COMP3511 Operating Systems

Topic 6: Process Synchronization (Part II)

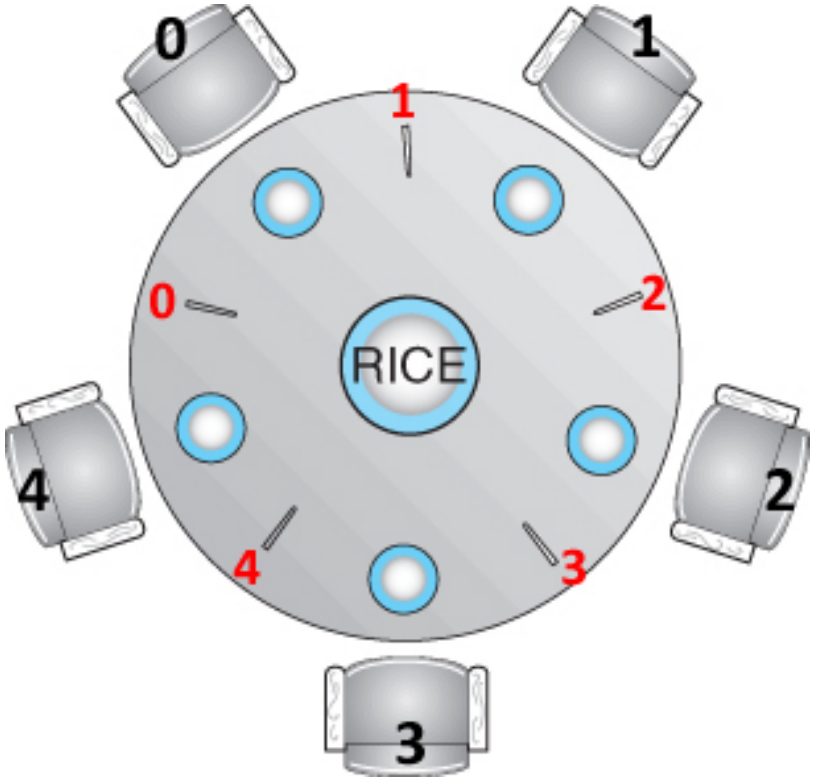
Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Acknowledgment: The lecture notes are based on various sources on the Internet

Dining-Philosophers Problem

- It's lunch time in the philosophy department
- Five philosophers, each either eats or thinks
- Share a circular table with five chopsticks
- Thinking: do nothing
- Eating: need two chopsticks, try to pick up two closest chopsticks
  - Block if neighbour has already picked up a chopstick
- After eating, put down both chopsticks and go back to thinking



- Shared data:
  - Bowl of rice (data set)
  - Semaphore: chopstick[5] initialized to 1

