Chapter 5: Classic Synchronization Problems

CSCI 3753 Operating Systems

Instructor: Chris Womack

University of Colorado at Boulder

All material by Dr. Rick Han

Announcements

- PA1 & PS1 due **Today**
  - Add a system call to the linux kernel
- PA2 & PS2 have been released
  - Add a device driver to Linux using kernel modules
  - Both are due June 26th

- Reading: Chapter 5
  - Next week 5 & 7

Recap

- Mutual exclusion/synchronization
  - Disabling interrupts in critical section
  - Locks: Acquire(lock) and Release(lock)
    - Disable interrupts within Acquire() so test-and-set is atomic
    - spinlock implementation
    - Test-and-Set implementation
  - Semaphores have integer state, unlike locks
    - P()/wait() atomically decrements and blocks if necessary
    - V()/signal() atomically increments and wakes if necessary

A Revised Semaphore Definition

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

atomic

```
P(semaphore *S) {
    S->value--;
    if (S->value<0) {
        add this process
            to S->list;
        block();
    }
}
```

atomic

```
V(semaphore *S) {
    S->value++;
    if (S->value<=0) {
        remove a process P
            from S->list;
        wakeup(P);
    }
}
```

- Efficiently sleep the process until it needs to be woken up by a V()/signal(), rather than spinlock

Mutual Exclusion with Semaphores

```
Semaphore S = 1;    // initial value of semaphore is 1
int counter;        // assume counter is set correctly somewhere in code
```

Process P1:

```
P(S);
   // execute crit sect
    counter++;
V(S);
```

Process P2:

```
P(S);
   // execute crit sect
    counter--;
V(S);
```

- Both processes atomically P() and V() the semaphore S, which enables mutual exclusion on critical section code, in this case protecting access to the shared variable counter

Recap

- Enforcing order with Semaphores
- Deadlock
  - Circular dependency:
    - Task P1 holds a lock/semaphore and P2 holds another lock/semaphore.
    - Each process then tries to get the other task's lock/semaphore without releasing its own
  - Other examples where a programmer
    - Forgets to V()
    - Calls P() instead of V()

Deadlock Example

```
Semaphore Q= 1;     // binary semaphore as a mutex lock
Semaphore S = 1;    // binary semaphore as a mutex lock
variable R1, R2;
```

Process P1:

```
P(S);                  (step 1)
P(Q);                  (step 3)
    modify R1 and R2;
V(S);
V(Q);
```

Process P2:

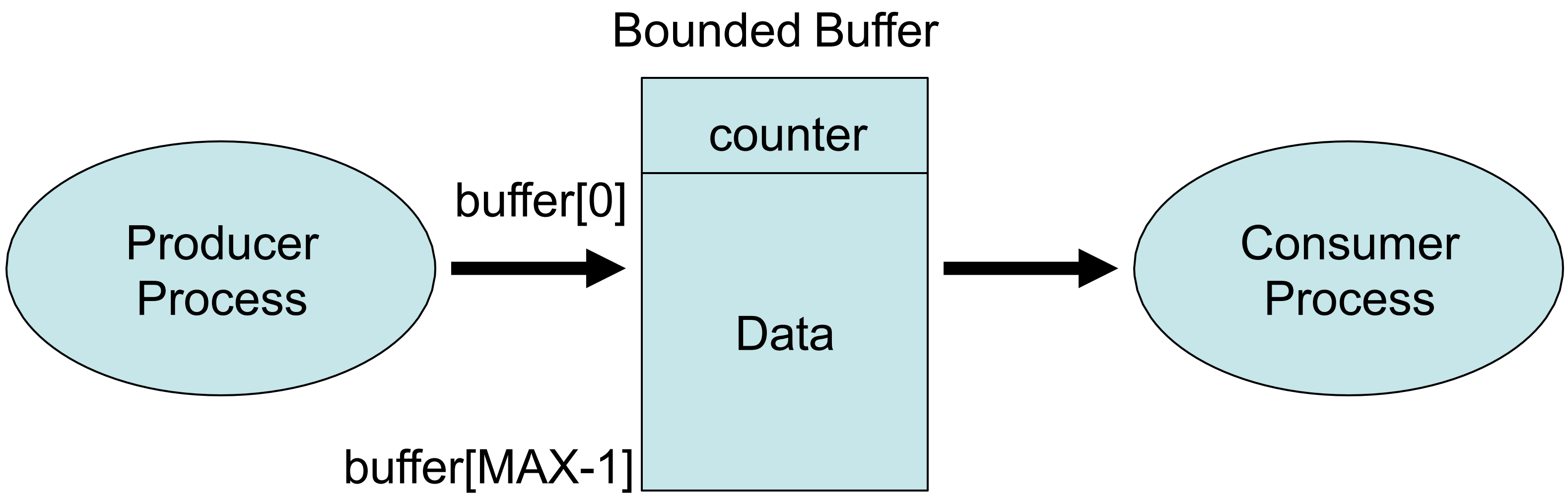
```
(step 2) P(Q);
(step 4) P(S);
Deadlock!    modify R1 and R2;
         V(Q);
         V(S);
```

