Synchronization

CS61, Lecture 18
Prof. Stephen Chong
November 3, 2011
Announcements

• Assignment 5
  • Tell us your group by Sunday Nov 6
  • Due Thursday Nov 17

• Talks of interest in next two days
  • “Towards Predictable, Heisenbug-Free Parallel Software Environments”
    • Bryan Ford, Yale
    • Thursday 4pm MD G-125
  • “Engineering Storage for the Data Age”
    • Friday 10am MD 319
    • Steve Swanson, UCSD
Topics for today

• Why to synchronize multiple threads
• Race conditions
  
  Concurrent access to shared resource without synch.
• Mutual exclusion and critical sections
  
  A way to prevent races.
• Locks
  
  A simple mechanism to synchronize threads.
• Efficiently implementing locks
Interleaved Execution

- The execution of the two threads can be **interleaved**
  - Assume **preemptive scheduling**
    - i.e., Thread may be context switched arbitrarily, without cooperation from the thread
  - Each thread may context switch after **each** assembly instruction
    (or, in some cases, part of an assembly instruction!)
  - We need to worry about the worst-case scenario!

```
balance = get_balance(account);
balance -= amount;  // local balance = $1400
balance = get_balance(account);
balance -= amount;  // local balance = $1400
put_balance(account, balance);
put_balance(account, balance);
```

- What's the account balance after this sequence?
  - And who's happier, the bank or you???
Little white lie...

• Sleeping does not help!
• Earlier I showed some examples to highlight which locations were shared between threads

```
int i = 0; // global variable
void bar() {
    i++;
    sleep(1);
    printf("i is %d\n", i);
}
```

```
int i = 0; // global variable
void bar() {
    i++;
    sleep(1);
    printf("i is %d\n", i);
}
```

• Possible outputs: 1 2, 1 2, 2 2, 2 2
• All are possible, not all equally likely.
It’s gets worse...

- Most programmers assume that memory is **sequentially consistent**
  - state of memory is due to some interleaving of threads, with instructions in each thread executed in order
  - E.g., Given $\begin{bmatrix} A \\ B \\ C \end{bmatrix}$ and $\begin{bmatrix} D \\ E \end{bmatrix}$, memory is result of some ordering such as

```
A B C
D E
```

- This is not true in most systems!
Example

• Suppose we have two threads
  • (x and y are global, a and b are thread-local, all variables initially 0)

  ```c
  x=1;
y=2;
  ```
  ```c
  a = y;
b = x;
printf("%d", a+b);
  ```

• What are the possible outputs?
  • 0  a=y  b=x  x=1  y=2
  • 1  a=y  x=1  b=x  y=2  and others
  • 3  x=1  y=2  a=y  b=x  and others
  • 2  Requires a=2 and b=0. Is possible, but no such order!
What the ...?

- What’s going on?
- Several things, including:
  - With multiple processors, multiple caches
    - A cache may not write values from cache to memory in same order as updates
    - Processor may have cache hits for some locations and not others
  - Compiler optimizations
    - Compiler may change order of instructions
Relaxed memory models

• A **model** of how memory behaves provides
  • 1) programmers with a way to think about memory
  • 2) compiler writers with limits on what optimizations they can do
  • 3) hardware designers with limits on what optimizations they can do
• Relaxed memory models provide a weaker model than sequential consistency
  • Can be complicated!
Race Conditions

• The problem: concurrent threads accessing a shared resource without any synchronization
  • This is called a race condition
  • The result of the concurrent access is non-deterministic, depends on
    • Timing
    • When context switches occurred
    • Which thread ran at which context switch
    • What the threads were doing

• A solution: mechanisms for controlling concurrent access to shared resources
  • Allows us to reason about the operation of programs
  • We want to re-introduce some determinism into the execution of multiple threads
Race conditions in real life

• Race conditions are bugs, and difficult to detect

• Northeast Blackout of 2003
  • About 55 million people in North America affected
  • Race condition in monitoring code in part responsible: alarm system failed
  • Code had been running since 1990, over 3 million hours of operation, without manifesting bug
Race conditions in real life

• Race conditions are bugs, and difficult to detect

• Therac-25 radiation therapy machine
  • Designed to give non-lethal doses of radiation to cancer patients
  • Race conditions contributed to incorrect lethal doses
  • Several fatalities in mid-80s.

Figure 1. Typical Therac-25 facility
Which resources are shared?

• Local variables in a function are not shared
  • They exist on the stack, and each thread has its own stack
  • Cannot safely pass a pointer from a local variable to another thread
    • Why?

• Global variables are shared
  • Stored in static data portion of the address space
  • Accessible by any thread

• Dynamically-allocated data is shared
  • Stored in the heap, accessible by any thread
Topics for today

• Why to synchronize multiple threads
• Race conditions
  Concurrent access to shared resource without synch.
• Mutual exclusion and critical sections
  A way to to prevent races.
• Locks
  A simple mechanism to synchronize threads.
• Efficiently implementing locks
We want to use **mutual exclusion** to synchronize access to shared resources

- Mutual exclusion: only one thread can access a shared resource at a time.

**Code that uses mutual exclusion to synchronize its execution is called a critical section**

- Only one thread at a time can execute code in the critical section
- All other threads are forced to wait on entry
- When one thread leaves the critical section, another can enter

```
Thread 1
(modify account balance)
```

```
Critical Section
```
Mutual Exclusion

- We want to use **mutual exclusion** to synchronize access to shared resources
  - Mutual exclusion: only one thread can access a shared resource at a time.
- Code that uses mutual exclusion to synchronize its execution is called a **critical section**
  - Only one thread at a time can execute code in the critical section
  - All other threads are forced to wait on entry
  - When one thread leaves the critical section, another can enter

```
Critical Section

Thread 1
(modify account balance)

Thread 2
Thread 2 must wait for critical section to clear
```

Thread 1 enters critical section
We want to use **mutual exclusion** to synchronize access to shared resources.

- Mutual exclusion: only one thread can access a shared resource at a time.

Code that uses mutual exclusion to synchronize its execution is called a **critical section**

- Only one thread at a time can execute code in the critical section
- All other threads are forced to wait on entry
- When one thread leaves the critical section, another can enter
Critical Section Requirements

• Mutual exclusion
  • At most one thread is currently executing in the critical section

• Progress
  • If thread T1 is outside the critical section, then T1 cannot prevent T2 from entering the critical section

• Bounded waiting (no starvation)
  • If thread T1 is waiting on the critical section, then T1 will eventually enter the critical section
    • Requires threads eventually leave critical sections

• Performance
  • The overhead of entering and exiting the critical section is small with respect to the work being done within it
Topics for today

• Why to synchronize multiple threads
• Race conditions
  *Concurrent access to shared resource without synch.*
• Mutual exclusion and critical sections
  *A way to prevent races.*
• Locks
  *A simple mechanism to synchronize threads.*
• Efficiently implementing locks
Locks

• A lock is an object (in memory) that provides two operations:
  • `acquire()`: a thread calls this before entering a critical section
    • May require waiting to enter the critical section
  • `release()`: a thread calls this after leaving a critical section
    • Allows another thread to enter the critical section

• A call to `acquire()` must have corresponding call to `release()`
  • Between `acquire()` and `release()`, the thread holds the lock
  • `acquire()` does not return until the caller holds the lock
    • At most one thread can hold a lock at a time (usually!)
    • We'll talk about the exceptions later...

• What can happen if `acquire()` and `release()` calls are not paired?
Using Locks

Why is the `return` statement outside of the critical section?
Execution with Locks

Execution sequence as seen by CPU

Thread 1 runs
acquire(lock);
balance = get_balance(account);
balance -= amount;

Thread 2 waits on lock
acquire(lock);

Thread 1 completes
Thread 2 resumes
put_balance(account, balance);
release(lock);

balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
Spinlocks

• Very simple way to implement a lock:

```c
struct lock {
    int held = 0;
}
void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}
void release(lock) {
    lock->held = 0;
}
```

The caller **busy waits** for the lock to be released

Why doesn't this work?
Implementing Spinlocks

- Problem: internals of the lock acquire/release have critical sections too!

```c
struct lock {
    int held = 0;
};
void acquire(lock) {
    while (lock->held) ;
    lock->held = 1;
}
void release(lock) {
    lock->held = 0;
}
```

- The `acquire()` and `release()` actions must be **atomic**
- Atomic means that the code cannot be interrupted during execution
  - “All or nothing” execution

What can happen if there is a context switch here?
Implementing Spinlocks

- Problem: internals of the lock acquire/release have critical sections too!

```c
struct lock {
    int held = 0;
};

void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}

void release(lock) {
    lock->held = 0;
}
```

- The `acquire()` and `release()` actions must be **atomic**
- Atomic means that the code cannot be interrupted during execution
  - “All or nothing” execution
Implementing Spinlocks

• Achieving atomicity requires hardware support
  • Disabling interrupts
    • Prevent context switches from occurring
    • Only works on uniprocessors. Why?
  • Atomic instructions – CPU guarantees entire action will execute atomically
    • Test-and-set
    • Compare-and-swap
Spinlocks using test-and-set

- CPU provides the following as one atomic instruction:

```
bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- So to fix our broken spinlocks, we do this:

```
struct lock {
    int held = 0;
}
void acquire(lock) {
    while(test_and_set(&lock->held));
}
void release(lock) {
    lock->held = 0;
}
```
What's wrong with spinlocks?

- So spinlocks work (if you implement them correctly), and are simple.
- What's the catch?

```c
struct lock {
    int held = 0;
};
void acquire(lock) {
    while(test_and_set(&lock->held));
}
void release(lock) {
    lock->held = 0;
}
```
Problems with spinlocks

- Inefficient!
  - Threads waiting to acquire locks spin on the CPU
  - Eats up lots of cycles, slows down progress of other threads
    - Note that other threads can still run ... how?
  - What happens if you have a lot of threads trying to acquire the lock?
- Usually, spinlocks are only used as **primitives** to build higher-level, more efficient, synchronization constructs
Efficiently implementing locks

- Really want a thread waiting to enter a critical section to **block**
  - Put the thread to sleep until it can enter the critical section
  - Frees up the CPU for other threads to run

1) Check lock state

Thread 1

Lock state: unlocked

Lock wait queue
Efficiently implementing locks

- Really want a thread waiting to enter a critical section to **block**
  - Put the thread to sleep until it can enter the critical section
  - Frees up the CPU for other threads to run

1) Check lock state
2) Set state to locked
3) Enter critical section
Efficiently implementing locks

- Really want a thread waiting to enter a critical section to **block**
  - Put the thread to sleep until it can enter the critical section
  - Frees up the CPU for other threads to run

1) Check lock state
2) Add self to wait queue (sleep)
Efficiently implementing locks

- Really want a thread waiting to enter a critical section to **block**
  - Put the thread to sleep until it can enter the critical section
  - Frees up the CPU for other threads to run

1) Check lock state
2) Add self to wait queue (sleep)

**Thread 1**

**Thread 2**

**Locked**

**Lock state**

**Lock wait queue**
Efficiently implementing locks

- Really want a thread waiting to enter a critical section to **block**
  - Put the thread to sleep until it can enter the critical section
  - Frees up the CPU for other threads to run

1) Check lock state
2) Add self to wait queue (sleep)
Efficiently implementing locks

• Really want a thread waiting to enter a critical section to **block**
  • Put the thread to sleep until it can enter the critical section
  • Frees up the CPU for other threads to run

1) Check lock state
2) Add self to wait queue (sleep)
Efficiently implementing locks

- Really want a thread waiting to enter a critical section to **block**
  - Put the thread to sleep until it can enter the critical section
  - Frees up the CPU for other threads to run

1) Thread 1 finishes critical section
Efficiently implementing locks

- Really want a thread waiting to enter a critical section to **block**
  - Put the thread to sleep until it can enter the critical section
  - Frees up the CPU for other threads to run

A blocked thread can now acquire lock
Efficiently implementing locks

- Really want a thread waiting to enter a critical section to **block**
  - Put the thread to sleep until it can enter the critical section
  - Frees up the CPU for other threads to run

A blocked thread can now acquire lock
No guarantee on which blocked thread will get the lock!!!
Locks in PThreads

- Pthreads provides a `pthread_mutex_t` to represent a lock for mutual exclusion, a `mutex`.
  - Threads using the mutex must have access to the `pthread_mutex_t` object.
  - Usually, this means declaring it as a global variable.

```c
pthread_mutex_t myLock; /* Must be global so all
* threads using the lock
* can access this variable. */

/* Initialize it. */
/* Only one thread has to do this. */
pthread_mutex_init(&myLock, NULL);

void *mythread(void *arg) {
    /* Do something with the lock */
    pthread_mutex_lock(&myLock);

    /* Do stuff... */
    pthread_mutex_unlock(&myLock);
}
```
Lock granularity

• Locks are great, and simple, but have limitations
• What if you have a more complex resource than a single location?

• **Coarse-grained lock:** Could use one lock to protect all resources
  • E.g., Many bank accounts, use one lock to protect access to all accounts

• **Fine-grained lock:** Protect each resource with a separate lock
  • E.g., Many bank accounts, one lock per account

• Coarse vs. fine-grained?
  • More locks → harder to manage locks
    • E.g., transfer money from account A to account B at same time as transferring from B to A. What order to acquire locks?
    • More on this next week...
  • Fewer locks → less concurrency
Next Lecture

• Higher level synchronization primitives: How do to fancier stuff than just locks
• Semaphores, monitors, and condition variables
  • Implemented using basic locks as a primitive
• Allow applications to perform more complicated coordination schemes