Process Synchronization

1 Introduction

Process Synchronization is a technique used to coordinate the process that use shared data. Concurrent access to shared data may result in data inconsistency.

- **Independent process**: It doesn't affect and isn't affected by other processes during execution.
- **Cooperative process**: It can affect or be affected by other processes executing in a system.

Race Condition is a condition where several processes tries to access the resources and modify the shared data concurrently and outcome of the process depends on the particular order of execution that leads to data inconsistency.

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

2 Critical Section Problem

Each process has a segment of code, **critical section**, in which it may be changing shared data. It is group of instructions that need to be executed atomically.

Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

Solution Criteria

- **Mutual exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress:** If some processes wish to enter their critical sections when no process is executing in its critical section, then only processes not executing in their remainder sections can participate in deciding which will enter its critical section next, and selection can't be postponed indefinitely.
- **Bounded waiting:** There exists a bound 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.

Two approaches are used to handle critical sections in operating systems. A **preemptive kernel** allows a process to be preempted while it is running in kernel mode. A **non-preemptive kernel** is free from race conditions on kernel data structures as only one process is active in the kernel at a time.

A preemptive kernel is suitable for real-time programming, as it allows real-time process to preempt a process running in kernel. It is also more **responsive**.

3 Peterson's Solution

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. To enter the critical section, process P_i first sets flag[i] to true and then sets turn to the value j asserting that if the P_j wishes to enter the critical section, it can do so.

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. It also involves **busy-waiting**.

4 Synchronization Hardware

In a uni-processor environment, if **interrupts are prevented** while a shared variable was being modified, it can be ensured that current sequence of instructions would be allowed to execute in order without preemption. This is often done by non-preemptive kernels.

Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

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

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

Race conditions are prevented by requiring that critical regions be protected by **locks**. Hardware features can make any programming task easier and improve system efficiency.

Computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**.

```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Although these algorithms satisfy the mutual-exclusion and progress requirement, they do not satisfy the bounded-waiting requirement. The next algorithm satisfies all requirements.

```
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);
```

5 Semaphores

To overcome complications in design of hardware=based solutions,a synchronization tool called a **semaphore** is used. A Semaphore is an integer variable, which can be accessed only through two operations *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**. That is, two processes cannot simultaneously modify that same semaphore value. Additionally, the testing and modification of its value must be executed without interruption.

- **Binary semaphore:** They can only be either 0 or 1. They are also known as **mutex locks**, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1.
- **Counting semaphore:** Their value is not restricted over a certain domain. They can be used to control **access to a resource** that has a limitation on the number of simultaneous accesses. Its value is initialized to the number of instances of the resource.

This type of semaphore is also called a **spinlock** because the process "spins" while busy-waiting for the lock. Spinlocks have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.

Implementation

When a process executes the *wait()* operation and finds its value is not positive, the process can block itself which places it into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation by a *wakeup()* operation, which changes the process from the waiting state to the ready state and is placed in the ready queue.

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

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

If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. To ensure bounded-waiting, FIFO queue can be used.

It is vital that semaphores be executed atomically. Interrupts is feasible for single-processor environments. In a multiprocessor environment, interrupts must be disabled on every processor; otherwise, instructions from different processes may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance.

Deadlock and Starvation

The implementation of a semaphore with a waiting queue may result in a **deadlock** where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. **Indefinite blocking** may occur if we remove processes from the list associated with a semaphore in LIFO order.

Priority Inversion

When multiple middle-priority processes preempt a low-priority process, preventing a high-priority process from attaining a resource, it is called priority inversion. Indirectly, a process with low-priority P_M has affected how long process P_H must wait for a P_L process to relinquish a resource.

According to **priority-inheritance protocol**, all 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.

6 Classic Problems of Synchronization

Bounded-Buffer Problem

We assume that the pool consists of n buffers, each capable of holding one item. The *mutex* semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The *empty* and *full* semaphores count the number of *empty* and *full* buffers. The semaphore *empty* is initialized to the value n; the semaphore *full* is initialized to the value 0.

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

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

Reader-Writers Problem

Suppose several concurrent processes (**readers**) want only to read, whereas others (**writers**) want to update the database. If a writer and some other process access the database simultaneously,chaos may ensue. To prevent this, we require that the writers have exclusive access to the shared database while writing. This synchronization problem is the **readers–writers problem**.

```
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);
```

```
do {
  wait(wrt);
    . . .
  // writing is performed
    . . .
  signal(wrt);
} while (TRUE);
```

Dining-Philosophers Problem

Consider 5 philosophers who share a circular table with a rice bowl and 5 chopsticks surrounded by 5 chairs. From time to time, a philosopher gets hungry and tries to pick up the 2 chopsticks that are closest to her, only one at a time. When a hungry philosopher has both her chopsticks at the same time, he eats without releasing her chopsticks. After eating, he puts down both of his chopsticks and starts thinking again. The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.