Lesson 13

Objectives

- Background
- The Critical-Section Problem
- Conditions for solution of critical section problem
- Algorithms for solution two process

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem (Chapter 4) allows at most n – 1 items in buffer at the same time. A solution, where all N buffers are used is not simple.
- Suppose that we modify the producer-consumer code by adding a variable counter, initialized to 0 and incremented each time a new item is added to the buffer, and decremented each time an item deleted from buffer.
- So the statements
  Counter++;Counter--; must be performed atomically. Atomic operation means an operation that completes in its entirety without interruption.
- The statement "count++" may be implemented in machine language as:

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

- The statement "count—" may be implemented as:

```
register2 = counter
register2 = register2 – 1
counter = register2
```

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.
- Assume counter is initially 5. One interleaving of statements is:

```
producer: register1 = counter          (register1 = 5)
producer: register1 = register1 + 1    (register1 = 6)
consumer: register2 = counter          (register2 = 5)
consumer: register2 = register2 – 1    (register2 = 4)
producer: counter = register1          (counter = 6)
consumer: counter = register2          (counter = 4)
```

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

Race condition: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last. To prevent race conditions, concurrent processes must be synchronized.

The Critical Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Conditions for Solution of Critical Section Problem

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 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.

Two Process Solution

Initial Approach

- Only 2 processes, P0 and P1
- General structure of process Pi (other process Pj)

```
do {
entry section
critical section
```

```
exit section
reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  int turn;
  initially turn = 0
  turn - i _ Pi can enter its critical section
- Process Pi

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

- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  boolean flag[2];
  initially flag [0] = flag [1] = false.
  flag [i] = true _ Pi ready to enter its critical section
- Process Pi

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

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process Pi

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

- Meets all three requirements; solves the critical-section problem for two processes.