CSE 153
Design of Operating Systems

Fall 2017

Lecture 5/6: Synchronization (1)
Administrivia

- Clarification:
  - Homework 1 IS graded;
  - Homework 2 (to be assigned next week) IS NOT

- Hopefully you have started on Lab 1 by now?
  - Be prepared with questions for this week's Lab
  - Read up on scheduling in textbook and start early!
  - Need to read and understand a lot of code before starting to code
Cooperation between Threads

- What is the purpose of threads?
- Threads cooperate in multithreaded programs

Why?
- To share resources, access shared data structures
  » Threads accessing a memory cache in a Web server
- To coordinate their execution
  » One thread executes relative to another
Threads: Sharing Data

int num_connections = 0;

web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}

handle_request(int sock) {
    ++num_connections;
    Process request
    close(sock);
}
Threads: Cooperation

- Threads voluntarily give up the CPU with thread_yield

```c
while (1) {
    printf("ping\n");
    thread_yield();
}

while (1) {
    printf("pong\n");
    thread_yield();
}
```
Synchronization

- For correctness, we need to control this cooperation
  - Threads *interleave executions arbitrarily* and at *different rates*
  - Scheduling is not under program control

- We control cooperation using *synchronization*
  - Synchronization enables us to restrict the possible inter-leavings of thread executions
What about processes?

- Does this apply to processes too?
  - Yes!

- Processes are a little easier because they don’t share by default

- But share the OS structures and machine resources so we need to synchronize them too
  - Basically, the OS is a multi-threaded program
Shared Resources

We initially focus on coordinating access to shared resources

- **Basic problem**
  - If two concurrent threads are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior

- Over the next couple of lectures, we will look at
  - Exactly what problems occur
  - How to build mechanisms to control access to shared resources
    - Locks, mutexes, semaphores, monitors, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    - Bounded buffer, producer-consumer, etc.
A First Example

● Suppose we have to implement a function to handle withdrawals from a bank account:

```java
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

● Now suppose that you and your father share a bank account with a balance of $1000

● Then you each go to separate ATM machines and simultaneously withdraw $100 from the account
Example Continued

- We’ll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank machine:

```java
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- What’s the problem with this implementation?
  - Think about potential schedules of these two threads
Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

  Execution sequence seen by CPU

  balance = get_balance(account);
  balance = balance – amount;

  balance = get_balance(account);
  balance = balance – amount;
  put_balance(account, balance);

  put_balance(account, balance);

- What is the balance of the account now?
Shared Resources

- Problem: two threads accessed a shared resource
  - Known as a race condition (remember this buzzword!)

- Need mechanisms to control this access
  - So we can reason about how the program will operate

- Our example was updating a shared bank account

- Also necessary for synchronizing access to any shared data structure
  - Buffers, queues, lists, hash tables, etc.
When Are Resources Shared?

- Local variables?
  - Not shared: refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2

- Global variables and static objects?
  - **Shared**: in static data segment, accessible by all threads

- Dynamic objects and other heap objects?
  - **Shared**: Allocated from heap with malloc/free or new/delete
How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever

```
get_balance(account);
balance = get_balance(account);
balance = .................
balance = balance - amount;
balance = balance - amount;
p
```
What do we do about it?

- Does this problem matter in practice?

- Are there other concurrency problems?

- And, if so, how do we solve it?
  - Really difficult because behavior can be different every time

- How do we handle concurrency in real life?
Mutual Exclusion

- **Mutual exclusion** to synchronize access to shared resources
  - This allows us to have larger atomic blocks
  - What does atomic mean?

- Code that uses mutual 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
  - When a thread leaves a critical section, another can enter
  - Example: sharing an ATM with others

- **What requirements would you place on a critical section?**
Critical Section Requirements

Critical sections have the following requirements:

1) **Mutual exclusion (mutex)**
   - If one thread is in the critical section, then no other is

2) **Progress**
   - A thread in the critical section will eventually leave the critical section
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section

3) **Bounded waiting (no starvation)**
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section

4) **Performance**
   - The overhead of entering and exiting the critical section is small with respect to the work being done within it
About Requirements

There are three kinds of requirements that we'll use:

- **Safety** property: nothing bad happens
  - Mutex
- **Liveness** property: something good happens
  - Progress, Bounded Waiting
- **Performance** requirement
  - Performance

Properties hold for each run, while performance depends on all the runs:

- Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!).
Mechanisms For Building Critical Sections

- **Locks**
  - Primitive, minimal semantics, used to build others

- **Semaphores**
  - Basic, easy to get the hang of, but hard to program with

- **Monitors**
  - High-level, requires language support, operations implicit

- **Architecture help**
  - Atomic read/write
    » Can it be done?
Switch to software lock slides
Mutual Exclusion with Atomic Read/Writes: First Try

```c
int turn = 1;
while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}
while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called **alternation**

**It satisfies mutex:**

- If blue is in the critical section, then `turn == 1` and if yellow is in the critical section then `turn == 2`
- `(turn == 1) ≡ (turn != 2)`

Is there anything wrong with this solution?
Mutual Exclusion with Atomic Read/Writes: First Try

```c
int turn = 1;

while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}
```

```c
while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called **alternation**

It **satisfies mutex:**

- If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2
- (turn == 1) \(\equiv\) (turn != 2)

It **violates progress:** blue thread could go into an infinite loop outside of the critical section, which will prevent the yellow one from entering
Mutex with Atomic R/W: Peterson's Algorithm

```c
int turn = 1;
bool try1 = false, try2 = false;
```

```c
while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}
```

```c
while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```

• This satisfies all the requirements
• Here's why...
Mutex with Atomic R/W: Peterson's Algorithm

```plaintext
int turn = 1;
bool try1 = false, try2 = false;
```

while (true) {
    {¬ try1 ∧ (turn == 1 ∨ turn == 2)}
    1 try1 = true;
    { try1 ∧ (turn == 1 ∨ turn == 2)}
    2 turn = 2;
    { try1 ∧ (turn == 1 ∨ turn == 2)}
    while (try2 && turn != 1) ;
    { try1 ∧ (turn == 1 ∨ ¬ try2 ∨
      (try2 ∧ (yellow at 6 or at 7)))}
    critical section
    4 try1 = false;
    {¬ try1 ∧ (turn == 1 ∨ turn == 2)}
    outside of critical section
}

(break)
```

while (true) {
    {¬ try2 ∧ (turn == 1 ∨ turn == 2)}
    5 try2 = true;
    { try2 ∧ (turn == 1 ∨ turn == 2)}
    6 turn = 1;
    { try2 ∧ (turn == 1 ∨ turn == 2)}
    while (try1 && turn != 2) ;
    { try2 ∧ (turn == 2 ∨ ¬ try1 ∨
      (try1 ∧ (blue at 2 or at 3)))}
    critical section
    8 try2 = false;
    {¬ try2 ∧ (turn == 1 ∨ turn == 2)}
    outside of critical section
}
```

... ⇒ (turn == 1 ∧ turn == 2)
Some observations

- This stuff (software locks) is hard
  - Hard to get right
  - Hard to prove right
- It also is inefficient
  - A spin lock – waiting by checking the condition repeatedly
- Even better, software locks don’t really work
  - Compiler and hardware reorder memory references from different threads
    - Something called memory consistency model
    - Well beyond the scope of this class
- So, we need to find a different way
  - Hardware help; more in a second
Abstraction: Locks

• A lock is an object in memory providing two operations
 ◆ acquire(): before entering the critical section
 ◆ release(): after leaving a critical section

• Threads pair calls to acquire() and release()
 ◆ Between acquire()/release(), the thread holds the lock
 ◆ acquire() does not return until any previous holder releases
 ◆ What can happen if the calls are not paired?
Using Locks

Why is the “return” outside the critical section? Is this ok?

What happens when a third thread calls acquire?
Lets step back

- Software lock implementation has critical sections, too!
  - Chicken and egg -- How do we stop the recursion?
- The implementation of acquire/release must be **atomic**
  - Atomic operations execute as though it is never interrupted
- How do we make code atomic?
  - We saw some software implementations, but they have issues

- Lets get help from hardware
  - Atomic instructions (e.g., test-and-set)
  - Disable/enable interrupts (prevents context switches)
Atomic Instructions: Test-And-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to indicate available
  - Return the old value

- Hardware executes it atomically!

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

- When executing test-and-set on “flag”
  - What is **value of flag** afterwards if it was initially False? True?
  - What is the **return result** if flag was initially False? True?
Using Test-And-Set

- Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- Does it satisfy critical region requirements? (mutex, progress, bounded wait, performance?)
Still a Spinlocks

● The problem with spinlocks is that they are wasteful
  ✷ Although still useful in some cases; let's discuss advantages and disadvantages

● If a thread is spinning on a lock, then the scheduler thinks that this thread needs CPU and puts it on the ready queue

● If N threads are contending for the lock, the thread which holds the lock gets only 1/N th of the CPU
Another solution: Disabling Interrupts

- Another implementation of acquire/release is to disable interrupts:

```c
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?
On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- In a “real” system, this is only available to the kernel
  - Why?

- Disabling interrupts is insufficient on a multiprocessor
  - Back to atomic instructions
- Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives
  - Don’t want interrupts disabled between acquire and release
Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

Spinlocks:
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted
- Memory consistency model causes problems (out of scope of this class)

Disabling Interrupts:
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Higher-Level Synchronization

- Spinlocks and disabling interrupts are useful for short and simple critical sections
  - Can be wasteful otherwise
  - These primitives are “primitive” – don’t do anything besides mutual exclusion

- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section

- All synchronization requires atomicity
- So we’ll use our “atomic” locks as primitives to implement them
Implementing Locks (4)

- Block waiters, interrupts enabled in critical sections

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

void acquire (lock) {
    Disable interrupts;
    if (lock->held) {
        put current thread on lock Q;
        block current thread;
    }
    lock->held = 1;
    Enable interrupts;
}

void release (lock) {
    Disable interrupts;
    if (Q)
        remove and unblock a waiting thread;
    else
        lock->held = 0;
    Enable interrupts;
}
```

acquire(lock)  
Interrupts Disabled  
Critical section  
Interrupts Enabled  
release(lock)  
Interrupts Disabled
Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks work, but they have some drawbacks when critical sections are long
  - Spinlocks – inefficient
  - Disabling interrupts – can miss or delay important events
- Instead, we want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section
- Look at two common high-level mechanisms
  - **Semaphores**: binary (mutex) and counting
  - **Monitors**: mutexes and condition variables
- Use them to solve common synchronization problems
Semaphores

- Semaphores are an abstract data type that provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within critical section
  - Described by Dijkstra in THE system in 1968

- Semaphores are integers that support two operations:
  - `wait(semaphore)`: decrement, block until semaphore is open
    - Also P(), after the Dutch word for test, or down()
  - `signal(semaphore)`: increment, allow another thread to enter
    - Also V() after the Dutch word for increment, or up()
  - That's it! No other operations – not even just reading its value – exist

- Semaphore safety property: the semaphore value is always greater than or equal to 0
Blocking in Semaphores

- Associated with each semaphore is a queue of waiting threads/processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() 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 come in two types

- **Mutex semaphore** (or **binary semaphore**)
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section

- **Counting semaphore** (or **general semaphore**)
  - Multiple threads pass the semaphore determined by count
    - mutex has count = 1, counting has count = N
  - Represents a resource with many units available
  - or a resource allowing some unsynchronized concurrent access (e.g., reading)
Using Semaphores

- Use is similar to our locks, but semantics are different

```c
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

It is undefined which thread runs after a signal
Using Semaphores

- We’ve looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account

- We’re going to use semaphores to look at more interesting examples
  - Counting critical region
  - Ordering threads
  - Readers/Writers
  - Producer consumer with bounded buffers
  - More general examples
Example Problem(s)

- Create a critical region where up to three threads (but no more) may enter at a time
  - Exploits the counting feature of semaphores

- Order operations across two threads; thread A executes first, then thread B executes
  - Exploits the ability to initialize semaphores to different values