CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2016 Lecture 21

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

FAQ

- Why not use a Boolean variable instead of Mutex?
- If there are more than one processes waiting, will the semaphore value be negative?
  - Negative: number of processes/threads waiting
  - 0: no waiting threads
  - Postitive: no waiting threads, a wait operation would not put in queue the invoking thread. Often +1
- How to keep a philosopher from starving?
  - There exist solutions that will avoid a deadlock. However they may allow starvation, unless solution is further refined.
- Why not give each philosopher 2 chopsticks?
  - Nice and elegant solution. Widely used in Chinese restaurants. But takes all the fun away from the problem.

FAQ

- Producer-consumer with bounded buffer
  - Should the production and consumption rates be a perfect match?
  - Can the producer add more than 1 item at a time?
- Monitors: what are they and how to implement them.
  - Details coming up.

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- Monitors

Notes

- PA 3 due Friday 11:59 PM
- Project topics and team approach: on canvas now

Notes

Research resources

- Books/Web articles
- Technical news
- IEEE Explore
- ACM Digital Library
- ScienceDirect

Accessing library resources from Home

Google Scholar

Problems with Semaphores

- Incorrect use of semaphore operations:
  - Omitting of wait (mutex)
    - Violation of mutual exclusion
  - or signal (mutex)
    - Deadlock!

Monitors

- Monitor: A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
  - Automatically provide mutual exclusion
- Originally proposed for Concurrent Pascal 1975
- Directly supported by Java but not C

