## Lecture 06: Concurrency Wrapup

## CS343 – Operating Systems Branden Ghena – Spring 2022

Some slides borrowed from: Stephen Tarzia (Northwestern), Harsha Madhyastha (Michigan), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS162

Northwestern

## Today's Goals

- Common synchronization bugs
  - Deadlock
  - Livelock
- Methods to avoid, prevent, and recover in the presence of deadlock

• Touch on what concurrency looks like in other languages

#### Outline

#### Synchronization bugs

- Deadlock
  - Solving Deadlocks
- Livelock
- Other languages

## Common synchronization bugs

- Atomicity violation
  - Critical section is violated (due to missing lock)
- Order violation
  - Something happens sooner (or later) than expected
- Deadlock
  - Two threads wait indefinitely on each other
- Livelock (not that common in practice)
  - Two threads repeatedly block each other from proceeding and retry

#### Atomicity violation

- It's relatively easy to find and protect critical sections,
  - But often we forget to add locks around other uses of the shared data.
- Obvious critical section is here:
  - Two threads should not enter this at once
- But, we also have to make sure that *file* is not modified elsewhere.
- Even if this one-line *close* was atomic we have to make sure it doesn't run during the above critical section.

```
Main Thread
lock(lck);
if (file == NULL) {
  file = open("~/myfile.txt");
}
write(file, "hello file");
unlock(lck);
...
Some Other Thread
close(file); // whoops!!
```

#### Order violation

- Code often requires a certain ordering of operations, especially:
  - Objects must be initialized before they're used
  - Objects cannot be freed while they are still in use



## Why is this difficult?

- It seems like we can just add lots of locks and CVs to be safe, right?
  - Still tricky! Too many locks can cause *deadlock* indefinite waiting.
- How about just one big lock?
  - (+) Cannot deadlock with one lock (unless there are interrupts)
  - (–) However, this would *limit concurrency* 
    - If every task requires the same lock, then unrelated tasks cannot proceed in parallel.
- $\bullet$  Concurrent code is always difficult to write  $\circledast$ 
  - Although somewhat easier with *some* higher-level languages

## Locking granularity

#### Coarse grained lock:

- Use one (or a few) locks to protect all (or large chunks of) shared state
- Linux kernel < version 2.6.39 used one "Big Kernel Lock"
- Essentially only one thread (CPU core) could run kernel code
- It's simple but there is much contention for this lock, and concurrency is limited

#### • Fine grained locks:

- Use many locks, each protecting small chunks of related shared state
- Leads to more concurrency and better performance
- However, there is greater risk of *deadlock*

#### Outline

- Synchronization bugs
- Deadlock
  - Solving Deadlocks
- Livelock
- Other languages



#### Deadlock

- A concurrency bug arising when:
  - Two threads are each waiting for the other to release a resource.
  - While waiting, the threads cannot release the resource already held.
    - Or at least *do not* release it
  - So the two threads *wait forever*.
- Can arise when *multiple* shared resources are used.
  - For example, acquiring two or more locks.

#### Deadlock versus starvation



- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- **Deadlock:** Two cars in opposite directions meet in middle
- Starvation (not deadlock): Eastbound traffic doesn't stop for westbound traffic

#### Simple example: four-way stop

- Traffic rules state that you must yield to the car on your right if you reach the intersection simultaneously.
- This rule usually works well.
- But there's a problem if four cars arrive simultaneously.



- A theoretical example of deadlock
- There are N philosophers sitting in a circle and N chopsticks
  - left and right of each philosopher
- Philosophers repeatedly run this loop:
  - 1. Think for some time
  - 2. Grab chopstick to left
  - 3. Grab chopstick to right
  - 4. Eat
  - 5. Replace chopsticks
- If they all grab the left chopstick simultaneously (step 2), they will deadlock and starve!