If steps (1) through (4) are executed in that order, then P1 and P2 will be deadlocked after statement (4) - verify this for yourself by stepping thru the semaphore values

Classic Synchronization Problems

- Bounded Buffer Producer-Consumer Problem
- Readers-Writers Problem
  - First Readers Problem
- Dining Philosophers Problem
- These are not just abstract problems
  - Represent classes of synchronization problems encountered in the real world when trying to synchronize access to shared resources among multiple processes or threads

Prior Bounded-Buffer P/C Approach




```
while(1) {
    while(counter==MAX);
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    Acquire(lock);
    counter++;
    Release(lock);
}
```

Busy-wait!

```
while(1) {
    while(counter==0);
    getdata = buffer[out];
    out = (out+1) % MAX;
    Acquire(lock);
    counter--;
    Release(lock);
}
```

Bounded-Buffer P/C Goals

- In the prior approach, both the producer and consumer are busy-waiting
- Instead, want both to sleep as necessary
  - Goal #1: Producer should block when buffer is full
  - Goal #2: Consumer should block when the buffer is empty
- Also, Goal #3: mutual exclusion when buffer is partially full
  - Producer and consumer should access the buffer in a synchronized mutually exclusive way

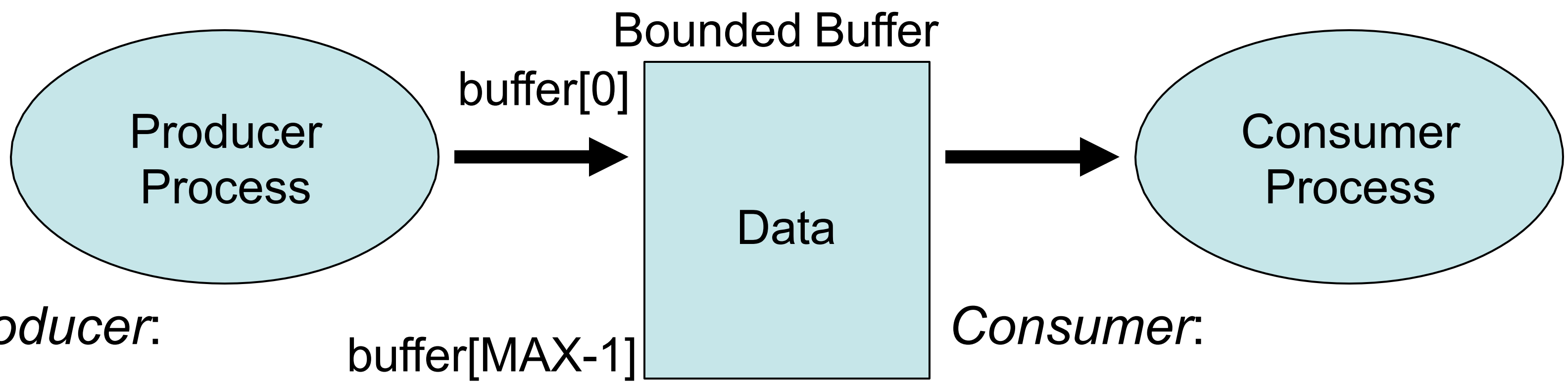
Bounded-Buffer P/C Design

- To achieve Goal #3 of mutual exclusion:
  – Use a mutex semaphore to protect access to buffer manipulation, $mutex_{init} = 1$
- To achieve Goal #1, producer blocks if buffer full:
  – Use a counting semaphore called *empty* that is initialized to $empty_{init} = MAX$
  – Each time the producer adds an object to the buffer, this decrements the # of empty slots, until it hits 0 and the producer blocks

Bounded-Buffer P/C Design

- To achieve Goal #2, consumer blocks if buffer empty:
  - Define a counting semaphore *full* that is initialized to $full_{init} = 0$
  - *full* tracks the # of full slots and is incremented by the producer
  - Each time the consumer removes a full slot, this decrements *full*, until it hits 0, then the consumer blocks

Bounded Buffer P/C Solution



Producer:

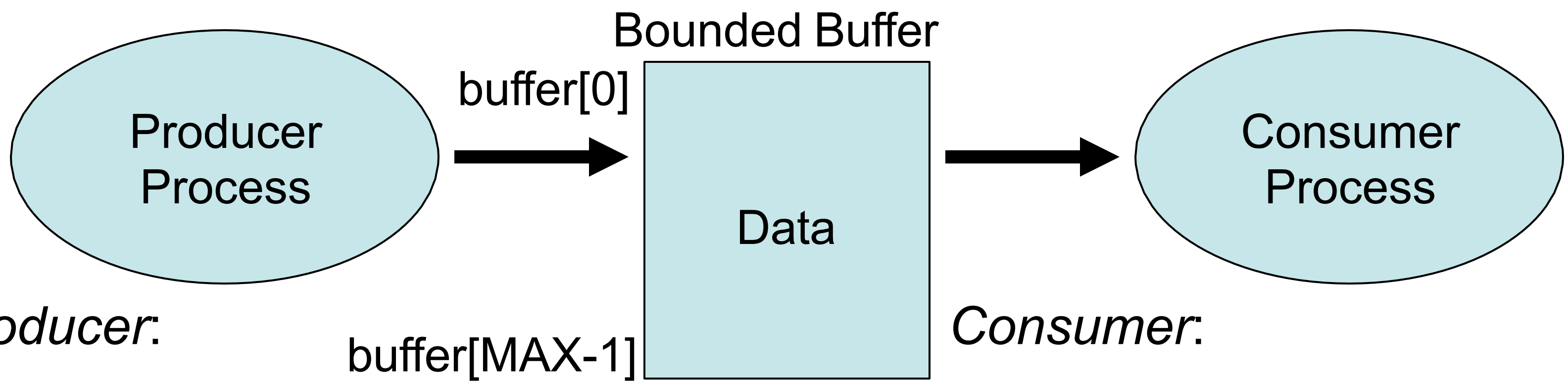
```
while(1) {
    …
    P(mutex)   // Goal 3: mutual
                  exclusion
       add item to buffer
    V(mutex)
    …
}
```

Consumer:

```
while(1) {
    ...
    P(mutex)   // Goal 3: mutual
                  exclusion
       remove item from buffer
    V(mutex)
    ...
}
```


Assume $mutex_{init}$=1

Bounded Buffer P/C Solution



Producer:

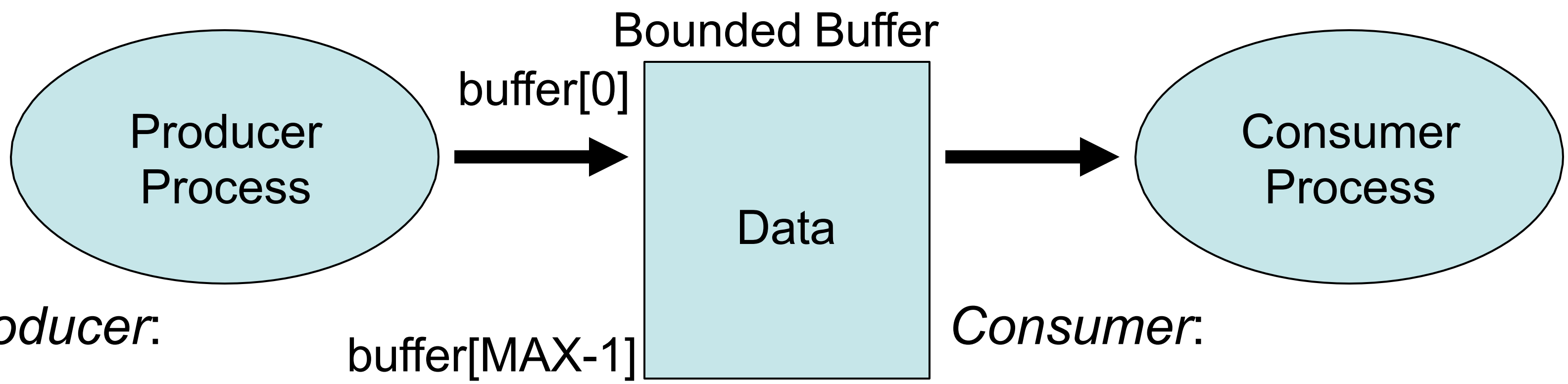
```
while(1) {
  P(empty) // Goal 1: full buffer
              blocks producer
    P(mutex)
       add item to buffer
    V(mutex)
    …
}
```

Consumer:

```
while(1) {
    ...
    P(mutex)
       remove item from buffer
    V(mutex)
  V(empty) // Goal 1: full buff
              blocks producer
}
```

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

Bounded Buffer P/C Solution



Producer:

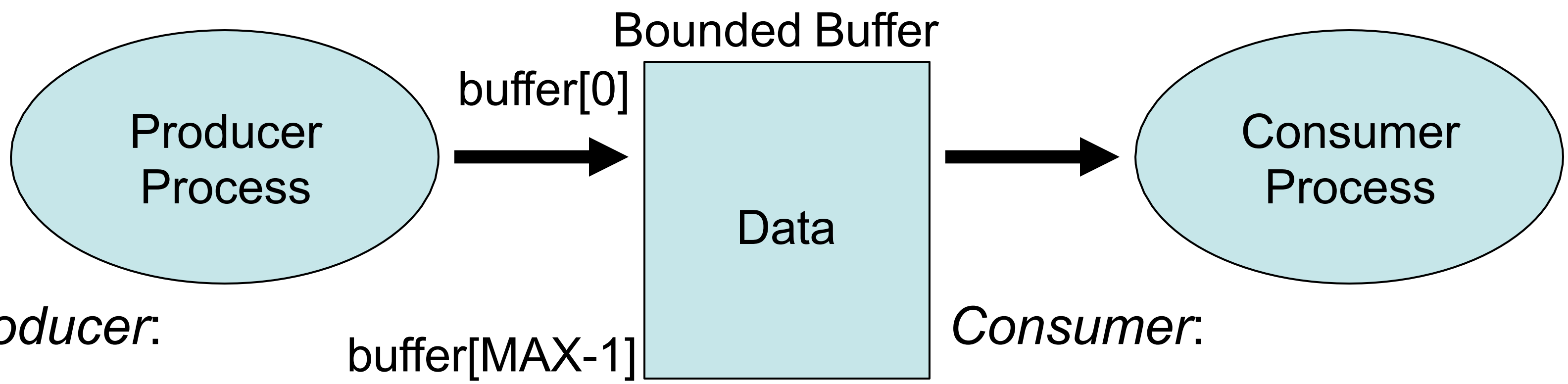
```
while(1) {
  P(empty)
    P(mutex)
      add item to buffer
    V(mutex)
  V(full) // Goal 2: empty buffer
             blocks consumer
}
```