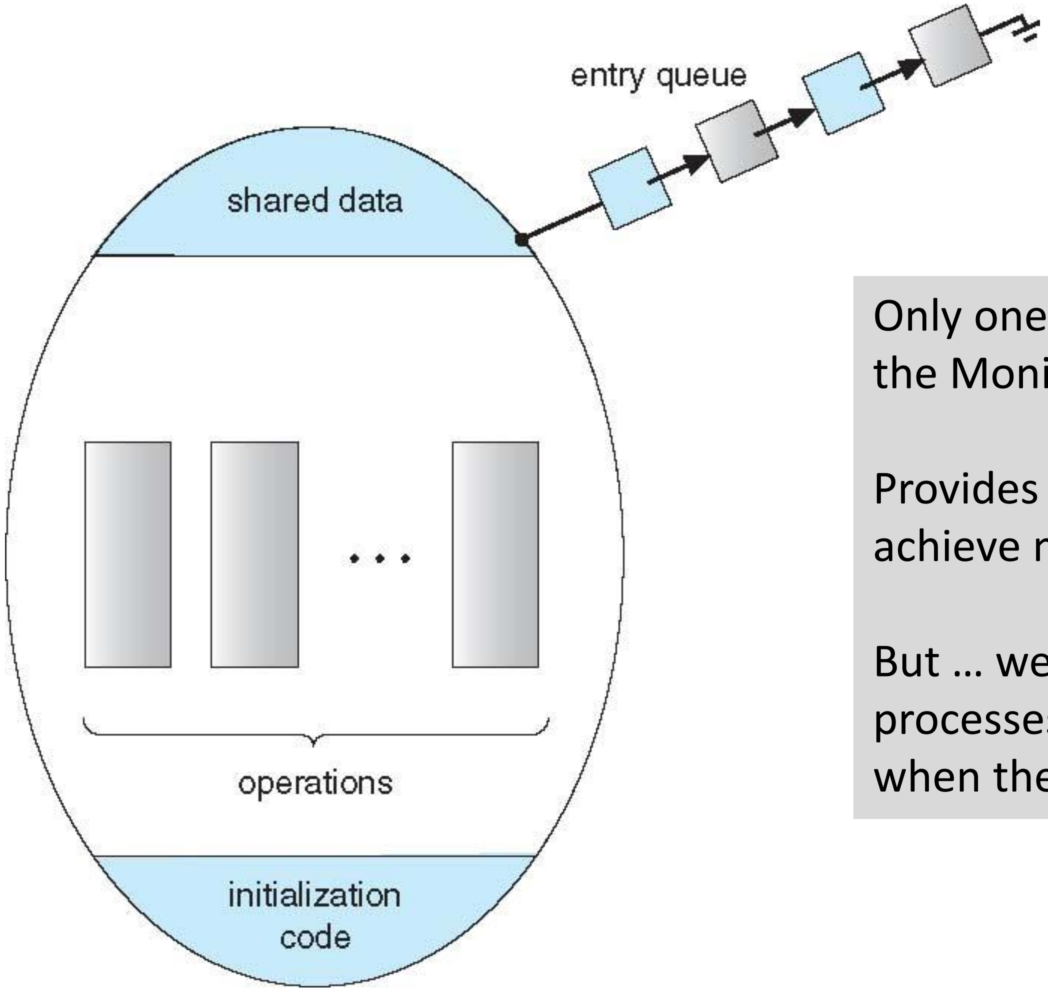
Monitors

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

  procedure Pn (…) {……}

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

Schematic view of a Monitor



Only one process/thread in the Monitor

Provides an easy way to achieve mutual exclusion

But … we also need a way for processes to block when they cannot proceed

Condition Variables

Compare with semaphore

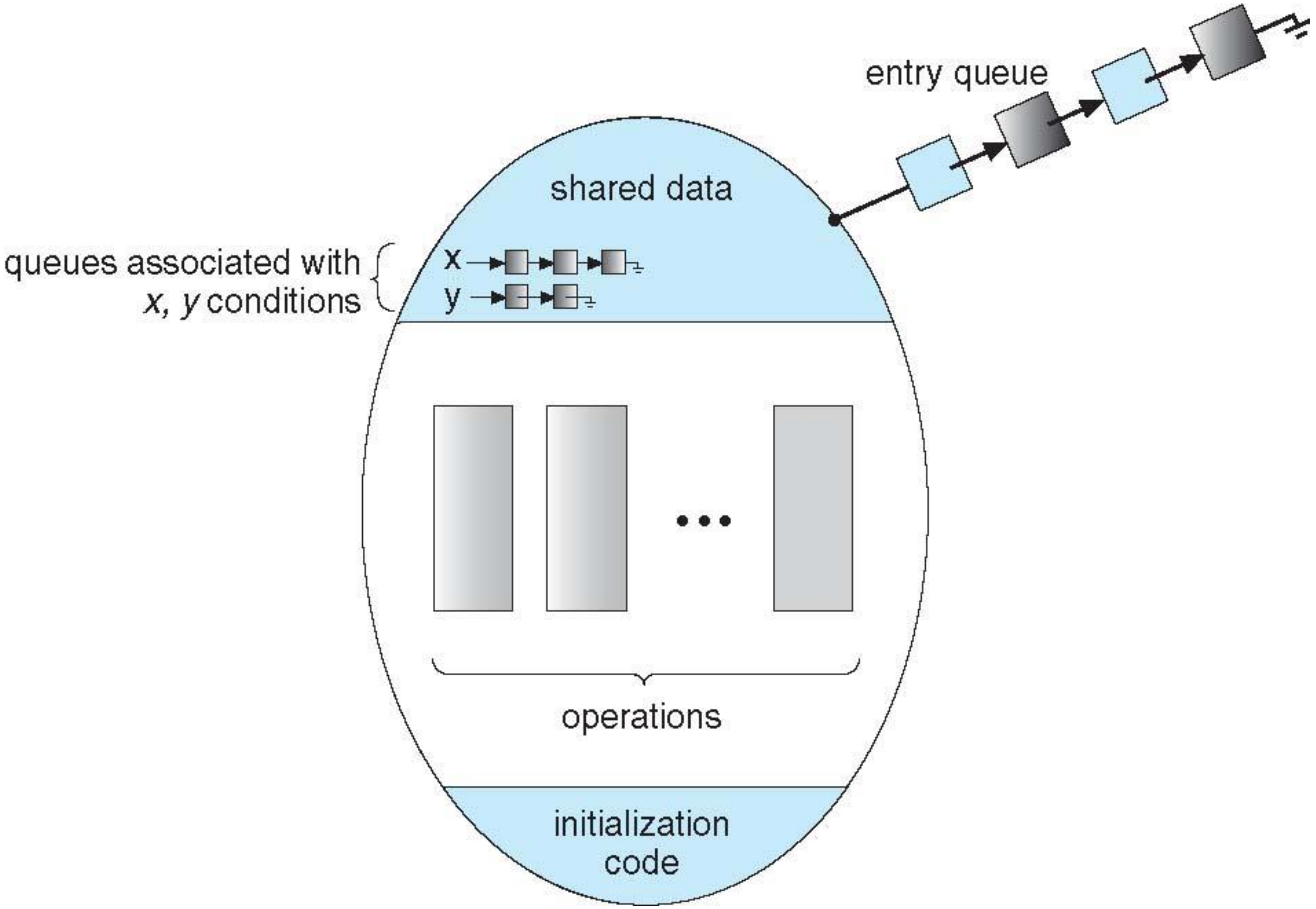
`The` **`condition`** `construct`

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Difference between the signal() in semaphores and monitors

- Condition variables in Monitors: Not persistent
  - If a signal is performed and no waiting threads?
    - Signal is simply ignored
  - During subsequent wait operations
    - Thread blocks
- Semaphores
  - Signal increments semaphore value even if there are no waiting threads
    - Future wait operations would immediately succeed!

