Department of Physics,
Computer Science & Engineering

CPSC 410 – Operating Systems I

Chapter 5:Concurrency
Mutual Exclusion & Synchronization

Keith Perkins

Adapted from original slides by Dr. Roberto A. Flores

Chapter 5 Topics

- Principles of Concurrency
  - Terminology, Process interaction
- Mutual Exclusion
  - Race condition, Requirements, Hardware approaches
- Semaphores
  - Definition, Implementation, Producer/Consumer
- Monitors
  - Definition, Producer/Consumer (revisited), Implementation
- Message Passing
  - Synchronization, Addressing, Mailboxes
- Readers/Writers Problem
  - Light switch pattern

Concurrency

- Definition (ad hoc)
  - The convergence of multiple actors upon a item
- OS design themes
  - Managing processes & processors
    - Multiprogramming: N processes, 1 processor (N:1)
    - Multiprocessing: N processes, M processors (N:M, N>M)
    - Distributed processing: Multiprocessing across computers
- 2+ processes running concurrently by…
  - Interleaving: processes alternate using a processor
    - Only way in uni-processors; multiprocessors can use it
  - Overlapping: processes use different processors
    - Only available in multiprocessors

Concurrency

- Normally OS…
  - keep track of processes
  - allocating & manage their resources
- …but with concurrency
  - protect resources against interference by other processes
  - ensure that processes & their outputs are independent of processing speed

  - In multiprogramming
    - a process relative speed of execution depends
      - on other processes activities
      - OS interrupt handling & scheduling policies

Terminology

- Atomic operation
  - Code segment whose execution is indivisible: once started it cannot be interrupted nor its state observed until it ends.
- Critical section
  - Code segment accessing a resource. It must be accessed by one thread at a time to maintain the integrity of the resource.
- Mutual exclusion
  - Requirement that only 1 thread accesses a critical section at a time.
- Race condition
  - When 2+ threads access a resource and their final result varies depending on the timing of their execution.
- Deadlock
  - When 2+ processes cannot proceed because each is waiting for the others to complete their work.
- Livelock
  - When 2+ processes repetitively change their state to adjust to changes in state of the others, without any advancing their work (e.g., corridor)
- Starvation
  - When a waiting thread is overlooked by the scheduler and never given a chance to proceed (e.g., lower priority).

Process interaction

- Degree of awareness (among projects)
  - Unaware: Competition (not working together)
    - Results obtained independently of others
  - Indirectly aware: Cooperation by sharing (indirect)
    - Results obtained by observing results of others
  - Directly aware: Cooperation by communication (direct)
    - Results obtained by communicating with others
- All require mechanisms addressing
  - Mutual exclusion (integrity), Deadlock & Starvation

Chapter 5 Topics

- Principles of Concurrency
  - Terminology, Process interaction
- Mutual Exclusion
  - Race condition, Requirements, Hardware approaches
- Semaphores
  - Definition, Implementation, Producer/Consumer
- Monitors
  - Definition, Producer/Consumer (revisited), Implementation
- Message Passing
  - Synchronization, Addressing, Mailboxes
- Readers/Writers Problem
  - Light switch pattern

Mutual Exclusion

- Race condition
  - When multiple threads read/write same data items
  - The final result depends on the order of execution
    - last process updating determines final value of variable

```
void withdrawal(int amount) {
    if (balance > amount) {
        System.out.println( "withdrawal approved" );
        balance = balance – amount;
    }
}
```

Thread 1 – Withdraw $40

Balance $50

Mutual Exclusion

- Race condition
  - When multiple threads read/write same data items
  - The final result depends on the order of execution
    - last process updating determines final value of variable

```
void withdrawal(int amount) {
   if (balance > amount) {
      System.out.println( “withdrawal approved” );
      balance = balance – amount;
   }
}
```

Thread 2 – Withdraw $25 zzz

Thread 1 – Withdraw $40

Balance $50

Mutual Exclusion

- Race condition
  - When multiple threads read/write same data items
  - The final result depends on the order of execution
    - last process updating determines final value of variable

```
void withdrawal(int amount) {
    if (balance > amount) {
        System.out.println( "withdrawal approved" );
        balance = balance – amount;
    }
}
```

Thread 2 – Withdraw $25

Thread 1 – Withdraw $40 zzz

Balance $50

Mutual Exclusion

- Race condition
  - When multiple threads read/write same data items
  - The final result depends on the order of execution
    - last process updating determines final value of variable

```
void withdrawal(int amount) {
   if (balance > amount) {
      System.out.println( “withdrawal approved” );
      balance = balance – amount;
   }
}
```

Thread 1 – Withdraw $40 zzz

Thread 2 – Withdraw $25

Balance $25

Mutual Exclusion

- Race condition
  - When multiple threads read/write same data items
  - The final result depends on the order of execution
    - last process updating determines final value of variable

```
void withdrawal(int amount) {
   if (balance > amount) {
      System.out.println( “withdrawal approved” );
      balance = balance – amount;
   }
}
```

Thread 1 – Withdraw $40

Balance $25

Mutual Exclusion

- Race condition
  - When multiple threads read/write same data items
  - The final result depends on the order of execution
    - last process updating determines final value of variable

```
void withdrawal(int amount) {
   if (balance > amount) {
      System.out.println( "withdrawal approved" );
      balance = balance – amount;
   }
}
```

Thread 1 – Withdraw $40

Balance -$15

huh?

Mutual Exclusion

- When is it needed?
  - when executing a critical section
    - …which is the code segment where a shared resource is used

```
void P_1 {
   while (true) {
      // non-critical
      enterCritical( resource );
      // critical section
      exitCritical( resource );
      // non-critical
   }
}
```

…

```
void P_N {
   while (true) {
      // non-critical
      enterCritical( resource );
      // critical section
      exitCritical( resource );
      // non-critical
   }
}
```

Mutual Exclusion

- When is it needed?
  - when executing a critical section
    - …which is the code segment where a shared resource is used

```
void P1 {
    while (true) {
        // non-critical
        enterCritical( resource );
        // critical section
        exitCritical( resource );
        // non-critical
    }
}
```

…

```
void PN {
    while (true) {
        // non-critical
        enterCritical( resource );
        // critical section
        exitCritical( resource );
        // non-critical
    }
}
```

need mutual exclusion!

Mutual Exclusion

- Requirements
  - Enforcement
    - only 1 thread can be in the critical section of a shared object.
  - Availability
    - if no thread is in a critical section then any thread can enter.
  - Minimal permanence
    - threads stay in a critical section for a minimal, finite time only.
  - Liveness (no deadlock/starvation)
    - mechanisms exist to avoid indefinite delays to access a resource.
  - No side-effects (from non-critical sections)
    - Halting in a non-critical section does not impact other processes.
  - Unpredictability
    - No assumptions about process speeds or number of processors.

Mutual Exclusion

- In Hardware
  - 1. Interrupt Disabling
    - disabling interrupts guarantees mutual exclusion
    - disadvantages
      - efficiency of execution degrades noticeably
        - independent critical sections cannot overlap (e.g. printer, disk)
        - would not work in a multiprocessor

Mutual Exclusion

- In Hardware
  - 2. Special (atomic) Instructions
    - Strategy
      - use a flag to indicate whether a thread is in the critical section
      - if flag is on, then critical section is taken; wait until available.
      - if flag is off, then set it on and execute the critical section; set flag off when exiting the critical section (to allow others to enter)
    - Implementations
      - Compare & Swap
      - Exchange

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka “compare & exchange”)
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

```
int flag = 0;
void withdrawal(int amount) {
    while (cswap( flag, 0, 1 ) == 1) { }
    if (balance > amount) {
        System.out.println( “withdrawal approved” );
        balance = balance – amount;
    }
    flag = 0;
}
```

compare

swap

Busy waiting

Busy waiting (technique)
a thread does nothing until it gets permission to enter its critical section

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka “compare & exchange”)
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

```
int flag = 0;
void withdrawal(int amount) {
    while (cswap( flag, 0, 1 ) == 1) { }
    if (balance > amount) {
        System.out.println( “withdrawal approved” );
        balance = balance – amount;
    }
    flag = 0;
}
```

flag 0 Balance $50

Thread 1 – Withdraw $40

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka “compare & exchange”)
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

```
int flag = 0;
void withdrawal(int amount) {
   while (cswap( flag, 0, 1 ) == 1) { }
   if (balance > amount) {
      System.out.println( “withdrawal approved” );
      balance = balance – amount;
   }
   flag = 0;
}
```

flag 1 Balance $50

Thread 2 – Withdraw $25 zzz

Thread 1 – Withdraw $40

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka "compare & exchange")
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

```
int flag = 0;
void withdrawal(int amount) {
    while (cswap( flag, 0, 1 ) == 1) { }
    if (balance > amount) {
        System.out.println( "withdrawal approved" );
        balance = balance – amount;
    }
    flag = 0;
}
```

flag 1 Balance $50

Thread 2 – Withdraw $25

Thread 1 – Withdraw $40 zzz

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka "compare & exchange")
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

flag 1 Balance $50

```
int flag = 0;
void withdrawal(int amount) {
    while (cswap( flag, 0, 1 ) == 1) { }
    if (balance > amount) {
        System.out.println( "withdrawal approved" );
        balance = balance – amount;
    }
    flag = 0;
}
```

