CSc 360
Operating Systems
Monitors

Jianping Pan

Summer 2006

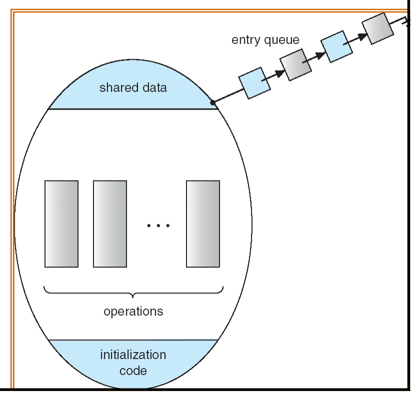
Review

- Hardware-assisted synchronization
- Semaphores
  - operations
    - wait(); signal();
  - problems with *using* semaphores
    - signal (mutex); … wait (mutex);
    - wait (mutex); … wait (mutex);
    - omitting of wait (mutex); or signal (mutex);
    - ordering of wait (A); and wait (B);

Monitor

- A high-level abstraction (OO design) that provides a convenient and effective mechanism for process synchronization
- Only ***one*** process may be active within the monitor at a time

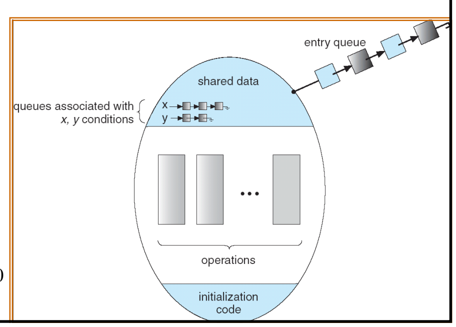
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }
        …
  procedure Pn (…) {……}
  Initialization code ( ….) { … }
        …
  }
}
```



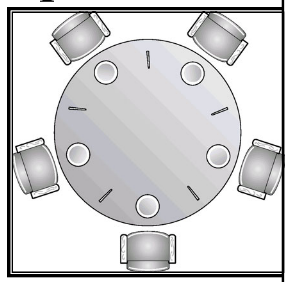
- Any problem?

Condition variables

- No busy-waiting!
  - condition variables
- 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 ()



Dining philosophers: semaphores



- Shared data

```
semaphore chopstick[5]; // initially 1
```

- Using semaphores, for Philosopher *i*:

```
do {
            wait(chopstick[i])
            wait(chopstick[(i+1) % 5])
              …
              eat
              …
            signal(chopstick[i]);
            signal(chopstick[(i+1) % 5]);
              …
              think
              …
} while (1);
```

- Any problem?

6/21/06 CSc 360 5

Dining philosophers: monitors

```
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);
     }
```

6/21/06 CSc 360 6

DP monitor: more

```
void test (int i) {
      if ( (state[(i + 4) % 5] != EATING) &&
      (state[i] == HUNGRY) &&
      (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
          self[i].signal () ; // no effect if not blocked
       }
 }

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

- Using monitors

```
dp.pickup (i)
...
EAT
...
dp.putdown (i)
```

- Any problem?

Implementing monitors

- Variables

```
semaphore mutex;    // for the monitor, initially  = 1
semaphore next;     // for suspended processes, initially  = 0
int next-count = 0; // # of suspended processes
```

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

```
wait(mutex);        // wait for the access to the monitor
                    …
                    body of F;
                    …
if (next-count > 0) // whether there are suspended processes
                    signal(next);
else                // free the monitor to other processes
                    signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementing Condition Variables

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

```
semaphore x-sem; // initially  = 0
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
    signal(next);  // wake up the first one of the suspended q
else
    signal(mutex);// free the monitor
wait(x-sem);       // join the x queue
x-count--;
```

- The operation x.signal can be implemented as:

```
if (x-count > 0) { // no effect if x is not blocked
    next-count++;
    signal(x-sem);// wake up the first one of the x queue
    wait(next);    // join the suspended queue
    next-count--;
}
```


This lecture

- Synchronization with monitors
  - monitor: a high-level ADT
  - using monitors
    - with condition variables
  - implementing monitors
    - with semaphores
- Practice semaphores/monitors with
  - classical synchronization problems