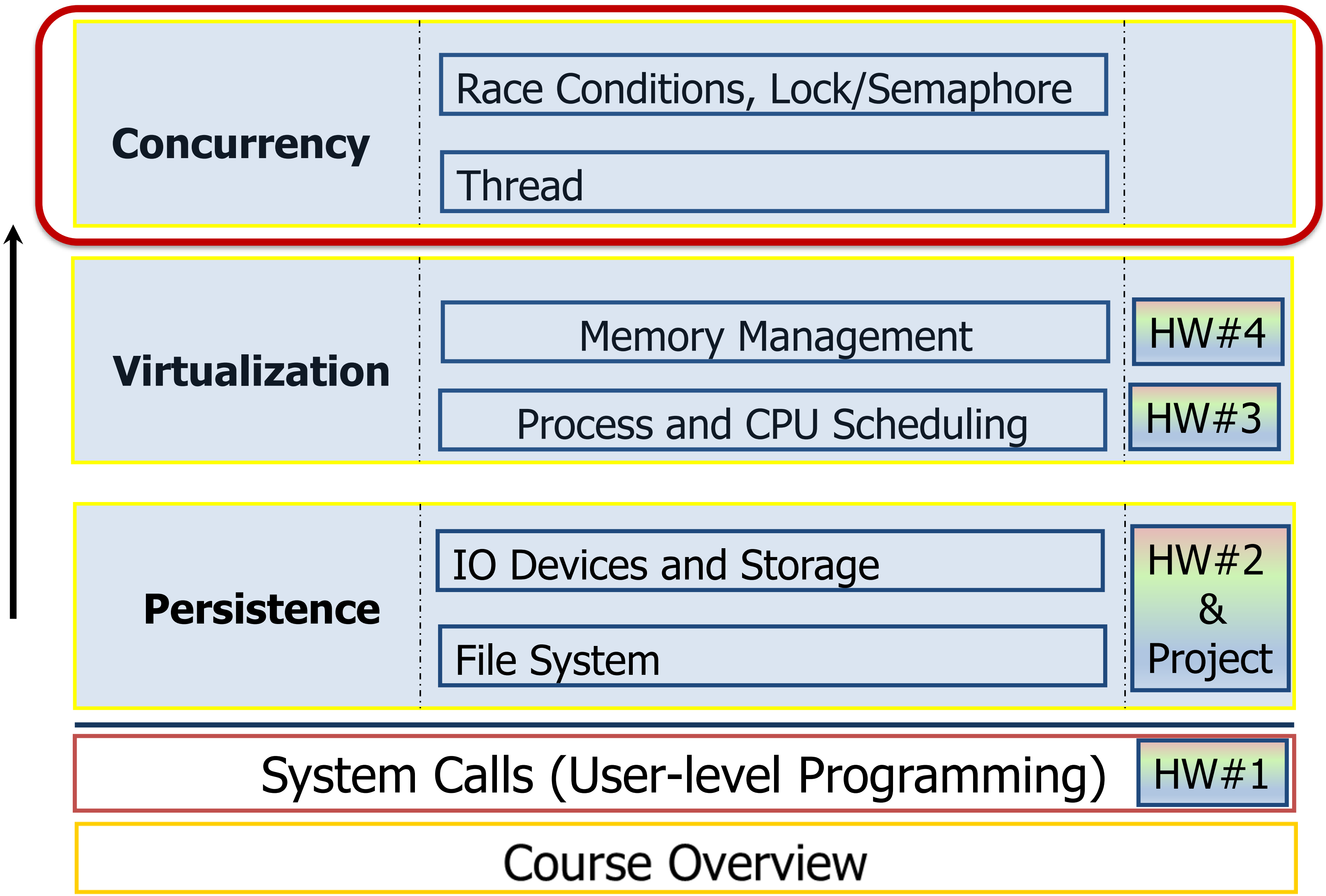
Lecture 15: Concurrency – Semaphore and Common Concurrency Problems

The Course Organization (Bottom-up)