Busy waiting

Thread 2 – Withdraw $25

Thread 1 – Withdraw $40 zzz

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka “compare & exchange”)
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

```
int flag = 0;
void withdrawal(int amount) {
  while (cswap( flag, 0, 1 ) == 1) { }
  if (balance > amount) {
    System.out.println( “withdrawal approved” );
    balance = balance – amount;
  }
  flag = 0;
}
```

Busy waiting

flag 1 Balance $50

Thread 2 – Withdraw $25

ZZZ

Thread 1 – Withdraw $40

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka “compare & exchange”)
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

```
int flag = 0;
void withdrawal(int amount) {
  while (cswap( flag, 0, 1 ) == 1) { }
  if (balance > amount) {
    System.out.println( “withdrawal approved” );
    balance = balance – amount;
  }
  flag = 0;
}
```

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka "compare & exchange")
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

```
int flag = 0;
void withdrawal(int amount) {
   while (cswap( flag, 0, 1 ) == 1) { }
   if (balance > amount) {
      System.out.println( "withdrawal approved" );
      balance = balance – amount;
   }
   flag = 0;
}
```

flag 1 Balance $10

Thread 2 – Withdraw $25

Mutual Exclusion

- In Hardware
  - Compare & Swap (aka “compare & exchange”)
    - instruction comparing a memory value and a test value
    - if values == then swap else nothing (always return swap)

```
int flag = 0;
void withdrawal(int amount) {
   while (cswap( flag, 0, 1 ) == 1) { }
   if (balance > amount) {
      System.out.println( “withdrawal approved” );
      balance = balance – amount;
   }
   flag = 0;
}
```

flag 0 Balance $10

Thread 2 – Withdraw $25

Mutual Exclusion

- In Hardware
  - 2. Special (atomic) Instructions (II)
    - “Exchange” is similar to “Compare & Swap”
    - Advantages
      - Simple and easy to verify
      - Works for any number of processes & processors sharing memory
      - Supports many critical sections (each has its own variable)
    - Disadvantages
      - When busy-waiting a thread consumes CPU without doing anything
      - Starvation is possible (if 2+ threads are busy-waiting)
      - Deadlock is possible (if 2+ resources are needed in a critical section

Chapter 5 Topics

- Principles of Concurrency
  - Terminology, Process interaction
- Mutual Exclusion
  - Race condition, Requirements, Hardware approaches
- Semaphores
  - Definition, Implementation, Producer/Consumer
- Monitors
  - Definition, Producer/Consumer (revisited), Implementation
- Message Passing
  - Synchronization, Addressing, Mailboxes
- Readers/Writers Problem
  - Light switch pattern

Semaphores

- Software module
  - encapsulating an integer variable & implementing three operations:
    - Initialize: to a non-negative value
    - Wait: decrements value (if it can, if not it waits)
    - Signal: increments value (no wait)

A binary semaphore (mutex) allows 1 thread-access to a critical section
A multiplex allows up to n thread-access at the same time

- Consequences (there is no way to know…)
  - …before decrementing whether it will block or not
  - …(when threads run concurrently) which one goes next
  - …how many threads are waiting (if any)

Semaphores

- Definition

```
struct semaphore {
      int count;
      queueType queue;
};
```

Waiting Queue

```
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Waiting Queue

- Strong semaphore: strictly FIFO
- Weak semaphore: selection can vary (e.g., priorities)

Semaphores

- Definition

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

If negative: add to queue & wait

If non-positive: enable next from queue

Semaphore

- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

- Use in programs

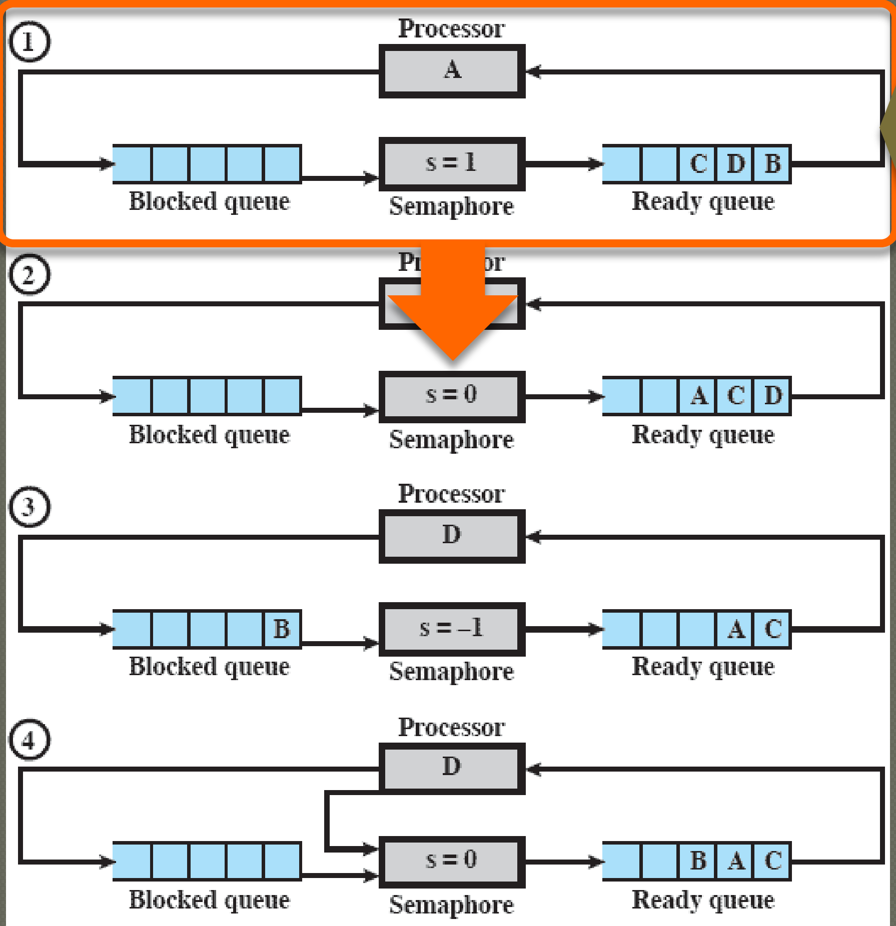
```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder    */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

- wait until resource available (s > 0)
- signal that a resource is available (s++)

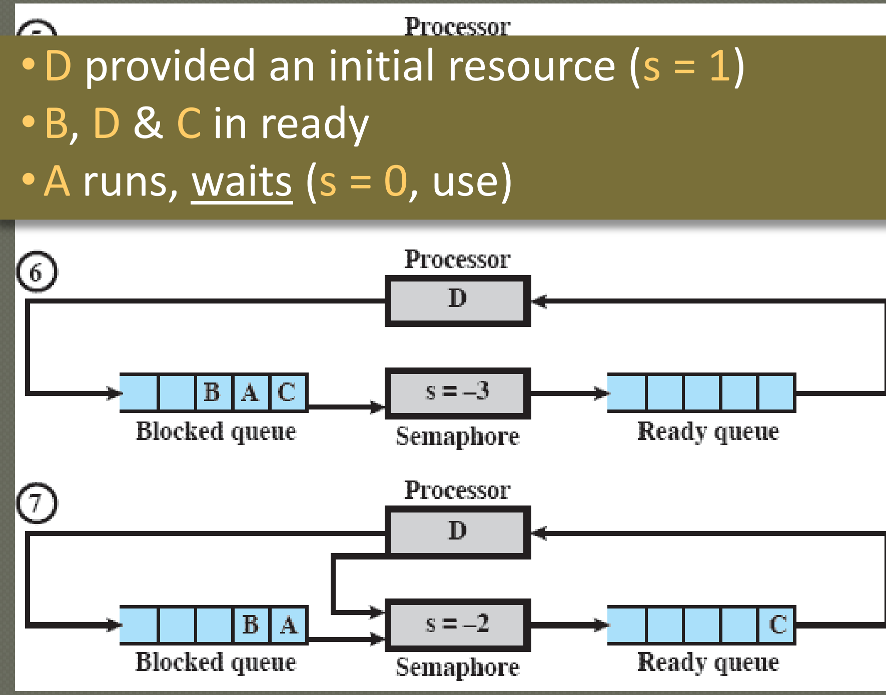
Semaphores

A, B & C use a resource from D

- Example



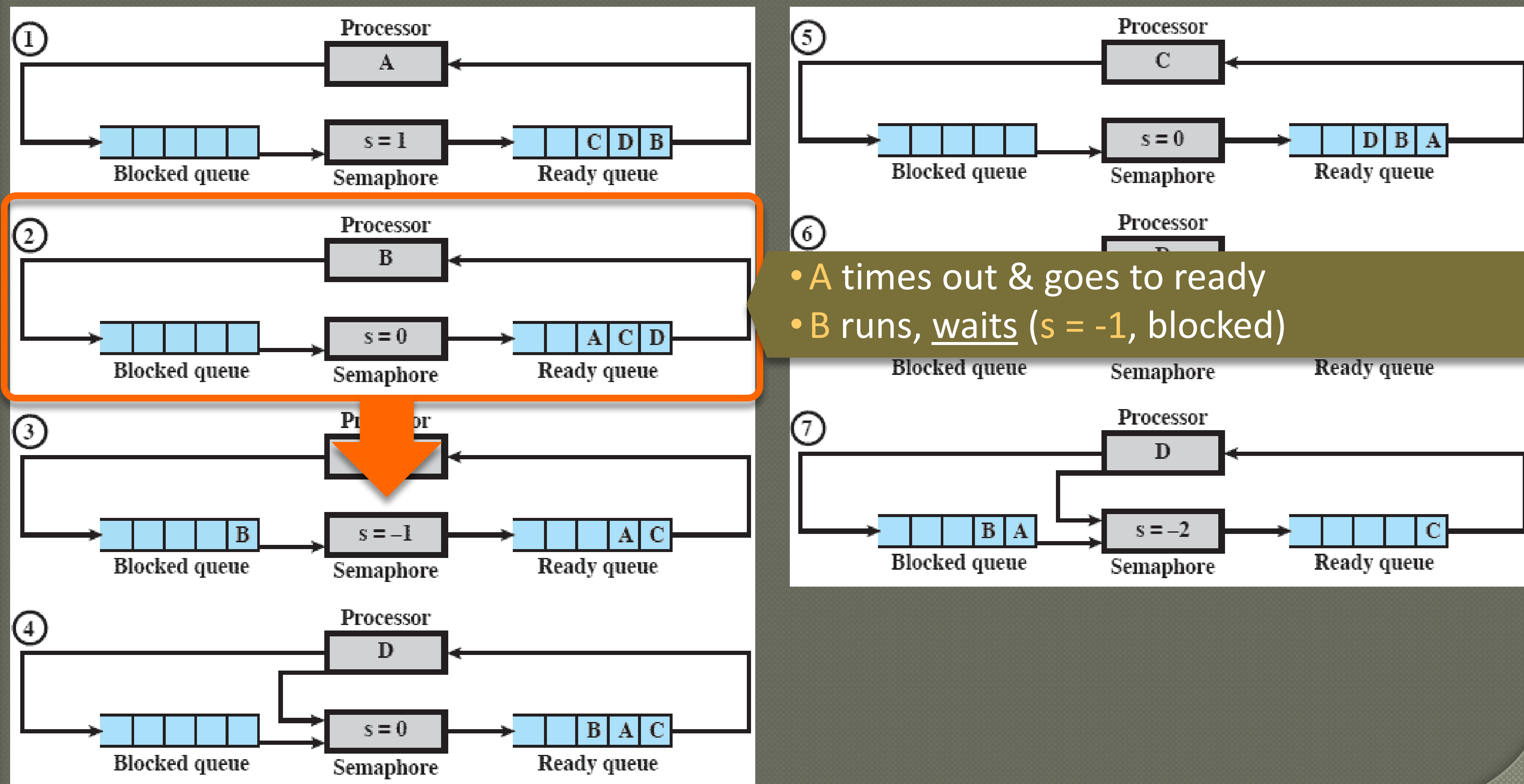
- D provided an initial resource (s = 1)
- B, D & C in ready
- A runs, <u>waits</u> (s = 0, use)



Semaphores

A, B & C use a resource from D

- Example

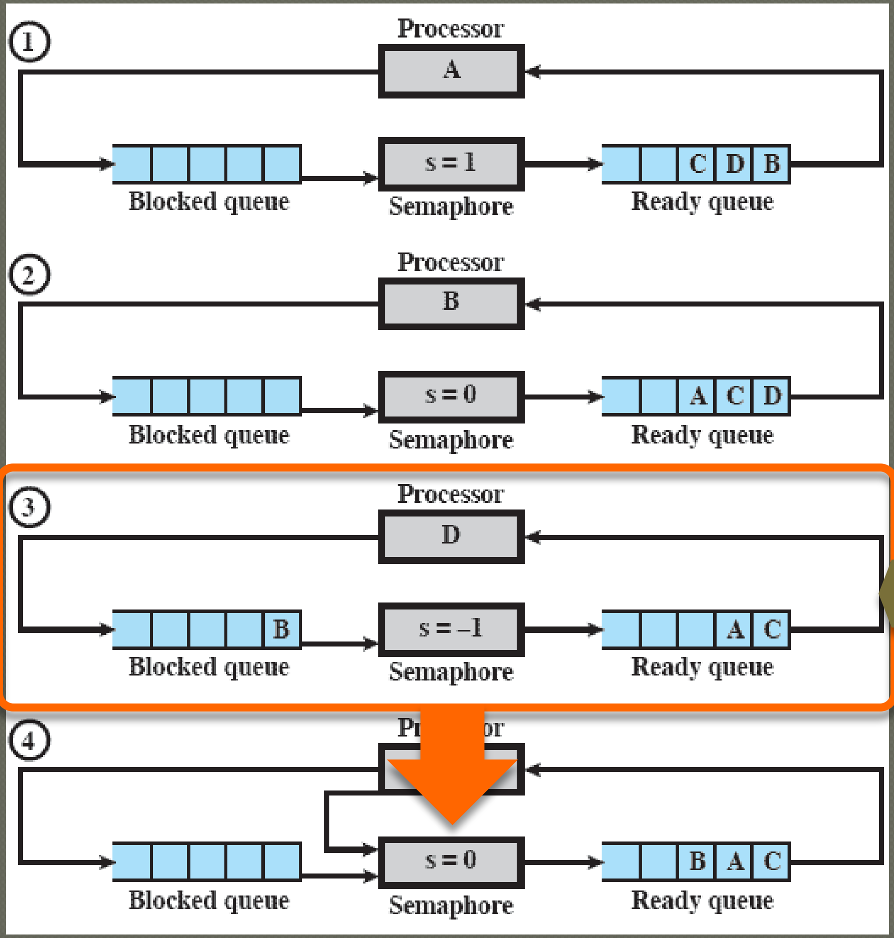


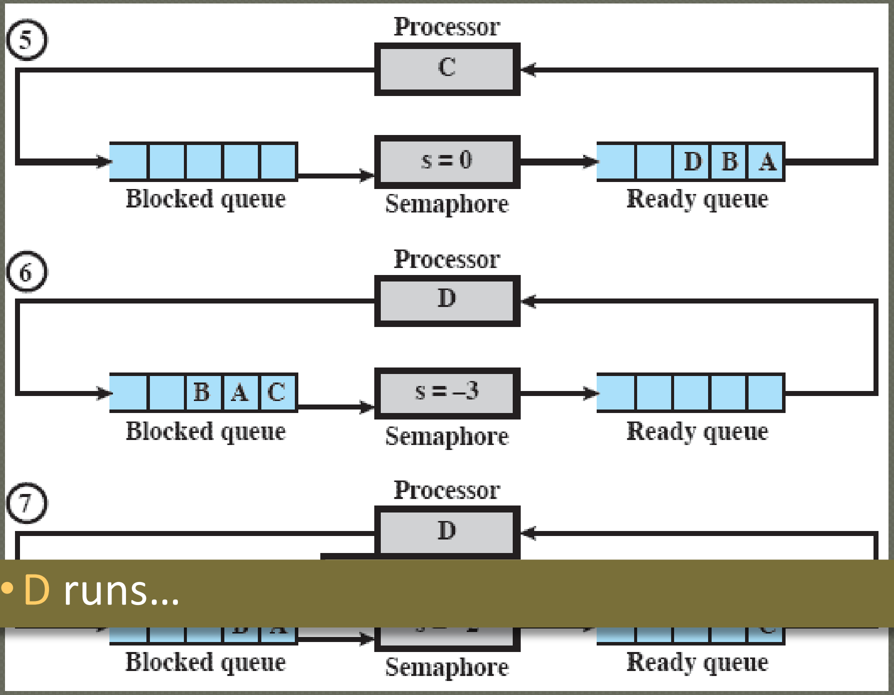
- A times out & goes to ready
- B runs, <u>waits</u> (s = -1, blocked)