Consumer:

```
while(1) {
  P(full) // Goal 2: empty
                buff blocks consumer
    P(mutex)
      remove item from buffer
    V(mutex)
  V(empty)
}
```


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

Bounded Buffer P/C Solution



Producer:

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

Consumer:

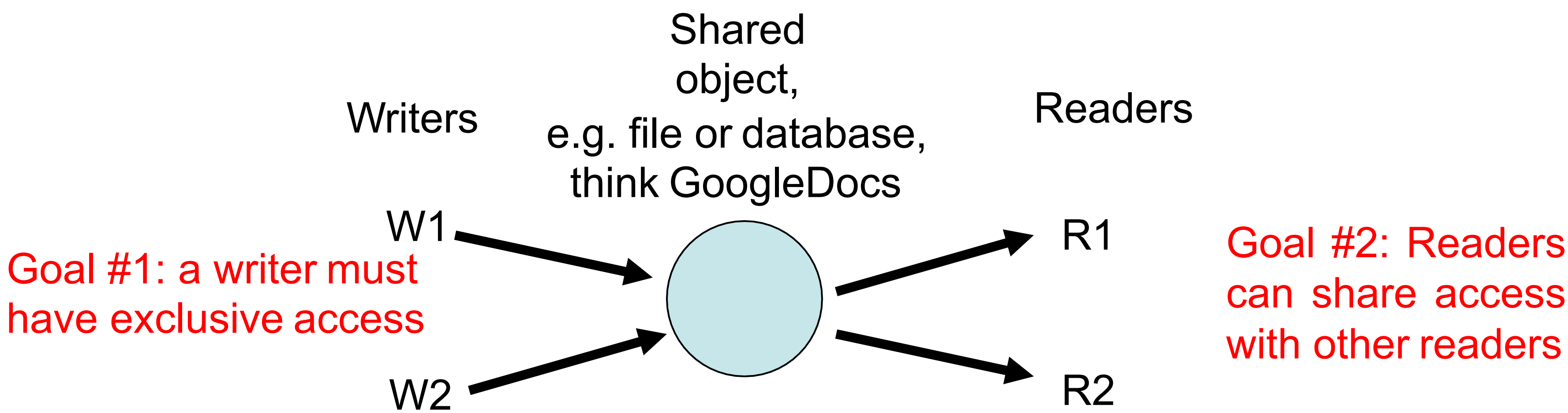
```
while(1) {
  P(full)
    P(mutex)
      remove item from buffer
    V(mutex)
  V(empty)
}
```

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)

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

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



- Goal #2: should support multiple concurrent readers
  - Hence, a single mutex lock won't support that

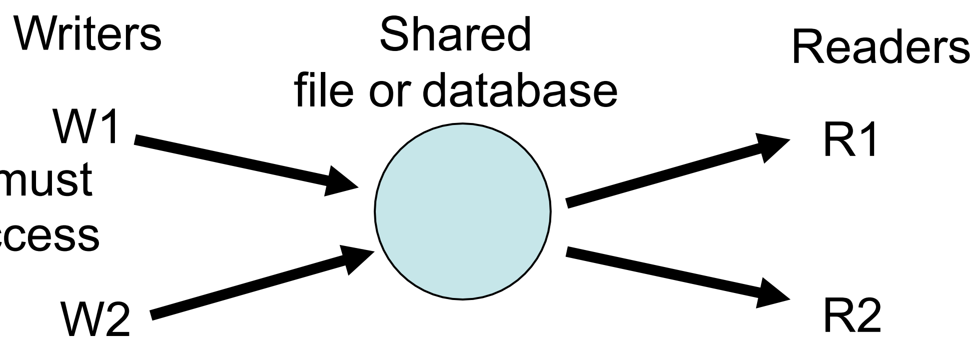
R/W vs P/C Comparison

	Readers/Writers	**BB Producer/ Consumer**
# of Tasks	N Writers, M Readers	1 producer, 1 consumer
Amount of Data	One shared data object	D data objects in buffer
Exclusion	• A writer excludes all other writers and readers. • A reader allows other readers but excludes writers (depending on formulation).	A producer excludes a consumer, and vice versa

