization problem.
- In TestAndSet, we have a shared lock variable which can take either of the two values, 0 or 1.
- 0 Unlock 1 Lock.
- Before entering into the critical section, a process inquires about the lock. If it is locked, it keeps on waiting until it becomes free and if it is not locked, it takes the lock and executes the critical section.

Definition:

```
        boolean test_and_set (boolean *target)
            {
                    boolean rv = *target;
                    *target = TRUE;
                    return rv:
            }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE

In TestAndSet, Mutual exclusion and progress are preserved but bounded waiting cannot be preserved.

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization

Producer-Consumer problem

- **i) Shared Memory Method**

Ex: Producer-Consumer problem

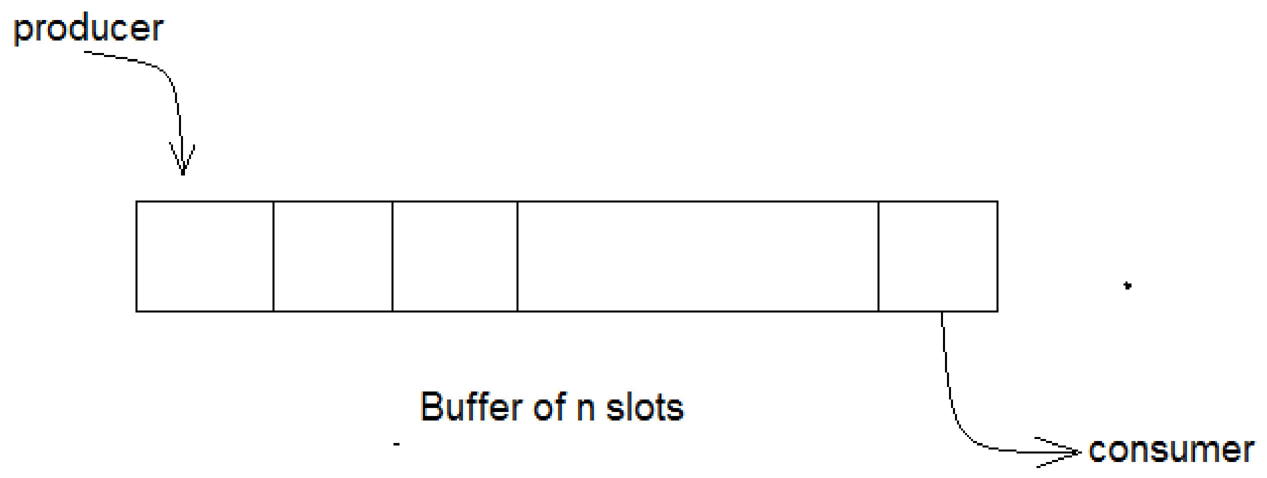
- There are two processes: Producer and Consumer.
- The producer produces some items and the Consumer consumes that item.
- The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed.
- There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it.

bounded buffer problem

- First, the Producer and the Consumer will share some common memory, then the producer will start producing items.
- If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer.
- Similarly, the consumer will first check for the availability of the item.
- If no item is available, the Consumer will wait for the Producer to produce it.
- If there are items available, Consumer will consume them.

Bounded Buffer Problem

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.



- A producer tries to insert data into an empty slot of the buffer.
- A consumer tries to remove data from a filled slot in the buffer.
- So, those two processes won't produce the expected output if they are being executed concurrently.
- There needs to be a way to make the producer and consumer work in an independent manner.

- One solution of this problem is to use semaphores.
- m, 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.
- 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.

The Producer Operation

- The structure of the producer process

```
do {
        ...
       /* produce an item in next_produced */
        ...
     wait(empty);
     wait(mutex);
         ...
       /* add next produced to the buffer */
         ...
     signal(mutex);
     signal(full);
  } while (true);
```

The Producer Operation

- 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.

The Consumer Operation

- The structure of the consumer process

```
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);
```

The Consumer Operation

- 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.

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem

- There is a shared resource which should be accessed by multiple processes.
- There are two types of processes in this context.
- They are **reader** and **writer**.
- Any number of **readers** can read from the shared resource simultaneously,
- but only one **writer** can write to the shared resource.
- When a **writer** is writing data to the resource, no other process can access the resource.
- A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

Readers-Writers Problem

- From the above problem statement, the readers that 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.
- Here, we use one **mutex** `m` and a **semaphore** `w`.
- An integer variable `read_count` is used to maintain the number of readers currently accessing the resource.
- The variable `read_count` is initialized to `0`.
- A value of `1` is given initially to `m` and `w`.

Readers-Writers Problem

- The structure of a writer process

```
do {
     wait(rw_mutex);
          ...
     /* writing is performed */
          ...
    signal(rw_mutex);
} while (true);
```

Readers-Writers Problem

- The structure of a reader process

```
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);
```

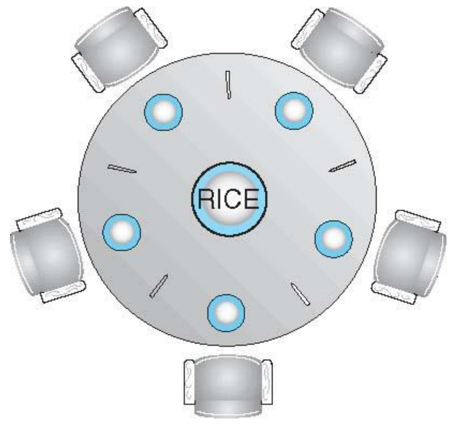
Readers-Writers Problem

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Dining Philosophers Problem

This problem is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
    - Need both to eat, then release both when done
- In the case of 5 philosophers
    - Shared data
        - Bowl of rice (data set)
        - Semaphore chopstick [5] initialized to 1



Dining Philosophers Problem

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                 //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                  //  think

} while (TRUE);
```

Dining Philosophers Problem

- From the problem statement, it is clear that a philosopher can think for an indefinite amount of time.
- But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.
- An array of five semaphores, stick[5], for each of the five chopsticks.
- When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.
- But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:
- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

References

- www.csee.wvu.edu/~jdmooney/classes/cs550/notes/tech/mutex/Peterson.html
- Operating System Concepts by Galvin et al.
- Lecture notes/ppt of Ariel J. Frank, Bar-Ilan University