Synchronization

Dr. Yingwu Zhu

Concurrency

- Concurrent threads/processes
  - The threads run at the same time in multiprocessing environments or their execution is interleaved through preemption
- Asynchronous
  - Threads require occasional synchronization & communication
  - For the most part, the execution of one thread neither speeds up nor slows down the execution of another.
- Independent
  - Do not have any reliance on each other
- Synchronous
  - Frequent synchronization with each other – order of execution is guaranteed!
- Parallel
  - Threads run at the same time on separate processors.

Race Condition

- A race condition is a bug
  - The outcome of concurrent threads are unexpectedly dependent on a specific sequence of events
- Cause:
  - Multiple threads access shared data and resources
  - Uncontrolled access to the shared data results in data inconsistency!

Classic Example

- Your current bank balance is $1,000
- Withdraw $500 from an ATM machine while a $5,000 direct deposit is coming in

Withdrawal	Deposit
1. Read account balance	1. Read account balance
2. Subtract 500.00	2. Add 5,000.00
3. Write account balance	3. Write account balance

- Possible outcomes
  – Total balance: $5500, $500, $6000

Synchronization

- Synchronization deals with developing techniques to avoid race conditions
- Something as simple as: x = x + 1
  - May have a race condition

```
movl   _x (%rip), %eax
addl   $1, %eax
movl   %eax, _x (%rip)
```

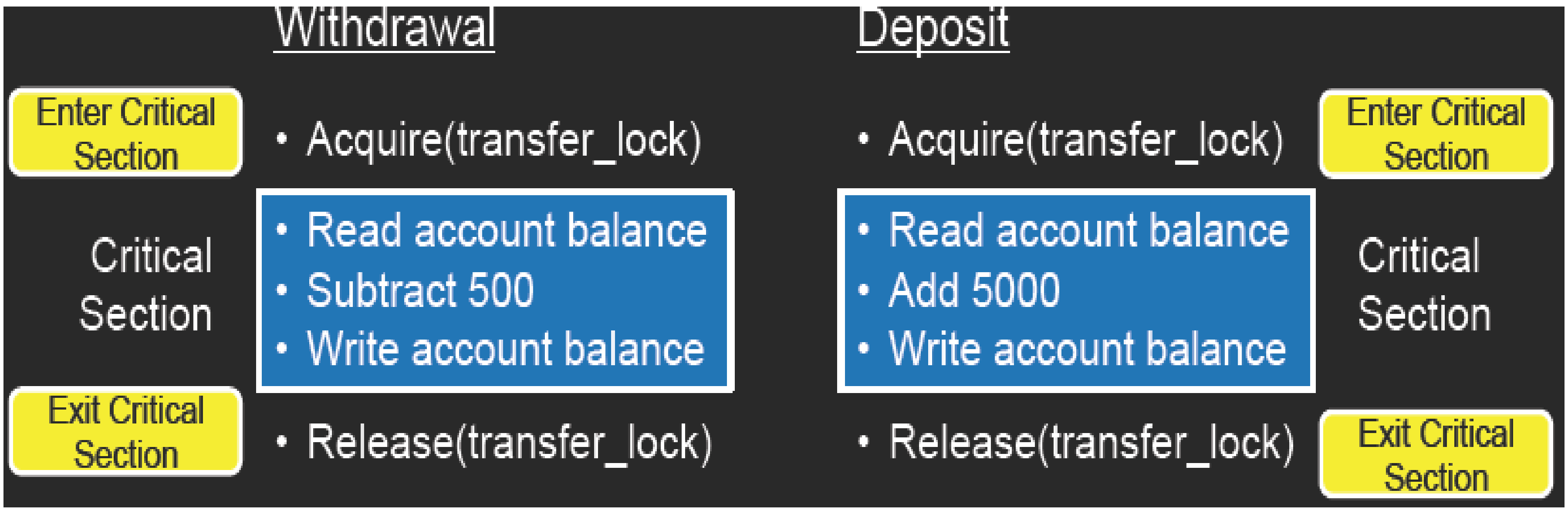
Potential points of preemption for a race condition (between the instructions above)

Mutual Exclusion

- Critical section
  - Region in a program where race conditions can arise
- Mutual exclusion
  - Allow only one thread to access a critical section at a time
- Deadlock
  - A thread is perpetually blocked (circular dependency on resources)
- Starvation
  - A thread is perpetually denied resources
- Livelock
  - Threads run but no progress in execution

Controlling Critical Section Access: Locks

- Grab and release locks around critical section
- What if cannot get a lock?



