



CS3281 / CS5281

Concurrency Bugs

CS3281 / CS5281

Spring 2025

**Some lecture slides borrowed and adapted from “Operating Systems: 3 Easy Pieces”
and CMU’s “Computer Systems: A Programmer’s Perspective”*



Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu



Non-Deadlock Bugs: Atomicity Violation

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

```
Thread1::  
if (thd->proc_info) {  
    ...  
    fputs(thd->proc_info , ...);  
    ...  
}  
  
Thread2::  
thd->proc_info = NULL;
```

Non-Deadlock Bugs: Atomicity Violation

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

```
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 before Thread 1
 - mThread is not initialized
 - Null-pointer dereference

```
Thread1::  
void init(){  
    mThread = PR_CreateThread(mMain, ...);  
}  
  
Thread2::  
void mMain(...){  
    mState = mThread->State  
}
```

Non-Deadlock Bugs: Order Violation

- 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(...){  
    ...  
}
```

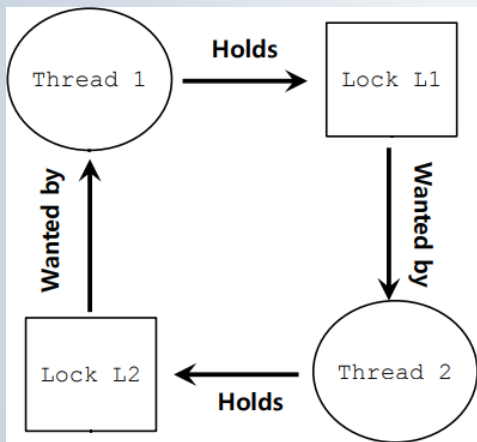
```
// wait for the thread to be initialized ...  
pthread_mutex_lock(&mtLock);  
while(mtInit == 0)  
    pthread_cond_wait(&mtCond, &mtLock);  
pthread_mutex_unlock(&mtLock);  
  
mState = mThread->State;  
...  
}
```

Deadlock

- Def: A process is *deadlocked* iff it is waiting for a condition that will never be true
- Typical Scenario
 - Processes 1 and 2 needs two resources (A and B) to proceed
 - Process 1 acquires A, waits for B
 - Process 2 acquires B, waits for A
 - Both will wait forever!

Deadlock Bugs

- Deadlock: A thread holds a lock and waits for another lock.



Thread 1:
`pthread_mutex_lock (L1);`
context switch to Thread 2
`pthread_mutex_lock (L2);`

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

Deadlock Bugs

- Encapsulation does not work well with locking
- Vector class in Java

Vector v1, v2;

Thread 1: *Thread 2:*
v1.addAll (v2); v2.addAll (v1);

- *addAll()* needs to be thread safe.
- Thread 1 acquires a lock for v1.
- Thread 2 acquires a lock for v2 at the same time.

Deadlock Bugs

- Conditions for deadlocks
 - Mutual exclusion: Thread grabs a lock
 - Hold-and-wait: Thread holds the lock and waits to acquire an additional lock
 - No preemption: Lock that is held cannot be taken away from the thread
 - Circular wait: *hold-and-wait* for a circular chain of threads
- Deadlock does not occur if **any** of the above conditions is not met

Deadlock Bugs

- Prevention for: Circular Wait
 - Total ordering (two locks): acquire L1 before L2
 - Partial ordering (multiple locks): group lock acquisition ordering
 - E.g., acquire L1 before L2, acquire L3 before L4
 - Prevention for: Hold-and-Wait
 - Acquire all locks at once
 - pthread_mutex_lock (prevention);
 - pthread_mutex_lock (L1);
 - pthread_mutex_lock (L2);
 - ...
 - pthread_mutex_unlock (prevention);
- Disadvantage:

Deadlock Bugs

- Prevention for: No Preemption
- Use `pthread_mutex_trylock()`
 - Acquire lock if it is available or return an error code (lock is already held)
 - Preemption: release 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

- Problems with trylock
 - Memory allocated after acquiring L1 should be released when acquiring L2 fails.
 - trylock does not preempt the ownership of a lock. It allows a thread to give back the ownership voluntarily.

