Revision
- Last lecture

Abhik Roychoudhury
CS 3211
National University of Singapore

Summary of previous 12 lectures
- Concurrency
  - As a concept.
  - Concurrent program execution – inter-leavings
  - Critical section and ensuring mutual exclusion
    - Semaphores, Monitors
  - Deadlocks, Starvation and preventing them.
- Concurrent programming
  - All of the above concepts as evidenced in multi-threaded Java
- Parallel programming
  - Message passing model studied via MPI

In today’s discussion
- Revision
  - Promela – concurrency concepts
  - Java – concurrent programming
  - MPI – parallel programming

Comment on the following protocol
```c
bool wantP = false, wantQ = false;

active proctype P() {
  do
    printf("noncritical section\n");
    wantP = true;
  od;
  printf("Crit. Section P\n");
  wantP = false
}

active proctype Q() {
  do
    printf("noncritical section\n");
    wantP = true;
  od;
  printf("Crit. Section Q\n");
  wantQ = false
}
```

Write process equation for:
```c
P = ((three -> lose) | ((one | two) -> win)) -> P
```

OR
```c
P = three -> Q | one -> R | two -> R
Q = lose -> P
R = win -> P
```
Concurrent Executions (from textbook)

A roller coaster control system only permits its car to depart when it is full. Passengers arriving at the departure platform are registered with the roller-coaster controller by a turnstile. The controller signals the car to depart when there are enough passengers on the platform (to fill the car to its capacity of $M$). The car goes round the roller-coaster track and waits for another $M$ passengers. A maximum of $M$ passengers can occupy the platform. Model three processes TURNSTILE, CONTROL, CAR. TURNSTILE and CONTROL interact via the arrival of a passenger. CONTROL and CAR interact via the departure of a car.

Answer:

1. **TURNSTILE** = (passenger -> TURNSTILE).
2. **CONTROL** = CONTROL[0],
   - when (i<M) passenger -> CONTROL[i+1]
   - when (i==M) depart -> CONTROL[0]
3. **CAR** = (depart -> CAR).
4. **ROLLERCOASTER** = (TURNSTILE || CONTROL || CAR).

Monitors – Dining Philosophers

Consider the following schematic code for the Dining Philosophers’ problem discussed in class.

Recall that

- `wait_on_cond(Cond)`
  - append $p$, the current process to queue for Cond
  - $p$.state = blocked
  - monitor.lock = released
- `signal_to_cond(Cond)`
  - if queue for Cond != empty{
    - remove head of queue, let it be process $x$; $x$.state = ready
  }

Monitor – Dining Philosophers

```c
monitor Fork{
    int array[0..4] fork = [2,2,2,2,2]
    condition array[0..4] OKtoEat
    operation takeForks(int i){
        if (fork[i] != 2){
            wait_on_cond(OKtoEat[i])
            fork[i] = fork[i+1] - 1;
            fork[i-1] = fork[i-1] – 1;
            if (fork[i+1] == 2){
                signal_on_cond(OKtoEat[i+1])
            }
        }
    }
    operation releaseForks(int i){
        fork[i+1] = fork[i+1] + 1;
        fork[i-1] = fork[i-1] + 1;
        if (fork[i+1] == 2){
            signal_on_cond(OKtoEat[i+1])
        }
    }
}
```

Philosopher $i$’s code

```c
loop forever{  takeForks($i$);  EAT;  releaseForks($i$); }
```

Questions

- Explain the working of the code.
- Does the code suffer from deadlocks?
- Does it suffer from starvation?
- Can you show any of the following
  1. $\text{eating}(i) \Rightarrow (\text{fork}[i] == 2)$
  2. $\text{eating}(i)$ is true when philosopher $i$ has executed takeForks(), and has not yet executed releaseForks().
  3. $\neg \text{empty}(\text{OKtoEat}(i)) \Rightarrow (\text{fork}[i] < 2)$
  4. $\sum_{i=0}^{4} \text{fork}[i] = 10 - 2 \times E$, where $E$ = # of phil. who are eating

No deadlock

- Deadlock implies $E = 0$
- Then $\text{fork}[0] + \text{fork}[1] + \text{fork}[2] + \text{fork}[3] + \text{fork}[4] = 10$
- Also, in a deadlock all philosophers should be enqueued on $\text{OKtoEat}$.
- Thus, for all $i$, $\text{fork}[i] < 2$
  - Hence $\text{fork}[0] + \text{fork}[1] + \text{fork}[2] + \text{fork}[3] + \text{fork}[4] < 10$
  - Contradiction!
Starvation scenario

phil1           phil2                phil3

- take(1)
- wait(OK[2])
- release(1)
- take(1)
- release(3) forever

---

Exercise on Parallel Programming

```c
int x, y, z; /* MPI_COMM_WORLD = {0,1,2} */
switch (rank) {
    case 0:  x = 0; y = 1; z = 2;
             MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
             MPI_Send(&y, 1, MPI_INT, 2, 43, MPI_COMM_WORLD);
             MPI_Bcast(&z, 1, MPI_INT, 1, MPI_COMM_WORLD); break;
    case 1:  x = 3; y = 4; z = 5;
             MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
             MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD); break;
    case 2:  x = 6; y = 7; z = 8;
             MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);
             MPI_Recv(&x, 1, MPI_INT, 0, 43, MPI_COMM_WORLD, &status); break;
}
```

What are the values of x, y, z when the code terminates?

---

Run it in class, and see

- Rank x  y  z
  - 1  0  4  5
  - 2  1  4  0
  - 0  0  1  4

Explain the reason behind each of the 9 values!

---

Matrix-vector mult. in parallel

- In class, we discussed dot product computation where two vectors were multiplied. Now, consider the multiplication of a matrix with a vector.

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

1 * -1 = -1
0 + 3 * 0 = 0
2 + 4 * 4 = 18

---

How to divide up the data?

- We are performing $A \mathbf{b} = \mathbf{c}$
  - Assume that rows of the matrix are distributed into proc.
  - Vector $\mathbf{b}$ is replicated into all processes.

- Steps
  - Perform local sum (row i of A) * $\mathbf{b}$ = element i of $\mathbf{c}$
  - Allgather MPI communication to gather all elements of $\mathbf{c}$.

---

Pictorially