COMP 3511
Operating Systems

Lab 05

Outline

- Monitor
- Dining Philosopher

Monitors

- Motivation

  Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

- Definition

  A lock and zero or more condition variables for managing concurrent access to shared data

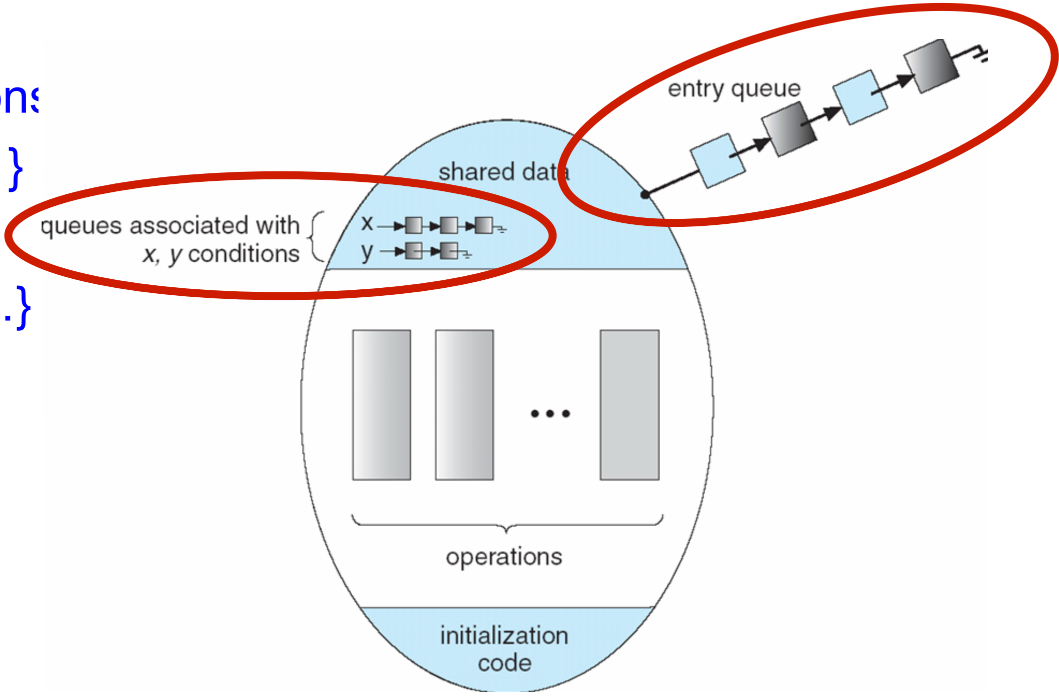
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization

- Only one process may be active within the monitor at a time

Monitors

```
monitor monitor-name
{
    // shared variable declarations
        procedure P1 (…) { …. }
            …
        procedure Pn (…) {……}

        Initialization code ( ….)
            { … }
            …
}
```



- Some languages like Java provide this natively
- Most others use actual locks and condition variables

Monitors

- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free

- Condition Variable: a queue of threads waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

- condition variables x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Difference between semaphore and condition

Semaphore	condition
count	don't count
wait: may pass immediately depending on the value of the count	wait: alway wait
signal: only increase semaphore, may wake up or may not wake up another process	signal: if there are process(es) waiting, wake up one. otherwise, nothing happens.

Monitor Implementation Using Semaphores

- Make full use of Semaphores (Queue)

Things need to handle	Simple Solution
1. Only one process is running	use mutex to protect
2. condition wait (it should stop and another	it should stop and another can run
3. condition signal	another might run, so it should stop

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

semaphore next is used to specify the entry queue, which is initialized to zero. Since the wait() is supposed to put the process on the entry queue immediately

