CS:3620 Operating Systems

Concurrency Bugs

Bugs in concurrent programs

- Writing multi-threaded programs is tricky

- Bugs are non-deterministic and occur based on execution order of threads – very hard to debug

- Two types of bugs
  - Deadlocks: threads cannot execute any further and wait for each other
  - Non-deadlock bugs: non deadlock but incorrect results when threads execute

Non deadlock bugs

- Atomicity bugs – atomicity assumptions made by programmer are violated during execution of concurrent threads
  - Fix: locks for mutual exclusion

- Order-violation bugs – desired order of memory accesses is flipped during concurrent execution
  - Fix: condition variables

Atomicity bug: example

- One thread reads and prints a shared data item, while another concurrently modifies it


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


- Atomicity bugs can occur, not just when writing to shared data, but even when reading it

Atomicity bug example: fix

- Always use locks when accessing shared data


```
1  pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread 1::
4  pthread_mutex_lock(&proc_info_lock);
5  if (thd->proc_info) {
6    fputs(thd->proc_info, ...);
7  }
8  pthread_mutex_unlock(&proc_info_lock);
9
10 Thread 2::
11 pthread_mutex_lock(&proc_info_lock);
12 thd->proc_info = NULL;
13 pthread_mutex_unlock(&proc_info_lock);
```

Order violation bug: example

- Thread1 assumes Thread2 has already run

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


- No assumptions can be made on order of execution of concurrent threads

Ordering violation bug example: fix

- Use condition variables or semaphores

```
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 }
17
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 bugs

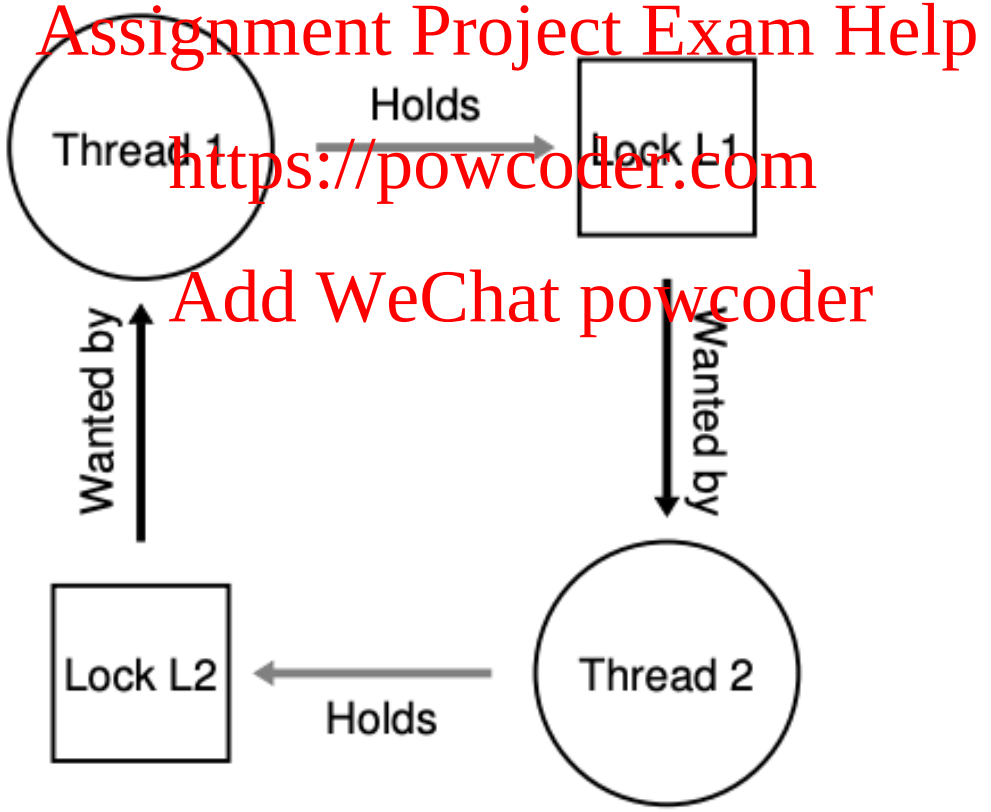
- Classic example: Thread1 holds lock L1 and is waiting for lock L2. Thread2 holds L2 and is waiting for L1.


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


- Deadlock need not always occur. Only occurs if executions overlap and context switch from a thread after acquiring only one lock.

Deadlock: a visual representation

- Cycle in a dependency graph



Conditions for deadlock

- Mutual exclusion: a thread claims exclusive control of a resource (e.g., lock)
- Hold-and-wait: thread holds a resource and is waiting for another
- No preemption: thread cannot be made to give up its resource (e.g., cannot take back a lock)
- Circular wait: there exists a cycle in the resource dependency graph
- ALL four of the above conditions must hold for a deadlock to occur

Preventing circular wait

- Acquire locks in a certain fixed order
  - E.g., both threads acquire L1 before L2
- Total ordering (or even a partial ordering on related locks) must be followed
  - E.g., order locks by address of lock variable


```
if (m1 > m2) { // grab in high-to-low address order
  pthread_mutex_lock(m1);
  pthread_mutex_lock(m2);
} else {
  pthread_mutex_lock(m2);
  pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (not the same lock)
```

Preventing hold-and-wait

- Acquire all locks at once, say, by acquiring a master lock first
- But this method may reduce concurrent execution and performance gains


```
1    pthread_mutex_lock(prevention); // begin acquisition
2    pthread_mutex_lock(L1);
3    pthread_mutex_lock(L2);
4    ...
5    pthread_mutex_unlock(prevention); // end
```

Other solutions to deadlocks

- Deadlock avoidance: if OS knew which process needs which locks, it can schedule the processes in that deadlock will not occur
  - Banker's algorithm is very popular, but impractical in real life to assume this knowledge
  - Example, below are locks needed by threads and a possible schedule decided by OS

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

Disclaimer

- *These lecture slides are based on a slide set by Youjip Won (Hanyang University) and Mythili Vutukuru (IIT Bombay)*