Interprocess Communication

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.

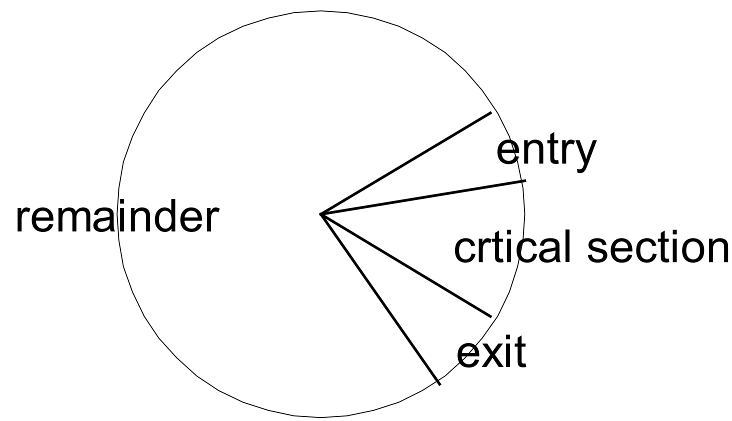
Critical sections/regions (informal)

- 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

Critical regions must meet the following 3 conditions:

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

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 process */
   interested[process] = true;
   turn = process;   /* set flag */

   /* Busy-wait until the following is true:
    * 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;
}
```

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 cannot be interrupted.

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?

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 contains an integer variable which is accessed by two operations known by many different names:

P (test “prohoben”)	wait	down*
V (increment “verhogen”)	signal	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

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

- up – Increment the counter value. If processes have blocked on the semaphore, unblock one of the processes.

Critical regions & semaphores

```
shared semaphore Sem = 1;

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

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

Ensuring 0 + 1 – 1 = 0 (again):

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

P1

```
Sem.down();
x++;
Sem.up();
```

P2

```
Sem.down();
x--;
Sem.up();
```

The producer/consumer 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();
            /* make sure we have room */
            AvailableSlots.down();

            /* Access buffer exclusively */
            mutex.down();
            Buffer.Insert(Item);
            mutex.up();

            Unconsumed.up();  /* inform consumer */
      }
}
```

Consumer process

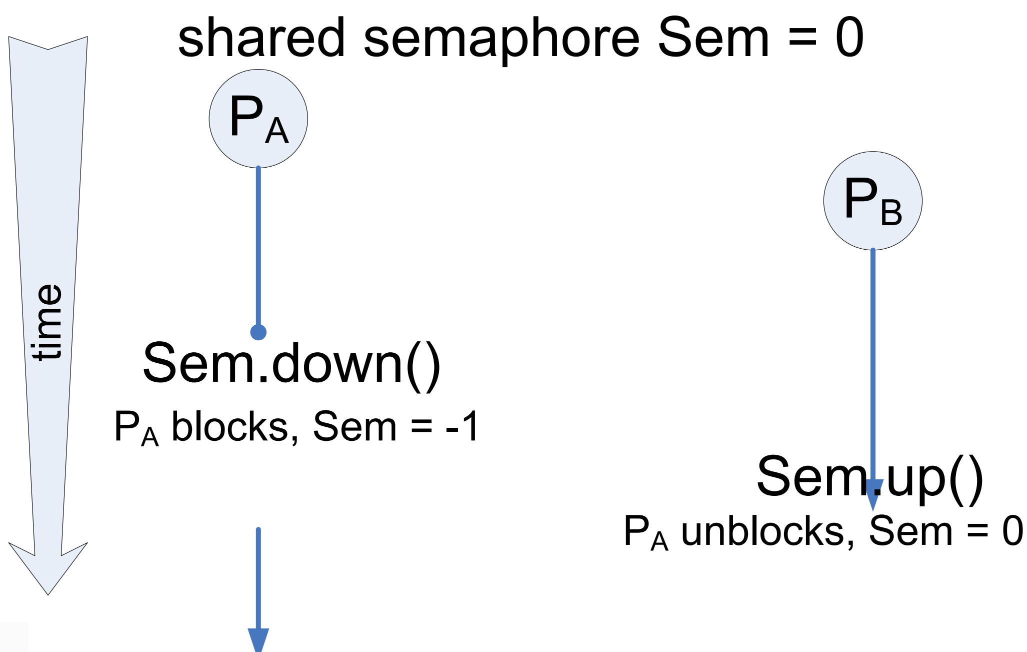
```
void consumer() {
	ItemType Item;
	while (true) {
		// Block until something to consume
		Unconsumed.down();

