Dining Philosophers, Monitors, and Condition Variables

CSCI 3753 Operating Systems

Spring 2005

Prof. Rick Han

Announcements

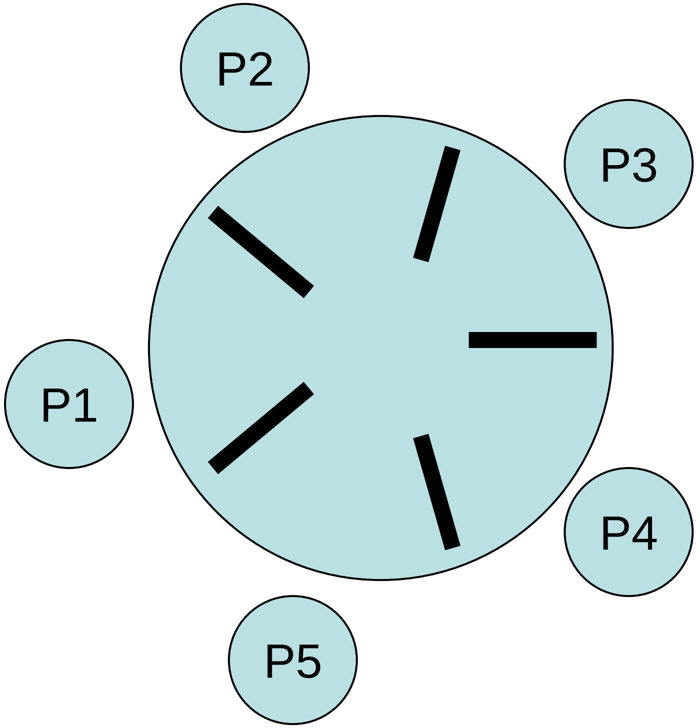
- HW #3 is due Friday Feb. 25, a week+ from now
  - submitting graphic: .doc OK? - will post an answer
  - extra office hours Thursday 1 pm - post this
- TA finished regrading some HWs that were cut off by moodle
- Slides on synchronization online
- PA #2 is coming, assigned around Tuesday night
- Midterm is tentatively Thursday March 10
- Read chapters 9 and 10

From last time...

- We discussed semaphores
- Deadlock
- Classic synchronization problems
  - Bounded Buffer Producer/Consumer Problem
  - First Readers/Writers Problem
  - Dining Philosophers Problem

Dining Philosophers Problem

- *N* philosophers seated around a circular table
  - There is one chopstick between each philosopher
  - A philosopher must pick up its two nearest chopsticks in order to eat
  - A philosopher must pick up first one chopstick, then the second one, not both at once
- Devise an algorithm for allocating these limited resources (chopsticks) among several processes (philosophers) in a manner that is
  - deadlock-free, and
  - starvation-free



Dining Philosophers Problem

- A simple algorithm for protecting access to chopsticks:
  - each chopstick is governed by a mutual exclusion semaphore that prevents any other philosopher from picking up the chopstick when it is already in use by another philosopher

```
semaphore chopstick[5];    // initialized to 1
```

  - Each philosopher grabs a chopstick *i* by P(chopstick[i])
  - Each philosopher releases a chopstick *i* by V(chopstick[i])

Dining Philosophers Problem

- Pseudo code for Philosopher i:

```
while(1) {
    // obtain the two chopsticks to my immediate right and left
    P(chopstick[i]);
    P(chopstick[(i+1)%N];

    // eat

    // release both chopsticks
    V(chopstick[(i+1)%N];
    V(chopstick[i]);
}
```

- Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one its two neighboring philosophers

Dining Philosophers Problem

- Unfortunately, the previous “solution” can result in deadlock
  - each philosopher grabs its right chopstick first
    - causes each semaphore’s value to decrement to 0
  - each philosopher then tries to grab its left chopstick
    - each semaphore’s value is already 0, so each process will block on the left chopstick’s semaphore
  - These processes will never be able to resume by themselves - we have deadlock!

Dining Philosophers Problem

- Some deadlock-free solutions:
  - allow at most 4 philosophers at the same table when there are 5 resources
  - odd philosophers pick first left then right, while even philosophers pick first right then left
  - allow a philosopher to pick up chopsticks only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.
    - we'll see this solution next using monitors
- A deadlock-free solution is not necessarily starvation-free
  - for now, we'll focus on breaking deadlock

Monitors and Condition Variables

- semaphores can result in deadlock due to programming errors
  - forgot to add a P() or V(), or misordered them, or duplicated them
- to reduce these errors, introduce high-level synchronization primitives, e.g. *monitors with condition variables*, that essentially automates insertion of P and V for you
  - As high-level synchronization constructs, monitors are found in high-level programming languages like Java and C#
  - underneath, the OS may implement monitors using semaphores and mutex locks

Monitors and Condition Variables

- Declare a monitor as follows (looks somewhat like a C++ class):

monitor *monitor_name* {

// shared local variables

function f1(...) {

...

