Process Management: Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- More sophisticated synchronization mechanisms: Monitors, Message Passing
- Classical synchronization problems

Process Management: Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- More sophisticated synchronization mechanisms: Monitors, Message Passing
- Classical synchronization problems

The Critical Section Problem: Example 1

```
void echo() {
  input(in, keyboard);
  out := in;
  output(out, display);
}
```

```
char in;  /* shared variables */
char out;
```

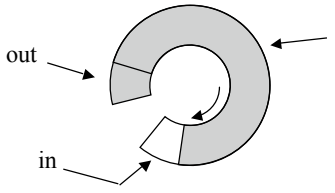
	Process 1	*Process 2*
Operation:	`Echo()`	`Echo()`

Interleaved execution	`/`
	`input(in,keyboard)`
	`out = in;`
	`/`
	`/`
	`/`
	`output(out,display)`
		`/`
		`/`
		`/`
		`input(in,keyboard);`
		`out = in;`
		`output(out,display);`
		`/`

Race condition !

The Critical Section Problem: Example 2

Producer-consumer with bounded, shared-memory, buffer.



circular buffer of size *n*

```
int    in, out;
Item   buffer[n];
int    counter;
```

Producer:

```
void deposit(Item * next) {
  while (counter == n) no_op;
  buffer[in] = next;
  in = (in+1) MOD n;
  counter = counter + 1;
}
```

Consumer:

```
Item * remove() {
  while (counter == 0) no_op;
  next = buffer[out];
  out = (out+1) MOD n;
  counter = counter - 1;
  return next;
}
```

This Implementation is not Correct!

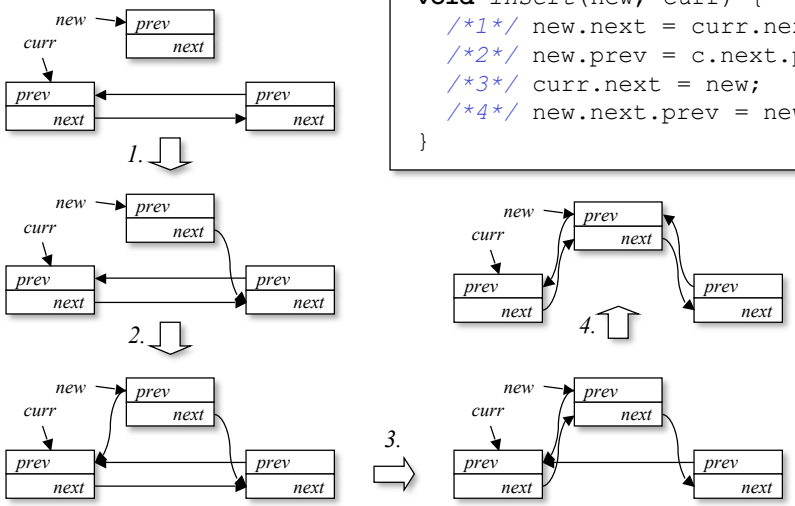
	Producer	Consumer
operation:	`counter = counter + 1`	`counter = counter - 1`
on CPU:	reg_1 = counter reg_1 = reg_1 + 1 counter = reg_1	reg_2 = counter reg_2 = reg_2 - 1 counter = reg_2
interleaved execution:	reg_1 = counter reg_1 = reg_1 + 1 counter = reg_1	 reg_2 = counter reg_2 = reg_2 - 1 counter = reg_2

- **Race condition!**
- Need to ensure that only one process can manipulate variable `counter` at a time : **synchronization**.

Critical Section Problem: Example 3

Insertion of an element into a list.

```
void insert(new, curr) {
   /*1*/ new.next = curr.next;
   /*2*/ new.prev = c.next.prev;
   /*3*/ curr.next = new;
   /*4*/ new.next.prev = new;
}
```



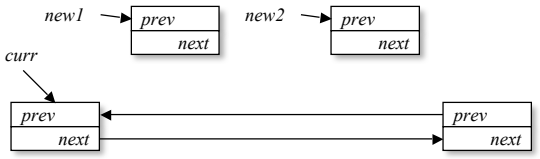
Interleaved Execution causes Errors!

