Lecture 16: Semaphores and Concurrency Bugs

Operating Systems

Content taken from: https://pages.cs.wisc.edu/~remzi/OSTEP/

https://www.cse.iitb.ac.in/~mythili/os/

Last Few Classes

- How should multiple threads concurrently access shared data?
- Avoid Race Conditions in Critical Section
- Mutual Exclusion using Locks
- Waiting and Signaling using Condition Variables
  - Parent waiting for child
  - Producer-Consumer Problem
- Semaphores
  - Higher level synchronization primitive which can be used as a lock or condition variable.

Semaphores

- Synchronization primitive like condition variables
- Semaphore is a variable with an underlying counter
- Two functions on a semaphore variable
  - post function increments the counter and wake up a thread which is waiting
  - wait function decrements the counter and blocks the calling thread if the resulting value is negative

```
1  #include <semaphore.h>
2  sem_t s;
3  sem_init(&s, 0, 1);
```

```
1  int sem_wait(sem_t *s) {
2      decrement the value of semaphore s by one
3      wait if value of semaphore s is negative
4  }
5
6  int sem_post(sem_t *s) {
7      increment the value of semaphore s by one
8      if there are one or more threads waiting, wake one
9  }
```

Figure 31.2: **Semaphore: Definitions Of Wait And Post**

Binary Semaphores (Locks)

```
1 sem_t m;
2 sem_init(&m, 0, X); // initialize to X; what should X be?
3
4 sem_wait(&m);
5 // critical section here
6 sem_post(&m);
```

Figure 31.3: **A Binary Semaphore (That Is, A Lock)**

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

Semaphores for Ordering

```
1  sem_t s;
2
3  void *child(void *arg) {
4      printf("child\n");
5      sem_post(&s); // signal here: child is done
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     sem_init(&s, 0, X); // what should X be?
11     printf("parent: begin\n");
12     pthread_t c;
13     Pthread_create(&c, NULL, child, NULL);
14     sem_wait(&s); // wait here for child
15     printf("parent: end\n");
16     return 0;
17 }
```

Figure 31.6: **A Parent Waiting For Its Child**

Semaphores for Ordering

```
1  sem_t s;
2
3  void *child(void *arg) {
4      printf("child\n");
5      sem_post(&s); // signal here: child is done
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     sem_init(&s, 0, X); // what should X be?
11     printf("parent: begin\n");
12     pthread_t c;
13     Pthread_create(&c, NULL, child, NULL);
14     sem_wait(&s); // wait here for child
15     printf("parent: end\n");
16     return 0;
17 }
```

Figure 31.6: **A Parent Waiting For Its Child**

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists, can run)*	Ready
0	call `sem_wait()`	Run		Ready
-1	`decr sem`	Run		Ready
-1	`(sem<0)`→`sleep`	Sleep		Ready
-1	*Switch→Child*	Sleep	`child runs`	Run
-1		Sleep	call `sem_post()`	Run
0		Sleep	`inc sem`	Run
0		Ready	`wake(Parent)`	Run
0		Ready	`sem_post()` returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	`sem_wait()` returns	Run		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists; can run)*	Ready
0	*Interrupt→Child*	Ready	`child runs`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`inc sem`	Run
1		Ready	`wake(nobody)`	Run
1		Ready	`sem_post()` returns	Run
1	`parent runs`	Run	*Interrupt→Parent*	Ready
1	call `sem_wait()`	Run		Ready
0	`decrement sem`	Run		Ready
0	`(sem≥0)`→`awake`	Run		Ready
0	`sem_wait()` returns	Run		Ready

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

Producer / Consumer using Semaphores

```
1  int buffer[MAX];
2  int fill = 0;
3  int use  = 0;
4
5  void put(int value) {
6      buffer[fill] = value;    // Line F1
7      fill = (fill + 1) % MAX; // Line F2
8  }
9
10 int get() {
11     int tmp = buffer[use];   // Line G1
12     use = (use + 1) % MAX;   // Line G2
13     return tmp;
14 }
```

