Inter-process Communication

Chapter 2.3
Tanenbaum 2015

Inter-process Communication

- Concurrency Issues
  - Race Condition
  - Dead Locks
  - Some typical real-world problems with concurrency:
    - Producers-Consumers
      - Bounded Buffer (Over-produce, Over-consume)
        - » e.g., Video/Audio Streaming
    - Readers-Writers
      - e.g, Banking system balances and updates
    - Classic coordination problem – Dining Philosophers
- Locks (Mutex), Semaphores, Monitors
  - Synchronization using Architectural programming constructs

Basic problem

shared int x = 0 at location 0x3000

P1

```
/* x++ */
move (0x3000), D2
add D2, 1
move D2, (0x3000)
```

P2

```
/* x-- */
move (0x3000), D3
sub D3, 1
move D3, (0x3000)
```

Contents of 0x3000 after P1 & P2 have run?

Race conditions

- A race condition occurs when the “ordering of execution” between two processes (or threads) can affect the outcome of an execution.

- In most situations, race conditions are unacceptable.

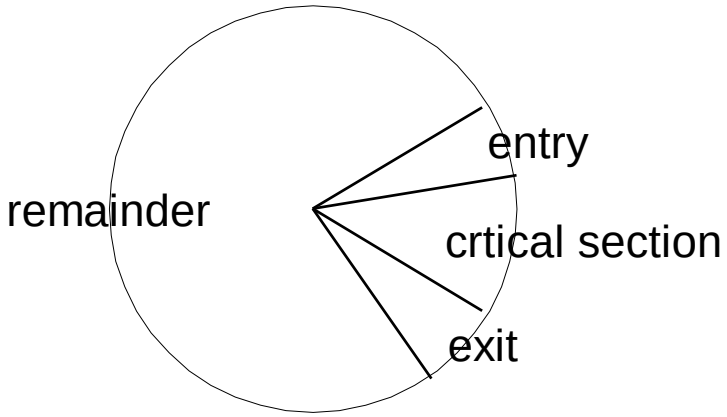
Critical sections/regions (informal)

- A section of code that ensures that only one process accesses a set of shared data. It consists of:
  - Entry (negotiation)
  - Critical section/region (mutual exclusion).
  - Exit (release)

Critical sections/regions

- The rest of the program is called the remainder

```
P1                          P2
// other code…              // other code…
entry();                    entry();
x++;                        x--;
exit();                     exit();
// other code…              // other code…
```



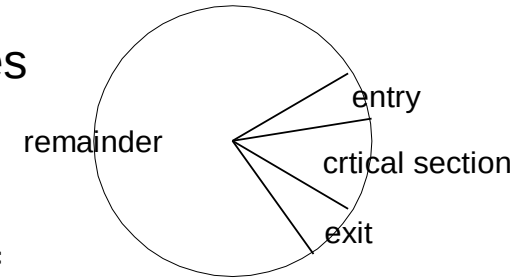
Critical sections/regions

To be a critical region, the following 3 conditions must be met:

1. Mutual exclusion – No more than one process can access the shared data in the critical section.

2. Progress – If no process is accessing the shared data, then:
   a) Only processes executing the entry/exit sections can affect the selection of the next process to enter the critical section.
   b) The process of selection must eventually complete.

3. Bounded waiting – Once a process executes its entry section, there is an upper bound on the number of times that other processes can enter the region of mutual exclusion.



(Note: Use this definition from Silberschatz et al. instead of the one provided by Tanenbaum on all homework, exams, etc.)

Critical sections/regions

- We will study the following types of solutions:
  - software only
  - hardware/software
  - abstractions of the critical region problem
    - data types
    - language constructs

Mutual Exclusion with Disabling Interrupts

Disable CPU interrupts while executing the critical section

User Process?

- Security concerns

Kernel Process?

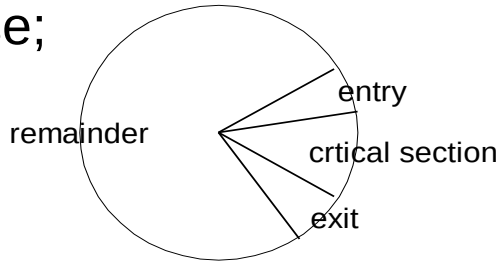
Multiple Core CPU?

Two process critical regions?

```
shared int turn = 0;

foobar()  {
      /* entry */
      while (turn != process_ident)
            do nothing
      /* critical region code */
      …
      /* exit */
      turn = (turn + 1) % 2;
}
```

spin locks

```
shared bool locked = false;

foobar() {
      /* entry */
      while (locked)
            do nothing
      locked = true;
      /* critical region code */
      …
      /* exit */
      locked = false;
}
```



Peterson's 2 process solution (1981)

```
shared boolean interested[2] = {false, false};
shared int turn;

void enter_region(int process)
{
    int other = (process + 1)  % 2;  /* other PID */
    interested[process] = true;
    turn = process;   /* set flag */

    /* Busy-wait until the following is true:
     *    not our turn or other process not interested
     * not our turn? – other process entered after us
     * other process not interested – we can go
     */
    while (turn == process && interested[other] == true)
    no-op;
}
```

```
void exit_region(int process) {
        /* We're all done, no longer
            interested */
        interested[process] = false;
}
```

Does this solution satisfy all critical section properties?

Ensuring 0 + 1 – 1 = 0:

- With Peterson's solution we would write:

P1	P2
// other code…	// other code…
entry(0);	entry(1);
x++;	x--;
exit(0);	exit(1);
// other code…	// other code…

- Other solutions such as the Bakery algorithm (not covered) provide solutions for more than 2 processes.

Atomic operations

- Recall our earlier experience with x++.

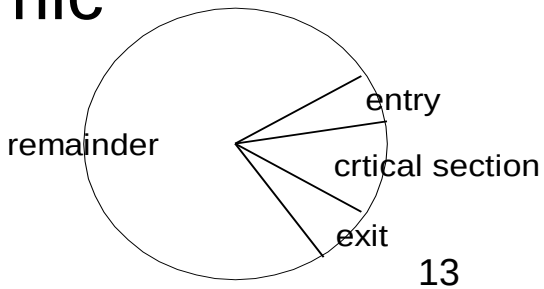
  move (0x3000), D2 ;; increment var at x3000

  add D2, 1

  move D2, (0x3000)

- Atomic instructions

- Use critical sections to achieve atomic operations on the shared data.



Hardware assistance

- Most modern CPUs provide atomic (non interruptible) instructions
  - test and set lock
  - swap word
- We will focus only on test and set lock

Test and set lock

- Pseudocode demonstrating functionality:

executed as if a single instruction

```
boolean TestAndSetLock(boolean *Target) {
    boolean Result;
    Result = *Target;
    *Target = true;
    return Result;
}
```

Mutual exclusion with TSL

```
shared boolean PreventEntry = false;
repeat
    // entry
    while TestAndSetLock(&PreventEntry)
        no-op;
    // mutual exclusion...
    PreventEntry = false;  //exit

until NoLongerNeeded();
```

Is this a critical section? Why or why not?

Test & Set Lock Critical Region

for your information only – you will not be tested over this

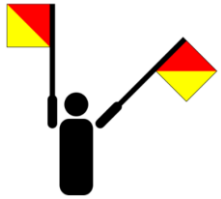
```
shared boolean waiting[N] = {false, false,
      ..., false};
shared boolean lock;

// process local data
int i, j; /* i is process of interest */
boolean key;

repeat
        // entry - Either we obtain the key or
            someone sets our waiting bit to
            false, indicating that we may
            proceed.
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
                key = TestAndSetLock(lock);
        waiting[i] = false;

        // critical section

        // exit – Set next waiting process to
            no longer waiting or if  // we make
            it all the way through release the
            lock.
        j = (i+1) % n;
        while (j != i) && (! waiting[j])
                j = (j+1) % n;
        if (j == i)
                lock = false;  // nobody was
                    waiting, unlock
        else
                waiting[j] = false;

        // remainder
until done;
```

Avoiding busy waiting

- So far, all of our solutions have relied on *spin locks.*

- There should be a better way…

Semaphores (Dijkstra 1965)



- A semaphore is an abstract data type for synchronization.
- A semaphore is a generalized lock.
- A semaphore contains an integer variable which is accessed by two operations known by many different names:

P (test “prohoben”)	wait	down*
V (increment “verhogen”)	signal, or post	up

* We will use down/up in this class, but you should be able to recognize all three.

semaphores

- Libraries frequently pick their own nonstandard names:

	POSIX	Windows
down	sem_wait	WaitForSingleObject
up	sem_post	ReleaseSemaphore

- When used properly, semaphores can implement critical regions.

Semaphore initialization

- When a semaphore is created it is given an initial value. Actual implementation varies, but we will write:

```
semaphore s = 1;     // initialize to 1
```

- It is *important* to always initialize your semaphores.

