Deadlock and Concurrency Bugs

ECE 469, Apr 01

Aravind Machiry

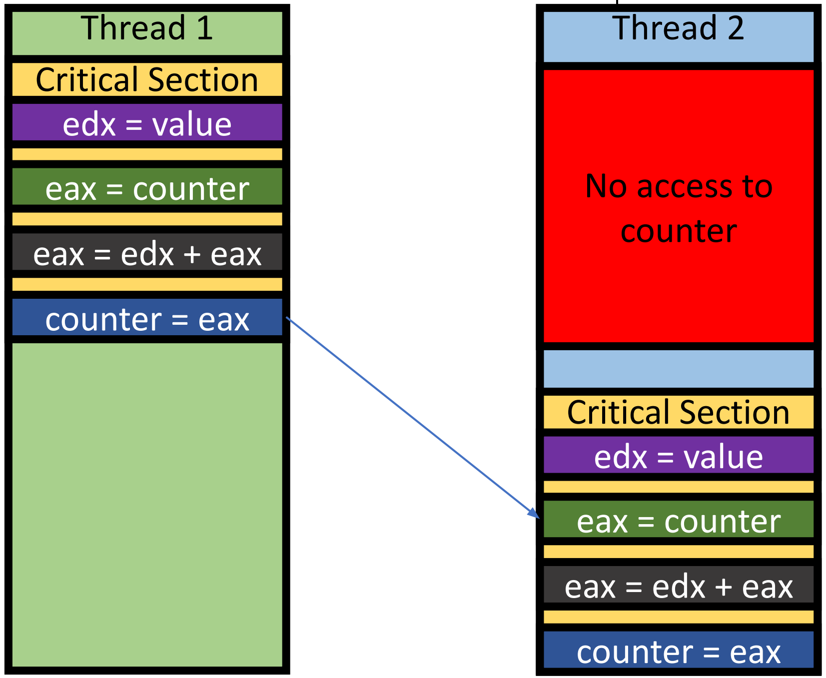
Recap: How can we prevent data races?

- *Critical section* – a section of code, or collection of operations, in which only one process shall be executing at a given time

- *Mutual exclusion (Mutex)* - mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded)

Recap: How can we prevent data races?

- Mutual Exclusion / Critical Section
  - Combine multiple instructions as a chunk
  - Let only one chunk execution runs
  - Block other executions



Recap: Mutual Exclusion through locks

- Lock
  - Prevent others enter the critical section
- Unlock
  - Release the lock, let others acquire the lock

- counter += value
  - **lock()**
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - **unlock()**

Recap: Manual Spinlock (bad_lock)

- What will happen if we implement lock
  - As bad_lock / bad_lock?

```
void *
count_bad_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        bad_lock(&lock);
        sched_yield();
        count += 1;
        bad_unlock(&lock);
    }
}
```

Critical Section

- bad_lock
  - Wait until lock becomes 0 (loops if 1)
  - And then, set lock as 1
    - Because it was 0, we can set it as 1
  - Others must wait! **Can pass this if lock=0**

Sets lock=1 to block others

- bad_unlock
  - Just set *lock as 0

Sets lock=0 to release

```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}

void
bad_unlock(volatile uint32_t *lock) {
    *lock = 0;
}
```

Recap: Why does bad_lock doesn't work?

- There is a room for race condition!

LOAD

STORE

```
mov     (%rdi),%eax
cmp     $0x1,%eax
je      0x400b60 <bad_lock>
movl    $0x1,(%rdi)
```

Race condition may happen

```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}
```

Recap: Lock using xchg

- xchg_lock
  - Use atomic ‘xchg’ instruction to
  - Load and store values atomically
  - Set value to 1, and compare ret
    - If 0, then you can acquire the lock
    - If 1, lock as 1, you must wait
- xchg_unlock
  - Use atomic ‘xchg’
  - Set value to 0
    - Do not need to check
    - You are the only thread that runs in the
    - Critical section..

```
void *
count_xchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        xchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

Critical Section

```
void
xchg_lock(volatile uint32_t *lock) {
    while(xchg(lock, 1));
}

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

Recap: Lock using test and set

- tts_xchg_lock
- Algorithm
  - Wait until lock becomes 0
  - After lock == 0
    - xchg (lock, 1)
    - This only updates lock = 1 if lock was 0

- Why xchg, why not *lock = 1 directly
  - while and xchg are not atomic
  - Load/Store must happen at
    - The same time!

```
void *
count_tts_xchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        tts_xchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

Critical Section

```
void
tts_xchg_lock(volatile uint32_t *lock) {
    while (1) {
        while(*lock == 1);
        if (xchg(lock, 1) == 0) {
            break;
        }
    }
}
```

Recap: Lock using cmpxchg_lock

- Cmpxchg_lock
  - Use cmpxchg to set lock = 1
    - Do not update if lock == 1
    - Only write 1 to lock if lock == 0

```
void *
count_cmpxchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        cmpxchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