Process 1

```
new1.next = curr.next;
new1.prev = c.next.prev;
…
…
…
…
curr.next = new1;
new.next.prev = new1;
```

Process 2

```
…
…
new2.next = curr.next;
new2.prev = c.next.prev;
curr.next = new2;
new.next.prev = new2;
…
…
```



- Must guarantee mutually exclusive access to list data structure!

Process Management: Synchronization

- Why? Examples
- What? The Critical Section Problem
- How? Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- More sophisticated synchronization mechanisms: Monitors, Message Passing
- Classical synchronization problems

Critical Sections

- Execution of critical section by processes must be mutually exclusive.
- Typically due to manipulation of shared variables.
- Need protocol to enforce mutual exclusion.

```
while (TRUE) {
   enter section;
   critical section;
   exit section;
   remainder section;
}
```

Criteria for a Solution of the C.S. Problem

1. Only one process at a time can enter the critical section.
2. A process that halts in non-critical section cannot prevent other processes from entering the critical section.
3. A process requesting to enter a critical section should not be delayed indefinitely.
4. When no process is in a critical section, any process that requests to enter the critical section should be permitted to enter without delay.
5. Make no assumptions about the relative speed of processors (or their number).
6. A process remains within a critical section for a finite time only.

A (Wrong) Solution to the C.S. Problem

- Two processes P_0 and P_1
- `int turn;` /* `turn == i` : P_i is allowed to enter c.s. */

```
Pi: while (TRUE) {
      while (turn != i) no_op;

      critical section;

      turn = j;

      remainder section;
    }
```

Another Wrong Solution

`bool flag[2];` /* initialize to `FALSE` */
/* `flag[i] == TRUE` : P_i intends to enter c.s.*/

```
Pi: while (TRUE) {
      while (flag[j]) no_op;
      flag[i] = TRUE;

      critical section;

      flag[i] = FALSE;

      remainder section;
    }
```

Yet Another Wrong Solution

```
bool flag[2]; /* initialize to FALSE */
  /* flag[i] == TRUE : Pi intends to enter c.s.*/

    while (TRUE) {

      flag[i] = TRUE;
      while (flag[j]) no_op;

      critical section;

      flag[i] = FALSE;

      remainder section;

    }
```

A Combined Solution (Petersen)

```
int turn;
bool flag[2]; /* initialize to FALSE */

    while (TRUE) {

      flag[i] = TRUE;
      turn    = j;
      while (flag[j]) && (turn == j) no_op;

      critical section;

      flag[i] = FALSE;

      remainder section;

    }
```

Process Management: Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- More sophisticated synchronization mechanisms: Monitors, Message Passing
- Classical synchronization problems

Hardware Support For Synchronization

- **Disallow interrupts**
  - simplicity
  - widely used
  - problem: interrupt service latency
  - problem: what about multiprocessors?
- **Atomic** operations:
  - Operations that check and modify memory areas in a single step (i.e. operation can not be interrupted)
  - **Test-And-Set**
  - **Exchange, Swap, Compare-And-Swap**

Test-And-Set

```
bool TestAndSet(bool & var) {
    bool temp;
    temp = var;
    var  = TRUE;
    return temp;
}
```

atomic!

Mutual Exclusion with *Test-And-Set*

```
bool lock; /* init to FALSE */

while (TRUE) {
    while (TestAndSet(lock)) no_op;

    critical section;

    lock = FALSE;

    remainder section;
}
```

Exchange (Swap)

```
void Exchange(bool & a, bool & b){
    bool temp;
    temp = a;
    a    = b;
    b    = temp;
}
```

atomic!