Semaphore operations

- wait (down) – Decrement the counter value. If the counter is less than zero, block.

- signal (up) – Increment the counter value. If there are processes that got blocked on the semaphore, unblock one of the processes.

Semaphore operations

- Semaphore.Wait and Semaphore.Signal are atomic operations
- Semaphore maintains a queue of processes waiting to enter critical section
- If a process executes S.Wait and semaphore S is greater or equal to zero, it would continue its execution; otherwise, OS would put the process on the wait queue for semaphore S.
- A S.Signal would unblock a process on the wait queue of semaphore S

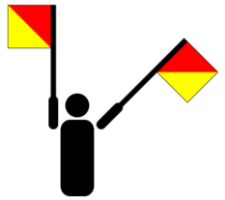
```
S.Wait()          // wait until semaphore S
                  // is available
<critical section>

S.Signal()        // signal to other processes
                  // that semaphore S is free
```

Wait and Signal Example

Two processes using a shared semaphore S with an initial value of 2:

				Execute or Block	
		Value	Queue	P1	P2
		2	Empty	Execute	Execute
P1	Wait	1	Empty	Execute	Execute
P2	Wait	0	Empty	Execute	Execute
P1	Wait	-1	P1	Blocked	Execute
P2	Signal	0	Empty	Execute	Execute
P1	Signal	1	Empty	Execute	Execute
P1	Signal	2	Empty	Execute	Execute

Semaphores



- Essentially an integer variable that can be updated by using two **atomic** operations (wait, signal)
- Two types of semaphores
  - Binary Semaphore (Lock)
  - Counting Semaphore

Using Semaphores

- Control access to 1 resource or N multiple resources
  - Mutex (Binary Semaphore): A lock to guard critical sections
    - Initial value = 1, call Wait(S) before entering a critical section, call Signal(S) when exiting a critical section
  - Track availability of multiple units of a resource
    - A process can acquire access as long as at least one unit of the resource is available
    - Initial value = N, call Wait(S) to acquire access to one of the N resources
- Scheduling / Precedence Constraints
  - Enforce ordering constraints where process / thread must wait for certain condition to proceed
  - Initial value = 0 (usually) or could be any value dependent on the specific setup
  - Example: Implement Thread join (waitpid(PID)) with semaphores

```
Semaphore S;

S.value = 0; // semaphore initialization

Thread.Join          Thread.Finish
  S.Wait();          S.Signal();
```

Critical regions & semaphores

```
shared semaphore Sem = 1;

/* common code */
enter_region() {
    Sem.wait(); // Sem.down()
}

exit_region() {
    Sem.signal(); // Sem.up()
}
```

Ensuring 0 + 1 – 1 = 0 (again):

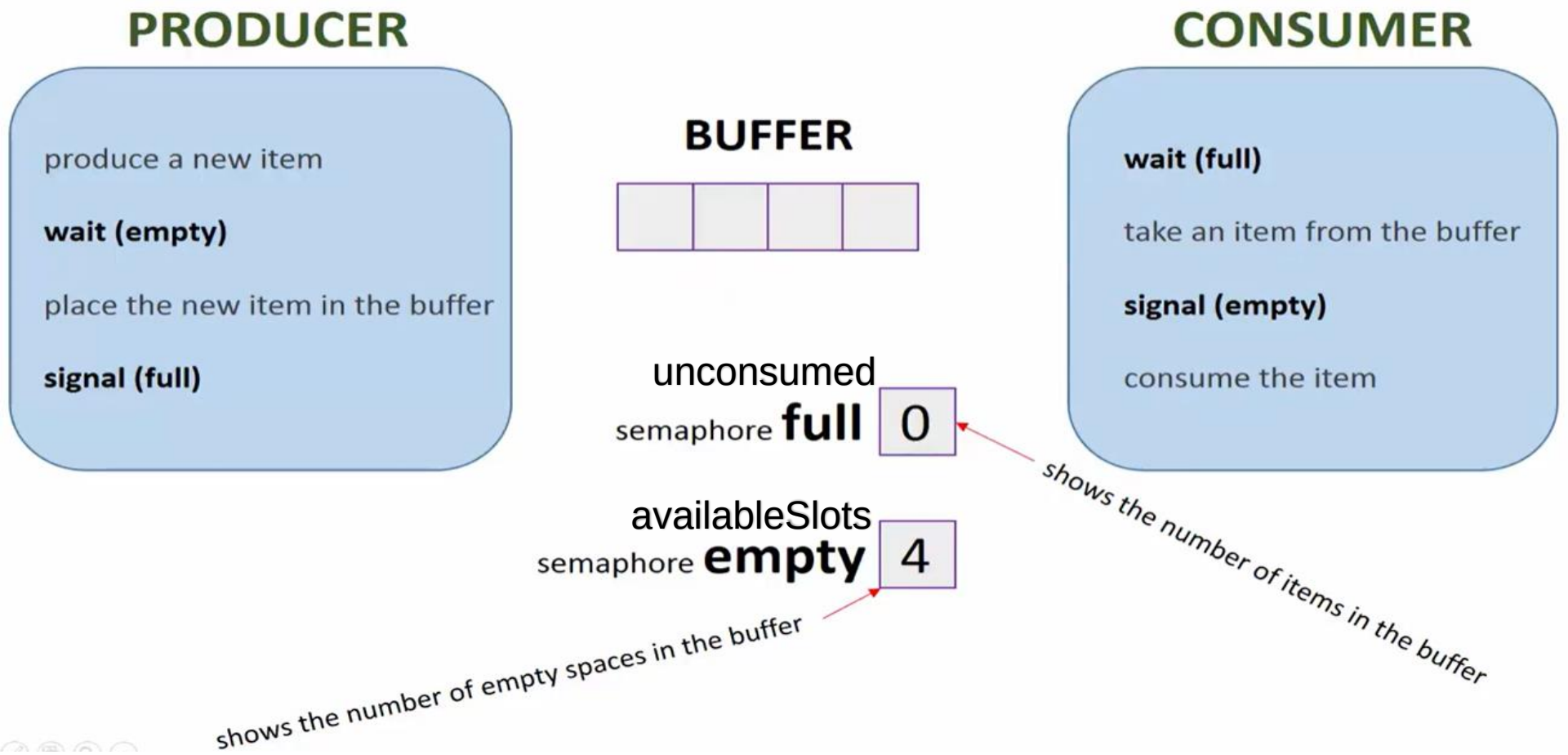
```
shared int x = 0;
shared semaphore Sem = 1;
```

P1	P2
Sem.down();	Sem.down();
x++;	x--;
Sem.up();	Sem.up();