Solution to Critical-Section Problem

MUST satisfy the following requirements:

1. Mutual Exclusion -- No threads is inside the same critical section simultaneously
2. Progress - If no thread is executing in its critical section and some thread or threads want to enter the critical section, the selection of a thread that can do so cannot be delayed indefinitely.
   - if only one thread wants to enter, it should be permitted to do so.
   - If more than one wants to enter, only one of them should be allowed to.
3. Bounded Waiting - No thread should wait forever to enter a critical section.
4. No thread running outside its critical section may block others from entering a critical section
5. Performance -- The overhead of entering and exiting the critical section is small with respect to the work being done within it

A good solution ensures…

- No assumptions are made on the number of processors.
  - Threads run at the same time on different processors
- No assumptions are made on the number of threads or processes
  - Support arbitrary # of threads/processes
- No assumptions are made on the relative speed of each thread.
  - No knowledge of when or if a thread will request a critical section again

Critical section & the kernel

- Multiprocessors
  - Multiple threads/processes on different processors may access the kernel simultaneously
  - Interrupts may occur on multiple processors simultaneously
- Preemptive kernels
  - Preemptive kernel: process can be preempted while running in kernel mode
  - Nonprememptive kernel: processes running in kernel mode cannot be preempted (but interrupts can still occur!)
- Single processor, nonpreemptive kernel: free from race conditions

Solution #1: Disable Interrupts

- Disable all interrupts just before entering its critical section and re-enable them when leaving
  - OS won’t get a timer interrupt & have its scheduler preempt the thread while it is in critical section

Solution #1: Disable Interrupts

- Bad!
  - Give the thread too much control over the system
  - Stop timer update and scheduling
  - What if the logic in the critical section is incorrect? (other threads never get change to run!)
  - What if the critical section itself has a dependency on some other interrupt, thread, or system call?
    - Need read data from disk but OS won't get the disk interrupt when data is ready!
  - What about multiple processors?
    - Disabling interrupts will only disable them on one processor
- Advantage
  - Simple, guaranteed to work on a uniprocessor system
  - Was a common approach to mutual exclusion in uniprocessor kernels, at least before multiprocessors spoiled the fun.

Solution #2: Software Test & Set Locks

- Keep a shared lock variable

```
while (locked) ;
locked = 1; /* set the lock */
/* do critical section */
locked = 0; /* release the lock */
```

- Disadvantages
  - Buggy! Race condition in setting the lock
- Advantages
  - Simple to understand. It's been used for things such as locking mailbox files

Solution #3: Lockstep Synchronization

- A shared variable that tells which thread's turn

Thread 0	Thread 1
`while (turn != 0);`	`while (turn != 1);`
`critical_section();`	`critical_section();`
`turn = 1;`	`turn = 0;`

- Disadvantages
  - Busy waiting or spin lock
  - Forces strict alternation between the threads. If thread 0 is really slow, thread 1 is slowed down with it; It turns asynchronous threads into synchronous threads.

Software Solutions for Mutual Exclusion

- Peterson's solution
- Others
- Disadvantages
  - Difficult to implement correctly – have to rely on volatile data types to ensure that compilers don't make the wrong optimizations
  - Relies on busy waiting

Looking for hardware solutions

Help from Processor

- Atomic (indivisible) CPU instructions to get locks
  - Test-and-set
  - Compare-and-swap
  - Fetch-and-increment

Test-and-set

ATOMIC

```
int test_and_set(int *x) {
    last_value = *x;
    *x = 1;
    return last_value;
}
```

Set the lock but get told if it already was set (in which case you don't have it)

```
while (test_and_set(&lock)) ;
/* do critical section */
lock = 0;
```

Compare & Swap (CAS)

- Compare the value of a memory location with an old value. If they match then replace with a new value

ATOMIC

```
int compare_and_swap(int *x, int old, int new) {
    int save = *x;
    if (save == old)
        *x = new;
    return save;  /* always return location contents */
}
```

