


Interprocess Communication

Chapter 2: Sections 2.3

IPC Issues

1. How can one process pass information to another 
2. Keep two processes from getting in each other's way
3. Proper sequencing when dependencies are present.

Race Conditions

- » Processes working together may share common storage.
- » A race condition is where two or more processes are reading or writing shared data and the final result depends on who runs precisely when.
- » Very common as multiprocessing with more and more cores

Race Condition Example

```

#include <assert.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

static void * simple_thread(void *) ;

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER ;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER ;
int x = 5 ;
int y = 0 ;

int main()
{
    pthread_t tid = 0 ;
    int count = 0 ;

    srand (time(NULL)) ;

    int retval = pthread_create(&tid, 0, &simple_thread, 0) ;

    while ( 1 )
    {
        x = 5 ;
        if ( x == 5 )
        {
            y = x * 2 ;
            printf("y = %d\n", y ) ;
            fflush( NULL ) ;

            assert( y == 10 ) ;
        }
        usleep( rand() % 100 ) ;
    }

    return 0 ;
}

static void * simple_thread(void * unused)
{
    while ( 1 )
    {
        x = 100 ;
        usleep( rand() % 100 ) ;
    }

    return NULL ;
}

```

```

#include <assert.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

static void * simple_thread(void *) ;

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER ;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER ;
int x = 5 ;
int y = 0 ;

int main()
{
    pthread_t tid = 0 ;
    int count = 0 ;

    srand (time(NULL)) ;

    int retval = pthread_create(&tid, 0, &simple_thread, 0) ;

    while ( 1 )
    {
        x = 5 ;
        if ( x == 5 )
        {
            y = x * 2 ;
            printf("y = %d\n", y ) ;
            fflush( NULL ) ;

            assert( y == 10 ) ;
        }
        usleep( rand() % 100 ) ;
    }

    return 0 ;
}

static void * simple_thread(void * unused)
{
    while ( 1 )
    {
        x = 100 ;
        usleep( rand() % 100 ) ;
    }

    return NULL ;
}

```

Shared data



Both threads modify/compare x
with no protection against
concurrent access



Real World Example

```

/*****
* read() service function
*****/
static ssize_t driver_read ( struct file *filePtr, char *buf, size_t count, loff_t *off )
{
    unsigned int b = ( unsigned int )( long ) filePtr->private_data;

    // Wait until we have data
    data_ready = FALSE;

    if ( ( wait_event_interruptible_timeout ( _wait, data_ready, HZ ) == 0 ) )
    {
        return -EIO;
    }

    statistics[b].read_count++;

    if( count == sizeof( int ) )
    {
        if ( copy_to_user ( buf, &fb[b].full_buffer, 4 ) !=0 )
        {
            return -EFAULT;
        }
    }
    else if ( count == fb_size )
    {
        if ( copy_to_user ( buf, fb[b].fb_addr[fb[b].full_buffer], count ) !=0 )
        {
            return -EFAULT;
        }
    }
    else
    {
        return -EINVAL;
    }

    return count;
}

```

While this copy is occurring other threads can be signaled to copy into the buffer

Critical Regions

- » Key is to prevent more than one process from reading and writing from a shared area at the same time
 - » Mutual exclusion
 - » pthread mutexes

Mutex Variables

```
pthread_mutex_lock (mutex)  
pthread_mutex_trylock (mutex)  
pthread_mutex_unlock (mutex)
```

- The `pthread_mutex_lock()` routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- `pthread_mutex_trylock()` will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
- `pthread_mutex_unlock()` will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An error will be returned if:
 - If the mutex was already unlocked
 - If the mutex is owned by another thread

```
#include <assert.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
static void * simple_thread(void *) ;
```

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER ;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER ;
int x = 5 ;
int y = 0 ;
```

```
int main()
{
    pthread_t tid = 0 ;
    int count = 0 ;

    srand (time(NULL)) ;

    int retval = pthread_create(&tid, 0, &simple_thread, 0) ;
```

```
while ( 1 )
{
```

```
    int retval = pthread_mutex_lock(&mutex1) ; // lock mutex 1
    if( retval )
    {
        perror("Acquire mutex 1: ");
    }
```

```
    x = 5 ;
    if ( x == 5 )
    {
        y = x * 2 ;
        printf("y = %d\n", y ) ;
        fflush( NULL ) ;
```