Deadlock Bugs

- Prevention for: Mutual Exclusion
 - Lock-free data structures
- Create atomic functions based on hardware instructions

```
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

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n->value = value ;
    n->next = head;
    head    = n; ← interrupt
}

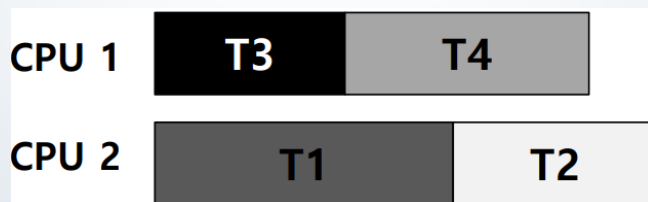
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

Deadlock Bugs

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

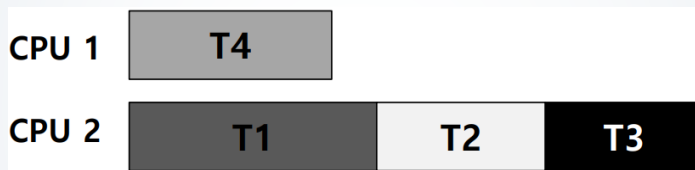
	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

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



Deadlock Bugs

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



- Decrease concurrency
- Trade-off between performance and deadlock avoidance
- It's impractical to gain a priori knowledge of lock acquisition for most applications.

Deadlocking with Semaphores

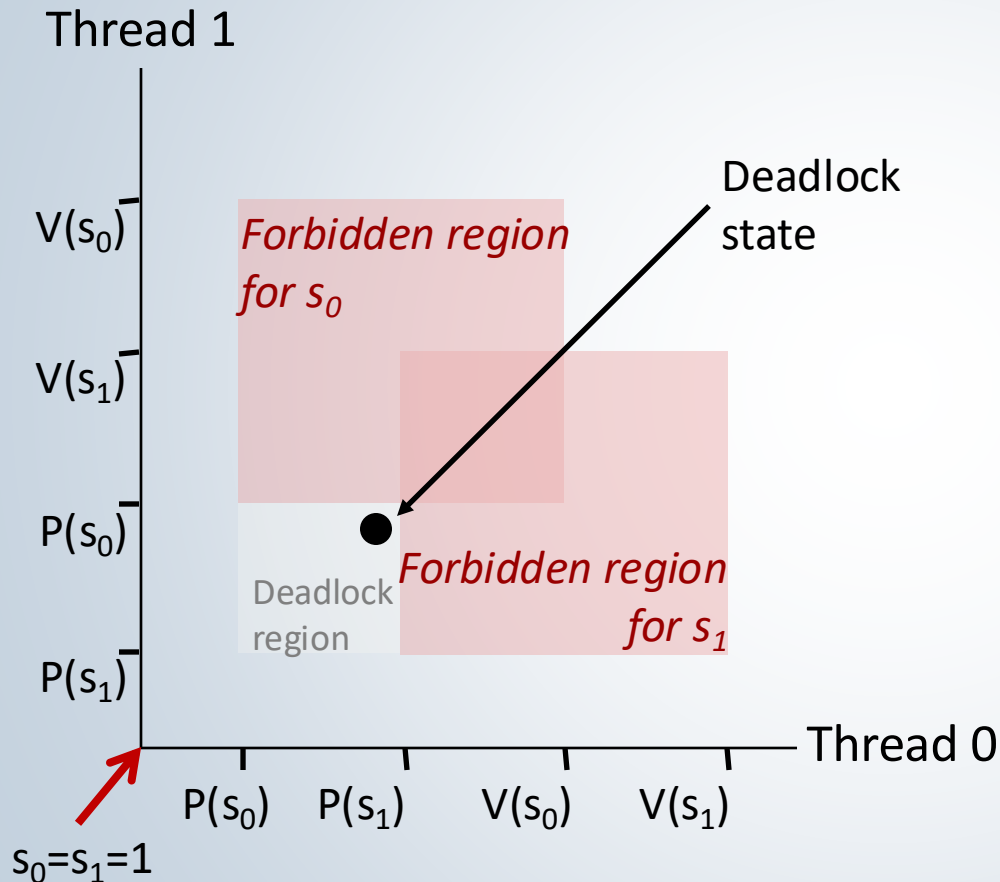
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

