CS 450: Operating Systems
Lecture 11: Monitors

Spring 2014, J. Sasaki
Dept of Computer Science
Illinois Institute of Technology

I

Monitors

Build in Mutual Exclusion

- Build mutex into language
  - python's `with mutex` ...
  - Monitor (ADT/module/object class)
    - Define collection of procedures that should execute mutually exclusively.
    - Define objects whose methods will execute mutually exclusively.
    - Helpful but not a panacea.

3

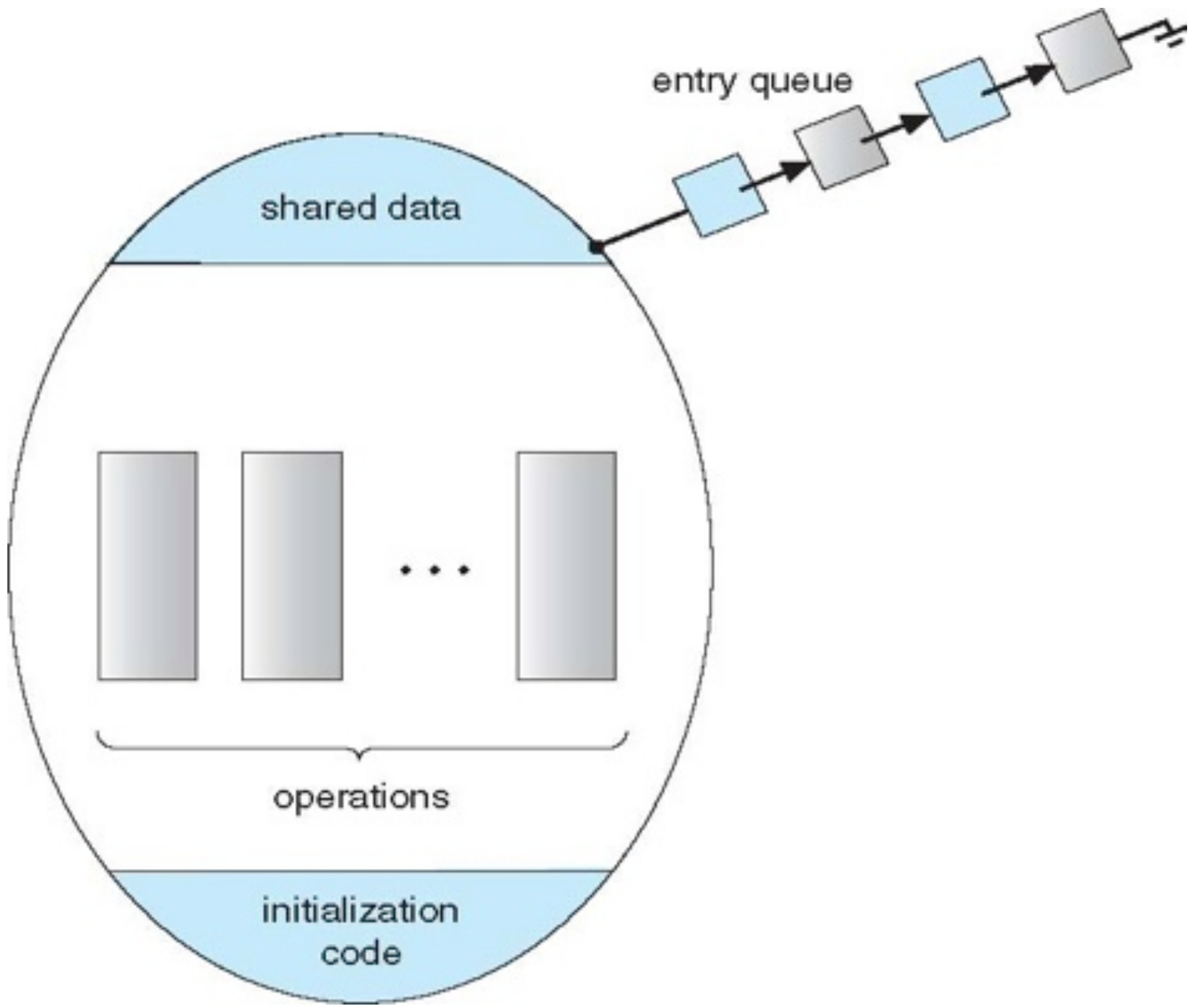
Monitor Mutex

- Typical monitor

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

- Only one procedure/method can be executing at any time; call blocks if necessary.