Critical Section

- Xchg_unlock
  - Use xchg_unlock to set lock = 0
  - Because we have 1 writer and
  - This always succeeds…

```
void
cmpxchg_lock(volatile uint32_t *lock) {
    while(cmpxchg(lock, 0, 1));
}

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

Recap: Using hardware features smartly

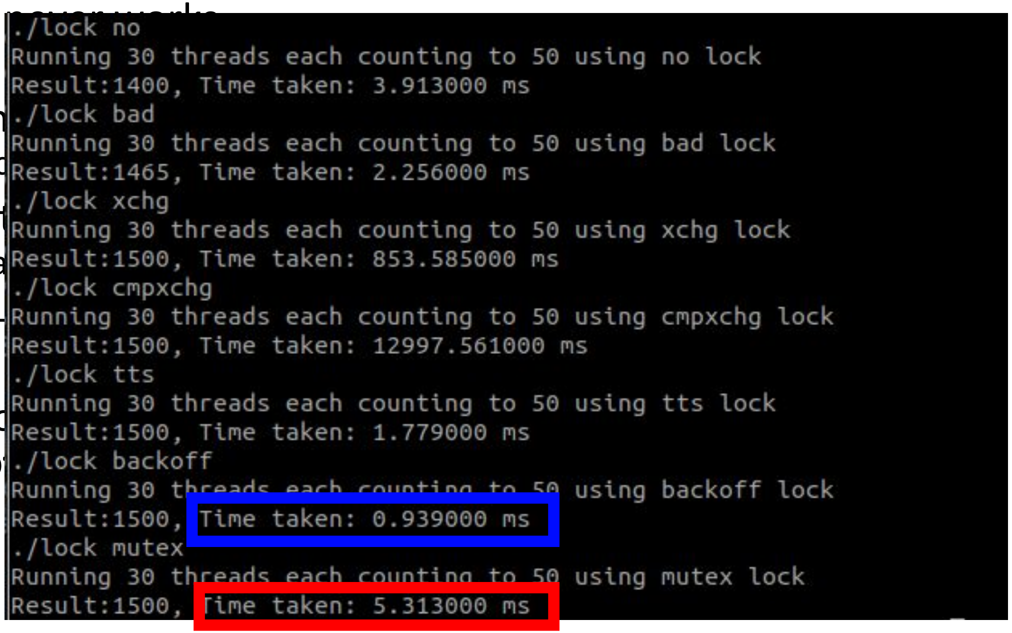
- backoff_cmpxchg_lock(lock)
- Try cmpxchg
  - If succeeded, acquire the lock.
  - If failed
    - Wait 1 cycle (pause) for 1st trial
    - Wait 2 cycles for 2nd trial
    - Wait 4 cycles for 3rd trial
    - …
    - Wait 65536 cycles for 17th trial..
    - Wait 65536 cycles for 18th trial..

```
void
backoff_cmpxchg_lock(volatile uint32_t *lock) {
    uint32_t backoff = 1;
    while(cmpxchg(lock, 0, 1)) {
        for (int i=0; i<backoff; ++i) {
            __asm volatile("pause");
        }
        if (backoff < 0x10000) {
            backoff <<= 1;
        }
    }
}
```

- https://en.wikipedia.org/wiki/Exponential_backoff

Recap: Summary

- Mutex is implemented with Spinlock
  - Waits until lock == 0 with a while loop (that's why it's called spin)
- Naïve code implementation
  - Load/Store must be atomic
- xchg is a "test and set" atom
  - Consistent, however, many c
- Lock cmpxchg is a "test and t
  - But Intel implemented this a
- We can implement test-and-
  - Faster!
- We can also implement expo
  - Much faster! **Faster Than p**



Other synchronization primitives

- We may want to have more than one thread/process to execute at same time

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

How many producers/consumers can run at a given time?

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

What we want!

- To be more efficient we want to be able to allow **more than one producer/consumer, i.e., equal to the number of items that can be inserted into/removed from the pool**

Producer

```
while (1) {

        produce an item;

        lock();
        insert(item to pool);
        unlock();
}
```

Consumer

```
While (1) {

        lock();
        remove(item from pool);
        unlock();

    consume the item;
}
```

Semaphore

A semaphore is like an **integer**, with three differences:

When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are **allowed to perform** are **increment** (increase by one) and **decrement** (decrease by one). *You cannot read the current value of the semaphore.*

When a thread **decrements** the semaphore, if the **result is negative**, the **thread blocks itself** and cannot continue until another thread increments the semaphore.

When a thread **increments** the semaphore, if there are **other threads waiting, one of the waiting threads gets unblocked**.

Semaphore operations

```
wait(S) {
  while (S<=0);
   S--;
}
```

```
signal(S) {
       S++;
}
```

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;
    wait(EMPTY);
    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;
    wait(EMPTY);
    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    wait(FULL);
    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;
    wait(EMPTY);
    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    wait(FULL);
    lock();
    remove(item from pool);
    unlock();
    signal(EMPTY);
    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Is Semaphore good for producers/consumers?

Need to know the size of buffer!

How to accommodate dynamic pool size?

Revising Producers/consumers

Producer

```
while (1) {

    produce an item;
    wait(EMPTY);
    lock(m);
    insert(item to pool);
    unlock(m);
    signal(FULL);
}
```

Consumer

```
While (1) {

    wait(FULL);
    lock(m);
    remove(item from pool);
    unlock(m);
    signal(EMPTY);
    consume the item;
}
```

Revising Producers/consumers

Producer

```
while (1) {

    produce an item;
    wait till there is space in pool
    lock(m);
    insert(item to pool);
    unlock(m);
    tell a waiting consumer
}
```

Consumer

```
While (1) {

    wait till there is an item in pool
    lock(m);
    remove(item from pool);
    unlock(m);
    tell a producer that item has been removed
    consume the item;
}
```

Revising Producers/consumers

Producer

```
while (1) {

    produce an item;
    if (!pool.has_space) {
        We need to wait for consumer
    }
    lock(m);
    insert(item to pool);
    unlock(m);
    tell a waiting consumer
}
```

Consumer

```
While (1) {

    if (pool.is_empty) {
        We need to wait for producer
    }
    lock(m);
    remove(item from pool);
    unlock(m);
    tell a waiting producer
    consume the item;
}
```

What's wrong?

Producer

```
while (1) {

    produce an item;
    if (!pool.has_space) {
        We need to wait for consumer
    }
    lock(m);
    insert(item to pool);
    unlock(m);
    tell a waiting consumer
}
```

Consumer

```
While (1) {

    if (pool.is_empty) {
        We need to wait for producer
    }
    lock(m);
    remove(item from pool);
    unlock(m);
    tell a waiting producer
    consume the item;
}
```

What's wrong?

Producer

Data Race

```
while (1) {

    produce an item;
    if (!pool.has_space) {
        We need to wait for consumer
    }
    lock(m);
    insert(item to pool);
    unlock(m);
    tell a waiting consumer
}
```

Consumer

```
While (1) {

    if (pool.is_empty) {
        We need to wait for producer
    }
    lock(m);
    remove(item from pool);
    unlock(m);
    tell a waiting producer
    consume the item;
}
```

Lets move the lock up!

Producer

