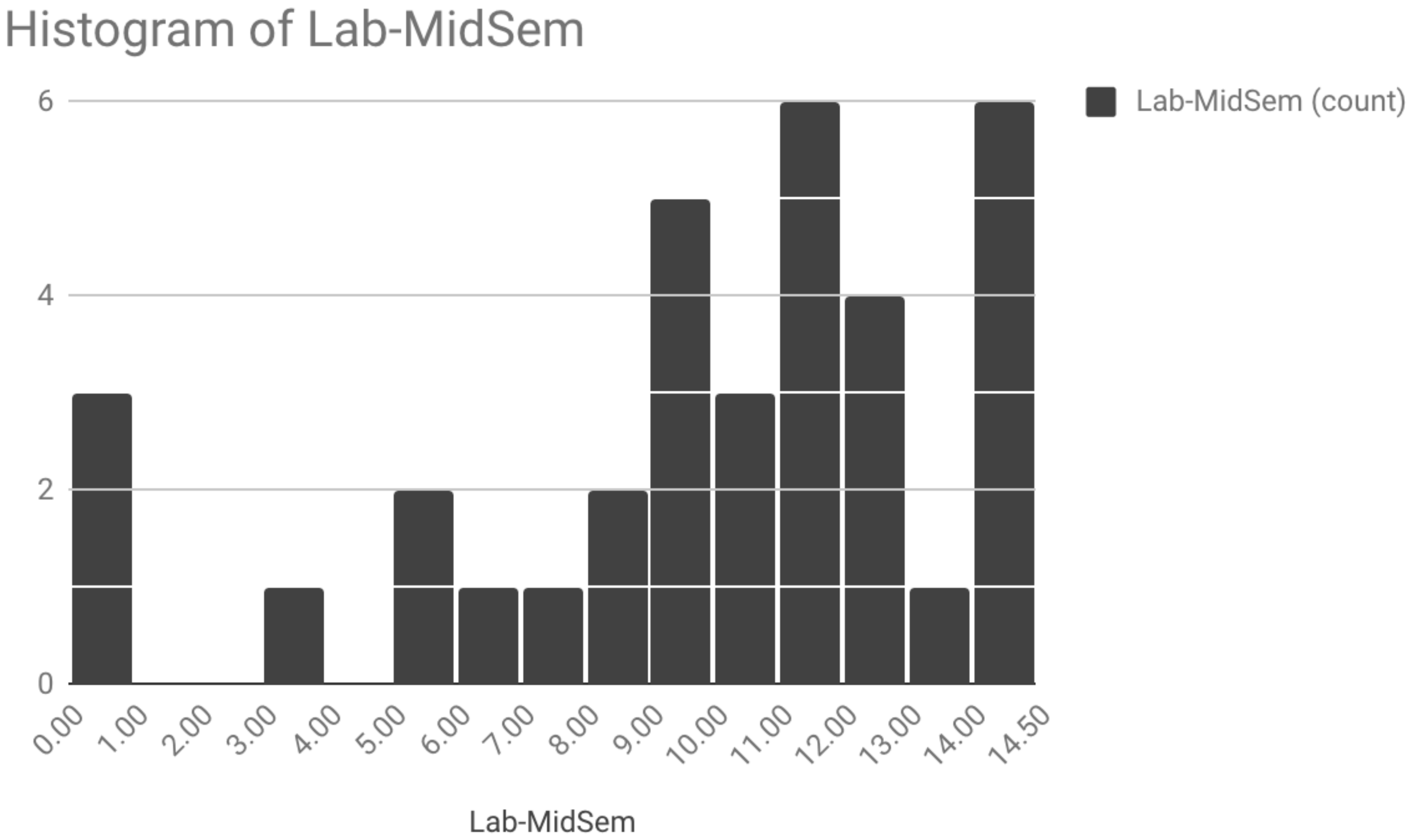
Operating Systems

Lecture 26: Semaphores Revision + Common Concurrency Problems

Nipun Batra
Nov 1, 2018


Histogram of Lab-MidSem
Lab-MidSem (count)
0
2
4
6
0.00
1.00
2.00
3.00
4.00
5.00
6.00
7.00
8.00
9.00
10.00
11.00
12.00
13.00
14.00
14.50
Lab-MidSem

Administrative

- Feedback from lab quiz?
  - How to run it better?
  - Automated checkers?
    - How?
- Assignment deadline postponed to 6th noon
- Quiz 3 on Mon, 12th November?
  - Will give the syllabus in writing, no additional questions entertained
- Project grading:
  - Second round of project grading:
    - 12th and 16th Nov
  - Third and final round:
    - 19th and 23rd Nov

