## Synchronization

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- Classical synchronization problems
- Monitors
- Reading: Stevens, Ch 12

## 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
- Monitors

## The Critical Section Problem: Example 1

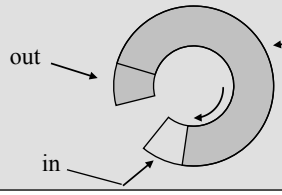
```
void echo() {                    char in;  /* shared variables */
  input(in, keyboard);           char out;
  out := in;
  output(out, display);
}
```

| | Thread 1 | Thread 2 |
|---|---|---|
| Operation | **echo()** | **echo()** |
| Interleaved execution | …<br>**input**(in, keyboard);<br>out = in; | …<br>…<br>… |
| | …<br>…<br>… | **input**(in, keyboard);<br>out = in;<br>**output**(out, display); |
| | **output**(out, display); | … |

**Race condition !**

## The Critical Section Problem: Example 2

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




circular buffer of size *n*

```
Item*  buffer[n];
int    in, out;
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;
}
```

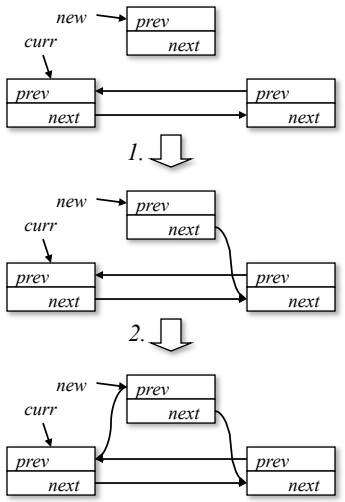
# This Implementation is not Correct!

| | Producer | Consumer |
|---|---|---|
| Operation | `counter = counter+1;` | `counter = counter-1;` |
| on CPU | $reg_1$ = counter;<br>$reg_1$ = $reg_1$ + 1;<br>counter = $reg_1$; | $reg_2$ = counter;<br>$reg_2$ = $reg_2$ - 1;<br>counter = $reg_2$; |
| Interleaved execution | $reg_1$ = counter;<br>$reg_1$ = $reg_1$ + 1; | |
| | | $reg_2$ = counter;<br>$reg_2$ = $reg_2$ - 1; |
| | counter = $reg_1$; | |
| | | counter = $reg_2$; |

**Race condition !**

# Critical Section Problem: Example 3

Insertion of an element into a list.




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

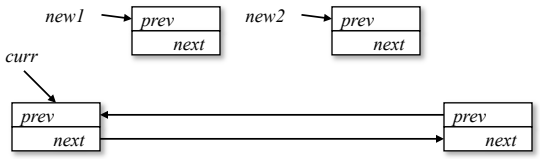
## Interleaved Execution causes Errors!

Thread 1

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

Thread 2

```
…
…
new2.next = curr.next;
new2.prev = curr;
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
- Monitors

## Critical Sections

- Execution of **critical section** by threads 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 thread at a time can enter the critical section.
2. A thread that halts in non-critical section cannot prevent other processes from entering the critical section.
3. A thread requesting to enter a critical section should not be delayed indefinitely.
4. When no thread is in a critical section, any thread 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 thread 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
- Monitors

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

- Two threads $T_0$ and $T_1$
- `int turn;` /* `turn == i` : $T_i$ is allowed to enter c.s. */

```
Ti: while (TRUE) {
      while (turn != i) no_op;
      critical section;
      turn = j;
      remainder section;
    }
```

## Another Wrong Solution (check & set)

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

```
Ti: while (TRUE) {

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

      critical section;

      flag[i] = FALSE;

      remainder section;

    }
```

## Yet Another Wrong Solution (set & check)

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

```
Ti: 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 */
```

```
T_i: 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
- Monitors

## Hardware Support For Synchronization

- Approach 1: **Disallow interrupts**
  - simplicity
  - widely used
  - problem: interrupt service latency
  - problem: what about multiprocessors?