Avoid the race condition.
Set *locked* to 1 only if *locked* is still set to 0.

```
while (compare_and_swap(&locked, 0, 1) != 0) ;
            /* spin until locked == 0 */
/* if we got here, locked got set to 1 and we have it */
/* do critical section */
locked = 0;   /* release the lock */
```

Fetch & Increment

- Simply increments a memory location but returns the previous value of that memory location.
- To implement a critical section, grab a ticket and wait for your turn

```
ticket = 0; turn = 0;
...
myturn = fetch_and_increment(&ticket);
while (turn != myturn) ;
/* do critical section */
fetch_and_increment(&turn);
```

ticket

turn

Spin Locks

- All these techniques rely on spin locks
  - Waste CPU cycles
- The process with the lock may not be allowed to run!
  - Lower priority process obtained a lock
  - Higher priority process is always ready to run but loops on trying to get the lock
  - Scheduler always schedules the higher-priority process
  - Priority inversion
    - If the low priority process would get to run & release its lock, it would then accelerate the time for the high priority process to get a chance to get the lock and do useful work
    - Try explaining that to a scheduler!

Priority Inheritance

- Technique to avoid priority inversion
- Increase the priority of any process to the maximum of any process waiting on any resource for which the process has a lock
- When the lock is released, the priority goes to its normal level

Spin locks aren’t great!
Can we block until we get the critical section?

Semaphores

- An integer variable
- Have two associated operations: **wait** (also known as *p* or *down*) and **signal** (known as *v* or *up*).
- A queue of waiting processes/threads

Semaphore Implementation

- Implementation of wait:

```
wait (S){
      value--;
      if (value < 0) {
            add this thread T to waiting queue
            block(P);
      }
}
```

```
Struct Semaphore {
      int value;
      Queue q;
} S;
```

- Implementation of signal:

```
signal (S){
      value++;
      if (value <= 0) {
            remove a thread T from the waiting queue
            wakeup(P);
      }
}
```

Semaphores

- Count the number of threads that may enter a critical section at any given time.
  - Each *wait decreases the number of future accesses*
  - When no more are allowed, processes have to wait
  - Each *signal lets a waiting process get in*
- Binary semaphores
  - Initialized to 1 and used by two or more threads to ensure that only one of them can enter a critical section

Producer-Consumer Problem

- Producer
  - Generates items that go into a buffer
  - Maximum buffer capacity = N
  - If the producer fills the buffer, it must wait (sleep)
- Consumer
  - Consumes things from the buffer
  - If there's nothing in the buffer, it must wait (sleep)
- This is also known as the *Bounded-Buffer Problem*

Producer-Consumer Problem

- Use three semaphores:
- mutex: mutual exclusion to shared set of buffers
  - Binary semaphore
- empty: count of empty buffers
  - Counting semaphore
- full: count of full buffers
  - Counting semaphore

Producer-Consumer: bounded buffer

Initialization: semaphores: mutex = 1, full = 0; empty = N;
integers: int = 0, out = 0;

```
void append(int d) {
  buffer[in] = d;
  in = (in + 1) % N;
}

int take() {
  int x = out;
  out = (out+1) %N;
  return buffer[x];
}
```

Producer:

```
While (1) {
    produce x;
    wait(empty);
    wait(mutex);
    append(x);
    signal(mutex);
    signal(full);

}
```

Consumer:

```
While (1) {
    wait(full);
    wait(mutex);
    x = take();
    signal(mutex);
    signal(empty);
    consume x;
}
```

Reader-Writer Problem

- Shared data store (e.g., database)
- Multiple processes can read concurrently
- Only one process can write at a time
  – And no readers can read while the writer is writing

Synchronization
Relying on Inter-Process Communication

What problems do previous solution have?

- Assumptions
  - All concurrent threads or processors have access to common memory and share the same operating system kernel
- What if distributed systems where each system has its own local memory and its own operating system?
- Rescue: Message Passing

Communicating Processes

- Must:
  - Synchronize
  - Exchange data
- Message passing offers
  - Data communication
  - Synchronization (via waiting for messages)
  - Works with processes on different machines

Message passing

- Two primitives
  - **send**(*destination, message*)
    - Sends a message to a given destination.
  - **receive**(*source, &message*)
    - Receives a message from a source. This call could block if there is no message.
- Operations may or may not be blocking