- A theoretical example of deadlock
- There are N philosophers sitting in a circle and N chopsticks
  - left and right of each philosopher
- Philosophers repeatedly run this loop:
  - 1. Think for some time
  - 2. Grab chopstick to left
  - 3. Grab chopstick to right
  - 4. Eat
  - 5. Replace chopsticks
- If they all grab the left chopstick simultaneously (step 2), they will deadlock and starve!
- A solution: one philosopher must grab right before left



• A solution: one philosopher must grab right before left



• A solution: one philosopher must grab right before left



- A solution: one philosopher must grab right before left
  - Adding an asymmetry will allow both resources to eventually be obtained



#### Deadlock with locks

Thread A
<pre>x.Acquire();</pre>
y.Acquire();
•••
y.Release();
<pre>x.Release();</pre>

```
Thread B
y.Acquire();
x.Acquire();
...
x.Release();
y.Release();
```

- This is a Nondeterministic Deadlock
  - Whether it occurs depends on scheduling

#### No deadlock in the lucky case

Thread A
x.Acquire();
y.Acquire();

#### Thread B

y.Acquire();

Thread B waits until Thread A is finished

```
y.Release();
x.Release();
```

...

```
x.Acquire();
...
x.Release();
y.Release();
```

#### But deadlock can still occur

Thread A
x.Acquire();

y.Acquire();

#### Thread B

y.Acquire();

x.Acquire();

Thread A waits until y is available

Thread B waits until x is available

#### --Unreachable--

y.Release();
x.Release();

...

x.Release();
y.Release();

...

#### Deadlocks involve *circular dependencies*



Deadlock can occur on any shared resource

• Example deadlock if the system only has 2 MB of memory

<u>Thread A</u>	<u>Thread B</u>
AllocateOrWait(1 MB)	AllocateOrWait(1 MB)
AllocateOrWait(1 MB)	AllocateOrWait(1 MB)
Free(1 MB)	Free(1 MB)
Free(1 MB)	Free(1 MB)

Could deadlock on access to hardware as well



release()

- Thread cannot continue until the interrupt is finished
- Interrupt cannot finish until the thread continues

#### **Reentrant library functions**

- Functions that can safely and successfully be called again while currently in the middle of its execution are called "reentrant"
  - Reentrant functions must only modify local variables and input
  - Must also never call non-reentrant functions

- Malloc is thread-safe because it uses locks around shared memory
  - Malloc is **NOT** reentrant and it will cause deadlock
  - Same goes for printf!!!
  - Must not be called in an interrupt or signal handler!
    - This matters in PCLab too

```
void List Insert(list t *L, int key) {
  pthread mutex lock(&L->lock);
  node t *new = malloc(sizeof(node t));
  if (new == NULL) {
    perror("malloc");
    pthread mutex unlock(&L->lock);
    return; // fail
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread mutex unlock(&L->lock);
  return; // success
```

Is it safe to call List\_Insert from an interrupt?

```
void List Insert(list t *L, int key) {
  pthread_mutex lock(&L->lock);
  node t *new = malloc(sizeof(node t));
  if (new == NULL) {
    perror("malloc");
    pthread mutex unlock(&L->lock);
    return; // fail
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread mutex unlock(&L->lock);
  return; // success
```

Not safe!

If another thread has acquired the mutex, there will be a deadlock

#### Outline

- Synchronization bugs
- Deadlock
  - Solving Deadlocks
- Livelock
- Other languages

How Should a System Deal With Deadlock?

- Three different approaches:
- <u>Deadlock avoidance</u>: dynamically delay resource requests so deadlock doesn't happen
- 2. <u>Deadlock prevention</u>: write your code in a way that it isn't prone to deadlock
- 3. <u>Deadlock recovery</u>: let deadlock happen, and then figure out how to recover from it

#### Deadlock avoidance

- Idea: When a thread requests a resource, OS checks if it would result in an unsafe state that could lead to deadlock
  - If not, grant the resource
  - If so, wait until other threads release resources

Thread A	Thread B	here to prevent
χ.Αεφατίε();	y.Acquire();	unsafe state
	<pre>x.Acquire();</pre>	
y.Acquire();	•••	
•••	<pre>x.Release();</pre>	
y.Release();	y.Release();	
<pre>x.Release();</pre>		

Must stan again

Banker's Algorithm for avoiding deadlock

- Each thread states maximum resource needs in advance
- OS allows a particular thread to claim a resource if
  - (available resources requested) ≥ maximum remaining that might be needed by any thread
- For Dining Philosophers, a request for a chopstick is allowed if:
  - 1. Not the last chopstick
  - 2. Or is the last chopstick but a philosopher will have two afterwards

• See the textbook for more details

How Should a System Deal With Deadlock?