- Better Approach: **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

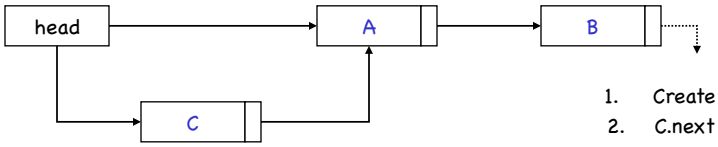
```
bool Compare&Swap (Type * x, Type old, Type new) {
    if *x == old {
        *x = new;
        return TRUE;
    } else {
        return FALSE
    }
}
```

*atomic!*

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

Example: **Shared Stack**

PUSH element *C* onto stack:

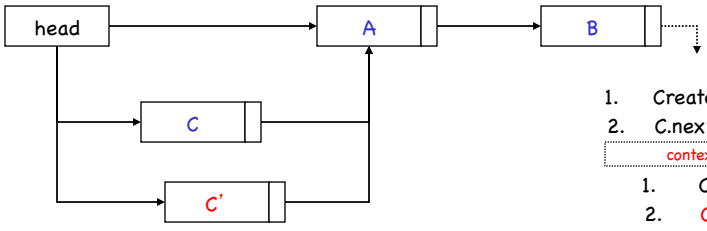



1. Create C
2. C.next = head
3. head = C

## 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      * mutex,
                       const pthread_mutexattr_t * 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

- Use mutex locks to: preserve critical sections or obtain exclusive access to resources.

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

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

- **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>

static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int randsafe(int * result) {
    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
- Monitors

# Semaphores

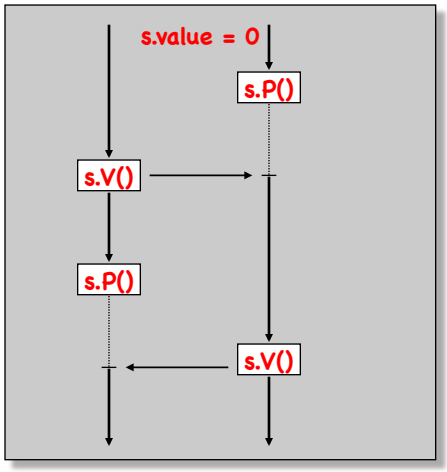
A Semaphore variable has two operations:

```
Semaphore::P();
    /* Decrement value of semaphore by 1. If the value becomes
       negative, the thread invoking the P operation is blocked. */
```

```
Semaphore::V();
    /* Increment value of semaphore by 1. If value is not
       positive, then a thread blocked by a P() is unblocked*/
```

# 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)

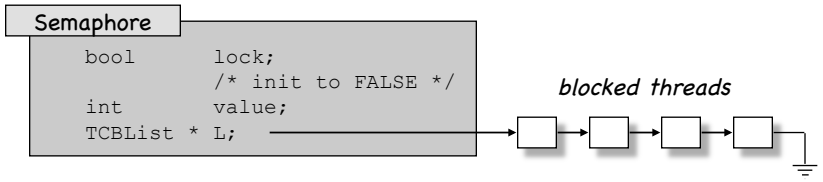
- Lock ("Mutex") Semaphore:

```
class Mutex {

bool lock = FALSE; // shared

Lock() {
  dummy = TRUE; // on stack
  do exchange(lock, dummy);
  while (dummy == TRUE);
}

Unlock() {
  lock = FALSE;
}

};
```

- General Semaphore:

```
class Semaphore {
Mutex mutex /*FALSE*/
Mutex delay /*TRUE*/
int value = 0; // set in init
P() {
 mutex.Lock();
 value--;
 if (value < 0)
   { mutex.Unlock(); delay.Lock(); }
 else mutex.Unlock();
}
V() {
 mutex.Lock();
 value++;
 if (value <= 0) delay.Unlock();
 mutex.Unlock();
}  };
```

## Implementation ("without" busy waiting)




```
P() {
 while (TestAndSet(lock))
   no_op;
 value--;
 if (value < 0) {
   append(this_thread, s->L);
   lock = FALSE;
   sleep();
 } else {
   lock = FALSE;
 }
}
```

```
V() {
 while (TestAndSet(lock))
   no_op;
 value++;
 if (value <= 0) {
   PCB * p = remove(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
- Monitors

## Classical Problems: Producer-Consumer

```
Semaphore n(0);          /* initialized to 0    */
Semaphore mutex(1);      /* initialized to 1 */
```

***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    */
Semaphore mutex(1); /* initialized to 1    */
```

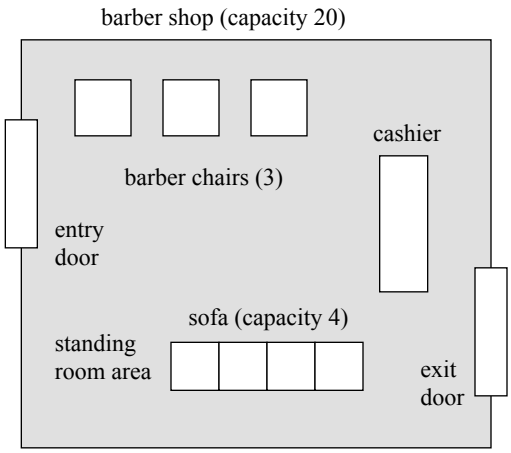
***Producer:***

```
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 * finished;
/* 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);
```

