27.3: Locks

- Locks provide **mutual exclusion** to a critical section so that only one thread can access it at a time
- int pthread_mutex_lock(pthread_mutext_t *mutex) as well as unlock
- Lock and unlock your critical section before and after use
- Intent of the code
    - If no other thread holds the lock when pthread_mutext_lock() is called, the thread will acquire the lock and enter the critical section
    - the thread will not return until it has released the lock via the unlock() call
- **lack of proper initialization:** All locks must be properly initialized in order to guarantee that they have the correct values to begin with and thus work
- pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER
    - or int rc= pthread_mutex_init(&lock, NULL)
    - when you are done with the lock, call pthread_mutext_destroy()

27.4: Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue
- To use a condition variable, one has to in addition have a lock that is associated with the condition
- pthread_cond_wait(), puts the calling thread to sleep, and thus waits for some other thread to signal it
- Ex: after initialization of the relevant lock and condition, a thread checks to see if the variable has been set to something other than zero -> calls sleep to wait until some other thread wakes it
- The wait call takes a lock as a second parameter, whereas the signal call only takes a condition
    - The wait call puts the thread to sleep and also **releases** the lock
    - However, before returning from being woken, the pthread_cond_wait() re-acquires the lock
- One last oddity: the waiting thread re-checks the condition in a while loop, instead of a simple if statement
- Don't ever use a simple thread flag to signal between two threads, instead of a condition variable and associated lock
    - Performs poorly in many cases (spinning for a long time just wastes CPU cycles)
    - Also very error prone

Chapter 28: Locks

- We would like to execute a series of instructions atomically, but due to the process of interrupts on a single processor (or multiple threads executing on multiple processors concurrently), we couldn't

28.1: Locks: The Basic Idea

- As an example, assume critical section is balance = balance+1
- To implement a lock we would use
    - lock_t mutext -> lock(&mutex); balance = balance +1 unlock(&mutex)
- A lock is just a variable that holds the state of the lock at any instant in time
    - either **available** (unlocked or free) or **acquired** (or locked or held) -> only one lock holds the critical section
- Calling the lock() routine tries to acquire the lock
    - if no other threads hold the lock, the thread will acquire the lock
    - If another thread then calls lock() on that same variable,it will not return while the lock is being held by another thread
- Once the lock is unlocked, the waiting threads in the queue will be notified and acquire the lock

28.2: Pthread Locks

- The name that POSIX uses for a lock is **mutex**

28.3: Building a Lock

- To build a lock, we need help from the hardware and the OS

28.4: Evaluating Locks

- We need the lock to the basic task of providing mutual exclusion, at least
- Locking Goals
- **Mutual Exclusion**
- **Fairness:** Does each thread contending for the lock get a fair shot at acquiring once free?
    - Does any thread contending for the lock **starve** while doing so, thus never obtaining it?
- **Performance:** Time overheads added by implementing/using the lock
    - What is the overhead with no contention;
        - when a single thread is running and grabs and releases the lock
    - Are there performance concerns over contending threads for a lock

28.5: Controlling Interrupts

- One of the earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections -> for single processor systems
- By turning off interrupts before entering a critical section, we can ensure atomicity
- When finished with the critical section, we can re-enable interrupts
- Advantages to this approach: Simplicity -> very easy to implement
- Disadvantages
    - Any calling thread must be able to call a privileged operation in order to disable and enable interrupts

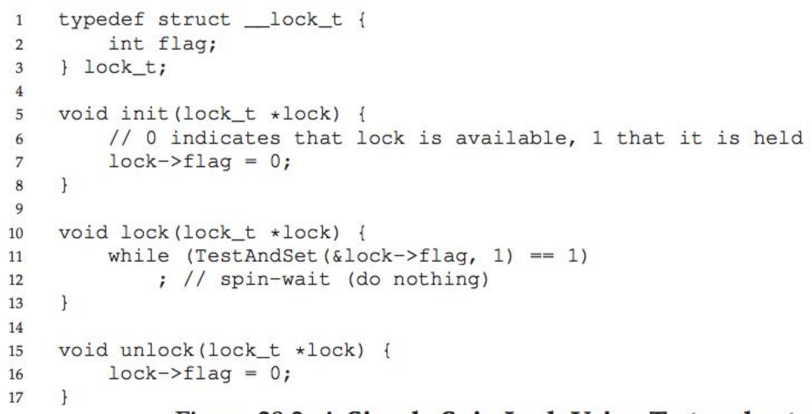
    - Game the processor: A greedy program can call lock() at the beginning of the operation and then monopolize the processor ->malicious function can keep looping through lock
    - Approach does not work on multiprocessors. If multiple threads are running on different CPUs, each try to enter the same critical section, it does not matter whether interrupts are disabled
        - Threads can run on different processors to enter the critical section
    - Turning off interrupts for extended periods of time can lead to interrupts being lost
    - Code that masks or unmasks interrupts can be slow and inefficient

28.6: A Failed Attempt: Just Using Loads/Stores

- 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 hold the lock
    - When finished with the critical section, the thread can call unlock() to clear the flag
- If another thread happens to call lock() while the first thread is in the critical section, it will simply **spin-wait** in the while loop for that thread to call unlock and then clear the flag
- If a thread waits to acquire a lock that is already held, it endlessly checks the value of flag, a technique known as **spin-waiting**
    - **wastes a lot of time**

28.7: Building Working Spin Locks with Test-And-Set

- Disabling interrupt approaches don't work with multiprocessors and simple approaches using load and store don't work because of the overall correctness + performance issues
- Invent hardware support for locking
- **test-and-set instruction (atomic exchange)**: Returns the old value pointed by the ptr and simultaneously updates said value to new -> performed atomically
- Enables you to "test" the old value (which is what is returned) while simultaneously "setting" the memory location to a new value.



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

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it 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;
}
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

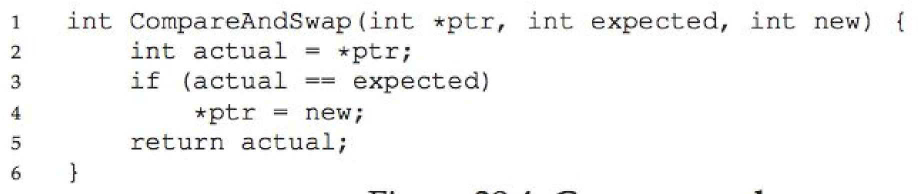
-
- When the thread calls TestAndSet, the routine will return the old value of flag (0), thus calling the thread, which is testing the value of the flag, will not get caught spinning in the while loop and will acquire the lock
    - atomically sets the value to 1 to indicate that the lock is now held
- 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
- By making both the **test** of the old lock value and **set of the new value** a single atomic operation, we ensure that only one thread acquires the lock
- On a single processor machine, we would need a preemptive scheduler to use the spin lock algorithm because a thread spinning on a CPU will never relinquish the CPU
- Summary: the test-andset instruction is used to write (set) to a memory location and return its old value as a single atomic operation

28.8: Evaluating Spin Locks

- Spin locks only allow one thread to enter the critical section at a time
- **Fairness**: Spin locks don't provide any fairness guarantees because a thread may spin forever under contention -> can lead to starvation
- **Performance**: In a Single CPU case, performance overheads can be quite painful
    - ex: If the thread holding the lock is pre-empted within the critical section
    - The scheduler might then run every other thread and each tries to acquire the lock -> threads will spin for the duration of the time slice -> wastes CPU cycles
    - On multiprocessor machines, threads do not waste as many CPU cycles because each thread can spin on a different CPU -> because the critical section is short, the lock becomes available for another processor to pick up

28.9: Compare and Swap

- **Compare and swap/Compare and exchange**
- Compare value at the address specified with ptr with expected -> if they are equal, update the memory location pointed to by ptr with the new value



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

Figure 28.4: **Compare-and-swap**

-
- Returns the actual value at that memory location, so that the code calling compare-and-swap knows whether it succeeded or not
- Summary: Compares the contents of a memory location with a given value, and if they are the same, modifies the contents of the memory location with the new given value. Result of this operation returns whether or not the function performed the substitution

Compare and Swap vs Test-and-Set

- Test-and-Set modifies the contents of memory location and returns its old value
- Compare-And-Swap atomically compares the contents of a memory location to a given value, **and only if they are the same**, returns the contents of that memory location
- MAIN DIFFERENCE: Returns only if they are the same vs anytime

28.12: Too Much Spinning: What Now?

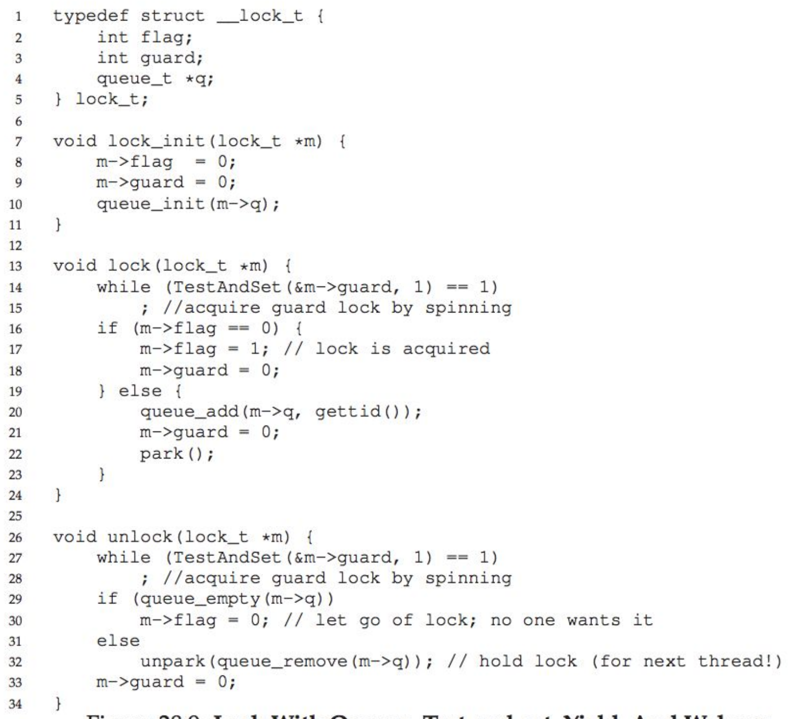
- Simple hardware-based locks can be quite inefficient
- Example: Thread 0 and thread 1 with thread 0 in the critical section
    - Thread 0 gets interrupted, and the second thread tries to acquire the lock. However, it finds that thread 0 is still holding the lock so it spins until a timer interrupt goes off and thread 0 is run again -> wastes CPU cycles
- Problem gets worse with more threads contending for one lock

28.13: Just Yield, Baby

- What to do when a context switch occurs in a critical section (through interruption/preemption) and starts to spin endlessly
- One simple approach: When the thread starts to spin, just yield the CPU to another thread so that they can try
    - thread can be in running, ready, or blocked state
    - yield simply moves the thread from a running state to a ready state -> deschedules
- 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
- If there are many threads contending for a single lock -> if one thread out of 100 acquires the lock and is preempted the other 99 will execute this run-and-yield pattern before the threading holding the lock gets to run again -> cost of the context switch is costly

28.14: Using Queues: Sleeping Instead of Spinning

- The scheduler determines which thread runs next -> if the scheduler makes a bad choice, then the thread runs that must either spin waiting for the lock or yield the CPU immediately
- Explicitly insert control over which thread gets control next + keep track of a queue that stores all of the threads waiting



```
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;
}
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

-
- when a thread cannot acquire the lock because it is already held, we are careful to add ourselves to a queue function to get the thread ID of the current thread, set guard to 0, and yield the CPU

28.15

- Linux provides a futex which is similar to the Solaris interface but provides more in-kernel functionality
- Each futex has associated with it a specific memory location, as well as a per-futex-in-kernel queue
    - Callers can use futexes to wake/sleep threads

Chapter 30: Condition Variables

- There are many cases where a thread wishes to check whether a **condition** is true before continuing execution
- Ex: parent thread may wish to check whether a child thread has completed before continuing -> join()

Chapter 30.1: Definition and Routines

- To wait for a condition to become true, a thread can make use of a **condition variable**, an explicit queue that threads can put themselves on when some state of execution is not as desired
    - When the state/condition changes, a signal is sent to this queue that wakes one or more threads up
- Wait() call is executed when a thread wishes to push itself to sleep
- Signal() is called when a thread has changed something and thus wants to wake a sleeping thread waiting on this condition
- wait() assumes that the mutex is locked -> responsibility is to release the lock and put the calling thread to sleep (atomically)
- When a thread wakes up, it must re-acquire the lock before returning to the caller
- Tip: Always hold the lock while signaling: Although it is strictly not necessary in all cases, it is likely simplest and best to hold the lock while signaling when using condition variables
    - Holding the lock while calling wait is mandated by the semantics of wait because wait always assumes the lock is held when you all it
    - Also releases lock when putting the caller to sleep
    - Also re-acquires the lock just before returning

Lecture Slides

Problem: Asynchronous Completion

- most procedure calls are synchronous
    - we call them, they do their job, they return
    - when the call returns, the result is ready
- many operations cannot happen immediately
    - waiting for a **held lock to be released**
    - waiting for an **I/O operation to complete**
    - waiting for a **response to a network request**
    - delaying execution for a fixed period of time
- we call such operations asynchronous

Approaches to waiting

- Spinning … “busy waiting”
    - works well if event is independent and prompt
    - wasted CPU, memory, bus bandwidth
    - may actually delay the desired event
- yield and spin
    - allows other processes access to CPU
    - wasted process dispatches
    - works very poorly with multiple waiters
- Either still requires mutual exclusion

Condition Variables

- Create a synchronization object
    - associate that object with a resource or request
    - requester blocks awaiting event on that object
    - upon completion, the event is “posted”
    - posting event to object unblocks the waiter