Part I: Semaphore

Semaphore: A definition

- An object with an integer value
  - We can manipulate with two routines; `sem_wait()` and `sem_post()`.
  - Initialization

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize s to the value 1
```

- Declare a semaphore `s` and initialize it to the value 1
- The second argument, 0, indicates that the semaphore is shared between *threads in the same process*.

Semaphore: Interact with semaphore

- `sem_wait()`

```
int sem_wait(sem_t *s) {
      decrement the value of semaphore s by one
      wait if value of semaphore s is negative
}
```

- When `sem_wait()` is called,
  - Decrement the value of semaphore s by one
  - If the value of the semaphore
    - *0* or *higher*, **return right away**;
    - otherwise (lower than 0), the caller will be suspended and wait for a subsequent post.
      - When negative, the value of the semaphore is equal to the number of waiting threads.

Semaphore: Interact with semaphore (Cont.)

- `sem_post()`

```
int sem_post(sem_t *s) {
      increment the value of semaphore s by one
      if there are one or more threads waiting, wake one
}
```

- Simply **increments** the value of the semaphore.
- If there is a thread waiting to be woken, **wakes** one of them up.

Binary Semaphores (Locks)

- What should **X** be?
  - The initial value should be **1**.

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X; what should X be?

sem_wait(&m);
//critical section here
sem_post(&m);
```

Thread Trace: Single Thread Using A Semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	`call sema_wait()`	
0	`sem_wait() returns`	
0	`(crit sect)`	
0	`call sem_post()`	
1	`sem_post() returns`	

Thread Trace: Two Threads Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() retruns	Running		Ready
0	(crit set: begin)	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0)→sleep	sleeping
-1		Running	*Switch → T0*	sleeping
-1	(crit sect: end)	Running		sleeping
-1	call sem_post()	Running		sleeping
0	increment sem	Running		sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Semaphores As Condition Variables

```
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- What should **X** be?
  - The value of semaphore should be set to is **0**.

Thread Trace: Parent Waiting For Child (Case 1)

- The parent call `sem_wait()` before the child has called `sem_post()`.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	`call sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem < 0)→sleep`	sleeping		Ready
-1	*Switch→Child*	sleeping	`child runs`	Running
-1		sleeping	`call sem_post()`	Running
0		sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post() returns`	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	`sem_wait() retruns`	Running		Ready

Thread Trace: Parent Waiting For Child (Case 2)

- The child runs to completion before the parent call `sem_wait()`.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() retruns`	Running		Ready

The Producer/Consumer (Bounded-Buffer) Problem

- **Producer**: `put()` interface
  - Wait for a buffer to become *empty* in order to put data into it.
- **Consumer**: `get()` interface
  - Wait for a buffer to become *filled* before using it.

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
     buffer[fill] = value;      // line f1
     fill = (fill + 1) % MAX;   // line f2
}

int get() {
     int tmp = buffer[use];     // line g1
     use = (use + 1) % MAX;     // line g2
     return tmp;
}
```

The Producer/Consumer (Bounded-Buffer) Problem

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&empty);           // line P1
            put(i);                     // line P2
            sem_post(&full);            // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
            sem_wait(&full);            // line C1
            tmp = get();                // line C2
            sem_post(&empty);           // line C3
            printf("%d\n", tmp);
    }
}
…
```

First Attempt: Adding the Full and Empty Conditions

The Producer/Consumer (Bounded-Buffer) Problem

```
int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX);      // MAX buffers are empty to begin with…
    sem_init(&full, 0, 0);         // … and 0 are full
    // …
}
```

First Attempt: Adding the Full and Empty Conditions (Cont.)

- Imagine that MAX is greater than 1 .
  - If there are multiple producers, race condition can happen at line *f1*.
  - It means that the old data there is overwritten.

- We've forgotten here is **mutual exclusion**.
  - The filling of a buffer and incrementing of the index into the buffer is a critical section.

A Solution: Adding Mutual Exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&mutex);          // line p0 (NEW LINE)
            sem_wait(&empty);          // line p1
            put(i);                    // line p2
            sem_post(&full);           // line p3
            sem_post(&mutex);          // line p4 (NEW LINE)
    }
}

(Cont.)
```