Figure 31.9: **The Put And Get Routines**

```
1  sem_t empty;
2  sem_t full;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          sem_wait(&empty);       // Line P1
8          put(i);                 // Line P2
9          sem_post(&full);        // Line P3
10     }
11 }
12
13 void *consumer(void *arg) {
14     int tmp = 0;
15     while (tmp != -1) {
16         sem_wait(&full);        // Line C1
17         tmp = get();            // Line C2
18         sem_post(&empty);       // Line C3
19         printf("%d\n", tmp);
20     }
21 }
22
23 int main(int argc, char *argv[]) {
24     // ...
25     sem_init(&empty, 0, MAX); // MAX are empty
26     sem_init(&full, 0, 0);    // 0 are full
27     // ...
28 }
```

Figure 31.10: **Adding The Full And Empty Conditions**

Adding mutual exclusion

```
1  void *producer(void *arg) {
2      int i;
3      for (i = 0; i < loops; i++) {
4          sem_wait(&mutex);        // Line P0 (NEW LINE)
5          sem_wait(&empty);        // Line P1
6          put(i);                  // Line P2
7          sem_post(&full);         // Line P3
8          sem_post(&mutex);        // Line P4 (NEW LINE)
9      }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&mutex);        // Line C0 (NEW LINE)
16         sem_wait(&full);         // Line C1
17         int tmp = get();         // Line C2
18         sem_post(&empty);        // Line C3
19         sem_post(&mutex);        // Line C4 (NEW LINE)
20         printf("%d\n", tmp);
21     }
22 }
```

Figure 31.11: **Adding Mutual Exclusion (Incorrectly)**

Avoiding Deadlock

```
1  void *producer(void *arg) {
2      int i;
3      for (i = 0; i < loops; i++) {
4          sem_wait(&empty);        // Line P1
5          sem_wait(&mutex);        // Line P1.5 (MUTEX HERE)
6          put(i);                  // Line P2
7          sem_post(&mutex);        // Line P2.5 (AND HERE)
8          sem_post(&full);         // Line P3
9      }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&full);         // Line C1
16         sem_wait(&mutex);        // Line C1.5 (MUTEX HERE)
17         int tmp = get();         // Line C2
18         sem_post(&mutex);        // Line C2.5 (AND HERE)
19         sem_post(&empty);        // Line C3
20         printf("%d\n", tmp);
21     }
22 }
```

Figure 31.12: **Adding Mutual Exclusion (Correctly)**

Reader-Writer Locks

- Consider a data structure concurrently accessed by multiple threads.
- Only a single writer should be allowed to update the data structure.
- Multiple readers can be allowed to read from the data structure simultaneously.

```
1  typedef struct _rwlock_t {
2    sem_t lock;       // binary semaphore (basic lock)
3    sem_t writelock; // allow ONE writer/MANY readers
4    int   readers;   // #readers in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8    rw->readers = 0;
9    sem_init(&rw->lock, 0, 1);
10   sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14   sem_wait(&rw->lock);
15   rw->readers++;
16   if (rw->readers == 1) // first reader gets writelock
17     sem_wait(&rw->writelock);
18   sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22   sem_wait(&rw->lock);
23   rw->readers--;
24   if (rw->readers == 0) // last reader lets it go
25     sem_post(&rw->writelock);
26   sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30   sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34   sem_post(&rw->writelock);
35 }
```

Figure 31.13: **A Simple Reader-Writer Lock**

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
  - **Deadlock Bugs:** 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-Violation Bugs** – atomicity assumptions made by programmer are violated during execution of concurrent threads
- **Order-Violation Bugs** – desired order of accessing shared data is flipped during concurrent execution

Atomicity-Violation Bugs

- A code region is intended to be atomic but the atomicity is not enforced during execution
  - i.e. a thread tries to access and update shared data but gets interrupted in between by another thread which modifies the same piece of shared data
- Fix using locks

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

Figure 32.2: **Atomicity Violation (`atomicity.c`)**

```
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);
```

Figure 32.3: **Atomicity Violation Fixed (`atomicity_fixed.c`)**

Order-Violation Bugs

- Desired order between two memory accesses is flipped
  - **A** should execute before **B** but the order is not enforced during execution
- Fix using condition variables

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

Figure 32.4: **Ordering Bug (`ordering.c`)**

```
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 }
```