1st and 2nd Readers/Writers Problems

- 1st R/W Problem:
  - Clarification to Goal #2: no reader is kept waiting unless a writer already has seized the shared object.
- 2nd R/W Problem:
  - Caveat to 1st R/W: 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



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:

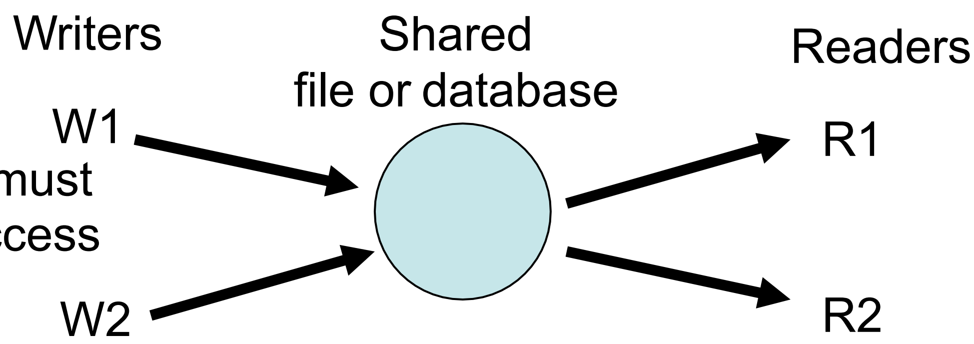
```
while(1) {
     ...
     wait(wrt);   // Goal 1 but not 2
       // reading
     signal(wrt);
     ...
}
```

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

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

1st Readers/Writers Solution

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);
}
```

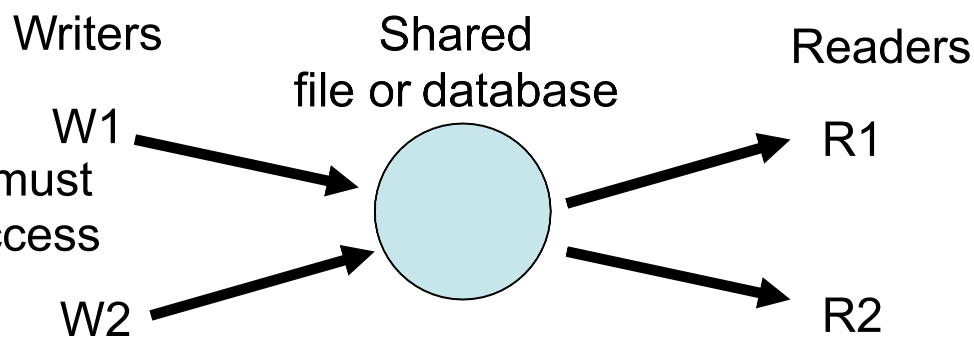
Reader:

```
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

Assume wrt_{init}=1, readcount initialized = 0

1st Readers/Writers Solution



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);
}
```

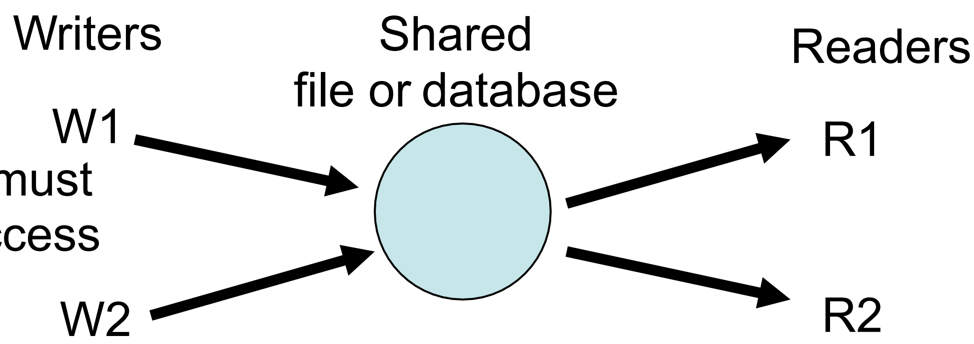
Reader:

```
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

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

1^{st} Readers/Writers Solution



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!

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

2nd Readers/Writers Problem

- Note that the 1st R/W problem 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

Original Solution to the 2nd Readers/Writers Problem

```
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 from 1st R/W problem

2nd Readers/Writers Starvation

- Once 1st writer grabs readBlock,
  - any number of writers can come through while the 1st reader is blocked on redBlock
  - 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 Solution to 2nd R/W

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)
}
```

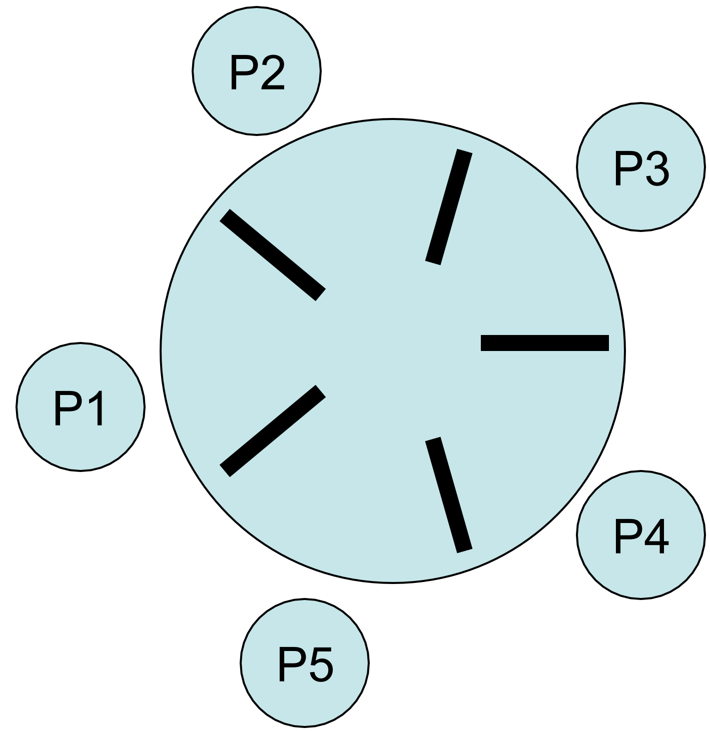
Reader:

```
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)
}
```

This is starvation-free solution is sometimes called the solution to the 3rd R/W problem. Note how it is a minor variant of the 1st R/W solution.

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:
  - Access to each chopstick is protected by a mutual exclusion semaphore
  - prevents any other philosopher from picking up the chopstick when it is already in use by a philosopher

