PROCESS SYNCHRONIZATION

Operating System (CSC1036 & INF1036)

Dr. Hasin A. Ahmed
Assistant Professor
Department of Computer Science
Gauhati University

PROCESS TYPES

- Processes that are executing concurrently in the operating system may be either independent processes or cooperating processes.
- If a process cannot affect or be affected by the other processes executing in the system then the process is said to be independent.
- So any process that does not share any data with any other process is independent.
- A process is said to be cooperating if it can affect or be affected by the other processes executing in the system.
- So it is clear that, any process which shares its data with other processes is a cooperating process.

PROCESS SYNCHRONIZATION

- *It is the task of coordinating the execution of processes in such a way that no two cooperating processes can have access to the same shared data and resources in a way that no inconsistencies arise*

Producer Consumer Problem

- The Producer-Consumer problem is a classic synchronization problem in operating systems.
- The problem is defined as follows: there is a fixed-size buffer and a Producer process, and a Consumer process.
- The Producer process creates an item and adds it to the shared buffer.
- The Consumer process takes items out of the shared buffer and “consumes” them.

PRODUCER CONSUMER PROBLEM

- Producer

```
while (true)
{
while(counter==BUFFER_SIZE);
buffer[in]=nextProduced;
in=(in+1)%BUFFER_SIZE;
counter++;
}
```

- Consumer

```
while(true)
{
while(counter==0);
nextConsumed=buffer[out];
out=(out+1)%BUFFER_SIZE;
counter--;
}
```

PRODUCER CONSUMER PROBLEM

- Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.
- Suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter= 5, which is generated correctly if the producer and consumer execute separately.

PRODUCER CONSUMER PROBLEM

- counter++
  - register1=counter
  - register1=register1+1
  - counter=register1
- counter--
  - register2=counter
  - register2=register2-1
  - counter=register2

- T0: Producer: register1=counter
- T1: Producer: register1=register1+1
- T2: Consumer: register2=counter
- T3: Consumer: register2=register2-1
- T4: Producer: counter=register1
- T5: Consumer: counter=register2

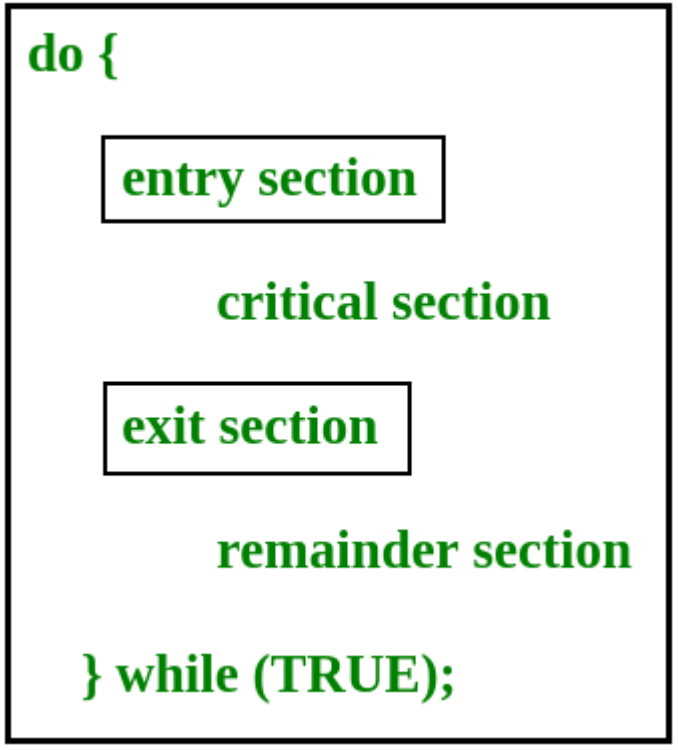
PRODUCER CONSUMER PROBLEM

- If we reverse the order of the statements at T4 and T5, we would arrive at the incorrect state "counter = 6"
- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- 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 above, 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 in some way.

CRITICAL SECTION PROBLEM

- Consider a system with some processes.
- Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- When one process is executing in its critical section, no other process should be allowed to execute in its critical section.
- The critical-section problem is to design a protocol that the processes can use to cooperate.
- 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**.

CRITICAL SECTION PROBLEM



CRITICAL SECTION PROBLEM

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

1) Mutual exclusion. If process Pi 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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3) Bounded waiting. There exists a bound, or limit, 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

- Peterson's solution is a classic software-based solution to the critical-section problem
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P0 and P1.
- For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1 – i.
- Peterson's solution requires two data items to be shared between the two processes:
  - int turn
  - boolean flag[2]

Peterson's Solution

- The variable turn indicates whose turn it is to enter its critical section.
- That is, if turn == i, then process Pi is allowed to execute in its critical section.
- The flag array is used to indicate if a process is ready to enter its critical section.
- For example, if flag [i] is true, this value indicates that Pi is ready to enter its critical section.

