Concurrency Bugs

Chapter 32
Tyler Harter
10/22/14
Review Semaphores
CV’s vs. Semaphores

CV rules of thumb:
- Keep state in addition to CV’s
- Always do wait/signal with lock held
- Whenever you acquire a lock, recheck state

How do semaphores eliminate these needs?
Condition Variable (CV)

Thread Queue:

Thread Queue:  Signal Queue:

Semaphore
Thread Queue:

Condition Variable (CV)

Semaphore

wait()
Condition Variable (CV)

Semaphore

Thread Queue: A

Signal Queue: A

ingnal()
Condition Variable (CV)

Thread Queue:

Semaphore

Thread Queue: Signal Queue:
<table>
<thead>
<tr>
<th><strong>Condition Variable (CV)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Queue:</td>
</tr>
<tr>
<td>Thread Queue: Signal Queue:</td>
</tr>
<tr>
<td>Semaphore</td>
</tr>
</tbody>
</table>
Condition Variable (CV)

Thread Queue:

Semaphore

Thread Queue:  Signal Queue:

signal
Condition Variable (CV)

Thread Queue:  
- B

Semaphore

Signal Queue:  
- signal

wait()
Thread Queue:

Condition Variable (CV)

Semaphore

wait()
Condition Variable (CV)

Thread Queue:

Semaphore

Thread Queue: Signal Queue:
Thread Queue:

B may wait forever (if not careful)

Thread Queue: Signal Queue:

Semaphore
Condition Variable (CV)

Thread Queue:

Thread Queue: B may wait forever (if not careful)

Signal Queue:

Semaphore

just use counter
int done = 0;
mutex_t m = MUTEX_INIT;
cond_t c = COND_INIT;
void *child(void *arg) {
    printf("child\n");
    Mutex_lock(&m);
    done = 1;
    cond_signal(&c);
    Mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    pthread_t c;
    printf("parent: begin\n");
Pthread_create(c, NULL, child, NULL);
    Mutex_lock(&m);
    while(done == 0)
        Cond_wait(&c, &m);
    Mutex_unlock(&m);
    printf("parent: end\n");
}
```c
int done = 0;
mutex_t m = MUTEX_INIT;
cond_t c = COND_INIT;
void *child(void *arg) {
    printf("child\n");
    Mutex_lock(&m);
    done = 1;
    cond_signal(&c);
    Mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    pthread_t c;
    printf("parent: begin\n");
    Pthread_create(c, NULL, child, NULL);
    Mutex_lock(&m);
    while(done == 0)
        Cond_wait(&c, &m);
    Mutex_unlock(&m);
    printf("parent: end\n");
}
```
# Join with CV

```c
int done = 0;
mutex_t m = MUTEX_INIT;
cond_t c = COND_INIT;
void *child(void *arg) {
    printf("child\n");
    Mutex_lock(&m);
    done = 1;
    cond_signal(&c);
    Mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    pthread_t c;
    printf("parent: begin\n");
    Pthread_create(c, NULL, child, NULL);
    Mutex_lock(&m);
    while(done == 0)
        Cond_wait(&c, &m);
    Mutex_unlock(&m);
    printf("parent: end\n");
}
```
sem_t s;
void *child(void *arg) {
    printf("child\n");
    sem_post(&s);
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0);
    pthread_t c;
    printf("parent: begin\n");
    Pthread_create(c, NULL, child, NULL);
    sem_wait(&s);
    printf("parent: end\n");
}
Semaphore Uses

For the following init’s, what might the use be?

(a) sem_init(&s, 0);

(b) sem_init(&s, 1);

(c) sem_init(&s, N);
Producer/Consumer

How many semaphores do we need?
How many semaphores do we need?

```
Sem_init(&empty, max); // max are empty
Sem_init(&full, 0);    // 0 are full
Sem_init(&mutex, 1);   // mutex
```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Sem_wait(&empty);
        Sem_wait(&mutex);
        do_fill(i);
        Sem_post(&mutex);
        Sem_post(&full);
    }
}

void *consumer(void *arg) {
    while (1) {
        Sem_wait(&full);
        Sem_wait(&mutex);
        tmp = do_get();
        Sem_post(&mutex);
        Sem_post(&empty);
        printf("%d\n", tmp);
    }
}
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Sem_wait(&empty);
        Sem_wait(&mutex);
        do_fill(i);
        Sem_post(&mutex);
        Sem_post(&full);
    }
}

void *consumer(void *arg) {
    while (1) {
        Sem_wait(&full);
        Sem_wait(&mutex);
        tmp = do_get();
        Sem_post(&mutex);
        Sem_post(&empty);
        printf("%d\n", tmp);
    }
}
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Sem_wait(&empty);
    Sem_wait(&mutex);
    do_fill(i);
    Sem_post(&mutex);
    Sem_post(&full);
  }
}

void *consumer(void *arg) {
  while (1) {
    Sem_wait(&full);
    Sem_wait(&mutex);
    tmp = do_get();
    Sem_post(&mutex);
    Sem_post(&empty);
    printf("%d\n", tmp);
  }
}
Concurrency Bugs
Concurrency in Medicine: Therac-25

“The accidents occurred when the high-power electron beam was activated instead of the intended low power beam, and without the beam spreader plate rotated into place. Previous models had hardware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The software interlock could fail due to a race condition.”