Dining-Philosophers Problem: Algorithm

- 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(1);
```

- This guarantees that no two neighbours are eating simultaneously
- What is the problem with this algorithm? Deadlock
  - Suppose all five philosophers become hungry at the same time, and each grabs its right chopstick... while waiting for its left chopstick

How to avoid this deadlock situation?

Problems with Semaphores

- Semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since such errors happen only if particular execution sequences take place, and these sequences do not always occur
- Incorrect use of semaphore operations results with deadlock and starvation
  - If wait and signal on mutex is interchanged, then it may result in several processes entering into their critical sections at the same time violating the mutual exclusion
    `signal(mutex) ... critical section ... wait(mutex)`
  - If a `wait(mutex)` call with a `wait(mutex)` call then deadlock will occur
    `wait(mutex) ... wait(mutex)`
  - Omitting of `wait(mutex)` or `signal(mutex)` or both, either mutual exclusion is violated or a deadlock will occur

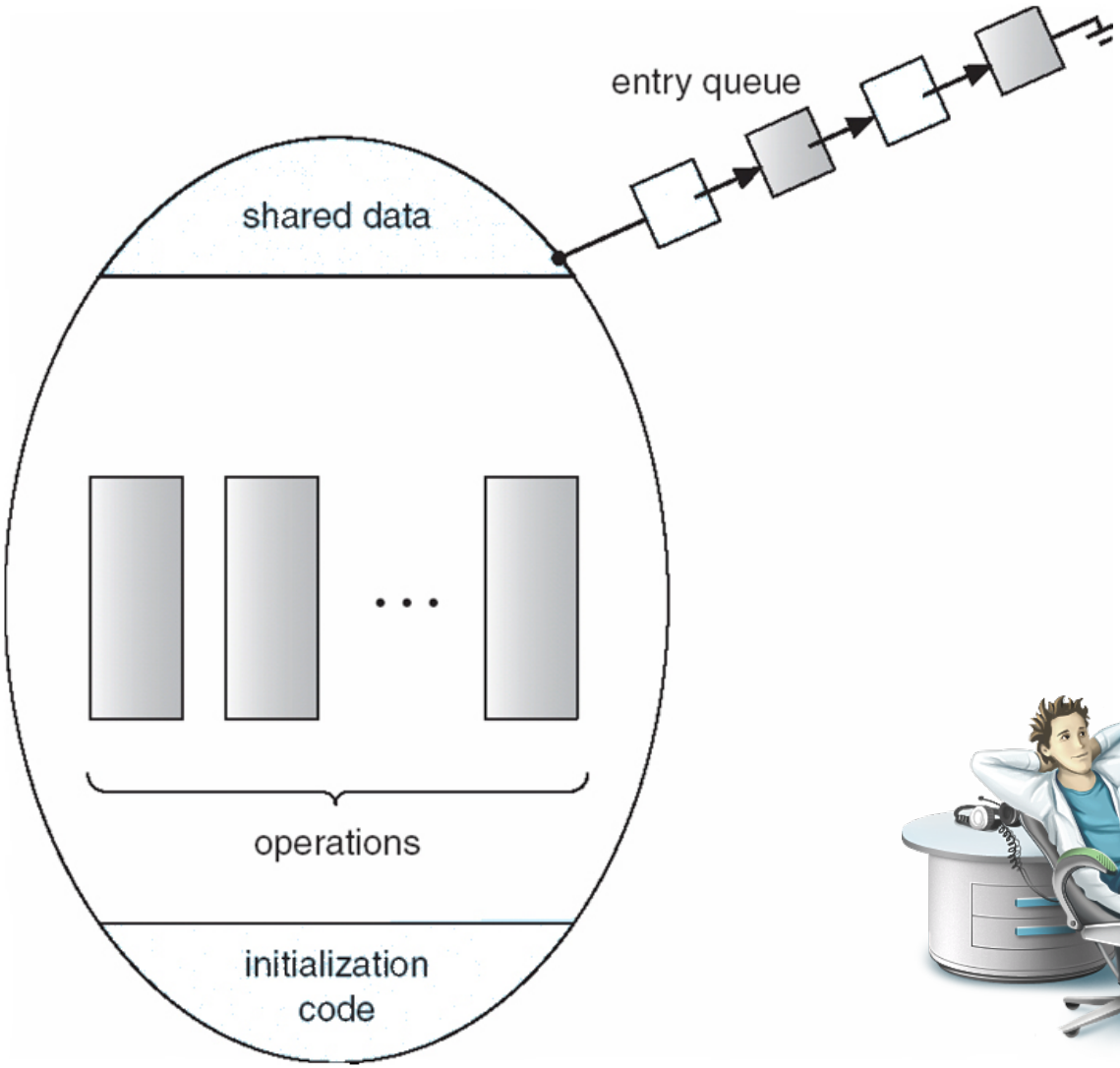
Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- It encapsulates
  - Mutex
  - Shared data
  - Access methods / Procedures that operate on the shared data
  - Synchronization between concurrent procedure invocations

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

- A monitor protects its data from unstructured access
- It guarantees mutual exclusion