}

...

function fN(...) {

...

}

init_code(...) {

...

}

}

- A monitor ensures that only 1 process/thread at a time can be active within a monitor
  - simplifies programming, no need to explicitly synchronize
- Implicitly, the monitor defines a mutex lock

  semaphore mutex = 1;
- Implicitly, the monitor also defines essentially mutual exclusion around each function
  - Each function’s critical code is surrounded as follows:

    function fj(...) {

    P(mutex)

    // critical code

    V(mutex)

    }
- The monitor’s local variables can only be accessed by local monitor functions
- Each function in the monitor can only access variables declared locally within the monitor and its parameters

Monitors and Condition Variables

- Example:

```
monitor sharedcounter {
   int counter;
   function add() { counter++;}
   function sub() { counter--;}
   init() { counter=0; }
}
```

- If two processes want to access this sharedcounter monitor, then access is mutually exclusive and only one process at a time can modify the value of counter
  - if a write process calls sharedcounter.add(), then it has exclusive access to modifying counter until it leaves add(). No other process, e.g. a read process, can come in and call sharedcounter.sub() to decrement counter while the write process is still in the monitor

Monitors and Condition Variables

- In the previous *sharedcounter* example, a writer process may be interacting with a reader process via a bounded buffer
  - like the solution to the bounded buffer producer/consumer problem, the writer should signal blocked reader processes when there are no longer zero elements in the buffer
  - monitors alone don't provide this signalling synchronization capability
- In general, there may be times when one process wishes to signal another process based on a condition, much like semaphores.
  - Thus, monitors alone are insufficient.
  - Augment monitors with *condition variables*.

Monitors and Condition Variables

- A condition variable *x* in a monitor allows two main operations on itself:
  - x.wait() -- suspends the calling process until another process calls x.signal()
  - x.signal() -- resumes exactly 1 suspended process. If none, then *no effect*.
    - Note that x.signal() is unlike the semaphore's signalling operation V(), which preserves state in terms of the value of the semaphore.
      - Example: if a process Y calls x.signal() on a condition variable x before process Z calls x.wait(), then Z will wait. The condition variable doesn't remember Y's signal.
      - Comparison: if a process Y calls V(mutex) on a binary semaphore mutex (initialized to 0) before process Z calls P(mutex), then Z will not wait, because the semaphore remembers Y's V() because its value = 1, not 0.
  - the textbook mentions that a third operation can be performed x.queue()
- Declare a condition variable with pseudo-code:

```
condition x,y;
```

Monitors and Condition Variables

- Semantics concerning what happens just after x.signal() is called by a process P in order to wake up a process Q waiting on this CV x
  - Hoare semantics, also called signal-and-wait
    - The signalling process P either waits for the woken up process Q to leave the monitor before resuming, or waits on another CV
  - Mesa semantics, also called signal-and-continue
    - The signalled process Q waits until the signalling process P leaves the monitor or waits on another condition

Monitor-based Solution to Dining Philosophers

- Key insight: pick up 2 chopsticks only if both are free
  - this avoids deadlock
  - reword insight: a philosopher moves to his/her eating state only if both neighbors are not in their eating states
    - thus, need to define a state for each philosopher
  - 2nd insight: if one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
    - thus, states of each philosopher are: thinking, hungry, eating
    - thus, need condition variables to signal() waiting hungry philosopher(s)
  - Also, need to Pickup() and Putdown() chopsticks

Monitor-based Solution to Dining Philosophers

- Some basic pseudo-code for monitor (we'll abbreviate DP for Dining Philosophers):

```
monitor DP {
   status state[5];
   condition self[5];
   Pickup(int i);
   Putdown(int i);
}
```

- Each philosopher *i* runs pseudo-code:

DP.Pickup(*i*);

...

DP.Putdown(*i*);

Monitor-based Solution to Dining Philosophers

- Full code for monitor solution (continued on next slide):

```
monitor DP {
   status state[5];
   condition self[5];

   Pickup(int i) {
      state[i] = hungry;
      test(i);
      if(state[i]!=eating) self[i].wait;
   }

   Putdown(int i) {
      state[i] = thinking;
      test((i+1)%5);
      test((i-1)%5);
   }
   ... monitor code continued next slide ...
```

- Pickup chopsticks
  - indicate that I'm hungry
  - set state to eating in test() only if my left and right neighbors are not eating
  - if unable to eat, wait to be signalled
- Put down chopsticks
  - if right neighbor R=(i+1)%5 is hungry and both of R's neighbors are not eating, set R's state to eating and wake it up by signalling R's CV

Monitor-based Solution to Dining Philosophers

... monitor code continued from previous slide...

```
...
test(int i) {
   if (state[(i+1)%5] != eating &&
      state[(i-1)%5] != eating &&
      state[i] == hungry) {

      state[i] = eating;
      self[i].signal();
   }
}

init() {
   for i = 0 to 4
      state[i] = thinking;
}

}  // end of monitor
```

- 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