```
        assert( y == 10 ) ;
```

```
    }

    retval = pthread_mutex_unlock(&mutex1) ; // unlock mutex 1
    if( retval )
    {
        perror("Release mutex 1: ");
    }
```

```
    usleep( rand() % 100 ) ;
```

```
}
```

```
return 0 ;
```

```
}
```

pthread_mutex_lock()

```
static void * simple_thread(void * unused)
{
    while ( 1 )
```

```
    {
        int retval = pthread_mutex_lock(&mutex1) ; // lock mutex 1
        if( retval )
        {
            perror("Acquire mutex 1: ");
        }
```

```
        x = 100 ;
```

```
        retval = pthread_mutex_unlock(&mutex1) ; // unlock mutex 1
        if( retval )
        {
            perror("Release mutex 1: ");
        }
        usleep( rand() % 100 ) ;
    }
```

```
return NULL ;
```

```
}
```

Critical Regions

- » Critical region is the part of a program where shared memory is accessed.
- » Four conditions to avoid race conditions
 1. No two processes may be simultaneously inside the critical region
 2. No assumptions may be made about speeds or number of CPUs
 3. No process running outside its critical region may block any process
 4. No process should have to wait forever to enter its critical

Critical Regions (2)

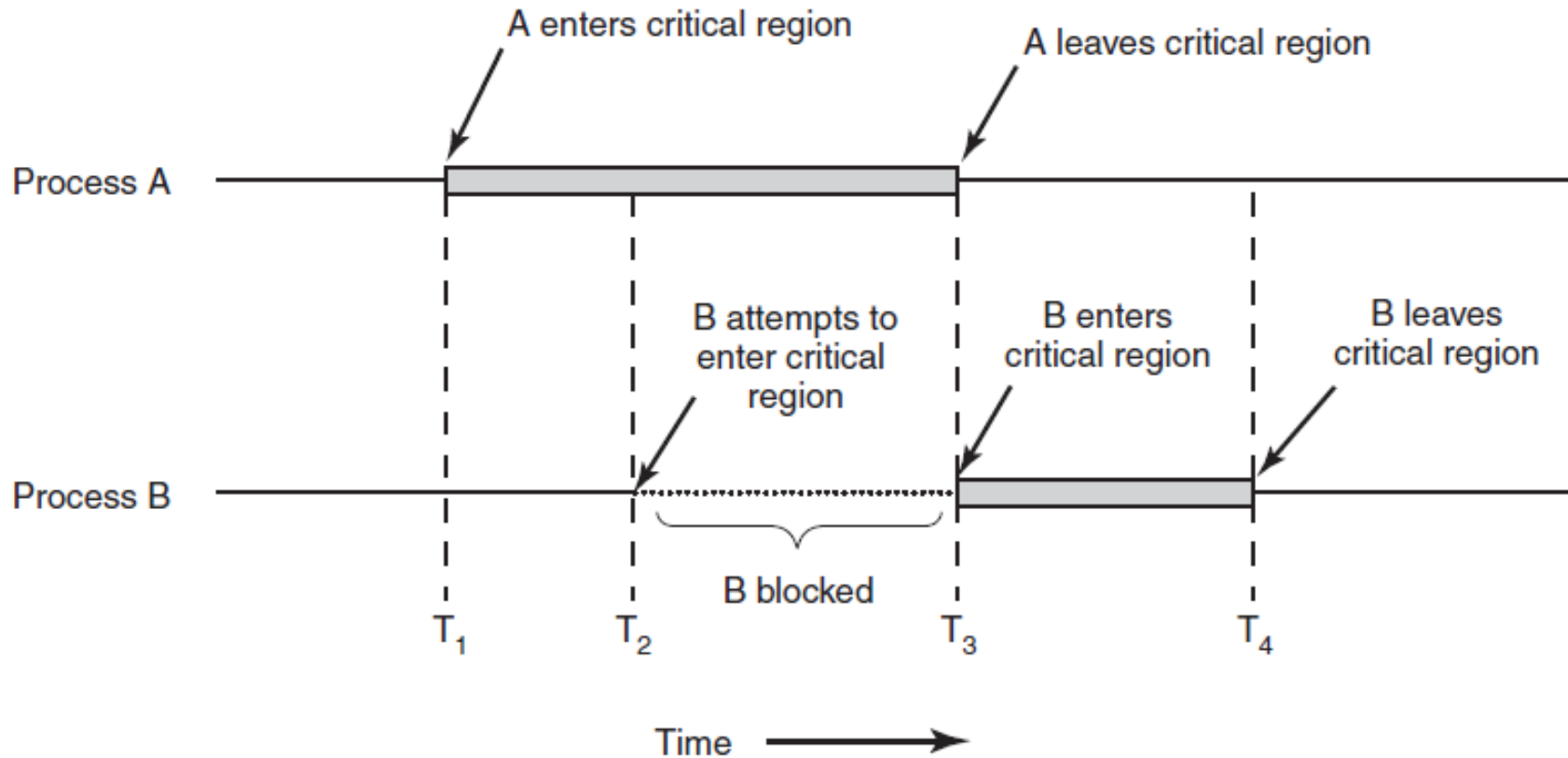


Figure 2-22. Mutual exclusion using critical regions.

Mutual exclusion: Busy waiting

- » Disabling Interrupts
 - » Only works on single processor system
 - » Disable interrupts before entering critical region.
 - » Re-enable before leaving the critical region.
 - » Generally a bad idea
 - » Kernel does it, though

Mutual exclusion: Busy waiting

- » Lock variables

- » Shared lock variable, initially 0

- » When a process wants to enter the critical region, test then set

- » Bad. Value can change after check but before set.

- » See my real world example

Mutual exclusion: Busy waiting

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

» Strict alternation

» Ok for C. Bad for garbage collected languages.

» Violates criteria #3

Mutual exclusion: Busy waiting

- » Busy waiting wastes CPU time
 - » Only should be used when there is a reasonable expectation that that wait will be short.
 - » spin-lock is a lock that uses busy waiting

Fix Attempt #1

