Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- Classical synchronization problems

- Reading: R&R, Ch 14 (and Ch 13)

Synchronization: Critical Sections & Semaphores

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- Classical synchronization problems

The Critical Section Problem: Example 1

```
void echo() {                  char in;  /* shared variables */
  input(in, keyboard);         char out;
  out := in;
  output(out, display);
}
```

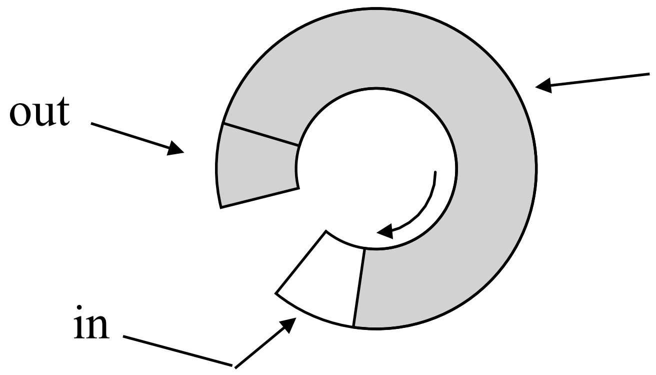
	Process 1	*Process 2*
Operation:	`Echo()`	`Echo()`

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!

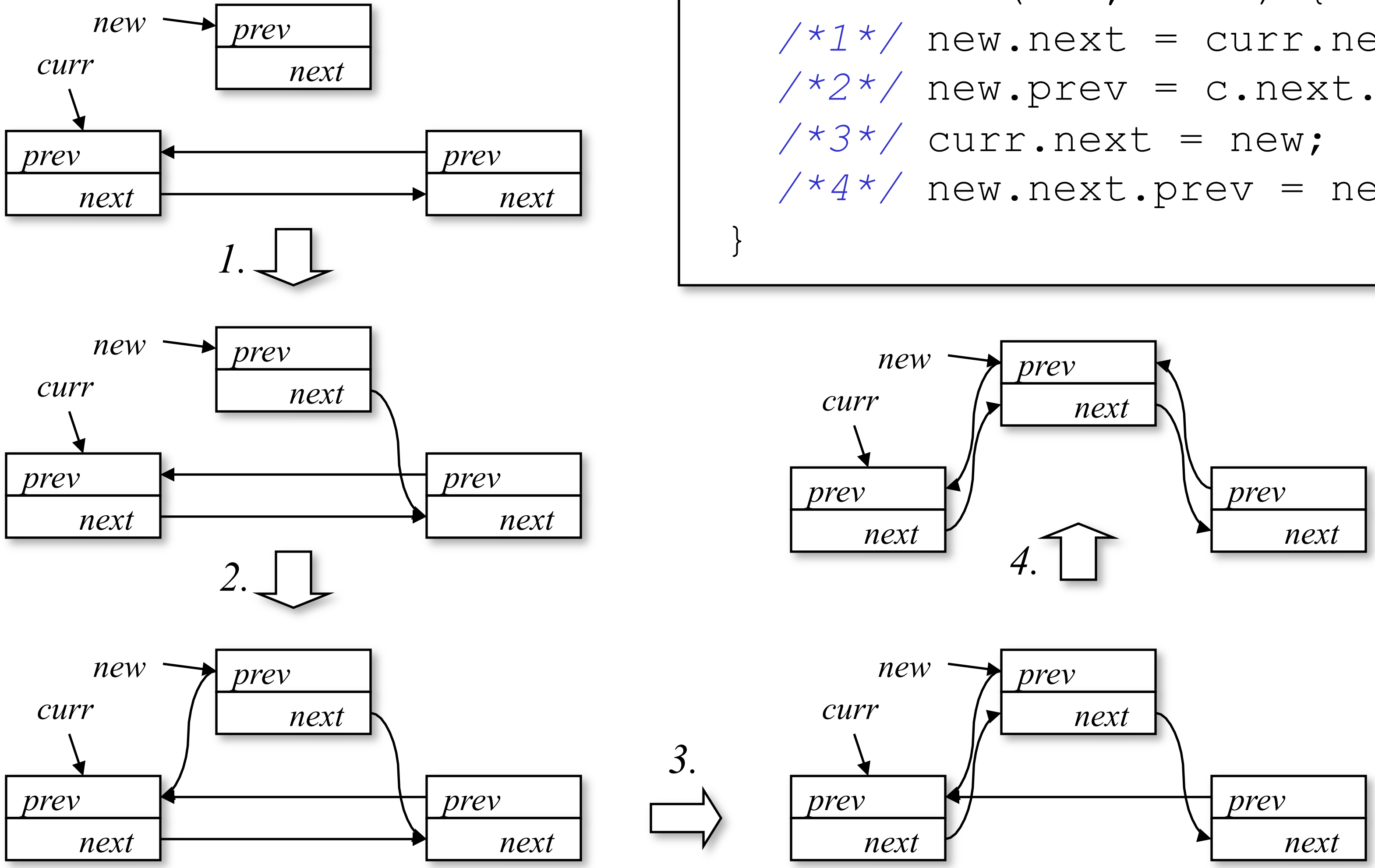
	Producer	**Consumer**
operation:	`counter = counter + 1`	`counter = counter - 1`
on CPU:	reg_1 = counter reg_1 = reg_1 + 1 counter = reg_1	reg_2 = counter reg_2 = reg_2 - 1 counter = reg_2
interleaved execution:	reg_1 = counter reg_1 = reg_1 + 1 counter = reg_1	 reg_2 = counter reg_2 = reg_2 - 1 counter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