- Three different approaches:
- 1. <u>Deadlock avoidance</u>: dynamically delay resource requests so deadlock doesn't happen
- 2. <u>Deadlock prevention</u>: write your code in a way that it isn't prone to deadlock
- 3. <u>Deadlock recovery</u>: let deadlock happen, and then figure out how to recover from it

#### Preventing Deadlocks: deadlock requires four conditions

#### 1. Mutual exclusion

- Threads cannot access a critical section simultaneously.
- In other words, we're using locks so there is the potential for waiting.

#### 2. Hold-and-wait

• Threads do not release locks while waiting for additional locks.

#### 3. No preemption

- Locks are always held until released by the thread.
  - E.g., if there is no method to *cancel* a lock.

#### 4. Circular wait

- Thread is waiting on a thread that is waiting on the original thread.
- This can involve just two threads or a chain of many threads.

Can eliminate deadlock by eliminating any one of these conditions

1. Do not have mutual exclusion

Lockfree/waitfree data structures

```
void* mythread(void* arg) {
  for (int i=0; i<LOOPS; i++) {
    pthread_mutex_lock(&lock);
    counter++;
    pthread_mutex_unlock(&lock);
  }
  return NULL;</pre>
```

```
void* mythread(void* arg) {
  for (int i=0; i<LOOPS; i++) {
    atomic_fetch_and_add(
       &counter, 1);
  }
  return NULL;
}</pre>
```

## Lockfree data structures

```
void insert(int val) {
  node t* n =
      malloc(sizeof(node_t));
  n->val = val;
  acquire(&m);
  n->next = head;
  head = n;
  release(&m);
```

```
void insert(int val) {
  node t* n =
      malloc(sizeof(node_t));
  n->val = val;
  do {
    n->next = head;
  } while (!cas(&head, n->next, n));
```

atomic\_compare\_and\_swap(destptr, oldval, newval)

- If \*destptr == oldval { \*destptr = newval, return True }
- Else { return false }

}

## 2. Avoid hold and wait with trylock()

- We can avoid deadlock if we release the first lock after noticing that the second lock is unavailable.
- Trylock() tries to acquire a lock, but returns a failure code instead of waiting if the lock is taken:

   top:
- This code *cannot deadlock*, even if another thread does the same with L2 first, then L1.

1	top:	
2	lock(L1);	
3	if $(trylock(L2) == -1)$	{
4	unlock(L1);	
5	goto top;	
6	}	

• However it can *livelock*... we'll come back to this

## 3. No preemption

- The OS *could* take away the lock from a blocked thread and give it back before the thread resumes
  - This sounds pretty complicated to get right
- Non-lock resources are easier here
  - Temporarily take away memory from a thread by swapping it to disk

## 4. Avoiding Circular Wait

- This is the most practical way to avoid deadlock.
- The simplest solution is to always acquire locks in the same order.
  - If you hold lock X and are waiting for lock Y,
  - Then holder of Y cannot be waiting on you,
  - Because they would have already acquired X before acquiring Y.
- However, in practice it can be difficult to know when locks will be acquired because they can be buried in subroutines.

## Ordered locking for dining philosophers



- The chopsticks are shared resources, like locks
- If we require the **lower-numbered chopstick to be grabbed first**, this eliminates circular waiting.
  - Philosophers A, B, C grab *left then right*.
  - However philosopher D will grab *right then left*.
  - If everyone tries to start at once, A & D race to grab chopstick 0 first, and the winner eats first.
  - While one is waiting to grab its first chopstick a neighbor will be able to grab two chopsticks.

#### Check your understanding

 In what order must Thread B acquire the three locks to avoid deadlock?

#### **Thread A**

y.Acquire(); x.Acquire(); z.Acquire();

- ...
  z.Release();
- x.Release();
- y.Release();

#### Thread B

<u>;</u>;;

## Check your understanding

- In what order must Thread B acquire the three locks to avoid deadlock?
  - The same order!! (at least y first, for the two-thread case)

Thread A
y.Acquire();
<pre>x.Acquire();</pre>
z.Acquire();
•••
z.Release();
<pre>x.Release();</pre>

#### **Thread B**

y.Acquire(); x.Acquire(); z.Acquire();

...
z.Release();
x.Release();

y.Release();

How Should a System Deal With Deadlock?