```
while (1) {

    produce an item;
    lock(m);
    if (!pool.has_space) {
        We need to wait for consumer
    }
    insert(item to pool);
    unlock(m);
    tell a waiting consumer
}
```

Consumer

```
While (1) {

    lock(m);
    if (pool.is_empty) {
        We need to wait for producer
    }
    remove(item from pool);
    unlock(m);
    tell a waiting producer
    consume the item;
}
```

What's wrong?

Producer may never get to run

Producer

```
while (1) {

   produce an item;
   lock(m);
   if (!pool.has_space) {
        We need to wait for consumer
   }
   insert(item to pool);
   unlock(m);
   tell a waiting consumer
}
```

Consumer

```
While (1) {

   lock(m);
   if (pool.is_empty) {
        We need to wait for producer
   }
   remove(item from pool);
   unlock(m);
   tell a waiting producer
   consume the item;
}
```

Lets release the lock and wait!

Producer

```
while (1) {

    produce an item;
    lock(m);
    if (!pool.has_space) {
        unlock(m);
        We need to wait for consumer
        lock(m);
    }
    insert(item to pool);
    unlock(m);
    tell a waiting consumer
}
```

Consumer

```
While (1) {

    lock(m);
    if (pool.is_empty) {
        unlock(m);
        We need to wait for producer
        lock(m);
    }
    remove(item from pool);
    unlock(m);
    tell a waiting producer
    consume the item;
}
```

Release, wait and reacquire

Producer

```
while (1) {

    produce an item;
    lock(m);
    if (!pool.has_space) {
        unlock(m);
        We need to wait for consumer
        lock(m);
    }
    insert(item to pool);
    unlock(m);
    tell a waiting consumer
}
```

Release lock, waiting for a condition and acquire lock

Consumer

```
While (1) {

    lock(m);
    if (pool.is_empty) {
        unlock(m);
        We need to wait for producer
        lock(m);
    }
    remove(item from pool);
    unlock(m);
    tell a waiting producer
    consume the item;
}
```

Condition Variable (CV)

```
CV full; full->lock = m;
CV empty; empty->lock = m;
```

Producer

```
while (1) {

    produce an item;
    lock(m);
    if (!pool.has_space) {
        wait(full);
    }
    insert(item to pool);
    signal(empty);
    unlock(m);
}
```

```
unlock(m);
We need to wait for consumer
lock(m);
```

Consumer

```
While (1) {

    lock(m);
    if (pool.is_empty) {
        wait(empty);
    }
    remove(item from pool);
    signal(full);
    unlock(m);
    consume the item;
}
```

Condition Variable operations

```
wait(S) {
 unlock(s->lock);
 block and add into s->queue
 lock(s->lock);
}
```

```
signal(S) {
 unlock(s->lock);
 p = remove process from s->queue
 unblock process p
 lock(s->lock);
}
```

Condition Variables

- Wait (condition)
  - Block on “condition”

- Signal (condition)
  - Wakeup one or more processes blocked on “condition”

- Conditions are like semaphores but:
  - signal is no-op if none blocked
  - There is no counting!

CVs for Ordering - Order 1

```
Thread 1::
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    ...
}

Thread 2::
void mMain(...) {
     ...
     mState = mThread->State;
     ...
}
```

CVs for Ordering - Order 2

```
Thread 2::
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

```
Thread 1::
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    ...
}
```

CVs for Ordering - Order 2

```
Thread 2::
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

Not Initialized…

```
Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);
   ...
}
```

CVs for Ordering

- Use locks and conditional variables to force a specific ordering…

```
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);
    ...
}

Thread 2::
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;
     ...
}
```

CVs for Ordering

- Use locks and conditional variables to force a specific ordering…

Waits for condition..

```
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);
    ...
}

Thread 2::
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;
    ...
}
```

CVs for Ordering

- Use locks and conditional variables to force a specific ordering…

```
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);
    ...
}

Thread 2::
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;
    ...
}
```

Sends Signal.. (`pthread_cond_signal(&mtCond);`)

Waits for condition.. (`pthread_cond_wait(&mtCond, &mtLock);`)

Wait free synchronization

- Can we ensure our programs run fine in presence of possible race conditions without explicitly using synchronization primitives (or waiting for critical section)?
  - Root cause of Data races:
    - Hint: Concurrent use of shared data.
      - Can we make this safe?

Wait free synchronization

- Design data structures in a way that allows safe concurrent accesses
  - no mutual exclusion (lock acquire & release) necessary
  - no possibility of deadlock

- Approach: use a single *atomic* operation to
  - commit all changes
    - move the shared data structure from one consistent state to another

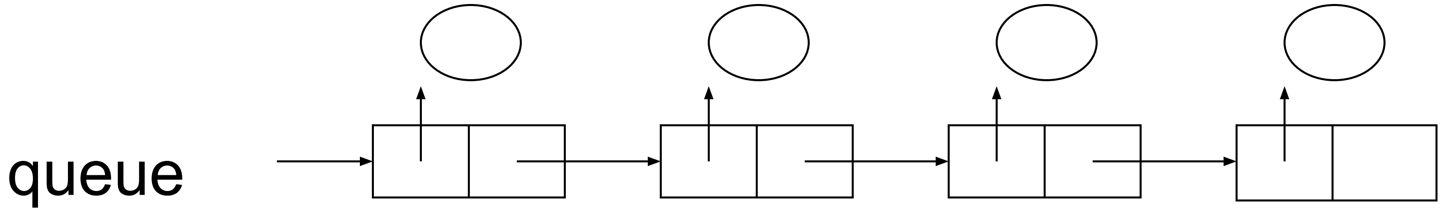
Simple queue insertion