Producer-Consumer Example

```
#define N 4          /* number of slots in the buffer */

producer() {
    int item;
    message m;

    for (;;) {
        produce_item(&item);          /* produce something */
        receive(consumer, &m);        /* wait for an empty message */
        build_message(&m, item);      /* construct the message */
        send(consumer, &m);           /* send it off */
    }
}
consumer() {
    int item, I;
    message m;

    for (i=0; i<N; ++i)
        send(producer, &m);           /* send N empty messages */
    for (;;) {
        receive(producer, &m)         /* get a message with the item */
        extract_item(&m, &item)       /* take item out of message */
        send(producer, &m);           /* send an empty reply */
        consume_item(item);           /* consume it */
    }
}
```

Messaging: Rendezvous

- Sending process blocked until receive occurs
- Receive blocks until a send occurs
- *No message buffering*
- Advantages:
  - No need for message buffering if on same system
  - Easy & efficient to implement
  - Allows for tight synchronization
- Disadvantage:
  - Forces sender & receiver to run in lockstep

Messaging: Direct Addressing

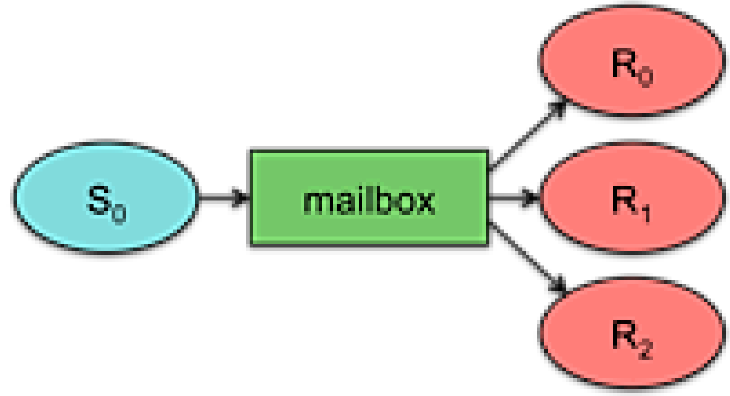
- Previous two solutions
  - Require the use of direct addressing
  - Sending process identifies receiving process
  - Receiving process can identify sending process
    - Or can receive it as a parameter

Messaging: Indirect Addressing

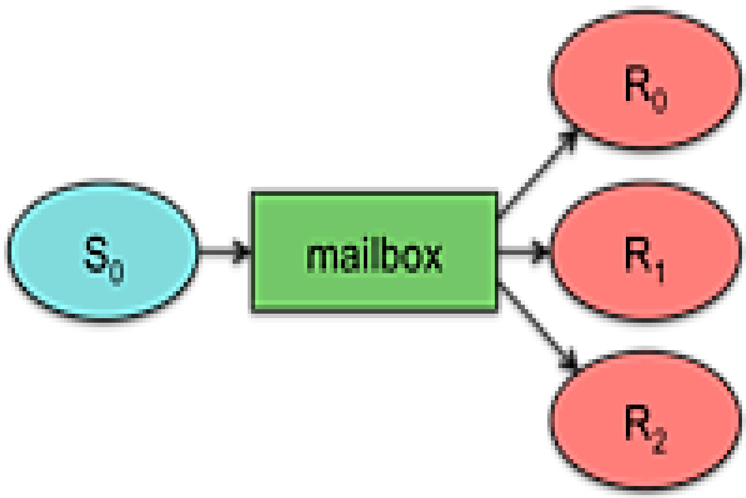
- Messages set to an intermediary data structure of FIFO queues
- Each queue is a mailbox
- Simplifies multiple readers
- Pros:
  - flexibility of having multiple senders and/or receivers.
  - Do not require the sender to know how to identify any specific receiver. Senders and receivers just need to coordinate on a mailbox identifier.
- Cons:
  - Data copying to mailbox and to receivers
  - Where should the mailbox be located?