```

/*****
* read() service function
*****/
static ssize_t driver_read ( struct file *filePtr, char *buf, size_t count, loff_t *off )
{
    unsigned int b = ( unsigned int )( long ) filePtr->private_data;

    // Wait until we have data
    data_ready = FALSE;

    if ( ( wait_event_interruptible_timeout ( _wait1, data_ready, HZ ) == 0 ) )
    {
        return -EIO;
    }

    if( atomic_read(&lock) == 0 ) ← Test if lock is unlocked
    {
        atomic_set( &lock, 1 ); ← If unlocked then lock it to
                                guard the critical section

        statistics[b].read_count++;

        if( count == sizeof( int ) )
        {
            if ( copy_to_user ( buf, &fb[b].full_buffer, 4 ) !=0 )
            {
                atomic_set( &lock, 0 );
                return -EFAULT;
            }
        }
        else if ( count == fb_size )
        {
            if ( copy_to_user ( buf, fb[b].fb_addr[fb[b].full_buffer], count ) !=0 )
            {
                atomic_set( &lock, 0 );
                return -EFAULT;
            }
        }
        else
        {
            atomic_set( &lock, 0 );
            return -EINVAL;
        }
    }

    atomic_set( &lock, 0 ); ← All done so unlock

    return count;
}

```

Critical Section

```

/*****
* read() service function
*****/
static ssize_t driver_read ( struct file *filePtr, char *buf, size_t count, loff_t *off )
{
    unsigned int b = ( unsigned int )( long ) filePtr->private_data;

    // Wait until we have data
    data_ready = FALSE;

    if ( ( wait_event_interruptible_timeout ( _wait1, data_ready, HZ ) == 0 ) )
    {
        return -EIO;
    }

    if( atomic_read(&lock) == 0 ) ←
    {
        atomic_set( &lock, 1 ); ←
        statistics[b].read_count++;

        if( count == sizeof( int ) )
        {
            if ( copy_to_user ( buf, &fb[b].full_buffer, 4 ) !=0 )
            {
                atomic_set( &lock, 0 );
                return -EFAULT;
            }
        }
        else if ( count == fb_size )
        {
            if ( copy_to_user ( buf, fb[b].fb_addr[fb[b].full_buffer], count ) !=0 )
            {
                atomic_set( &lock, 0 );
                return -EFAULT;
            }
        }
        else
        {
            atomic_set( &lock, 0 );
            return -EINVAL;
        }
    }

    atomic_set( &lock, 0 );

    return count;
}

```

That's a potential race condition
if an interrupt modifies lock in between
the read and set

Critical Section

TSL Instructions

- » Test and Set Lock
 - » Atomic instruction
 - » An instruction used to write 1 (set) to a memory location and return its old value as a single **atomic** (i.e., non-interruptible) operation.
 - » If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process's test-and-set is finished.

Fix Attempt #2