```
QElem *queue;


void Insert(item) {
   QElem *new = malloc(sizeof(QElem));
   new->item = item;
   new->next = queue;
   queue = new;
}
```

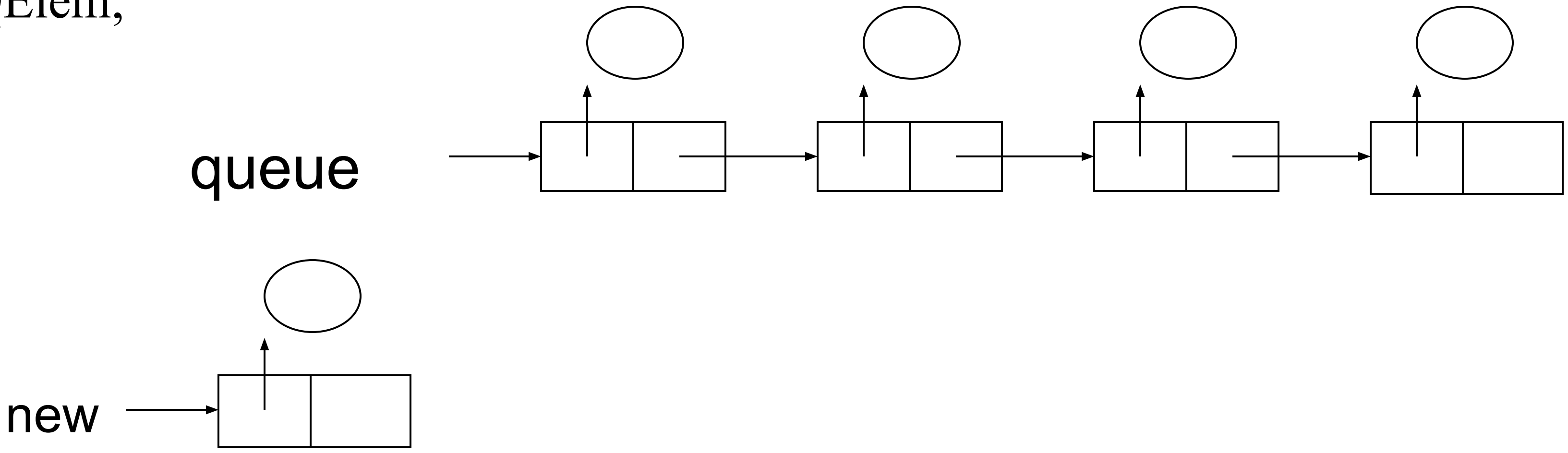
Simple queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



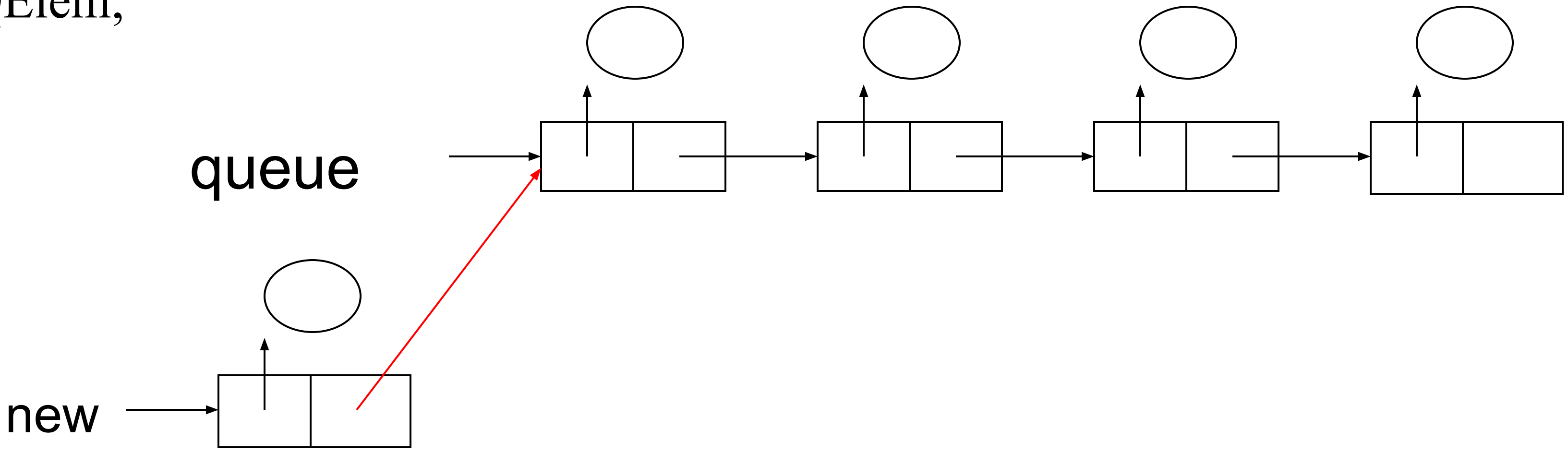
Simple queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



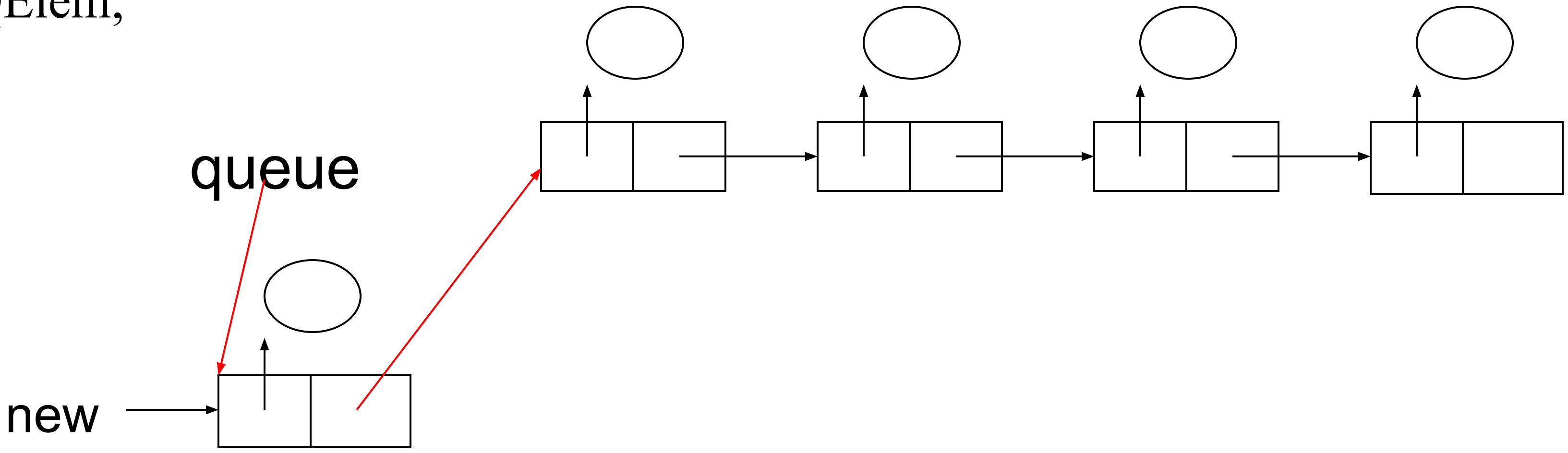
Simple queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



Simple queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



Possible data races?

