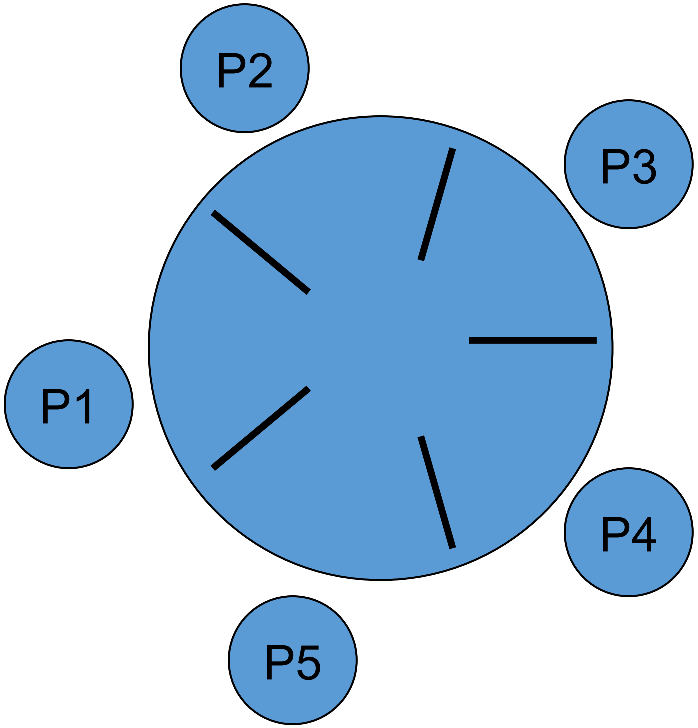
Lecture 12
More Synchronization

Dining Philosophers Problem

- **Simple algorithm for protecting access to chopsticks:**
  - **Access to each chopstick is protected by a mutual exclusion semaphore**
  - **Prevent any other philosophers from pickup the chopstick when it is already in use by a philosopher**
- **Semaphore chopstick[5];**
  - **Each philosopher grabs a chopstick I by P(chopstick[i])**
  - **Each philosopher release a chopstick I by V(chopstick[i])**



