Process Management

Synchronization

Introduction

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space (threads) or be allowed to share data only through files or messages.
- While concurrency is desirable to achieve parallelism, concurrent access to shared data may result in data inconsistency.
- In this unit, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space so that data consistency is maintained.

Introduction

- Consider a scenario in which two processes are sharing a global variable counter. In code of process 1, there is a statement counter++ and in code of process 2 there is a statement counter - -

Process 1

```
{
   ---
  counter++ ;
}
```

Process 2

```
{
   ---
  counter - - ;
}
```


Introduction

- The statements **counter++ and counter-- ;** must be performed *atomically*.
- Atomic operation means an operation that completes in its entirety without interruption.
- The statements counter++ may be implemented in machine language as:
  register1 = counter
  register1 = register1 + 1
  counter = register1

Introduction

- The counter - - may be implemented in machine language as:
  register2 = counter
  register2 = register2 – 1
  counter = register2
- If both the processes attempt to update the value of counters concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the two processes are scheduled.

Introduction

- Assume **counter** is initially 5. One interleaving of statements is:

  Process 1: **register1 = counter** (*register1 = 5*)
  Process 1 : **register1 = register1 + 1** (*register1 = 6*)
  Process 2 : **register2 = counter** (*register2 = 5*)
  Process 2 : **register2 = register2 – 1** (*register2 = 4*)
  Process 1 : **counter = register1** (*counter = 6*)
  Process 2 : **counter = register2** (*counter = 4*)

- The value of **counter** may be either 4 or 6, where the correct result should be 5.

Introduction

- The situation where several processes access and manipulate shared data concurrently and the outcome or final value of the shared data depends on the particular order in which the access takes place, is called a **RACE CONDITION**.
- To prevent race conditions, concurrent processes must be **SYNCHRONIZED**.

The Critical-Section Problem

- Consider a system consisting of n processes [P0, P1, P2, . . . Pn] all competing to use some shared data
- Each process has a code segment, called the **critical section**, in which the shared data is accessed.
- Problem – To ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

The Critical-Section Problem

- Solution - Design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. This section of code is called the Entry Section. The critical section may be followed by the Exit section. The remaining code is the Remainder section.

```
do {
        Entry Section
            Critical Section
        Exit Section
            Remainder Section
} while (TRUE);
```


Solution to Critical-Section Problem

1. **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.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the *n* processes.

Initial Attempts to Solve Problem

- Assume only 2 processes, P_i and P_j are cooperating
- General structure of process P_i

```
do {
        entry section
          critical section
        exit section
          reminder section
} while (1);
```


Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn = i** $\Rightarrow$ P_i can enter its critical section
- Process P_i

```
do {
   while (turn != i) ; /busy waiting
      critical section
   turn = j;
      reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement; strictly turn making, a process crashing before giving turn to the other process blocks the other process

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow$ P_i ready to enter its critical section
- Process P_i

```
do {
   flag[i] := true;
   while (flag[j]) ;
   critical section
   flag [i] = false;
      remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement; may lead to deadlock; this may happen if both the processes set their flag simultaneously and then are busy waiting

Algorithm 3: Peterson's Solution

- Shared variables
  - **boolean flag[2]**; **int turn;**
    initially **flag [0] = flag [1] = false**
- The variable turn indicates whose turn it is to enter the critical section and the flag array indicate if a process is ready to enter its critical section.
- To enter the critical section, process Pi first sets its own flag as true and then gives turn to Pj, thereby asserting that if the other process wishes to enter the critical section it can do so.
- If both the 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 final value of turn decides which of the two processes is allowed to enter its critical section first.

Algorithm 3: Peterson's Solution

- Process P_i

```
do {
   flag [i]:= true;
   turn = j;
   while (flag [j] and turn = j) ;
     critical section
   flag [i] = false;
     remainder section
} while (1);
```


Algorithm 3: Peterson's Solution

- **Mutual exclusion is preserved:** Process Pi can enter its critical section only when flag[j] = false or turn = i. For both processes to be in the critical section flag[i] = flag[j] should be true and turn should be set to both i and j; which is not possible. Eventually the process whose turn is set, only that can enter the critical section. Thus, Mutual exclusion is preserved.