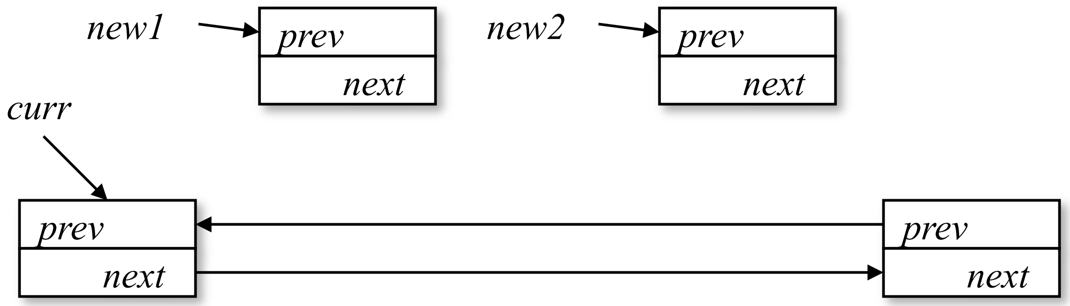
Interleaved Execution causes Errors!

Process 1

```
new1.next = curr.next;
new1.prev = c.next.prev;
…
…
…
…
curr.next = new1;
new1ext.prev = new1;
```

Process 2

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



- Must guarantee mutually exclusive access to list data structure!

Synchronization

- Why? Examples
- **What?** The Critical Section Problem
- How? Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- 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.

Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- Classical synchronization problems

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 : Pi 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 : P_i 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;

    }
```

Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- 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;
}
```

Compare-And-Swap

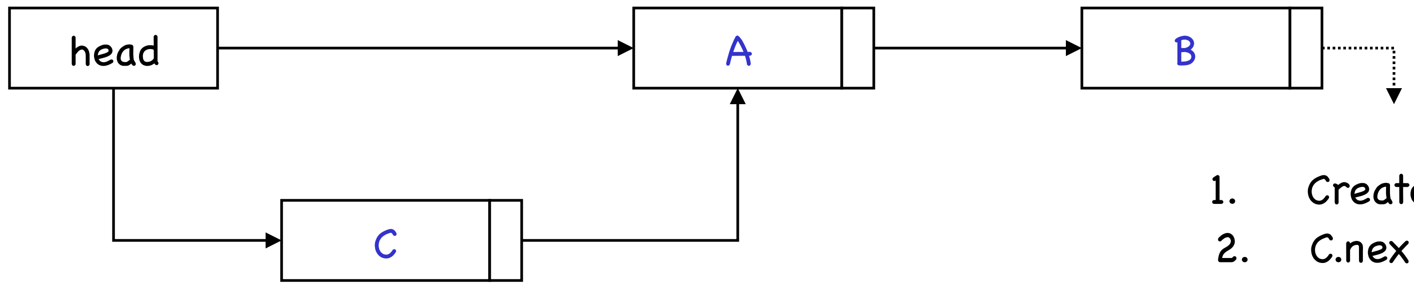
```
bool Compare&Swap (Type * x, Type old, Type new) {
      if *x == old {
            *x = new;
            return TRUE;
      }  else  {
            return FALSE
      }
}
```

atomic!