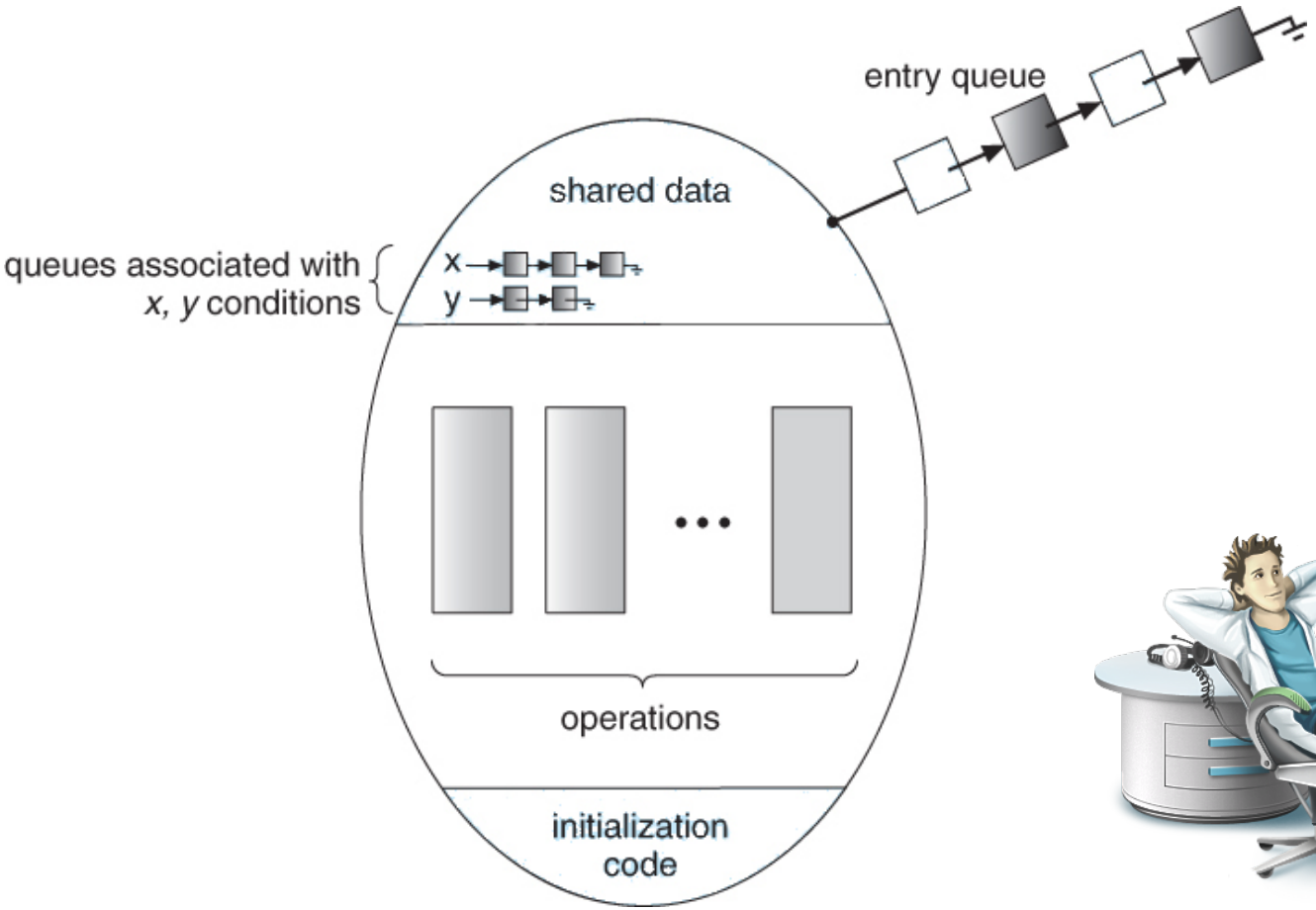
Monitors: Schematic View of a Monitor



Condition Variables

- There are many cases where a process wishes to check whether a condition is true before continuing its execution
- Condition variable lets us block until a condition is met
- `condition x, y;`
- Two operations on a condition variable:
  - `x.wait()` - a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` - resumes one of the processes (if any) that invoked `x.wait()`
    - ★ If no `x.wait()` on the variable, then it has no effect on the variable
    - ★ Note that `x.signal()` is unlike the semaphore's signal() operation, which preserves state in terms of the value of the semaphore

Monitor with Condition Variables



Condition Variables Choices

If process P invokes x.signal(), with process Q in x.wait() state, what should happen next?

- If Q is resumed, then P must wait, since they cannot be inside the monitor simultaneously

- Options include
  - Signal and wait - P either waits until Q leaves monitor or waits for another condition
  - Signal and continue - Q either waits until P leaves the monitor or waits for another condition
- Both have pros and cons - language implementer can decide
- Monitors implemented in Concurrent Pascal compromise
  - P executing signal immediately leaves the monitor, Q is resumed

Solution to Dining Philosophers using Monitor

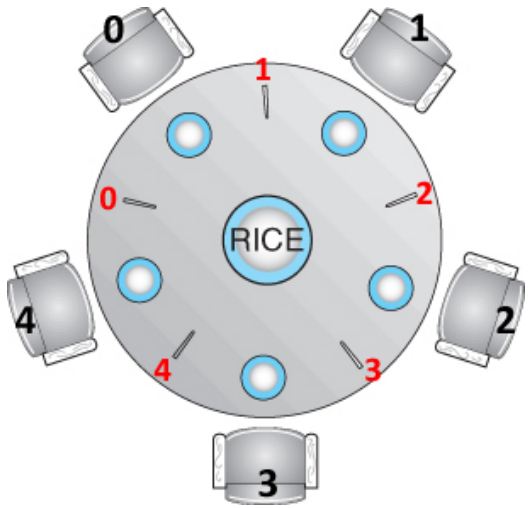
```
monitor DiningPhilosophers {
  enum {
    THINKING,
    HUNGRY,
    EATING
  } state[5];
  condition self[5];

