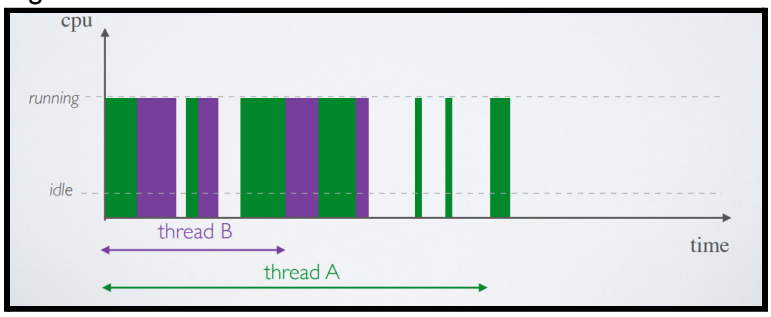
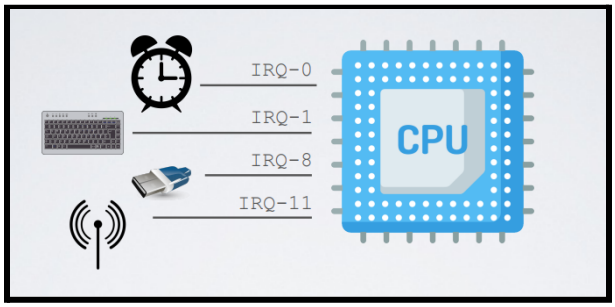
Lecture Notes:

- **Program vs Thread:**
- **Program:** Static data on some storage.
- **Thread:** An instance of a program execution.
- Different threads executing the same program can run concurrently.
- **Running threads concurrently:**
- A CPU core will run multiple threads concurrently by running each thread for a little amount of time before switching to another one.
- There is limited direct execution. The CPU will switch to another thread when either the running thread yields the CPU (non-blocking IO for instance) or the CPU stops the running thread (system clock interrupt).
  E.g.

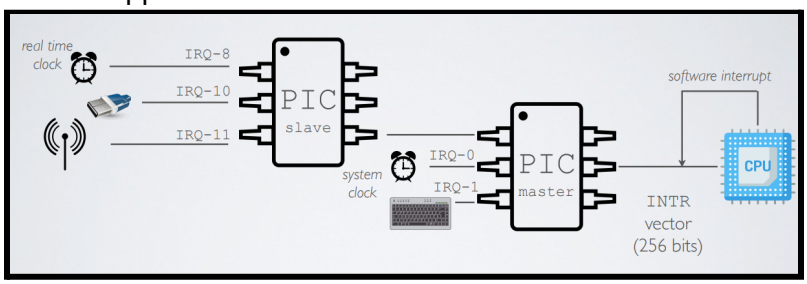


- Advantages of concurrency:
  - From the system perspective, there is better CPU usage resulting in a faster execution overall but not individually.
  - From the user perspective programs seem to be executed in parallel.
- **Interrupts:**
- There are 2 types of interrupts:
  1. **External Interrupts/Hardware Interrupts**. These are caused by an I/O device that needs some attention (asynchronous).
  2. **Internal Interrupts/System calls, exceptions and faults**. These are caused by executing instructions that have faults, such as dividing by 0 or page fault (synchronous).
- For hardware/external interrupts, this is the naive approach:



Here, I/O devices are wired to **Interrupt Request lines** (IRQs). This means that it's not flexible (hardwired) and that the CPU might get interrupted all the time.

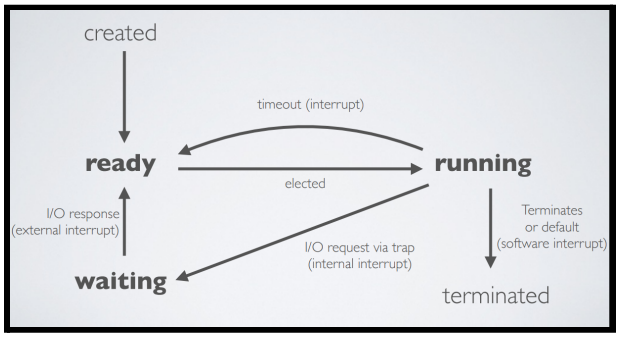
- A better approach is this:



Here, I/O devices have unique or shared IRQs that are managed by two **Programmable Interrupt Controllers** (PIC).

- The PIC is responsible to tell the CPU when and which devices wish to interrupt through the INTR vector. There are 16 lines of interrupt (IRQ0 - IRQ15), interrupts have different priority and interrupts can be masked.
- Here are the steps for handling an interrupt:
    1. The CPU receives an interrupt on the INTR vector
    2. The CPU stops the running program and transfers control to the corresponding handler in the **Interrupt Descriptor Table** (IDT)
    3. The handler saves the current running program state
    4. The handler executes the functionality
    5. The handler restores or halt the running program
- These interrupt handlers are defined in:
    - Linux: cat /proc/interrupt
    - Windows: msinfo32.exe
- E.g. When a key is pressed:
    1. The keyboard controller tells PIC to cause an interrupt on IRQ #1
    2. The PIC, which decides if the CPU should be notified
    3. If so, IRQ 1 is translated into a vector number to index into CPU's Interrupt Descriptor Table
    4. The CPU stops the current running program
    5. The CPU invokes the current handler
    6. The handler talks to the keyboard controller via IN and OUT instructions to ask what key was pressed
    7. The handler does something with the result (e.g write to a file in Linux)
    8. The handler restores the running program
- **Context Switching:**
- A **context switch** is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single (CPU), and is an essential feature of a multitasking operating system.
- When the CPU runs threads concurrently:
    - Only one thread at a time is running (on one core).
    - Several threads might be ready to be executed.
    - Several threads might be waiting for an I/O response.
- For each thread, the OS needs to keep track of its state (ready, running, waiting) and its execution context (registers, stack, heap and so on).

