BITS Pilani

K K Birla Goa Campus

Operating Systems

Dr. Lucy J. Gudino

Dept. of CS and IS

Process Synchronization

Last class

- Producer - Consumer
- 3 cases
- Two types of buffer: Un- Bounded and Bounded buffer
- Shared memory using bounded buffer

Producer Consumer

```
while (true) {
       int nextProduced;
        /*   produce an item and put in nextProduced  */
        while ( ( (in+1) % BUFFER_SIZE ) == out)
                   ; // do nothing and wait
               buffer [in] = nextProduced;
               in = (in + 1) % BUFFER_SIZE;
}
```

```
while (true) {
          int nextConsumed;
           while (in == out)
                  ; // do nothing, but wait
                  nextConsumed =  buffer[out];
                   out = (out + 1) % BUFFER_SIZE;
                  /*   consume the item in nextConsumed
    }
```

Contd…

- Main flaw: one location has to be empty
- use counter to eliminate above problem
- Initially counter is set to zero
  - increment the counter whenever producer produces an item
  - decrement the counter whenever consumer consumes an item
  - When counter = 0 ➔ buffer is empty
  - When counter = BUFFER_SIZE ➔ buffer is full

Producer - Consumer

innovate achieve lead

The producer process

```
while (true) {
      int nextProduced;
       /*  produce an item and put in nextProduced  */
       while (count == BUFFER_SIZE)
               ; // do nothing
           buffer [in] = nextProduced;
           in = (in + 1) % BUFFER_SIZE;
           count++;
}
```

Consumer Process

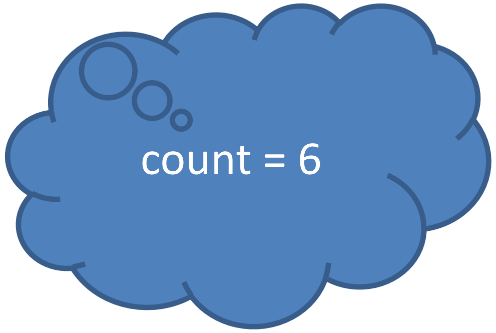
```
while (true) {
        int nextConsumed;
       while (count == 0)
            ; // do nothing
           nextConsumed =  buffer[out];
            out = (out + 1) % BUFFER_SIZE;
             count--;
             /*  consume the item in nextConsumed
    }
```

Contd…

Implementation of counter

Producer Process	Consumer Process
A : R1 = count	D : R2 = count
B: R1 = R1+1	E: R2 = R2-1
C: count = R1	F: count = R2

A
B
D
E
C
F



A : R1 = 6
B : R1 = 7
D : R2 = 6
E : R2 = 5
C : count = 7
F : count = 5

Race Condition

- 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.**

The Critical Section Problem

- Consider a system with n processes {P_0, P_1, ….P_{N-1}} competing to use some shared data
- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.
- Ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.
- design a protocol that the processes can use to cooperate
- Process needs to request before trying to execute code in CS
- General structure of the process contains three sections
  - Entry section
  - Leave or Exit section
  - Remainder section

General structure of a typical process P_i

```
do{
        Entry section
            critical section
        Exit section
            remainder section
} while (TRUE);
```

Solution to Critical-Section Problem

- 3 requirements
  1. Mutual Exclusion
  2. Progress
  3. Bounded Waiting

1. Mutual Exclusion

- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

  Implications:

  - Critical sections better be focused and short.
  - Better not get into an infinite loop in there.
  - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

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 process that will enter the critical section next cannot be postponed indefinitely:
  - If only one process wants to enter, it should be able to.
  - If two or more want to enter, one of them should succeed.
  - Decision should be taken in finite time

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.

Critical Section Solution1

- Consider two processes P0 and P1.
- Processes share a common variable turn(=0 or 1).

Process P_i

```
while (1) {
    While (turn != i)
          ; // wait
    Critical section
    turn = j;
    Remainder section
}
```

Critical Section Solution1

Process P0: i = 0, j =1

```
while (1) {
    While (turn != i)
          ; // wait
    Critical section
    turn = j;
    Remainder section
}
```

Process P1: i = 1, j =0

```
while (1) {
    While (turn != i)
          ; // wait
    Critical section
    turn = j;
    Remainder section
}
```

Mutual Exclusion ✓

Progress X

Bounded wait X

Critical Section Solution 2

- Consider two processes P0 and P1.
- Shared variables int turn; boolean flag[2];
  - flag[i] is true when Pi is ready to enter its critical section

Process P0: i = 0, j =1

```
while (1) {
    flag[i] = TRUE; turn = j;
    while (flag[j] && turn == j)
            ;// wait
    Critical section
    flag[i] = FALSE;
    Remainder section
}
```

Process P1: i = 1, j =0

```
while (1) {
    flag[i] = TRUE; turn = j;
    while (flag[j] && turn == j)
            ;// wait
    Critical section
    flag[i] = FALSE;
    Remainder section
}
```

Mutual Exclusion
Progress
Bounded wait

Also known as Peterson's Solution

Example

Develop a solution for CS with n processes