Lecture 8

Locks Mechanism

Previously

- A **multi-threaded** process has more than one point of execution.
- More than one threads want to access the share variable at the same time.
- Race condition occurs. Critical sections need to define. Off cause, guarantee mutual exclusion is our goal.
- Introduce Lock/Unlock function
  - Programmers annotate source code with locks
  - Putting them around critical sections to ensure atomicity

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks

- Mechanism
  - acquired, i.e. exactly one thread holds the lock
  - available (free), i.e. no thread holds the lock
- A lock variable holds the state of the lock at any instant in time
- The name that the POSIX library uses for a lock is a mutex, as it is used to provide mutual exclusion between threads
- Next … how should we build a lock?
  - Can we build a purely software lock?
  - What hardware support is needed?
  - What OS support?

Evaluating Lock Implementation

- **Mutual Exclusion**: the lock should prevent multiple threads from executing a critical section at the same time
- **Absence of Deadlock**: if two or more threads are trying to enter their critical sections, at least one will succeed
- **Fairness**: each thread trying to acquire the lock should get a fair chance at acquiring it
- **Performance**: the time overheads added by using the lock
  - Overhead of acquiring lock when if no other thread wants to free it
  - Overhead of acquiring lock when there is contention for the lock

```
static volatile int counter = 0;

void *mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t T0, T1;
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&T0, NULL, mythread, "A");
    pthread_create(&T1, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(T0, NULL);
    pthread_join(T1, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Recall The Code From Last Week

Review Uncontrolled Scheduling

- Accessing shared data needs careful control because of the possible interleaving of threads
  - Two threads each increment a shared variable, **counter**

Thread 1	**Thread 2**
load counter	load counter
add 1	add 1
store counter	store counter

Many possible interleavings, including:

load counter	
add 1	
	load counter
store counter	
	add 1
	store counter

Incorrect answer !!

Controlling Interrupts

- This was an early approach to implementing locks
  - The OS ensures code inside the critical section is not interrupted, and thus executes as if it were atomic
  - `void lock() { disableInterrupts(); }`
  - `void unlock() { enableInterrupts(); }`
- This is a simple approach but has many disadvantages
  - It allows any thread to perform a privileged operation (turning interrupts on and off) which is very risky – the thread may not enable interrupts again
  - The approach does not work on multi-core or multiprocessors

Software Test and Set

- A purely software attempt at implementing a lock

```
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 flag and spin-wait
    mutex->flag = 1;// now SET it!
}

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

What about Correctness and Performance ???

Note Area for Online Teaching

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

```
void lock(lock_t *mutex) {
    while (mutex->flag == 1);
    mutex->flag = 1;
}
```

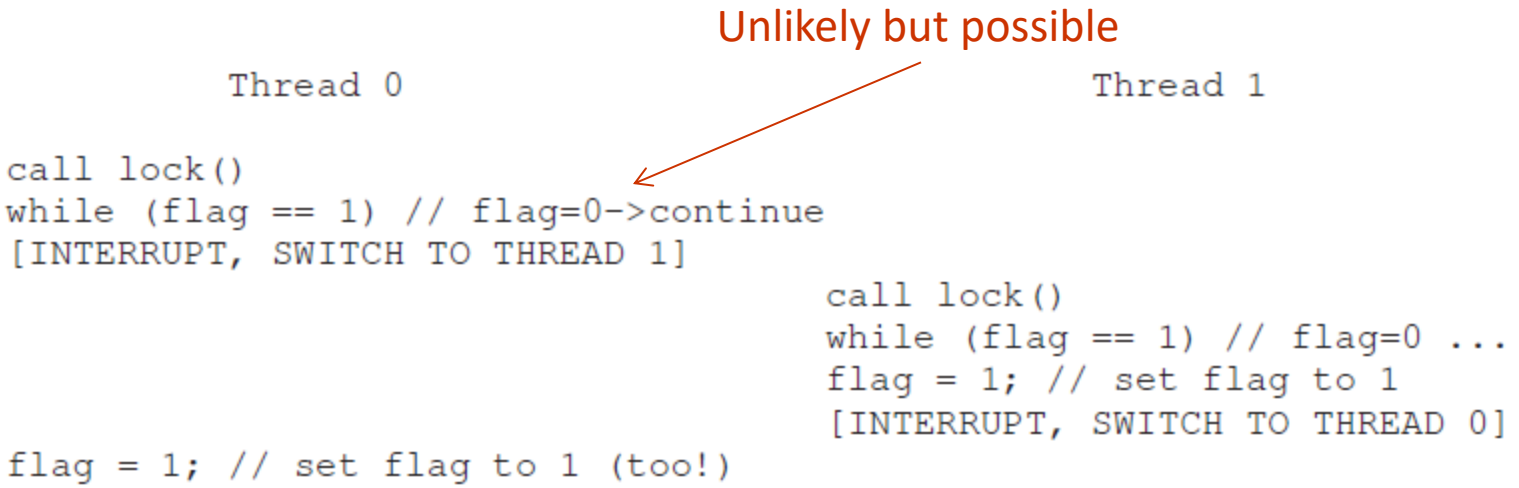
```
void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

```
lock_t mu;
…
init(&mu);
…
lock(&mu);
counter=counter+1;
unlock(&mu);
…
```

What about Performance

- Spin-waiting wastes time waiting for another thread to release the lock
- On a single processor, the thread that holds the lock cannot even run (at least, not until a context switch occurs)!
- Thus, as we move forward and develop more sophisticated solutions, we should also consider ways to avoid this kind of waste

What about Correctness



- Both threads can enter the critical section at the same time – no mutual exclusion

Hardware Support for Locking

- Although there are purely software solutions to locking (e.g. Peterson's algorithm), some hardware support can provide a simpler and better solution
- The hardware support generally takes the form of a primitive instruction that is guaranteed by the hardware to execute atomically
- Examples provided on different architectures include:
  - Test-and-Set
  - Compare-and-Swap
  - Fetch-and-Add
  - Load-Linked and Store-Conditional
  - etc.

Test-and-Set (Hardware Support for Locking)

- A Test-and-Set instruction is added to support the creation of locks based on spin-wait
- It returns the old value in the location pointed to by ptr, and simultaneously updates the location to the new value
- The key is that this sequence of operations is performed atomically

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

Next … how to use the Test-and-Set instruction to build a lock

```
int TestAndSet(int *ptr, int new)
    int old = *ptr; // fetch old
    *ptr = new;     // store 'new
    return old;     // return the
}
```

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

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

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1); // spin-lock
}

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

Test-and-set is an atomic instruction

This is called "spin lock"
spins using CPU cycles, until the lock becomes available.

- As long as the lock is held by another thread, TestAndSet() will repeatedly return 1
- By making both the ***test*** (of the old value of the lock) and ***set*** (of the new value) a single atomic operation, we can ensure that only one thread acquires the lock

Evaluating Test-and-Set

- Correctness?
  - Mutual Exclusion is guaranteed and Deadlock cannot occur
- Performance?
  - On a single processor, performance overheads can be quite painful
  - On multi-core or multiprocessor, spin locks work reasonably well
- Fairness?
  - Cannot guarantee that a waiting thread will ever enter the critical section
  - A thread spinning may spin forever….starvation

Spinning to wait for a lock held on another processor doesn't waste many cycles

Compare-and-Swap (1)

- Another hardware primitive is the Compare-and-Swap instruction
- It tests whether the value at the location pointed to by *ptr* is equal to an expected value
  - If so, it updates the memory location pointed to by *ptr* with the new value
  - If not, it does nothing
- Again this sequence of operations is performed atomically

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

Compare-and-Swap (2)

- In both cases the actual value at the memory location is returned allowing the code calling Compare-and-Swap to know whether it succeeded or not
- A lock can be built in a similar way to that with Test-and-Set

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

- The rest of the code is the same as the Test-and-Set example

Compare-and-Swap (3)

- Compare-and-swap is a more powerful instruction than Test-and-Set
- Test-and-Set will return the old value and set the new value always
- Compare-and-Swap will check if the old value is equal to the expected value and only set the value to the new one if it is.
- For implementing a lock, they are the same, but Compare-and-Swap may be used for some other purposes.

Fetch-and-Add (1)

- Fetch-and-Add atomically increments a value at the location pointed to by ptr, while returning the old value at that location

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

- Fetch-and-Add can be used to build a more interesting ticket lock
- 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

Fetch-and-Add (2)

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
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) {
    FetchAndAdd(&lock->turn);
}
```

Get a queue ticket

Wait for a turn

Now, it's somebody else turn

Note Area for Online Teaching

```
void init(lock_t *mu) {
    mu->ticket = 0;
    mu->turn = 0;
}
```

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

```
void unlock(lock_t *mutex) {
    FetchAndAdd(&mu->turn);
}
```

```
lock_t mu;
…
init(&mu);
…
lock(&mu);
counter=counter+1;
unlock*(&mu);
…
```

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Fetch-and-Add (3)

- 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
- This method ensures progress for all threads
- Once a thread is assigned its ticket value, it will be scheduled at some point in the future
- In previous methods, no such guarantee existed

So Much Spinning

- Hardware-based locks are simple and they work, but … they are inefficient with so much spinning
- Threads can waste an entire time slice doing nothing but checking a value
  - If a context switch occurs in a critical section, the value will never change until another context switch occurs
- With N threads contending for a lock, N−1 time slices may be wasted
- How can we develop a lock that does not needlessly waste time spinning on the CPU?
- Hardware support alone cannot solve the problem – we'll need OS support too!

Yield (1)

- When a thread is going to spin, instead it gives up the CPU to another thread
- An operating system primitive yield() can be called when a thread wants to give up the CPU
- Yield ()
  - Moves the caller from the running state to the ready state
  - And thus allows another thread to run

```
void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}
```

Yield (2)

- What if there are many threads contending for a lock repeatedly?
- If one thread acquires the lock and is preempted before releasing it, the other N–1 threads will each call lock(), find the lock held, and yield the CPU
- Each of the N–1 threads will execute this run-and-yield pattern until the thread holding the lock gets to run again
- There are still plenty of waste cycles
- And … the starvation problem still exists with some cases

Using Queues: Sleep instead of Spin

- Explicitly exert some control over who gets to acquire the lock next
- To do this, we need a little more OS support, as well as a queue (keep track of which threads are waiting to enter the lock)
- For simplicity, Solaris's support is considered
  - park() to put a calling thread to sleep
  - unpark(threadID) to wake a particular
- Put a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free

Basic Idea

- Lock: A thread checks a lock
  - If free, takes the lock and enters the critical section
  - If not, sleeps in a queue – Someone else will give it the lock later on
- Unlock: A Thread unlocks the critical section
  - If the waiting queue is empty, releases the lock
  - If not, pass the lock to the next thread in the queue
  - The next thread will be woken up with the lock and can proceed into the critical section

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

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

void lock(lock_t *m) {
    [redacted]
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        [redacted]
    } else {
        queue_add(m->q, gettid());
        [redacted]
        park();
    }
}

void unlock(lock_t *m) {
    [redacted]
    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!)
    [redacted]
}
```

Enter the queue

sleep

Why the 'guard' ?

How about Starvation ?

Queue help control who get the lock next
….avoid starvation



This has to be a mini critical section

But a much smaller section in comparison to user-defined critical Section … less spinning time

This has to be a mini critical section

```
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();
    }
}
```

A thread will be about to park, assuming that it should sleep until the lock is no longer held

What if there is thread-switch at that time to another thread?

If switch to the thread holding the lock …

~~If that thread then releases the lock (instead of passing the lock to a thread in the queue~~

The subsequent park by the first thread would potentially sleep forever

This scenario has a lot of 'if', but it certainly is possible that the race condition occurs – 'waiting race'

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 the thread happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping

Modify the 'lock' →

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
park();
```

- A different solution could pass the guard into the kernel
  - the kernel could take precautions to atomically release the lock
  - dequeue the running thread

Different OS, Different Support

- Linux provides something called a **futex** which is similar to the Solaris interface but provides a bit more in-kernel functionality
- Two-phase lock: Spinning can be useful if not for too long
  - In the first phase, the lock spins for a while, hoping that it can acquire the lock
  - If the lock is not acquired during the first spin phase, a second phase is entered, where the caller is put to sleep, and only woken up when the lock becomes free later

A Lock Needs …

- Some hardware support (in the form of a more powerful instruction)
- Some operating system support
  - park() and unpark() primitives on Solaris
  - futex on Linux
- The details differ for each OS
- The exact code to perform such locking is usually highly tuned