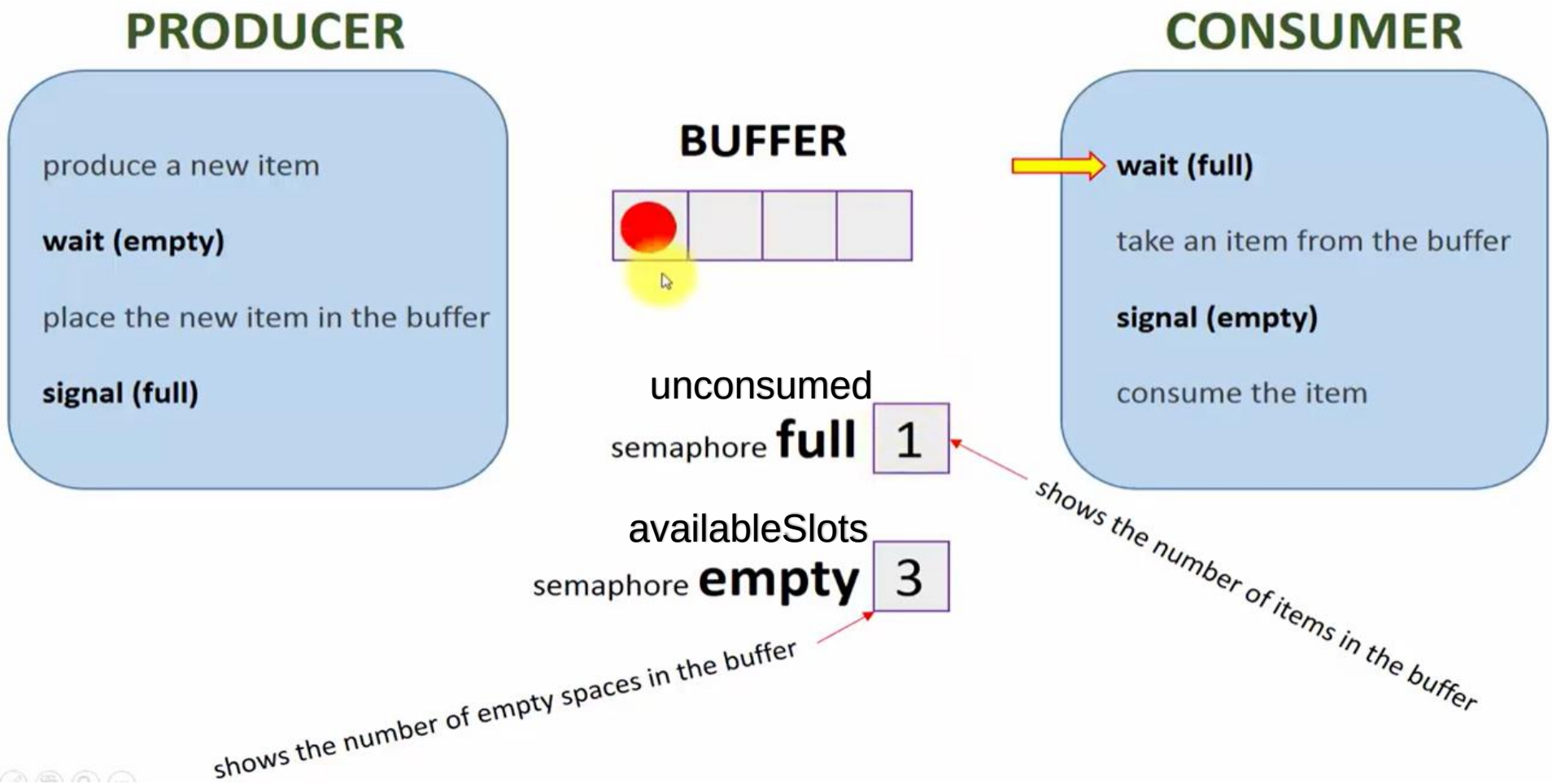
Semaphores for Resource Allocation

- Problem:
  - Pool of N resources
  - A group of processes, each may need resources for an uninterrupted period
  - Guarantee at most N resources are in use at once at any time
  - Initial value: Semaphore s = N;
  - Using resources:
    - S.Wait(); //use or allocate a resource by a process
    - S.Signal(); //return the resource to the pool
  - Semaphore value = number of resources available in the pool

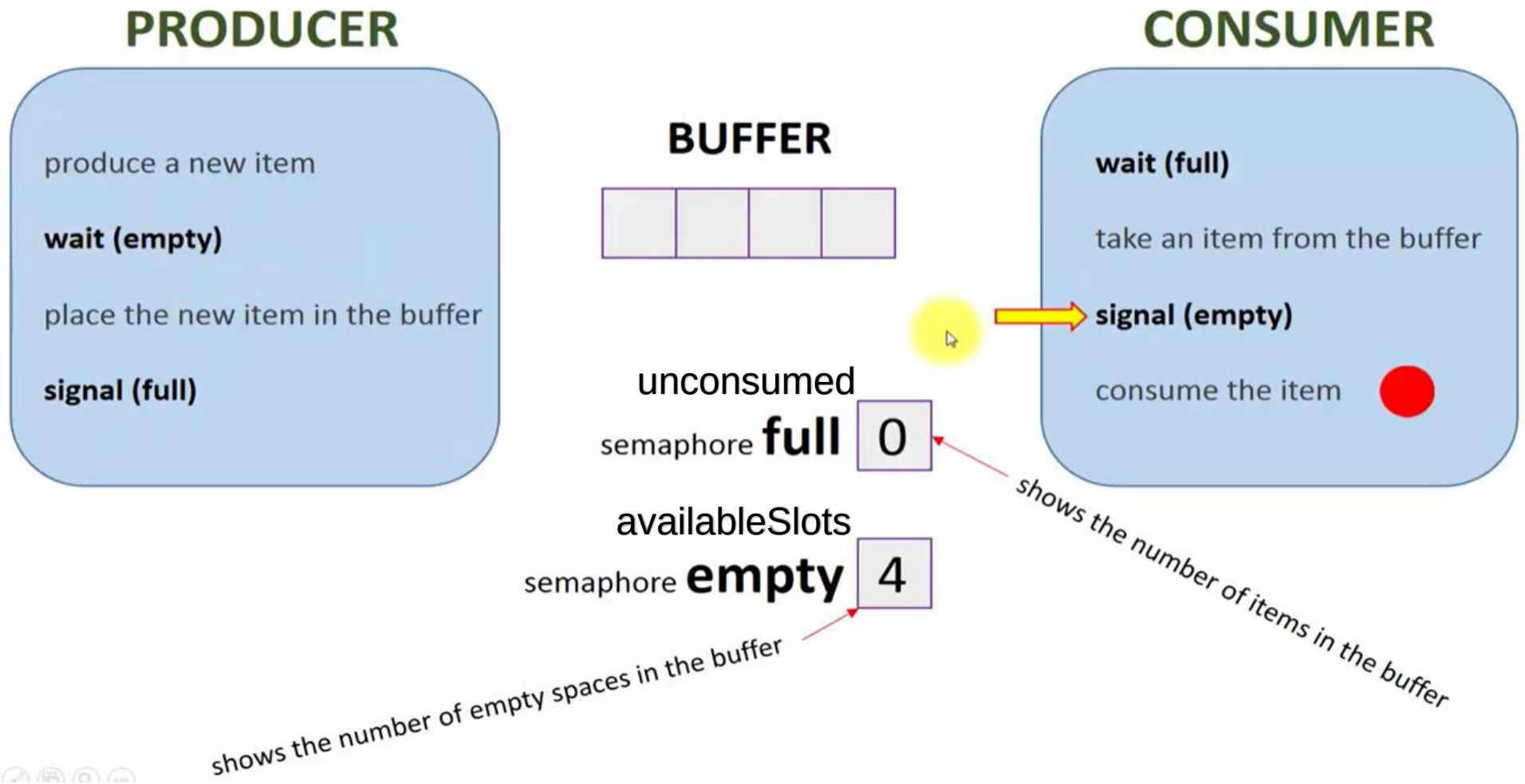
The producer/consumer (bounded buffer) problem solution with semaphores



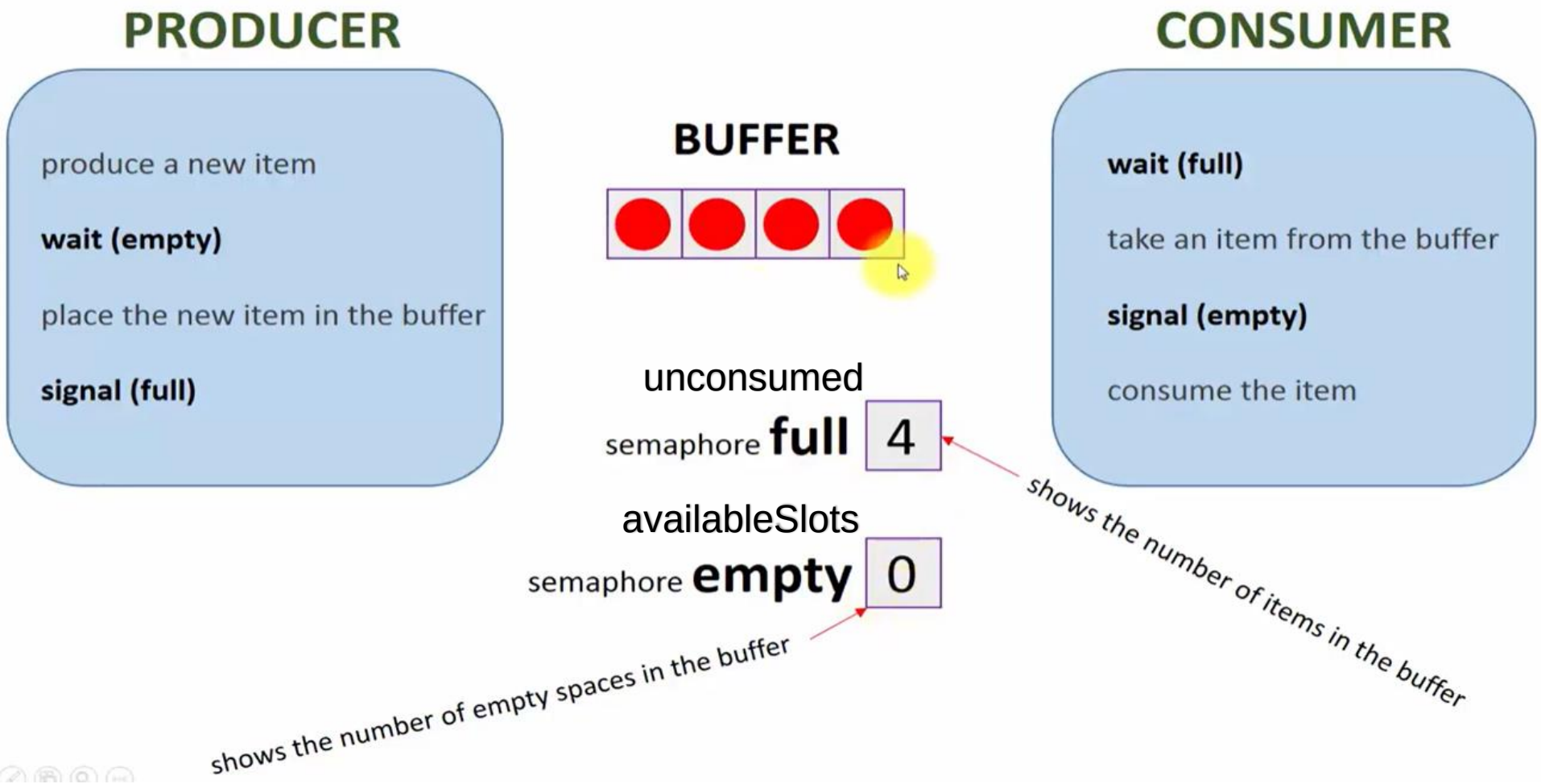
The producer/consumer (bounded buffer) problem solution with semaphores