```

/*****
* read() service function
*****/
static ssize_t sppDriverRead ( struct file *filePtr, char *buf, size_t count, loff_t *off )
{
    unsigned int b = ( unsigned int ) ( long ) filePtr->private_data;

    // Wait until we have data
    data_ready = FALSE;

    if ( ( wait_event_interruptible_timeout ( spp_wait_spp1, data_ready, HZ ) == 0 ) )
    {
        return -EIO;
    }

    if( test_and_set_bit(LOCK, &lock) == 0 )
    {
        statistics[b].read_count++;

        if( count == sizeof( int ) )
        {
            if ( copy_to_user ( buf, &fb[b].full_buffer, 4 ) !=0 )
            {
                clear_bit(LOCK, &lock);
                return -EFAULT;
            }
        }
        else if ( count == fb_size )
        {
            if ( copy_to_user ( buf, fb[b].fb_addr[fb[b].full_buffer], count ) !=0 )
            {
                clear_bit(LOCK, &lock);
                return -EFAULT;
            }
        }
        else
        {
            clear_bit(LOCK, &lock);
            return -EINVAL;
        }
        clear_bit(LOCK, &lock);
    }

    return count;
}

```

This operation is atomic and cannot be reordered. Test and set the bit in one operation.

Fixed

Peterson's Solution

```
bool flag[2] = {false, false};  
int turn;
```

```
P0:    flag[0] = true;  
P0_gate: turn = 1;  
    while (flag[1] && turn == 1)  
    {  
        // busy wait  
    }  
    // critical section  
    ...  
    // end of critical section  
    flag[0] = false;
```

```
P1:    flag[1] = true;  
P1_gate: turn = 0;  
    while (flag[0] && turn == 0)  
    {  
        // busy wait  
    }  
    // critical section  
    ...  
    // end of critical section  
    flag[1] = false;
```

» 1981, G. L. Peterson came up with a mutual exclusion algorithm

Sleep and Wakeup

- » Peterson's and TSL are correct but. have the defect of busy waiting.
 - » Wastes CPU time
 - » Also can have cause priority inversion and starvation

Priority Inversion and Starvation

- Indefinite blocking or starvation
 - process is not deadlocked
 - but is never removed from the semaphore queue
- Priority inversion
 - lower-priority process holds a lock needed by higher-priority process !
- Assume three processes L, M, and H
 - Priorities in the order $L < M < H$
 - L holds shared resource R, needed by H
 - M preempts L, H needs to wait for both L and M !!

Priority Inversion and Starvation

- Solutions
 - Only support at most two priorities
 - Priority inheritance protocol – lower priority process accessing shared resource inherits higher priority

Mars Pathfinder



- July 4th 1997 - Mars Pathfinder lands on Mars

Mars Pathfinder

- VxWorks OS on the spacecraft provided priority with preemption scheduler
- Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft.
- A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

Mars Pathfinder

- The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
- The spacecraft also contained a communications task that ran with medium priority.

Mars Pathfinder

- Very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread.
- After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

Mars Pathfinder

- JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren't important, using the rationale "it was probably caused by a hardware glitch".

Producer / Consumer Problem

- » Also known as the bounded buffer problem.
- » The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again.
- » At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Producer / Consumer Problem

- » The solution for the producer is to either go to sleep or discard data if the buffer is full.
- » The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty.

Producer / Consumer Problem

- » Count is unconstrained, and a potential race condition
- 1. Empty buffer and consumer just read `count == 0`
- 2. Scheduler, at that instant, decides to stop running the consumer and starts the producer before consumer can act
- 3. Producer inserts an item into the buffer, increments count and notices `count == 1`. Assumes consumer is sleeping and issues wakeup.
- 4. Consumer was not yet logically asleep and misses the wakeup.
- 5. Consumer rescheduled and sees value it previously read is 0 and sleeps
- 6. Producer fills buffer then sleeps
- 7. Both sleep

POSIX Semaphores

- » There are two actions defined on semaphores (we'll go with the classic terminology): $P(\text{Sem } s)$ and $V(\text{Sem } s)$.
- » P and V are the first letters of two Dutch words *proberen* (to test) and *verhogen* (to increment).
 - The inventor of semaphores was Edsger Dijkstra who was very Dutch.

POSIX Semaphores

- » $P(\text{Sem } s)$ decrements $s \rightarrow \text{value}$, and if this is less than zero, the thread is blocked, and will remain so until another thread unblocks it. This is all done atomically.
- » $V(\text{Sem } s)$ increments $s \rightarrow \text{value}$, and if this is less than or equal to zero, then there is at least one other thread that is blocked because of s .

POSIX Semaphores

» All POSIX semaphore functions and types are prototyped or defined in `semaphore.h`. To define a semaphore object, use