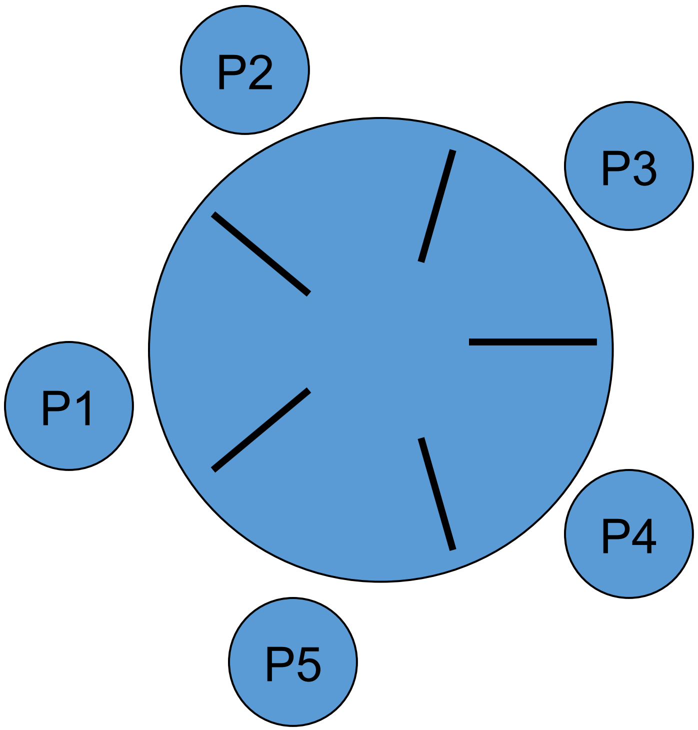
Dining Philosophers Problem

- 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]);
}
```



Guarantees that no two neighbors eat simultaneously, since a chopstick can only be used by one its two neighboring philosophers

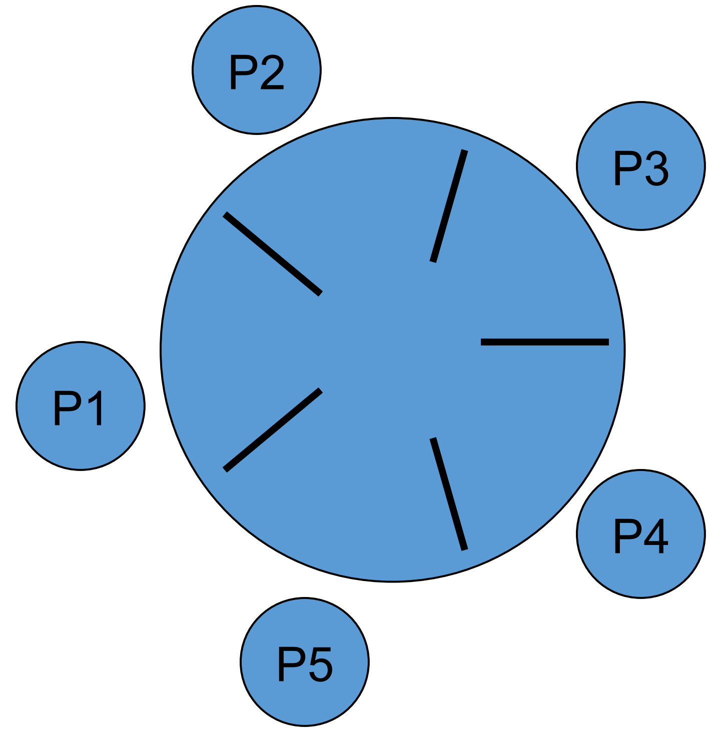
Dining Philosophers Problem

- 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]);
}
```



Problem?

Guarantees that no two neighbors eat simultaneously, since a chopstick can only be used by one its two neighboring philosophers

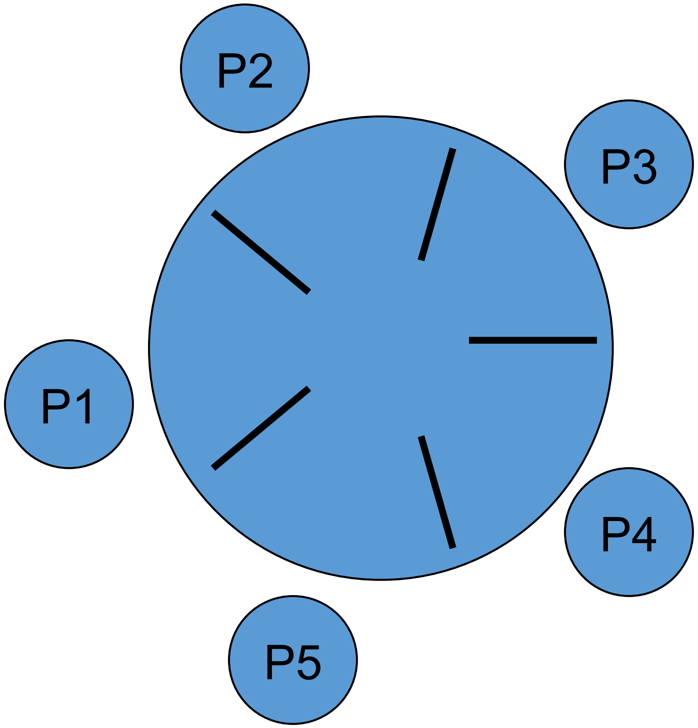
Dining Philosophers Problem

- 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

If all philosophers pick a up a single chopstick, then they all wait for a second chopstick. We will study **circular wait** next week.

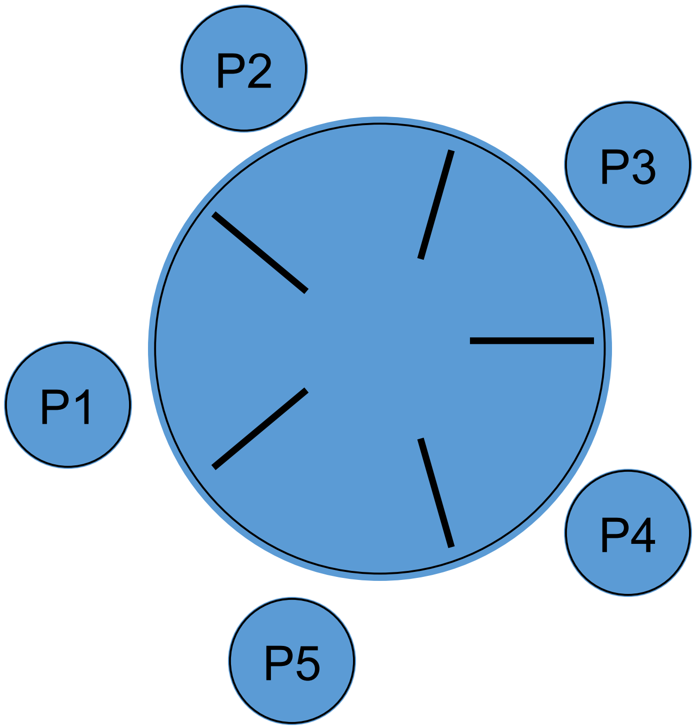
Dining Philosophers Problem

- 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

Monitor-based Solution to Dining Philosophers

1st insight: Pick up 2 chopsticks only if both are free

- This avoids deadlock
- A philosopher begins eating only if both neighbors are not currently eating
- Need to define a state for each philosopher



