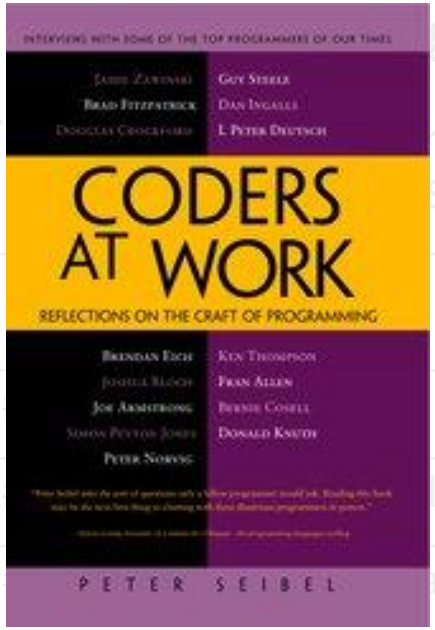
ITP 30002 Operating System

Concurrency Bugs

OSTEP Chapters 32

Shin Hong

Multithreaded Programs are Popular Yet Error-prone



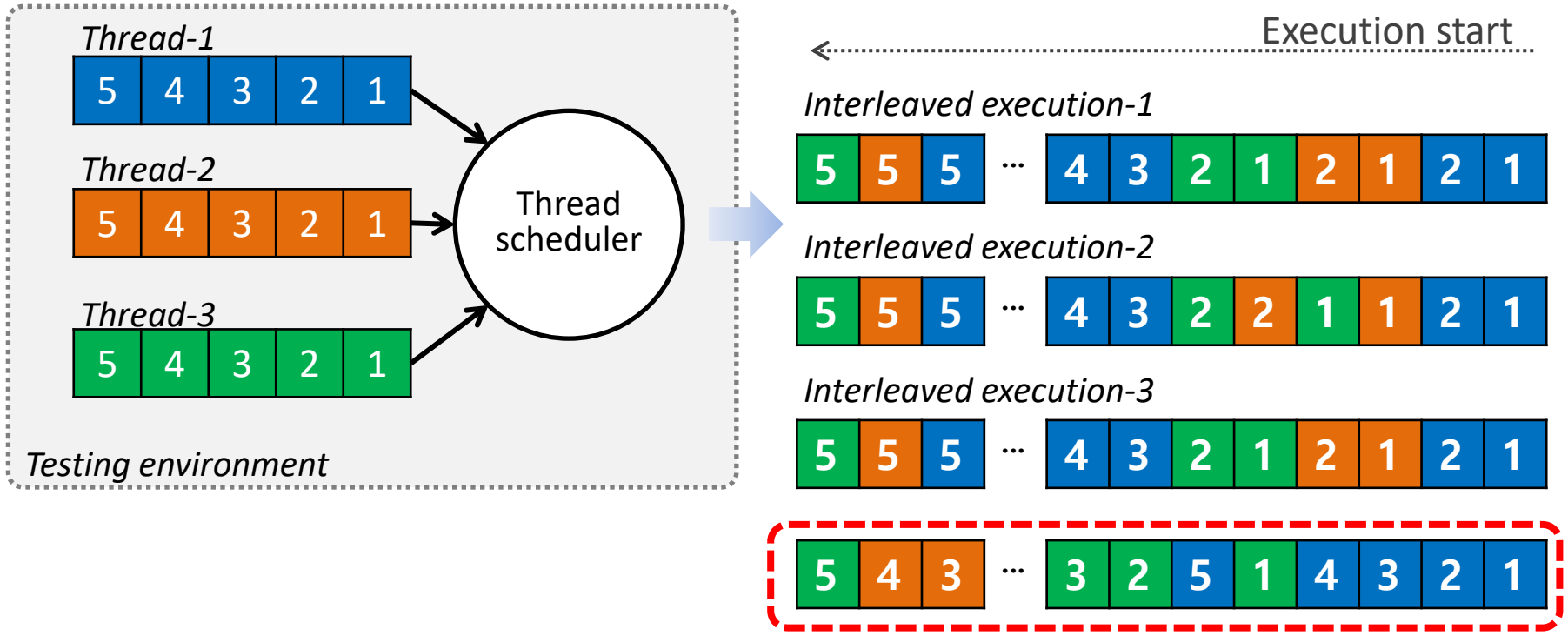
Most of my subjects (interviewee) have found that the hardest bugs to track down are in concurrent code

…

And almost every one seem to think that ubiquitous multi-core CPUs are going to force some serious changes in the way software is written

P. Siebel, *Coders at work* (2009) -- interview with 15 top programmers of our times:
Jamie Zawinski, Brad Fitzpatrick, Douglas Crockford, Brendan Eich, Joshua Bloch, Joe Armstrong, Simon Peyton Jones, Peter Norvig, Guy Steele, Dan Ingalls, L Peter Deutsch, Ken Thompson, Fran Allen, Bernie Cosell, Donald Knuth

Testing Multithreaded Programs is Difficult



- Testing with **the basic thread scheduler under stress** is **not effective to generate comprehensive schedules** which are possible for field environments

Common Concurrency Bugs

- Unintended Race conditions
  - Data race
  - **Ordering violation**
  - **Atomicity violation**

- Deadlock
  - **Resource deadlock**
  - Communication deadlock

Atomicity Violation

- A multithreaded program has an atomicity violation if a series of accesses on a shared data structure are not guaranteed to be executed atomically
  - a critical section results an invalid state when another thread interferes in a middle of its execution

- Bug example

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

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

Atomicity Violation - Solution

- Guard whole critical sections with a mutex
- Example

```
Thread 1::                                  ||  Thread 2::
pthread_mutex_lock(&proc_info_lock);        ||  pthread_mutex_lock(&proc_info_lock);
if (thd->proc_info) {                       ||  thd->proc_info = NULL;
  fputs(thd->proc_info, ...);               ||  pthread_mutex_unlock(&proc_info_lock);
}                                           ||
pthread_mutex_unlock(&proc_info_lock);      ||
```

Ordering Violation

- A multithreaded program has an ordering violation when there is no synchronization that correctly enforces an intended specific ordering between two operations

- Example

```
1  Thread 1::
2  void init() {
3     mThread = PR_CreateThread(mMain, ...);
4  }
```

```
6  Thread 2::
7  void mMain(...) {
8      mState = mThread->State;
9  }
```

Ordering Violation – Fix

- Add a synchronization primitive to enforce the desired ordering
  - conditional variable or semaphore

- Example

```
1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit             = 0;
4
5  Thread 1::
6  void init() {
7      ...
8      mThread = PR_CreateThread(mMain, ...);
9
10     // signal that the thread has been created...
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
```

```
18 Thread 2::
19 void mMain(...) {
20     ...
21     // wait for the thread to be initialized...
22     pthread_mutex_lock(&mtLock);
23     while (mtInit == 0)
24         pthread_cond_wait(&mtCond, &mtLock);
25     pthread_mutex_unlock(&mtLock);
26
27     mState = mThread->State;
28     ...
29 }
```

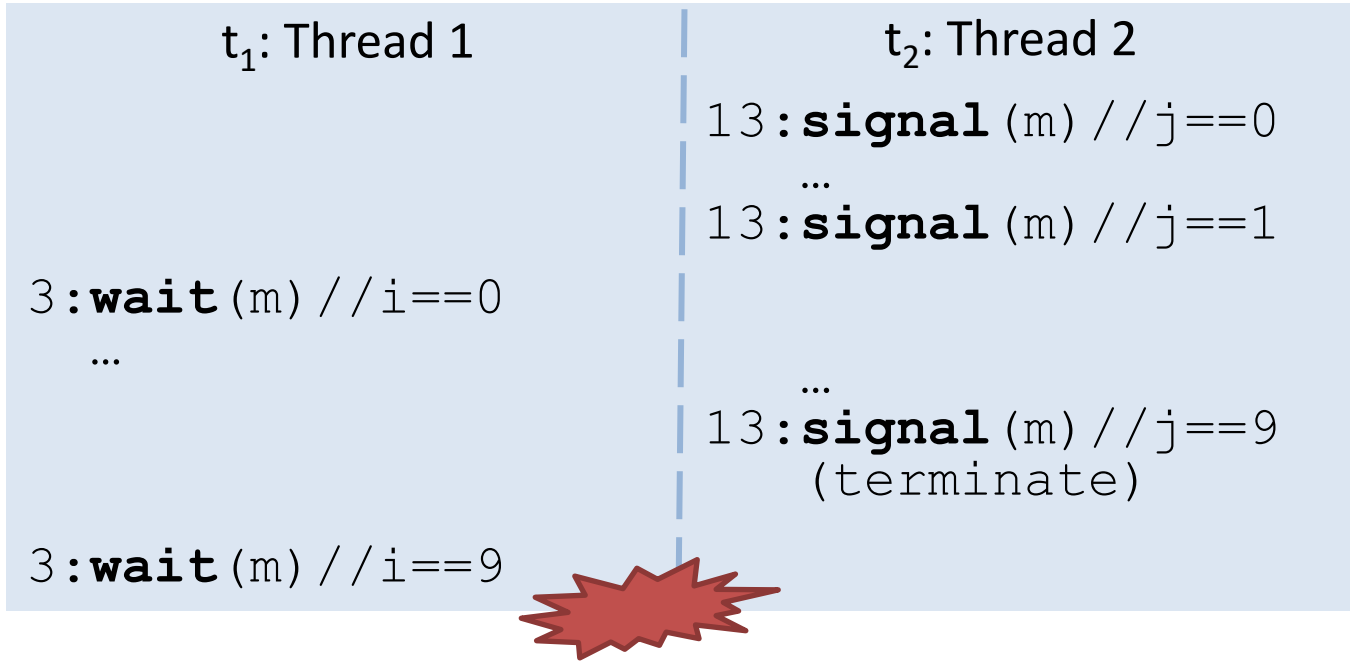
Deadlock

- A deadlock occurs when each of a set of threads is blocked, waiting for another thread in the set to satisfy certain condition
  - **release shared resource**
  - **raise event**

Communication Deadlock

- Lost signal

```
   Thread1() {                    Thread2() {
1: ...                         11: ...
2: for(i=0;i<10;i++){          12: for(j=0;j<10;j++){
3:   wait(m) ;}                13:   signal(m);}
   }                              }
```