Mutual Exclusion with *Exchange*

```
bool lock; /*init to FALSE */

while (TRUE) {
    dummy = TRUE;
    do Exchange(lock, dummy);
    while(dummy);

    critical section;

    lock = FALSE;

    remainder section;
}
```

Process Management: Synchronization

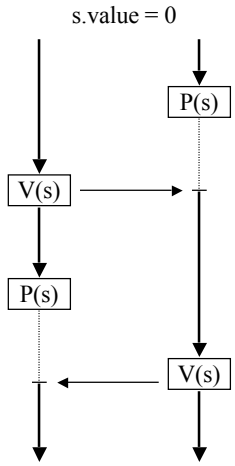
- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- More sophisticated synchronization mechanisms: Monitors, Message Passing
- Classical synchronization problems

Semaphores

- Problems with solutions above:
  - Although requirements simple (mutual exclusion), addition to programs complex.
  - Based on busy waiting.
- A Semaphore variable has two operations:
  - `V(Semaphore * s);`
    /* Increment value of **s** by 1 in a single indivisible action. If value is not positive, then a process blocked by a **P** is unblocked*/
  - `P(Semaphore * s);`
    /* Decrement value of **s** by 1. If the value becomes negative, the process invoking the **P** operation is blocked. */
- **Binary semaphore**: The value of `s` can be either 1 or 0 (TRUE or FALSE).
- **General semaphore**: The value of `s` can be any integer.

Effect of Semaphores

- Synchronization using semaphores:



- Mutual exclusion with semaphores:

```
BinSemaphore * s;
/* init to TRUE*/

while (TRUE) {
  P(s);
  critical section;
  V(s);
  remainder section;
}
```

Implementation (with busy waiting)

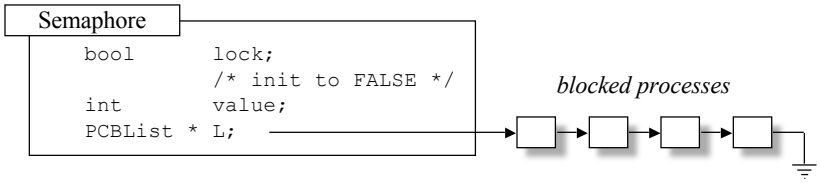
- Binary Semaphores:

```
P(BinSemaphore * s) {
  key = FALSE;
  do exchange(s.value, key);
  while (key == FALSE);
}

V(BinSemaphore * s) {
  s.value = TRUE;
}
```

- General Semaphores:

```
BinSemaphore * mutex /*TRUE*/
BinSemaphore * delay /*FALSE*/

P(Semaphore * s) {
 P(mutex);
 s.value = s.value - 1;
 if (s.value < 0)
   { V(mutex); P(delay); }
 else V(mutex);
}
V(Semaphore * s) {
 P(mutex);
 s.value = s.value + 1;
 if (s.value <= 0) V(delay);
 V(mutex);
}
```

Implementation ("without" busy waiting)




```
P(Semaphore * s) {
 while (TestAndSet(lock))
   no_op;
 s.value = s.value - 1;
 if (s.value < 0) {
   append(this_process, s.L);
   lock = FALSE;
   sleep();
 }
 lock = FALSE;
}
```

```
V(Semaphore * s) {
 while (TestAndSet(lock))
   no_op;
 s.value = s.value + 1;
 if (s.value <= 0) {
   PCB * p = remove(s.L);
   wakeup(p);
 }
 lock = FALSE;
}
```

Problems with Semaphores

- **Deadlocks**:
  - Process is blocked waiting for an event only it can generate.

```
s.value = 1
q.value = 1
```

P_1	P_2
P(s)	P(q)
P(q)	P(s)
...	...
V(s)	V(q)
V(q)	V(s)

Process Management: Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- More sophisticated synchronization mechanisms: Monitors, Message Passing
- Classical synchronization problems

Classical Problems: **Producer-Consumer**

```
Semaphore    * n;     /* initialized to 0    */
BinSemaphore * mutex; /* initialized to TRUE */
```

Producer:

```
while (TRUE) {

  produce item;

  P(mutex);

  deposit item;

  V(mutex);
  V(n);

}
```

Consumer:

```
while (TRUE) {

  P(n);
  P(mutex);

  remove item;

  V(mutex);

  consume item;

}
```

Classical Problems:
Producer-Consumer with Bounded Buffer

```
Semaphore     * full;  /* initialized to 0     */
Semaphore     * empty; /* initialized to n     */
BinSemaphore  * mutex; /* initialized to TRUE  */
```