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
  - Assumes lock is held when wait() is called
  - Puts caller to sleep + atomically releases lock
  - When awoken, reacquires lock before returning
- **signal (cond_t *cv)**
  - Wake a single waiting thread
  - If there is no waiting thread, just return, do nothing

Condition Variables **wait** Pseudocode

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State
  - Move to Ready State
  - pthread_mutex_lock(lock)

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
    - Assume lock is held initially
    - pthread_mutex_unlock(lock)
    - block_on_condition(cv)
    - Move from Ready to Waiting/Blocked State
    - Move to Ready State
    - pthread_mutex_lock(lock)

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State
  - Move to Ready State
  - pthread_mutex_lock(lock)

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State
  - Move to Ready State
  - pthread_mutex_lock(lock)

Atomic

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State

Atomic

  - Move to Ready State

Signal

  - pthread_mutex_lock(lock)

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State
  - Move to Ready State
  - pthread_mutex_lock(lock)

Atomic

Signal

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State
  - Move to Ready State
  - pthread_mutex_lock(lock)

Atomic

Signal

When does wait return?

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State
  - Move to Ready State
  - pthread_mutex_lock(lock)

Atomic

Signal

When does wait return?
When lock has been acquired.

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State
  - Move to Ready State
  - pthread_mutex_lock(lock)

Atomic

Signal

When does wait return?
When lock has been acquired.
What if some other thread holds the lock — block!

Condition Variables **wait** Pseudocode

Which of these are atomic?

- **wait (cond_t *cv, mutex_t *lock)**
  - Assume lock is held initially
  - pthread_mutex_unlock(lock)
  - block_on_condition(cv)
  - Move from Ready to Waiting/Blocked State
  - Move to Ready State
  - pthread_mutex_lock(lock)

Atomic

Signal

When does wait return?
When lock has been acquired.
What if some other thread holds the lock — block!

Why do we need to lock again?

Exercise: order using condition variables
Correct Solution

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    thread_join()
    printf("parent: end\n");
    return 0; }
```

Exercise: order using condition variables Correct Solution

```
void thread_exit {
    mutex_lock(&m)
    Done  = 1
    cond_signal(&c)
    mutex_unlock(&m)
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
      Pthread_create(&c, NULL, child, NULL); // create child
      thread_join()
      printf("parent: end\n");
      return 0; }
```

Exercise: order using condition variables
Correct Solution

```
void thread_exit {
    mutex_lock(&m)
    Done  = 1
    cond_signal(&c)
    mutex_unlock(&m)
```

```
void thread_join {
    mutex_lock(&m)              //w
    while (done==0)             //x
        cond_wait(&c, &m)       //y
    mutex_unlock(&m) }          //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    thread_join()
    printf("parent: end\n");
    return 0; }
```

Exercise: Build a lock using semaphores

```
sem_t m;
sem_init(&m, 0, 1);

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

Refresher Notes

```
int sem_wait(sem_t *s) {
  s->value -= 1
    wait if s->value <0
}
```

```
int sem_post(sem_t *s) {
    s->value += 1
    wake one waiting thread if any
}
```

Value	Thread 0	State	Thread 1	State	T2	State

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running
-2	call sem_post()	Running		sleeping	(sem<0) -> Sleep	Sleeping

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running
-2	call sem_post()	Running		sleeping	(sem<0) -> Sleep	Sleeping
			Switch T1			

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running
-2	call sem_post()	Running		sleeping	(sem<0) -> Sleep	Sleeping
			Switch T1			
-1	Increment sem					

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running
-2	call sem_post()	Running		sleeping	(sem<0) -> Sleep	Sleeping
			Switch T1			
-1	Increment sem					
-1	wake(T1)	Running		Ready		

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running
-2	call sem_post()	Running		sleeping	(sem<0) -> Sleep	Sleeping
			Switch T1			
-1	Increment sem					
-1	wake(T1)	Running		Ready		
-1	sem_post() returns	Running		Ready		

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running
-2	call sem_post()	Running		sleeping	(sem<0) -> Sleep	Sleeping
			Switch T1			
-1	Increment sem					
-1	wake(T1)	Running		Ready		
-1	sem_post() returns	Running		Ready		
-1	Interrupt; Switch → T1	Ready		Running		

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running
-2	call sem_post()	Running		sleeping	(sem<0) -> Sleep	Sleeping
			Switch T1			
-1	Increment sem					
-1	wake(T1)	Running		Ready		
-1	sem_post() returns	Running		Ready		
-1	Interrupt; Switch → T1	Ready		Running		

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call sem_wait()	Running		Ready		Ready
0	sem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call sem_wait()	Running		Ready
-1		Ready	decrement sem	Running		Ready
-1		Ready	(sem < 0)→sleep	sleeping		Ready
-1		Running	Switch → T2	sleeping		Running
-1					Call sem_wait()	Running
-2	(crit sect: end)	Running		sleeping	Decrement sem	Running
-2	call sem_post()	Running		sleeping	(sem<0) -> Sleep	Sleeping
			Switch T1			
-1	Increment sem					
-1	wake(T1)	Running		Ready		
-1	sem_post() returns	Running		Ready		
-1	Interrupt; Switch → T1	Ready		Running		

Can T1 run before sem_post returns?