- Each procedure ***F*** will be replaced by

```
wait(mutex);
   …
            body of F;

   …
if (next_count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x-count = 0;
```

semape x-sem is used to specify the queue associated with conditional variable

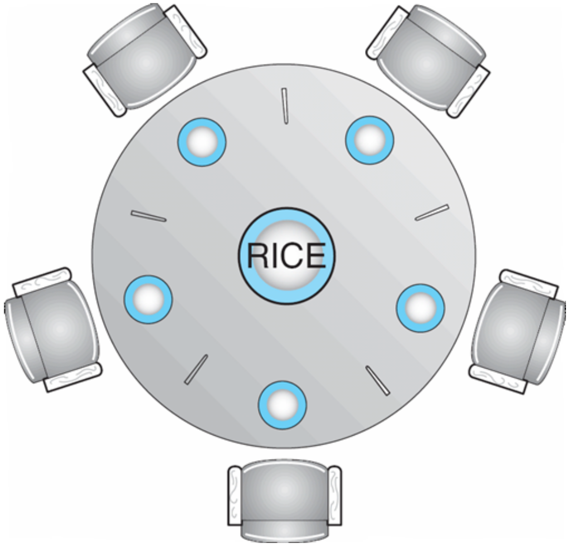
- The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x-count--;
```

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
}
```

Dining-Philosophers Problem



- Problem: each philosopher thinks and eats, and need two chopsticks while eating
  - To avoid the situation, holding one chopstick and waiting for another
  - Solution: either have two chopsticks or no chopstick
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Solution to Dining Philosophers

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

  DiningPhilosophters.pickup (i);

  EAT

  DiningPhilosophers.putdown (i);

Solution to Dining Philosophers

```
monitor DP
  {
     enum { THINKING; HUNGRY, EATING)
      state [5] ;
     condition self [5];

     void pickup (int i) {
            state[i] = HUNGRY;
            test(i);
            if (state[i] != EATING) self [i].wait;
     }

      void putdown (int i) {
            state[i] = THINKING;
                  // test left and right neighbors
             test((i + 4) % 5);
             test((i + 1) % 5);
      }

     void test (int i) {
            if ( (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                  state[i] = EATING ;
                  self[i].signal () ;
             }
      }

      initialization_code() {
             for (int i = 0; i < 5; i++)
             state[i] = THINKING;
     }
}
```

Q. 1

- A race condition ____.
- A) results when several threads try to access the same data concurrently
- B) results when several threads try to access and modify the same data concurrently
- C) will result only if the outcome of execution does not depend on the order in which instructions are executed
- D) None of the above

Q. 1

- A race condition ____.
- A) results when several threads try to access the same data concurrently
- B) results when several threads try to access and modify the same data concurrently
- C) will result only if the outcome of execution does not depend on the order in which instructions are executed
- D) None of the above
- Ans: B

Q. 2

- An instruction that executes atomically ____.
- A) must consist of only one machine instruction
- B) executes as a single, uninterruptible unit
- C) cannot be used to solve the critical section problem
- D) All of the above

Q. 2

- An instruction that executes atomically ____.
- A) must consist of only one machine instruction
- B) executes as a single, uninterruptible unit
- C) cannot be used to solve the critical section problem
- D) All of the above
- Ans: B

Q. 3

- In Peterson's solution, the ____ variable indicates if a process is ready to enter its critical section.
- A) turn
- B) lock
- C) flag[i]
- D) turn[i]

Q. 3

- In Peterson's solution, the _____ variable indicates if a process is ready to enter its critical section.
- A) turn
- B) lock
- C) flag[i]
- D) turn[i]
- Ans: C

Q. 4

- Please explain the Peterson’s solution satisfy all three conditions of the critical section solution
- This algorithm satisfies the three conditions of mutual exclusion.
- (1) **Mutual exclusion** is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn.
- (2) **Progress** is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It sets turn to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again—before the other process—it repeats the process of entering its critical section and setting turn to the other process upon exiting.

Q. 4

- (3) **Bounded waiting** is preserved through the use of the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered—and exited—its critical section. However, Peterson's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.