S

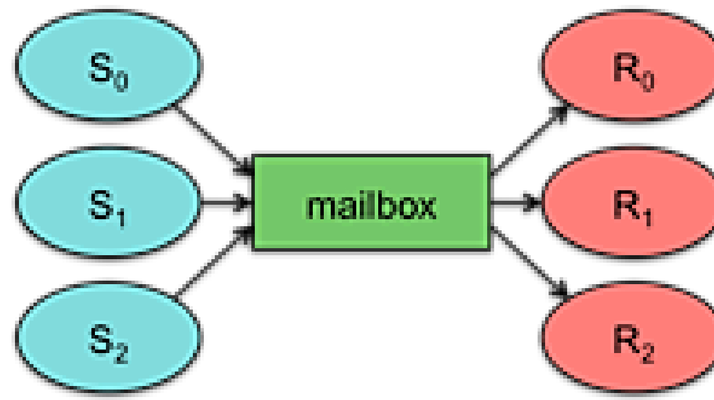
Mailbox: Single sender, single reader



Mailbox: Multiple senders, single reader



Mailbox: Single sender, multiple readers



Mailbox: Multiple senders, multiple readers

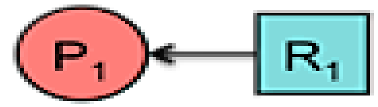
Deadlocks

- <u>Four conditions must hold</u>
  - Mutual exclusion
    - a resource can be held by at most one process.
  - Hold and wait
    - processes that already hold resources can wait for another resource.
  - Non-preemption of resources
    - Resources can only be released voluntarily
  - Circular wait
    - two or more processes are waiting for resources held by one of the other processes

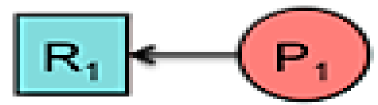
Deadlocks

- Resource allocation graph
  - Resource R1 is allocated to process P1: assignment edge
  - Resource R1 is requested by process P1:

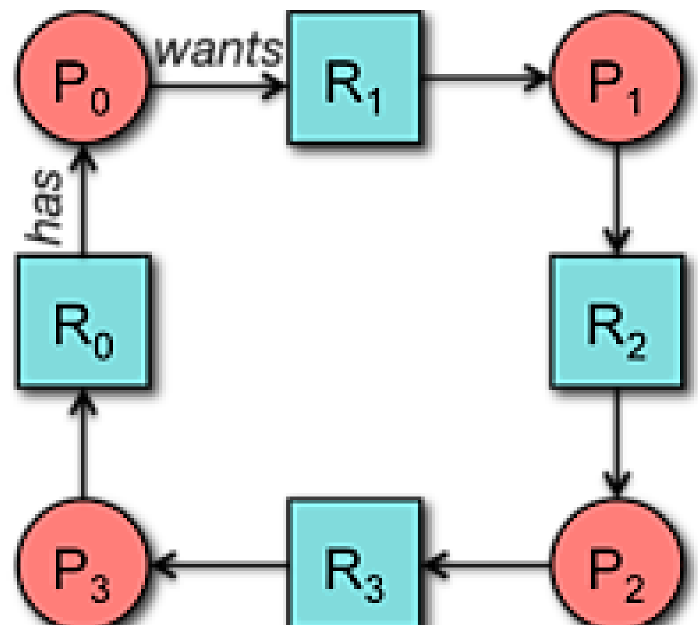
    request edge
- Deadlock is present when the graph has cycles



Assignment edge



Request edge



Resource allocation graph: deadlock!

Dealing with Deadlocks

- Deadlock prevention
  - Ensure that at least one of the necessary conditions cannot hold
- Deadlock avoidance
  - Provide advance information to the OS on which resources a process will request.
  - OS can then decide if the process should wait
- Ignore the problem
  - Let the user deal with it (most common solution)

Conditional Variables

Conditional Variables

- Condition variables provide a mechanism to wait for events (a “rendezvous point”)
  - Resource available, no more writers, etc.
- Condition variables support three operations:
  - Wait – release monitor lock, wait for C/V to be signaled
    - So condition variables have wait queues, too
  - Signal – wakeup one waiting thread
  - Broadcast – wakeup all waiting threads
- Note: Condition variables are not boolean objects
  - “if (condition_variable) then” … does not make sense
  - “if (num_resources == 0) then wait(resources_available)” does
  - An example will make this more clear

Condition Variables != Semaphores

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics
  - However, they each can be used to implement the other
- Usage: Combined with a lock
  - wait() blocks the calling thread, and gives up the lock
    - To call wait, the thread has lock
    - Semaphore::wait just blocks the thread on the queue
  - signal() causes a waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
    - Condition variables have no history