```
QElem *queue;


void Insert(item) {
   QElem *new = malloc(sizeof(QElem));
   new->item = item;
   new->next = queue;
   queue = new;
}
```

Possible data races?

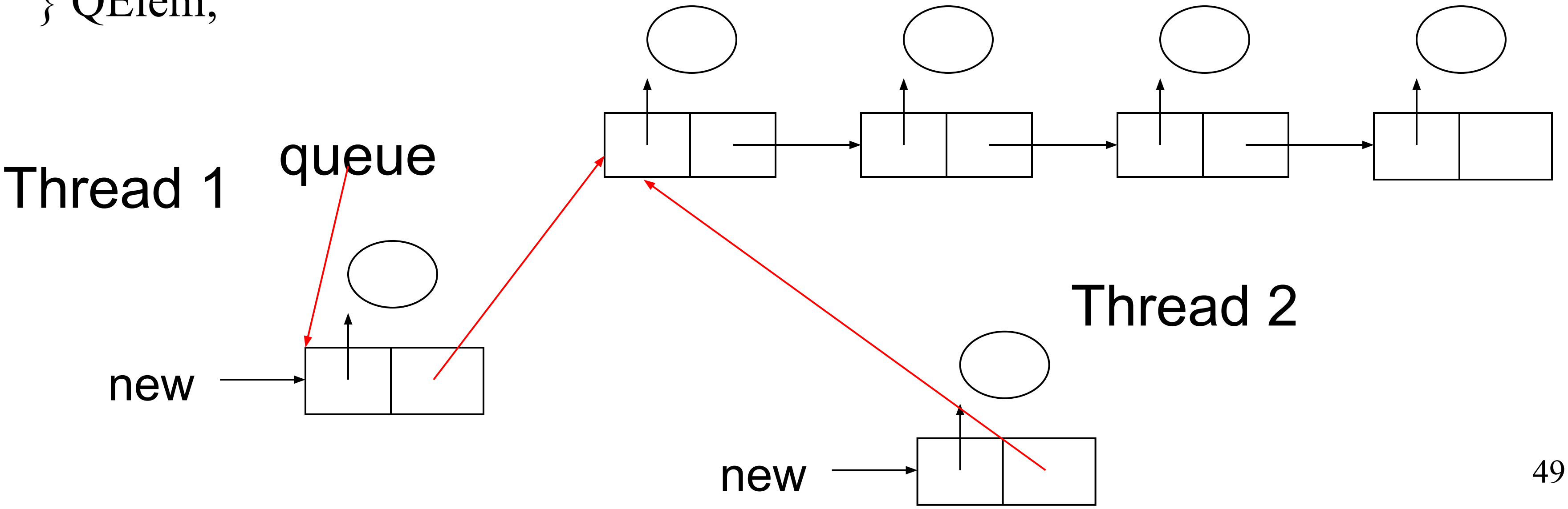
```
QElem *queue;


void Insert(item) {
  QElem *new = malloc(sizeof(QElem));
  new->item = item;
  new->next = queue;
  queue = new;
}
```

Data race

Simple queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



Simple queue insertion with xchg

```
QElem *queue;


void Insert(item) {
  QElem *new = malloc(sizeof(QElem));
  new->item = item;
  do {
        new->next = queue;
  } while (xchg(&queue, new) != new->next);
}
```

Wait free synchronization

- Example only works for simple data structures where changes can be committed with *one store instruction*

- Complex data structures need synchronization

Concurrency Bugs

Defects that occur because of not using or improperly using synchronization primitives.

- TOCTOU:
    - Time of check to time of use

TOCTOU

```
Thread 1::
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}

Thread 2::
thd->proc_info = NULL;
```

TOCTOU

Read

```
Thread 1::
if (thd->proc_info) {
   ...
   fputs(thd->proc_info, ...);
   ...
}

Thread 2::
thd->proc_info = NULL;
```

Write!

TOCTOU

Read

```
Thread 1::
if (thd->proc_info) {
   ...
   fputs(thd->proc_info, ...);
   ...
}

Thread 2::
thd->proc_info = NULL;
```

Write!

Time-of-check-to-time-of-use bug

TOCTTOU

TOCTOU

Read

```
Thread 1::
if (thd->proc_info) {   Time of check
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

Write!

Time-of-check-to-time-of-use bug

TOCTTOU

TOCTOU

Read

```
Thread 1::
if (thd->proc_info) {
   ...
   fputs(thd->proc_info, ...);
   ...
}

Thread 2::
thd->proc_info = NULL;
```

Time of check (line `if (thd->proc_info) {`)

Time of use (line `fputs(thd->proc_info, ...);`)

Write!