- The different states of a thread:



- Context switching occurs when:
    a. When the OS receives a fault:
        1. It suspends the execution of the running thread.
        2. It terminates the thread.
    b. When the OS receives a System Clock Interrupt or a System Call Trap (I/O request):
        1. It suspends the execution of the running thread.
        2. It saves its execution context.
        3. It changes the thread's state to ready (timeout) or waiting (I/O request).
        4. It elects a new thread from the ones in the ready state.
        5. It changes its state to running.
        6. It restores its execution context.
        7. It resumes its execution.
    c. When the OS receives any other I/O interrupt:
        1. It executes the I/O operation.
        2. It switches the thread that was waiting for that I/O operation, into the ready state.
        3. It resumes the execution of the current programs.
- The **Thread Control Block (TCB)** is a data structure in the operating system kernel which contains thread-specific information needed to manage it. It contains the following information:
    1. Tid (thread id)
    2. State (as either running, ready, waiting)
    3. Registers (including eip and esp)
    4. User (forthcoming lecture on user space)
    5. Pointer to a Process Control Block of the process that the thread lives on
- The OS maintains a collection of state queues with the TCBs of all the threads. There is one queue for the threads in the ready state and multiple queues for the threads in the waiting state (one queue for each type of I/O requests).
- **Synchronization:**
- **Process synchronization** is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.
- It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.
- One main error of synchronization is **race condition**. Race condition occurs when a system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

- A race condition is a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes. One or more possible outcomes may be undesirable, resulting in a bug. We refer to this kind of behavior as **nondeterministic**.
- Race condition is very hard to catch and fix.
- Example of a race condition:

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    int pid = fork();

    // Parent process
    if (pid > 0){
        printf("1\n");
    }

    // Child process
    else{
        printf("2\n");
    }
    return 0;
}
```

In the code above, we are using the fork() function to create a child process. Then, the parent process will print 1 while the child process will print 2.

On my computer, the output is 12.

```
user@Rick:~/Desktop$ gcc -Wall fork.c
user@Rick:~/Desktop$ ./a.out
1
2
```

On the UTSC computer, the output is 21.