```
sem_t sem_name;
```

Initializing a semaphore

» To initialize a semaphore, use `sem_init`:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `sem` points to a semaphore object to initialize
- `pshared` is a flag indicating whether or not the semaphore should be shared with `fork()`ed processes.
- `value` is an initial value to set the semaphore to

Example of use:

```
sem_init(&sem_name, 0, 10);
```

Waiting on a Semaphore

- » To wait on a semaphore, use `sem_wait`:

```
int sem_wait(sem_t *sem);
```

- » Example of use:

```
sem_wait(&sem_name);
```

- » `sem_wait` is an implementation of the DOWN operation. If the value of the semaphore is 0, the calling process blocks; one of the blocked processes wakes up when another process calls `sem_post`.

Incrementing a Semaphore

» To increment the value of a semaphore, use

`sem_post:`

```
int sem_post(sem_t *sem);
```

Example of use:

```
sem_post(&sem_name);
```

» `sem_post` is an implementation of the UP operation. It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

Query a Semaphore Value

» To find out the value of a semaphore, use

```
int sem_getvalue(sem_t *sem, int *valp);
```

» gets the current value of sem and places it in the location pointed to by valp

Example of use:

```
int value;  
sem_getvalue(&sem_name, &value);  
printf("The value of the semaphors is %d\n", value);
```

Destroy a Semaphore

» To destroy a semaphore, use

```
int sem_destroy(sem_t *sem) ;
```

» destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:

```
sem_destroy(&sem_name) ;
```

Semaphore Example

Semaphores (1)

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */
```

```
void consumer(void)
```

Figure 2-28. The producer-consumer problem using semaphores.

Semaphores (2)

```
        up(&full);                /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                /* infinite loop */
        down(&full);               /* decrement full count */
        down(&mutex);             /* enter critical region */
        item = remove_item();      /* take item from buffer */
        up(&mutex);               /* leave critical region */
        up(&empty);               /* increment count of empty slots */
        consume_item(item);        /* do something with the item */
    }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Mutex

- » Simplified version of a semaphore
- » Shared variable that can be in one of two states: locked or unlocked.
- » Can't be in a disjoint address space.

Futex

- » Spin locks are fast, but waste CPU time if wait isn't short.
- » Fast User Space Mutex (Futex)
- » Linux feature that implements basic locking, like a mutex, but avoid dropping into the kernel unless it really has to.

Futex

- Two parts
 1. Kernel service - provides a wait queue to allow multiple processes to wait for a lock
 2. User space library - works in absence of contention.
- Processes share common 32-bit variable.
 - Perform decrement and test
 - Inspect if successful, if not then system call puts thread on wait queue

Monitors

- Higher level synchronization primitive than mutex and semaphores.
 - Need to avoid race conditions and deadlocks
- Group of functions, variables and data structures grouped into a module or package
- Processes call upon the monitor but may not directly access its internals
- Language concept.
 - C does not have them.
 - `java.lang.Object` can be used as a monitor

Monitors

- Important property: can only be used by one process at a time.
- Provide mutual exclusion
- Still need solution to block processes when they can not proceed.

Condition Variables

- Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock.

Condition Variables

Main Thread <ul style="list-style-type: none">◦ Declare and initialize global data/variables which require synchronization (such as "count")◦ Declare and initialize a condition variable object◦ Declare and initialize an associated mutex◦ Create threads A and B to do work	
Thread A <ul style="list-style-type: none">◦ Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)◦ Lock associated mutex and check value of a global variable◦ Call <code>pthread_cond_wait()</code> to perform a blocking wait for signal from Thread-B. Note that a call to <code>pthread_cond_wait()</code> automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.◦ When signalled, wake up. Mutex is automatically and atomically locked.◦ Explicitly unlock mutex◦ Continue	Thread B <ul style="list-style-type: none">◦ Do work◦ Lock associated mutex◦ Change the value of the global variable that Thread-A is waiting upon.◦ Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.◦ Unlock mutex.◦ Continue
Main Thread Join / Continue	

Create and Destroy Condition Variables