Tid[0]:

Tid[1]:

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Deadlock Visualization in a Progress Graph



Locking introduces the potential for *deadlock*: waiting for a condition that will never be true

Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state*, waiting for either s_0 or s_1 to become nonzero

Other trajectories luck out and skirt the deadlock region

Unfortunate fact: deadlock is often nondeterministic (race)

Avoiding Deadlock

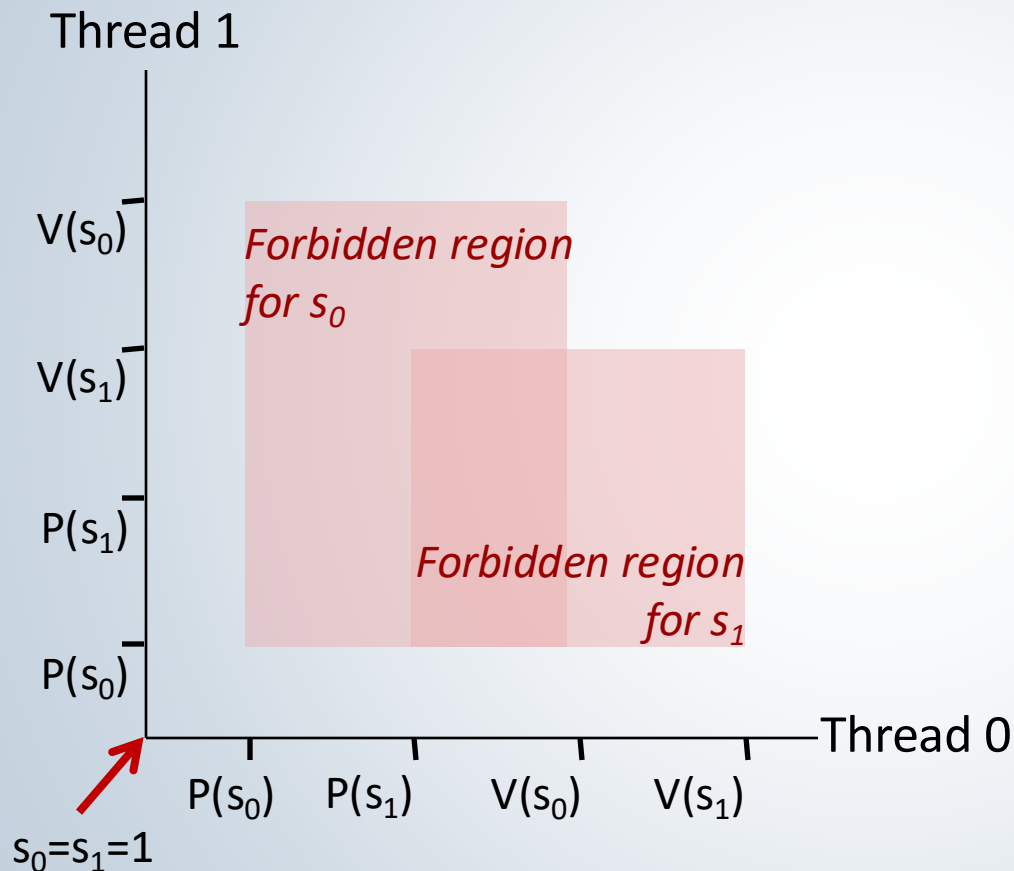
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
P(s0);
P(s1);
cnt++;
V(s0);
V(s1);

Tid[1]:
P(s0);
P(s1);
cnt++;
V(s1);
V(s0);

Avoiding Deadlock in a Progress Graph



No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

Summary

- Common solutions for deadlocks
 - Prevent deadlock
 - Order lock acquisition
 - Acquire lock atomically
 - Release lock voluntarily
 - Build lock-free atomic operations
 - Avoid deadlock
 - Schedule threads with global knowledge of lock acquisitions