Figure 32.5: **Fixing The Ordering Violation (`ordering_fixed.c`)**

Deadlock bugs

- Deadlock is said to occur when no threads are able to make progress as they are waiting for each other to release some resources/locks.
- 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);
```

Figure 32.6: **Simple Deadlock (`deadlock.c`)**

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

Deadlock Dependency Graph

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

Figure 32.6: **Simple Deadlock (`deadlock.c`)**

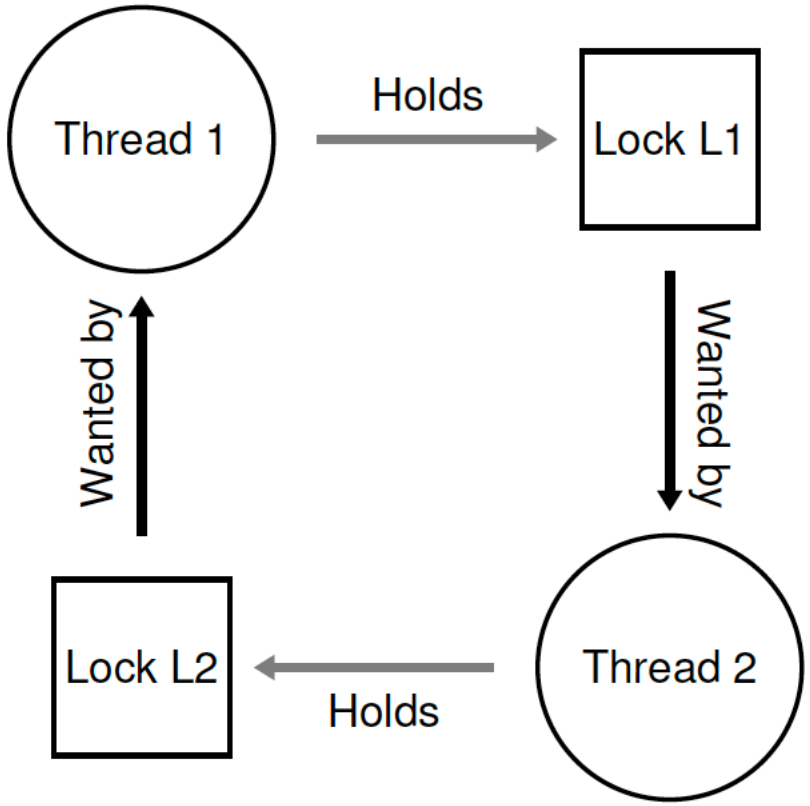


Figure 32.7: **The Deadlock Dependency Graph**

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

```
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
```

Allowing Preemption

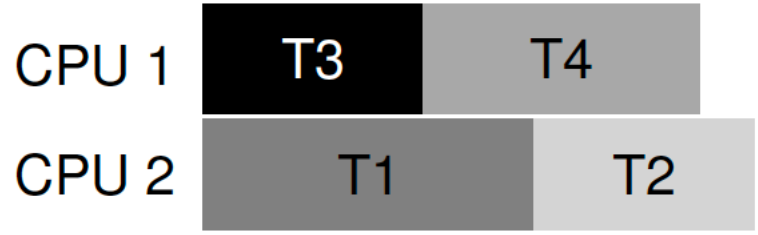
- E.g. Allow a thread to give up a lock if another lock is not available.

```
1  top:
2    pthread_mutex_lock(L1);
3    if (pthread_mutex_trylock(L2) != 0) {
4      pthread_mutex_unlock(L1);
5      goto top;
6    }
```

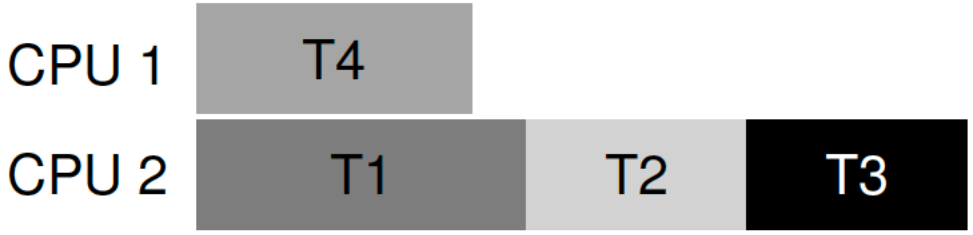
Deadlock Avoidance via Scheduling

- Avoidance requires global knowledge of which locks may be acquired by any thread during execution
- Schedule threads in a way so that no deadlock can occur.
- Impractical in real life to have this knowledge

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



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



Detect and Recover

- If deadlocks are rare, just reboot system or kill deadlocked processes