  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;
  }
}
```



Explanation

```
monitor DiningPhilosophers {
  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);
  }
```

Explanation:

- Pickup chopsticks
  - Indicate that I am hungry
  - Set state to eating in test() only if my left and right neighbors are not eating
  - If unable to eat, wait to be signaled
- Put down chopsticks
  - If right neighbor R = (i+4)%5 is hungry and both of R's neighbors are not eating, set R's state to eating and wake it up by signaling R's condition variable

Explanation (Cont'd)

```
    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;
    }
}
```

Explanation:

- test
  - signal() has no effect during pickup(), but is important to wake up waiting hungry philosophers during putdown()
- Execution of pickup(), putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing
- Verify that this monitor-based solution is
  - deadlock-free
  - mutually exclusive in that no 2 neighbors can eat simultaneously

Solution to Dining Philosophers (Cont'd)

- 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
  (Reason: A particular philosopher may be unable to get both chopsticks because of a timing problem)

Monitor Implementation using Semaphores

- Variables

```
semaphore mutex; // initially = 1
semaphore next;  // initially = 0, means next to exit
int next_count = 0; // number of processes blocked on next
```

- A signaling process must wait until the resumed process either leaves or waits, then signaling processes can use `next` (initialized to 0) to suspend themselves. An integer variable `next_count` is used to count the number of processes suspended on `next`
- The monitor "compiler" has to automatically insert the following code into compiled procedure F:

```
Procedure F:
  wait(mutex);
  ...
  body of F;
  ...
  if(next_count > 0) signal(next); // resume a suspended process
  else signal(mutex); // allow a new process into the monitor
end;
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation using Semaphores (Cont'd)

- For each condition variable x, we have:

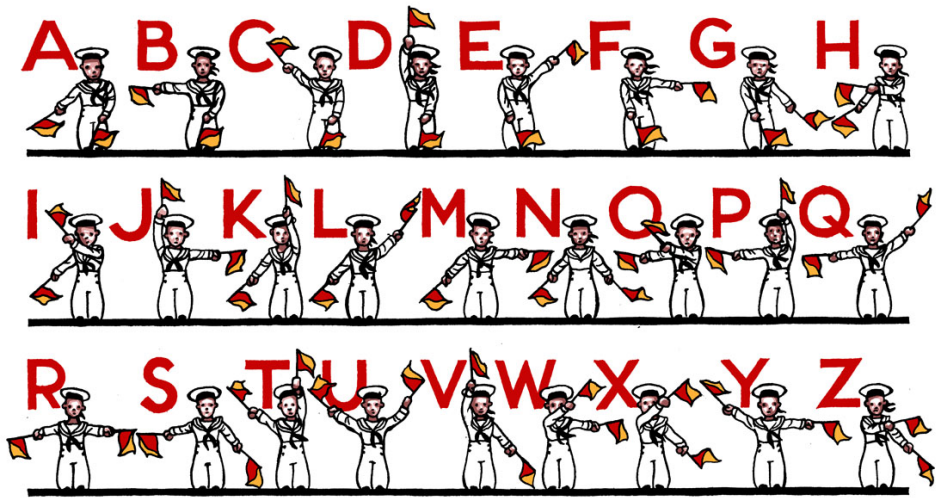
```
semaphore x_sem; // initially = 0
int x_count = 0;
```