Concurrency in Medicine: Therac-25

“The accidents occurred when the high-power electron beam was activated instead of the intended low power beam, and without the beam spreader plate rotated into place. Previous models had hardware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The software interlock could fail due to a race condition.”

“…in three cases, the injured patients later died.”

Concurrency in Medicine: Therac-25

“The accidents occurred when the high-power electron beam was activated instead of the intended low power beam, and without the beam spreader plate rotated into place. Previous models had hardware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The software interlock could fail due to a race condition.”

“…in three cases, the injured patients later died.”

Getting concurrency right can sometimes save lives!

Concurrency Bugs are Common and Various

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

Concurrent Bugs are Common and Various

Lu et al. Study:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

Thread 1:

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

Thread 2:

thd->proc_info = NULL;

What’s wrong?
Thread 1:

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

Thread 2:

```c
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```
Concurrent Bugs are Common and Various

Lu et al. Study:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

Concurrent Bugs are Common and Various

Lu etal. Study:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

Thread 1:

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

Thread 2:

```c
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```
Thread 1:

```c
void init() {
    mThread = PR_CreateThread(mMain, ...);
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    ...
}
```

Thread 2:

```c
void mMain(...) {
    Mutex_lock(&mtLock);
    while(mtInit == 0)
        Cond_wait(&mtCond, &mtLock);
    Mutex_unlock(&mtLock);
    mState = mThread->State;
    ...
}
```
Concurrency Bugs are Common and Various

Lu et al. Study:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

Concurrency Bugs are Common and Various

Lu et al. Study:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).

who goes?
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Deadlock

Cooler name: the **deadly embrace** (Dijkstra).
Cooler name: the **deadly embrace** (Dijkstra).

who goes?
Deadlock!
Boring Code Example

Thread 1 [RUNNING]:
lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:
lock(&B);
lock(&A)
Boring Code Example

Thread 1 [RUNNING]:
lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:
lock(&B);
lock(&A)
Boring Code Example

Thread 1 [RUNNABLE]:

lock(&A);
lock(&B)

Thread 2 [RUNNING]:

lock(&B);
lock(&A)
Boring Code Example

Thread 1 [RUNNABLE]:
lock(&A);
lock(&B)

Thread 2 [RUNNING]:
lock(&B);
lock(&A)
Boring Code Example

Thread 1 [RUNNING]:
lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:
lock(&B);
lock(&A)
Boring Code Example

Thread 1 [SLEEPING]:
lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:
lock(&B);
lock(&A)
Boring Code Example

Thread 1 [SLEEPING]:
lock(&A);
lock(&B)

Thread 2 [RUNNING]:
lock(&B);
lock(&A)
Boring Code Example

Thread 1 [SLEEPING]:
lock(&A);
lock(&B)

Thread 2 [SLEEPING]:
lock(&B);
lock(&A)
Boring Code Example

Thread 1 [SLEEPING]:
lock(&A);
lock(&B)

Thread 2 [SLEEPING]:
lock(&B);
lock(&A)

Deadlock!
Circular Dependency

Thread 1 holds Lock A

Lock B wanted by Thread 2

Lock A wanted by Thread 1
Boring Code Example

Thread 1 [RUNNING]:
lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:
lock(&A);
lock(&B)
Boring Code Example

Thread 1 [RUNNING]:
lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:
lock(&A);
lock(&B)

Can’t deadlock.
Non-circular Dependency (fine)

Thread 1 holds Lock A

Lock B wanted by Thread 2

Lock A wanted by Thread 2
set_t *set_union (set_t *s1, set_t *s2) {
    set_t *rv = Malloc(sizeof(*rv));
    Mutex_lock(&s1->lock);
    Mutex_lock(&s2->lock);

    for(int i=0; i<s1->len; i++) {
        if(set_contains(s2, s1->items[i])
            set_add(rv, s1->items[i]);
    }
    Mutex_unlock(&s2->lock);
    Mutex_unlock(&s1->lock);
}
Encapsulation

Modularity can make it harder to see deadlocks.

Thread 1:
rv = set_union(setA, setB);

Thread 2:
rv = set_union(setB, setA);
Encapsulation

Modularity can make it harder to see deadlocks.