## 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;
}
```

## In-Class Practice: Dining Philosopher's Problem

(wikipedia)

```
for(ever) {
        <think>
        <eat>
}
```

## In-Class Practice: Dining Savages Problem

- A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he/she helps him/herself from the pot, unless the pot is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

- Any number of savage threads run the following code:

```
for(;;) { /* loop forever */
    getServingFromPot();
    eat();
}
```

- One cook thread runs this code:

```
for(;;) { /* forever cooking and serving */
    putServingsInPot(M);
}
```

## 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
- Monitors

## 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)

Local variables accessible only through monitor's procedures.

Threads can enter monitor only by invoking monitor procedures.

**Only one thread can be active in monitor at any given time.**

- Safe and effective sharing of data structures among several threads.
- Monitors can be modules, or classes, or objects.

## Monitors (Hoare / Brinch Hansen, 1973)

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

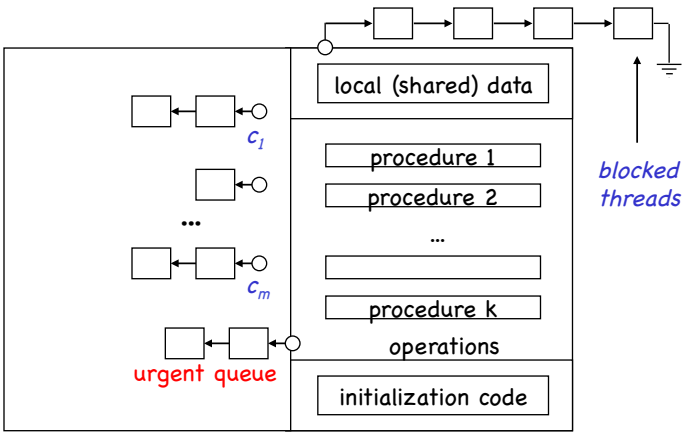
  `Condition c;`

  `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: Mutex Lock

```
monitor MutexLock{

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

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

  entry void Unlock() {
      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 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;
}
```

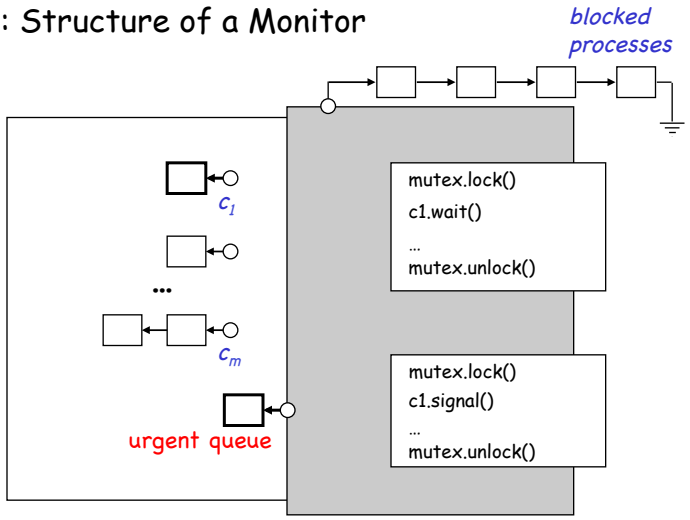
## Example: Synchronized Block

(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 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

```
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`.

# 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) {

    pthread_mutex_lock(&bmutex))

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

    pthread_mutex_unlock(&bmutex);
    return 0;
}
```

This code does no error checking!

# 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) {

    pthread_mutex_lock(&bmutex))

    if (limit <= 0) { /* barrier not initialized?! */
        pthread_mutex_unlock(&bmutex);
        return EINVAL;
    }
    count++;
    while (count < limit)
        pthread_cond_wait(&bcond, &bmutex);
    /* wake up everybody */
    pthread_cond_broadcast(&bcond);

    pthread_mutex_unlock(&bmutex);

    return 0;
}
```

This code does no error checking!

## 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_cond_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);
```

```
Example – wait for 5 sec:

pthread_mutex_lock(&mutex);
clock_gettime(CLOCK_REALTIME, &ts);
ts.tv_sec += 5;

rc = pthread_cond_timedwait(&cond, &mutex, &ts);
if (rc == ETIMEDOUT)
    printf("wait timed out!");
```