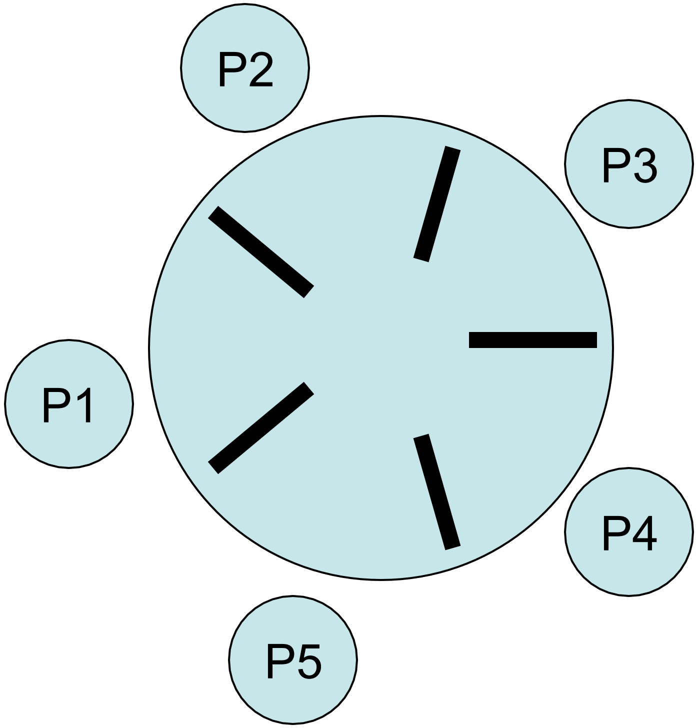
- `semaphore chopstick[5];` // initialized to 1

  - 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 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, 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

- 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
    - Also, there is a construct called an AND semaphore
- A deadlock-free solution is not necessarily starvation-free
  - for now, we’ll focus on breaking deadlock

Monitors

- 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*,
  - 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

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

```
function fj(...) {
    P(mutex)
    // critical code
    V(mutex)
}
```

Monitors

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

```
monitor monitor_name {
    // shared local variables

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

- 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

Supplementary Slides

Details on the 2nd R/W Problem

- Comparing the solution of the 2nd R/W problem to the solution for the 1st R/W problem:
  - The reader has not changed much from the 1st R/W problem, just adding logic to block new readers if there's a pending writer
  - Writer has changed substantially, but it resembles the reader in the 1st R/W problem, i.e. the first writer that arrives blocks all future readers, multiple writers are allowed in (but synchronized one at a time using writeBlock), and the last writer out starts activating the readers
- Scenario for reasoning through the solution:
  - suppose there are multiple "current" readers - they could all be in read(resource)
  - then the 1st writer arrives.
    - This 1st writer blocks future reads by setting readBlock, then blocks on writeBlock, waiting for current readers to finish
    - Subsequent writers also block on writeBlock.
  - Then a new reader arrives - it blocks on readBlock.
    - All subsequent new readers will block on writePending.
  - Once the current readers finish, the last current reader will awaken the 1st writer on writeBlock by V'ing writeBlock
  - Multiple writers can now arrive and be serviced in synchronized order
    - they will continue to block all new readers – starving new readers…
    - … until the last writer finishes, and V's readBlock, releasing the 1st new reader, who will V(writePending) and release the 2nd new reader, etc., eventually freeing up all multiple readers so that they can again execute in read(resource)

Details on the 2nd R/W Problem

- writePending is an optimization in the previous example, because without it, all readers would block on readBlock, and a new writer would not be able to quickly gain access to the shared file over pending Readers
  - Want the behavior to be that if a new writer comes along, that stops all future reads, *including* currently blocked readers who have not yet entered their reading critical sections
  - Without writePending, all readers would block on readBlock, waiting for a writer to finish. Once that writer finishes, the 1st reader proceeds into its critical section. If another writer comes in before the 2nd queued reader can get started, then want writer to block the 2nd reader. Without writePending, the 1st reader will wake up the 2nd reader blocked on readBlock, who will then wake up the 3rd reader blocked on readBlock, emptying readBlock before the writer (who is blocked on readBlock but is at the end of the FIFO sleep queue) gets access to its critical section
  - Instead, want a new writer to stop queued Readers from proceeding. With writePending, if there are multiple readers waiting for writer #1 to finish, then 1st reader blocks on readBlock, and the 2nd reader, and 3rd, 4th, … will all block on writePending. When writer #1 is finished, reader #1 proceeds, and let's say reader #2 as well. But if writer #2 arrives, it will grab readBlock, which will block reader #3 from proceeding, i.e. reader #3 will block on readBlock, while readers #4, #5, … are still stuck on writePending. In this way, writer #2 is able to proceed more quickly to gain access the shared file.