Chapter 5: Condition Variables, Monitors

CSCI 3753 Operating Systems

William Mortl, MS and Rick Han, PhD

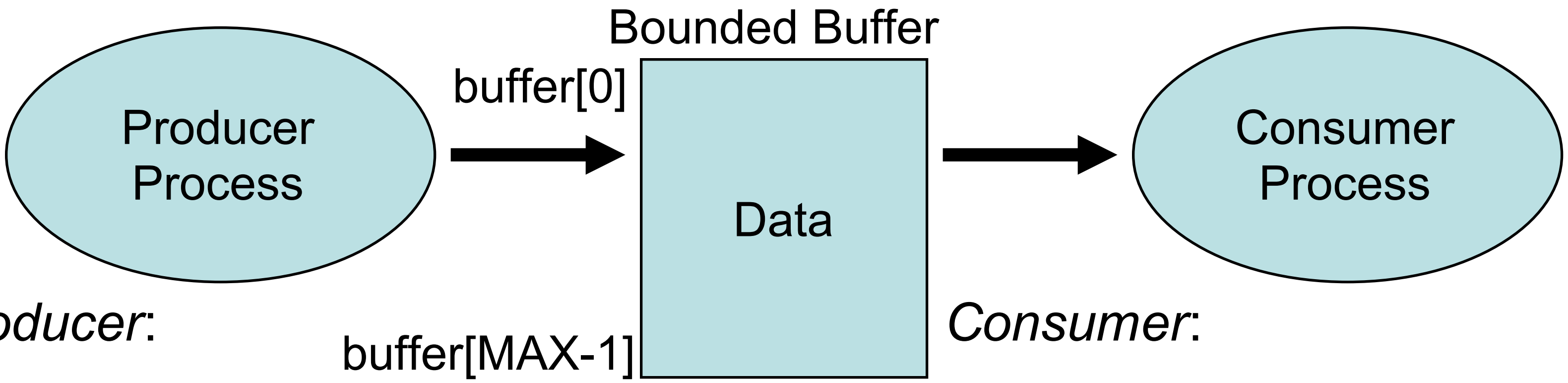
University of Colorado at Boulder

University of Colorado **Boulder**

Recap

- Classic synchronization problems
  1. Bounded Buffer Producer/Consumer
  2. Readers/Writers Problem
  3. Dining Philosophers Problems

A Bounded Buffer P/C Solution



Producer:

```
while(1) {
  P(empty)
    P(mutex)
      add item to buffer
    V(mutex)
  V(full)
}
```

Consumer:

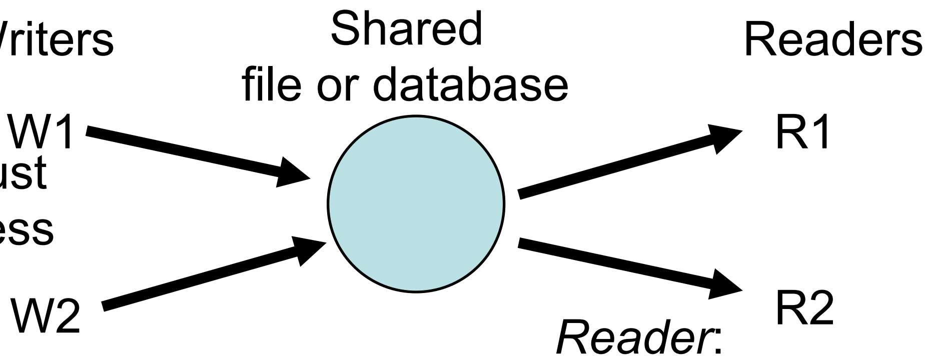
```
while(1) {
  P(full)
    P(mutex)
      remove item from buffer
    V(mutex)
  V(empty)
}
```

Achieves 1) mutual exclusion 2) blocked producer if buffer full, 3) blocked consumer if buffer empty, 4) no deadlock, and 5) is efficient (no busy wait)

University of Colorado **Boulder**

Assume $mutex_{init}$=1, $empty_{init}$=MAX, $full_{init}$=0

A Readers/Writers Solution



Goal #1: a writer must have exclusive access

Goal #2: Readers can share access with other readers

Semaphore wrt_{init}=1, $mutex_{init}$=1,
readBlock=1
int readcount = 0

Writer:

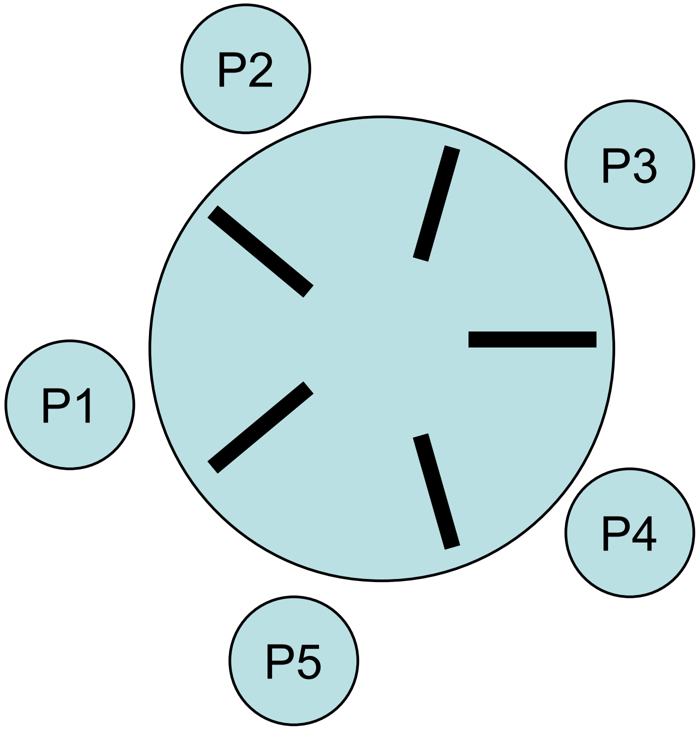
```
while(1) {
  wait(readBlock)
    wait(wrt);  // Goal 1
      // writing
    signal(wrt);
  signal(readBlock)
}
```

Reader:

```
while(1) {
    wait(readBlock)
      wait(mutex)
      readcount++;
      if (readcount==1) wait(wrt);
      signal(mutex)
    signal(readBlock)
       // reading
    wait(mutex)
    readcount--;
    if (readcount==0) signal(wrt);
    signal(mutex)
}
```

Dining Philosophers Problem

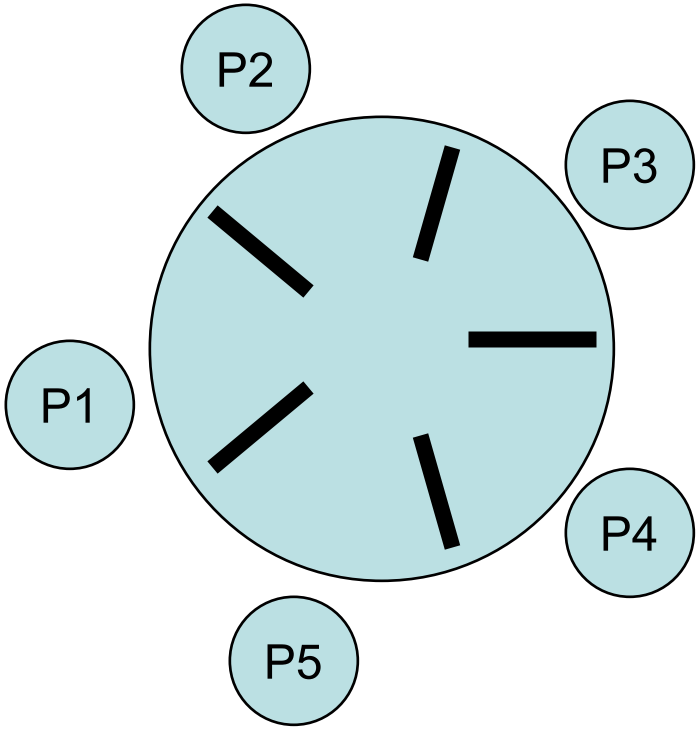
- *N* philosophers seated around a circular table
  - There is one chopstick between each philosopher
  - A philosopher must pick up its two nearest chopsticks in order to eat
  - A philosopher must pick up first one chopstick, then the second one, not both at once
- Devise an algorithm for allocating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



Dining Philosophers Problem

- Pseudo code for Philosopher i:

```
while(1) {
    // obtain 2 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]);
}
```



Deadlock!