Some Fun with Compare-and-Swap: Lock-Free Concurrent Data Structures

Example: **Shared Stack**

PUSH element ***C*** onto stack:

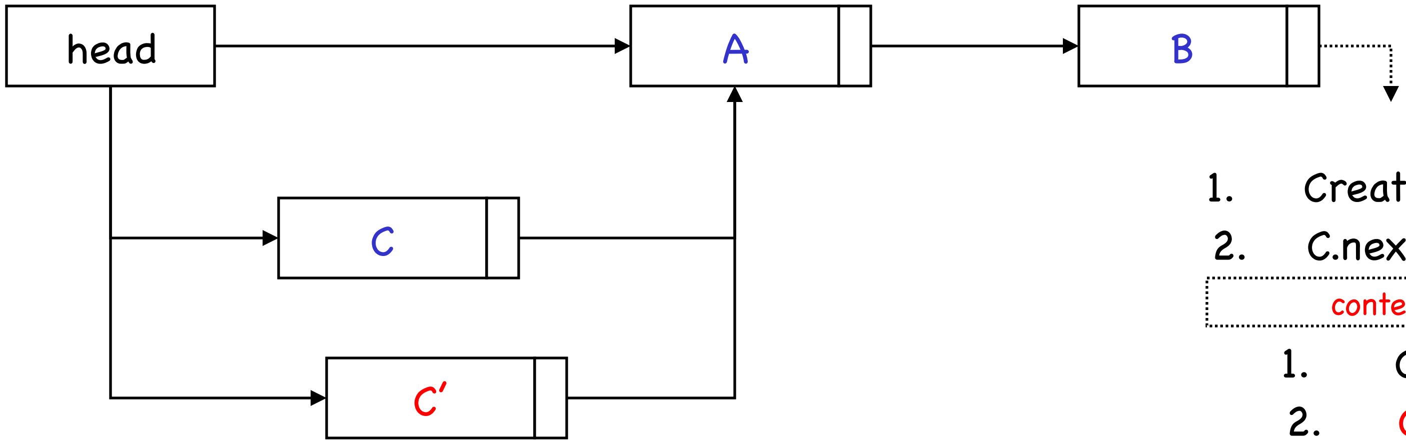


1. Create C
2. C.next = head
3. head = C

Some Fun with Compare-and-Swap: Lock-Free Concurrent Data Structures

Example: **Shared Stack**

PUSH element ***C*** onto stack: What can go wrong?!



1. Create C
2. C.next = head

context switch!

1. Create C′
2. C′.next = head
3. head = C′

context switch back!

3. head = C

Solution: compare-and-swap(head, C.next, C),
i.e. compare and swap head, new value C, and expected value C.next.
If fails, go back to step 2.

Simple Locking in POSIX: Mutex Locks

```
#include <pthread.h>

int pthread_mutex_init(      pthread_mutex_t     * restrict mutex,
                       const pthread_mutexattr_t * restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

`EAGAIN`: System lacks non-memory resources to initialize `*mutex`
`ENOMEM`: System lacks memory resources to initialize `*mutex`
`EPERM`: Caller does not have appropriate privileges

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

```
int pthread_mutex_lock   (pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

`EINVAL`: mutex configured with priority-ceiling on, and caller's priority is higher than mutex's current priority ceiling.
`EBUSY`: another thread holds the lock (returned to `mutex_trylock`)

Mutex Locks: Operations

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mylock);
      /* critical section */
pthread_mutex_unlock(&mylock);
```

- Use mutex locks to preserve critical sections or obtain exclusive access to resources.
- **Hold mutexes for short periods of time only!**
- “Short periods”?!
  - For example, changes to shared data structures.
- Use **Condition Variables** (see later) when waiting for events!

Uses for Mutex Locks: Unsafe Library Functions

Def: **Thread-safe** function: Exhibits no race conditions in multithreaded environment.

- Many library functions are not thread-safe!
- Can be made thread-safe with mutexes.