- Three different approaches:
- 1. <u>Deadlock avoidance</u>: dynamically delay resource requests so deadlock doesn't happen
- 2. <u>Deadlock prevention</u>: write your code in a way that it isn't prone to deadlock
- **3.** <u>**Deadlock recovery</u>:** let deadlock happen, and then figure out how to recover from it</u>

Deadlock Recovery: how to deal with a deadlock?

- Terminate thread, force it to give up resources
  - Dining Philosophers Example: Remove a dining philosopher
  - In AllocateOrWait example, OS kills a process to free up some memory
  - Not always possible—killing a thread holding a lock leaves world inconsistent
- Roll back actions of deadlocked threads
  - Common techniques in databases (transactions)
  - Of course, if you restart in exactly the same way, you may enter deadlock again

#### Modern OS approach to deadlocks

- Make sure the *system* isn't involved in any deadlock
  - Hopefully by prevention
  - Generally, be very careful about this stuff in the kernel
- Ignore deadlock in applications ("Ostrich Algorithm")
  - User can just restart them anyways

- Is there a possibility of deadlock?
  - If so, how could we fix it?

# Thread A usb.Acquire(); webcam.Acquire();

...
webcam.Release();
usb.Release();

#### Thread B

...

printer.Acquire();
usb.Acquire();

```
usb.Release();
printer.Release();
```

#### Thread C

...

webcam.Acquire();
printer.Acquire();

```
printer.Release();
webcam.Release();
```

- Is there a possibility of deadlock? Yes
  - If so, how could we fix it? One solution: Global ordering of resources
    - Example: usb, then webcams, then printers always in that order

Thread A
usb.Acquire();
webcam.Acquire();

...
webcam.Release();
usb.Release();

#### **Thread B**

...

```
printer.Acquire();
usb.Acquire();
usb.Acquire();
printer.Acquire();
```

usb.Release();
printer.Release();
usb.Release();

#### Thread C

...

webcam.Acquire();
printer.Acquire();

```
printer.Release();
webcam.Release();
```

- Is there a possibility of deadlock? Yes
  - If so, how could we fix it? One big lock still works too!

#### **Thread A**

lock.acquire(); usb.Acquire(); webcam.Acquire();

#### 

webcam.Release(); usb.Release(); lock.release();

#### **Thread B**

lock.acquire();
printer.Acquire();
usb.Acquire();

#### •••

usb.Release();
printer.Release();
lock.release();

#### **Thread C**

lock.acquire();
webcam.Acquire();
printer.Acquire();
...
printer.Release();
webcam.Release();
lock.release();

#### Outline

- Synchronization bugs
- Deadlock
  - Solving Deadlocks
- Livelock
- Other languages

## Common synchronization bugs

- Atomicity violation
  - Critical section is violated (due to missing lock)
- Order violation
  - Something happens sooner (or later) than expected
- Deadlock
  - Two threads wait indefinitely on each other
- Livelock (not that common in practice)
  - Two threads repeatedly block each other from proceeding and retry

## Livelock while avoiding deadlock

```
// thread 1
getLocks12(lock1, lock2) {
  lock1.acquire();
  while (lock2.locked()) {
    // attempt to step aside
    // for the other thread
    lock1.release();
    wait();
    lock1.acquire();
  }
  lock2.acquire();
```

```
// thread 2
getLocks21(lock1, lock2) {
  lock2.acquire();
  while (lock1.locked()) {
    // attempt to step aside
// for the other thread
    lock2.release();
    wait();
    lock2.acquire();
  ļ
  lock1.acquire();
```

#### Avoiding hold and wait could lead to livelock

- Avoiding hold and wait can *livelock* 
  - Two threads *could* get stuck in this loop forever
  - Unlikely to occur for any length in personal computing setting
  - Very possibly stuck forever (or at least extended periods) in a constrained computing setting
    - Example: embedded system with known tasks at the start

```
1 top:
2 lock(L1);
3 if (trylock(L2) == -1) {
4 unlock(L1);
5 goto top;
6 }
```