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

Monitors

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

```
monitor monitor_name {
    // shared local variables

    function f1(...) {
    ...
    }
    ...
    function fN(...) {
    ...
    }
    init_code(...) {
    ...
    }
  }
```

- Implicitly defines mutual exclusion around each function
  - Each function's critical code is effectively:

```
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

Monitor Example

```
monitor Account {
private int balance;  // shared var requiring mutual exclusion
public function deposit(int amount) {
    balance = balance + amount;
}
public function withdraw(int amount) {
    balance = balance - amount;
}
private init() { balance=0; }
}
```

- Access to this Account monitor and its protected variable `balance` is mutually exclusive and atomic
- Only one task at a time can enter monitor/modify `balance`
  - a task X calling Account.deposit() has exclusive access to modifying `balance` until it leaves deposit().
  - Another task Y calling Account.withdraw() must wait for X to finish.

Condition Variables

- Augment the mutual exclusion of a monitor with an ordering mechanism
  - Recall: Semaphore P() and V() provide both mutual exclusion *and* ordering
  - Monitors alone only provide mutual exclusion
- A condition variable provides ordering
  - Used when one task wishes to wait until a condition is true before proceeding
    - Such as a queue being full enough or data being ready
  - A 2nd task will signal the waiting task, thereby waking up the waiting task to proceed

Condition Variables

- A condition variable *x* allows primarily two main operations on itself:
  - *x.wait()* -- suspends the calling task
    - Can have many processes blocked
    - typically released in FIFO order
    - textbook describes another variation specifying a priority p, i.e. call x.wait(p)
  - *x.signal()* -- resumes exactly 1 suspended task. If none, then *no effect*

- Declare a condition variable with pseudo-code:

```
condition x,y;
```

Condition Variables Example

Block Task 1 until a condition holds true, e.g. queue is empty

```
condition wait_until_empty;
```

Task 1:

```
wait_until_empty.wait();
…  // proceed after queue
         empty
```

Task 2:

```
…
// queue is empty so signal
wait_until_empty.signal();
```

Problem: if Task 2 signals before Task 1 waits, then Task 1 waits *forever* because CV *has no state*!

Condition Variables vs Semaphores

Both have wait() and signal(), but semaphore's signal()/V() *preserves state* in its integer value

```
Semaphore wait_until_empty=0;
```

Task 1:

```
wait(wait_until_empty);
…  // proceed after queue
         empty
```

Task 2:

```
…
// queue is empty so signal
signal(wait_until_empty); //V()
```

if Task 2 signals before Task 1 waits, then Task 1 does *not* wait forever because semaphore has state and remembers earlier signal()

Complex Conditions

- Suppose you want task T1 to wait until a complex set of conditions set by T2 becomes TRUE
  - use a condition variable

```
condition x;
int count=0;
float f=0.0;
```

T1

```
----
…
while(f!=7.0 && count<=0) {
    x.wait();
}
… // proceed
```

T2

```
----
…
f=7.0;
count++;
x.signal();
```

Problem: could be a race condition in testing and setting shared variables

Complex Conditions (2)

```
lock mutex;
condition x;
int count=0;
float f=0.0;
```

- Surround the test of the conditions with a mutex to atomically test the set of conditions

T1

```
----
…
Acquire(mutex);
while(f!=7.0 && count<=0) {
   Release(mutex);
   x.wait();
   Acquire(mutex);
}
Release(mutex);
… // proceed
```

T2

```
----
…
Acquire(mutex);
f=7.0;
count++;
Release(mutex);
x.signal();
```

Complex Conditions (3)

```
lock mutex;
condition x;
int count=0;
float f=0.0;
```

- pthreads replaces complex sequence of release/wait/acquire() with:

```
pthread_cond_wait(&cond_var,&mutex)
```

T1

```
----
…
Acquire(mutex);
while(f!=7.0 && count<=0) {
  pthread_cond_wait(&x,&mutex);
}
Release(mutex);
… // proceed
```

T2

```
----
…
Acquire(mutex);
f=7.0;
count++;
Release(mutex);
x.signal();
```

Broadcast Signals

- In some cases, you may want to wake all tasks blocked on a condition variable x, not just one
  - x.signal() only wakes one task
- *x.broadcast()* is a 3rd operation provided for CVs on some systems
  - Wakes all tasks blocked on a CV
  - In pthreads, `pthread_cond_broadcast()` wakes all waiting threads

Monitors and Condition Variables