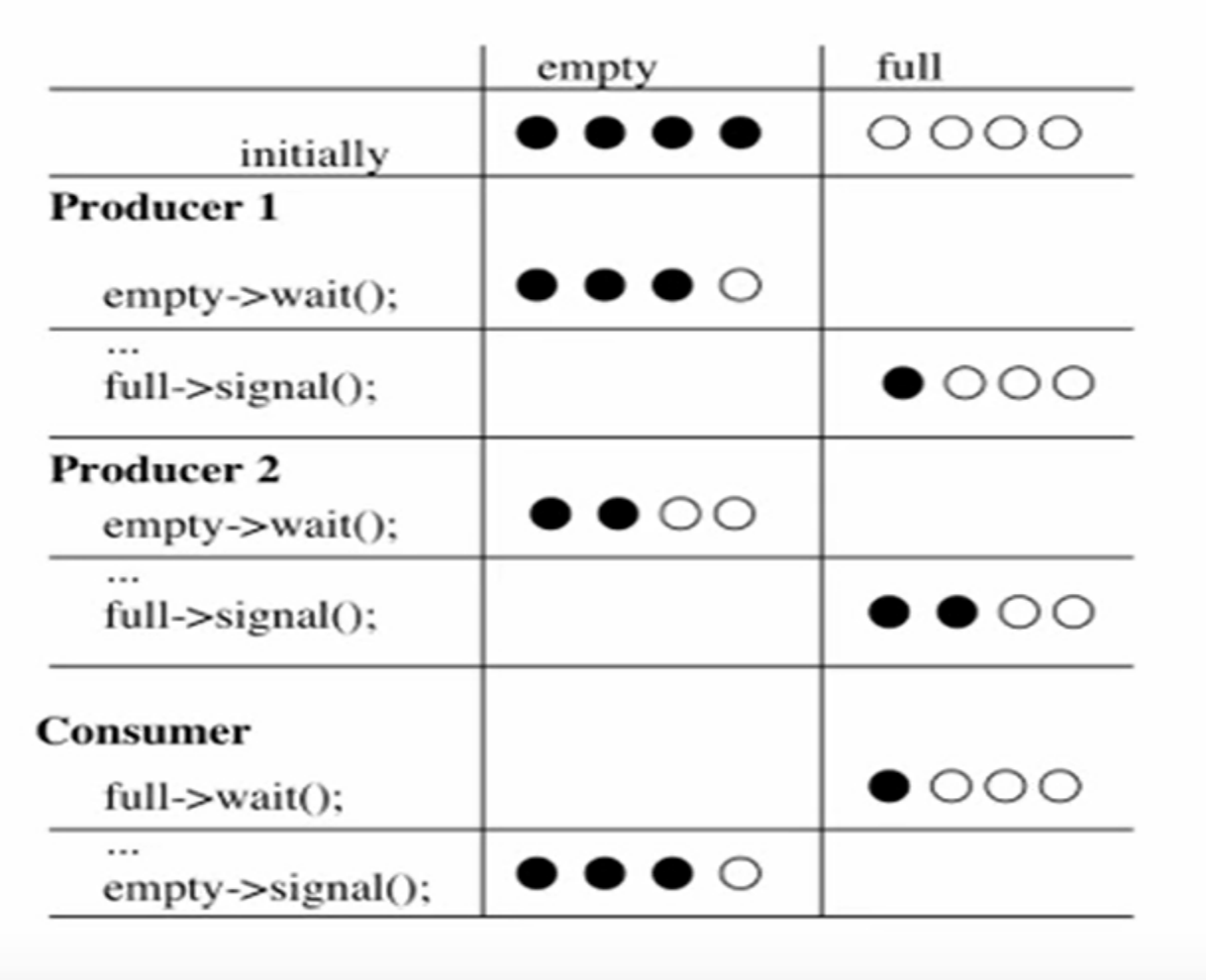
The producer/consumer (bounded buffer) problem solution with semaphores



The producer/consumer (bounded buffer) problem solution with semaphores



The producer/consumer (bounded buffer) problem solution with semaphores



	empty	full
initially	● ● ● ●	○ ○ ○ ○
Producer 1		
empty->wait();	● ● ● ○	
...		
full->signal();		● ○ ○ ○
Producer 2		
empty->wait();	● ● ○ ○	
...		
full->signal();		● ● ○ ○
Consumer		
full->wait();		● ○ ○ ○
...		
empty->signal();	● ● ● ○	

The producer/consumer (bounded buffer) problem solution with semaphores

```
/* for implementing the critical region */
shared semaphore mutex = 1;

/* items in buffer */
shared semaphore unconsumed = 0;

/* space in buffer */
shared semaphore availableSlots = BufferSize;

shared BufferADT buffer;    /* queue, tree, etc */
```

Producer process

```
void producer() {
      ItemType Item;
      while (true) {
            item = new ItemType(); //Producing an item
            /* make sure we have room */
            availableSlots.wait(); //down

            /* Access buffer exclusively */
            mutex.wait(); //down
            buffer.insert(item);
            mutex.signal(); //up

            unconsumed.signal();  //up, inform consumer
      }
}
```

Consumer process

```
void consumer() {
        ItemType item;
        while (true) {
                // Block until something to consume
                unconsumed.wait(); //down

                // Access buffer exclusively
                mutex.wait(); //down
                Item = buffer.remove();
                mutex.signal(); //up

                AvailableSlots.signal(); //up
                consume(item);    //consume or use Item
        }
}
```

Down semaphore implementation

```
down(Semaphore S) { //Non-textbook version
        S.value = S.value – 1;
        if (S.value < 0) {
                add(ProcessId, WaitingProcesses)
                set state to blocked;
        }
}
down(Semaphore S) { //Textbook version
        if (S.value > 0) {
                S.value = S.value – 1;
        }
        else {
                add(ProcessId, WaitingProcesses)
                set state to blocked;
        }
}
```

Up semaphore implementation

```
up(Sempahore S) {
    S.value = S.value + 1;
    if (S.value <= 0) {
        // At least one waiting process.
        // select a process to run
        NextProcess =
          SelectFrom(WaitingProcesses);
        set state of NextProcess to ready;
    }
}
```

Semaphore implementation

- Proposed implementation not atomic!
- Possible solutions?

Semaphore implementation

- Proposed implementation not atomic!

- Possible solutions
  - disable interrupts
  - hardware/software synchronization (Test&Set)
  - software synchronization

Semaphore implementation

Semaphores are themselves shared data and implementing WAIT and SIGNAL operations will require read/modify/write sequences that must executed as critical sections. So how do we guarantee mutual exclusion in these particular critical sections without using semaphores?