Algorithm 3: Peterson's Solution

- **Progress and bounded waiting requirement is met:** Process Pi can be prevented from entering the critical section only if it is stuck in the while loop with flag[j] = true and turn = j. If Pj is not ready to enter the critical section then flag[j] = false and Pi can enter the critical section. If Pj wants to enter the critical section than it has set its flag as true and is also executing its while loop. Now who will enter the critical section that depends solely on the value of turn. Either of the process can enter and after exiting will make its flag as false letting other process to enter.

Algorithm 3: Peterson's Solution

- Consider the scenario when Pj sets its flag to true and sets the turn to i. At this point, if Pi is in its while loop, it will enter the critical section (progress) after at most one entry by Pj (bounded waiting)
- However, this two process software solution to critical section problem does not guarantee to work on modern computer architectures.
- A general solution requires a simple tool called Lock. Race conditions are prevented by requiring that critical regions are protected by locks.

Solution to Critical Section problem using Locks

- Several hardware and software based APIs solutions exists to critical section problem; all are based on the premise of locking. The design of such locks can be quite sophisticated

```
do {
        Acquire Lock
            Critical Section
        Release Lock
            Remainder Section
} while (TRUE);
```


Synchronization Hardware

- In a uniprocessor environment, it is easy to solve the critical section problem; disable the interrupt when any process is accessing the shared object. This ensures correct sequence of the execution of the instructions, without preemption.
- However, this solution is not feasible in a multiprocessor environment. The message to disable the interrupts needs to be send to all the processors. This may delay the process from accessing the shared object, till all processors disable their interrupts, reducing the efficiency considerably.

Hardware Instruction: Test and Set

- Test and modify the content of a word atomically, supported by the hardware. Even if two such instructions are executed simultaneously, each on a different CPU, they will be executed sequentially in some arbitrary order, ensuring mutual exclusion. The definition of this instruction is as follows:
- **boolean TestAndSet(boolean &target)**

```
{    boolean rv = *target;
  *target = true;
  return rv;
}
```


Hardware Instruction: Test and Set

Mutual-exclusion implementation

Shared data:

boolean lock = false;

Process P_i

```
do {
        while (TestAndSet(lock)); // do nothing
          //critical section
        lock = false;
            //remainder section
    } while (True);
```

Hardware Instruction: Test and Set

- Here, the shared variable is lock which is initialized to false. TestAndSet(lock) algorithm works in this way – 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. 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

Hardware Instruction: Swap

```
void Swap(boolean &a, boolean &b)
    {
      boolean temp = a;
      a = b;
      b = temp;
    }
```

- A global variable lock is declared and is initialized to false. In addition, each process also has a local Boolean variable key.

Hardware Instruction: Swap

Mutual Exclusion Implementation

Process P_i

```
do {
   key = true;
   while (key == true)
            Swap(lock, key);
      critical section
   lock = false;
      remainder section
}
```


Hardware Instruction: Swap

- Instead of directly setting lock to true in the swap function, key is set to true and then swapped with lock. First process will be executed, and in while(key), since key=true , swap will take place and hence lock=true and key=false. Again next iteration takes place while(key) but key=false , so while loop breaks and first process will enter in critical section. Now another process will try to enter in Critical section, so again key=true and hence while(key) loop will run and swap takes place so, lock=true and key=true (since lock=true in first process). Again on next iteration while(key) is true so this will keep on executing and another process will not be able to enter in critical section. Therefore Mutual exclusion is ensured. Again, out of the critical section, lock is changed to false, so any process finding it gets to enter the critical section. Progress is ensured.

Semaphores

- The hardware based solutions are difficult for the application programmers to use; to overcome this difficulty, we use a synchronization tool called Semaphores.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
- The wait operation was originally termed **P** (from the Dutch **proberen**, "to test) and signal was called **V** (from **verhogen**, "to increment").