```
@mathlab:~$ gcc -Wall fork.c
@mathlab:~$ ./a.out
2
1
```

In this example, the order doesn't really matter, but you can imagine in cases when order does matter, if the program executes in the wrong order, it can be disastrous.

Another issue with race condition is that the order executed depends on the computer the program is running on. Therefore, it can be hard to detect and fix.

- We want to use **mutual exclusion** to synchronize access to shared resources.
- Code that uses mutual exclusion to synchronize its execution is called a **critical section**.
- Only one thread at a time can execute in the critical section.
- All other threads are forced to wait on entry. The entry to the critical section is handled by the wait() function, and it is represented as P().
- When a thread leaves a critical section, another can enter. The exit from a critical section is controlled by the signal() function, represented as V().
- In the critical section, only a single process can be executed. Other processes, waiting to execute their critical section, need to wait until the current process completes its execution.
- Requirements for Critical Section:
    1. **Mutual Exclusion:** It states that if one thread is in the critical section, then no other is. Mutual exclusion ensures safety property (nothing bad happens).
    2. **Progress:** If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section. A thread in the critical section will eventually leave it.
    3. **Bound Waiting:** If some thread T is waiting on the critical section, then T will eventually enter the critical section. Progress and bounded waiting ensures the liveness property (something good happens).
    4. **Performance:** The overhead of entering and exiting the critical section is small with respect to the work being done within it.
- There are 3 main ways to achieve synchronization:
    1. **Lock/Mutex:**
        - This is a busy waiting solution which can be used for more than two processes.
        - In this mechanism, a variable or flag is used. The flag can have 2 values, 0 or 1. If the flag has a value of 0, that means the critical section is vacant. If the flag has a value of 1, that means the critical section is occupied.
        - A process which wants to get into the critical section first checks the value of the flag. If it is 0 then it sets the value to 1 and enters into the critical section, otherwise it waits.
        - The lock supports three operations:
          init() creates an unlocked mutex.
          acquire() waits until the mutex is unlocked, then locks it to enter the C.S.
          release() unlocks the mutex to leave the C.S, waking up anyone waiting for it.
    2. **Condition Variable:**
        - Condition variables are used to wait for a particular condition to become true.
        - A condition variable supports three operations:
          cond_wait(cond, lock) unlock the lock and sleep until cond is signaled then re-acquire lock before resuming execution.
          cond_signal(cond) signal the condition cond by waking up the next thread.
          cond_broadcast(cond) signal the condition cond by waking up all threads

3. **Semaphore:**
    - A semaphore is simply a variable that is non-negative and shared between threads. It is used to provide mutual exclusion.
    - A semaphore supports two operations:
      P()/decrement() blocks until semaphore is open.
      V()/increment() allows another thread to enter.
    - A semaphore safety property is that the semaphore value is always greater than or equal to 0.
    - Associated with each semaphore is a queue of waiting threads.

      When P() is called by a thread:
        - If semaphore is open, the thread continues.
        - If semaphore is closed, the thread blocks on queue.

      Then V() opens the semaphore:
        - If a thread is waiting on the queue, the thread is unblocked.
        - If no threads are waiting on the queue, the signal is remembered for the next thread.

- Semaphores may cause deadlock. **Deadlock** occurs when one thread tries to access a resource that a second holds, and vice-versa.

Textbook Notes:

- **Concurrency - An Introduction:**
- Concurrency - An Introduction:
- A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, and a thread ID.
- A **multi-threaded program** has more than one point of execution. Another way to think of this is that each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.
- The state of a single thread is thus very similar to that of a process. It has a **program counter (PC)** that tracks where the program is fetching instructions from. Each thread has its own private set of **registers** it uses for computation; thus, if there are two threads, T1 and T2, that are running on a single processor, when switching from running T1 to running T2, a **context switch** must take place.
- One other major difference between threads and processes concerns the stack. In a single-threaded process, there is a single stack, usually residing at the bottom of the address space. However, in a multi-threaded process, each thread runs independently and of course may call into various routines to do whatever work it is doing. That means, there will be a stack per thread. Any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called **thread-local storage**, the stack of the relevant thread.

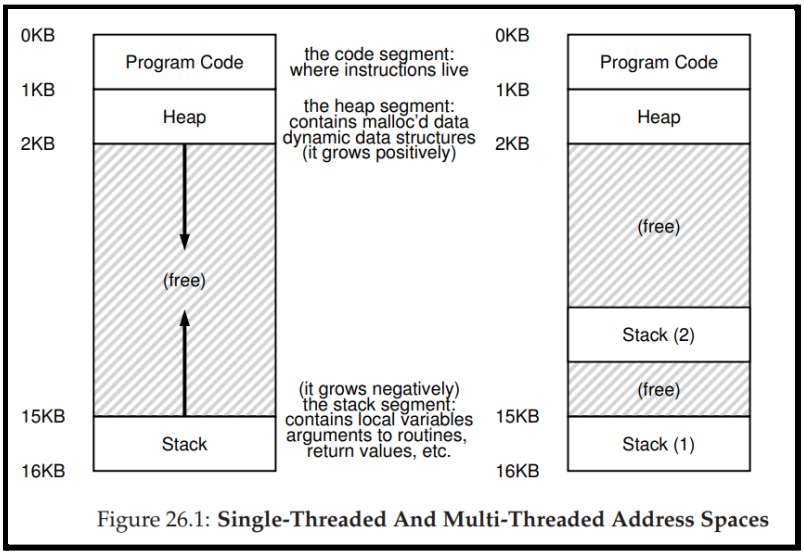


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

- Why Use Threads?:
- There are at least two major reasons you should use threads:
    1. **Parallelism.** Imagine you are writing a program that adds two large arrays together. If you are running on just a single processor, the task is straightforward: just perform each operation and be done. However, if you are executing the program on a system with multiple processors, you have the potential of speeding up this process considerably by using the processors to each perform a portion of the work. The task of transforming your standard single-threaded program into a program that does this sort of work on multiple CPUs is called **parallelization**, and using a thread per CPU to do this work is a natural and typical way to make programs run faster on modern hardware.
    2. To avoid blocking program progress due to slow I/O. Imagine that you are writing a program that performs different types of I/O. Instead of waiting, your program may wish to do something else, including utilizing the CPU to perform computation, or even issuing further I/O requests. Using threads is a natural way to avoid getting stuck; while one thread in your program **waits** (is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful. Threading enables overlap of I/O with other activities within a single program, much like multiprogramming did for processes across programs. As a result, many modern server-based applications (web servers, database management systems, etc) make use of threads in their implementations.
- An Example - Thread Creation:
- One issue with concurrency and multi-threading is **race condition**. Consider the following example: We have a function that creates 2 threads, T1 and T2. T1 will print “A” and T2 will print “B” and we want the order of the letters printed to be AB. Unfortunately, if there’s no control process or scheduler, then we might get different outputs depending on different computers.
- Why It Gets Worse - Shared Data:
- Now, consider the previous example, but this time, instead of T1 and T2 printing A and B, respectively, they will be updating shared data. Suppose that we have a variable called num, initialized to be 0, and that T1 performs some mathematical operations and updates num and T2 uses the new value of num for another function. If T2 runs before T1 updates num, then the end result will be incorrect.
- The Heart Of The Problem - Uncontrolled Scheduling:
- What we have demonstrated here is called **race condition**. The results depend on the timing execution of the code. With some bad luck, such as context switches that occur at untimely points in the execution, we will get the wrong result. In fact, we may get a different result each time; thus, instead of a nice deterministic computation, which we are used to from computers, we call this result indeterminate, where it is not known what the output will be and it is indeed likely to be different across runs. Because multiple threads executing this code can result in a race condition, we call this code a **critical section**. A critical section is a piece of code that accesses a shared variable and must not be concurrently executed by more than one thread. What we really want for this code is what we call **mutual exclusion**. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

- The Wish For Atomicity:
- **Atomic operations** are one of the most powerful underlying techniques in building computer systems, from the computer architecture, to concurrent code, to file systems, database management systems, and even distributed systems.
- The idea behind making a series of actions atomic is simply expressed with the phrase “all or nothing”; it should either appear as if all of the actions you wish to group together occurred, or that none of them occurred, with no in-between state visible. Sometimes, the grouping of many actions into a single atomic action is called a **transaction**, an idea developed in great detail in the world of databases and transaction processing.
- **Locks:**
- Locks - The Basic Idea:
- A **lock** is just a variable, and thus to use one, you must declare a lock variable of some kind. This lock variable holds the state of the lock at any instant in time. It is either available and thus no thread holds the lock, or acquired and thus exactly one thread holds the lock and presumably is in a critical section. We could store other information in the data type as well, such as which thread holds the lock, or a queue for ordering lock acquisition, but information like that is hidden from the user of the lock.
- Furthermore, we will have 2 functions, lock() and unlock(). Calling lock() tries to acquire the lock. If the lock is free, then the thread will acquire the lock and enter the critical section. This thread is sometimes said to be the owner of the lock. If another thread then calls lock() on that same lock variable, it will not return while the lock is held by another thread. In this way, other threads are prevented from entering the critical section while the first thread that holds the lock is in there.
- Once the owner of the lock calls unlock(), the lock is now free again. If no other threads are waiting for the lock, the state of the lock is simply changed to free. If there are waiting threads, one of them will notice this change of the lock’s state, acquire the lock, and enter the critical section.
- Locks provide some minimal amount of control over scheduling to programmers. In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses. Locks yield some of that control back to the programmer; by putting a lock around a section of code, the programmer can guarantee that no more than a single thread can ever be active within that code. Thus locks help transform the chaos that is traditional OS scheduling into a more controlled activity.
- Pthread Locks:
- The name that the POSIX library uses for a lock is a **mutex**, as it is used to provide mutual exclusion between threads.
- Here is the POSIX library lock code:
  ```
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
  Pthread_mutex_lock(&lock); // wrapper; exits on failure
  balance = balance + 1;
  Pthread_mutex_unlock(&lock);
  ```
- You might also notice here that the POSIX version passes a variable to lock and unlock, as we may be using different locks to protect different variables. Doing so can increase concurrency. Instead of one big lock that is used any time any critical section is accessed, one will often protect different data and data structures with different locks, thus allowing more threads to be in locked code at once.

- Evaluating Locks:
- There are 3 basic criterias for locks to be useful:
    1. Whether the lock provides **mutual exclusion**.
    2. Is there **fairness**? Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: does any thread contending for the lock starve while doing so, thus never obtaining it?
    3. The final criterion is **performance**, specifically the time overheads added by using the lock. There are a few different cases that are worth considering here. One is the case of no contention; when a single thread is running and grabs and releases the lock, what is the overhead of doing so? Another is the case where multiple threads are contending for the lock on a single CPU; in this case, are there performance concerns? Finally, how does the lock perform when there are multiple CPUs involved, and threads on each contending for the lock?
- Controlling Interrupts:
- One of the earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections. This solution was invented for single-processor systems.
- By turning off interrupts before entering a critical section, we ensure that the code inside the critical section will not be interrupted, and thus will execute as if it were atomic. When we are finished, we re-enable interrupts and thus the program proceeds as usual.
- The main positive of this approach is its simplicity. You certainly don't have to scratch your head too hard to figure out why this works. Without interruption, a thread can be sure that the code it executes will execute and that no other thread will interfere with it.
- The negatives, unfortunately, are many:
    1. This approach requires us to allow any calling thread to perform a privileged operation (turning interrupts on and off), and thus trust that this facility is not abused. As you already know, any time we are required to trust an arbitrary program, we are probably in trouble. Here, the trouble manifests in numerous ways: a greedy program could call lock() at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call lock() and go into an endless loop. In this latter case, the OS never regains control of the system, and there is only one recourse: restart the system. Using interrupt disabling as a general purpose synchronization solution requires too much trust in applications.
    2. This approach does not work on multiprocessors. If multiple threads are running on different CPUs, and each try to enter the same critical section, it does not matter whether interrupts are disabled; threads will be able to run on other processors, and thus could enter the critical section. As multiprocessors are now commonplace, our general solution will have to do better than this.
    3. Turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems. Imagine, for example, if the CPU missed the fact that a disk device has finished a read request. How will the OS know to wake the process waiting for said read?
    4. This approach can be inefficient. Compared to normal instruction execution, code that masks or unmasks interrupts tends to be executed slowly by modern CPUs.

- A Failed Attempt - Just Using Loads/Stores:
- To move beyond interrupt-based techniques, we will have to rely on CPU hardware and the instructions it provides us to build a proper lock.
- Let's first try to build a simple lock by using a single flag variable. In this failed attempt, we'll see some of the basic ideas needed to build a lock, and see why just using a single variable and accessing it via normal loads and stores is insufficient.
- Here is the code:

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
        // 0 -> lock is available, 1 -> held
        mutex->flag = 0;
}

void lock(lock_t *mutex) {
        while (mutex->flag == 1) // TEST the flag
                ; // spin-wait (do nothing)
        mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
        mutex->flag = 0;
}
```

