CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2016 Lecture 20

Slides based on

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

Recall …

- Critical Section solution
  - Entry section, Critical section, Exist section
- In addition to Mutual Exclusion, we need
  - Progress, Bounded Waiting
- Locks: need atomicity
- Mutex locks: acquire () and release ( )
- Semaphores: wait (S) and signal(S)
  - Counting and Binary (Mutex)
  - block and wakeup processes, lists associated with semaphore S

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Binary semaphore (mutex)
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1
- Counting semaphores
  - empty: Number of empty slots available
    - Initialized to n
  - full: Number of filled slots available n
    - Initialized to 0

Bounded-Buffer : Note

- Producer and consumer must be ready before they attempt to enter critical section
- Producer readiness?
  - When a slot is available to add produced item
    - wait(empty)
      - empty is initialized to n
- Consumer readiness?
  - When a producer has added new item to the buffer
    - wait(full)
      - full initialized to 0

Bounded Buffer Problem (Cont.)

The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);          wait till slot available
    wait(mutex);      Allow producer OR consumer to (re)enter critical section
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex); Allow producer OR consumer to (re)enter critical section
    signal(full);         signal consumer that a slot is available
} while (true);
```

Bounded Buffer Problem (Cont.)

The structure of the consumer process

```
Do {
    wait(full);   wait till slot available for consumption
    wait(mutex);  Only producer OR consumer can be in critical section
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);  Allow producer OR consumer to (re)enter critical section
    signal(empty);  signal producer that a slot is available to add
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```

Notes

- PA 3 available
- Project topics and team approach to be announced soon

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 (mutual exclusion for writer)
  - Semaphore **`mutex`** initialized to 1 (mutual exclusion for read_count)
  - Integer **`read_count`** initialized to 0 (how many readers?)

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
    wait(rw_mutex);
        ...
    /* writing is performed */
        ...
    signal(rw_mutex);
} while (true);
```

When: writer in critical section
and if n readers waiting:
- 1 reader is queued on rw_mutex
- (n-1) readers queued on mutex

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
     wait(mutex);
       read_count++;
       if (read_count == 1)
            wait(rw_mutex);
     signal(mutex);
        ...
       /* reading is performed */
        ...
     wait(mutex);
       read count--;
       if (read_count == 0)
            signal(rw_mutex);
     signal(mutex);
} while (true);
```

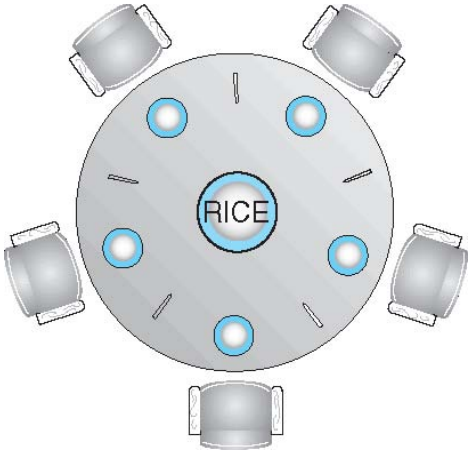
mutex for mutual exclusion to readcount

When:
writer in critical section
and if n readers waiting
1 is queued on rw_mutex
(n-1) queued on mutex

Readers-Writers Problem Variations

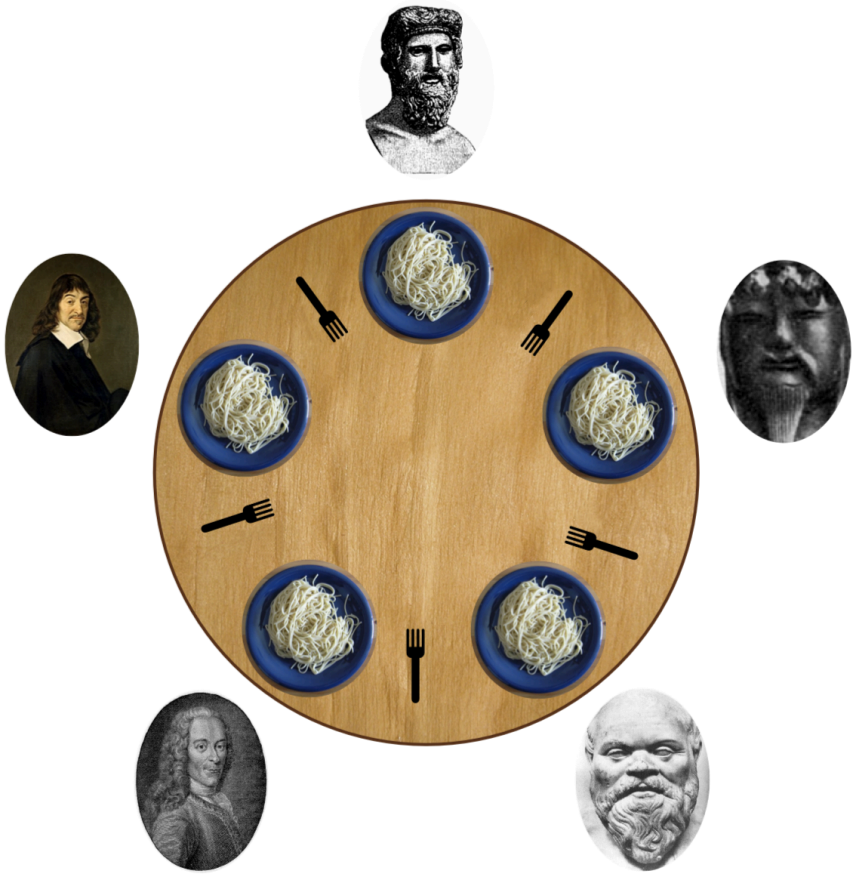
- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat,
  - then release both when done
- Each chopstick is a semaphore
  - Grab by executing wait ( )
  - Release by executing signal ( )
- Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem



Plato, Confucius, Socrates, Voltaire and Descartes

Colorado State University

Dining-Philosophers Problem Algorithm: Simple solution?

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                // eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                // think

} while (TRUE);
```

- What is the problem with this algorithm?
  - If all of them pick up the the left chopstick first - Deadlock

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table (with the same 5 forks).
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex): what happens?
    - Several processes in critical section

Problems with Semaphores

- Incorrect use of semaphore operations:
  - wait (mutex) … wait (mutex)): what happens?
    - deadlock!

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

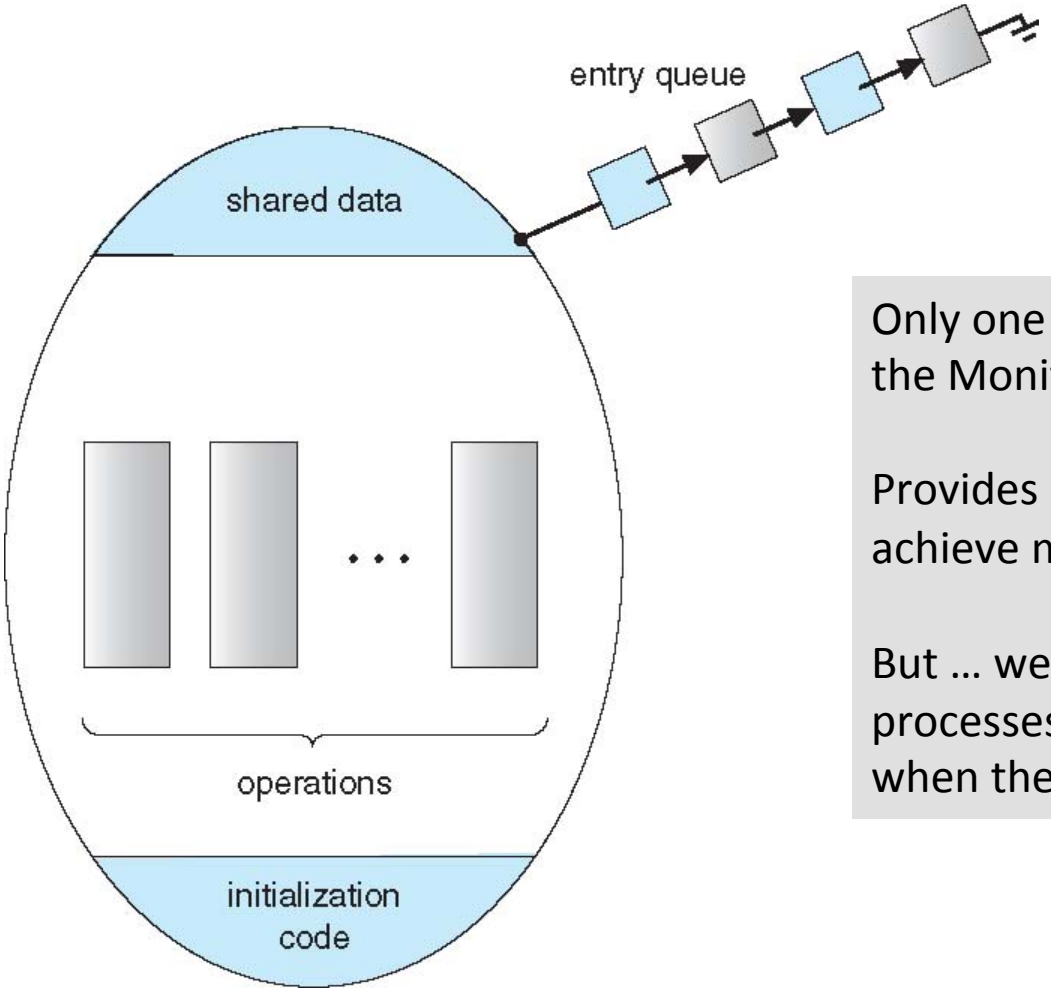
Monitors

