

Deadlock and Other Concurrency Bugs

How to handle common concurrency bugs?

By Matthew Tancreti
For COM S 352
Iowa State University

Deadlock and Other Concurrency Bugs

Concurrency bugs can be difficult to find and reproduce

It is good to be aware of common bugs and know methods for preventing or dealing with them

How to handle common concurrency bugs?



Common Non-Deadlock Bugs


We have seen race condition bug (e.g., counter++ in two threads)

Atomicity violation and **order-violation** are closely related to race condition

Thread 1 assumes `thd->proc_info` will not change between lines 2 and 3
- it assumes its operations are atomic

Atomicity violation

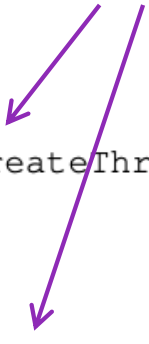
```
1 Thread 1::
2 if (thd->proc_info) {
3     fputs(thd->proc_info, ...);
4 }
5
6 Thread 2::
7 thd->proc_info = NULL;
```



The programmer expected thread 1 to execute before thread 2, there is not guarantee that will be the case

Order-violation

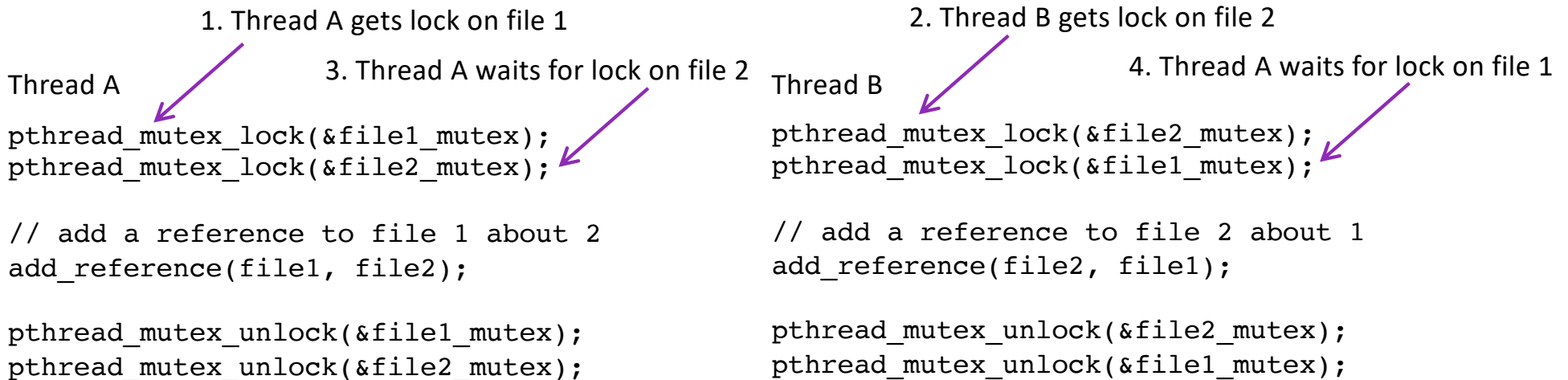
```
1 Thread 1::
2 void init() {
3     mThread = PR_CreateThread(mMain, ...);
4 }
5
6 Thread 2::
7 void mMain(...) {
8     mState = mThread->State;
9 }
```



Deadlock Example

Suppose threads A and B both require exclusive access to two files, each protected by a mutex lock