```
#include <pthread.h>
#include <stdlib.h>

int randsafe(int * result) {
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int error;

    if (error = pthread_mutex_lock(&lock))
        return error;
    *result = rand();
    return pthread_mutex_unlock(&lock);
}
```

Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- Classical synchronization problems

Semaphores

- Problems with solutions above:
  - Although requirements simple (mutual exclusion), addition to programs complex.
  - Based on busy waiting.

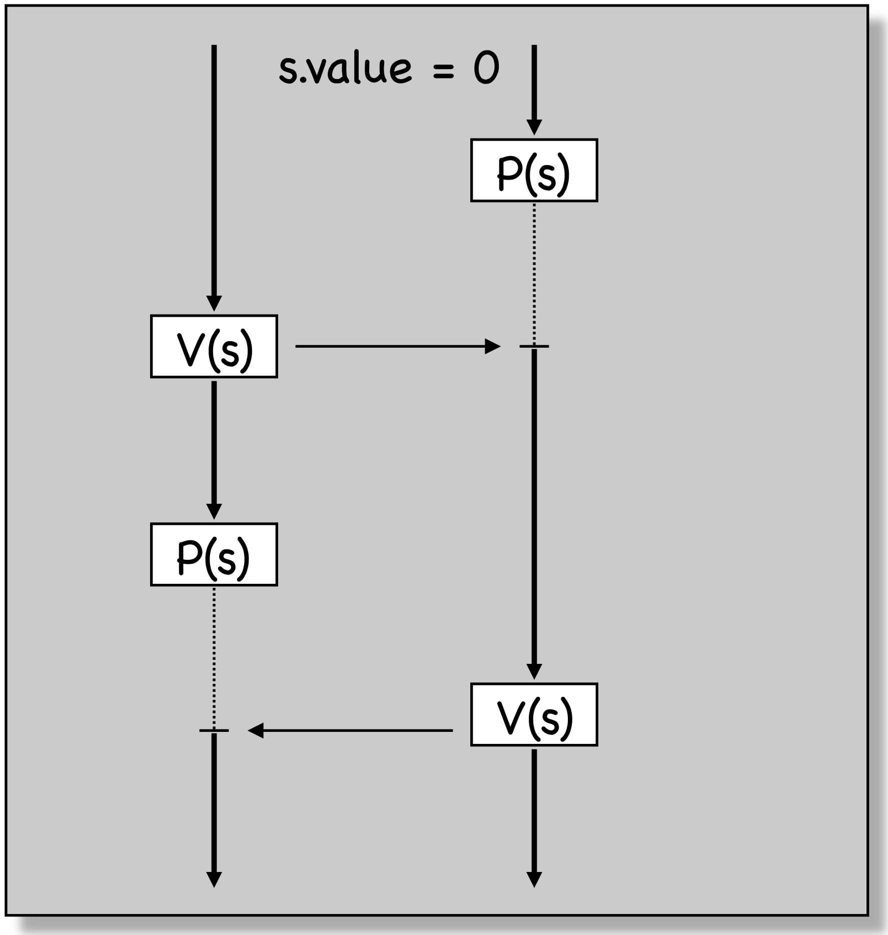
- A Semaphore variable has two operations:
  - **`V(Semaphore * s);`**

    /* 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. */

Effect of Semaphores

- Synchronization using semaphores:



- Mutual Exclusion with semaphores:

```
Semaphore s(1);
/* init to 1 */

while (TRUE) {
    s.P();

    critical section;

    s.V();

    remainder section;
}
```

Implementation (with busy waiting)

- Lock ("Mutex"):

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


Unlock(Mutex * s) {
   s.value = TRUE;
}
```

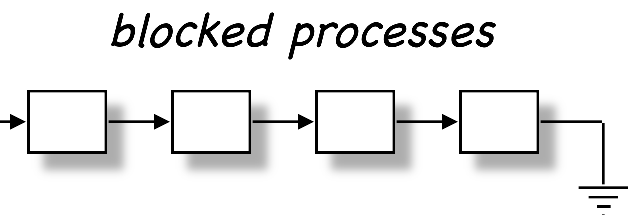
- General Semaphore:

```
Mutex * mutex /*TRUE*/
Mutex * delay /*FALSE*/

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

Implementation ("without" busy waiting)

Semaphore

```
bool        lock;
            /* init to FALSE */
int         value;
PCBList *   L;
```



```
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;
}
```

Semaphores POSIX Style?

- We will talk about this later…

Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- Classical synchronization problems

Classical Problems: **Producer-Consumer**

