Concurrency Bugs

Chapter 32

Previously in CS212…

- We've looked at locks, condition variables, and semaphores
- We know that using threads and introducing concurrency in our applications can cause extra complications
- We discussed deadlocks where the threads all in a perpetual state of waiting on each other for resources
  - An example was the dining philosophers
- Here we take closer look at deadlock and non-deadlocking bugs

Example

- What is wrong with this code?

```
Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

Thread 2::
thd->proc_info = NULL;
```

thd->proc_info is being modified by both threads

What happens if Thread 1 is interrupted after the if condition check?

- This is an **Atomicity Violation**

Atomicity Violation Bugs

- "The desired serializability among multiple memory accesses is violated"
  - A critical section of code intended to be atomic does not have atomicity enforced

- Caused by an "*atomicity assumption*"

How can we fix this?

```
Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

Thread 2::
thd->proc_info = NULL;
```

thd->proc_info is being modified by both threads

What happens if Thread 1 is interrupted after the if condition check?

Atomicity Violation Example Fix

- Locks around both the access and modification of thd->proc_info

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock(&proc_info_lock);
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}
pthread_mutex_unlock(&proc_info_lock);

Thread 2::
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```

Example

- What is wrong with this code?

```
Thread 1::
void init() {
    mThread = PR_CreateThread(mMain, ...);
}

Thread 2::
void mMain(...) {
     mState = mThread->State;
}
```

What happens if Thread 2 runs first?

- This is an **Order-Violation Bug**

Order-Violation Bugs

- "The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)"
- We assume instructions will take place in a specific order without a guarantee

How can we fix this?

- What is wrong with this code?

```
Thread 1::
void init() {
    mThread = PR_CreateThread(mMain, ...);
}

Thread 2::
void mMain(...) {
     mState = mThread->State;
}
```

What happens if Thread 2 runs first?

Order-Violation Example Fix

- We can fix this code by using a condition variable to ensure the proper ordering and avoid busy waiting.

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
int mtInit             = 0;

Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);

   // signal that the thread has been created...
   pthread_mutex_lock(&mtLock);
   mtInit = 1;
   pthread_cond_signal(&mtCond);
   pthread_mutex_unlock(&mtLock);
   ...
}

Thread 2::
void mMain(...) {
    ...
    // wait for the thread to be initialized...
    pthread_mutex_lock(&mtLock);
    while (mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

    mState = mThread->State;
    ...
}
```

Non-Deadlock Bugs

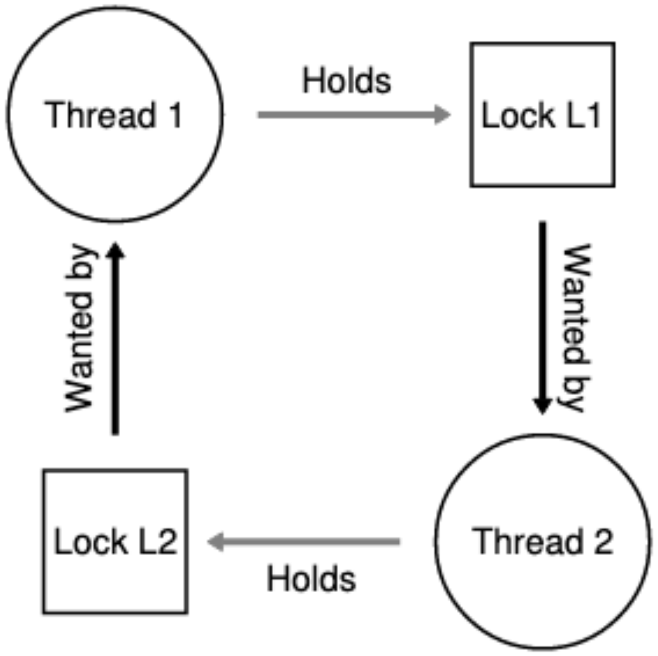
- Atomicity and Order-Violation bugs are considered non-deadlock bugs
- They don’t prevent the threads from executing code, but the way in which they allow operations to occur can yield incorrect results or crashing
- Roughly 97% of the non-deadlock bugs in the Lu et al. study were one of these bugs

Example

- What is wrong with this code?

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

What happens if Thread 1 holds L1 and is interrupted by Thread 2 who manages to hold L2?



- This is a (simple) **Deadlock**

Deadlocks

- While the previous example is obvious, these arise with complicated interactions over large code bases

- **Encapsulation** can exacerbate this issue
  - Hiding implementation behavior to make software easier to develop and modular
  - Some APIs have thread safe functions/objects, where the locking order and handling are obscured or arbitrary

Conditions for a Deadlock

- **Mutual Exclusion:** threads get exclusive control of resources (grabs a lock)
- **Hold-and-wait:** Threads hold onto resources allocated to them (locks owned) and wait for additional resources (locks needed)
- **No preemption:** Resources cannot be forcibly removed from threads holding them
- **Circular wait:** a circular chain of threads such that each thread holds one or more resources (lock) that are needed by other threads in the chain

Avoiding Circular Wait

- Ensure that locks in the system are acquired using a **strict** or **partial** ordering
- If you have a small number of locks, acquire them in the same order for each thread (strict ordering)
- If you have many locks, you can carefully create sub groups and identify the orderings to avoid dead locks (partial ordering)
- Note that this is a conversion, and not enforced, so lapses in order could create deadlock opportunities

Avoiding Hold-and-wait

- Limited options for this solution
- We can have a single lock used to protect the lock acquisition process
  - This means collecting all the locks for protected resources becomes an atomic operation
- Problem is this requires us to know all needed locks ahead of time
- It also limits the amount of concurrency as all threads will need to wait for the single lock

Avoiding No Preemption

- While not forcibly removing the lock from a thread, we could maintain a lock conditionally
- Using something like `pthread_mutex_trylock()` we could attempt to grab the resource (lock) we need, but if we can't, then we let go of the resources we currently hold
- This can work, but becomes increasingly complicated with the number of "steps" in the synchronization process
  - Needing to free acquired memory, other locks/resources, etc. to "undo" the process
- Could still cause **livelock**, where threads continuously acquire and release the locks as the cannot gain access to both without interruption

Avoiding Mutual Exclusion

- Difficult to do especially with complex operations
- Lock-free/wait free approaches can leverage atomic operations provided by instruction set and hardware to perform operations in a thread safe way

```
int CompareAndSwap(int *address, int expected, int new) {
  if (*address == expected) {
    *address = new;
    return 1; // success
  }
  return 0; // failure
}
```

```
void AtomicIncrement(int *value, int amount) {
  do {
    int old = *value;
  } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

Still need to watch out for livelock!

Deadlock Avoidance via Scheduling

- What if instead of prevention, we simply tried to avoid deadlock by detecting which locks are needed by the threads

- So instead of coding a solution, the OS and it's mighty scheduler, handles this for us

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

Higher resource utilization limits concurrency

Detect and Recover

- A scheduling approach requires very specific circumstances and full knowledge of the thread tasks…we aren’t likely to have that most times
- What if instead we let deadlocks occur…
- If we kept a graph in memory of the resources requested, we could check the graph for cycles which would indicate a deadlock
- The system can then either attempt recovery automatically, with human intervention, terminating threads holding important resources, restart the system/service, etc.

So what do we do…

- Code and develop your applications carefully
  - Establish a clear and well defined lock acquisition order (Linux)

- Follow documentation guidelines for thread safe objects/data structures

- If possible lock-free/wait-free solutions might be applicable

Next Time

- We switch gears to start talking about persistence and associated I/O devices