Semaphores

- The definition of wait() and signal():

```
wait (S)
{       while S≤ 0
                ; // no-op
          S-- ;
 }
signal (S):
{       S++;
 }
```

- All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly.

Semaphores

- Semaphores can be used to solve various synchronization problems.
- For example, consider a scenario where two concurrently running processes; P1 with a statement S1 and P2 with a statement S2 require that S2 be executed only after S1 has completed. This can be achieved with the two processes sharing a semaphore variable say synch initialized to 0 and the following statements inserted in the code of P1 and P2

P1	P2
-----	----
S1;	wait(synch)
signal(synch);	S2;

Types of Semaphores

- **Binary Semaphores:** The value of a binary semaphore can be only 0 or 1. They are used to deal with the critical section problem, i.e. to control access of a single instance of a resource between multiple processes. The n processes share a semaphore, say mutex, initialized to 1. each process P_i has code skeleton as follows:

```
do {    wait(mutex);
                // critical section
        signal(mutex);
                // remainder section
  } while(true);
```

Types of Semaphores

- **Counting Semaphores:** The value of a counting semaphores can range over a finite integer value. They are used to control access to a given resource with finite instances. The semaphore is initialized to the number of resources available. Each process wishing to use the resource performs a wait() operation on the semaphore, thereby decrementing the count. When a process release the resource, it performs a signal() operation, incrementing the count. When the count for the semaphore goes to 0, all resources are being used. After that, processes wishing to use the resource will block until the count becomes greater than 0.

Practical Considerations

- **Busy Waiting**

The main disadvantage of the standard definition of semaphore (wait() and signal())is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. In a single CPU environment, busy waiting wastes CPU cycles that some other process might be able to use productively.

Practical Considerations

- **Starvation**

A process that is waiting is inserted into a FIFO queue; the process that enters the queue first is the one that enters the critical section after every signal(). However, instead of being processed in first in first out manner, if the processes are processed in last in first out manner, it may result in starvation of processes.

Practical Considerations

- **Deadlock**

Consider a system consisting of two processes P0 and P1, each accessing two semaphores, S and Q, set to the value 1. following code will lead to a deadlock:

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

Practical Considerations

- **Priority Inversion**

A scheduling challenge arises when a lower priority process say P inside the critical section makes a higher priority process say R to wait. The scenario becomes more complicated if now another process Q having lower priority than R but higher than P pre-empt process P. Indirectly, process Q with a lower priority has affected how long process R must wait for P to relinquish the shared resource and come out of critical section.

Practical Considerations

- Priority inversion problem is solved using a Priority inheritance protocol. According to this protocol, all the processes that are accessing resources needed by a higher-priority process, inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values.
- For the above example, this would means that process P will temporarily inherit the priority of process R. Once finished, it will revert back to the original priority. Now process R, and not Q, will run next.

Classic Problems of Synchronization

- In this course, we will discuss three synchronization problems as examples of a large class of concurrency-control problems; with semaphores as the synchronization tool.
- These problems are used for testing every newly proposed synchronization scheme.
- These problems are
  - The Producer-Consumer problem
  - The Readers-Writers problem
  - The Dining –Philosophers problem

The Producer-Consumer Problem

- This is a common paradigm among cooperating processes.
- A Producer process produces information that is consumed by a Consumer process.
- For example, a complier may produce assemble code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.
- A server may be thought of as a producer and a client as a consumer. For e.g. a Web server produces (provides) HTML files which is consumed (read) by the client web browser.

The Producer-Consumer Problem

- To allow producer and consumer processes to run concurrently, we must have a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by both the processes.
- The two processes must be synchronized; consumer cannot consume an item that is not yet produced !!!!
- We can consider two types of buffer; unbounded buffer and bounded buffer.

The Producer-Consumer Problem

- The unbounded buffer places no practical limit on the size of the buffer; producer can produce any amount of items, it does not have to wait for the item to be consumed or buffer to have empty place to store new items. However, consumer has to wait for the item to be produced before it can be consumed.
- The bounded buffer assumes fixed size of buffer; producer must wait if the buffer is full and consumer must wait if the buffer is empty.