Adding Mutual Exclusion (Incorrectly)

A Solution: Adding Mutual Exclusion

```c
(Cont.)
void *consumer(void *arg) {
        int i;
        for (i = 0; i < loops; i++) {
                sem_wait(&mutex);           // line c0 (NEW LINE)
                sem_wait(&full);            // line c1
                int tmp = get();            // line c2
                sem_post(&empty);           // line c3
                sem_post(&mutex);           // line c4 (NEW LINE)
                printf("%d\n", tmp);
        }
}
```

Adding Mutual Exclusion (Incorrectly)

A Solution: Adding Mutual Exclusion (Cont.)

- Imagine two thread: one producer and one consumer.
  - The consumer **acquire** the `mutex` (line c0).
  - The consumer **calls** `sem_wait()` on the full semaphore (line c1).
  - The consumer is **blocked** and **yield** the CPU.
    - The consumer <u>still holds the mutex</u>!
  - The producer **calls** `sem_wait()` on the binary `mutex` semaphore (line p0).
  - The producer is now **stuck** waiting too – A classic deadlock.

Finally, A Working Solution

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
     int i;
     for (i = 0; i < loops; i++) {
          sem_wait(&empty);          // line p1
          sem_wait(&mutex);          // line p1.5 (MOVED MUTEX HERE...)
          put(i);                    // line p2
          sem_post(&mutex);          // line p2.5 (... AND HERE)
          sem_post(&full);           // line p3
     }
}

(Cont.)
```

Adding Mutual Exclusion (Correctly)

Finally, A Working Solution

```
(Cont.)
 void *consumer(void *arg) {
     int i;
     for (i = 0; i < loops; i++) {
             sem_wait(&full);            // line c1
             sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE…)
             int tmp = get();            // line c2
             sem_post(&mutex);           // line c2.5 (… AND HERE)
             sem_post(&empty);           // line c3
             printf("%d\n", tmp);
     }
}

int main(int argc, char *argv[]) {
     // …
     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with …
     sem_init(&full, 0, 0);    // ... and 0 are full
     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
     // …
}
```

Adding Mutual Exclusion (Correctly)

Reader-Writer Locks

- Imagine a number of concurrent list operations, including **inserts** and simple **lookups**.
  - **insert:**
    - Change the state of the list
    - A traditional critical section makes sense.
  - **lookup:**
    - Simply *read* the data structure.
    - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

This special type of lock is known as a reader-writer lock.

A Reader-Writer Locks

- Only **a single writer** can acquire the lock.
- Once a reader has acquired a read lock,
  - **More readers** will be allowed to acquire the read lock too.
  - A writer will have to wait until all readers are finished.