- In this first attempt, the idea is quite simple: use a simple variable to indicate whether some thread has possession of a lock. The first thread that enters the critical section will call lock(), which tests whether the flag is equal to 1 and then sets the flag to 1 to indicate that the thread now holds the lock. When finished with the critical section, the thread calls unlock() and clears the flag, thus indicating that the lock is no longer held. If another thread happens to call lock() while that first thread is in the critical section, it will simply spin-wait in the while loop for that thread to call unlock() and clear the flag. Once that first thread does so, the waiting thread will fall out of the while loop, set the flag to 1 for itself, and proceed into the critical section.
- There are 2 problems with this design:
    1. With untimely interrupts, we can have 2 threads both set the variable to 1.
    2. This design has bad performance. While a thread waits to acquire a lock that is already held it endlessly checks the value of the flag, a technique known as **spin-waiting**. Spin-waiting wastes time waiting for another thread to release a lock. The waste is exceptionally high on a uniprocessor, where the thread that the waiter is waiting for cannot even run.

- Building Working Spin Locks with Test-And-Set:
- Because disabling interrupts does not work on multiple processors, and because simple approaches using loads and stores don't work, system designers started to invent hardware support for locking.
- The simplest bit of hardware support to understand is known as a **test-and-set** (or **atomic exchange**) instruction.
- Here is the code:

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

- What the test-and-set instruction does is as follows. It returns the old value pointed to by the old ptr, and simultaneously updates said value to new. The key is that this sequence of operations is performed atomically. The reason it is called "test and set" is that it enables you to "test" the old value while simultaneously setting the memory location to a new value. As it turns out, this slightly more powerful instruction is enough to build a simple spin lock.
- Let's make sure we understand why this lock works. Imagine first the case where a thread calls lock() and no other thread currently holds the lock; thus, flag should be 0. When the thread calls TestAndSet(flag, 1), the routine will return the old value of flag, which is 0; thus, the calling thread, which is testing the value of flag, will not get caught spinning in the while loop and will acquire the lock. The thread will also atomically set the value to 1, thus indicating that the lock is now held. When the thread is finished with its critical section, it calls unlock() to set the flag back to zero.
- The second case we can imagine arises when one thread already has the lock held (i.e., flag is 1). In this case, this thread will call lock() and then call TestAndSet(flag, 1) as well. This time, TestAndSet() will return the old value at flag, which is 1 (because the lock is held), while simultaneously setting it to 1 again. As long as the lock is held by another thread, TestAndSet() will repeatedly return 1, and thus this thread will spin and spin until the lock is finally released. When the flag is finally set to 0 by some other thread, this thread will call TestAndSet() again, which will now return 0 while atomically setting the value to 1 and thus acquire the lock and enter the critical section.

- By making both the test and set a single atomic operation, we ensure that only one thread acquires the lock. And that's how to build a working mutual exclusion primitive.
- You may also now understand why this type of lock is usually referred to as a **spin lock**. It is the simplest type of lock to build, and simply spins, using CPU cycles, until the lock becomes available. To work correctly on a single processor, it requires a **preemptive scheduler** (one that will interrupt a thread via a timer, in order to run a different thread, from time to time). Without preemption, spin locks don't make much sense on a single CPU, as a thread spinning on a CPU will never relinquish it.
- Evaluating Spin Locks:
- Given our basic spin lock, we can now evaluate how effective it is along our previously described axes. The most important aspect of a lock is correctness: does it provide mutual exclusion? The answer here is yes: the spin lock only allows a single thread to enter the critical section at a time. Thus, we have a correct lock.
- The next axis is fairness. How fair is a spin lock to a waiting thread? Can you guarantee that a waiting thread will ever enter the critical section? The answer here, unfortunately, is bad news: spin locks don't provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention. Simple spin locks, as discussed thus far, are not fair and may lead to starvation.
- The final axis is performance. What are the costs of using a spin lock? To analyze this more carefully, we suggest thinking about a few different cases. In the first, imagine threads competing for the lock on a single processor; in the second, consider threads spread out across many CPUs. For spin locks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is preempted within a critical section. The scheduler might then run every other thread (imagine there are N − 1 others), each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles. However, on multiple CPUs, spin locks work reasonably well, if the number of threads roughly equals the number of CPUs. The thinking goes as follows: imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock. If Thread A (CPU 1) grabs the lock, and then Thread B tries to, B will spin (on CPU 2). However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective.
- Compare-And-Swap:
- Another hardware primitive that some systems provide is known as the **compare-and-swap** instruction or **compare-and-exchange**.
- Here is the code:

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

- The basic idea is for compare-and-swap to test whether the value at the address specified by ptr is equal to expected. If so, update the memory location pointed to by ptr with the new value. If not, do nothing. In either case, return the original value at that memory location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

- Load-Linked and Store-Conditional:
- Some platforms provide a pair of instructions that work in concert to help build critical sections. On the MIPS architecture, for example, the **load-linked** and **store-conditional** instructions can be used in tandem to build locks and other concurrent structures.
- Here is the code:

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

- The load-linked operates much like a typical load instruction, and simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place. In the case of success, the storeconditional returns 1 and updates the value at ptr to value; if it fails, the value at ptr is not updated and 0 is returned.
- Fetch-And-Add:
- One final hardware primitive is the **fetch-and-add** instruction, which atomically increments a value while returning the old value at a particular address.
- Here is the code:

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

- Instead of a single value, this solution uses a ticket and turn variable in combination to build a lock. The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread's "turn" (myturn). The globally shared lock->turn is then used to determine which thread's turn it is; when (myturn == turn) for a given thread, it is that thread's turn to enter the critical section. Unlock is accomplished simply by incrementing the turn such that the next waiting thread, if there is one, can now enter the critical section.
- Note one important difference with this solution versus our previous attempts: it ensures progress for all threads. Once a thread is assigned its ticket value, it will be scheduled at some point in the future. In our previous attempts, no such guarantee existed; a thread

spinning on test-and-set could spin forever even as other threads acquire and release the lock.