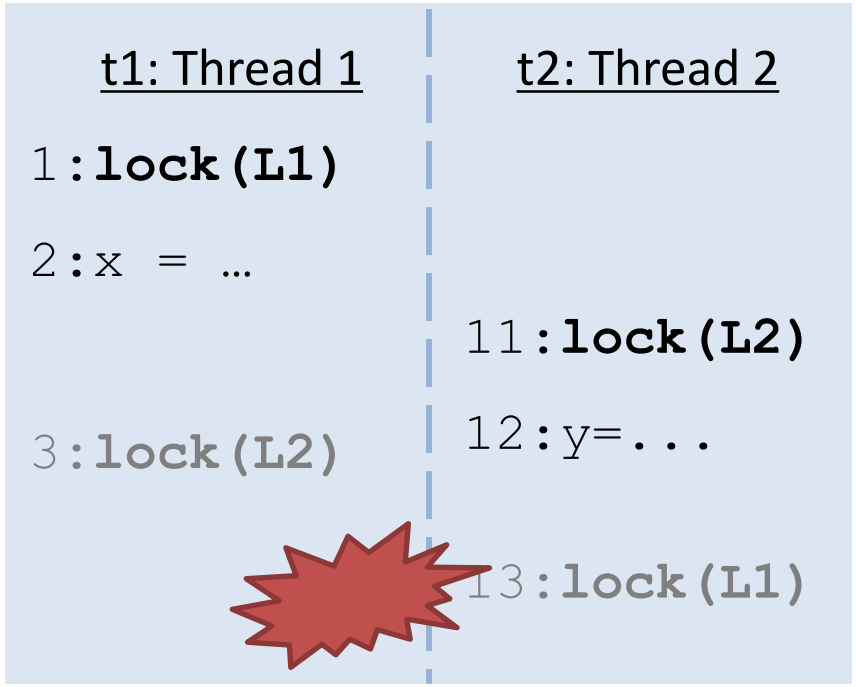


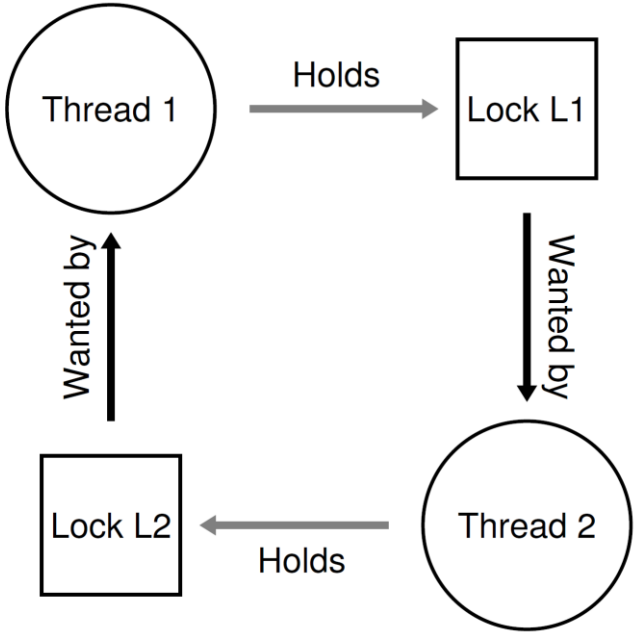
Resource Deadlock

- ABBA deadlock

```
Thread1() {
1: lock(L1)
2: x = … ;
3: lock(L2)
4: y = … ;
5: unlock(L2)
6: unlock(L1)
}
```

```
Thread2() {
11: lock(L2)
12: y = … ;
13: lock(L1)
14: x = … ;
15: unlock(L1)
16: unlock(L2)
}
```





Deadlock Bugs is Difficult to Find

- A deadlock bug induces deadlock situations **only under certain thread schedules**
- Systems software creates a **massive number of locks** for fine-grained concurrency controls
- **Function caller-callee relation** complicates the reasoning about possible nested lockings

Treatments

- Approaches
  - Prevention
  - Prediction
  - Avoidance

- Condition for resource deadlock
  - **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.

Deadlock Prevention (1)

- Do not allow any chance of circular wait

- Keep lock ordering as partial order
  - lock ordering: order of lock acquisition in a nested locking
    - lock A < lock B if a thread can acquire B while holding A
  - partial ordering: two conflicting orderings cannot exist in a same program
    - if lock A < lock B, there must not be a case of lock B < lock A

Deadlock Prevention (2)

- Do not allow hold-and-wait situations
  - Guard nested locking with a one big lock
  - Example

```
1    pthread_mutex_lock(prevention);
2    pthread_mutex_lock(L1);
3    pthread_mutex_lock(L2);
4    ...
5    pthread_mutex_unlock(prevention);
```

- Allow preemption by releasing a lock when a nested locking is blocking
  - Example

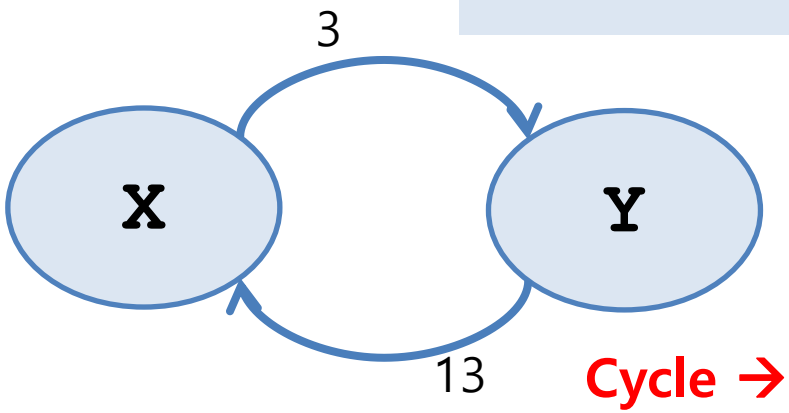
```
1  top:
2    pthread_mutex_lock(L1);
3    if (pthread_mutex_trylock(L2) != 0) {
4      pthread_mutex_unlock(L1);
5      goto top;
6    }
```

Resource Deadlock Prediction

```
Thread1() {
1: lock(X)
2: a = … ;
3: lock(Y)
4: b = … ;
5: unlock(Y)
6: unlock(X)
}
```

```
Thread2() {
11: lock(Y)
12: b = … ;
13: lock(X)
14: a = … ;
15: unlock(X)
16: unlock(Y)
}
```

t1:Thread 1	t2:Thread 2
1:lock(X)	
2:a = …	
3:lock(Y)	
4:b = …	
5:unlock(Y)	
	11:lock(Y)
6:unlock(X)	
	12:b=...
	13:lock(X)
	...



Cycle → Potential deadlock

Concurrency Bug Detection Techniques for Multithreaded Programs