```
pthread_cond_init (condition, attr)  
pthread_cond_destroy (condition)  
pthread_condattr_init (attr)  
pthread_condattr_destroy (attr)
```

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used. There are two ways to initialize a condition variable:

1. Statically, when it is declared. For example:

```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
```

2. Dynamically, with the `pthread_cond_init()` routine. The ID of the created condition variable is returned to the calling thread through the condition parameter. This method permits setting condition variable object attributes, `attr`.

Create and Destroy Condition Variables

```
pthread_cond_init (condition,attr)  
pthread_cond_destroy (condition)  
pthread_condattr_init (attr)  
pthread_condattr_destroy (attr)
```

- The optional attr object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type `pthread_condattr_t` (may be specified as NULL to accept defaults). Note that not all implementations may provide the process-shared attribute.
- The `pthread_condattr_init()` and `pthread_condattr_destroy()` routines are used to create and destroy condition variable attribute objects.
- `pthread_cond_destroy()` should be used to free a condition variable that is no longer needed.

Waiting on and Signaling Condition Variables

```
thread_cond_wait (condition,mutex)  
pthread_cond_signal (condition)  
pthread_cond_broadcast (condition)
```

- `pthread_cond_wait()` blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.
- Recommendation: Using a WHILE loop instead of an IF statement to check the waited for condition can help deal with several potential problems, such as:
 - If several threads are waiting for the same wake up signal, they will take turns acquiring the mutex, and any one of them can then modify the condition they all waited for.
 - If the thread received the signal in error due to a program bug
 - The pthreads library is permitted to issue spurious wake ups to a waiting thread without violating the standard

Waiting on and Signaling Condition Variables

```
thread_cond_wait (condition,mutex)  
pthread_cond_signal (condition)  
pthread_cond_broadcast (condition)
```

- The `pthread_cond_signal()` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for `pthread_cond_wait()` routine to complete.
- The `pthread_cond_broadcast()` routine should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.
- It is a logic error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.

Condition Variables Example

Barriers

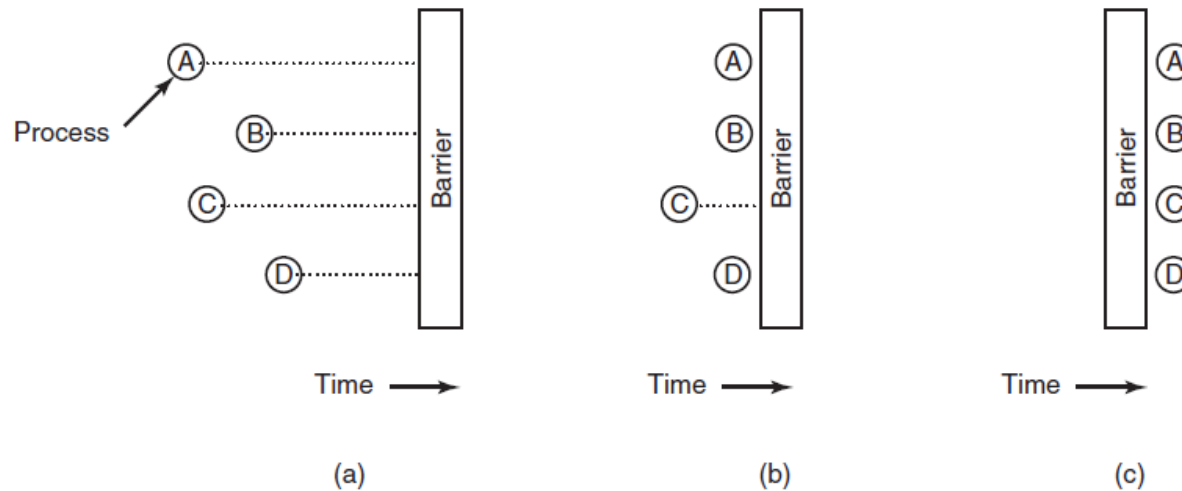


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Barrier

```
int barrierCounter
int barrierFlag

function barrier()
{
    Lock()

    if (barrierCounter == 0)
        barrierFlag = 0 //reset flag is first

    barrierCounter = barrierCounter + 1
    myCount = barrierCounter //myCount local

    UnLock()

    if (myCount == numProcessors) //last thread
        barrierCounter = 0 //reset
        barrierFlag = 1 //release
    else
        wait for barrierFlag == 0
}
```