Concurrency Bugs
Non-Deadlock Bugs

- Atomicity violation (MySQL)
- Thread 1 is interrupted before it runs fputs() function
- Dereference the null pointer exception inside fputs()
- Memory access (thd->proc_info) needs to be serialized.

Thread1::
if (thd->proc_info) {
    ...
    fputs (thd->proc_info , ...);  
    ...
}

Thread2::

thd->proc_info = NULL;
Non-Deadlock Bugs

Fix: place thd->proc_info within lock and unlock routines

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread1:
pthread_mutex_lock(&lock);
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}
pthread_mutex_unlock(&lock);

Thread2:
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```
Non-Deadlock Bugs

- Order violation
- Thread 2 runs after creation.
  - mThread is not initialized.
  - Null-pointer dereference

```c
Thread1::
void init(){
    mThread = PR_CreateThread(mMain, ...);
}
```

```c
Thread2::
void mMain(...){
    mState = mThread->State
}
```
Non-Deadlock Bugs

• Fix: Use condition variables
• Enforce the order of execution between memory accesses

```
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);
  ...
}

Thread2:
void mMain(...)
{
  ...
```
Non-Deadlock Bugs

// 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

• Deadlock
  • A thread holds a lock and waits for another lock.

*Thread 1:*
```
pthread_mutex_lock (L1);
```

*Thread 2:*
```
context switch to Thread 2

pthread_mutex_lock (L2);

pthread_mutex_lock (L1);
```
Deadlock Bugs

• Encapsulation does not work well with locking.
• Vector class in Java

Thread 1: 
Thread 2:
Vector v1, v2;
\texttt{v1.addAll(v2);} \quad \texttt{v2.addAll(v1);}

• AddAll () needs to be thread safe.
Thread 1 acquires a lock for \texttt{v1}.
Thread 2 acquires a lock for \texttt{v2} at the same time.
Deadlock Bugs

• Conditions for deadlocks
  ▪ Mutual exclusion: a thread grabs a lock.
  ▪ Hold-and-wait: a thread holds a lock and waits for acquiring an additional lock.
  ▪ No preemption: a lock that is held by a thread cannot be taken away from the thread.
  ▪ Circular wait: *hold-and-wait* for a circular chain of threads.

• Deadlock occurs if all of conditions are met.
Deadlock Bugs

• Prevent circular wait
  ▪ Total ordering (common approach): acquire L1 before L2.
  ▪ Partial ordering: acquire L1 before L2, acquire L3 before L4

• Prevent Hold-and-Wait
  ▪ Acquire all locks at once

```c
pthread_mutex_lock (prevention);
pthread_mutex_lock (L1);
pthread_mutex_lock (L2);
...
pthread_mutex_unlock (prevention);
```

➢ Disadvantage: must know ahead of time all needed locks
Deadlock Bugs

• Prevent no preemption
• Use pthread_mutex_trylock()
  ▪ Hold lock if it is available or return an error code (lock is already held)
  ▪ Preemption: returning the ownership of a lock
    top:
    lock(L1);
    if( tryLock(L2) == -1 ){
        unlock(L1);
        goto top;
    }

• Livelock
  ▪ Another thread holds L2 and attempts to acquire L1.
  ▪ Both threads execute their code blocks at the same time.
• Solution: adding a random delay before jumping back
Deadlock Bugs

- Prevent mutual exclusion
- lock-free data structures
- Hardware instructions

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

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

- List insertion
- Race condition occurs if called by multiple threads

```c
void insert(int value) {
    node_t * n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    n->next = head;
    head = n;  // interrupt
}
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n));
}
Deadlock Bugs

- Avoid deadlock
- Find out what locks are acquired by what threads

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

- Scheduler does not run T1 and T2 at the same time.
- T3 grabs only one lock. It can run with T1.
Deadlock Bugs

- Decrease concurrency
- Trade-off between performance and deadlock avoidance

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

CPU 1: T4

CPU 2: T1 T2 T3