- A Simple Approach - Just Yield:
- Hardware support got us pretty far: working locks, and even fairness in lock acquisition. However, we still have a problem: what to do when a context switch occurs in a critical section, and threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again?
- Our first try is a simple and friendly approach: when you are going to spin, instead give up the CPU to another thread.
- In this approach, we assume an operating system primitive yield() which a thread can call when it wants to give up the CPU and let another thread run. A thread can be in one of three states (running, ready, or blocked). Yield is simply a system call that moves the caller from the running state to the ready state, and thus promotes another thread to running. Thus, the yielding thread essentially deschedules itself.
- Think about the example with two threads on one CPU; in this case, our yield-based approach works quite well. If a thread happens to call lock() and find a lock held, it will simply yield the CPU, and thus the other thread will run and finish its critical section. In this simple case, the yielding approach works well.
- Let us now consider the case where there are many threads (say 100) contending for a lock repeatedly. In this case, if one thread acquires the lock and is preempted before releasing it, the other 99 will each call lock(), find the lock held, and yield the CPU. Assuming some kind of round-robin scheduler, each of the 99 will execute this run-and-yield pattern before the thread holding the lock gets to run again. While better than our spinning approach, this approach is still costly; the cost of a context switch can be substantial, and there is thus plenty of waste. Worse, we have not tackled the starvation problem at all. A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section. We clearly will need an approach that addresses this problem directly.
- One good reason to avoid spin locks is performance. If a thread is interrupted while holding a lock, other threads that use spin locks will spend a large amount of CPU time just waiting for the lock to become available. However, it turns out there is another interesting reason to avoid spin locks on some systems: **priority inversion**. Priority inversion is a scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively inverting the relative priorities of the two tasks.
- Let's assume there are two threads in a system. Thread 2 (T2) has a high scheduling priority, and Thread 1 (T1) has lower priority. In this example, let's assume that the CPU scheduler will always run T2 over T1, if indeed both are runnable; T1 only runs when T2 is not able to do so (e.g., when T2 is blocked on I/O). Now, the problem. Assume T2 is blocked for some reason. So T1 runs, grabs a spin lock, and enters a critical section. T2 now becomes unblocked (perhaps because an I/O completed), and the CPU scheduler immediately schedules it (thus descheduling T1). T2 now tries to acquire the lock, and because it can't (T1 holds the lock), it just keeps spinning. Because the lock is a spin lock, T2 spins forever, and the system is hung.
- Just avoiding the use of spin locks, unfortunately, does not avoid the problem of inversion. Imagine three threads, T1, T2, and T3, with T3 at the highest priority, and T1 the lowest. Imagine now that T1 grabs a lock. T3 then starts, and because it is higher priority than T1, runs immediately (preempting T1). T3 tries to acquire the lock that T1 holds, but gets stuck waiting, because T1 still holds it. If T2 starts to run, it will have higher priority than T1, and thus it will run. T3, which is higher priority than T2, is stuck waiting for T1, which may never run now that T2 is running.

- You can address the priority inversion problem in a number of ways. In the specific case where spin locks cause the problem, you can avoid using spin locks. More generally, a higher-priority thread waiting for a lower-priority thread can temporarily boost the lower thread's priority, thus enabling it to run and overcoming the inversion, a technique known as priority inheritance. A last solution is simplest: ensure all threads have the same priority.
- Using Queues - Sleeping Instead Of Spinning:
- The real problem with our previous approaches is that they leave too much to chance. The scheduler determines which thread runs next; if the scheduler makes a bad choice, a thread runs that must either spin waiting for the lock, or yield the CPU immediately. Either way, there is potential for waste and no prevention of starvation.
- For simplicity, we will use the support provided by Solaris, in terms of two calls: park() to put a calling thread to sleep, and unpark(threadID) to wake a particular thread as designated by threadID. These two routines can be used in tandem to build a lock that puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free.
- We do a couple of interesting things in this example. First, we combine the old test-and-set idea with an explicit queue of lock waiters to make a more efficient lock. Second, we use a queue to help control who gets the lock next and thus avoid starvation.
- Here is the code:

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

- You might notice how the guard is used basically as a spin-lock around the flag and queue manipulations the lock is using. This approach thus doesn't avoid spin-waiting entirely. A thread might be interrupted while acquiring or releasing the lock, and thus cause other threads to spin-wait for this one to run again. However, the time spent spinning is quite limited (just a few instructions inside the lock and unlock code, instead of the user-defined critical section), and thus this approach may be reasonable.
- You might also observe that in lock(), when a thread can not acquire the lock, we are careful to add ourselves to a queue, set guard to 0, and yield the CPU. A question for the reader: What would happen if the release of the guard lock came after the park(), and not before? Hint: something bad.
- You might further detect that the flag does not get set back to 0 when another thread gets woken up. This is not an error, but rather a necessity. When a thread is woken up, it will be as if it is returning from park(); however, it does not hold the guard at that point in the code and thus cannot even try to set the flag to 1. Thus, we just pass the lock directly from the thread releasing the lock to the next thread acquiring it.