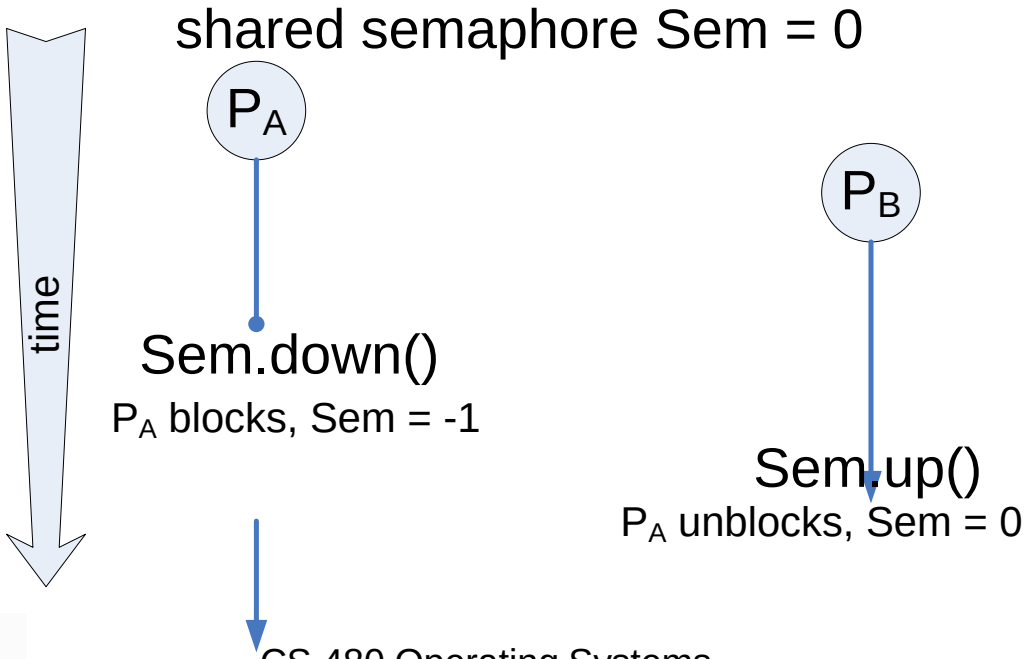
Approaches:

- SVC implementation, using atomicity of kernel handlers. Works in timeshared processor sharing a single uninterruptable kernel.
- Implementation of a simple lock using a special instruction (e.g. "test and set"), depends on atomicity of single instruction execution. Works with shared-bus multiprocessors supporting atomic read-modify-write bus transactions. Using a simple lock to implement critical sections, we can use software to implement other semaphore functionality.
- Implementation using atomicity of individual read or write operations. For example, see "Dekker's Algorithm" on Wikipedia.

For example, see "Dekker's Algorithm" on Wikipedia.

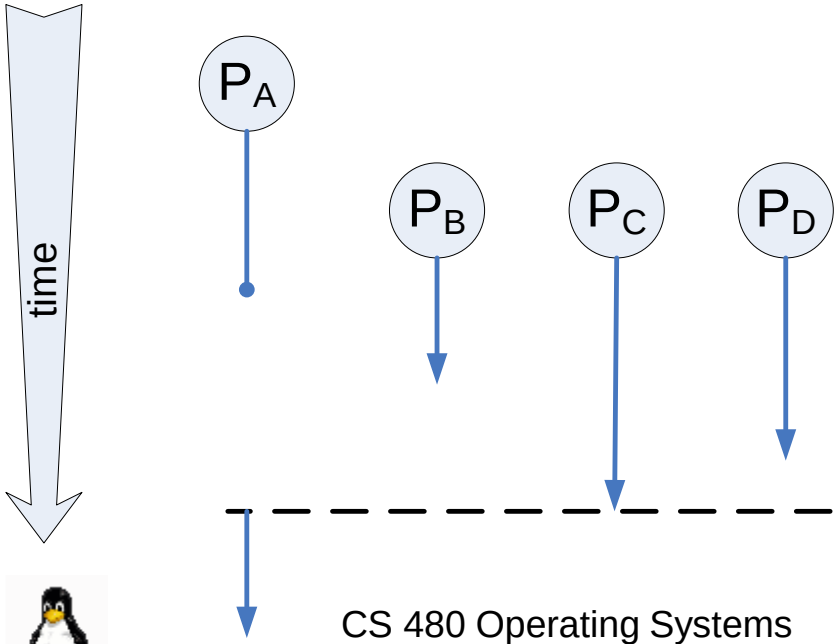
Barriers with semaphores (Precedence constraints)

- In general, when we want a process to block until something else occurs, we use a semaphore initialized to zero:

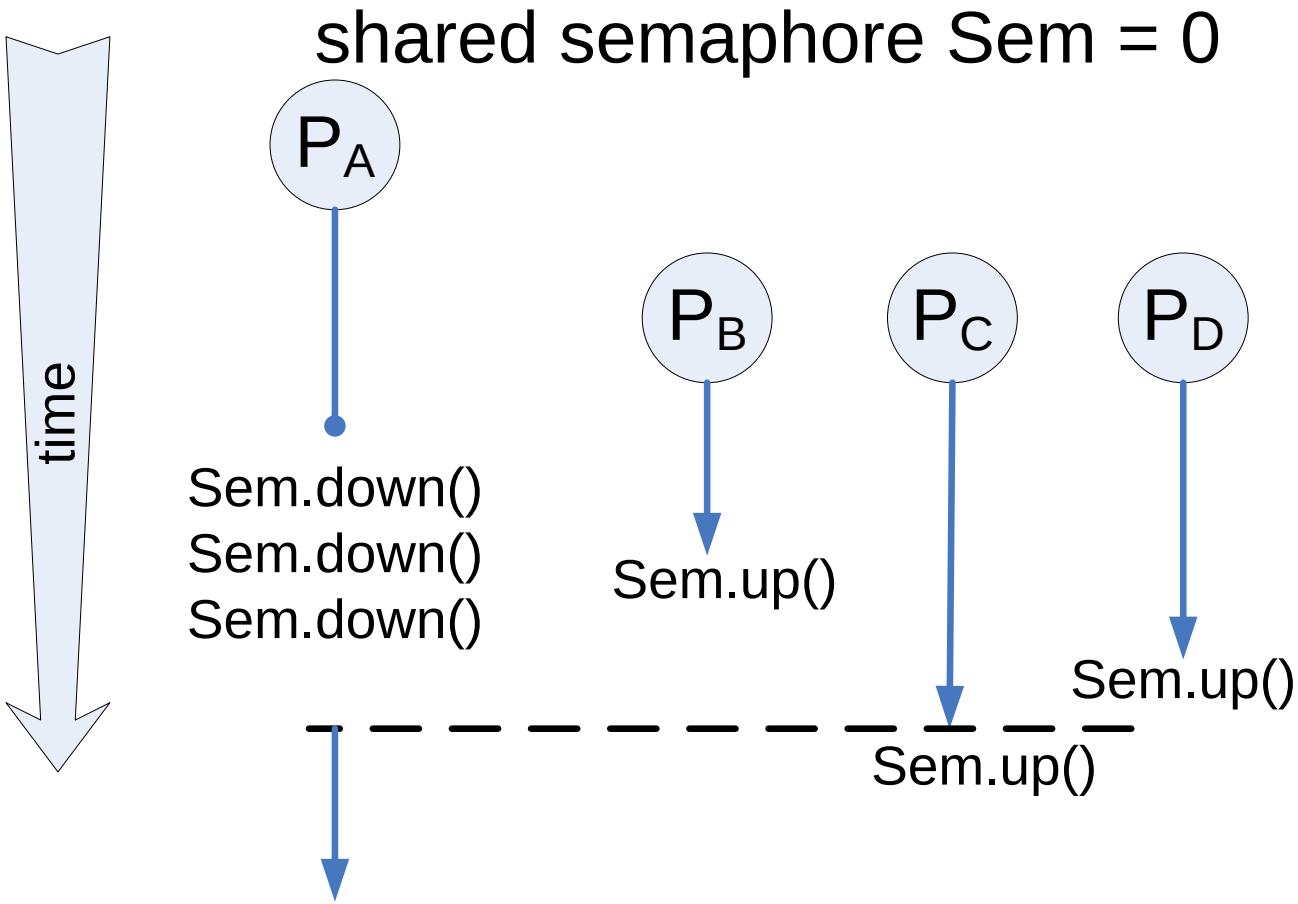


Barriers with semaphores

- Suppose P_A has spawned P_B, P_C, and P_D and we wish P_A to wait until its children have terminated:



Barriers with semaphores



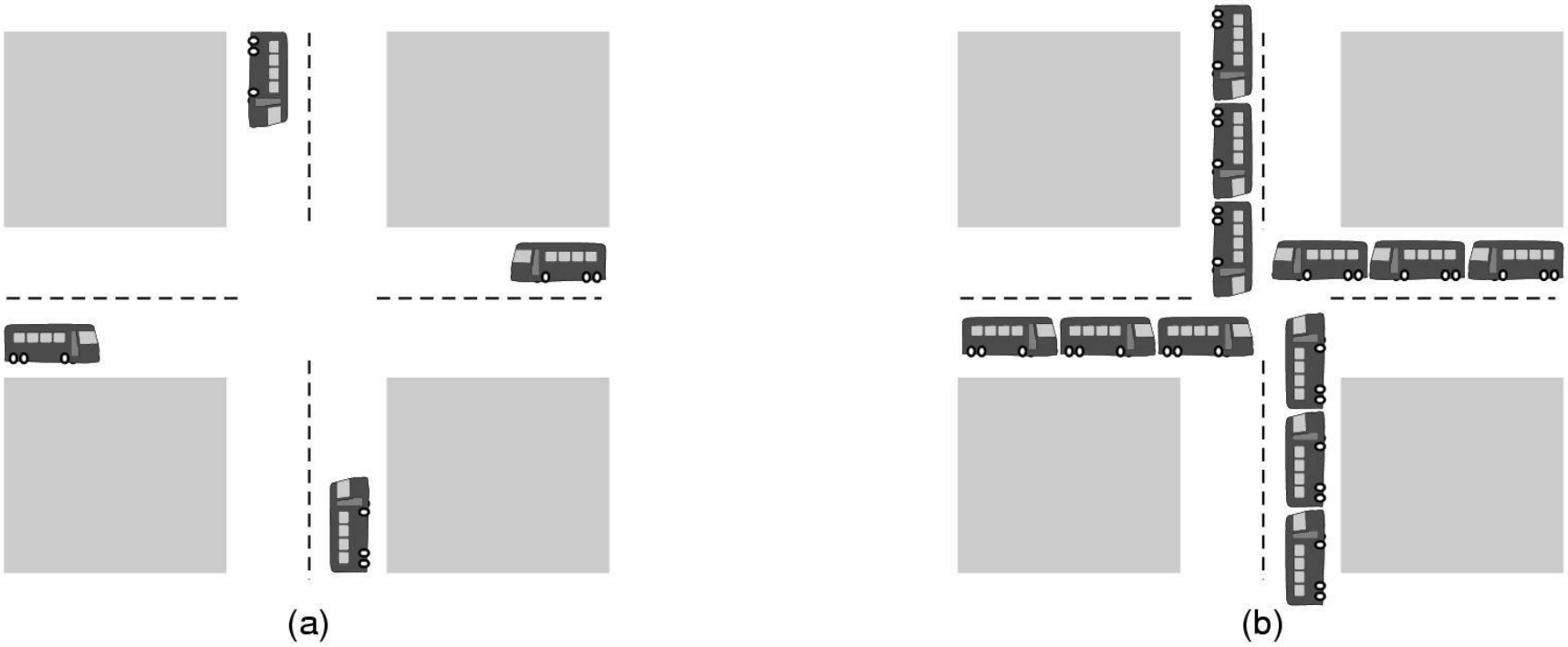
Barriers with semaphores – Another example (Scheduling / Precedence constraints)

- 3 Threads
  - Take CS240 (Machine Org and Assembly)
  - Take CS210 (Data Structures)
  - Take CS480 (Operating Systems)
  - How to ensure “Take CS480” to be done after “Take CS240” and “Take CS210”
    - S240, S210, what initial values should they have?
      - 0
    - CS480 – S240.Wait, S210.Wait
    - CS240 – S240.Signal
    - CS210 – S210.Signal

Semaphore summary

- Semaphores are generalized locks
- Semaphores use cases:
  - Ensure mutual exclusive access to critical sections – Binary Semaphore
  - Control access to a pool of resources – Counting Semaphore
  - Enforce certain ordering / precedence constraints between processes / threads
    - Ensure one thread to wait for specific action to be signaled from another thread.

Concepts: Deadlocks



(a) A potential deadlock. (b) an actual deadlock.

Classic coordination problems

- Dining philosophers (2.5.1)
  - Dijkstra's resource management problem
  - Philosophers think and eat, but need two utensils to eat.
  - How do we get them to eat without starving?

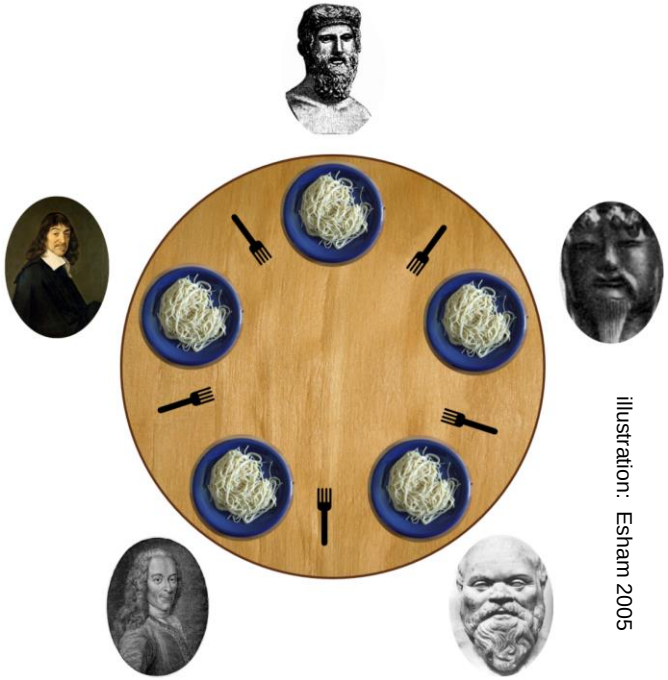


illustration: Esham 2005

Naïve implementation

```
N is number of philosophers

/* code for ith philosopher */
philosopher(i) {
        while (true) {
                think();  // deep thoughts…
                get_utensil(i);  // one on left
                get_utensil((i+1) % N) // one on right
                eat(); // fuel the brain (expensive organ)
                // put down utensils
                release_utensil(i);
                release_utensil((i+1) % N);
        }
}
```

With semaphores

```
// One to the left, one to the right
left(i) {return (i+N-1) % N;} // get index of the left
right(i) {return (i+1) % N;}  // get index of the right

shared int state[N];  // initialized to THINKING,
                      // possible states are THINKING, HUNGRY,
                      // EATING.
shared semaphore mutex = 1; // protect the access to shared data
shared semaphore s[N];  // Per philosopher sem init to 0.
                        // scheduling constraint
philosopher(i) {
        think();
         // Avoid "Hold and wait, need to access multiple resources"
        take_utensils();
        eat();
        release_utensils();
}
```

with semaphores

```
take_utensils(i) {
        mutex.down();  // critical section
        state[i] = HUNGRY;
        test(i);  // increment semaphore if we're good
        mutex.up(); // exit critical section
        s[i].down();  // blocks if no forks
}

test(i) {
        if (state[i] == HUNGRY &&
         state[left(i)] != EATING && state[right(i)] != EATING)
                s[i].up();
}
```

with semaphores

```
release_utensils(i) {
        mutex.down();  // critical section
        state[i] = thinking;
        // if neighbors were blocked, we might be able
        // to release them
        test(left(i));
        test(right(i));
        mutex.up(); // exit critical section
}
```

Classic coordination problems

- Readers and writers problem (in the book)
- Sleeping barber problem

You are not responsible for these, but you may want to read about them if you want more practice or think this is fun.

Problems with Semaphores

- Could easily make mistakes in using them
- No control of proper usage, why?
  - They are essentially shared global variables
  - No linguistic connection between semaphores and the data to which they control access
  - Access to semaphores could come from anywhere
  - They serve multiple purposes (mutex, counting semaphore, scheduling constraints…)
- Solution: a higher level programming construct called Monitor

Monitors (Hoare 1974/Hansen 1975)

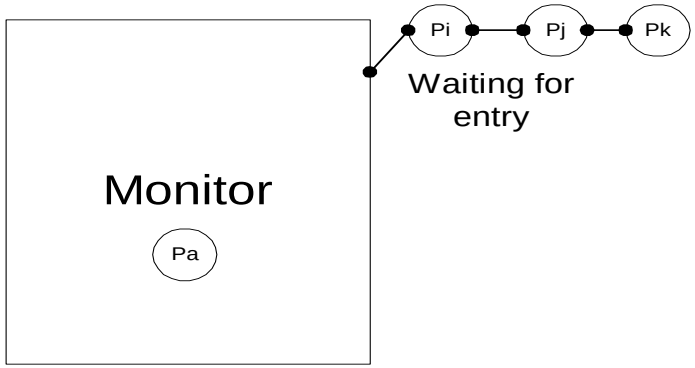
- Programming language construct which solves the critical region problem
- Sample monitor from a fictional language.
- A variant of monitors is supported in Java

```
monitor MonitorName {
    variable declarations

    procedure MonProc1(...) {
    }
    ...
    procedure MonProcN(...) {
    }

    MonitorName() {
        // initialization code
    }
}
```

Monitors

- Defines a lock and zero or more condition variables for managing concurrent access to shared data.
- Provides encapsulation of shared data.
  - Data declared in monitor.
  - Data cannot be accessed outside the monitor.
- Guarantees mutual exclusive access to the shared data.
- Only one active process in the monitor at any given time.
- Compiler generates critical region code

```
monitor MonitorName {
variable declarations
procedure MonProc1(...) {
}
...
procedure MonProcN(...) {
}
MonitorName() {
// initialization code
}
```



A monitor conundrum

- Suppose a process enters the monitor and finds a resource is not available.
- Example:

  Producer/Consumer problem

  Producer calls an AddItem(Item) method

  Buffer used for products is full.

  It would be nice to do something other than exit without adding the item and trying to invoke the method again...