NO!

Semaphores Implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

`#include <semaphore.h>`
`sem_t s;`	

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
#include <semaphore.h>
sem_t s;
```

Our implementation

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

```

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
int sem_init(sem_t *s, int init_val) {
  s->value=init_val;
  }
```

Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
int sem_init(sem_t *s,
int init_val) {
    s->value=init_val;
    }
```

Our implementation

```
// 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);
}
```

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- <u>Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads</u>

API

```
int sem_post(sem_t *s) {
    s->value += 1
    wake one waiting thread if any
}
```

Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
int sem_post(sem_t *s) {
    s->value += 1
    wake one waiting thread if any
}
```

Atomic operation

Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- <u>Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads</u>

API

```
int sem_post(sem_t *s) {
    s->value += 1
    wake one waiting thread if any
}
```

Atomic operation

Our implementation

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

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

`int sem_wait(sem_t *s) {`
`  s->value -= 1`
`   wait if s->value <0`
`}`	

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

Our implementation

```
int sem_wait(sem_t *s) {
  s->value -= 1
   wait if s->value <0
}
```

Atomic operation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
int sem_wait(sem_t *s) {
  s->value -= 1
   wait if s->value <0
}
```

Atomic operation

Our implementation

```
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);
}
```

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
int sem_wait(sem_t *s) {
  s->value -= 1
   wait if s->value <0
}
```

Atomic operation

Our implementation