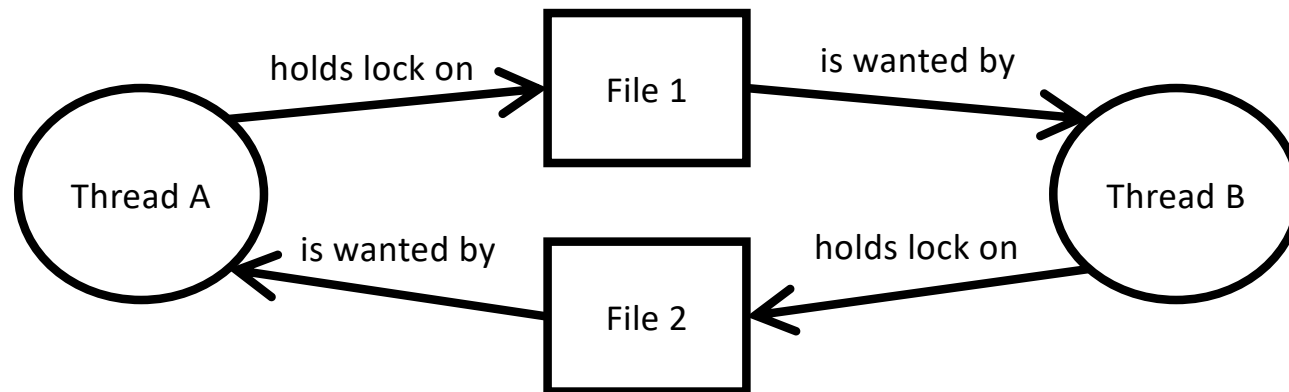
If thread A locks file1 and thread B locks file2, neither can proceed, they are deadlocked



Why deadlock happens? – Circular Wait

Why was there deadlock in the previous example?

First observation, there is a **circular wait** for resources (like in the dining philosophers problem)



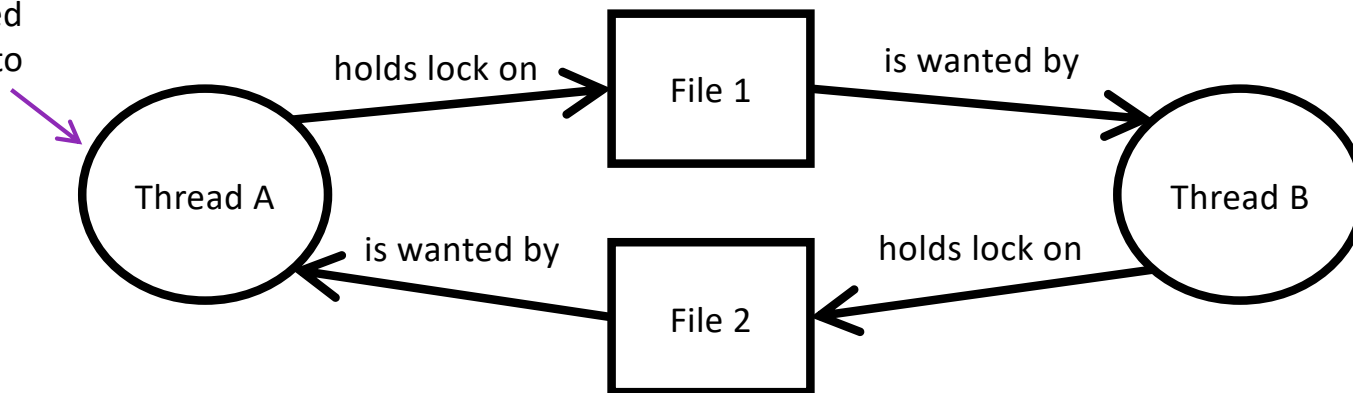
Why deadlock happens? – Mutual Exclusion

In the code both threads require **mutual exclusion** over the resources they need

If it were possible for Thread A and B to be using file 1 at the same time, there would be no deadlock

Remember that mutual exclusion is required to prevent race condition bugs

Both threads need
exclusive access to
both files



Why deadlock happens? – Hold and Wait

Deadlock is possible because both threads in the example hold some resource and then wait for another resource

```
pthread_mutex_lock(&file1_mutex);  
pthread_mutex_lock(&file2_mutex);
```

Holding a resource

Now waiting for another one, if there is
deadlock this is where it will happen

```
add_reference(file1, file2);
```

```
pthread_mutex_unlock(&file1_mutex);  
pthread_mutex_unlock(&file2_mutex);
```