Monitor producers/consumers

```
monitor ProducerConsumer {
        condition       unconsumed, availableSlots; // full, empty
        BufferADT       buffer;         // Some abstract data type for a buffer

        // insert adds an item to the shared buffer
        void insert (ItemType item) {
                boolean onlyItem;       // Will this be the only item in the buffer?

                lock.acquire();
                if (buffer.full())
                        ……      // How can we change here to wait until the buffer is
                                not full

                // If empty, then Item will be the only one after we add it
                onlyItem = buffer.empty();
                buffer.insert(item);

                if (onlyItem) {
                        ……...
                }
                lock.release();
        }
```

Monitors: condition variables

- Condition variables (associated with a monitor) allow us to do this.
  - Enable a thread to sleep inside a critical section
  - Lock held by the thread is atomically released when the thread is put to sleep
  - Wait….
- Abstract data type
  - Similar to semaphores
  - Two operations
    - wait – similar to semaphore wait (down)
    - signal – similar to semaphore signal (up) (or broadcast)
    - no need to initialize, value always initialized to 0

Monitors: condition variables

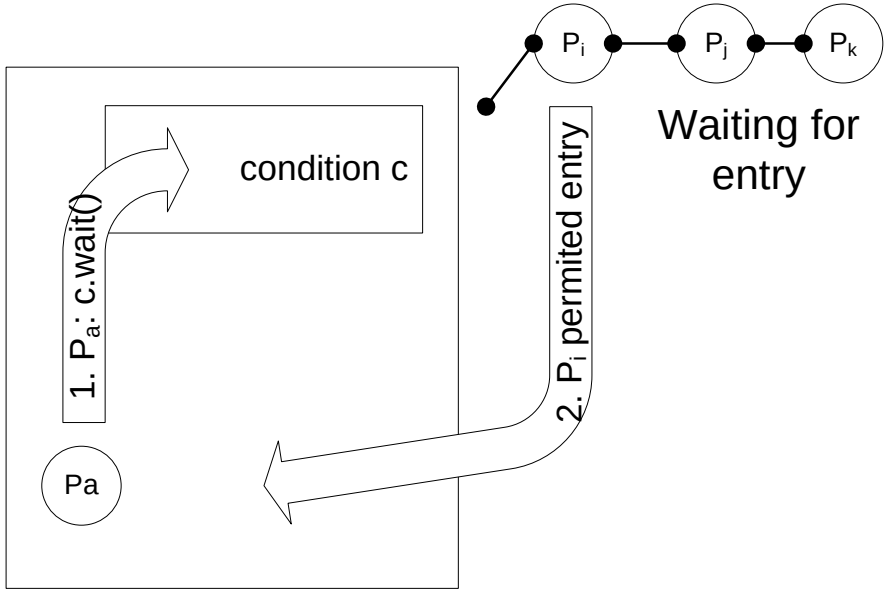
- Condition variable: Maintain a queue of processes / threads waiting for a condition inside a critical section
- Two operations:
  - Wait (monitor lock): release lock and go to sleep atomically, reacquires lock when the process / thread wakes up
  - Signal: wake a waiting process / thread if there exists one; otherwise, nothing happens
    - Signal broadcast: wake up all waiting processes / threads
- A process / thread must hold the monitor lock when doing condition variable operations.

Monitor producers/consumers

```
monitor ProducerConsumer {
        condition       unconsumed, availableSlots;
        BufferADT       buffer;         // Some abstract data type for a buffer

        // insert adds an item to the shared buffer
        void insert (ItemType Item) {
                boolean OnlyItem;       // Will this be the only item in the buffer?

                lock.acquire();

                if (buffer.full())
                        availableSlots.wait();  // sleep if no room

                // If empty, then Item will be the only one after we add it
                onlyItem = buffer.empty();
                buffer.insert(item);

                if (onlyItem) {
                        // wake up any consumer waiting for items
                        unconsumed.signal();
                }

                lock.release();
```

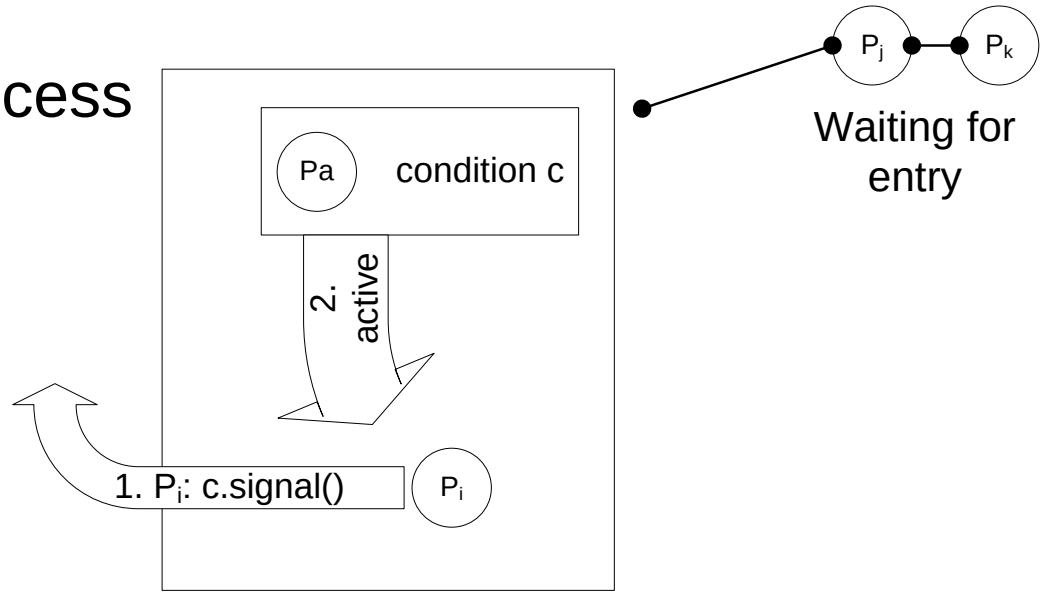
wait operation

- Blocks caller and puts caller to sleep.
- As caller is no longer active, next process allowed to enter monitor.



signal operation

```
if no process blocked on condition variable {
    no op
} else {
    exit from monitor
    start blocked process
}
```



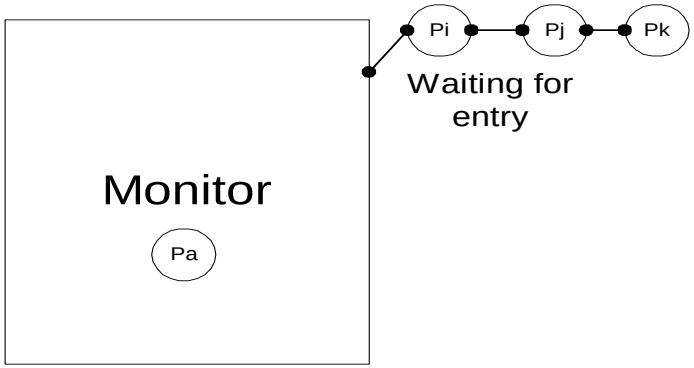
This behavior was specified by Brinch Hansen, there are other possibilities which we will not study. They vary only in the else clause.

signal operation

- 3 different thoughts on transferring monitor access from a signaling process to wakened processes:
  - Brinch Hansen: Signaling process is ejected from monitor immediately (we'll assume this)
  - Hoare: Suspend signaling process, wakened process runs
  - 3rd solution (Mesa Style): Signaling process continues to run, wakened process executes only after signaling process leaves monitor. **Most operating systems using this solution**

Monitor operations

- Encapsulates the shared data you want to operate.
- Acquires the mutex at the start.
- Operates on the shared data.
- Temporarily releases the mutex if it can’t complete. (using condition variables)
- Reacquire the mutex when it can continue.
- Release the mutex at the end.



Monitor producers/consumers

```
monitor ProducerConsumer {
        condition       unconsumed, availableSlots;
        BufferADT       buffer;         // Some abstract data type for a buffer

        // insert adds an item to the shared buffer
        void insert (ItemType Item) {
                boolean OnlyItem;       // Will this be the only item in the buffer?

                lock.acquire();

                if (buffer.full())
                        availableSlots.wait();  // sleep if no room

                // If empty, then Item will be the only one after we add it
                onlyItem = buffer.empty();
                buffer.insert(item);

                if (onlyItem) {
                        // wake up any consumer waiting for items
                        unconsumed.signal();
                }

                lock.release();
```

Monitor producers/consumers

```
// remove item from the shared buffer
ItemType remove() {
        ItemType        item;
        boolean         atCapacity;     // Is buffer currently at capacity?

        lock.acquire();

        if (buffer.empty()) //Hansen or Hoare style; if using mesa style, it
                            //needs to be using while (Buffer.empty()) here
                unconsumed.wait();    // sleep until producer signals

        // If full, there will be one space in the buffer after we remove
        atCapacity = buffer.full();

        Item = buffer.remove();

        if (atCapacity) {
                // We have moved from being at capacity to one
                // under capacity.  Signal any producer who might
                // be waiting to add items.
                availableSlots.signal();
        }

        lock.release();
    }
} // end monitor
```

Monitor producers/consumers

Monitor ProducerConsumer is shared by both processes, the following is separate.