```
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);
}
```

Should it be ==0?

Value	Thread 0	State	Thread 1	State	T2	State

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping
0	(crit sect: end)	Running		sleeping	Switch —> T1	Sleeping

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping
0	(crit sect: end)	Running		sleeping	Switch —> T1	Sleeping
0	call sem_post()	Running		sleeping		

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping
0	(crit sect: end)	Running		sleeping	Switch —> T1	Sleeping
0	call sem_post()	Running		sleeping		
	Acquires lock	Running				

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping
0	(crit sect: end)	Running		sleeping	Switch —> T1	Sleeping
0	call sem_post()	Running		sleeping		
	Acquires lock	Running				
1	Increments zem	Running				

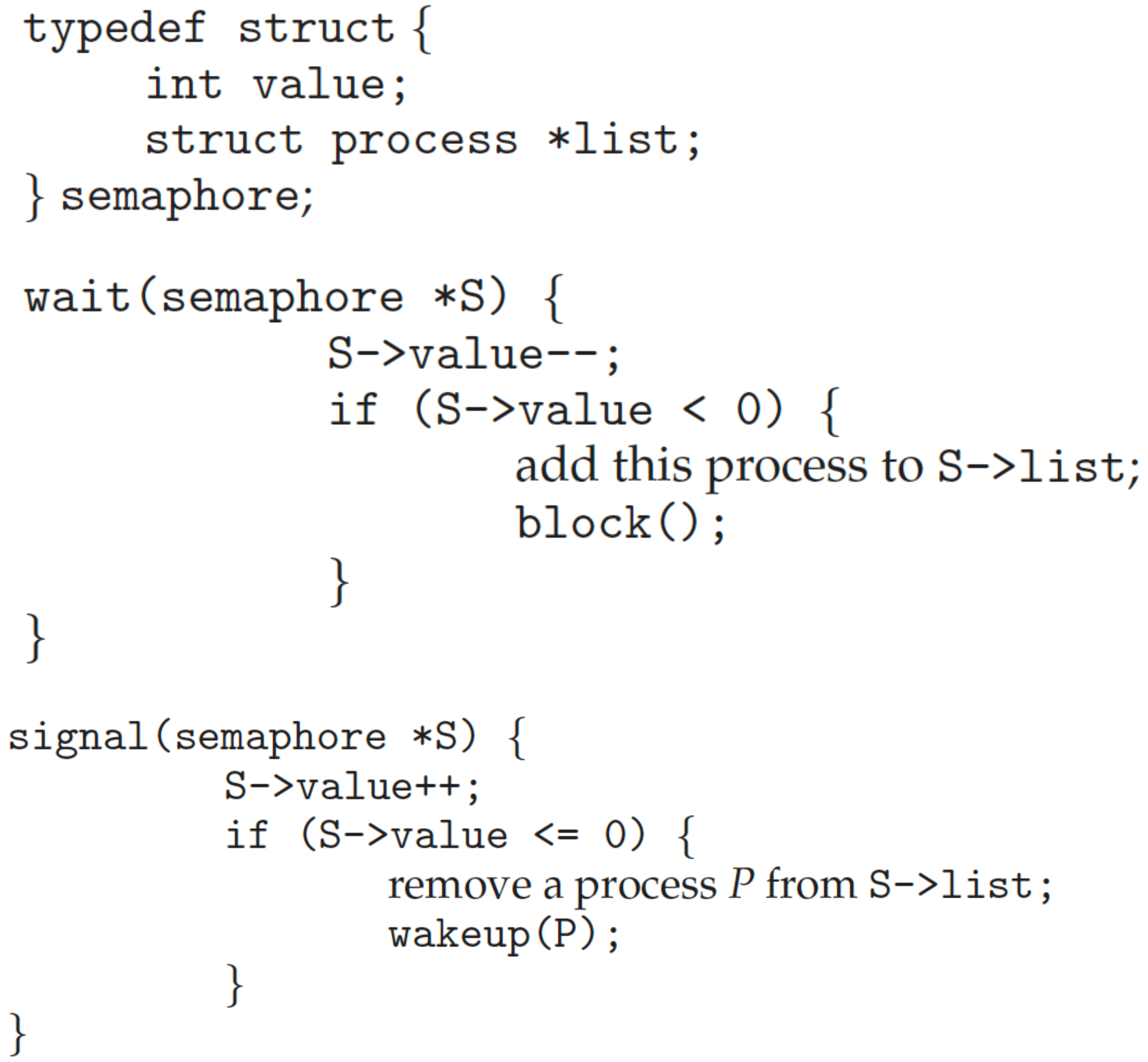
Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping
0	(crit sect: end)	Running		sleeping	Switch —> T1	Sleeping
0	call sem_post()	Running		sleeping		
	Acquires lock	Running				
1	Increments zem	Running				
1	Wakes up thread T1	Running		Ready		

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping
0	(crit sect: end)	Running		sleeping	Switch —> T1	Sleeping
0	call sem_post()	Running		sleeping		
	Acquires lock	Running				
1	Increments zem	Running				
1	Wakes up thread T1	Running		Ready		
			Condition wait will return once it gets lock	Ready		

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping
0	(crit sect: end)	Running		sleeping	Switch —> T1	Sleeping
0	call sem_post()	Running		sleeping		
	Acquires lock	Running				
1	Increments zem	Running				
1	Wakes up thread T1	Running		Ready		
			Condition wait will return once it gets lock	Ready		
1	zem_post() returns	Running		Ready		

Value	Thread 0	State	Thread 1	State	T2	State
1		Running		Ready		Ready
1	call zem_wait()	Running		Ready		Ready
0	zem_wait() retruns	Running		Ready		Ready
0	(crit set: begin)	Running		Ready		Ready
0	Interrupt; Switch → T1	Ready		Running		Ready
0		Ready	call zem_wait()	Running		Ready
0		Ready	(zem <= 0)→sleep	sleeping		Ready
0		Running	Switch → T2	sleeping		Running
0					Call zem_wait()	Running
0					(zem<0) -> Sleep	Sleeping
0	(crit sect: end)	Running		sleeping	Switch —> T1	Sleeping
0	call sem_post()	Running		sleeping		
	Acquires lock	Running				
1	Increments zem	Running				
1	Wakes up thread T1	Running		Ready		
			Condition wait will return once it gets lock	Ready		
1	zem_post() returns	Running		Ready		
0			Zem = zem - 1	Running		

Another Implementation …

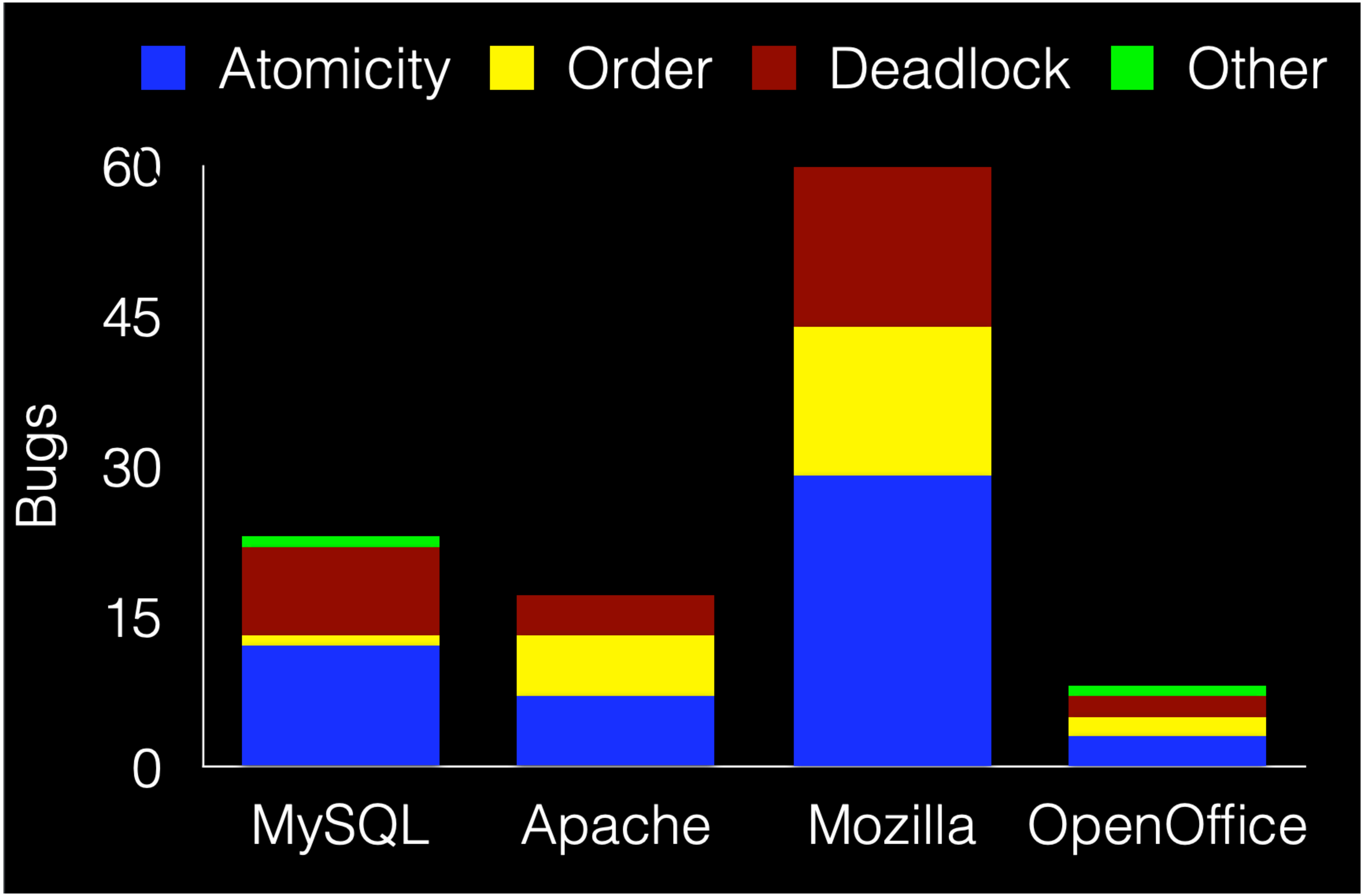