```
typedef struct _rwlock_t {
    sem_t lock;      // binary semaphore (basic lock)
    sem_t writelock; // used to allow ONE writer or MANY readers
    int readers;     // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    …
```

A Reader-Writer Locks (Cont.)

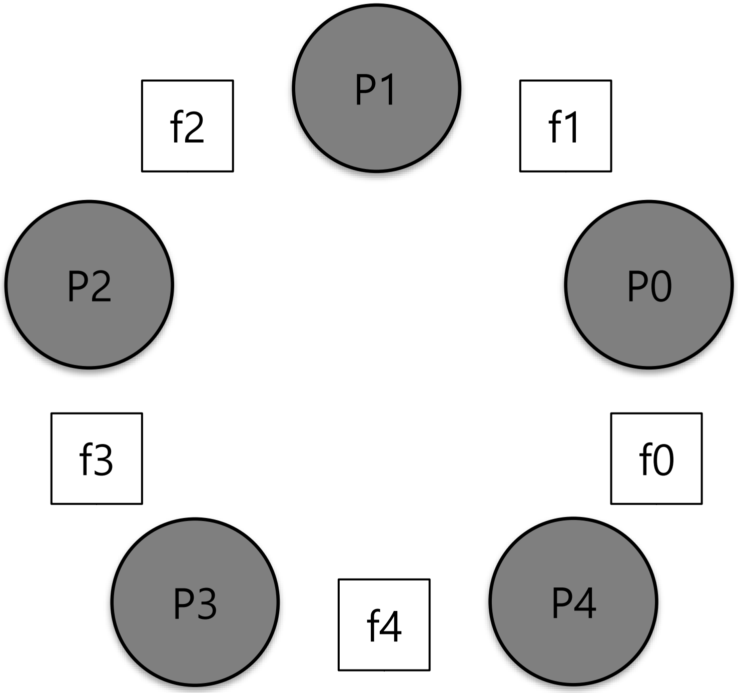
```
    rw->readers++;
    if (rw->readers == 1)
            sem_wait(&rw->writelock); // first reader acquires writelock
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0)
            sem_post(&rw->writelock); // last reader releases writelock
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock);
}
```

A Reader-Writer Locks (Cont.)

- The reader-writer locks have fairness problem.
  - It would be relatively easy for reader to **starve writer**.
  - How to <u>prevent</u> more readers from entering the lock once a writer is waiting?

The Dining Philosophers

- Assume there are five "**philosophers**" sitting around a table.
  - Between each pair of philosophers is <u>a single fork</u> (five total).
  - The philosophers each have times where they **think**, and don't need any forks, and times where they **eat**.
  - In order to *eat*, a philosopher needs two forks, both the one on their *left* and the one on their *right*.
  - **The contention for these forks.**



The Dining Philosophers (Cont.)

- Key challenge
  - There is **no deadlock**.
  - **No** philosopher **starves** and never gets to eat.
  - **Concurrency** is high.

```
while (1) {
        think();
        getforks();
        eat();
        putforks();
}
```

Basic loop of each philosopher

```
// helper functions
int left(int p) { return p; }

int right(int p) {
        return (p + 1) % 5;
}
```

Helper functions (Downey's solutions)

- Philosopher `p` wishes to refer to the for on their left → call `left(p)`.
- Philosopher `p` wishes to refer to the for on their right → call `right(p)`.

The Dining Philosophers (Cont.)

- We need some **semaphore**, one for each fork: `sem_t forks[5]`.

```
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

void putforks() {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```

The `getforks()` and `putforks()` Routines (Broken Solution)

- Deadlock occur!
  - If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right.
  - Each will be stuck *holding one fork* and waiting for another, *forever*.

A Solution: Breaking The Dependency

- Change how forks are acquired.
  - Let's assume that philosopher 4 acquire the forks in a *different order*.

```
void getforks() {
    if (p == 4) {
        sem_wait(forks[right(p)]);
        sem_wait(forks[left(p)]);
    } else {
        sem_wait(forks[left(p)]);
        sem_wait(forks[right(p)]);
    }
}
```

    - There is no situation where each philosopher grabs one fork and is stuck waiting for another. **The cycle of waiting is broken**.

How To Implement Semaphores

- Build our own version of semaphores called Zemaphores

```c
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
    Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}
…
```

How To Implement Semaphores (Cont.)

```
void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

- Zemaphore does not maintain the invariant for *the value of* the semaphore.
  - The value should never be lower than zero.
  - This behavior is **easier** to implement and **matches** the current Linux implementation.

Part II. Common Currency Problems

Common Concurrency Problems

- More recent work focuses on studying other types of common concurrency bugs.
  - Take a brief look at some example concurrency problems found in real code bases.
- Focus on four major open-source applications
  - MySQL, Apache, Mozilla, OpenOffice.

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
Open Office	Office Suite	6	2
Total		**74**	**31**

Bugs In Modern Applications

Non-Deadlock Bugs

- Make up a majority of concurrency bugs.
- Two major types of non deadlock bugs:
  - Atomicity violation
  - Order violation

Atomicity-Violation Bugs

- The desired **serializability** among multiple memory accesses is *violated*.
  - Simple Example found in MySQL:
    - Two different threads access the field `proc_info` in the `struct thd`.

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

Thread2::
thd->proc_info = NULL;
```

Atomicity-Violation Bugs (Cont.)