- Finally, you might notice the perceived race condition in the solution, just before the call to park(). With just the wrong timing, a thread will be about to park, assuming that it should sleep until the lock is no longer held. A switch at that time to another thread (say, a thread holding the lock) could lead to trouble, for example, if that thread then released the lock. The subsequent park by the first thread would then sleep forever, a problem sometimes called the **wakeup/waiting race**.
- Solaris solves this problem by adding a third system call: setpark(). By calling this routine, a thread can indicate it is about to park. If it then happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping.
- **Semaphores:**
- Semaphores - A Definition:
- A **semaphore** is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are sem wait() and sem post().
- Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value.
- Here is the code for initializing a semaphore:

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

- In the code, we declare a semaphore s and initialize it to the value 1 by passing 1 in as the third argument. The second argument to sem_init() will be set to 0 in all of the examples we'll see. This indicates that the semaphore is shared between threads in the same process.
- After a semaphore is initialized, we can call one of two functions to interact with it, sem wait() or sem post().
- Here is the implementation for sem_wait() and sem_post():

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

- **sem_wait(sem_t *sem)** decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement, or a signal handler interrupts the call.
- **sem_post(sem_t *sem)** increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait() call will be woken up and proceed to lock the semaphore.

- Binary Semaphores (Locks):
- Our first use will be one with which we are already familiar: using a semaphore as a lock.
- Here is the code:

```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

- Looking back at the definition of the sem wait() and sem post() routines above, we can see that the initial value should be 1.

  To make this clear, let's imagine a scenario with two threads. The first thread (Thread 0) calls sem wait(); it will first decrement the value of the semaphore, changing it to 0. Then, it will wait only if the value is not greater than or equal to 0. Because the value is 0, sem wait() will simply return and the calling thread will continue; Thread 0 is now free to enter the critical section. If no other thread tries to acquire the lock while Thread 0 is inside the critical section, when it calls sem post(), it will simply restore the value of the semaphore to 1 and not wake a waiting thread, because there are none.
  Here is a trace of the above example:

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	
1	`sem_post()` returns	

  A more interesting case arises when Thread 0 holds the lock (it has called sem wait() but not yet called sem post()), and another thread (Thread 1) tries to enter the critical section by calling sem wait(). In this case, Thread 1 will decrement the value of the semaphore to -1, and thus wait (putting itself to sleep and relinquishing the processor). When Thread 0 runs again, it will eventually call sem post(), incrementing the value of the semaphore back to zero, and then wake the waiting thread (Thread 1), which will then be able to acquire the lock for itself. When Thread 1 finishes, it will again increment the value of the semaphore, restoring it to 1 again.
  Here is a trace of the above example:

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

- Thus we are able to use semaphores as locks. Because locks only have two states (held and not held), we sometimes call a semaphore used as a lock a **binary semaphore**.

- Semaphores For Ordering:
- Semaphores are also useful to order events in a concurrent program.
- For example, a thread may wish to wait for a list to become non-empty, so it can delete an element from it. In this pattern of usage, we often find one thread waiting for something to happen, and another thread making that something happen and then signaling that it has happened, thus waking the waiting thread.
- The Producer/Consumer (Bounded Buffer) Problem:
- The next problem we will confront in this chapter is known as the **producer/consumer problem**, or sometimes as the **bounded buffer problem**.
- Problem Statement: We have a buffer of fixed size. A producer can produce an item and can place it in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time the consumer should not consume any item. In this problem, the buffer is the critical section.
- To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of the number of items in the buffer at any given time and "Empty" keeps track of the number of unoccupied slots.

Initialization of semaphores –

mutex = 1

Full = 0 // Initially, all slots are empty. Thus full slots are 0

Empty = n // All slots are empty initially

Solution for Producer –

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

Solution for Consumer –

```
do{

wait(full);
wait(mutex);

// remove item from buffer

signal(mutex);
signal(empty);