Producer:

```
while (TRUE) {

  produce item;

  P(empty);
  P(mutex);

  deposit item;

  V(mutex);
  V(full);

}
```

Consumer:

```
while (TRUE) {

  P(full);
  P(mutex);

  remove item;

  V(mutex);
  V(empty);

  consume item;

}
```

Classical Problems:
Dining Philosophers

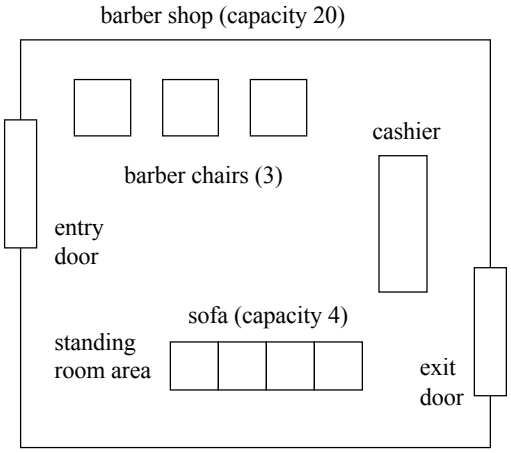
- 5 philosophers around a table, a plate in front of each philosopher, one chopstick between any two plates.
- When philosopher get hungry, he must grab both chopsticks in order to be able to eat.

```
Semaphore * chopstick[4]; /* initialize to 1 */
```

```
while (TRUE) {
  P(chopstick[i]);
  P(chopstick[(i+1) mod 5]);
  eat ...
  V(chopstick[i]);
  V(chopstick[(i+1) mod 5]);
  think ...
}
```

- Problem: deadlock

Classical Problems: The Barbershop




```
Semaphore * max_capacity;
/* init to 20 */
Semaphore * sofa;
/* init to 4 */
Semaphore * barber_chair;
/* init to 3 */
Semaphore * coord;
/* init to 3 */
Semaphore * cust_ready;
/* init to 0 */
Semaphore * leave_b_chair;
/* init to 0 */
Semaphore * payment;
/* init to 0 */
Semaphore * receipt;
/* init to 0 */
```

The Barbershop (cont)

Process *customer*:

```
P(max_capacity);
<enter shop>
P(sofa);
<sit on sofa>
P(barber_chair);
<get up from sofa>
V(sofa);
<sit in barber chair>
V(cust_ready);
P(finished);
<leave barber chair>
V(leave_b_chair);
<pay>
V(payment);
P(receipt);
<exit shop>
V(max_capacity);
```

Process *cashier*:

```
for(;;){
  P(payment);
  P(coord);
  <accept pay>
  V(coord);
  V(receipt);
 }
```

Process *barber*:

```
for(;;){
  P(cust_ready);
  P(coord);
  <cut hair>
  V(coord);
  V(finished);
  P(leave_b_chair);
  V(barber_chair);
}
```

The Fair Barbershop

Process *customer*:

```
P(max_capacity);
<enter shop>
P(mutex1);
custnr := ++count;
V(mutex1);
P(sofa);
<sit on sofa>
P(barber_chair);
<get up from sofa>
V(sofa);
<sit in barber chair>
P(mutex2);
enqueue(custnr);
V(cust_ready);
V(mutex2);
P(finished[custnr]);
<leave barber chair>
V(leave_b_chair);
<pay>
V(payment);
P(receipt);
<exit shop>
V(max_capacity);
```

Process *barber*:

```
for(;;){
  P(cust_ready);
  P(mutex2);
  dequeue(b_cust);
  V(mutex2);
  P(coord);
  <cut hair>
  V(coord);
  V(finished[b_cust]);
  P(leave_b_chair);
  V(barber_chair);
}
```

Process *cashier*:

```
for(;;){
  P(payment);
  P(coord);
  <accept pay>
  V(coord);
  V(receipt);
 }
```

Classical Problems: Readers/Writers

- Multiple readers can access data element concurrently.
- Writers access data element exclusively.

```
Semaphore * mutex, * wrt;  /* initialized to 1 */
int         nreaders;      /* initialized to 0 */
```