- **Solution**: Simply add locks around the shared-variable references.

```
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);
```

Order-Violation Bugs

- The desired order between two memory accesses is flipped.
  - i.e., **A** should always be executed before **B**, but the order is not enforced during execution.
  - **Example**:
    - The code in Thread2 seems to assume that the variable `mThread` has already been *initialized* (and is not `NULL`).

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

Thread2::
void mMain(…){
     mState = mThread->State
}
```

Order-Violation Bugs (Cont.)

- **Solution**: Enforce ordering using condition variables

```
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(…){
    …
```

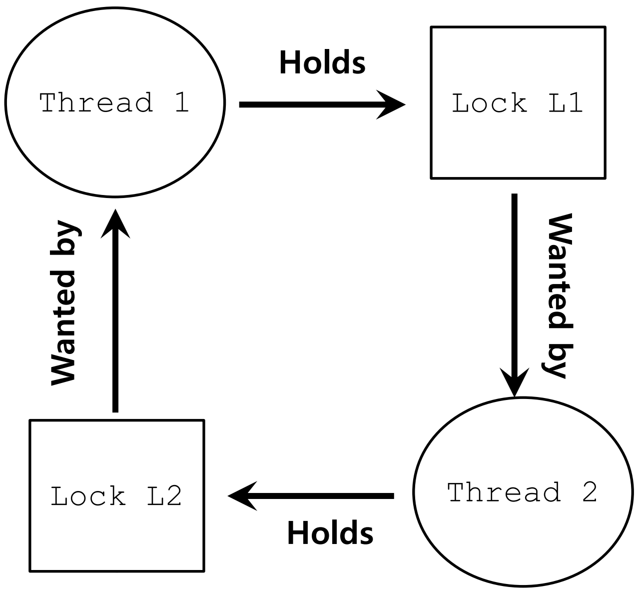
Order-Violation Bugs (Cont.)

```
        // 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

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

- The presence of a cycle
  - `Thread1` is holding a lock `L1` and waiting for another one, `L2`.
  - `Thread2` that holds lock `L2` is waiting for `L1` to be release.



Why Do Deadlocks Occur?

- Reason 1:
  - In large code bases, complex dependencies arise between components.
- Reason 2:
  - Due to the nature of encapsulation
    - Hide details of implementations and make software easier to build in a modular way.
    - Such modularity *does not mesh* well with locking.

Why Do Deadlocks Occur? (Cont.)

- **Example**: Java Vector class and the method `AddAll()`

```
Vector v1,v2;
v1.AddAll(v2);
```

  - **Locks** for both the vector being added to (`v1`) and the parameter (`v2`) *need to be acquired*.
    - The routine acquires said locks in some arbitrary order (v1 then v2).
    - If some other thread `calls v2.AddAll(v1)` at nearly the same time → We have the potential for deadlock.

Conditional for Deadlock

- Four conditions need to hold for a deadlock to occur.

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

  - If any of these four conditions are not met, **deadlock cannot occur**.

Prevention – Circular Wait

- Provide a total ordering on lock acquisition
  - This approach requires *careful design* of global locking strategies.
- **Example**:
  - There are two locks in the system (L1 and L2)
  - We can prevent deadlock by always acquiring L1 before L2.

Prevention – Hold-and-wait

- Acquire all locks at once, atomically.

```
lock(prevention);
lock(L1);
lock(L2);
…
unlock(prevention);
```

  - This code guarantees that **no untimely thread switch can occur** *in the midst of* lock acquisition.
  - **Problem**:
    - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time.
    - Decrease *concurrency*

Prevention – No Preemption

- **Multiple lock acquisition** often gets us into trouble because when waiting for one lock we are holding another.
- `trylock()`
  - Used to build a *deadlock-free*, *ordering-robust* lock acquisition protocol.
  - Grab the lock (if it is available).
  - Or, return -1: you should try again later.

```
top:
    lock(L1);
    if( tryLock(L2) == -1 ){
        unlock(L1);
        goto top;
    }
```

Prevention – No Preemption (Cont.)