Time-of-check-to-time-of-use bug

TOCTTOU

TOCTOU

```
Thread 1::
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}

Thread 2::
thd->proc_info = NULL;
```

thd_proc_info was not NULL

TOCTOU

```
Thread 1::
if (thd->proc_info) {
   ...
   fputs(thd->proc_info, ...);
   ...
}

Thread 2::
thd->proc_info = NULL;
```

thd_proc_info was not NULL

thd_proc_info becomes NULL

TOCTOU

```
Thread 1::
if (thd->proc_info) {
   ...
   fputs(thd->proc_info, ...);
   ...
}

Thread 2::
thd->proc_info = NULL;
```

thd_proc_info was not NULL

Uh-oh

...

thd_proc_info becomes NULL

Concurrency Bugs

- Deadlock:
  - Two or more threads are waiting for the other to take some actions thus neither make any progress

Deadlocks

- Two or more threads are waiting for the other to take some actions thus neither make any progress

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

Deadlocks

- Two or more threads are waiting for the other to take some actions thus neither make any progress

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

Deadlocks

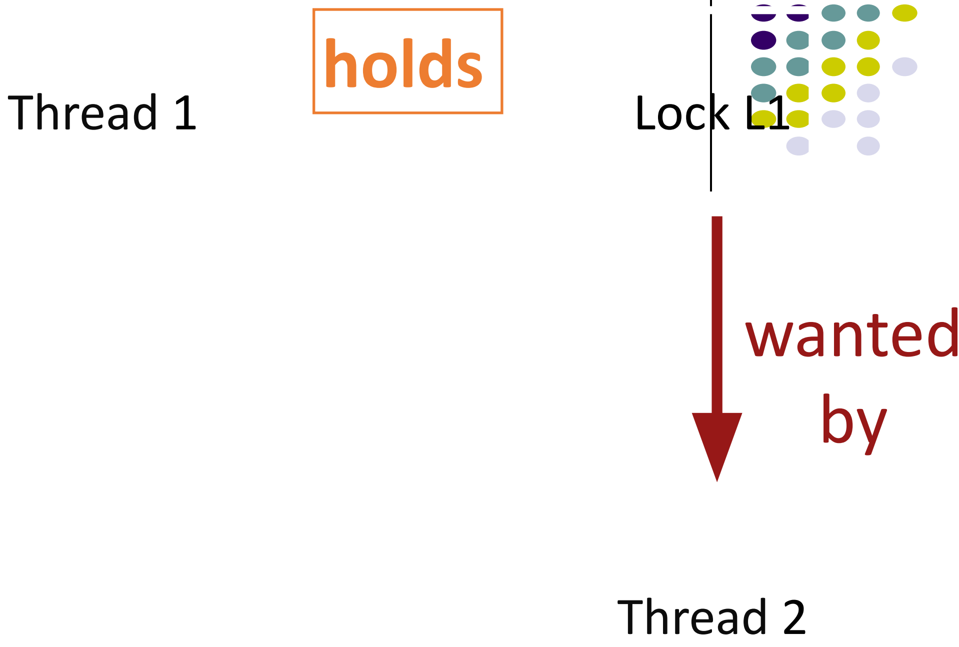
Thread 1 **holds** Lock L1

- Two or more threads are waiting for the other to take some actions thus neither make any progress

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

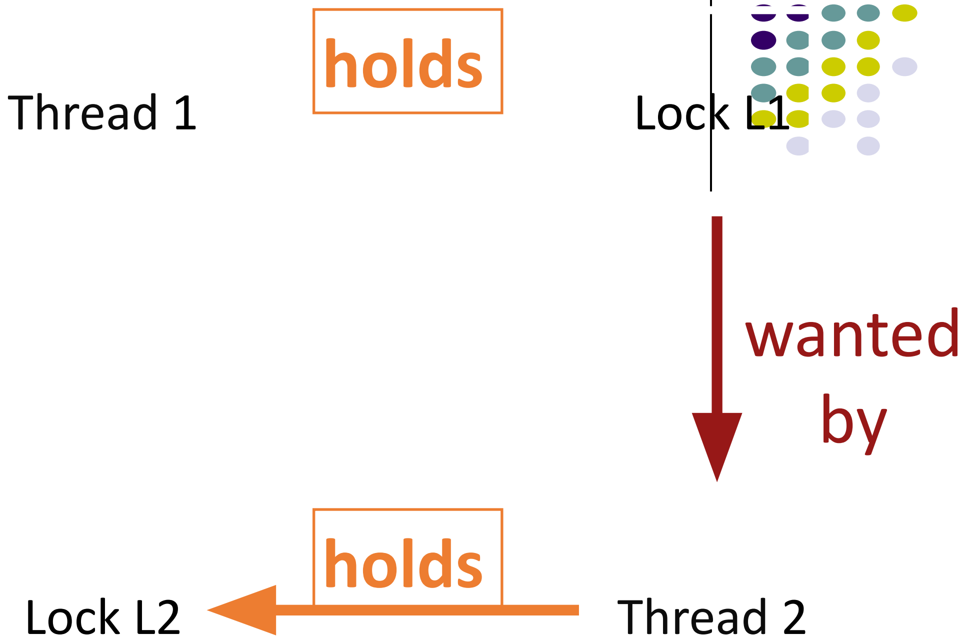
Deadlocks

- Two or more threads are waiting for the other to take some actions thus neither make any progress




