CSE 120
Principles of Operating Systems

Spring 2023

Lecture 5: Synchronization

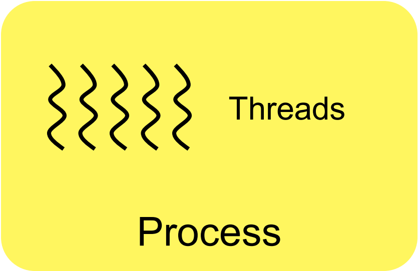
Amy Ousterhout

Administrivia

- Homework #1
  - Due 4/18 11:59 pm (today), submit via Gradescope
- Project 1
  - Due 4/25
  - Get started!
- Homework #2
  - Will be posted today

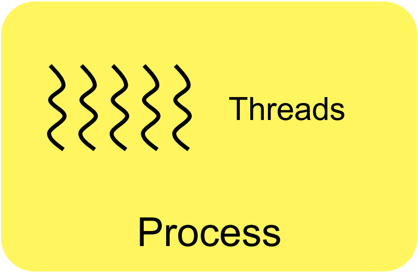
Processes and Threads

- Abstractions for resource management and execution:
  - Process: address space and resources
  - Thread: a sequential execution stream within a process (PC, SP, registers)



Concurrency

- Threads cooperate in multithreaded programs
  - To share resources, access data structures
    - » E.g., threads accessing a memory cache in a web server
  - To coordinate their execution
    - » One thread executes relative to another (recall ping-pong)
- How can different threads running concurrently safely share state?



Today's Outline

- The problem with concurrency
  - What can go wrong with concurrency?
- Synchronization
  - How can we avoid the problem?
- Locks
  - How can we implement a fix?

Bank Withdrawals – One Thread

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

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

- One independent thread:
  - Deterministic results
  - Scheduling order doesn’t matter

Bank Withdrawals – Multiple Threads

- Now suppose that you and your partner share a bank account
- Two threads running on the server:

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

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

- Suppose the shared account has $500 in it
- Then you both simultaneously withdraw $100 from the account
- What could go wrong with this implementation?
  - Hint: 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 the CPU

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

Context switch

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

Context switch

```
put_balance(account, balance);
return balance;
```

- What is the balance of the account now? $400
- Is the bank happy with our implementation?

Bank Withdrawals

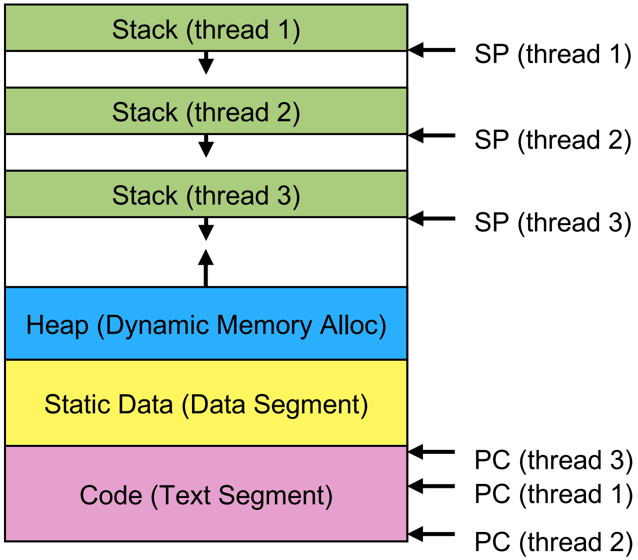
- One independent thread:
  - Deterministic results
  - Scheduling order doesn't matter
- Multiple cooperating threads:
  - Non-deterministic results
  - Scheduling order does matter

The Source of Concurrency Problems

- Problem: race conditions
  - Results depend on the timing execution of the code
- Interleaved executions
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control
- Shared resources
  - Threads can access shared resources, e.g., variables
  - What happens if they read/modify/write those variables?
  - Applies to any shared data structure
    - » Buffers, queues, lists, hash tables, etc.

Which Resources Are Shared?

- Local variables - not shared
  - Refer to data on each thread's own stack
  - Never pass/share/store a pointer to a local variable on the stack between threads
- Global variables and static objects - shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects - shared
  - Allocated from the heap with `malloc/free` or `new/delete`



How Interleaved Can It Get?

- How contorted can the interleavings be?
- We'll assume that all instructions are atomic
  - Either execute completely or not at all
  - E.g., read or write of a word
- We'll assume that a context switch can occur at any time
  - Examples may show code, but actually at instruction granularity
- We'll assume that a thread can be delayed arbitrarily long as long as it's not delayed forever

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

Today's Outline

- The problem with concurrency
  - What can go wrong with concurrency?
- Synchronization
  - How can we avoid the problem?
- Locks
  - How can we implement synchronization?

