Process Synchronization

By
Dr. Upasana Pandey
Department of Computer Science & Engineering
IMS Engineering College (College Code:143)

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

P()	Case 1: P1 P2 (Sequentially access)	Case 2: P1 P2 (interleaving access)

{
Read (a)
a=a+1
Write (a)
}
	11 12	10
		11
		11

In case 2, data has been inconsistent.

Producer

- 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.

```
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}

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);
```

Criteria 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

Using turn variable two process solution for critical section

Boolean int turn=0;

P0	P1

while (1)
{
while (turn !=0);
Critical Section
turn=1;
Remainder section;
}
	while (1)
	{
	while (turn !=1);
	Critical Section
	turn=0;
	Remainder section;
	}

Outcome:
Mutual Exclusion is satisfied but Progress criteria is not satisfied. Therefore it is not consistent solution.

Using flag variable two process solution for critical section

Boolean int flag[2];
flag [0]=false;
flag [1]=false;

P0	P1

1. while (1)
2. {
3. flag[0]=true;
4. while (flag[1]);
5. Critical Section
6. flag[0]=false;
}
	1. while (1)
	2. {
	3. flag[1]=true;
	4. while (flag[0]);
	5. Critical Section
	6. flag[1]=false;
	}

Outcome:
Mutual Exclusion and Progress criteria, both are satisfied. But if P0 executed till line 3 and context switch occurs for P1, P1 executed till line 3 and check the condition which is false then cannot enter into critical section. Same is happening with P0 also. In this situation no process can enter into critical section. Deadlock occurred.

Thank You