- livelock
  - Both systems are running through the code sequence *over and over again*.
  - <u>Progress is not being made</u>.
  - Solution:
    - Add **a random delay** before looping back and trying the entire thing over again.

Prevention – Mutual Exclusion

- wait-free
  - Using powerful hardware instruction.
  - You can build data structures in a manner that *does not require* explicit locking.

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

Prevention – Mutual Exclusion (Cont.)

- We now wanted to atomically increment a value by a certain amount:

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

  - Repeatedly tries to update the value to *the new amount* and uses the `compare-and-swap` to do so.
  - **No lock** is acquired
  - **No deadlock** can arise
  - **livelock** is still a possibility.

Prevention – Mutual Exclusion (Cont.)

- **More complex example**: list insertion

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

  - If called by multiple threads at the "*same time*", this code has a race condition.

Prevention – Mutual Exclusion (Cont.)

- **Solution**:
  - Surrounding this code with a lock acquire and release.

```
void insert(int value){
     node_t * n = malloc(sizeof(node_t));
     assert( n != NULL );
     n->value = value ;
     lock(listlock); // begin critical section
     n->next  = head;
     head     = n;
     unlock(listlock) ;  //end critical section
}
```

  - wait-free manner using the `compare-and-swap` instruction

```
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 Avoidance via Scheduling

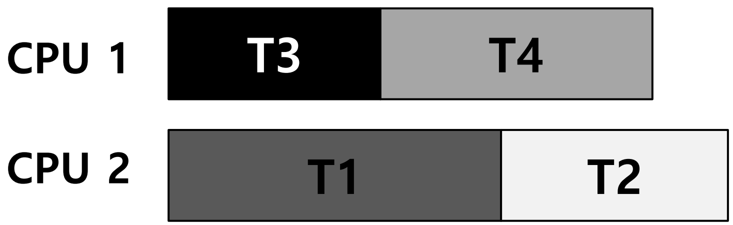
- In some scenarios deadlock avoidance is preferable.
  - **Global knowledge** is required:
    - Which locks various threads might grab during their execution.
    - Subsequently schedules said threads in a way as to guarantee no deadlock can occur.

Example of Deadlock Avoidance via Scheduling (1)

- We have two processors and four threads.
  - Lock acquisition demands of the threads:

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

  - A smart scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise.

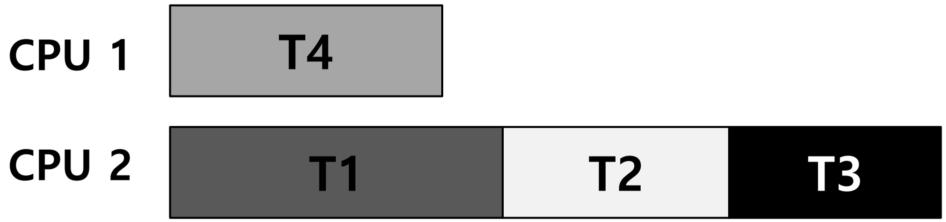


Example of Deadlock Avoidance via Scheduling (2)

- More contention for the same resources

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

  - A possible schedule that guarantees that *no deadlock* could ever occur.



    - The total time to complete the jobs is lengthened considerably.

Detect and Recover

- Allow deadlock to occasionally occur and then *take some action*.
  - **Example**: if an OS froze, you would reboot it.

- Many database systems employ *deadlock detection* and *recovery technique*.
  - A deadlock detector **runs periodically**.
  - Building a **resource graph** and checking it for cycles.
  - In deadlock, the system **need to be restarted**.

Summary

- Semaphore
  - Definition: sem_init(), sem_wait(), and sem_post()
  - Applications: Lock, Condition Variable, Producer/Consumer, Read/Write Locks, and Dinning Philoshpers
- Common Concurrency Problems
  - Non-deadlock bugs
    - Atomicity Violation and Order Violation
  - Deadlock bugs
    - Conditions of deadlock
    - Prevention
- Next: Review