```
Semaphore n(0);          /* initialized to 0    */
Mutex     mutex(TRUE);   /* initialized to TRUE */
```

```
Producer:

while (TRUE) {

    produce item;

    mutex.P();

    deposit item;

    mutex.V();
    n.V();

}
```

```
Consumer:

while (TRUE) {

    n.P();
    mutex.P();

    remove item;

    mutex.V();

    consume item;

}
```

Classical Problems:
Producer-Consumer with Bounded Buffer

```
Semaphore full(0);  /* initialized to 0    */
Semaphore empty(n); /* initialized to n    */
Mutex     mutex;    /* initialized to TRUE */
```

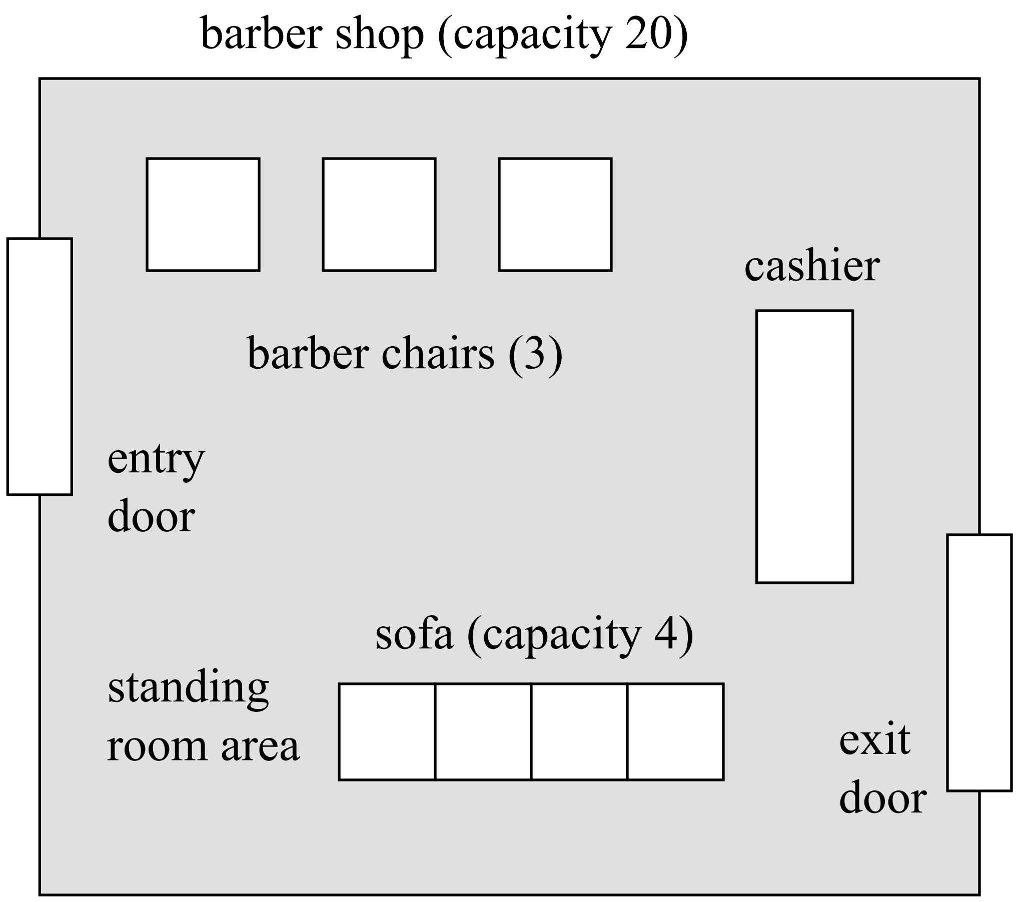
Producer:

```
while (TRUE) {

  produce item;

  empty.P();

  mutex.P();
  deposit item;
  mutex.V();

  full.V();

}
```

Consumer:

```
while (TRUE) {

  full.P();

  mutex.P();
  remove item;
  mutex.V();

  empty.V();

  consume item;

}
```

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 *barber*:

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

Process *cashier*:

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

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);
```

Reader/Writer Locks in POSIX: RWLocks

- R/W locks differentiate between exclusive (write) and shared (read) access.
- Reader *vs.* writer priority not specified in POSIX.

```
#include <pthread.h>

int pthread_rwlock_init(      pthread_rwlock_t     * rwlock,
                        const pthread_rwlockattr_t * attr);
```

`EAGAIN`: System lacks non-memory resources to initialize `*rwlock`
`ENOMEM`: Yada ... Yada ...