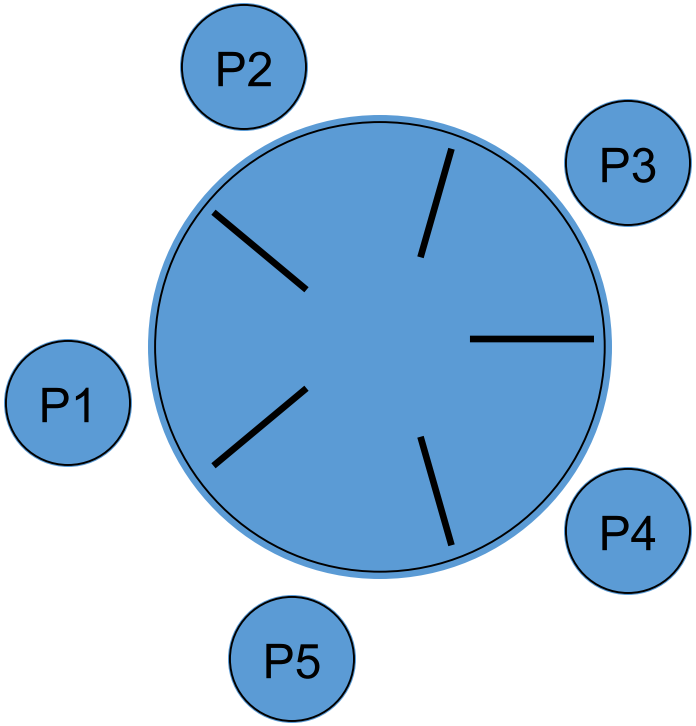
Monitor-based Solution to Dining Philosophers

1st insight: Pick up 2 chopsticks only if both are free

- This avoids deadlock
- A philosopher begins eating only if both neighbors are not currently eating
- 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

```
monitor DP
{
    enum {thinking, hungry, eating} state[5];
    condition self[5]; //to block a philosopher when hungry

    void pickup(int i) {
        //Set state[i] to hungry
        //If at least one neighbor is eating,  (test the neighbors)
            //            block on self[i]
        //Otherwise return
    }

    void putdown(int i) {
        //Change state[i] to thinking and signal neighbors
          //    in case they are waiting to eat
    }
```

Monitor-based Solution to Dining Philosophers

```
void test(int i) {
   //Check if both neighbors of i are not eating
        // and i is hungry
   //If so,

        //     set state[i] to eating //     and
       signal philosopher i

}


init( ) {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;

}
}
```

Monitor-based Solution to Dining Philosophers

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

```
Philosopher(int i) {
  While(1) {
    // thinking
    // hungry
    DP.Pickup(i);
        // eating
    DP.Putdown(i);
  }
}
```

