BCA233: Operating System

Unit – 3 (Part-1)

Chapter 5: Process Synchronization

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Synchronization Examples

Background

- 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.

Producer

```
while (true) {
   /* produce an item in next produced 
*/

   while (counter == BUFFER_SIZE) ;
      /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer

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

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

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

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

- 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.**

Critical Section Problem

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

Critical Section

- General structure of process P_i

```
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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- a classic software-based solution to the critical-section problem
- Good algorithmic description of solving the problem
- 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 $\mathbf{P_i}$ is ready

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);
       critical section
    flag[i] = false;

       remainder section

 } while (true);
```

Peterson’s Solution (Cont.)

Example:

- bool flag[0] = {false};
- bool flag[1] = {false};
- int turn;

```
P0: flag[0] = true;
P0_gate:  turn = 1;
          while (flag[1] == true && turn == 1)
          {
             // busy wait
          }
             // critical section
                    ...
            // end of critical section

            flag[0] = false;
```

Peterson’s Solution (Cont.)

```
P1: flag[1] = true;
P1_gate:  turn = 0;
          while (flag[0] == true && turn == 0)
          {
             // busy wait
          }
              // critical section
                     ...
             // end of critical section

             flag[1] = false;
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
    1. Mutual exclusion is preserved
       P_i enters CS only if:
         either **flag[j] = false** or **turn = i**
    2. Progress requirement is satisfied
    3. Bounded-waiting requirement is met

Semaphore

- Synchronization tool that provides more sophisticated ways for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - Originally called **P( )** and **V( )**
- Definition of the **wait() operation**

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

- Definition of the **signal() operation**

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore**– integer value can range only between 0 and 1
- Same as a **mutex lock**
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a 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 that wish to use a resource will block until the count becomes greater than 0.

Semaphore Usage

- We can use semaphores to solve various synchronization problems.
- For example, consider two concurrently running processes: ***P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only* after *S1 has completed. We can implement this scheme readily by letting P1* and *P2 share a common semaphore synch, initialized to 0.***

  In process P1, we insert the statements

  ```
  S1;
    signal(synch);
  ```

  In process P2, we insert the statements

  ```
  wait(synch);
    S2;
  ```

- Because synch is initialized to 0, *P2 will execute S2 only after P1 has invoked* signal(synch), which is after statement *S1 has been executed.*

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the 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
- **typedef struct{**

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

Implementation with no Busy waiting (Cont.)

```
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);
   }
}
```

Deadlock and Starvation

- **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** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**