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_rdlock    (pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock    (pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock    (pthread_rwlock_t *rwlock);
```

R/W Lock Example: Vanilla Shared Container

```
/* shared variable */
static pthread_rwlock_t listlock;
static int lockiniterror = 0;
```

```
int init_container(void){
    return pthread_rwlock_init(&listlock, NULL)
}
```

```
/* add an item */
int add_data_r(data_t data, key_t key) {
    int error;
    if (error = pthread_rwlock_wrlock(&listlock)) {
        errno = error;
        return -1;
    }
    add_data(data, key);
    if (error = pthread_rwlock_unlock(&listlock)) {
        errno = error;
        error = -1;
    }
    return error;
}
```

R/W Lock Example: Vanilla Shared Container

```
/* shared variable */
static pthread_rwlock_t listlock;
static int lockiniterror = 0;
```

```
/* add an item */
int get_data_r(key_t key, data_t * datap) {
    int error;
    if (error = pthread_rwlock_rdlock(&listlock)) {
        errno = error;
        return -1;
    }
    get_data(key, datap);
    if (error = pthread_rwlock_unlock(&listlock)) {
        errno = error;
        error = -1;
    }
    return error;
}
```

Higher-Level Synchronization Primitives: Monitors

- 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
  - …

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 enter monitor only by invoking monitor procedure

- **Only one process can be active in monitor.**

- Additional synchronization through **conditions** (similar to semaphores)

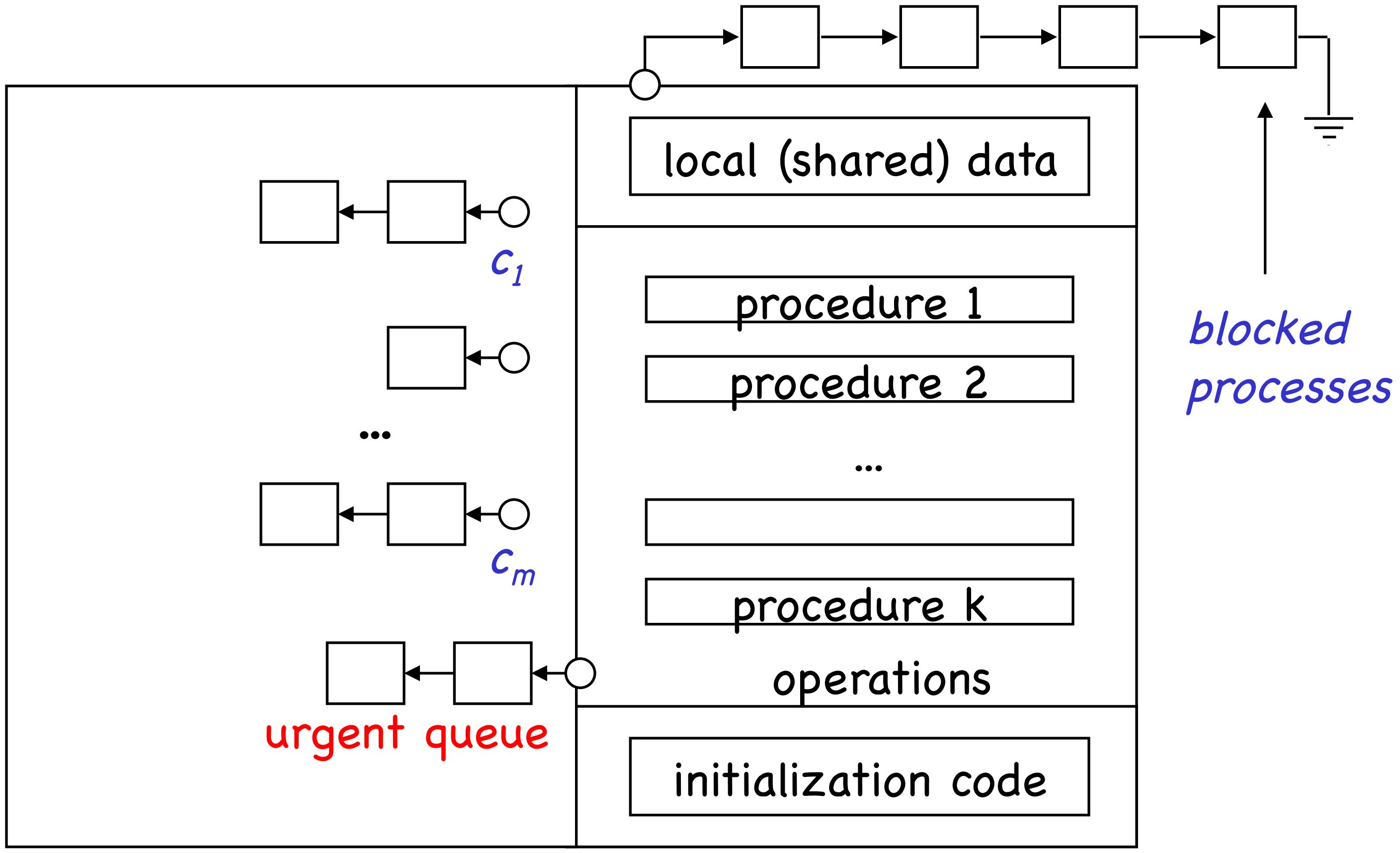
  `Condition c;`

  `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();
}
```