Monitor-based Solution to Dining Philosophers

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

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

  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);
  }
}
```

All philosophers are thinking, initially

Indicate that a philosopher is hungry, then test if both left and right neighbors are not eating. If so, then set the philosophers state to eating

If unable to eat, wait to be signaled

If philosopher *i* is hungry and both of *i*'s neighbors are not eating, set *i*'s state to eating and wake it up by signaling *i*'s CV

Signal() has no effect during Pickup(), but is important to wake up waiting hungry philosophers during Putdown()

Philosophers who are done eating eventually turn hungry neighbors into eating philosophers

Monitor-based Solution to Dining Philosophers

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

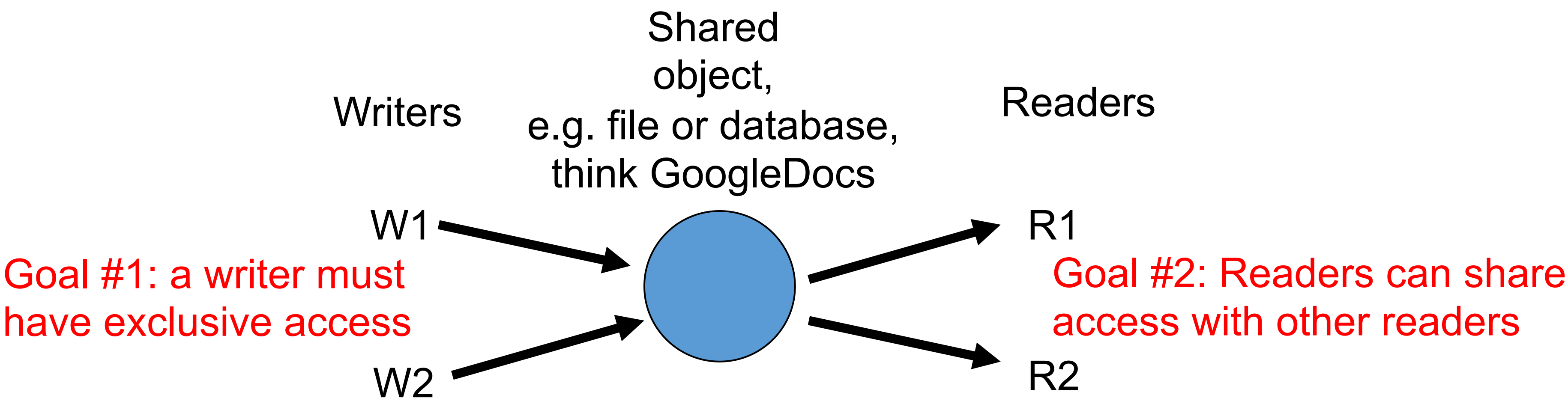
Monitor-based Solution to Dining Philosophers

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 Condition Variable (CV)
- When P1 ends eating, it 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 can eat again, repeatedly, starving P2

The Readers/Writers Problem

- N tasks want to write to a shared file
- M other tasks want to read from same shared file
- Must synchronize access

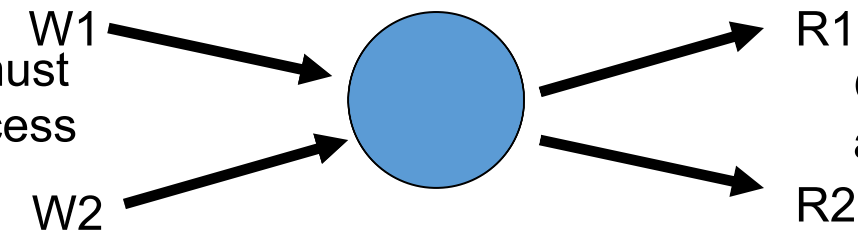


The Readers/Writers Problem

- Additional requirement #1:
  - no reader is kept waiting unless a writer already has seized the shared object
- Additional requirement #2:
  - a pending writer should not be kept waiting indefinitely by readers that arrived after the writer
  - i.e. a pending writer cannot starve

1st Readers/Writers Solution

Assume wrt_{init}=1 is a mutex lock/binary semaphore



Goal #1: a writer must have exclusive access

Goal #2: Readers can share access with other readers

Writer:

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

Reader:

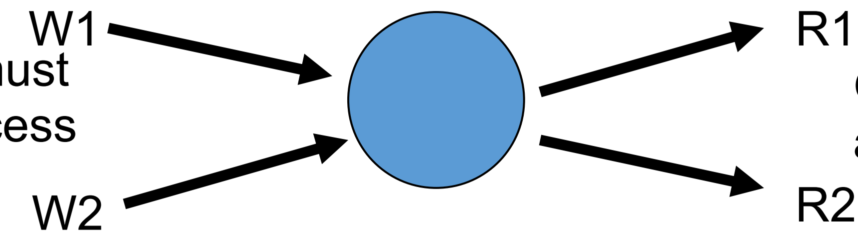
```
while(1) {
    ...
    wait(wrt);  // Goal 1 but not 2
      // reading
    signal(wrt);
    ...
}
```

Problem: first reader grabs lock, preventing other readers (& writers)
Solution: only the first reader needs to grab the lock, and last reader release the lock.

1st Readers/Writers Solution

Assume wrt_{init}=1, $readcount_{init}$ = 0

Writers Shared file or database Readers



Goal #1: a writer must have exclusive access

Goal #2: Readers can share access with other readers

Writer:

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

Reader:

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

Problem: both readcount++ and readcount-- lead to race conditions
Solution: surround access to readcount with a 2nd mutex

1st Readers/Writers Solution

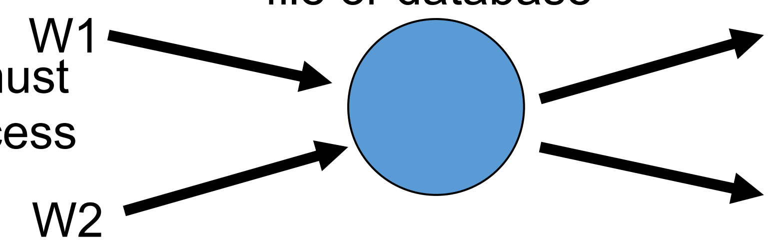
Assume wrt_{init}=1, $readcount_{init}$ = 0, $mutex_{init}$=1

Writers

Shared
file or database

Readers

Goal #1: a writer must have exclusive access



Goal #2: Readers can share access with other readers

Writer:

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

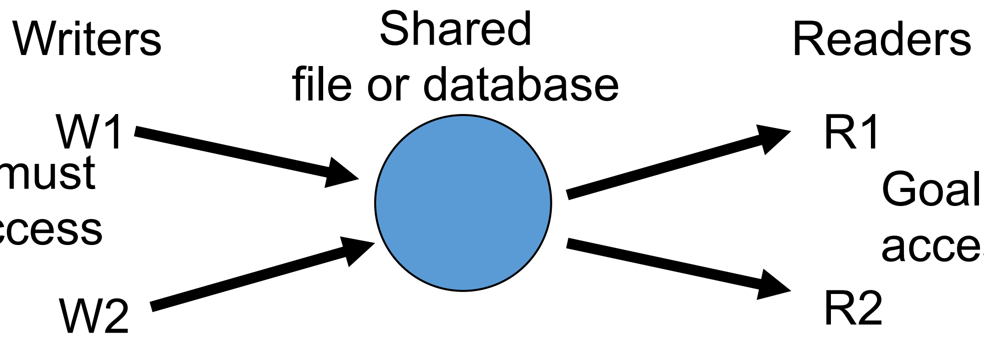
Reader:

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

So a writer excludes other writers and readers.
Multiple readers are allowed and exclude writers while at least 1 reader

1^{st} Readers/Writers Solution

Assume $wrt_{init}=1$, $readcount_{init} = 0$, $mutex_{init}=1$



Goal #1: a writer must have exclusive access

Goal #2: Readers can share access with other readers

Writer:

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

Reader:

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

Problem: this solution could starve pending writers!

2^{nd} Readers/Writers Solution

A pending writer should not be kept waiting indefinitely by readers that arrived after the writer

- 1^{st} R/W solution gave precedence to readers
  - new readers can keep arriving while any one reader holds the write lock, which can starve writers until the last reader is finished

- Instead, allow a pending writer to block future reads
  - This way, writers don't starve.
  - If there is a writer,
    - New readers should block
    - Existing readers should finish then signal the waiting writer

2nd Readers/Writers Solution