```
typedef struct {
     int value;
     struct process *list;
} semaphore;

wait(semaphore *S) {
             S->value--;
             if (S->value < 0) {
                     add this process to S->list;
                     block();
             }
}

signal(semaphore *S) {
          S->value++;
          if (S->value <= 0) {
                  remove a process P from S->list;
                  wakeup(P);
          }
}
```

Concurrency Bugs



Types of bugs in 4 major projects from 500K bug reports

Concurrency Bugs — Atomicity

Concurrency Bugs — Atomicity

MySQL bug …

Concurrency Bugs — Atomicity

MySQL bug …

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

Concurrency Bugs — Atomicity

MySQL bug …

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

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

Concurrency Bugs — Atomicity

MySQL bug …

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

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

- Is this problematic?

Concurrency Bugs — Atomicity

MySQL bug …

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

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

- Is this problematic?
    - Yes, else we wouldn’t be discussing …

Concurrency Bugs — Atomicity

MySQL bug …

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

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

- Is this problematic?
  - Yes, else we wouldn’t be discussing …
  - How?

Concurrency Bugs — Atomicity

```
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);
```

Concurrency Bugs — Atomicity

Simple Solution

```
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);
```

Concurrency Bugs — Order Violation

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

```

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

Concurrency Bugs — Order Violation

Mozilla bug …

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

```

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

Concurrency Bugs — Order Violation

Mozilla bug …

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

```

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

- Is this problematic?