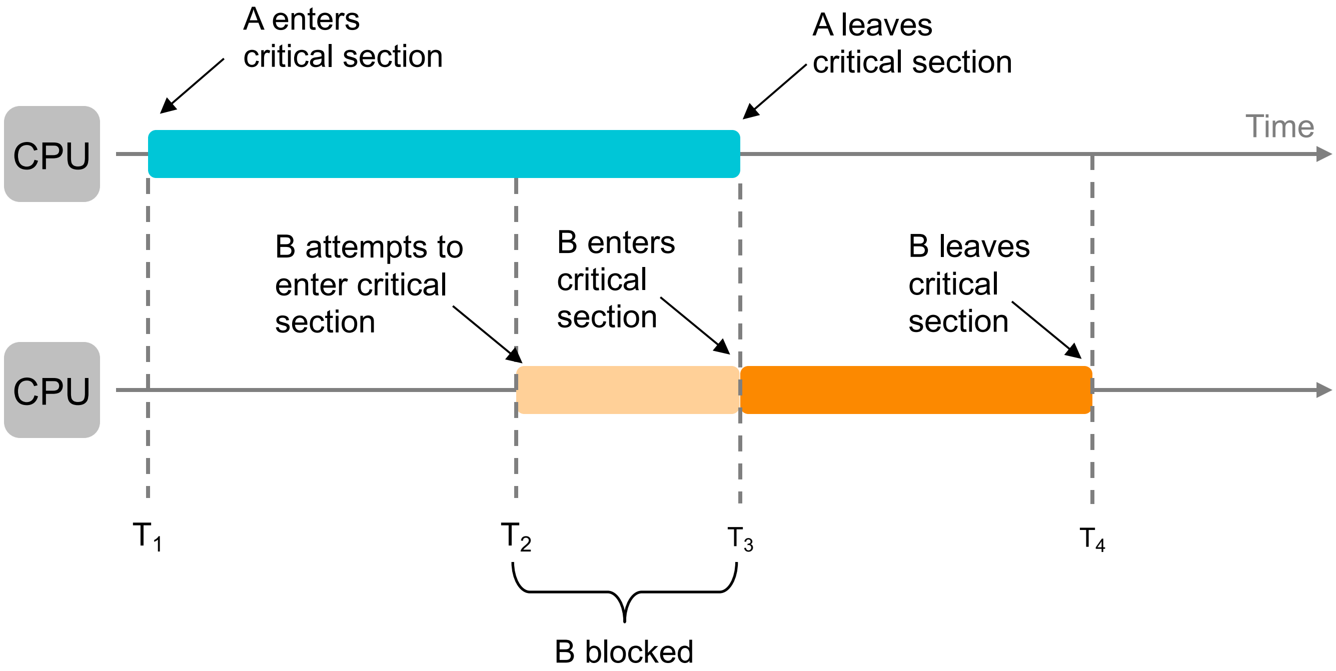
Synchronization

- Goal: restrict the possible interleavings of thread executions
- We control cooperation using synchronization
  - Synchronization ensures proper coordination among threads
- Many ways to provide synchronization:
  - Mechanisms to control access to shared resources
    - Locks, mutexes, semaphores, monitors, condition variables, etc.
  - Patterns for coordinating access to shared resources
    - Producer-consumer, reader-writer, etc.

Mutual Exclusion

- We want to create critical sections
  - Section of code in which only one thread may be executing at a given time
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: bathroom on airplanes
- We can use mutual exclusion to create critical sections
  - This allows us to have larger atomic blocks

Mutual Exclusion Using Critical Sections



Critical Section Goals

- Mutual exclusion
  - If one thread is in the critical section, then no other thread is
- Progress
  - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
  - A thread in the critical section will eventually leave it
- Bounded waiting (no starvation)
  - If some thread T is waiting on the critical section, then T will eventually enter the critical section
- Performance
  - The overhead of entering and exiting the critical section is small relative to the work being done within it

About the Goals

- Goals can also be expressed as three properties:
  - Safety property: nothing bad happens
    - Mutual exclusion
  - Liveness property: something good happens
    - Progress, bounded waiting
  - Performance property:
    - Performance
- Rule of thumb: when designing a concurrent algorithm, worry about safety first (but don’t forget liveness!)
  - Performance is nice to have but won’t affect correctness

Today's Outline

- The problem with concurrency
  - What can go wrong with concurrency?
- Synchronization
  - How can we avoid the problem?
- Locks
  - How can we implement synchronization?

Mechanisms for Building Critical Sections

- Atomic read/write
- Locks
  - Primitive, minimal semantics, used to build others
- Semaphores
  - Basic, easy to get the hang of, but harder 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

Locks

- A lock is an object in memory providing two operations
  - `acquire()` (or `lock()`): to enter a critical section
  - `release()` (or `unlock()`): to leave 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

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

critical section

Execution sequence seen by the CPU

```
acquire(lock);
balance = get_balance(account);
balance = balance – amount;
```

```
acquire(lock);
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
release(lock);
```

- What happens when orange tries to acquire the lock?
- Why is the “return” outside the critical section? Is this ok?
- What happens if a third thread calls acquire?