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;
    };
  };
```

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.

Monitors JAVA-Style: `synchronized`

- **Critical sections**:
  - `synchronized` statement
- **Synchronized methods:**
  - Only one thread can be in <u>any</u> 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;
```

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

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

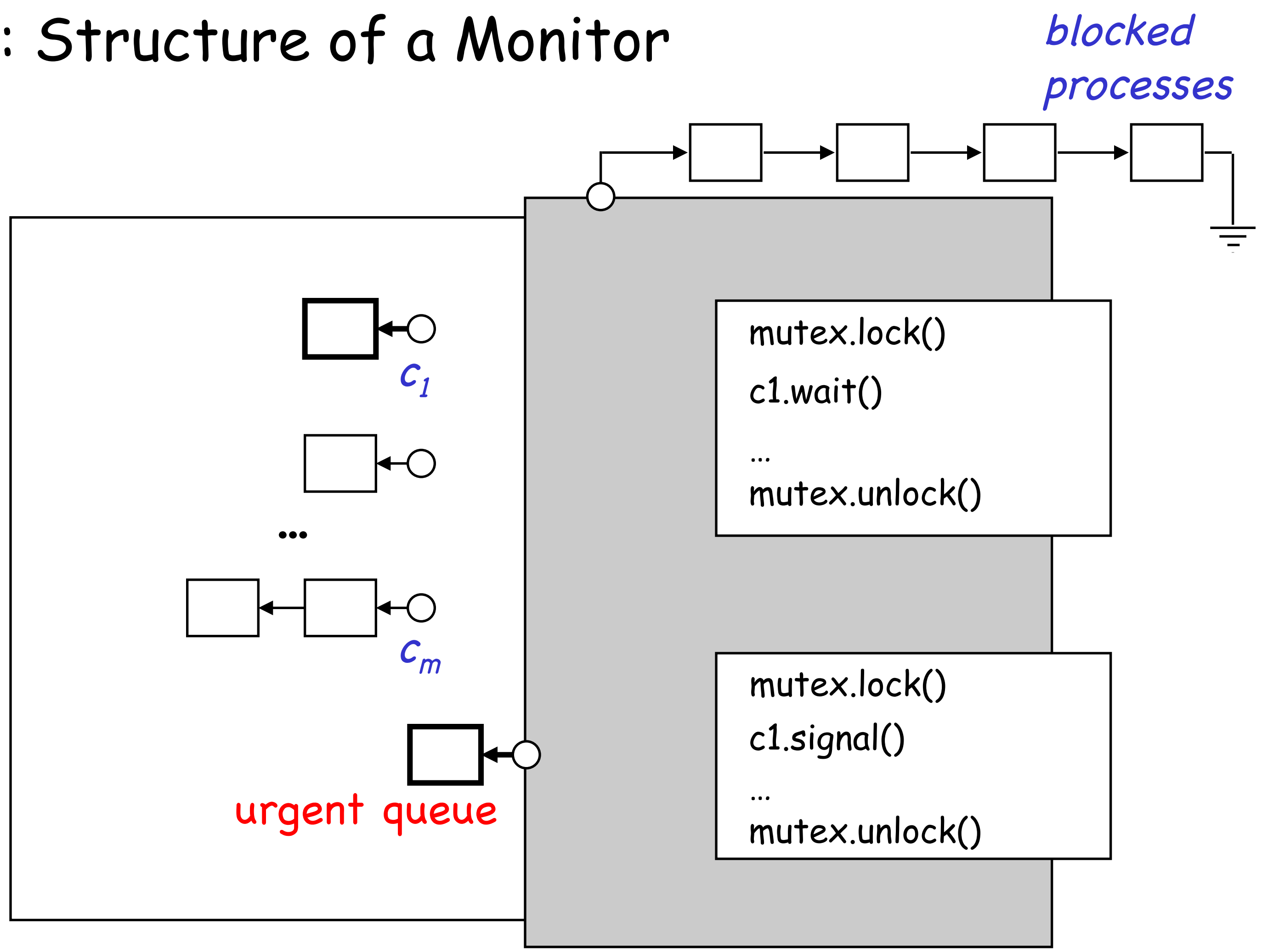
```
synchronized public Object remove() {
   Object x;
   if (count == 0) nextout.wait();
   x = buffer[nextout];
   nextout = (nextout+1) mod N;
   count  = count - 1;
   nextin.notify();
   return x;
}
```