Concurrency Bugs — Order Violation

Mozilla bug …

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

```

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

- Is this problematic?
  - Yes, else we wouldn’t be discussing …

Concurrency Bugs — Order Violation

Mozilla bug …

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

```

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

- Is this problematic?
  - Yes, else we wouldn't be discussing …
  - How?

Concurrency Bugs — Order Violation

Concurrency Bugs — Order Violation

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0;
```

Concurrency Bugs — Order Violation

```
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);
  ...
}
```

Concurrency Bugs — Order Violation

```
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;
}
```

Concurrency Bugs — Deadlock

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1

Concurrency Bugs — Deadlock

Thread 1

```
Lock(L1);
Lock(L2);
```

Thread 2

```
Lock(L2);
Lock(L1);
```

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2
- Thread T1 completes critical section

Concurrency Bugs — Deadlock

Thread 1

```
Lock(L1);
Lock(L2);
```

Thread 2

```
Lock(L2);
Lock(L1);
```

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2
- Thread T1 completes critical section
- Context Switch

Concurrency Bugs — Deadlock

Thread 1

```
Lock(L1);
Lock(L2);
```

Thread 2

```
Lock(L2);
Lock(L1);
```

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2
- Thread T1 completes critical section
- Context Switch
- Thread T2 gets Lock L2 and Lock L1

Concurrency Bugs — Deadlock

Thread 1

```
Lock(L1);
Lock(L2);
```

Thread 2

```
Lock(L2);
Lock(L1);
```

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2
- Thread T1 completes critical section
- Context Switch
- Thread T2 gets Lock L2 and Lock L1
- Works :)

Concurrency Bugs — Deadlock

Concurrency Bugs — Deadlock

Thread 1

