Lecture 16: Concurrency Bugs

Mythili Vutukuru
IIT Bombay
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.

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

Thread 2::
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

```c
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);
```
Order violation bug: example

• Thread1 assumes Thread2 has already run

```c
1   Thread 1::
2       void init () {  
3           ...
4           mThread = PR_CreateThread(mMain, ...);
5           ...
6       }
7
8   Thread 2::
9       void mMain( ...) {  
10           ...
11           mState = mThread->State;
12           ...
13       }
```

• No assumptions can be made on order of execution of concurrent threads
Ordering violation bug example: fix

- Use condition variables or semaphores

```c
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;
    ...
}
```
Deadlock bugs

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

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

```c
Thread 2:
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

Figure 32.2: The Deadlock 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

```c
if (m1 > m2) { // grab locks 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 (it is 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

```c
1   pthread_mutex_lock(prevention);  // begin lock 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

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>T2</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>T3</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>T4</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

• Detect and recover: reboot system or kill deadlocked processes