		// Access buffer exclusively
		mutex.down();
		Item = Buffer.Remove();
		mutex.up();

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

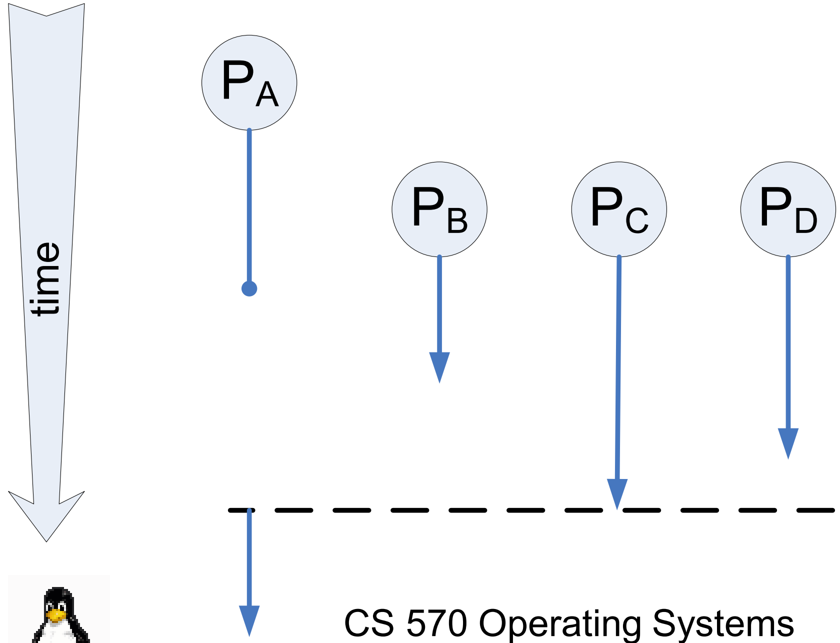
Barriers with semaphores

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



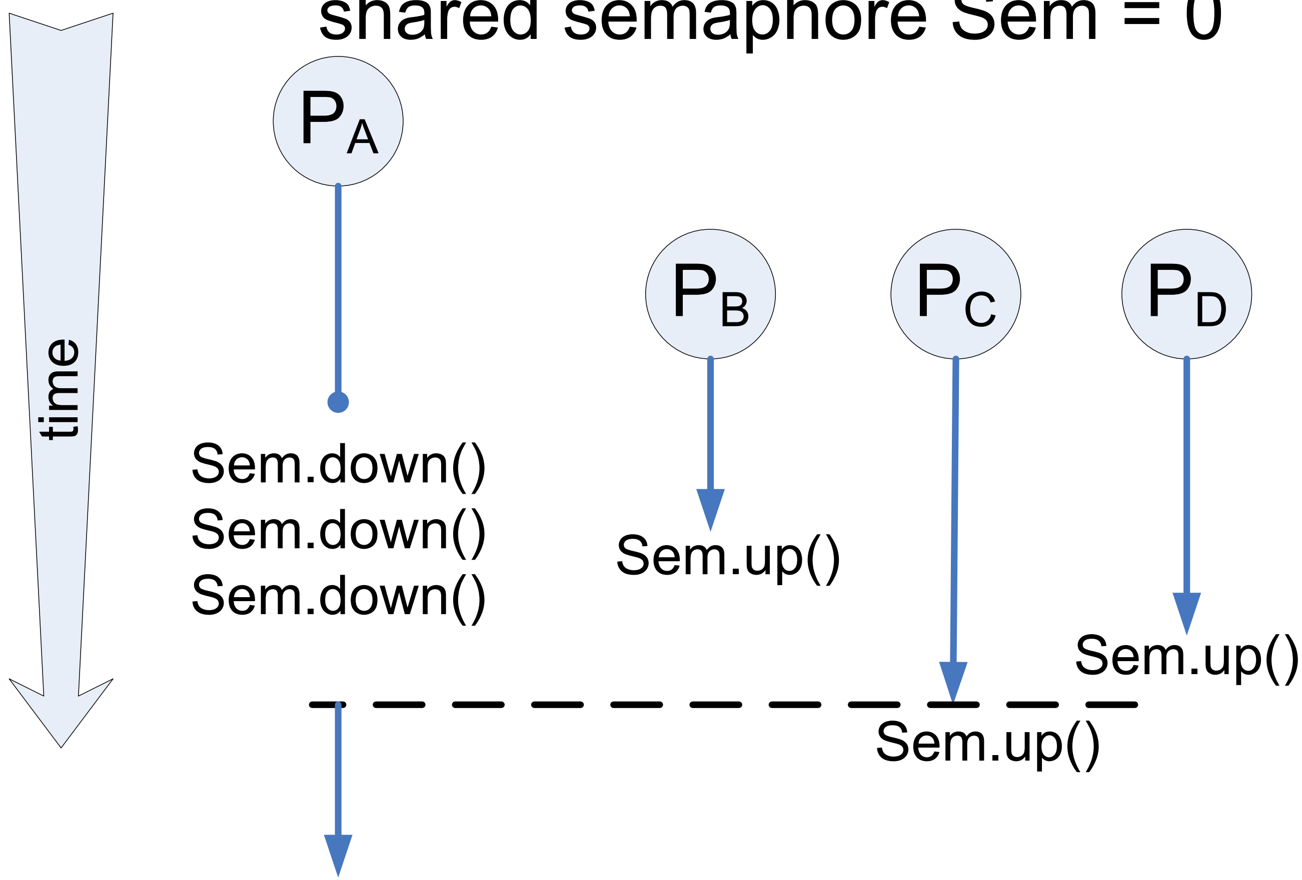
Barriers with semaphores

- 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

shared semaphore Sem = 0



Down semaphore implementation

```
down(Semaphore S) {
    S.value = S.value – 1;
    if (S.value < 0) {
        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
  - software synchronization

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?