Monitor with Condition Variables



Colorado State University

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal ('75) compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including C#, Java

Difference between the signal() in semaphores and monitors

- Monitors {condition variables}: Not persistent
  - If a signal is performed and no waiting threads?
    - Signal is simply ignored
  - During subsequent wait operations
    - Thread blocks
- Semaphores
  - Signal increments semaphore value even if there are no waiting threads
  - Future wait operations would immediately succeed!

Guest: Dr. Shrideep Pallickara

- Research in distributed systems and predictive analytics
  - Big data, virtualized cloud
- CS455: Introduction to Distributed Systems
  - Spring 2017

Monitor Solution to Dining Philosophers: Deadlock-free

```
enum {THINKING,HUNGRY,EATING} state[5];
```

- `state[i] = EATING only if`
  - `state[(i+4)%5] != EATING &&   state[(i+1)%5] != EATING !`
- `condition self[5]`
  - Delay self when **HUNGRY but unable** to get chopsticks

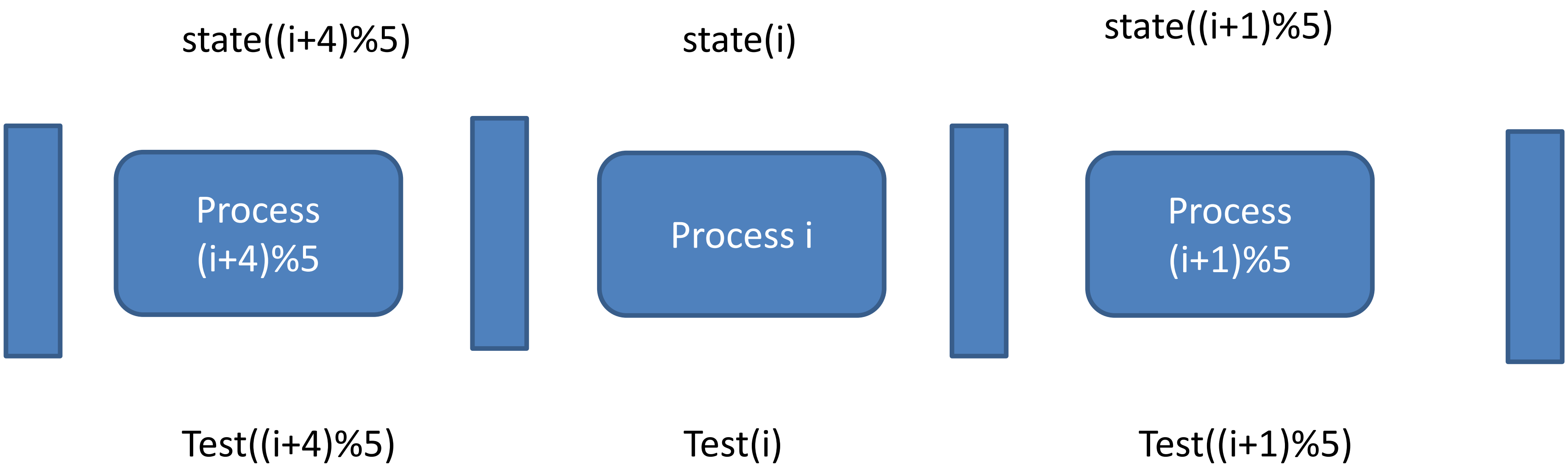
Sequence of actions

- Before eating, must invoke pickup()
  - May result in suspension of philosopher process
  - After completion of operation, philosopher may eat

```
 think
DiningPhilosophers.pickup(i);
eat
DiningPhilosophers.putdown(i);
think
```

Monitor Solution to Dining Philosophers: Deadlock-free

```
enum {THINKING,HUNGRY,EATING} state[5];
```



The pickup() and putdown() operations

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

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);    //on next slide
          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);
}
```

Suspend self if unable to acquire chopstick

Check to see if person on left or right can use the chopstick

test() to see if philosopher I can eat

```
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;
     }
}
```

Eat only if HUNGRY and Person on Left AND Right are not eating

Signal a process that was suspended while trying to eat

Possibility of starvation

- Philosopher i can starve if eating periods of philosophers on left and right overlap
- Possible solution
  - Introduce new state: STARVING
  - Chopsticks can be picked up if no neighbor is starving
    - Effectively wait for neighbor's neighbor to stop eating
    - REDUCES concurrency!

Monitor Implementation Using Semaphores

Monitor Implementation Mutual Exclusion

For each monitor

- Semaphore mutex initialized to 1
- Process must execute
  - wait(mutex) : Before entering the monitor
  - signal(mutex): Before leaving the monitor

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1) allows only one process to be active
semaphore next;   // (initially  = 0) causes signaler to sleep
int next_count = 0;   num of sleepers since they signalled
```

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

```
wait(mutex);
      …
    body of F;
      …
if (next_count > 0)
 signal(next)
else
 signal(mutex);
```

Both mutex and next have an associated queue

- Mutual exclusion within a monitor is ensured