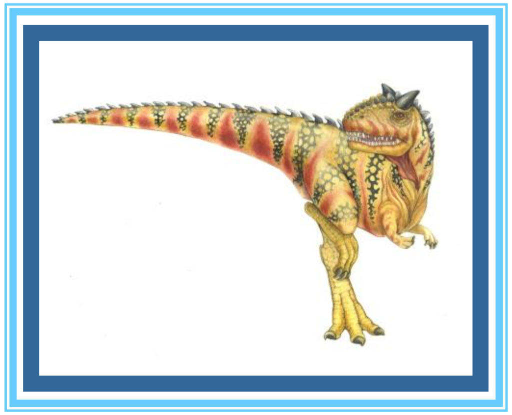
Chapter 6: Process Synchronization



Module 6: Process Synchronization

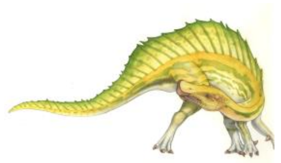
- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Process Synchronization

- Process Synchronization is a task in a multi-process system to ensure that the access of shared resources in done in a controlled and predictable manner.
- It aims to resolve the problem of **race conditions** and other synchronization issues in a **concurrent system.**
- The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of **inconsistent data** due to concurrent access.



Background

Consider the example of Producer-consumer problem that fills **all** the buffers.

- We can do so by having an integer **count** that keeps track of the number of full buffers.
- Initially, count 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.

Producer

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

Consumer

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

Race Condition

- count++ could be implemented as in machine language
  register1 = count   //register 1 is local CPU registers
  register1 = register1 + 1
  count = register1

- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with “count = 5” initially:

  S0: producer execute register1 = count   {register1 = 5}
  S1: producer execute register1 = register1 + 1   {register1 = 6}
  S2: consumer execute register2 = count   {register2 = 5}
  S3: consumer execute register2 = register2 - 1   {register2 = 4}
  S4: producer execute count = register1   {count = 6 }
  S5: consumer execute count = register2   {count = 4}

Race Condition

- 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 we need to ensure that only one process at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the processes be **synchronized** using **process synchronization** techniques

Critical Section Problem

Critical section problem is a means of designing a way for **cooperative processes** to access **shared resources** without creating data inconsistencies.

A critical section is a **code segment** that can be accessed by only one process at a time.

Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}

- Each process has **critical section**
- Here, 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
- Each process must ask permission to enter critical section in **entry section**, that follow **critical section** with **exit section**, then **remainder section**

Critical Section

- General structure of process ***P_i***

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

entry section: Each process must request permission to enter its critical section. Entry section is the section of code implementing this request.

Solution to Critical-Section Problem

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

- **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

Is a classic software-based solution to the critical-section problem

- Is restricted to **two process solution**
- 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 Pi is ready!

Algorithm for Process P_i

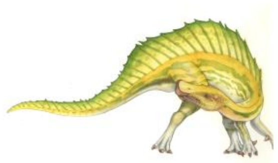
```
do {
     flag[i] = true;
     turn = j;
while (flag[j] && turn = = j); //wait until j exit
          critical section
flag[i] = false;//indicate Pi is out of critical section
          remainder section
 } while (true);
```

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.

Synchronization Hardware

- Many systems provide **hardware support** for implementing the **critical section code**.
- All solutions are based on idea of **locking**
  - Protecting critical regions via locks
- In Uniprocessors – **interrupts could be disabled** from occurring during a shared variable was being modified.
- Currently running code would execute without preemption
  - Generally **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);
```

- The abstract of main concepts behind these hardware instructions for critical section problem are discussed using
- **test and set()** and **compare and swap()** instructions.

TestAndSet Instruction

- Definition:

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

Must be executed atomically

Solution using TestAndSet

- Shared Boolean variable lock, initialized to false
- Solution:

```
do {
        while ( TestAndSet (&lock ))
                ;   // do nothing
             //    critical section
        lock = FALSE;
             //      remainder section
 } while (TRUE);
```

TestAndSet Instruction

- **Test and Set:** The shared variable **lock** is initialized to false.
- TestAndSet(lock) algorithm working –
- It always returns whatever value is sent to it and sets lock to **true.**
- The first process will enter the critical section at once as TestAndSet(lock) will return **false** and it'll break out of the while loop.
- The other processes cannot enter now as lock is set to **true** and so the while loop continues to be true.
- Mutual exclusion is ensured.

TestAndSet Instruction

- Once the first process gets out of the critical section, lock is changed to **false**
- So, now the other processes can enter one by one. Progress is also ensured.
- However, after the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter.
- So bounded waiting is not ensured.

compare_and_swap Instruction

Definition:

```
 int compare _and_swap(int *value, int expected, int new_value) {
     int temp = *value;
     if (*value == expected)
        *value = new_value;
       return temp;
 }
```

- Returns the original value of passed parameter **"value"**
- **value** is set to new value only if the expression (*value == expected) is true.

Solution using compare_and_swap

- Shared integer "lock" initialized to **0 (false)**
- **Solution:**

```
      do {
        while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
    /* critical section */
  lock = 0;
    /* remainder section */
 } while (true);
```

compare_and_swap Instruction

Mutual exclusion can be provided as follows:

- A global variable **(lock)** is declared and is **initialized to 0.**
- The first process that invokes compare and swap() will set **lock to 1**.
- It will then enter its critical section, because the original value of lock (i.e 0) was equal to the expected value of 0.
- Subsequent calls to compare and swap() will not succeed, because lock now is not equal to the expected value of 0 **(changed to 1 by previous entered process).**
- When a process exits its critical section, **it sets lock back to 0**, which allows another process to enter its critical section.

Bounded-waiting Mutual Exclusion with TestandSet()

The common data structures are

- boolean waiting[n]; //for each process which checks whether or not a process has been waiting.
- boolean lock; //These data structures are initialized to **false**
- A ready queue is maintained with respect to the process in the critical section.
- All the processes coming in next are added to the ready queue with respect to their process number,
- Once the ith process gets out of the critical section, it does not turn lock to false, Instead, it checks if there is any process waiting in the queue.
- If there is **no process** waiting then the **lock value is changed to false** and any process which comes next can enter the critical section.
- If there is, **then that process' waiting value is turned to false**, so that the **first while loop** becomes false and it can enter the critical section.
- This ensures **bounded waiting**.
- Thus, the problem of process synchronization can be solved through this algorithm.

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {   waiting[i] = TRUE;
       key = TRUE;
       while (waiting[i] && key)    // loop will continue until key becomes false
       key = TestAndSet(&lock); // TestAnsSet will set the value of key
       waiting[i] = FALSE;
               // critical section
       j = (i + 1) % n;
       while ((j != i) && !waiting[j]) // check any process is waiting in queue
               j = (j + 1) % n;
       if (j == i)
               lock = FALSE;
       else
               waiting[j] = FALSE; // Indicates that the process j is selected
               // remainder section
} while (TRUE);
```

Mutex Locks

- Previous hardware solutions are complicated and generally inaccessible to application programmers
- OS designers build **software tools** to solve critical section problem. Simplest is **mutex** lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Uses boolean variable "available" indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

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

-
  ```
  release() {
     available = true;
  }
  ```

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

Semaphore

- Semaphore is a synchronization tool similar to mutex but more sophisticated that does not require busy waiting. It can be used to manage access to a **pool of resources** rather than just one
- Semaphore *S* – **integer variable**
- Semphore is accessed through two standard atomic operations wait() and signal()
  - Originally called P() and V()
- Less complicated

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

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

Semaphore

- **Wait (P) Operation:**
  - If the semaphore's count is greater than zero, decrement the count and allow the thread to proceed.
  - If the count is zero, the thread is blocked until the count becomes greater than zero.
- **Signal (V) Operation:**
  - Increment the semaphore's count.
  - If there are threads blocked on a wait operation, one of them is unblocked.

Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
        // Critical Section
     signal (mutex);
      // remainder section
} while (TRUE);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Deadlock and Starvation

- Implementation of a semaphore with a waiting queue may result in a deadlock state, if one or two process are waiting for a event that can be caused by only one of the waiting process. Such a state is called **deadlock.**
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
              P0                          P1
          wait (S);                   wait (Q);
          wait (Q);                    wait (S);   .
             .                           .
             .                           .
        signal  (S);                 signal (Q);
         signal (Q);                  signal (S);
```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- As an example, assume we have **three processes: *L*, *M*, and *H***, whose priorities follow the order *L* < *M* < *H*.
- Assume **process *H*** requires **resource *R***, which is currently being accessed by **process *L***.
- Suppose that **process *M*** becomes runnable, thereby preempting process *L*. Indirectly, a process with a lower priority: **process *M***, has affected how long process *H* must wait for *L*(**Priority Inversion)**
- **priority-inheritance protocol** is used to solve this problem:
- According to this protocol, all processes that are accessing resources needed by a higher-priority process **inherit the higher priority** until they are finished with the resources. When they are finished, their priorities revert to their original values.
- In this example **process *L*** will inherit the priority of **process *H***, thereby preventing **process *M*** from preempting its execution.
- Thus, resource *R* would now be available, process *H* not *M*.