  - The operation `x.wait()` can be implemented as:

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



Monitor Implementation: Explanation

- `x.wait()`
  - The process checks if there are other processes waiting to enter the monitor (*next_count*), if there is, let one of them enter; otherwise it relinquishes the monitor.
  - After that, it suspends itself `wait(`*x_sem*`)`.
  - *x_count*`--` will be executed when it is waked up by later by another process
- `x.signal()`
  - If there is no process waiting on condition `x`, `x.signal` has no effect
  - The process after waking up a process waiting on *x_sem*, will need to give up the monitor, and join the entry queue (next) to wait for its next turn to enter the monitor

Solaris Synchronization

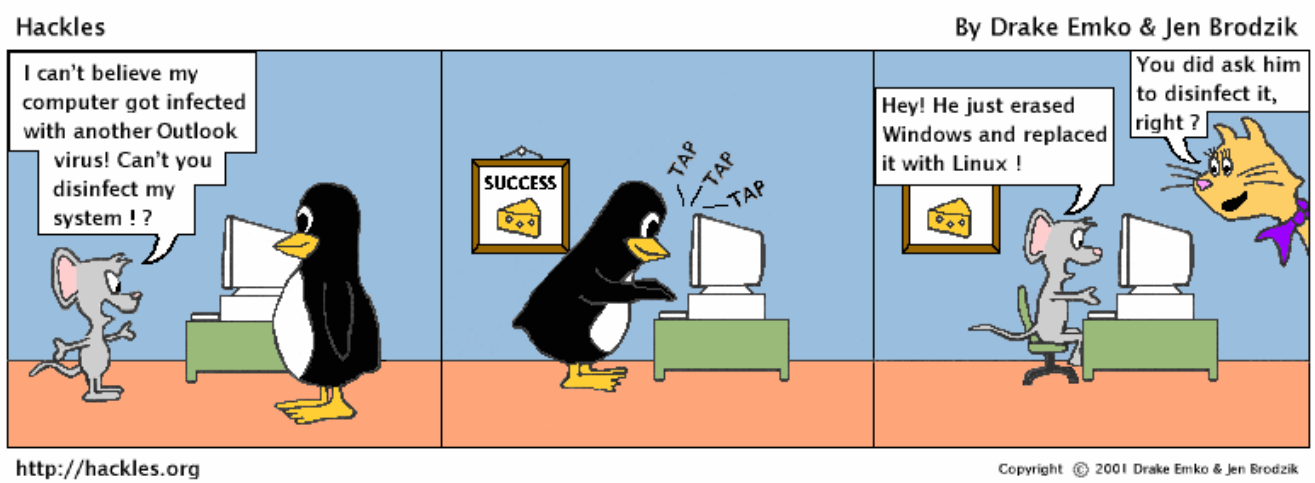
- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutex for efficiency when protecting data from short code segments, less than a few hundred instructions
  - Starts as a standard semaphore implemented as a spinlock in a multiprocessor system
  - If lock held, and by a thread running on another CPU, spins to wait for the lock to become available
  - If lock held by a non-run-state thread, block and sleep waiting for signal of lock being released
- Uses condition variables
- Uses readers-writers locks when longer sections of code need access to data. These are used to protect data that are frequently accessed, but usually in a read-only manner. The reader-writer locks are relatively expensive to implement

Windows Synchronization

- The kernel uses interrupt masks to protect access to global resources on uniprocessor systems
- The kernel uses spinlocks in multiprocessor systems
  - For efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock
- For thread synchronization outside the kernel, Windows provides dispatcher objects, threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers
- Events are similar to condition variable; they may notify waiting thread when a desired condition occurs
- Timers are used to notify one or more thread that a specified amount of time has expired
- Dispatcher objects either signaled-state (object available) or non-signaled state (thread will block)

Linux Synchronization

- Prior to kernel version 2.6, disables interrupts to implement short critical sections
- Version 2.6 or later, fully preemptive kernel
- It provides:
  - Semaphores
  - Spinlocks
  - Reader-writer versions of both
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption



That's all!

Any questions?