**Thread 1:**

```
rv = set_union(setA, setB);
```

**Thread 2:**

```
rv = set_union(setB, setA);
```

Solutions?
Deadlock Theory

Deadlocks can only happen with these **four conditions**:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait
Deadlock Theory

Deadlocks can only happen with these **four conditions**:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating **one condition**.
Deadlock Theory

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.
Mutual Exclusion

Def:

Threads claim exclusive control of resources that they require (e.g., thread grabs a lock).
Wait-Free Algorithms

Strategy: eliminate lock use.

Assume we have:
int CompAndSwap(int *addr, int expected, int new)
0: fail, 1: success

void add_v1(int *val, int amt) {
    Mutex_lock(&m);
    *val += amt;
    Mutex_unlock(&m);
}

void add_v2(int *val, int amt) {
    int old = *value;
    do {
        } while(!CompAndSwap(val, old, old+amt);
}

Wait-Free Algorithms

Strategy: eliminate lock use.

Assume we have:
int CompAndSwap(int *addr, int expected, int new)

```
void insert(int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    lock(&m);
    n->next = head;
    head = n;
    unlock(&m);
}
```

eliminate the lock!
Wait-Free Algorithms

Strategy: eliminate lock use.

Assume we have:

```c
int CompAndSwap(int *addr, int expected, int new)
```

```c
void insert(int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    do {
        n->next = head;
    } while (!CompAndSwap(&head, n->next, n));
}
```
Deadlock Theory

Deadlocks can only happen with these **four conditions**:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating **one condition**.
Deadlock Theory

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.
Hold-and-Wait

Def:

Threads hold resources allocated to them (e.g., locks they have already acquired) while waiting for additional resources (e.g., locks they wish to acquire).
Eliminate Hold-and-Wait

Strategy: acquire all locks atomically **once** (cannot acquire again until all have been released).

For this, use a meta lock, like this:

```c
lock(&meta);
lock(&L1);
lock(&L2);
...
unlock(&meta);
```
Eliminate Hold-and-Wait

Strategy: acquire all locks atomically once (cannot acquire again until all have been released).

For this, use a meta lock, like this:

```c
lock(&meta);
lock(&L1);
lock(&L2);
...
unlock(&meta);
```

Discuss:
- how should unlock work?
- disadvantages?
Deadlock Theory

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.
Deadlock Theory

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no-preemption
- circular wait

Eliminate deadlock by eliminating one condition.
No preemption

Def:

Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
Support Preemption

Strategy: if we can’t get what we want, release what we have.

```
top:
    lock(A);
    if (trylock(B) == -1) {
        unlock(A);
        goto top;
    }
    ...
```
Support Preemption

Strategy: if we can’t get what we want, release what we have.

top:
  lock(A);
  if (trylock(B) == -1) {
    unlock(A);
    goto top;
  }

Discuss:
  - disadvantages?
Support Preemption

Strategy: if we can’t get what we want, release what we have.

top:
lock(A);
if (trylock(B) == -1) {
    unlock(A);
goto top;
}
...

Discuss:
- disadvantages? (livelock)
Deadlock Theory

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.
Deadlock Theory

Deadlocks can only happen with these four conditions:
- mutual exclusion
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating one condition.
Circular Wait

Def:

There exists a *circular chain* of threads such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.
Eliminating Circular Wait

Strategy:
- decide which locks should be acquired before others
- if A before B, never acquire A if B is already held!
- document this, and write code accordingly
Lock Ordering in Linux

In linux-3.2.51/include/linux/fs.h

/*
 * inode->i_mutex nesting subclasses for the lock
 * validator:
 *
 * 0: the object of the current VFS operation
 * 1: parent
 * 2: child/target
 * 3: quota file
 *
 * The locking order between these classes is
 * parent -> child -> normal -> xattr -> quota
 */
Lock Ordering in Linux

In linux-3.2.51/include/linux/fs.h

/*
 * inode->i_mutex nesting subclasses for the lock validator:
 *
 * 0: the object of the current VFS operation
 * 1: parent
 * 2: child/target
 * 3: quota file
 *
 * The locking order between these classes is
 * parent -> child -> normal -> xattr -> quota
 */
Linux lockdep Module

Idea:
- track order in which locks are acquired
- give warning if circular

Extremely useful for debugging!
Example Output

[ INFO: possible circular locking dependency detected ]
3.1.0rc4test00131g9e79e3e #2

insmod/1357 is trying to acquire lock:
(lockC){+.+...}, at: [<fffffffffa000d438>] pick_test+0x2a2/0x892
[lockdep_test]

but task is already holding lock:
(lockB){+.+...}, at: [<fffffffffa000d42c>] pick_test+0x296/0x892
[lockdep_test]

Summary

Concurrency is hard, *encapsulation* makes it harder!

Have a *strategy* to avoid deadlock and stick to it.

Choosing a *lock order* is probably most practical.

When possible, *avoid concurrent solutions* altogether!
Announcements

Office hours: 1pm in office.

p3a due Friday.

Start p3b!

Thursday discussion: hand back and discuss test.