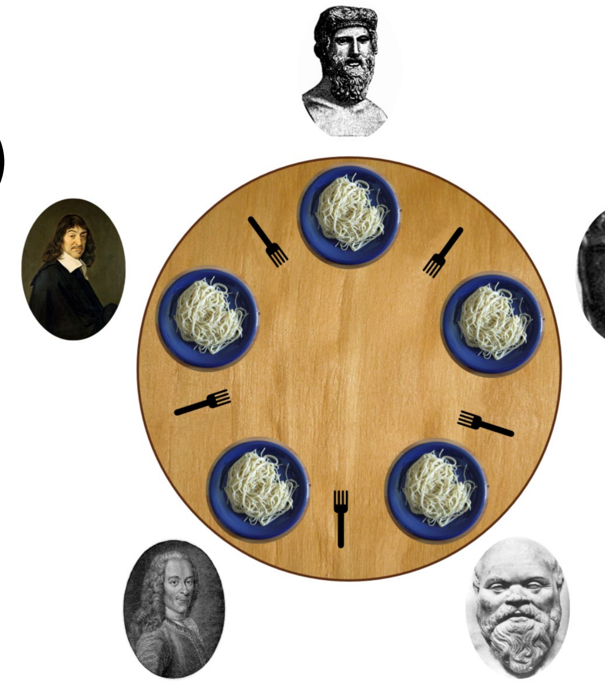


illustration: Esham 2005

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;}
right(i) {return (i+1) % N;}

shared int state[N];  // all initialized to THINKING
shared semaphore mutex = 1;
shared semaphore s[N];  // Per philosopher sem init to 0.

philosopher(i) {
       think();
       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) {
                state[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
        s[i].down();  // blocks if no forks
}
```

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.

mutexes

- Specialized semaphore
- Behaves as a semaphore initialized to 1
- Read section 2.3.6