#### Livelock in agents

- Livelock is more common in agent-based programs
  - All of agent's options lead to a lack of forward progress
- One example: video games
  - The character can still move and take actions
  - But cannot complete the level



#### Livelock versus Deadlock

 Livelock is a condition where two threads repeatedly take action, but still don't make progress.

```
1 top:
2 lock(L1);
3 if (trylock(L2) == -1) {
4 unlock(L1);
5 goto top;
6 }
```

- Differs from deadlock because deadlock is always permanent.
- Livelock involves retries that *may* lead to progress, but there is no *guarantee* of progress.
  - A malicious scheduler can always keep the livelock stuck
- Any randomness in the timing of retries will fix livelock.
- In practice, livelock is a much less serious concern than deadlock.

## Helgrind tool

- Helgrind (part of the Valgrind tool) detects many common errors when using the POSIX pthreads library in C & C++, such as:
  - Race conditions (missing locks)
  - Lock ordering problems (leading to deadlock)
  - Double-unlocking
  - Freeing a locked lock
  - ... and *much*, *much* more
  - <u>http://valgrind.org/docs/manual/hg-manual.html</u>

#### Outline

- Synchronization bugs
- Deadlock
  - Solving Deadlocks
- Livelock
- Other languages

## Javascript

- Javascript (in browsers) is strictly single-threaded
  - Therefore, no data races!
- A Javascript function will never be interrupted unless it makes an asynchronous call

console.log("1");

setTimeout(function(){console.log("2");},0);

console.log("3");

setTimeout(function(){console.log("4");},1000);

- Will always output: **1 3 2 4** in that order
  - Even timers only trigger whenever the current code is finished

#### Python

- Provides all the same primitives we discussed! <u>https://docs.python.org/3/library/concurrency.html</u>
- threading Thread-based parallelism
  - Thread-Local Data
  - Thread Objects
  - Lock Objects
  - RLock Objects
  - Condition Objects
  - Semaphore Objects
    - Semaphore Example
  - Event Objects
  - Timer Objects
  - Barrier Objects

And some nicer things
with some\_lock:
 # do something...

Is equivalent to
 some\_lock.acquire()
 try:
 # do something...
 finally:
 some\_lock.release()

Python threads are concurrent but not parallel

- Python uses one big lock technique for thread safety
  - Global Interpreter Lock (GIL)
  - Threads that are I/O bound still get a performance boost
  - Threads that are CPU bound do not increase performance
- Multiprocessing library does employ parallelism by spawning entirely new processes
  - Each with their own python interpreter



https://hackernoon.com/concurrent-programming-in-python-is-not-what-you-think-it-is-b6439c3f3e6a

Java

- Java has synchronized keyword for surrounding critical sections
- Automatically releases the lock when exiting early:
- Similar to
  - Python: "with self.lock:"
  - Objective-C: "@synchronized"

```
public class Counter {
    int mTotal = 0;
    public synchronized void addOne() {
        int val = miotal:
       val++;
       mTotal = val:
    }
    public void addOneVersion2() throws Exception {
       int val:
       synchronized(this) {
           val = mTotal;
            val++;
            if (val == Integer.MAX_VALUE) {
             throw new Exception("value is too large");
           mTotal = val;
        System.out.println("new value is " + val);
```

}

• Rust's opinion on sharing memory is amusingly to refer to Go's opinion

Do not communicate by sharing memory; instead, share memory by communicating.

--<u>Effective Go</u>

- Rust has a strong concept of ownership
  - A writeable (mutable) reference to an object can only be held in one place
  - Once an object is passed to another thread, the passer no longer has access
  - Solves many concurrency issues due to lack of shared memory
- Rust locks have lifetimes enforced by the compiler
  - Lock goes out-of-scope at the end of the function, relocking automatically

#### Advice for the future

- Be aware of issues when writing multithreaded code
- Use threadsafe data structures when possible
  - In languages that provide them...
- Map your problem onto a classical concurrency problem
  - Producer/Consumer
  - Readers/Writers
- One big lock for *correctness* isn't the worst idea ever
  - But with some care (possibly a lot of care) we can do better

#### Outline

- Synchronization bugs
- Deadlock
  - Solving Deadlocks
- Livelock
- Other languages