The Bounded Buffer Problem

- Consider buffer with size n. The semaphore mutex provides mutual exclusion for access to the buffer and is initialized to 1.
- The semaphores empty and full count the number of empty and full buffer slots. The semaphore empty is initialized to n and semaphore full is initialized to 0.
- Shared data

semaphore: full, empty, mutex;

Initially:

full = 0, empty = n, mutex = 1

Producer Process

```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);
        …
    add nextp to buffer
        …
    signal(mutex);
    signal(full);
} while (true);
```

Consumer Process

```
do {
        wait(full)
        wait(mutex);
          …
        remove an item from buffer to nextc
          …
        signal(mutex);
        signal(empty);
          …
        consume the item in nextc
          …
    } while (true);
```


The Readers-Writers Problem

- Suppose a data base is to be shared among several concurrent processes. Some of them may want only to read the database, whereas, others may want to update(read and write) the database. These two types of processes are distinguished by referring the former as Readers and the later as Writers.
- While a writer must have an exclusive access to the database, more than one reader can access the database.

The Readers-Writers Problem

- There are many variations to this problem, all involving priorities.
- The first Reader-Writers problem requires that no reader should wait for other readers to finish simply because a writer is waiting.
- The second Reader-Writers problem requires that if a writer is waiting to access the shared object, no new readers may start reading.
- Either problem may result in starvation; in the first case, writers may starve, in the second case readers may starve.
- For this reason, other variations of the problem have also been proposed.

The First Readers-Writers Problem

- Shared data
  semaphore mutex, wrt;
  int readcount;
  Initially
  mutex = 1, wrt = 1, readcount = 0;
- The variable readcount keeps track of how many processes are currently reading the shared object.
- The semaphore wrt is common to both reader and writer processes. It is used as the mutual exclusion semaphore for the writers. It is also used by the first reader who enters the critical section and the last reader who exits.
- The semaphore mutex is used to ensure mutual exclusion when the variable readcount is updated.

The First Readers-Writers Problem

- Writer Process

```
do {
        wait(wrt);
            …
        writing is performed
            …
        signal(wrt);
} while(true);
```


The First Readers-Writers Problem

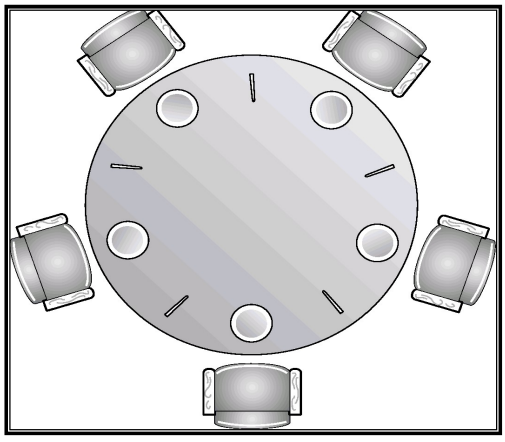
- Reader Process

```
do{     wait(mutex);
        readcount++;
        if (readcount == 1)
            wait(wrt);
        signal(mutex);
            …
        reading is performed
            …
        wait(mutex);
        readcount--;
        if (readcount == 0)
            signal(wrt);
        signal(mutex):
} while(true);
```

The Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, he/she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks to his/her immediate left and right; if successful he/she eats, when finished puts the chopsticks down and starts thinking again.

The Dining-Philosophers Problem



Solution using Semaphores

Philosopher *i*:

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


Solution using Semaphores

- Although, this solution ensures mutual exclusion, it may lead to deadlock.
- A deadlock free solution does not necessarily eliminate the possibility of starvation.

Problems with semaphore as a synchronization tool

- Problems that may be caused by programming error or uncooperative programmer
  - Reversing the sequence of wait() and signal() operations may violate mutual exclusion.
  - Two successful wait() on same semaphore causes deadlock
  - Omission of either wait() or signal(), or both may either violate mutual exclusion or cause deadlock