Signal Semantics

- There are two flavors of monitors that differ in the scheduling semantics of signal()
  - Hoare monitors (original)
    - signal() immediately switches from the caller to a waiting thread
    - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
    - Signaler must restore monitor invariants before signaling
  - Mesa monitors (Mesa, Java)
    - signal() places a waiter on the ready queue, but signaler continues inside monitor
    - Condition is not necessarily true when waiter runs again
      - Returning from wait() is only a hint that something changed
      - Must recheck conditional case

Hoare vs. Mesa

- Hoare

```
if (empty)
    wait(condition);
```

- Mesa

```
while (empty)
    wait(condition);
```

- Tradeoffs
  - Mesa monitors easier to use, more efficient
    - Fewer context switches, easy to support broadcast
  - Hoare monitors leave less to chance
    - Easier to reason about the program

Condition Variables vs. Locks

- Condition variables are used in conjunction with blocking locks

Summary

- Semaphores
  - wait()/signal() implement blocking mutual exclusion
  - Also used as atomic counters (counting semaphores)
  - Can be inconvenient to use
- Condition variables
  - Used by threads as a synchronization point to wait for events
  - Used with locks

Case Study: Pthread Synchronization

Mutual Exclusion

- Bad things can happen when two threads "simultaneously" access shared data structures: Race condition → critical section problem
  - Data inconsistency!
  - These types of bugs are really nasty!
    - Program may not blow up, just produces wrong results
    - Bugs are not repeatable
- Associate a separate lock (mutex) variable with the shared data structure to ensure "one at a time access"

Mutual Exclusion in PThreads

- pthread_mutex_t   mutex_var;
  - Declares mutex_var as a lock (mutex) variable
  - Holds one of two values: "locked" or "unlocked"
- pthread_mutex_lock (&mutex_var)
  - Waits/blocked until mutex_var in unlocked state
  - Sets mutex_var into locked state
- pthread_mutex_unlock (&mutex_var)
  - Sets mutex_var into unlocked state
  - If one or more threads are waiting on lock, will allow one thread to acquire lock

Example:

```
//pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t   m;   //pthread_mutex_init(&m, NULL);
...
pthread_mutex_lock (&m);
<access shared variables>
pthread_mutex_unlock(&m);
```

Pthread Semaphores

- #include <semaphore.h>
- Each semaphore has a counter value, which is a non-negative integer

Pthread Semaphores

- Two basic operations:
  - A *wait operation decrements the value of the semaphore by 1. If the value is* already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread).When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns. → sem_wait()
  - A *post operation increments the value of the semaphore by 1. If the semaphore* was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero). → sem_post()

Slightly different from our discussion on semaphores

Pthread Semaphores

- sem_t s; //define a variable
- sem_init(); //must initialize
  - 1st para: pointer to sem_t variable
  - 2nd para: must be zero
    - A nonzero value would indicate a semaphore that can be shared across processes, which is not supported by GNU/Linux for this type of semaphore.
  - 3rd para: initial value
- sem_destroy(): destroy a semaphore if do not use it anymore

Pthread Semaphores

- int sem_wait(): wait operation
- int sem_post(): signal operation
- int sem_trywait():
  - A nonblocking wait function
  - if the wait would have blocked because the semaphore's value was zero, the function returns immediately, with error value EAGAIN, instead of blocking.

Example

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>
struct job {
/* Link field for linked list. */
struct job* next;
/* Other fields describing work to be
    done...*/
};
/* A linked list of pending jobs. */
struct job* job_queue;
/* A mutex protecting job_queue. */
pthread_mutex_t job_queue_mutex =
    PTHREAD_MUTEX_INITIALIZER;
```

```
/* A semaphore counting the number of jobs in the queue. */
sem_t job_queue_count;
/* Perform one-time initialization of the job queue. */
void initialize_job_queue ()
{
    /* The queue is initially empty. */
    job_queue = NULL;
    /* Initialize the semaphore which counts jobs in the
    queue. Its initial value should be zero. */
    sem_init (&job_queue_count, 0, 0);
}
```

Assume infinite queue capacity.

Example