```
monitor MON{
    condition x;
    // shared local variables

    function f1(...) {
    ...
      x.wait();
    ...
    }

    function f2(...) {
    ...
      x.signal();
    ...
    }

    init_code(...) {
    ...
    }
 }
```

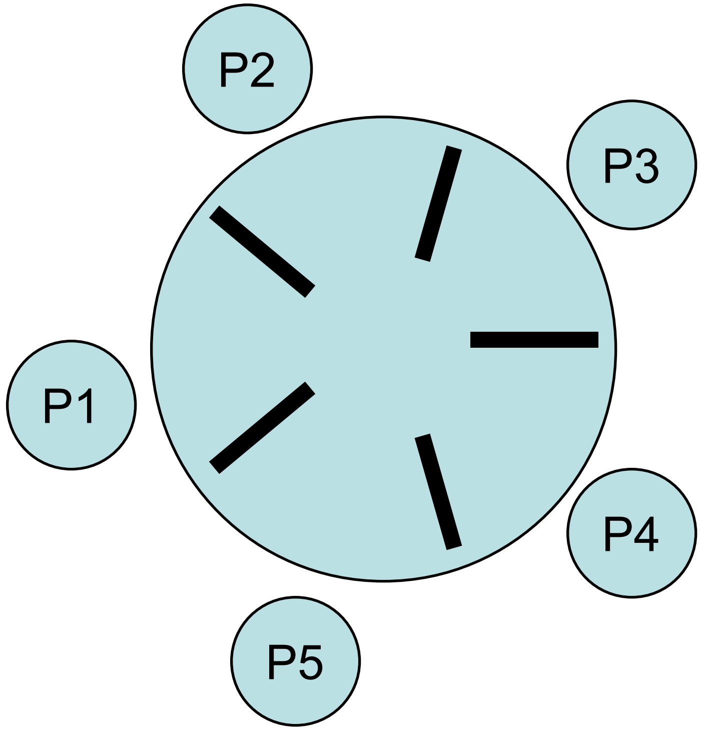
- If task T1 calls MON.f1(), this blocks T1 on x.wait()
- If task T2 then calls MON.f2(), this calls x.signal(), which unblocks T1
- Now both tasks are inside the monitor, which is a violation of mutual exclusion!
- Need a way such that only 1 task is executing in the monitor after a signal.
    - Either the signaling task, or
    - The signaled task

Hoare vs Mesa Semantics

- Hoare semantics, also called signal-and-wait
  - The signaling process P1 either waits for the woken up process P2 to leave the monitor before resuming, or waits on another CV
- Mesa semantics, also called signal-and-continue
  - The signaled process P2 waits until the signaling process P1 leaves the monitor or waits on another condition

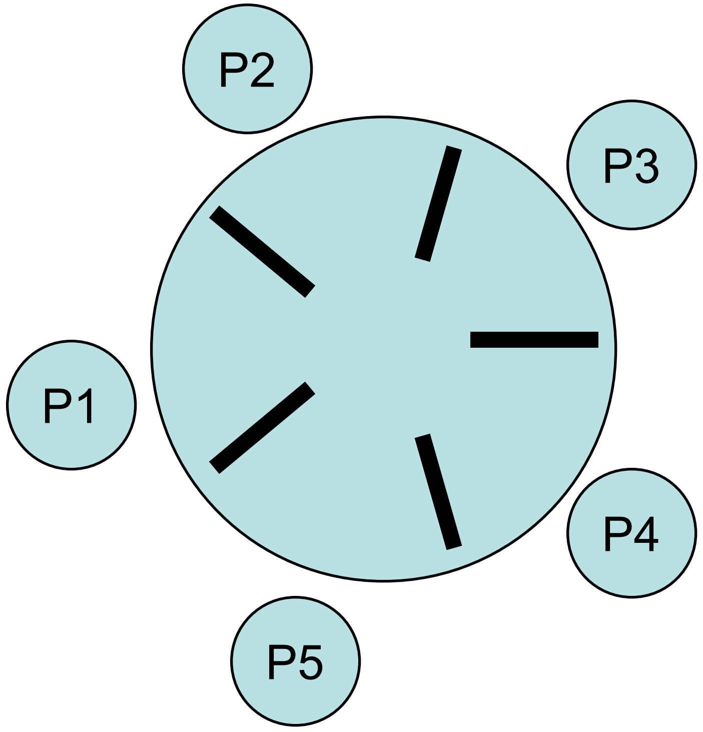
Monitor-based Solution to Dining Philosophers

- 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



Monitor-based Solution to Dining Philosophers (2)

- 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 (3)

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

- Each philosopher *i* runs pseudo-code:

```
DP.Pickup(i);
 ... // eat – grab both
        chopsticks
DP.Putdown(i);
```

Monitor-based Solution to Dining Philosophers (4)

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

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

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

         state[i] = eating;
         self[i].signal();
      }
   }
... monitor code continued next slide ...
```

atomic

atomic

- Pickup chopsticks (atomic)
  - indicate that I' m hungry
  - Atomically test if both my left and right neighbors are not eating. If so, then atomically set my state to eating.
  - if unable to eat, wait to be signaled
- signal() has no effect during Pickup(), but is important to wake up waiting hungry philosophers during Putdown()

Monitor-based Solution to Dining Philosophers (5)

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

```
  ...

Putdown(int i) {
        state[i] = thinking;
        test((i+1)%5);
        test((i-1)%5);
    }


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

}   // end of monitor
```

atomic

- Put down chopsticks (atomic)
  - if left neighbor L=(i+1)%5 is hungry and both of L's neighbors are not eating, set L's state to eating and wake it up by signaling L's CV
- Thus, eating philosophers are the ones who (eventually) turn waiting hungry neighbors into active eating philosophers
  - not all eating philosophers trigger the transformation
  - At least one eating philosophers will be the trigger

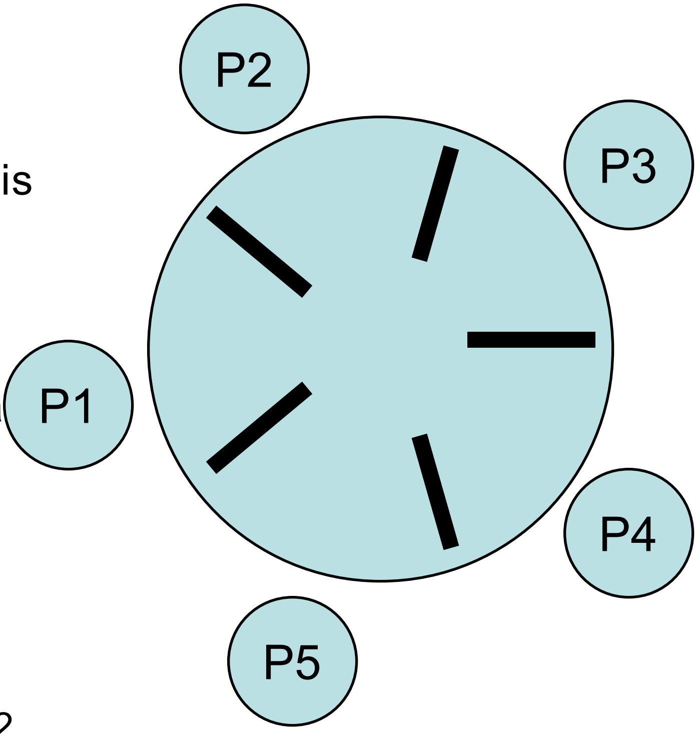
Complete Monitor-based Solution to Dining Philosophers

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

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

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

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

   Putdown(int i) {
      state[i] = thinking;
      test((i+1)%5);
      test((i-1)%5);
   }


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

}   // end of monitor
```

- 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

DP Monitor Deadlock Analysis

- Try various scenarios to verify for yourself that deadlock does not occur in them
- Start with one philosopher P1
- Now suppose P2 arrives to the left of P1 while P1 is eating
  - What is the perspective from P1?
  - What is the perspective from P2?
- Now supposes P5 arrives to the right of P1 while P1 is eating and P2 is waiting
  - Perspective from P1?
  - Perspective from P5?
  - Perspective from P2?
- Now suppose P4 arrives to the right of P5 while P2 a P5 are waiting and P1 is eating
  - Perspective from P5?
  - Perspective from P4?
  - Perspective from P1?
- Suppose P2 arrives while both P1 and P3 are eating
  - If P1 finishes first, it can't wake up P2
  - But when P3 finishes, its call to test(P2) will wake up P2, so no deadlock
- Suppose there are 6 philosophers and the evens are eating. How do the odds get to eat?



DP Monitor Solution

- Note that starvation is still possible in the DP monitor solution
  - Suppose P1 and P3 arrive first, and start eating, then P2 arrives and sets its state to hungry and blocks on its CV
  - When P1 and P3 end, they will call test(P2), but nothing will happen, i.e. P2 won't be signaled because the signal only occurs inside the if statement of test, and the if condition is not satisfied
  - Next, P1 and P3 can eat again, repeatedly, starving P2

DP Solution Analysis

- Signal() happening before the wait() doesn't matter
  - Signal() in Pickup() has no effect the 1st time
  - Signal() called in Putdown() is the actual wakeup