Semaphores

A, B & C use a resource from D

- Example



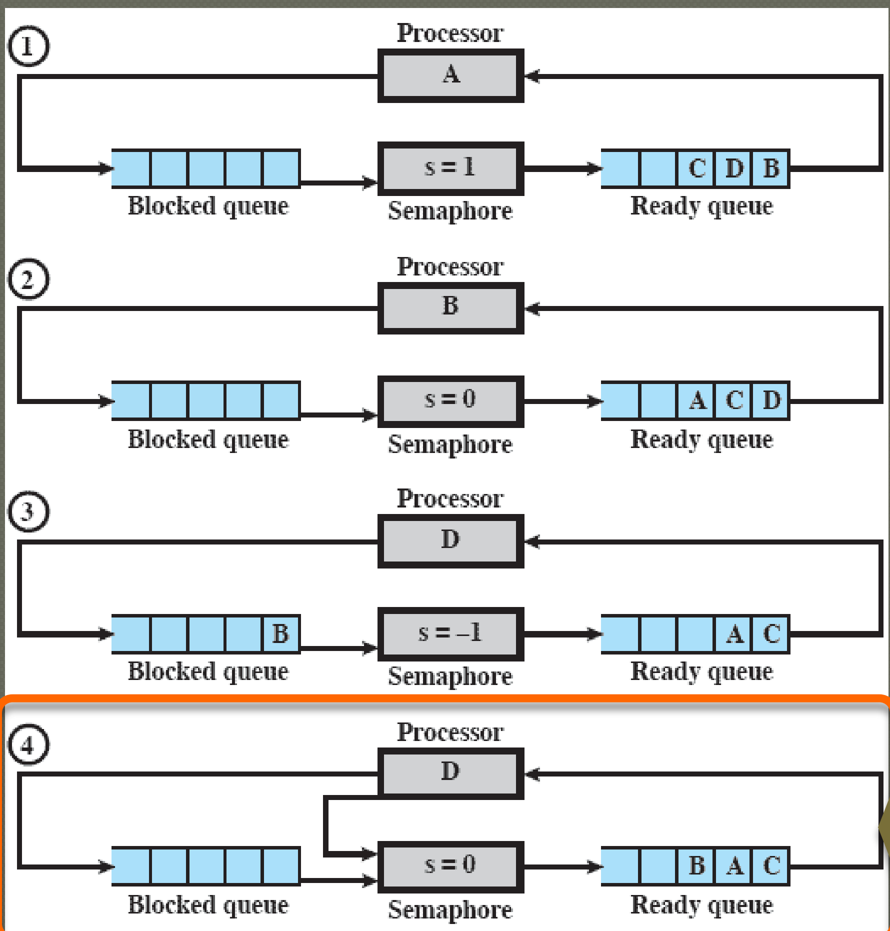


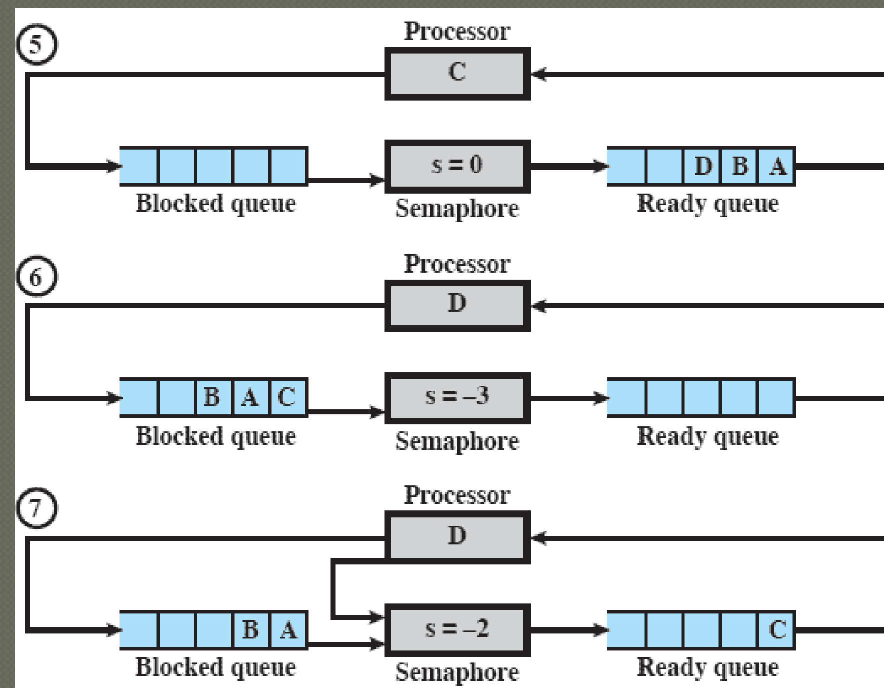
- D runs…

Semaphores

A, B & C use a resource from D

- Example



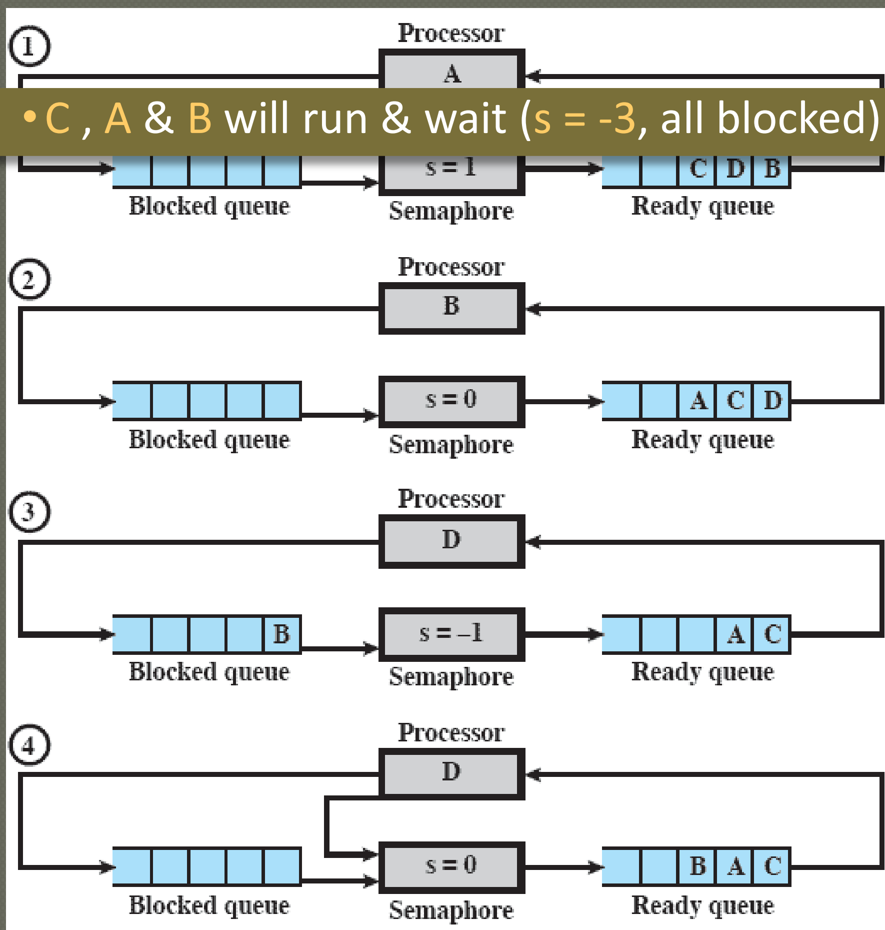


- … (D) , signals (s = 0, dequeue B), times out & goes to ready

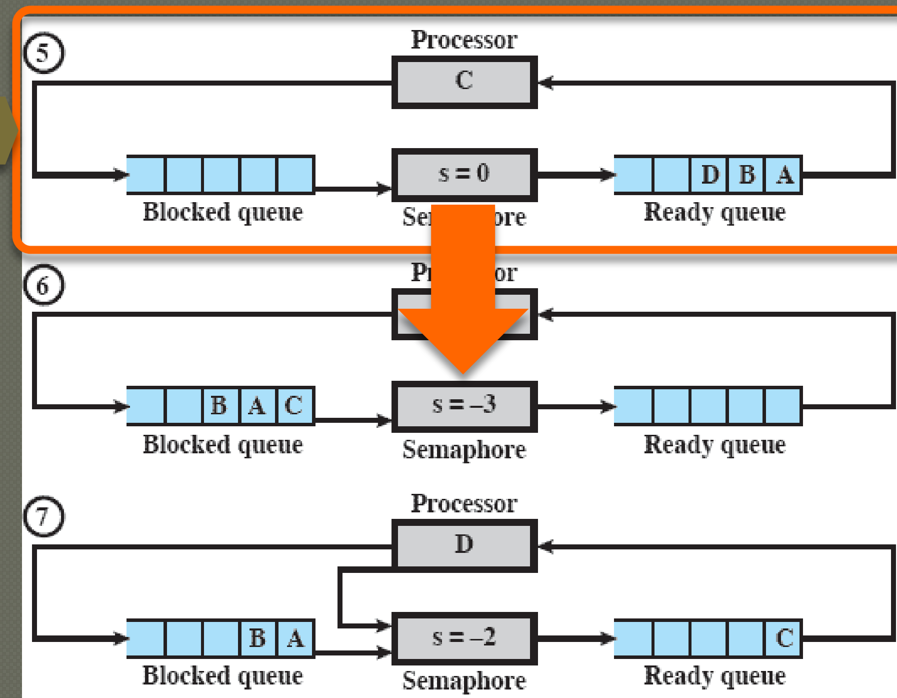
Semaphores

A, B & C use a resource from D

- Example



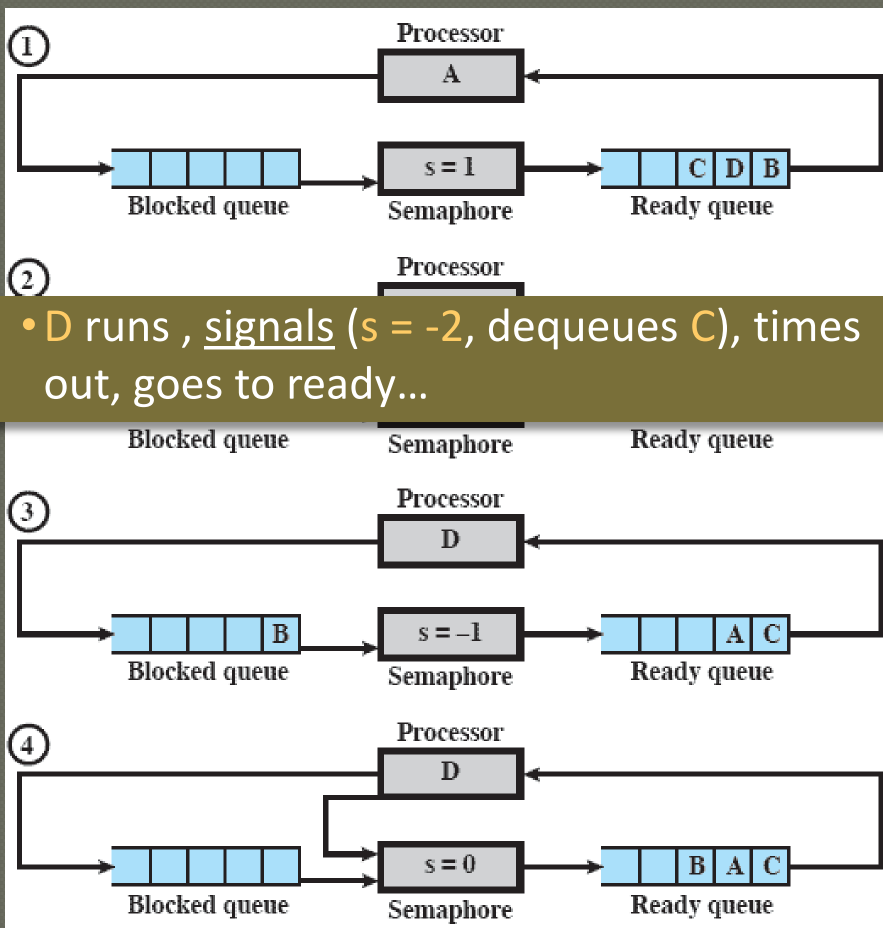
- C , A & B will run & wait (s = -3, all blocked)



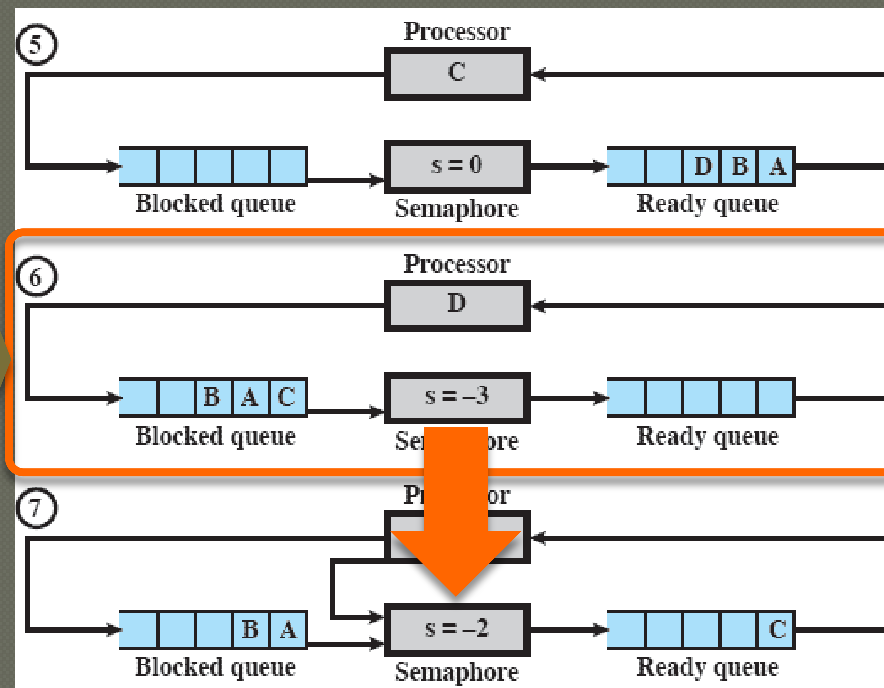
Semaphores

A, B & C use a resource from D

Example



- D runs , signals (s = -2, dequeues C), times out, goes to ready…



…and will continue (forever) alternating D with one of C, A & B.

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

- Producer/Consumer Problem
  - Description
    - A producer adds an item to a buffer
    - A consumer removes an item out of the buffer
    - There might be many producers/consumers
      - but only one can access the buffer at any one time
  - The Problem (ensure that…)
    - …producers can't add items into a full buffer (overflow)
    - …consumer can't remove from an empty buffer (underflow)

Initially, let's focus on buffer underflows (i.e., removing when empty)

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
      s.sWait();              // wait for critical section
      r++;                    // produce
      if (r == 1) {           // if this is first item…
         delay.sSignal();     // …awake consumers
      }
      s.sSignal();            // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();             // wait for an item
   while (true) {
      s.sWait();              // wait for critical section
      r--;                    // consume
      s.sSignal();            // release critical section
      if (r == 0) {           // if there are no items...
         delay.sWait();       // ...wait for an item
      }
   }
}
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();             // wait for critical section
         r++;                   // produce
         if (r == 1) {          // if this is first item...
            delay.sSignal();    // ...awake consumers
         }
         s.sSignal();           // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();               // wait for an item
   while (true) {
         s.sWait();             // wait for critical section
         r--;                   // consume
         s.sSignal();           // release critical section
         if (r == 0) {          // if there are no items...
            delay.sWait();      // ...wait for an item
         }
   }
}
```

```
delay.flag = 0
s.flag     = 1
r          = 0
start( producer, consumer1, consumer2 )
```

current thread

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
      s.sWait();           // wait for critical section
      r++;                 // produce
      if (r == 1) {        // if this is first item...
         delay.sSignal();  // ...awake consumers
      }
      s.sSignal();         // release critical section
   }
}
```

producer 1

Consumer

```
public void run() {
   delay.sWait();          // wait for an item
   while (true) {
      s.sWait();           // wait for critical section
      r--;                 // consume
      s.sSignal();         // release critical section
      if (r == 0) {        // if there are no items...
         delay.sWait();    // ...wait for an item
      }
   }
}
```

consumer 1 zzz
consumer 2 zzz

```
delay.flag = 0 | -2
s.flag     = 1 | 0
r          = 0
start( producer, consumer1, consumer2 )
```

Semaphore

- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();            // wait for critical section
         r++;
         if (r == 1) {         // if this is first item...
            delay.sSignal();   // ...awake consumers
         }
         s.sSignal();          // release critical section
   }
}
```

producer 1

Consumer

```
public void run() {
   delay.sWait();
   while (true) {
         s.sWait();            // wait for critical section
         r--;                  // consume
         s.sSignal();          // release critical section
         if (r == 0) {         // if there are no items...
            delay.sWait();     // ...wait for an item
         }
   }
}
```

consumer 1 zzz consumer 2 zzz

```
delay.flag = 0 | -2
s.flag     = 1 | 0
r          = 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();            // wait for critical section
         r++;                  // produce
         if (r == 1) {         // [producer 1] ...rst item...
            delay.sSignal();   // ...awake consumers
         }
         s.sSignal();          // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();              [consumer 1] zzz [consumer 2] zzz
   while (true) {
         s.sWait();            // wait for critical section
         r--;                  // consume
         s.sSignal();          // release critical section
         if (r == 0) {         // if there are no items...
            delay.sWait();     // ...wait for an item
         }
   }
}
```

```
delay.flag = 0 | -2
s.flag     = 1 | 0
r          = 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
       s.sWait();          // wait for critical section
       r++;                // produce
       if (r == 1) {       // if this is first item...
          delay.sSignal(); // ...consumers
       }
       s.sSignal();        // release critical section
   }
}
```

producer 1

Consumer

```
public void run() {
   delay.sWait();
   while (true) {
       s.sWait();           // wait for critical section
       r--;                 // consume
       s.sSignal();         // release critical section
       if (r == 0) {        // if there are no items...
          delay.sWait();    // ...wait for an item
       }
   }
}
```

consumer 1

consumer 2 zzz

```
delay.flag = 0 | -2 | -1
s.flag     = 1 | 0
r          = 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
        s.sWait();             // wait for critical section
        r++;                   // produce
        if (r == 1) {          // if this is first item...
           delay.sSignal();    // ...awake consumers
        }                      producer 1
        s.sSignal();           // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();              // wait for an   consumer 2 zzz
   while (true) {
        s.sWait();             consumer 1 zzz ritical section
        r--;                   // consume
        s.sSignal();           // release critical section
        if (r == 0) {          // if there are no items...
           delay.sWait();      // ...wait for an item
        }
   }
}
```

```
delay.flag = 0 | -2 | -1
s.flag     = 1 | 0 | -1
r          = 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphores

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();           // wait for critical section
         r++;                 // produce
         if (r == 1) {        // if this is first item...
            delay.sSignal();  // ...awake consumers
         }
         s.sSignal();     producer 1 itical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();          // wait for an  consumer 2 zzz
   while (true) {
         s.sWait();    consumer 1 ritical section
         r--;                 // consume
         s.sSignal();         // release critical section
         if (r == 0) {        // if there are no items...
            delay.sWait();    // ...wait for an item
         }
   }
}
```

```
delay.flag = 0 | -2 | -1
s.flag     = 1 | 0 | -1 | 0
r          = 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore

- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();              producer 1 zzz itical section
         r++;                    // produce
         if (r == 1) {           // if this is first item...
            delay.sSignal();     // ...awake consumers
         }
         s.sSignal();            // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();              // wait for an  consumer 2 zzz
   while (true) {
         s.sWait();            // wait for critical section
         r--;                  consumer 1
         s.sSignal();          // release critical section
         if (r == 0) {         // if there are no items...
            delay.sWait();     // ...wait for an item
         }
   }
}
```

```
delay.flag = 0 | -2 | -1
s.flag     = 1 | 0 | -1 | 0 | -1
r          = 0 | 1 | 0
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();              // wait for critical section
         r++;                    // produce
         if (r == 1) {           // if this is first item...
            delay.sSignal();     // ...awake consumers
         }
         s.sSignal();            // release critical section
   }
}
```

producer 1

Consumer

```
public void run() {
   delay.sWait();                // wait for an item
   while (true) {
         s.sWait();              // wait for critical section
         r--;                    // consume
         s.sSignal();            // release critical section
         if (r == 0) {           // if there are no items...
            delay.sWait();       // ...wait for an item
         }
   }
}
```

consumer 2 zzz

consumer 1

```
delay.flag = 0 | -2 | -1
s.flag     = 1 | 0 | -1 | 0 | -1 | 0
r          = 0 | 1 | 0
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
      s.sWait();           // wait for critical section
      r++;                 // producer 1
      if (r == 1) {        // if this is first item...
         delay.sSignal();  // ...awake consumers
      }
      s.sSignal();         // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();          // wait for an  consumer 2 zzz
   while (true) {
      s.sWait();           // wait for critical section
      r--;                 // consume
      s.sSignal();         // release critical section
      if (r == 0) {        // consumer 1 re no items...
         delay.sWait();    // ...wait for an item
```

could producer increment r before consumer reads it?

```
delay.flag = 0 | -2 | -1
s.flag     = 1 | 0 | -1 | 0 | -1 | 0
r          = 0 | 1 | 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
    while (true) {
        s.sWait();              // wait for critical section
        r++;                    // produce
        if (r == 1) {           producer 1 rst item...
            delay.sSignal();    // ...awake consumers
        }
        s.sSignal();            // release critical section
    }
}
```

Consumer

```
public void run() {
    delay.sWait();              // wait for an  consumer 2 zzz
    while (true) {
        s.sWait();              // wait for critical section
        r--;                    // consume
        s.sSignal();            // release critical section
        if (r == 0) {           // if there are no items...
            delay.sWait();      // ...wait for an item
        }                       consumer 1
    }
}
```

```
delay.flag = 0 | -2 | -1
s.flag     = 1 | 0 | -1 | 0 | -1 | 0
r          = 0 | 1 | 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

- Producer/Consumer Problem
  - applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();              // wait for critical section
         r++;                    // produce
         if (r == 1) {           // if this is first item...
             delay.sSignal();   producer 1  consumers
         }
         s.sSignal();            // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();              // wait for an  consumer 2
   while (true) {
         s.sWait();            consumer 1 zzz ritical section
         r--;                  // consume
         s.sSignal();          // release critical section
         if (r == 0) {         // if there are no items...
             delay.sWait();    // ...wait for an item
         }
   }
}
```

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1
r          = 0 | 1 | 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore

- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();               // wait for critical section
         r++;                     // produce
         if (r == 1) {            // if this is first item...
            delay.sSignal();      // ...awake consumers
         }                        producer 1
         s.sSignal();             // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();                 // wait for an item
   while (true) {
         s.sWait();               consumer 1 zzz   consumer 2 zzz
         r--;                     // consume
         s.sSignal();             // release critical section
         if (r == 0) {            // if there are no items...
            delay.sWait();        // ...wait for an item
         }
   }
}
```

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1 | -2
r          = 0 | 1 | 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
    while (true) {
        s.sWait();          // wait for critical section
        r++;                // produce
        if (r == 1) {       // if this is first item...
            delay.sSignal(); // ...awake consumers
        }
        s.sSignal();        producer 1 itical section
    }
}
```

Consumer

```
public void run() {
    delay.sWait();          // wait for an item
    while (true) {
        s.sWait();          consumer 1 cri consumer 2 zzz
        r--;                // consume
        s.sSignal();        // release critical section
        if (r == 0) {       // if there are no items...
            delay.sWait();  // ...wait for an item
        }
    }
}
```

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1 | -2 | -1
r          = 0 | 1 | 0 | 1
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
        s.sWait();             // wait for critical section
        r++;                   // produce
        if (r == 1) {          // if this is first item...
            delay.sSignal();   // ...awake consumers
        }
        s.sSignal();           // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();              // wait for an item
   while (true) {
        s.sWait();             // wait for critical section
        r--;
        s.sSignal();           // release critical section
        if (r == 0) {          // if there are no items...
            delay.sWait();     // ...wait for an item
        }
   }
}
```

producer 1 zzz

consumer 2 zzz

consumer 1

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1 | -2 | -1 | -2
r          = 0 | 1 | 0 | 1 | 0
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
      s.sWait();          // ...critical section
      r++;                // produce
      if (r == 1) {       // if this is first item...
         delay.sSignal(); // ...awake consumers
      }
      s.sSignal();        // release critical section
   }
}
```

producer 1 zzz

Consumer

```
public void run() {
   delay.sWait();         // wait for an item
   while (true) {
      s.sWait();          // wait for cri...
      r--;                // consume
      s.sSignal();        // ...ritical section
      if (r == 0) {       // if there are no items...
         delay.sWait();   // ...wait for an item
      }
```

consumer 2 zzz

consumer 1

could signal awake consumer2 instead of producer?

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1 | -2 | -1 | -2 | -1
r          = 0 | 1 | 0 | 1 | 0
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
        s.sWait();              // wait for critical section
        r++;                    // produce
        if (r == 1) {           // if this is first item...
           delay.sSignal();     // ...awake consumers
        }
        s.sSignal();            // release critical section
   }
}
```

producer 1 zzz

Consumer

```
public void run() {
   delay.sWait();               // wait for an item
   while (true) {
        s.sWait();              // wait for critical section
        r--;                    // consume
        s.sSignal();            // release critical section
        if (r == 0) {           // if there are no items...
           delay.sWait();       // ...wait for an item
        }
   }
}
```

consumer 2

consumer 1

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1 | -2 | -1 | -2 | -1
r          = 0 | 1 | 0 | 1 | 0 | -1
start( producer, consumer1, consumer2 )
```

halt! resource underflow!

What is the problem?

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();              // wait for critical section
         r++;                    // produce
         if (r == 1) {           // if this is first item...
            delay.sSignal();     // ...awake consumers
         }
         s.sSignal();            // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();                // wait for an item
   while (true) {
         s.sWait();              // wait for critical section
         r--;                    // consume
         s.sSignal();            // release critical section
         if (r == 0) {           // if there are no items...
            delay.sWait();       // ...wait for an item
         }
   }
}
```

it is possible to change the value of r between the time the critical section is freed and the time r is read (in if)

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1 | -2 | -1 | -2 | -1
r          = 0 | 1 | 0 | 1 | 0 | -1
start( producer, consumer1, consumer2 )
```

How to solve it?

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
      s.sWait();          // wait for critical section
      r++;                // produce
      if (r == 1) {       // if this is first item...
         delay.sSignal(); // ...awake consumers
      }
      s.sSignal();        // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();         // wait for an item
   while (true) {
      s.sWait();          // wait for critical section
      r--;                // consume
      if (r == 0) {       // if there are no items...
         delay.sWait();   // ...wait for an item
      }
      s.sSignal();
   }
}
```

moving signal after "if".
would it work?

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1 | -2 | -1 | -2 | -1
r          = 0 | 1 | 0 | 1 | 0 | -1
start( producer, consumer1, consumer2 )
```

How to solve it?

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
       s.sWait();           // wait for critical section
       r++;                 // produce
       if (r == 1) {        // if this is first item...
          delay.sSignal();  // ...awake consumers
       }
       s.sSignal();         // release critical section
   }
}
```

Consumer

```
public void run() {
   delay.sWait();           // wait for an item
   while (true) {
       s.sWait();           // wait for critical section
       int m = --r;         // consume
       s.sSignal();         // release critical section
       if (m == 0) {        // if there are no items...
          delay.sWait();    // ...wait for an item
       }
   }
}
```

add a local variable m to remember the value of r at the time it was modified

a solution

```
delay.flag = 0 | -2 | -1 | 0
s.flag     = 1 | 0 | -1 | 0 | -1 | 0 | -1 | -2 | -1 | -2 | -1
r          = 0 | 1 | 0 | 1 | 0 | -1
start( producer, consumer1, consumer2 )
```

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         s.sWait();          // wait for critical section
         r++;                // produce
         s.sSignal();        // release critical section
         n.sSignal();        // item available
   }
}
```

Consumer

```
public void run() {
   while (true) {
         n.sWait();          // wait for item
         s.sWait();          // wait for critical section
         r--;                // consume
         s.sSignal();        // release critical section
   }
}
```

avoiding underflow

using a semaphore n signaling
when items are available (delay is gone)

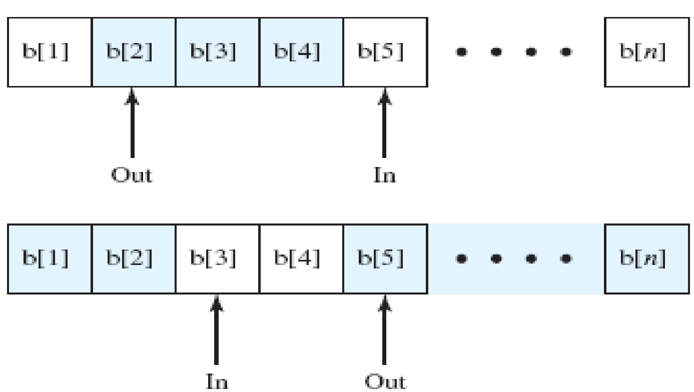
improved solution

```
n.flag    = 0
s.flag    = 1
r         = 0
start( producer, consumer1, consumer2 )
```

Semaphores

- Producer/Consumer Problem
  - initially, we prevented buffer underflow
    - i.e., removing when empty
  - now, let's prevent buffer overflow
    - i.e., adding beyond capacity

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed



Implemented using a circular list
- adding at in, removing from out
- may wrap around

how to control it?

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

Producer/Consumer Problem

- applying mutual exclusion in a critical section

Producer

```
public void run() {
   while (true) {
         c.sWait();          // wait for capacity to add
         s.sWait();          // wait for critical section
         r++;                // produce
         s.sSignal();        // release critical section
         n.sSignal();        // item available
   }
}
```

Consumer

```
public void run() {
   while (true) {
         n.sWait();          // wait for item
         s.sWait();          // wait for critical section
         r--;                // consume
         s.sSignal();        // release critical section
         c.sSignal();        // capacity available
   }
}
```

```
c.flag    = 10
n.flag    = 0
s.flag    = 1
r         = 0
start( producer, consumer1, consumer2 )
```

using a semaphore c restricting the maximum number of items in the buffer (e.g., 10)

avoiding overflow

Semaphore
- wait: --flag < 0 ? block : continue
- signal: ++flag < 1 ? dequeue & continue

Semaphores

- Implementation
  - sWait & sSignal must be implemented as atomic operations (in hardware or software)

Java

```java
public class Semaphore {
    private int counter;

    public Semaphore(int _counter) {
        counter = _counter;
    }

    public synchronized void sWait() {
        if (--counter < 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }

    public synchronized void sSignal() {
        if (++counter < 1) {
            notify();
        }
    }
}
```

Chapter 5 Topics

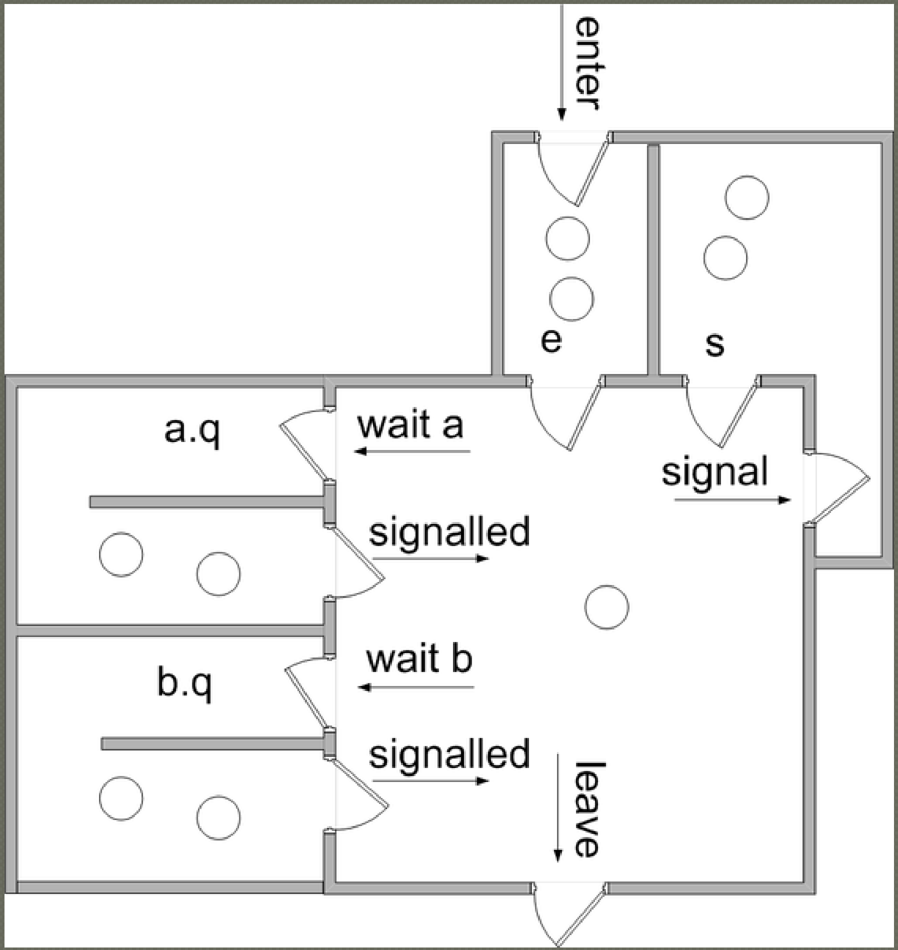
- Principles of Concurrency
  - Terminology, Process interaction
- Mutual Exclusion
  - Race condition, Requirements, Hardware approaches
- Semaphores
  - Definition, Implementation, Producer/Consumer
- Monitors
  - Definition, Producer/Consumer (revisited), Implementation
- Message Passing
  - Synchronization, Addressing, Mailboxes
- Readers/Writers Problem
  - Light switch pattern

Monitors

- Software module that…
  - has methods (invoked by threads)
  - encapsulates data structures (accessible only through its methods)
  - only 1 thread can execute its methods at any one time
- Synchronization
  - achieved through conditional variables
    - encapsulated in the monitor
    - each conditional variable…
      - manages a block queue
      - uses cWait() and cSignal() to block/dequeue threads.

Monitors

- Bird's eye view
- Implemented in
  - Pascal-descendants
    - Concurrent Pascal & Modula
  - Java



from: wikipedia.org

Monitors

- Producer/Consumer Problem
  - using a monitor m to encapsulate the buffer
  - synchronization happens within the monitor
    - not in your programs!

Producer

```
public void run() {
   while (true) {
         char   c = produce();          // create next
         m.add( c );                    // add to buffer
   }
}
```

Consumer

```
public void run() {
   while (true) {
         char c = m.get();     // get from buffer
         …                     // do whatever
   }
}
```

Monitor
- add: resource == MAX ? wait (until false) then add
- get: resource == 0 ?   wait (until false) then remove & return removed

How does that happen?

Monitor

- cWait: condition ? block in condition(queue) : continue
- cSignal: condition(dequeue)

Monitors

Implementation

Java

```
public class Monitor {
    private char[] data  = new char[ size ];
    private int       count = 0, in = 0, out = 0;

    public synchronized void add(char c) { … }
    public synchronized char get() { … }
}
```

cWait(condition)

cSignal()

```
public synchronized void add(char c) {
    while (count == data.length) {          // isFull
          try { wait(); }
          catch (InterruptedException e) { }
    }
    data[ in ] = c;
    in        = (in + 1) % data.length;
    count++;
    notifyAll();
}
```

cWait(condition)

cSignal()

```
public synchronized char get() {
    while (count == 0) {                    // isEmpty
          try { wait(); }
          catch (InterruptedException e) { }
    }
    char c = data[ out ];
    out    = (out + 1) % data.length;
    count--;
    notifyAll();
    return c;
}
```

Chapter 5 Topics

- Principles of Concurrency
  - Terminology, Process interaction
- Mutual Exclusion
  - Race condition, Requirements, Hardware approaches
- Semaphores
  - Definition, Implementation, Producer/Consumer
- Monitors
  - Definition, Producer/Consumer (revisited), Implementation
- Message Passing
  - Synchronization, Addressing, Mailboxes
- Readers/Writers Problem
  - Light switch pattern

Message Passing

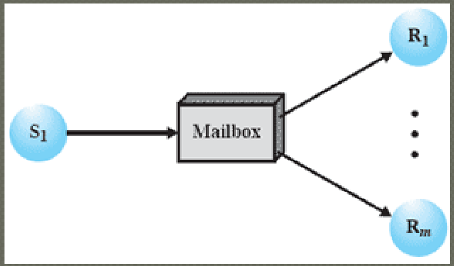
- It achieves
  - synchronization (to reach mutual exclusion) through communication (by exchanging information)
- Communication primitives
  - send(destination, message)
    - deliver message to destination process
  - receive(source, message)
    - receive a message from the source (if non? tough luck!)
  - either synchronous (wait) or asynchronous (no wait)

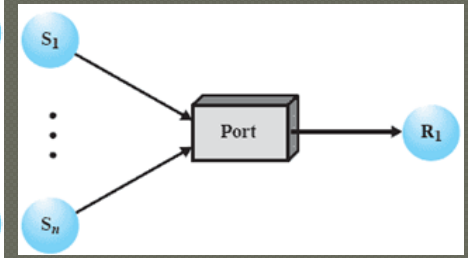
Message Passing

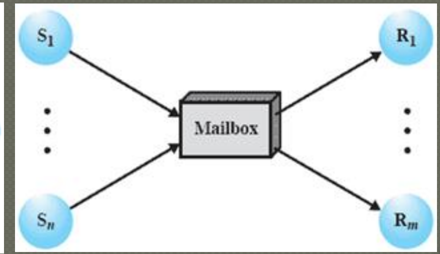
- Synchronicity
  - send & receive (block)
    - aka "rendezvous" allows for tight synchronization
  - send (doesn't block)
    - sender continue (other things) after sending
    - receive (blocks)
      - most common
      - implemented natively (e.g., Java sockets)
    - receive (doesn't block)
      - implemented through libraries (e.g., multi-agents)

Message Passing

- Addressing
  - Direct
    - aka Point-to-point (1:1)
  - Indirect
    - through a shared queue entity (aka mailbox)
      - messages are sent to mailbox
      - receivers pick up messages from mailbox (if any)
    - mailboxes can be used 1:1, 1:N, N:1, M:N







Message Passing

Mailboxes & Mutual Exclusion

- how to use mailboxes for mutual exclusion
  - create a mailbox with a generic message (any receiver)

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Message Passing

- Mailboxes & Mutual Exclusion
  - for the producer/consumer problem
    - create mailboxes "mayProduce" (buffer not full) & "mayConsume" (buffer not empty)

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
      receive (mayproduce, pmsg);
      pmsg = produce();
      send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
      receive (mayconsume, cmsg);
      consume (cmsg);
      send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce, null);
    parbegin (producer, consumer);
}
```

critical section

critical section

Chapter 5 Topics

- Principles of Concurrency
  - Terminology, Process interaction
- Mutual Exclusion
  - Race condition, Requirements, Hardware approaches
- Semaphores
  - Definition, Implementation, Producer/Consumer
- Monitors
  - Definition, Producer/Consumer (revisited), Implementation
- Message Passing
  - Synchronization, Addressing, Mailboxes
- Readers/Writers Problem
  - Light switch pattern

Readers/Writers Problem

- A data resource is shared among processes
  - some only read (readers), some only write (writers)
  - Mutual exclusion rules
    - any readers can read (if no one is writing)
    - only one writer writes at any one time
- Categorical Exclusion (generalization)
  - any number of A or B threads can access a critical section…
  - but no A thread can be in it with a B thread, and vice versa.
- The “Light Switch” Pattern
  - The first A or B in the room turns ON the switch, and the last A or B in the room turns OFF the switch.
  - Any A (or B) can enter the room if…
    - lights are OFF, or
    - there is an A (or B – i.e., if there is someone like you) in the room.

Readers/Writers Problem

- Light Switch Pattern
  - 1 writer only (enters with semaphore)
  - N readers simultaneously (enter with light switch)

Writer

```
public void run() {
   while (true) {
         roomEmpty.sWait();          // wait to enter
         ++r;                        // write
         roomEmpty.sSignal();        // leaving
   }
}
```

Reader

```
public void run() {
   while (true) {
         lightSwitch.   lock( roomEmpty );   // wait
         int blah = r;                       // reading
         lightSwitch.unlock( roomEmpty );   // leaving
   }
}
```

```
roomEmpty.flag = 1
start( writer1, writer2, reader1, reader2, reader3 )
```

Readers/Writers Problem

Reader

```
public void run() {
   while (true) {
         lightSwitch.    lock( roomEmpty );
         int blah = r;
         lightSwitch.unlock( roomEmpty );
   }
}
```

Writer

```
public void run() {
   while (true) {
         roomEmpty.sWait();
         ++r;
         roomEmpty.sSignal();
   }
}
```

Java

```
public class Lightswitch {
   private int              counter = 0; // number inside room
   private Semaphore mutex   = new Semaphore(1 );

   public void lock(Semaphore semaphore) {
         mutex.sWait();                  // can we enter critical?
         if (++counter == 1) {           // if first customer...
               semaphore.sWait();    // ...wait to turn lights ON
         }
         mutex.sSignal();                // exit critical (lights are ON)
   }

   public void unlock(Semaphore semaphore) {
         mutex.sWait();                  // can we enter critical?
         if (--counter == 0) {           // if last customer...
               semaphore.sSignal();  // ...turn lights OFF
         }
         mutex.sSignal();                // exit critical
   }
}
```

```
roomEmpty.flag = 1
start( writer1, writer2, reader1, reader2, reader3 )
```

Chapter 5 Topics

- Principles of Concurrency
  - Terminology, Process interaction
- Mutual Exclusion
  - Race condition, Requiremen ware approaches
- Semaphores
  - Definition, Impleme Producer/Consumer
- Monitors
  - Definition, Produc Consumer (revisited), Implementation
- Message Passing
  - Synchronization, Addressing, Mailboxes
- Readers/Writers Problem
  - Light switch pattern

Done!

Chapter 5 Topics

-
- --------------------------------
- Given
- a1 x = 5
- a2 print x

- b1 x = 7

- What path yields output 5 and final value 5?
- 7
  7?
- 5
  7?
- 7
  5?
- --------------------------------
- a1 x = x + 1

- b1 x = x + 1

- If initial value of x is 1, what is its value after both threads run?

Chapter 5 Topics

- Unisex bathroom: Men and women can use the bathroom, and a maximum of 3 people can use the bathroom at once, but no people of different gender can occupy it at the same time.

- female:
- femaleSwitch.lock(emptyRoom);
- femaleMultiplex.wait();
- //bathroom code
- femaleMultiplex.signal();
- femaleSwitch.lock(emptyRoom);

- male:
- maleSwitch.lock(emptyRoom);
- maleMultiplex.wait();
- //bathroom code
- maleMultiplex.signal();
- maleSwitch.lock(emptyRoom);