```
Lock(L1);
Lock(L2);
```

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Context Switch

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch
- Thread T1 waits since it doesn’t have Lock 2

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch
- Thread T1 waits since it doesn’t have Lock 2
- Context Switch

Concurrency Bugs — Deadlock

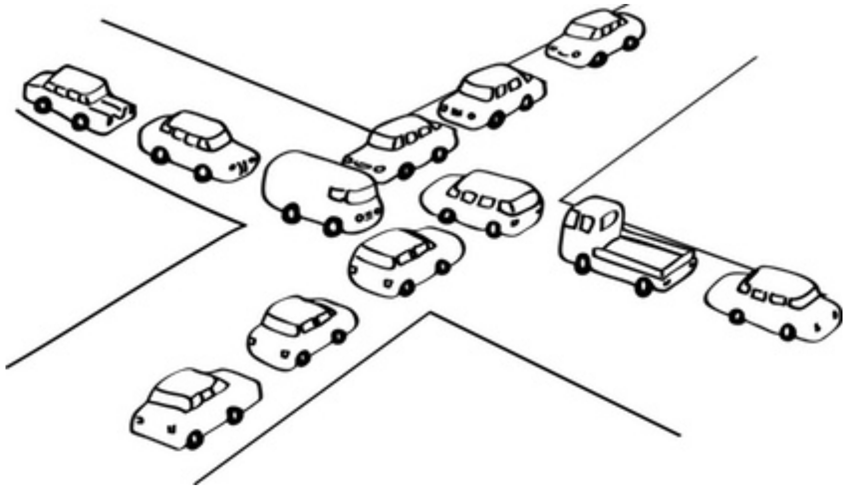
Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch
- Thread T1 waits since it doesn’t have Lock 2
- Context Switch
- Thread t2 waits since it doesn’t have Lock 1

Concurrency Bugs — Deadlock

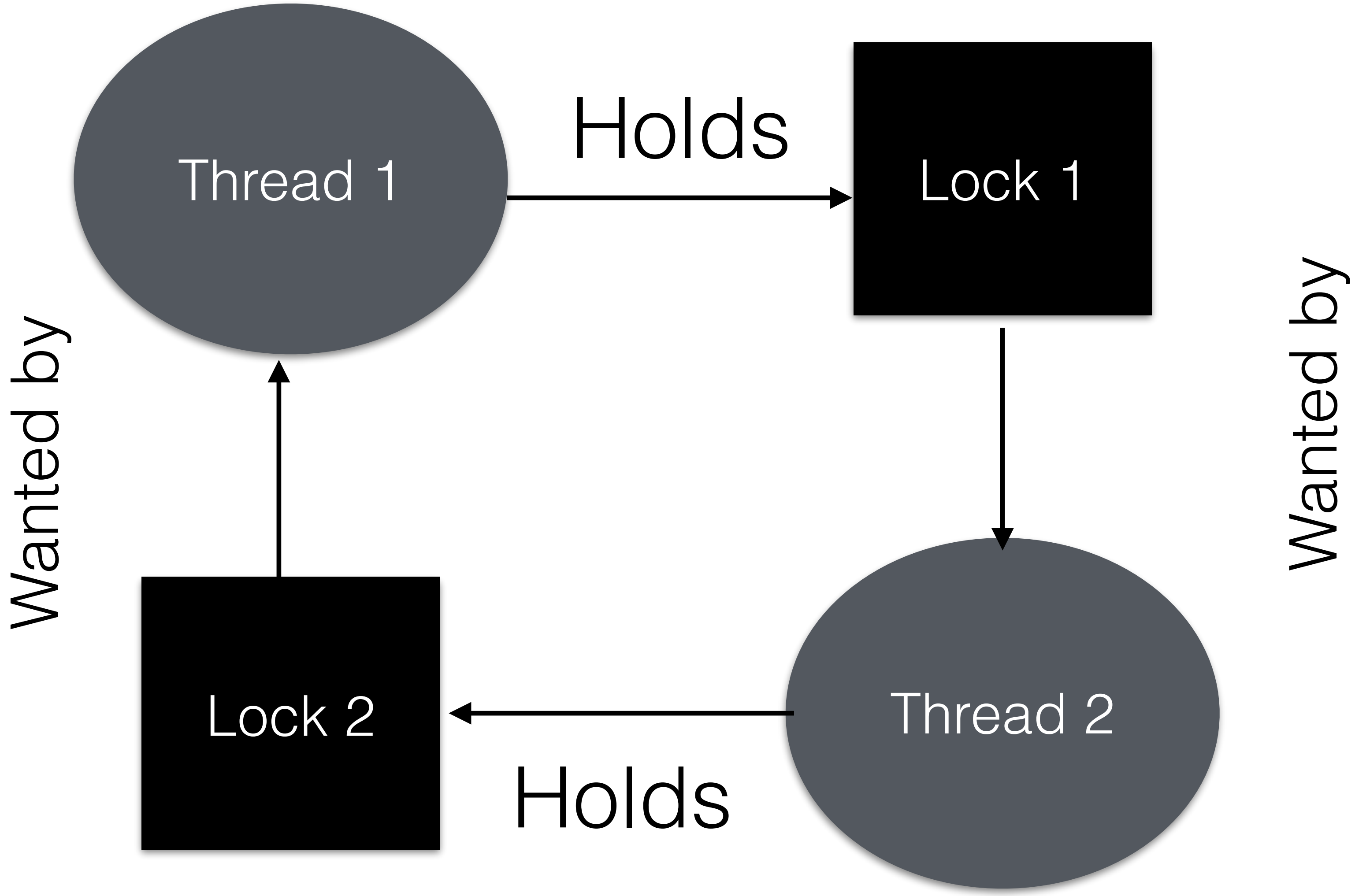
Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch
- Thread T1 waits since it doesn’t have Lock 2
- Context Switch
- Thread t2 waits since it doesn’t have Lock 1



Concurrency Bugs — Deadlock
Dependency Graphs

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);



Why Deadlocks Occur

Why Deadlocks Occur

- Encapsulation

Why Deadlocks Occur

- Encapsulation
  - Example: Java vector addAll method

Why Deadlocks Occur

- Encapsulation
  - Example: Java vector addAll method
    - V1 = [1, 4, 5]

Why Deadlocks Occur

- Encapsulation
  - Example: Java vector addAll method
    - V1 = [1, 4, 5]
    - V2 = [6, 7, 8]

Why Deadlocks Occur

- Encapsulation
  - Example: Java vector addAll method
    - V1 = [1, 4, 5]
    - V2 = [6, 7, 8]
    - V1.addAll(V2) —> [1, 4, 5, 6, 7, 8]

Why Deadlocks Occur

- Encapsulation
  - Example: Java vector addAll method
    - V1 = [1, 4, 5]
    - V2 = [6, 7, 8]
    - V1.addAll(V2) —> [1, 4, 5, 6, 7, 8]
    - If addAll is multithreaded & assuming it locks V1, V2 in that order

Why Deadlocks Occur

- Encapsulation
    - Example: Java vector addAll method
        - V1 = [1, 4, 5]
        - V2 = [6, 7, 8]
        - V1.addAll(V2) —> [1, 4, 5, 6, 7, 8]
        - If addAll is multithreaded & assuming it locks V1, V2 in that order
        - If some thread all calls V2.add(V1), we have a deadlock!

Why Deadlocks Occur

- Encapsulation
  - Example: Java vector addAll method
    - V1 = [1, 4, 5]
    - V2 = [6, 7, 8]
    - V1.addAll(V2) —> [1, 4, 5, 6, 7, 8]
    - If addAll is multithreaded & assuming it locks V1, V2 in that order
    - If some thread all calls V2.add(V1), we have a deadlock!
- Complex dependecies

Why Deadlocks Occur

- Encapsulation
  - Example: Java vector addAll method
    - V1 = [1, 4, 5]
    - V2 = [6, 7, 8]
    - V1.addAll(V2) —> [1, 4, 5, 6, 7, 8]
    - If addAll is multithreaded & assuming it locks V1, V2 in that order
    - If some thread all calls V2.add(V1), we have a deadlock!
- Complex dependecies
  - Virtual memory system calls filesystem

Why Deadlocks Occur

- Encapsulation
  - Example: Java vector addAll method
    - V1 = [1, 4, 5]
    - V2 = [6, 7, 8]
    - V1.addAll(V2) —> [1, 4, 5, 6, 7, 8]
    - If addAll is multithreaded & assuming it locks V1, V2 in that order
    - If some thread all calls V2.add(V1), we have a deadlock!
- Complex dependecies
  - Virtual memory system calls filesystem
  - Filesystem calls virtual memory

Concurrency Bugs — Deadlock
Dependency Graphs

Concurrency Bugs — Deadlock
Dependency Graphs

Condition	Description

Concurrency Bugs — Deadlock
Dependency Graphs

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.

Concurrency Bugs — Deadlock Dependency Graphs

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

Concurrency Bugs — Deadlock Dependency Graphs

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.

Concurrency Bugs — Deadlock Dependency Graphs

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 or more resources that are being requested by the next thread in the chain

Concurrency Bugs — Deadlock
Dependency Graphs

Concurrency Bugs — Deadlock
Dependency Graphs

Condition	Description

Concurrency Bugs — Deadlock
Dependency Graphs

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.

Concurrency Bugs — Deadlock Dependency Graphs

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

Concurrency Bugs — Deadlock Dependency Graphs

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.

Concurrency Bugs — Deadlock
Dependency Graphs

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 or more resources that are being requested by the next thread in the chain

Preventing Circular Wait

Thread 1