```
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

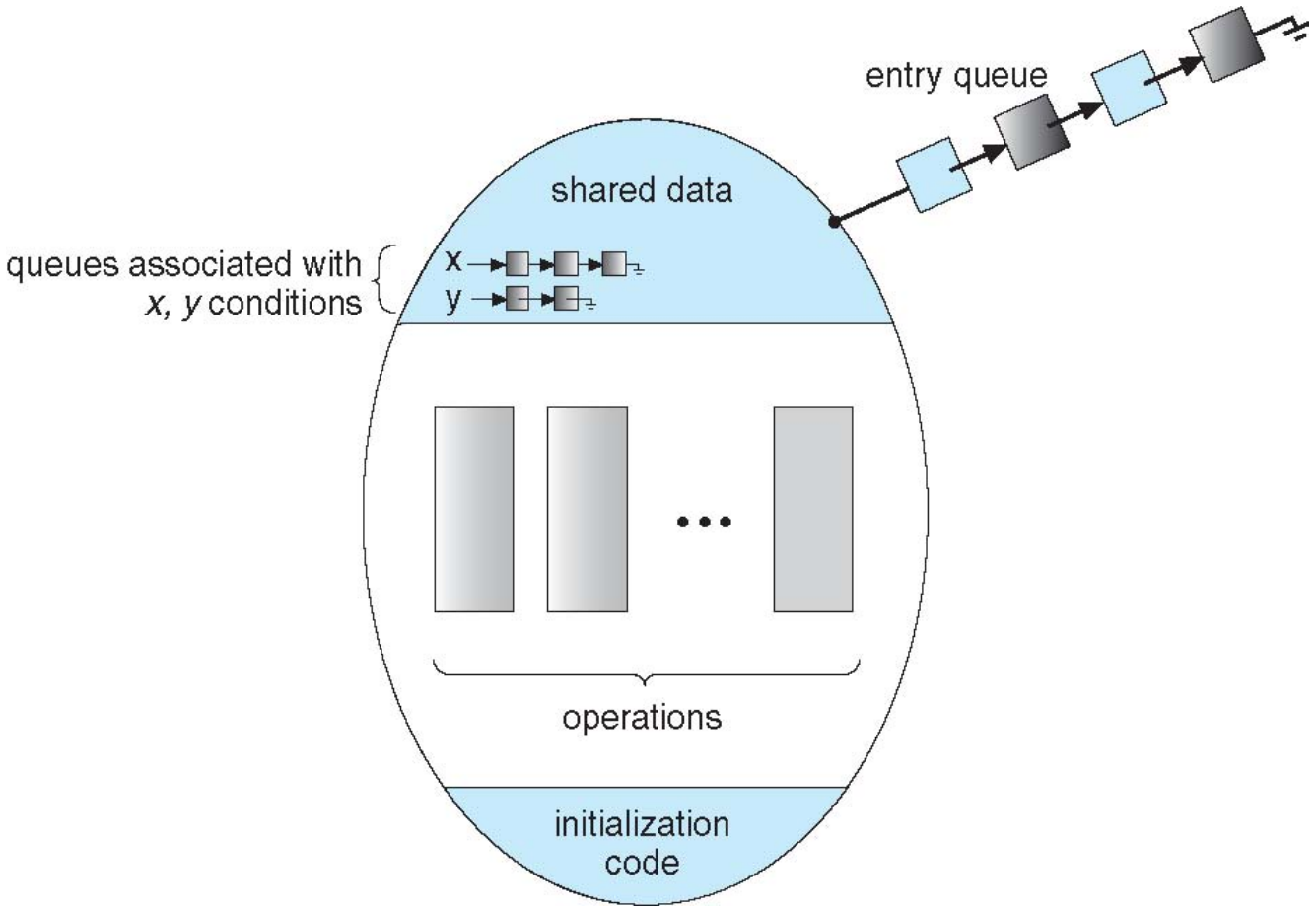
Condition Variables

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

Compare with semaphore

Monitor with Condition Variables



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!