Peterson's Solution

```
do {

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

              critial section

        flag[i] = FALSE ;

              remainder section

    }  while (TRUE) ;
```

Peterson's Solution

```
1   // P0 için
2   flag[0] = 1;
3   turn = 1;
4   // bariyer kodlaması
5   while (flag[1] == 1 && turn == 1);
6   // kritik alan
7   // www.bilgisayarkavramlari.com
8   // critical section
9       ...
10  flag[0] = 0;
```

```
1   // P1 için
2   flag[1] = 1;
3   turn = 0;
4   // bariyer kodlaması
5   while (flag[0] == 1 && turn == 0);
6   // kritik alan
7   // www.bilgisayarkavramlari.com
8   // critical section
9       ...
10  flag[1] = 0;
```

Peterson's Solution

- To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

Peterson's Solution

- Does it satisfy mutual exclusion?
- We note that each Pi enters its critical section only if either flag[j] == false or turn == i.
- Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] == true.
- These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.
- Does it satisfy progress condition?
- After passing the exit section, a process Pi makes flag[i] false and it has no influence on execution of Pj in its critical section.

Peterson's Solution

- Does it satisfy bounded waiting?
- If Pj resets flag [j] to true, it must also set turn to i.
- Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section after at most one entry by Pj (bounded waiting).

Bakery Algorithm

- Lamport proposed a bakery algorithm, a software solution, for n processes.
- This algorithm solves a critical problem, following the fairest, first come, first serve principle.
- This algorithm is known as the bakery algorithm as this type of scheduling is adopted in bakeries where token numbers are issued to set the order of customers.
- Each process waiting to enter its critical section gets a token number, and the process with the lowest number enters the critical section.
- If two processes have the same token number, the process with a lower process ID enters its critical section.

```
do
{
   entering[i] :=  true; // show interest in critical section
  // get a token number
   number[i] := 1 + max(number[0],  number[1], ..., number[n - 1]);
   entering [i] :=  false;
   for ( j :=  0 ;  j<n; j++)
   {
 // busy wait until process Pj receives its token number
       while (entering [j])
       { ; } // do nothing
      while (number[j] !=  0 &&  (number[j], j) < (number[i], i)) // token comparison
        { ; } // do nothing
      }
       // critical section
        number[i] :=  0;  // Exit section
       // remainder section
   } while(1);
```

Bakery Algorithm

- All entering variables are initialized to false, and all indices are all initialized to 0.
- The value of number variable is used to form token numbers.
- When a process wishes to enter a critical section, it chooses a greater token number than any earlier number.
- Consider a process Pi wishes to enter a critical section, it sets entering[i] to true to make other processes aware that it is choosing a token number.
- It then chooses a token number greater than those held by other processes and writes its token number.
- Then it sets entering[i] to false after reading them.

Bakery Algorithm

- Then it enters a loop to evaluate the status of other processes.
- It waits until some other process Pj is choosing its token number.
- Pi then waits until all processes with smaller token numbers or the same token number but with higher priority are served first.
- Bakery algorithm fulfills mutual exclusion, progress and bounded waiting

Semaphore

- Though some hardware instructions specially used for synchronization can be used, but such instructions are complicated for the application programmers to use
- To overcome this difficulty, we can use a synchronization tool called a semaphore.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
- Semaphore cannot be implemented in the user mode because race condition may always arise when two or more processes try to access the variable simultaneously. It always needs support from the operating system to be implemented.

Semaphore

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

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

Counting and Binary Semaphore

- The value of a counting semaphore can range over an unrestricted domain.
- The value of a binary semaphore can range only between 0 and 1.
- On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.
- We can use binary semaphores to deal with the critical-section problem for multiple processes.
- The n processes share a semaphore, mutex, initialized to 1

Binary Semaphore

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

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

Counting Semaphore

- 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.

Binary Semaphore

- We can also 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, and by inserting following statements in P1 and P2, respectively.

```
S1;

signal(sync);
```

```
wait(sync);

S2;
```

Binary Semaphore

- 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

```
S1;

signal(sync);
```

```
wait(sync);

S2;
```

Producer Consumer Problem and Semaphore

- To solve this problem, We employ three semaphore variables:
  - mutex - The lock is acquired and released using a mutex, a binary semaphore.
  - empty - empty is a counting semaphore that is initialized on the basis of the number of slots present in the buffer, at first all the slots are empty.
  - full - a counting semaphore with a value of zero as its starting value
- At any particular time, the current value of empty denotes the number of vacant slots in the buffer, while full denotes the number of occupied slots.

Producer Consumer Problem and Semaphore

- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1
- The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
- Semaphores
  - mutex: binary
  - empty: counting
  - full: counting

Producer

```
do
{
    // process will wait until the empty > 0 and further decrement of 'empty'
    wait(empty);
    // To acquire the lock
    wait(mutex);

    /* Here we will perform the insert operation in a particular slot */

    // To release the lock
    signal(mutex);
    // increment of 'full'
    signal(full);
}
while(TRUE)
```

Consumer

```
do
{
    // need to wait until full > 0 and then decrement the 'full'
    wait(full);
    // To acquire the lock
    wait(mutex);

     /* Here we will perform the remove operation in a particular slot */

    // To release the lock
    signal(mutex);
    // increment of 'empty'
    signal(empty);
}
while(TRUE);
```


thank you

? ? ?