Reader:

```
P(mutex);
  nreaders = nreaders + 1;
  if (nreaders == 1) P(wrt);
V(mutex);

do the reading ....

P(mutex);
  nreaders = nreaders - 1;
  if (nreaders = 0) V(wrt);
V(mutex);
```

Writer:

```
P(wrt);

do the writing ...

V(wrt);
```

Incorrect Implementation of Readers/Writers

```
monitor ReaderWriter{
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;

    /* READERS */
    procedure startRead() {
      while (numberOfWriters != 0);
      numberOfReaders = numberOfReaders + 1;
    }
    procedure finishRead() {
      numberOfReaders = numberOfReaders - 1;
    }

    /* WRITERS */
    procedure startWrite() {
      numberOfWriters = numberOfWriters + 1;
      while (busy || (numberOfReaders > 0));
      busy = TRUE;
    };
    procedure finishWrite() {
      numberOfWriters = numberOfWriters - 1;
      busy = FALSE;
    };
  };
```

A Correct Implementation

```
monitor ReaderWriter{
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;
    condition okToRead, okToWrite;

    /* READERS */
    procedure startRead() {
      if (busy || (okToWrite.lqueue)) okToRead.wait;
      numberOfReaders = numberOfReaders + 1;
      okToRead.signal;
    }
    procedure finishRead() {
      numberOfReaders = numberOfReaders - 1;
      if (numberOfReaders = 0) okToWrite.signal;
    }

    /* WRITERS */
    procedure startWrite() {
      if (busy || (numberOfReaders > 0)) okToWrite.wait;
      busy = TRUE;
    };
    procedure finishWrite() {
      busy = FALSE;
      if (okToWrite.lqueue) okToWrite.signal;
      else                  okToRead.signal;
    };
  };
```

Process Management: Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- Classical synchronization problems
- More sophisticated synchronization mechanisms: Monitors, Message Passing

Higher-Level Synchronization Primitives

- Semaphores as the "GOTO" among the synchronization primitives.
  - very powerful, but tricky to use.
- Need higher-abstraction primitives, for example:
  - Monitors
  - `synchronized` primitive in JAVA
  - Protected Objects (Ada95)
  - Conditional Critical Regions
  - Message Passing

Monitors (Hoare / Brinch Hansen, 1973)

- Safe and effective sharing of abstract data types among several processes.
- Monitors can be modules, or objects.
  - local variable accessible only through monitor's procedures
  - process can entrer monitor only by invoking monitor procedure
- Only one process can be active in monitor.
- Additional synchronization through **conditions** (similar to semaphores)

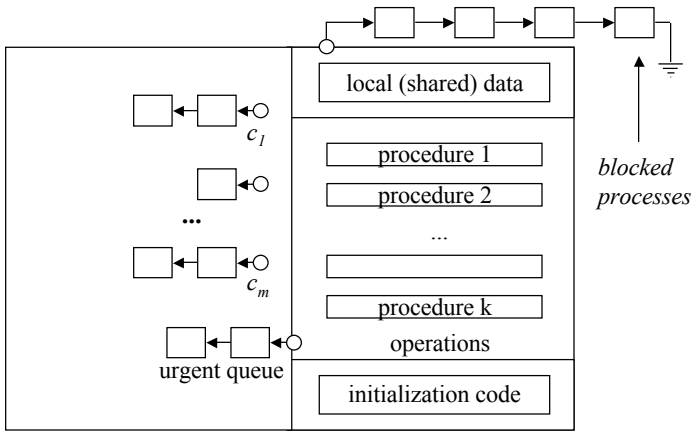
  `Condition c;`

  `c.cwait()` : suspend execution of calling process and enqueue it on condition `c`. The monitor now is available for other processes.

  `c.csignal()` : resume a process enqueued on `c`. If none is enqueued, do nothing.

  - `cwait/csignal` different from `P/V`: `cwait` always waits, `csignal` does nothing if nobody waits.