Implementing Locks (Attempt #1)

- How do we implement locks? Here is one attempt using a shared variable:

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

busy-wait (spin-wait) for lock to be released

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

Implementing Locks (Attempt #1)

- No, this does not work

```
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 (at `while (lock->held);`)

- Two independent threads may both notice that a lock has been released and thereby acquire it

Implementing Locks

- The problem is that the implementation of a lock 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 cannot be interrupted
  - Code that executes “all or nothing”
- How do we make them atomic?

How do we make a piece of code atomic?

- What can cause the few lines to *not* be atomic?
  - Involuntary context switches
- What causes involuntary context switches?
  - Interrupts (e.g., timer interrupts)
- Need help from the hardware
  - Disable/restore interrupts (prevents context switches)
  - Atomic instructions (e.g., test-and-set)

Implementing Locks (Attempt #2)

- Another implementation of acquire/release that disables interrupts

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

- 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., a timer)
  - This is what Nachos uses
- In a "real" system, this is only available to the kernel
  - Why?
- Disabling interrupts is insufficient on a multi-core CPU
  - Interrupts are only disabled on a per-core basis

Atomic Instructions: Test-And-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to true
  - Return the old value
- Hardware executes it atomically!
- When executing test-and-set on “flag”
  - What is the value of flag afterward if it was initially False? True?
  - What is the return result if flag was initially False? True?

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

Implementing Locks (Attempt #3)

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

```
struct lock {
    bool held = False;
}
void acquire(lock) {
    while (test_and_set(&lock->held));
}
void release(lock) {
    lock->held = False;
}
```

- When will the `while` return? What is the value of `held`?
- What about multi-core CPUs?

Problems with Spinlocks

- Spinlocks are wasteful (busy wait!)
  - If a thread is spinning on a lock, then the thread holding the lock cannot make progress (on a single-core CPU) unless:
    - Lock holder voluntarily calls yield or sleep, or
    - Involuntary context switch
- Only want to use spinlocks as primitives to build higher-level synchronization constructs

Summary of 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 (CS) 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

```
acquire(lock)
…
Critical section
…
release(lock)
```

Disabling Interrupts:

♦ Doesn't work on multicore CPUs

♦ 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” – can’t do anything besides mutual exclusion
- Need higher-level synchronization primitives that:
  - Block waiters (move them to a queue)
  - Leave interrupts enabled within the critical section
- All synchronization requires atomicity
- So we’ll use our “atomic” locks as primitives to implement synchronization

Implementing Locks (Attempt #4)

- Use a queue to block waiters, enable interrupts within critical sections

```
struct lock {
    bool held = False;
    queue Q;
}
```

```
void acquire(lock) {
    disable interrupts;
    if (lock->held) {
        put current thread on lock->Q;
        block current thread;
    }
    lock->held = True;
    enable interrupts;
}
```

```
void release(lock) {
    disable interrupts;
    if (lock->Q is empty)
        lock->held = False;
    else
        move a waiting thread to the
ready queue;
    enable interrupts;
}
```

```
acquire(lock)
…
Critical section
…
release(lock)
```

- acquire(lock): Interrupts Disabled
- Critical section: Interrupts Enabled
- release(lock): Interrupts Disabled

Implementing Locks (Attempt #5)

- Use a queue to block waiters, use a guard on the lock itself

```
struct lock {
    bool held = False;
    bool guard = False;
    queue Q;
}
```

```
void acquire(lock) {
    while (test_and_set(&lock->guard));
    if (lock->held) {
        put current thread on lock->Q;
        block current thread and
            lock->guard = False;
    }
    lock->held = True;
    lock->guard = False;
}
```

```
void release(lock) {
    while (test_and_set(&lock->guard));
    if (lock->Q is empty)
        lock->held = False;
    else
        move a waiting thread to the
ready queue;
    lock->guard = False;
}
```

```
acquire(lock)
…
Critical section
…
release(lock)
```

} Interrupts Enabled

Cornucopia of Locks

- OSes are very sensitive to the overheads of locking
  - Want to minimize overhead, optimize for the common case
- Many different kinds of locks have been invented
  - test-and-test-and-set (avoid cache, bus contention)
  - test-and-yield (allow another thread to run)
  - test-and-sleep (avoid spinning)
  - reader-writer locks (allow multiple simultaneous readers)
  - read-copy-update (optimized for reads)
  - distributed locks (avoid cache, bus contention)
  - …

Today's Outline

- The problem with concurrency
  - Interleaving while accessing shared resources can result in race conditions
- Synchronization
  - Avoid race conditions by using mutual exclusion to protect critical sections
- Locks
  - Rely on hardware (disabling interrupts, atomic instructions) to help implement locks

For next class…

- Read chapter 31
- Work on PR0