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 protects 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...

}
```

Monitors POSIX-Style: Condition Variables

Recall: Structure of a Monitor



POSIX Condition Variables

#include <pthread.h>

```
int pthread_cond_wait(pthread_cond_t  * cond,
                      pthread_mutex_t * mutex);
```

The `pthread_cond_wait()` function atomically unlocks the `mutex` argument and waits on the `cond` argument. Before returning control to the calling function, `pthread_cond_wait()` re-acquires the `mutex`.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

The `pthread_cond_signal()` function unblocks one thread waiting for the condition variable `cond`.

POSIX Condition Variable: Scenario

waiting for a particular condition (e.g. x==y)

```
while (x != y);
```

correct strategy to wait for condition

while (x != y):
1. lock a mutex
2. test the condition (x==y)
3. if TRUE, unlock mutex and exit loop
4. if FALSE, **suspend thread** and unlock mutex (?!!!)

```
pthread_mutex_lock(&m);
while (x != y)
    pthread_cond_wait(&v, &m);
/* now we are in the "critical section" */
pthread_mutex_unlock(&m);
```

```
pthread_mutex_lock(&m);
x++;
pthread_cond_signal(&v);
pthread_mutex_unlock(&m);
```

- When `cond_wait` returns, thread owns mutex and can test condition again.
- Call `cond_wait` only if you own mutex!

Example: Thread-Safe Barrier Locks

```
/* shared variables */
static pthread_cond_t  bcond  = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t bmutex = PTHREAD_MUTEX_INITIALIZER;
static int count = 0; /* how many threads are waiting? */
static int limit = 0;
```

```
/* initialize the barrier */
int initbarrier(int n) {
    int error;
    if (error = pthread_mutex_lock(&bmutex))
        return error;
    if (limit != 0) { /* don't initialize barrier twice! */
        pthread_mutex_unlock(&bmutex);
        return EINVAL;
    }
    limit = n;
    return pthread_mutex_unlock(&bmutex);
}
```

Example: Thread-Safe Barrier Locks

```
/* shared variables */
static pthread_co
static pthread_mu
static int count
static int limit
```

```
/* wait at barrier until all n threads arrive */
int waitbarrier(void) {
    int berror = 0;
    int error;
    if (error = pthread_mutex_lock(&bmutex))
        return error;
    if (limit <= 0) { /* barrier not initialized?! */
        pthread_mutex_unlock(&bmutex);
        return EINVAL;
    }
    count++;
    while ((count < limit) && !berror)
        berror = pthread_cont_wait(&bcond, &bmutex);
    if (!berror) /* wake up everybody */
        berror = pthread_cond_broadcast(&bcond);
    error = pthread_mutex_unlock(&bmutex);
    if (berror)
      return berror;
    return error;
}
```

Timed Wait on Condition Variables

```
#include <pthread.h>

int pthread_cond_timedwait(       pthread_cond_t   * cond,
                                  pthread_mutex_t  * mutex,
                            const struct timespec * abstime);
```