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

Deadlocks

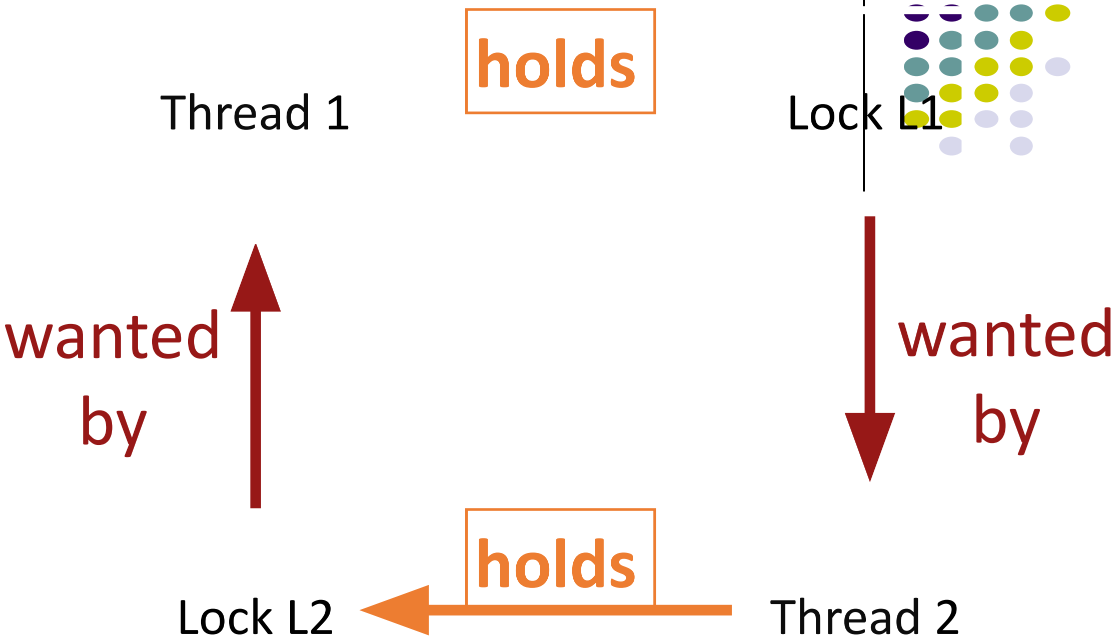


- Two or more threads are waiting for the other to take some actions thus neither make any progress

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

Deadlocks

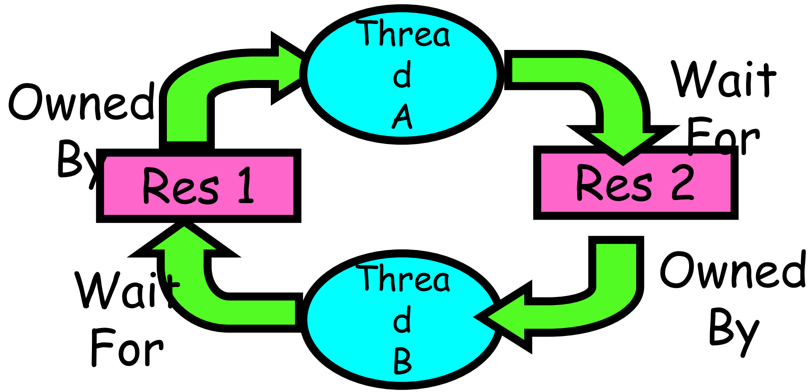
- Two or more threads are waiting for the other to take some actions thus neither make any progress




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

Starvation v/s Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



Starvation v/s Deadlock

- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn’t have to)
  - Deadlock can’t end without external intervention

Deadlocks can be hard to reason

```
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = new set_t();
    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);
}
```

Scenario 1: Any problem?

Thread 1:

rv = set_intersection(setA, setB);

Thread 2:

rv = set_intersection(setA, setB);

Scenario 1: Any problem?

Thread 1:

rv = set_intersection(setA, setB);

```
Mutex_lock(&setA->lock);
Mutex_lock(&setB->lock);

…
Mutex_unlock(&setB->lock);
Mutex_unlock(&setA->lock);
```

Thread 2:

rv = set_intersection(setA, setB);

```
Mutex_lock(&setA->lock);
Mutex_lock(&setB->lock);

…
Mutex_unlock(&setB->lock);
Mutex_unlock(&setA->lock);
```

Scenario 2: Any problem?

Thread 1:

rv = set_intersection(**setA, setB**);

Thread 2:

rv = set_intersection(**setB, setA**);

Scenario 2: Any problem?

Thread 1:

rv = set_intersection(**setA**, **setB**);

```
Mutex_lock(&setA->lock);
Mutex_lock(&setB->lock);
```

Thread 2:

rv = set_intersection(**setB**, **setA**);

```
Mutex_lock(&setB->lock);
Mutex_lock(&setA->lock);
```

Deadlock!

Modelling Deadlock

- Resources
  - Resource types $R_1, R_2, \ldots, R_m$
    - *CPU cycles, memory space, I/O devices, mutex*
  - Each resource type R_i has W_i instances
  - *Preemptable:* can be taken away by scheduler, e.g. CPU
  - *Non-preemptable:* cannot be taken away, to be released voluntarily, e.g., mutex, disk, files, ...

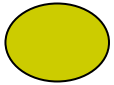
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Modelling Deadlock: Resource allocation graph

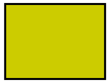
- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Modelling Deadlock

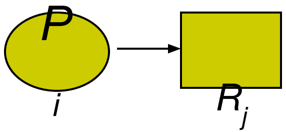
- Process



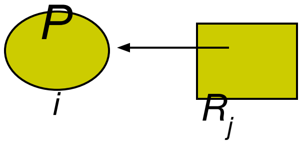
- Resource type



- P_i requests instance of R_j

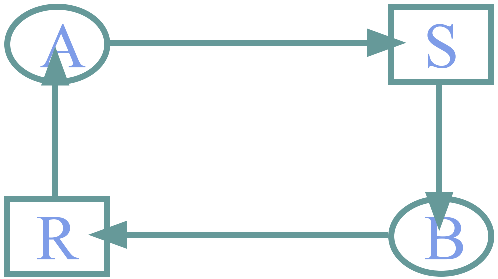


- P_i is holding an instance of R_j

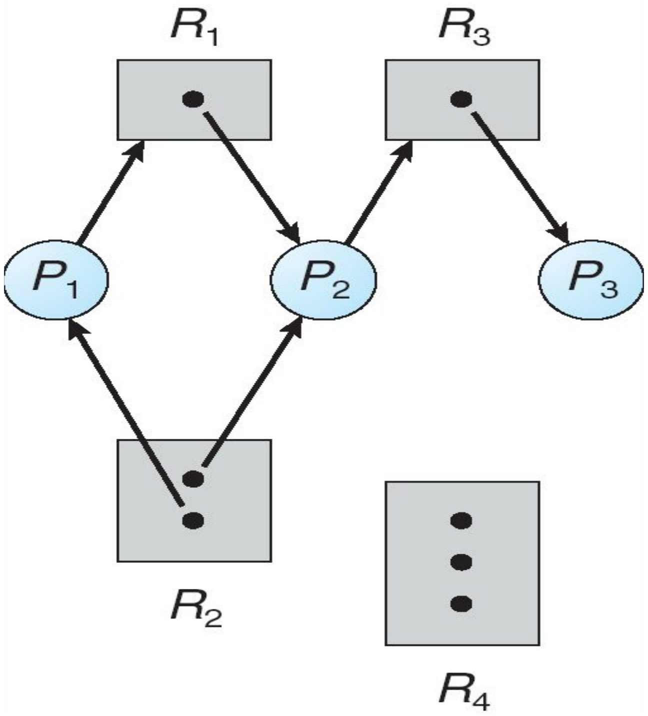


Cycle in resource allocation graph!?