```
int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1,
          writePending = 1;
writer() {
  while(TRUE) {

    P(mutex2);
      writeCount++;
      if(writeCount == 1)
        P(readBlock);
    V(mutex2);

    P(writeBlock);
      write(resource);
    V(writeBlock);

    P(mutex2)
      writeCount--;
      if(writeCount == 0)
        V(readBlock);
    V(mutex2);
  }
}
```

```
reader() {
  while(TRUE) {
    P(writePending);
      P(readBlock);
        P(mutex1);
          readCount++;
          if(readCount == 1)
            P(writeBlock);
        V(mutex1);
      V(readBlock);
    V(writePending);

    read(resource);

    P(mutex1);
      readCount--;
      if(readCount == 0)
        V(writeBlock);
    V(mutex1);
  }
}
```

Red = changed from1st R/W problem solution

Copyright © 2004 Pearson Education, Inc.

2nd Readers/Writers Solution

- Once 1st writer grabs readBlock,
  - any number of writers can come through while the 1st reader is blocked on readBlock
  - and subsequent readers are blocked on writePending
  - So, behavior is that a writer can block not just new readers, but also some earlier readers
  - Note now that readers can be starved!
- Instead, want a solution that is starvation-free for both readers and writers

Starvation-free 2nd Readers/Writers Solution

int $readcount_{init} = 0$, $readBlock_{init} = 1$

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)
}
```

This is a starvation-free solution
Note how it is a minor variant of the 1st R/W solution.