Structure of Monitor



Example: Binary Semaphore

```
monitor BinSemaphore {

  bool        locked; /* Initialize to FALSE */
  condition   idle;

  entry  void P() {
      if (locked) idle.cwait();
      locked = TRUE;
  }

  entry void V() {
      locked = FALSE;
      idle.csignal();
  }
}
```

Example: Bounded Buffer Producer/Consumer

```
monitor boundedbuffer {
  Item      buffer[N];    /* buffer has N items  */
  int       nextin;       /* init to 0           */
  int       nextout;      /* init to 0           */
  int       count;        /* init to 0           */
  condition notfull;      /* for synchronization */
  condition notempty;

void deposit(Item x) {
  if (count == N)
    notfull.cwait();
  buffer[nextin] = x;
  nextin = nextin + 1 mod
  N;
  count  = count + 1;
  notempty.csignal();
}
```

```
void remove(Item & x) {
  if (count == 0)
    notempty.cwait();
  x = buffer[nextout];
  nextout = nextout + 1 mod N;
  count  = count - 1;
  notfull.csignal();
}
```

Monitors: Issues, Problems

- What happens when the `x.csignal()` operation invoked by process P wakes up a suspended process Q?
  - Q waits until P leaves monitor?
  - P waits until Q leaves monitor?
  - `csignal()` *vs* `cnotify()`
- Nested monitor call problem.
- *Conditional wait* construct (better called *priority wait* construct):

```
x.cwait(c);   /* c is integer expression. */
```

- Caution when implementing schedule-sensitive code using monitors! (*e.g.* When moving resource-access control algorithms into monitors.) Resource scheduling may operate according to monitor scheduling algorithm, rather than the one that is being coded.

Synchronization in JAVA

- Critical sections:
  - **`synchronized`** statement
- Synchronized methods:
  - Only one thread can be in any synchronized method of an object at any given time.
  - Realized by having a single lock (also called monitor) per object.
- Synchronized static methods:
  - One lock per class.
- Synchronized blocks:
  - Finer granularity possible using synchronized blocks
  - Can use lock of any object to define critical section.
- Additional synchronization:
  - **`wait()`**, **`notify()`**, **`notifyAll()`**
  - Realized as methods for all objects

Java Synchronized Methods: vanilla Bounded Buffer Producer/Consumer

```
public class BoundedBuffer {
  Object[]  buffer;
  int       nextin
  int       nextout;
  int       size
  int       count;
```

```
public BoundedBuffer(int n) {
  size    = n;
  buffer  = new Object[size];
  nextin  = 0;
  nextout = 0;
  count   = 0;
}
```

```
synchronized public deposit(Object x){
  if (count == size) nextin.wait();
  buffer[nextin] = x;
  nextin = (nextin+1) mod N;
  count  = count + 1;
  nextout.notify();
}
```

```
synchronized public Object remove() {
  Object x;
  if (count == 0) nextout.wait();
  x = buffer[nextout];
  nextout = (nextout+1) mod N;
  count  = count - 1;
  nextin.notify();
  return x;
}
```

Example: Synchronized Block

(D. Flanagan, *JAVA in a Nutshell*)

```
public static void SortIntArray(int[] a) {
  // Sort array a.  This is synchronized so that
  // some other thread cannot change elements of
  // the array or traverse the array while we are
  // sorting it.
  // At least no other thread that protect their
  // accesses to the array with synchronized.

  // do some non-critical stuff here...

  synchronized (a) {
     // do the array sort here.
  }

  // do some other non-critical stuff here...

}
```

Message Passing

- The Primitives:

```
send(destination, message);
receive(source, message);
```

- Issues:
  - Synchronization (blocking vs non-blocking primitives)
  - Addressing (direct vs. indirect communication)
  - Reliability / Ordering (reliable vs. unreliable)

Message Passing: Synchronization

	blocking	non-blocking
send	Returns control to user only after message has been sent, or until acknowledgment has been received.	Returns control as soon as message queued or copied.
receive	Returns only after message has been received.	Signals willingness to receive message. Buffer is ready.
problems	•*Reduces concurrency.*	•*Need buffering:* •*still blocking* •*deadlocks!* •*Tricky to program.*

Message Passing: Synchronization (cont)

Combinations of primitives:

- Blocking send, blocking receive
  - rendezvous
- Nonblocking send, blocking receive
- Nonblocking send, nonblocking receive