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

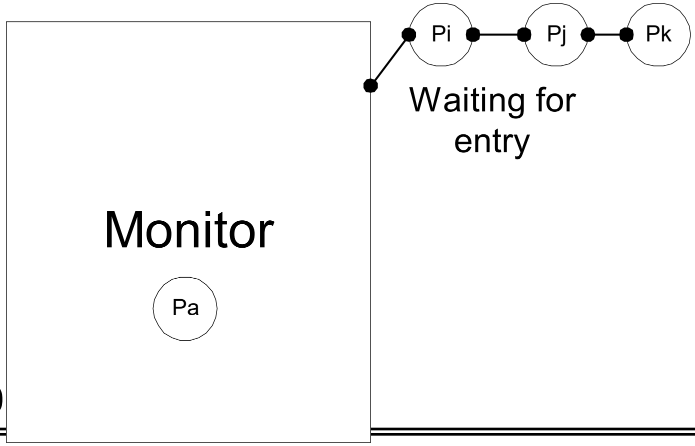
```
monitor MonitorName {
    variable declarations

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

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

Monitors

- Provides encapsulation of shared data.
  - Data declared in monitor.
  - Data cannot be accessed outside the monitor.
- Compiler generates critical region code
- Only one active process in the monitor at any given time.



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.

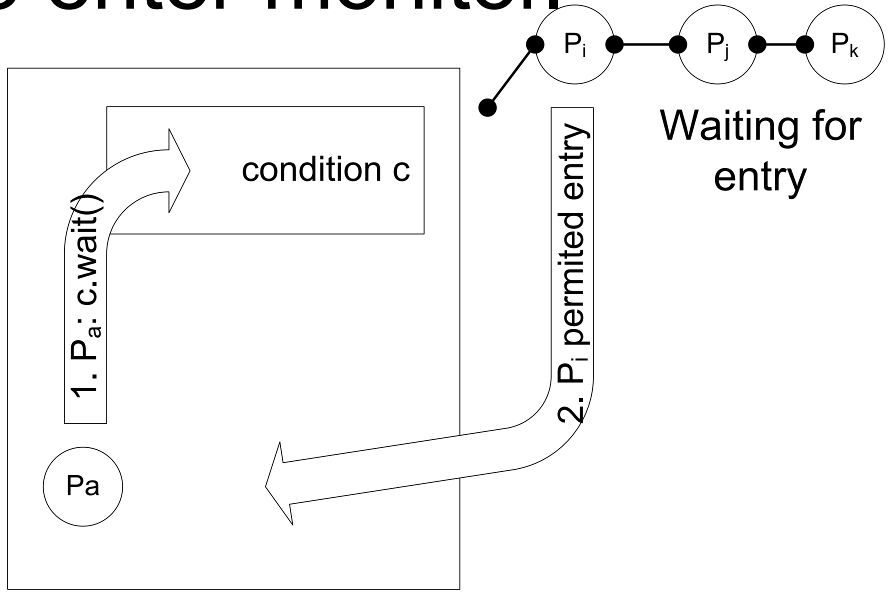
    It would be nice to do something other than exit without adding the item and trying to invoke the method again...

Monitors: condition variables

- Condition variables allow us to do this.
- Abstract data type
  - Similar to semaphores
  - Two operations
    - wait – similar to semaphore wait (down)
    - signal – similar to semaphore signal (up)
    - no need to initialize, value always initialized to 0

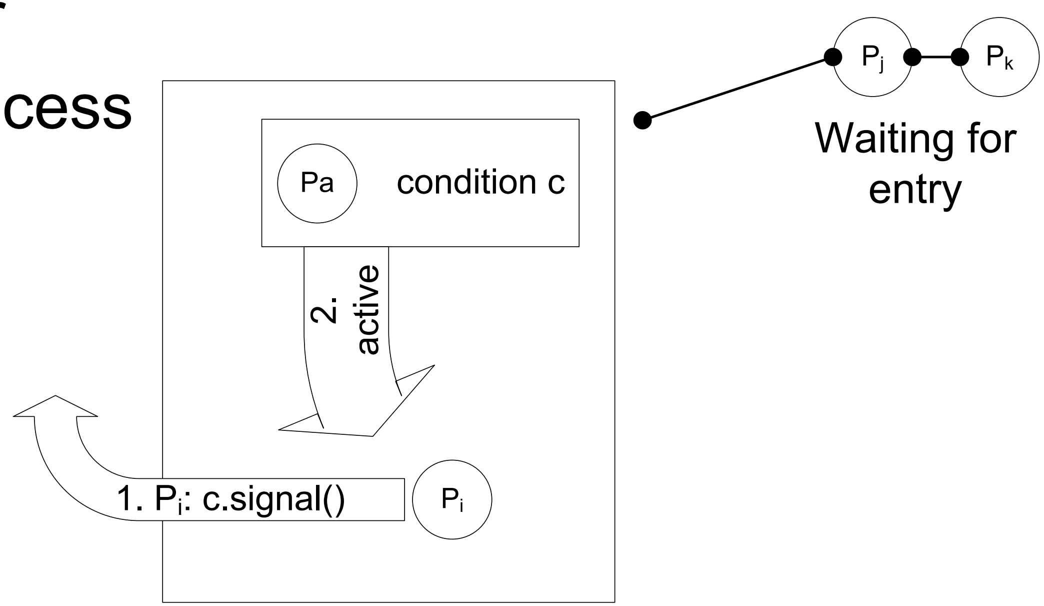
wait operation

- Decrements counter and blocks caller.
- 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.

Monitor producer/consumer