Schematic view of a Monitor



5

Non-Mutex Blocking

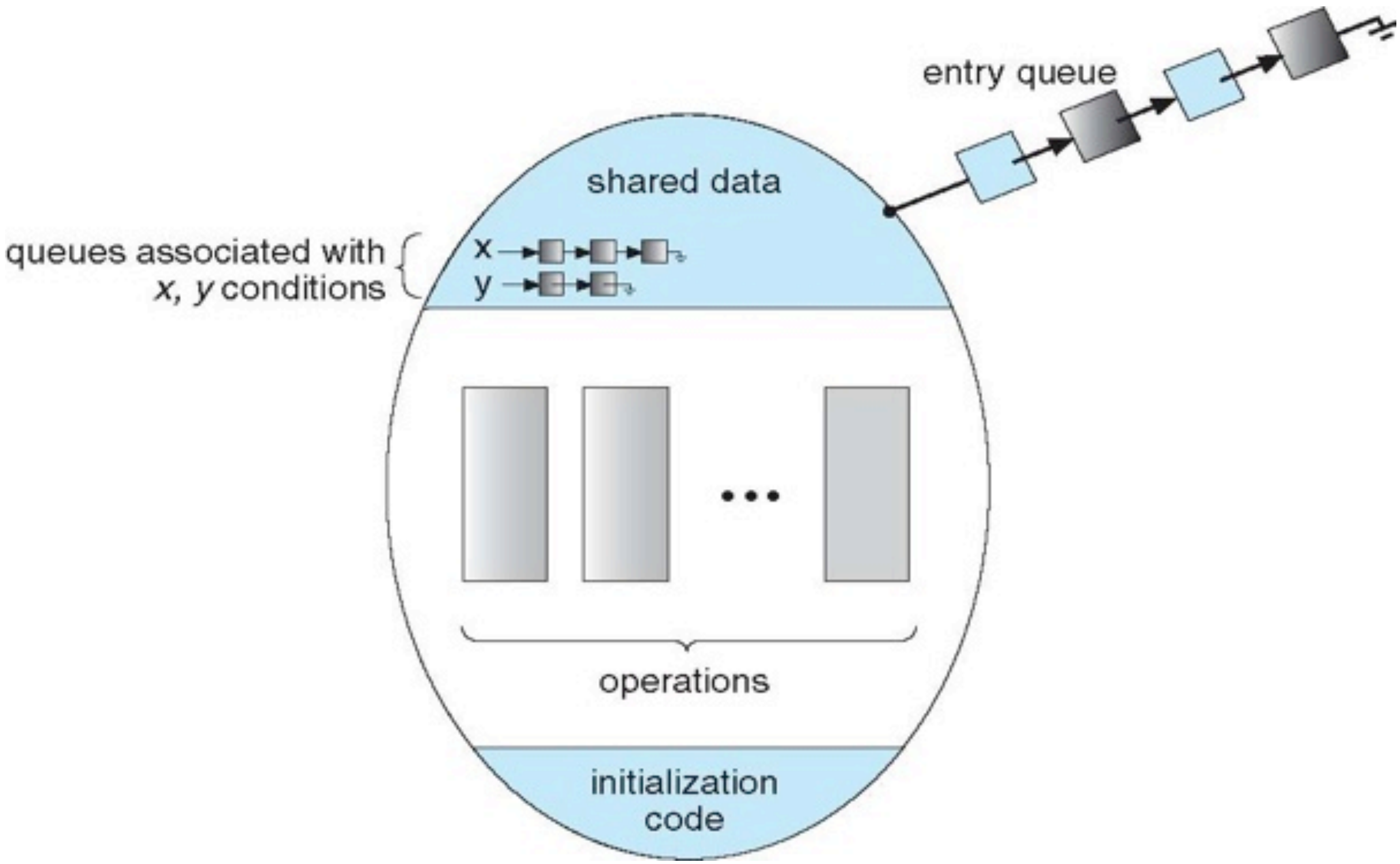
- Semaphore combines notions of atomic action, atomic test, and thread blocking.
  - Mutex permits atomic actions and tests.
  - Semaphores often used as blocks released when a thread achieves some state.
  - Monitors need a blocking mechanism.

Condition Variables

```
condition x, y;
```

- Two operations on a condition variable:
  - `x.wait()` – block this thread.
  - `x.signal()` – resumes one thread of those (if any) that invoked `x.wait()`
    - If no threads are waiting for `x`, then there's no effect.

Monitor with Condition Variables



8

Condition Variables Choices

- If procedure *P* invokes `x.signal()` with *Q* in `x.wait()` state, what should happen next?
- To maintain mutex, if *Q* is resumed then *P* must wait.
    - Should signal stop *P* immediately or when it leaves monitor?

Signaling Options

- Concurrent Pascal, Mesa, C#, Java…:
  - *Signal and leave*: Executing `x.signal()` causes *P* to leave the monitor; *Q* is resumed.
- Older techniques
  - *Signal and wait*: *P* waits until *Q* leaves the monitor or does a `wait()`.
  - *Signal and continue*: *Q* waits until *P* leaves the monitor or does a `wait()`.

Dining Philosophers Solution

- Each philosopher `i` invokes the operations `pickup()` and `putdown()` in the following sequence:
  - `DiningPhilosophers.pickup(i);`
  - *EAT*
  - `DiningPhilosophers.putdown(i);`
- No deadlock, but starvation is possible

Dining Philosophers (Cont.)

```
monitor DiningPhilosophers {
    enum {THINKING, HUNGRY, EATING} state [5];
    condition can_eat[5];

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

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

Dining Philosophers (Cont.)

```
void putdown (int i) {
   state[i] = THINKING;
   // Let neighbors try to eat
   test(left_neighbor(i));
   test(right_neighbor(i));
}

void test(int j) {
   if (state[j] == HUNGRY
      && state[left_neighbor(j)] != EATING
      && state[right_neighbor(j)] != EATING )
   {
      state[j] = EATING;
      can_eat[j].signal();
   }
}
```


13

Monitor Implementation Using Semaphores

- For each monitor:

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

- Wrap body of each procedure with mutex code:

```
wait(mutex);
```

… body of procedure …

```
if (next_count > 0)
    signal(next);   // let someone enter
else
    signal(mutex);
```

Implementing Condition Vars

- For each condition variable `x`:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- `x.wait()` code (uses `x_sem` as a barrier):

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

wait(x_sem);
x_count--;
```

Monitor Implementation (Cont.)

- `x.signal()` code:

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