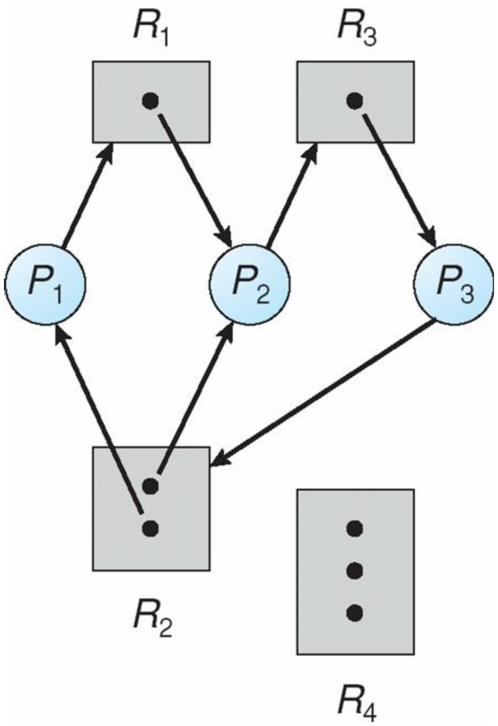
- What happens if there is a cycle in the resource allocation graph?



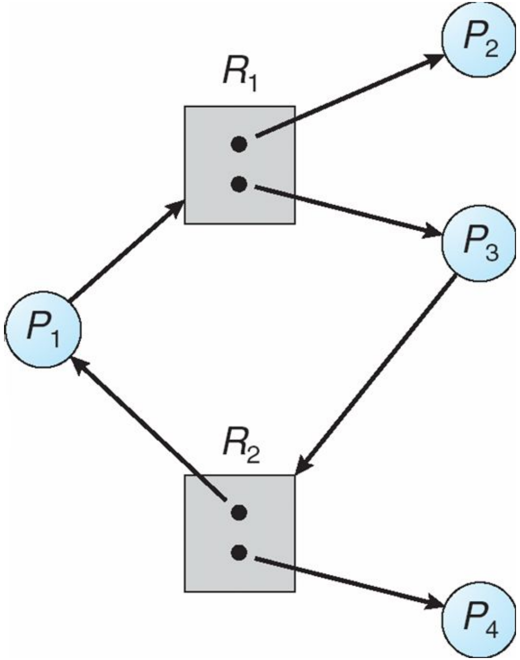
Is there a deadlock?



Is there a deadlock?



Is there a deadlock?



Modelling Deadlocks Using Resource allocation graphs

- If graph contains no cycles ⇒ no deadlock

- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Necessary conditions for a deadlock

- *Mutual exclusion*
  - Each resource instance is assigned to exactly one process
- *Hold and wait*
  - Holding at least one and waiting to acquire more
- *No preemption*
  - Resources cannot be taken away
- *Circular chain of requests*

Necessary conditions for a deadlock

- *Mutual exclusion*
  - Each resource instance is assigned to exactly one process
- *Hold and wait*
  - Holding at least one and waiting to acquire more
- *No preemption*
  - Resources cannot be taken away
- *Circular chain of requests*

Resource nature

Program behavior

Necessary conditions for a deadlock

- *Mutual exclusion*
  - Each resource instance is assigned to exactly one process
- *Hold and wait*
  - Holding at least one and waiting to acquire more
- *No preemption*
  - Resources cannot be taken away
- *Circular chain of requests*

Resource nature

Program behavior

Eliminating ***any*** condition eliminates deadlock!

Handling deadlock

1. Ignore the problem
   - It is user's fault
   - used by most operating systems, including UNIX
2. Detection and recovery (by OS)
   - Fix the problem afterwards
3. Dynamic avoidance (by OS & programmer)
   - Careful allocation
4. Prevention (by programmer & OS)
   - Negate one of the four conditions

2. Detect and Recovery

- Programmer does nothing
- Allow system to enter deadlock state
- Run some detection algorithm
  - E.g., build a resource graph to check for cycles
- Try to recover somehow
  - E.g., reboot the machine

3. Dynamic Avoidance

Definition:

An algorithm that is run by the OS whenever a process requests resources, the algorithm avoids deadlock by denying or postponing the request

if

it finds that accepting the request could put the system in an unsafe state (one where deadlock could occur).

3. Dynamic Avoidance

- Requirement:
  - each process <u>declares</u> the *maximum number* of resources of each type it *may* need

- Key idea:
  - The deadlock-avoidance algorithm <u>dynamically</u> examines the <u>resource-allocation state</u> to ensure there can never be a deadlock condition
  - *No matter what future requests will be*

3. Dynamic Avoidance

- Needs to know the entire set of tasks that must be run and the locks that they need
- Reduce concurrency
- Not used widely in practice
  - E.g., used in embedded systems

4. Preventing deadlock

- *Mutual exclusion*
  - Each resource instance is assigned to exactly one process
- *Hold and wait*
  - Holding at least one and waiting to acquire more
- *No preemption*
  - Resources cannot be taken away
- *Circular chain of requests*

Eliminating ***any*** condition eliminates deadlock!

Eliminating Circular Wait

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

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

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

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

Eliminating Circular Wait

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

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

Lock variable is mostly a pointer, then provide a correct order of having a lock

e.g.,

```
if(l1 > l2)  {
        Mutex_lock(l1);
        Mutex_lock(l2);
}
else {
        Mutex_lock(l2);
        Mutex_lock(l1);
}
```

Summary

- Need to be careful while using synchronization primitives
- Concurrency bugs: improper use of synchronization primitives