```
/* Process dequeued jobs until the queue is empty. */
void* thread_function (void* arg)
{
while (1) {
        struct job* next_job;
    /* Wait on the job queue semaphore. If its value is positive,indicating that the queue is not empty,
       decrement the count by 1. If the queue is empty, block until a new job is enqueued. */
     sem_wait (&job_queue_count);
    /* Lock the mutex on the job queue. */
     pthread_mutex_lock (&job_queue_mutex);
    /* Because of the semaphore, we know the queue is not empty. Get the next available job. */
     next_job = job_queue;
    /* Remove this job from the list. */
     job_queue = job_queue->next;
    /* Unlock the mutex on the job queue because we're done with the queue for now. */
     pthread_mutex_unlock (&job_queue_mutex);
    /* Carry out the work. */
     process_job (next_job);
    /* Clean up. */
     free (next_job);
 }
return NULL;
}
```

Example

```
/* Add a new job to the front of the job queue. */
void enqueue_job (/* Pass job-specific data here... */)
{
        struct job* new_job;
        /* Allocate a new job object. */
        new_job = (struct job*) malloc (sizeof (struct job));
        /* Set the other fields of the job struct here... */
        /* Lock the mutex on the job queue before accessing it. */
        pthread_mutex_lock (&job_queue_mutex);
        /* Place the new job at the head of the queue. */
        new_job->next = job_queue;
        job_queue = new_job;
        /* Post to the semaphore to indicate that another job is available. If
        threads are blocked, waiting on the semaphore, one will become
        unblocked so it can process the job. */
        sem_post (&job_queue_count);
        /* Unlock the job queue mutex. */
        pthread_mutex_unlock (&job_queue_mutex);
}
```

Can they switch order?

Waiting for Events: Condition Variables

- Mutex variables are used to control access to shared data
- Condition variables are used to wait for specific events
  - Buffer has data to consume
  - New data arrived on I/O port
  - 10,000 clock ticks have elapsed

Condition Variables

- Enable you to implement a condition under which a thread executes and, inversely, the condition under which the thread is blocked

Condition Variables in PThreads

- pthread_cond_t c_var;
  - Declares **c_var** as a condition variable
  - Always associated with a **mutex** variable (say m_var)
- pthread_cond_wait (&c_var, &m_var)
  - Atomically unlock **m_var** and block on **c_var**
  - Upon return, mutex m_var will be re-acquired
  - Spurious wakeups may occur (i.e., may wake up for no good reason - always recheck the condition you are waiting on!)
- pthread_cond_signal (&c_var)
  - If no thread blocked on **c_var**, do nothing
  - Else, unblock a thread blocked on c_var to allow one thread to be released from a pthread_cond_wait call
- pthread_cond_broadcast (&c_var)
  - Unblock all threads blocked on condition variable c_var
  - Order that threads execute unspecified; each reacquires mutex when it resumes

Waiting on a Condition

```
pthread_mutex_t
    m_var=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c_var=PTHREAD_COND_INITIALIZER;
          //pthread_cond_init()
pthread_mutex_lock (m_var);
while (<some blocking condition is true>)
     pthread_cond_wait (c_var, m_var);
<access shared data structrure>
pthread_mutex_unlock(m_var);
```

Note: Use “while” not “if”; Why?

Revisit on the example

```
void* thread_function (void* thread_arg)
{
  /* Loop infinitely.  */
  while (1) {
    /* Lock the mutex before accessing the flag value.  */
    pthread_mutex_lock (&thread_flag_mutex);
    while (!thread_flag)
      /* The flag is clear.  Wait for a signal on the condition
         variable, indicating that the flag value has changed.  When the
         signal arrives and this thread unblocks, loop and check the
         flag again.  */
      pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
    /* When we've gotten here, we know the flag must be set.  Unlock
       the mutex.  */
    pthread_mutex_unlock (&thread_flag_mutex);
    /* Do some work.  */
    do_work ();
  }
  return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Lock the mutex before accessing the flag value.  */
  pthread_mutex_lock (&thread_flag_mutex);
  /* Set the flag value, and then signal in case thread_function is
     blocked, waiting for the flag to become set.  However,
     thread_function can't actually check the flag until the mutex is
     unlocked.  */
```

Example continued…

```
  thread_flag = flag_value;
  pthread_cond_signal (&thread_flag_cv);
  /* Unlock the mutex.  */
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

Exercise

- Design a multithreaded program which handles bounded buffer problem using semaphores or conditional variables
  - int buffer[10]; //10 buffers
  - Implement producers and consumers threads