// consumes item

}while(true)
```

When a producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

As the consumer is removing an item from the buffer, therefore the value of “full” is reduced by 1 and the value of mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

I.e. Semaphores solve the problem of lost wakeup calls. In the solution below, we use two semaphores, Full and Empty, to solve the problem. Full is the number of items already in the buffer and available to be read, while Empty is the number of available spaces in the buffer where items could be written. Full is incremented and Empty decremented when a new item is put into the buffer. If the producer tries to decrement Empty when its value is zero, the producer is put to sleep. The next time an item is consumed, Empty is incremented and the producer wakes up. The consumer works analogously.

- Reader-Writer Locks:
- Another classic problem stems from the desire for a more flexible locking primitive that admits that different data structure accesses might require different kinds of locking.
- For example, imagine a number of concurrent list operations, including inserts and simple lookups. While inserts change the state of the list, lookups simply read the data structure; as long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently. The special type of lock we will now develop to support this type of operation is known as a **reader-writer lock**.
- Here is the code:

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int   readers;   // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

- The code is pretty simple. If some thread wants to update the data structure in question, it should call the new pair of synchronization operations: rwlock acquire writelock(), to acquire a write lock, and rwlock release writelock(), to release it. Internally, these simply use the writelock semaphore to ensure that only a single writer can acquire the lock and thus enter the critical section to update the data structure in question.
- More interesting is the pair of routines to acquire and release read locks. When acquiring a read lock, the reader first acquires lock and then increments the readers variable to track how many readers are currently inside the data structure. The important step then taken within rwlock to acquire readlock() occurs when the first reader acquires the lock; in that case, the reader also acquires the write lock by calling sem wait() on the writelock semaphore, and then releasing the lock by calling sem post().
- Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until all readers are finished; the last one to exit the critical section calls sem post() on “writelock” and thus enables a waiting writer to acquire the lock.
- This approach works as desired, but does have some negatives, especially when it comes to fairness. In particular, it would be relatively easy for readers to starve writers.
- Finally, it should be noted that reader-writer locks should be used with some caution. They often add more overhead and thus do not end up speeding up performance as compared to just using simple and fast locking primitives.
- Thread Throttling:
- One other simple use case for semaphores is this: How can a programmer prevent too many threads from doing something at once and bogging the system down? Answer: decide upon a threshold for too many, and then use a semaphore to limit the number of threads concurrently executing the piece of code in question. We call this approach **thread throttling**, and consider it a form of admission control.
- Let’s consider a more specific example. Imagine that you create hundreds of threads to work on some problem in parallel. However, in a certain part of the code, each thread acquires a large amount of memory to perform part of the computation; let’s call this part of the code the memory-intensive region. If all of the threads enter the memory-intensive region at the same time, the sum of all the memory allocation requests will exceed the amount of physical memory on the machine. As a result, the machine will start thrashing, and the entire computation will slow to a crawl.

  A simple semaphore can solve this problem. By initializing the value of the semaphore to the maximum number of threads you wish to enter the memory-intensive region at once, and then putting a sem wait() and sem post() around the region, a semaphore can naturally throttle the number of threads that are ever concurrently in the dangerous region of the code.

- **Common Concurrency Problems:**
- Non-Deadlock Bugs:
    - **Atomicity-Violation Bugs:** The desired serializability among multiple memory accesses is violated.
      E.g.

```
Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

Thread 2::
thd->proc_info = NULL;
```

      In the example, two different threads access the field proc info in the structure thd. The first thread checks if the value is non-NULL and then prints its value; the second thread sets it to NULL. Clearly, if the first thread performs the check but then is interrupted before the call to fputs, the second thread could run in-between, thus setting the pointer to NULL; when the first thread resumes, it will crash, as a NULL pointer will be dereferenced by fputs.
    - **Order-Violation Bugs:** The desired order between two groups of memory accesses is flipped. The fix to this type of bug is generally to enforce ordering. As discussed previously, using condition variables is an easy and robust way to add this style of synchronization into modern code bases.
      E.g.

```
Thread 1::
void init() {
   mThread = PR_CreateThread(mMain, ...);
}

Thread 2::
void mMain(...) {
    mState = mThread->State;
}
```

      The code in Thread 2 seems to assume that the variable mThread has already been initialized and is not NULL. However, if Thread 2 runs immediately once created, the value of mThread will not be set when it is accessed within mMain() in Thread 2, and will likely crash with a NULL-pointer dereference. Note that we assume the value of mThread is initially NULL; if not, even stranger things could happen as arbitrary memory locations are accessed through the dereference in Thread 2.
- Deadlock Bugs:
- Beyond the concurrency bugs mentioned above, a classic problem that arises in many concurrent systems with complex locking protocols is known as **deadlock**.
- Deadlock occurs, for example, when a thread (Thread 1) is holding a lock (L1) and waiting for another one (L2). Unfortunately, the thread (Thread 2) that holds lock L2 is waiting for L1 to be released.

- One reason why deadlocks occur is that in large code bases, complex dependencies arise between components. Take the operating system, for example. The virtual memory system might need to access the file system in order to page in a block from disk. The file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system. Thus, the design of locking strategies in large systems must be carefully done to avoid deadlock in the case of circular dependencies that may occur naturally in the code.
- Another reason is due to the nature of encapsulation. As software developers, we are taught to hide details of implementations and thus make software easier to build in a modular way. Unfortunately, such modularity does not mesh well with locking.
- Conditions for Deadlock - There are four conditions need to hold for a deadlock to occur:
    1. **Mutual exclusion:** Threads claim exclusive control of resources that they require.
    2. **Hold-and-wait:** Threads hold resources allocated to them while waiting for additional resources.
    3. **No preemption:** Resources cannot be forcibly removed from threads that are holding them.
    4. **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain.
- **Circular wait:** Probably the most practical prevention technique and certainly one that is frequently employed is to write your locking code such that you never induce a circular wait. The most straightforward way to do that is to provide a total ordering on lock acquisition. For example, if there are only two locks in the system (L1 and L2), you can prevent deadlock by always acquiring L1 before L2. Such strict ordering ensures that no cyclical wait arises; hence, no deadlock.

  Of course, in more complex systems, more than two locks will exist, and thus total lock ordering may be difficult to achieve. Thus, a partial ordering can be a useful way to structure lock acquisition so as to avoid deadlock.
- **Hold-and-wait:** The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically.
- **No Preemption:** Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble because when waiting for one lock we are holding another. Many thread libraries provide a more flexible set of interfaces to help avoid this situation. Specifically, the routine pthread mutex trylock() either grabs the lock if it is available and returns success or returns an error code indicating the lock is held. In the latter case, you can try again later if you want to grab that lock.
- **Deadlock Avoidance via Scheduling:** Instead of deadlock prevention, in some scenarios deadlock avoidance is preferable. Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently schedules said threads in a way as to guarantee no deadlock can occur.
- **Detect and Recover:** One final general strategy is to allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected.