Lecture 5: Synchronization

Geoffrey M. Voelker

Threads cooperate in multithreaded programs
- 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 (recall ping-pong)

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 interleavings of thread executions

Discuss in terms of threads, also applies to processes
Shared Resources

- We will initially focus on coordinating access to shared resources
- **Basic problem**
  - If two concurrent threads (processes) 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
  - **Mechanisms to control access to shared resources**
    - Locks, mutexes, semaphores, monitors, condition variables, ...
  - **Patterns for coordinating accesses to shared resources**
    - Bounded buffer, producer-consumer, etc.

Classic Example

- Suppose we have to implement a function to handle withdrawals from a bank account:
  ```c
  withdraw (account, amount) {
      balance = get_balance(account);
      balance = balance – amount;
      put_balance(account, balance);
      return balance;
  }
  ```
- Now suppose that you and your significant other share a bank account with a balance of $1000.
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account.
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:

```c
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.

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

- What is the balance of the account now?
- Is the bank happy with our implementation?
Shared Resources

- The problem is that two concurrent threads (or processes) accessed a shared resource (account) without any synchronization
  - Known as a race condition (memorize this buzzword)
- We need mechanisms to control access to these shared resources in the face of concurrency
  - 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 are not shared (private)
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread’s stack
- Global variables and static objects are shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete
Mutual Exclusion

- 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
  - When a thread leaves a critical section, another can enter

Critical Section Requirements

Critical sections have the following requirements:

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

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

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
Mechanisms For Building Critical Sections

- Locks
  - Very 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
- Messages
  - Simple model of communication and synchronization based on atomic transfer of data across a channel
  - Direct application to distributed systems
  - Messages for synchronization are straightforward (once we see how the others work)

While one thread executes “withdraw”, we want some way to prevent other threads from executing in it

Locks are one way to do this

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?

Locks can spin (a spinlock) or block (a mutex)
Using Locks

withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}

- What happens when blue tries to acquire the lock?
- Why is the "return" outside the critical section? Is this ok?

Implementing Locks (1)

- How do we implement locks? Here is one attempt:

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

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

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

- This is called a spinlock because a thread spins waiting for the lock to be released
- Does this work?
Implementing Locks (2)

• No. Two independent threads may both notice that a lock has been released and thereby acquire it.

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

A context switch can occur here, causing a race condition

Implementing Locks (3)

• The problem is that the implementation of locks has critical sections, too
• How do we stop the recursion?
• The implementation of acquire/release must be atomic
  • An atomic operation is one which executes as though it could not be interrupted
  • Code that executes “all or nothing”
• How do we make them atomic?
• Need 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
cstruct lock {
    int held = 0;
}

tvoid acquire (lock) {
    while (test-and-set(&lock->held));
}

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

- When will the while return?
- What about multiprocessors?
**Problems with Spinlocks**

- The problem with spinlocks is that they are wasteful
  - If a thread is spinning on a lock, then the thread holding the lock cannot make progress
- How did the lock holder give up the CPU in the first place?
  - Lock holder calls yield or sleep
  - Involuntary context switch
- Only want to use spinlocks as primitives to build higher-level synchronization constructs

**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)
  - This is what Nachos uses as its primitive
- In a “real” system, this is only available to the kernel
  - Why? (From your homework)
  - What could user-level programs use instead?
- 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

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 only for very short and simple critical sections
  - 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

Next time...

- Same as last -- Read Section 2.3
**TODO**

- More annotations on pictures
- Change example?
- Difference between sync for logical vs. physical parallelism (uni vs. multiprocessor)
- What happens when three threads try to acquire lock?
  - Scheduling
- Picture showing use of spinlocks and interrupts just for acquire and release
- See yield after calling test-and-set (Nutt:p206)

**Administrivia**

- Homework #2
  - Due 10/19
  - The answers to almost all problems are short
  - But still require some thinking...
- Speaking of race conditions...
  - Chancellor’s 5K Run on the 27th
  - Finish before me and
  - You get 1 point added to your lecture grade
  - Or a gold star on your homework (your choice)
- Shouldn’t be hard…I don’t run that fast
- Programming Contest Tomorrow!
Semaphores

- Semaphores are data structures that also provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within CS
  - Described by Dijkstra in THE system in 1968
- Semaphores count the number of threads using a critical section (more later)
- Semaphores support two operations:
  - `wait(semaphore)`: decrement, block until semaphore is open
    » Also P() after the Dutch word for test
  - `signal(semaphore)`: increment, allow another thread to enter
    » Also V() after the Dutch word for increment

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes
- When `wait()` is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- `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
    » In other words, `signal()` has “history”
    » This history uses a counter
Using Semaphores

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

```c
struct account {
    double balance;
    semaphore S;
}:
withdraw (account, amount) {
    wait(account->S);
    tmp = account->balance;
    tmp = tmp - amount;
    account->balance = tmp;
    signal(account->S);
    return tmp;
}
```

Threads block

It is undefined which thread runs after a signal