Producer process:

```
void producer {
        ItemType        Item;
        while (true) {
                Item = new ItemType;
                ProducerConsumer.insert(Item);
        }
}
```

Consumer process:

```
void consumer {
        ItemType Item;
        while (true) {
                Item = ProducerConsumer.remove();
                process(Item);
        }
}
```

Monitor producers/consumers
A Java implementation

```
static class our_monitor {  // this is a monitor
   private int buffer[ ] = new int[N];
   private int count = 0, lo = 0, hi = 0;  // counters and indices

   public synchronized void insert(int val) {
      if (count == N) go_to_sleep( );    // if the buffer is full, go to sleep
      buffer [hi] = val; // insert an item into the buffer
      hi = (hi + 1) % N;       // slot to place next item in
      count = count + 1;       // one more item in the buffer now
      if (count == 1) notify( );       // if consumer was sleeping, wake it up
   }

   public synchronized int remove( ) {
      int val;
      if (count == 0) go_to_sleep( );    // if the buffer is empty, go to sleep
      val = buffer [lo]; // fetch an item from the buffer
      lo = (lo + 1) % N;       // slot to fetch next item from
      count = count – 1;       // one few items in the buffer
      if (count == N – 1) notify( ); // if producer was sleeping, wake it up
      return val;
   }
   private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

Monitor producers/consumers - A Java implementation

```
public class BlockingQueue<T> {

    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(T element) throws InterruptedException {
        while(queue.size() == capacity) {
            wait();
        }

        queue.add(element);
        notify(); // notifyAll() for multiple producer/consumer threads
    }

    public synchronized T take() throws InterruptedException {
        while(queue.isEmpty()) {
            wait();
        }

        T item = queue.remove();
        notify(); // notifyAll() for multiple producer/consumer threads
        return item;
    }
}
```

Monitor producers/consumers - A Java implementation

```
public class BlockingQueue<T> {
   private Queue<T> queue = new LinkedList<T>();
   private int capacity;
   private Lock lock = new ReentrantLock();
   private Condition notFull = lock.newCondition();
   private Condition notEmpty = lock.newCondition();

   public BlockingQueue(int capacity) {
      this.capacity = capacity;
   }
   public void put(T element) throws InterruptedException {
      lock.lock();
      try {
         while(queue.size() == capacity) {
            notFull.await();
         }
         queue.add(element);
         notEmpty.signal();
      } finally {
         lock.unlock();
      }
   }
   public T take() throws InterruptedException {
      lock.lock();
      try {
         while(queue.isEmpty()) {
            notEmpty.await();
         }

         T item = queue.remove();
         notFull.signal();
         return item;
      } finally {
         lock.unlock();
      }
   }
}
```

Monitor Summary

- A monitor defines a mutex and 0 or more condition variables for managing concurrent access to shared data
  - Uses the mutex (lock) to ensure that only one process / thread can be active in the monitor at once, provides mutual exclusive access to shared data
  - Condition variables enables threads to go to sleep inside the monitor and release its lock atomically

Semaphores vs Condition Variables

- Semaphores remember history (states), condition variables do NOT
  - Wait:
    - semaphore decremented, thread may or may not block;
    - condition variable - thread will block
  - Signal:
    - semaphore incremented
    - condition variable – does nothing if no thread is waiting, or wake up a blocked thread

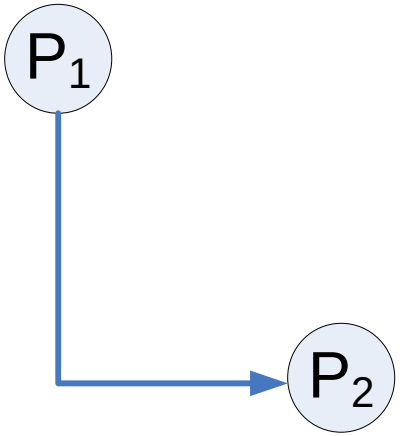
Message passing

- Inter-process communication without need for shared memory

- Primitives
  - send(destination, &message)
  - receive(source, &message)

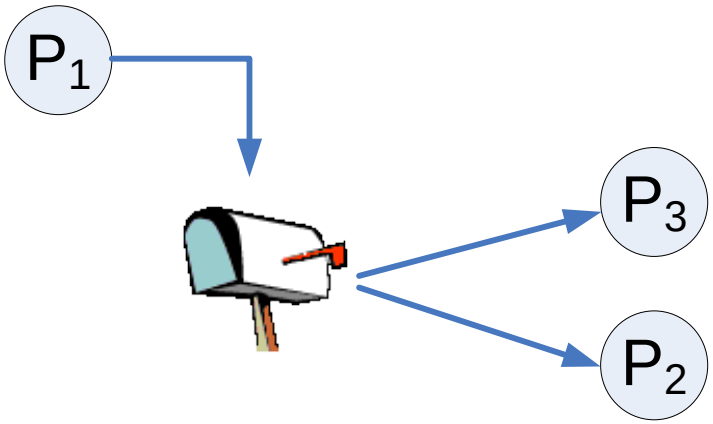
Defining source and destination

- Processes



  - processes must have unique names
  - send(destination, &message)
  - receive(source, &message)

- Mailboxes (ports)



  - mailboxes must have unique names
  - multiple receivers possible

Implementation issues

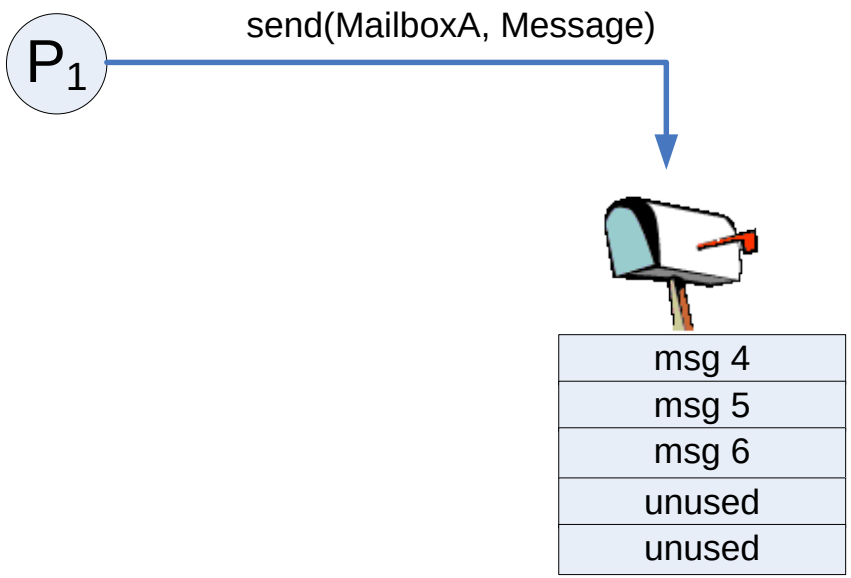
- Assuring reliable delivery
- Naming processes/mailboxes uniquely
- Buffer size

Message delivery

- We will assume reliable delivery
- In reality
  - Messages can be lost on networks
  - Reliable delivery subject of networking class
  - Basic idea (better schemes exist)
    - Receiver sends acknowledgement
    - If sender does not receive acknowledgement within reasonable amount of time, retransmit message.
    - What if the acknowledgement is lost.
  - Message passing efficiency

Buffers

- Messages must be stored in a temporary area until the receiver retrieves them.
- Process blocks if there is not enough room.



Buffer capacity

- Size of buffer affects behavior
  - zero: Sender blocks until receiver reads message.
    - Enforces coordination, known as a rendez-vous.
    - Can be used for remote procedure calls
  - bounded: Asynchronous call as long as there is room in buffer
  - unbounded: Always asynchronous

Producer consumer problem

```
void producer() {
        ItemType Item;

        while (true) {
                Item = new ItemType();

                /* Note that in many
                 * implementations
                 * we may have place Item
                 * inside a Message
                 * structure and then send
                 * the Message
                 */

                /* Send item to consumer */
                send(Consumer, Item);
        }
}
```

```
/* consumer process */
void consumer() {
        ItemType Item;

        while (true) {
                // get Item
                receive(Producer, Item);
                // use Item
                consume(Item);
        }
}
```

Producer consumer problem

```
void producer(void)
{
     int item;
     message m;                          /* message buffer */

     while (TRUE) {
          item = produce_item( );        /* generate something to put in buffer */
          receive(consumer, &m);         /* wait for an empty to arrive */
          build_message(&m, item);       /* construct a message to send */
          send(consumer, &m);            /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);         /* get message containing item */
          item = extract_item(&m);       /* extract item from message */
          send(producer, &m);            /* send back empty reply */
          consume_item(item);            /* do something with the item */
     }
}
```

Figure 2-36. The producer-consumer problem with N messages.