```
monitor ProducerConsumer {
        condition       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?

                if (Buffer.full())
                        full.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
                        empty.signal();
                }
        }
```

Monitor producer/consumer

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

        if (Buffer.empty())
            empty.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.
            full.signal();
        }
    }
} // end monitor
```

Monitor producer/consumer

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);
        }
}
```

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;
```

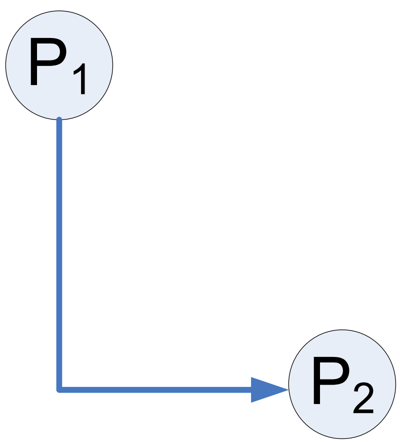
Message passing

- Interprocess communication without need for message passing

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

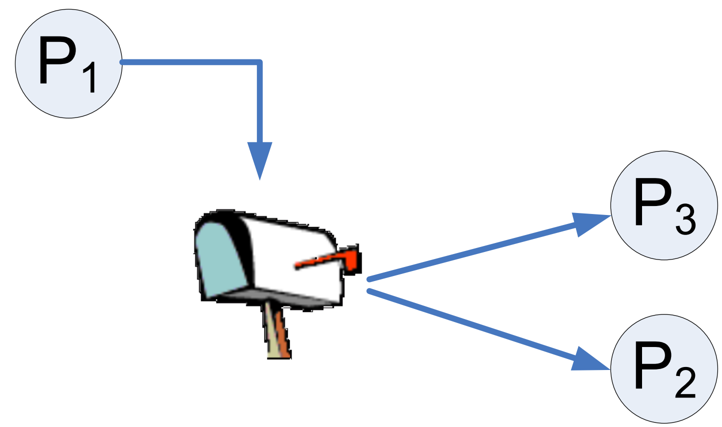
Defining source and destination

- Processes



  - processes must have unique names

- Mailboxes (ports)



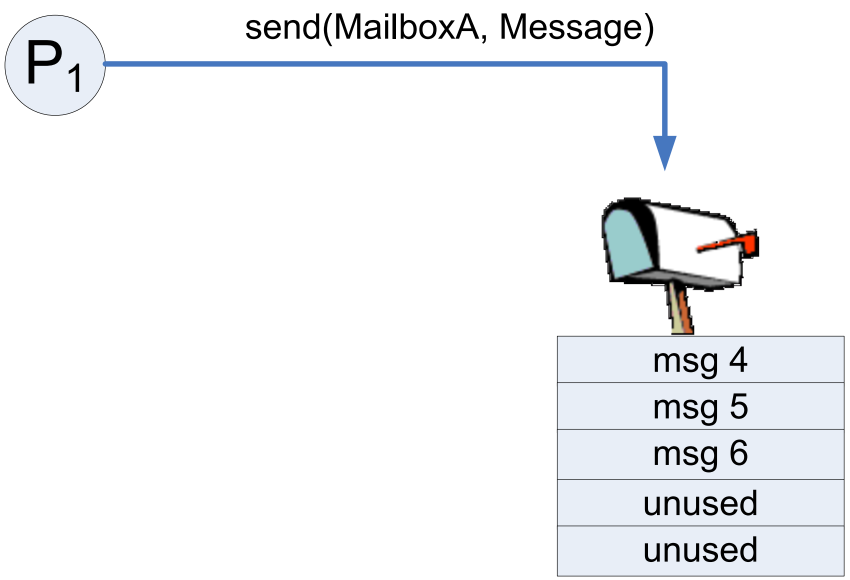
  - mailboxes must have unique names
  - multiple receivers possible

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.

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

Implementation issues

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

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);
        }
}
```