Why deadlock happens? – No Preemption

Recall concept of preemption when one thread takes the CPU (a resource) from another

We can generalize preemption to any type of resource

If the OS could just force one of the threads to give up its resource, then there would be no deadlock possible

```
pthread_mutex_lock(&file1_mutex);  
pthread_mutex_lock(&file2_mutex);  
  
add_reference(file1, file2);  
  
pthread_mutex_unlock(&file1_mutex);  
pthread_mutex_unlock(&file2_mutex);
```

Holding a resource

If Thread A could steal the lock from
Thread B on file 2, the deadlock would not
be possible

The Four Conditions for Deadlock

All four conditions must hold for deadlock to be possible

- **Mutual Exclusion:** threads claim exclusive control of resources that they require (e.g., a thread grabs a lock)
- **Hold-and-wait:** threads hold resources allocated to them (*e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire)
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them
- **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain

We just need to stop one of these to prevent deadlocks

Removing Mutual Exclusion

One solution to the deadlock example would be to remove the need for mutual exclusion

If we treat data as non-mutable (read only) there is no need for mutual exclusion

An approach taken by *functional programming*, a style of programming that is popular for concurrency programs for this reason

If resources are only created, never modified, no need for mutual exclusion

Thread A

```
// add reference to file1 about file2,  
// store the result in file3  
add_referece(file1, file2, file3);
```

Thread B

```
// add reference to file2 about file1,  
// store the result in file4  
append(file2, file1, file4);
```

Removing Hold and Wait

When possible, it is a good idea not to hold multiple locks at the same time (i.e., don't hold a lock and wait for another)

Sometimes it is more efficient to have multiple resources locked at the same time

The example below shows avoiding hold and wait by adding an extra buffer

```
pthread_mutex_lock(&file2_mutex);  
read(file2, buffer);  
pthread_mutex_unlock(&file2_mutex);
```

Perfectly fine to hold a lock as long as the thread doesn't wait for another one while still holding the lock

```
pthread_mutex_lock(&file1_mutex);  
add_reference(file1, buffer);  
pthread_mutex_unlock(&file1_mutex);
```

Not holding the lock of file2, so fine to wait for lock on file 1

Adding Preemption

What if a thread can have its resource preempted? Then the deadlock would be broken

If not careful, preemption can lead to race condition, one strategy is restart the thread or process (or in extreme case reboot the machine)

Another alternative is a trylock – try to get the lock but don't block (wait) if it is not available

```
1  top:
2      pthread_mutex_lock(L1);
3      if (pthread_mutex_trylock(L2) != 0) {
4          pthread_mutex_unlock(L1);
5          goto top;
6      }
```

Preventing Circular Wait

One strategy to prevent circular wait is to enforce a **total ordering** of locks

For example, suppose all threads are required to only lock file 2 after file 1

Deadlock is no longer possible

Thread A

```
pthread_mutex_lock(&file1_mutex);  
pthread_mutex_lock(&file2_mutex);  
  
// add a reference to file 1 about 2  
add_reference(file1, file2);  
  
pthread_mutex_unlock(&file1_mutex);  
pthread_mutex_unlock(&file2_mutex);
```

Thread B

```
pthread_mutex_lock(&file1_mutex);  
pthread_mutex_lock(&file2_mutex);  
  
// add a reference to file 2 about 1  
add_reference(file2, file1);  
  
pthread_mutex_unlock(&file1_mutex);  
pthread_mutex_unlock(&file2_mutex);
```

Deadlock Avoidance and Detect and Recover

Preventing deadlocks is a difficult problem, there is no really good general solution

Alternatives to preventing deadlocks

Avoid deadlocks by modifying the scheduler, a clever scheduler might know that two threads should never be scheduled concurrently because it can result in hold and wait

Detect and recover from a dead lock, some deadlocks are so difficult to prevent or avoid that OSes use the fallback strategy of detect the deadlock and reboot the system when it happens