```
Lock(L1);
Lock(L2);
```

Deadlock Version

Thread 2

```
Lock(L2);
Lock(L1);
```

Preventing Circular Wait

- Provide a total ordering of lock acquisition

Deadlock Version

Thread 1

```
Lock(L1);
Lock(L2);
```

Thread 2

```
Lock(L2);
Lock(L1);
```

Preventing Circular Wait

- Provide a total ordering of lock acquisition

Deadlock Version

Thread 1

```
Lock(L1);
Lock(L2);
```

Thread 2

```
Lock(L2);
Lock(L1);
```

Preventing Circular Wait

- Provide a total ordering of lock acquisition

Thread 1

Lock(L1);
Lock(L2);

Deadlock Version

Thread 2

Lock(L2);
Lock(L1);

Thread 1

Lock(L1);
Lock(L2);

Non-deadlock Version

Thread 2

Lock(L1);
Lock(L2);

Preventing Circular Wait

Preventing Circular Wait

- Provide a total ordering of lock acquisition

Preventing Circular Wait

- Provide a total ordering of lock acquisition

Preventing Circular Wait

- Provide a total ordering of lock acquisition
- Define a function do_something which works correct even if two threads call it as:

Preventing Circular Wait

- Provide a total ordering of lock acquisition
- Define a function do_something which works correct even if two threads call it as:
    - T1 — do_something (L1, L2) and

Preventing Circular Wait

- Provide a total ordering of lock acquisition
- Define a function do_something which works correct even if two threads call it as:
  - T1 — do_something (L1, L2) and
  - T2 — do_something (L2, L1)

Preventing Circular Wait

- Provide a total ordering of lock acquisition
- Define a function do_something which works correct even if two threads call it as:
  - T1 — do_something (L1, L2) and
  - T2 — do_something (L2, L1)

```
do something(mutex t *m1, mutex t *m2)
```

Preventing Circular Wait

- Provide a total ordering of lock acquisition
- Define a function do_something which works correct even if two threads call it as:
  - T1 — do_something (L1, L2) and
  - T2 — do_something (L2, L1)

```
do something(mutex t *m1, mutex t *m2)
if (m1 > m2)
 { // grab locks 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 (it is not the same lock)
```

Concurrency Bugs — Deadlock
Dependency Graphs

Concurrency Bugs — Deadlock
Dependency Graphs

Condition	Description

Concurrency Bugs — Deadlock Dependency Graphs

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.

Concurrency Bugs — Deadlock Dependency Graphs

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

Concurrency Bugs — Deadlock Dependency Graphs

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.

Concurrency Bugs — Deadlock Dependency Graphs

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 or more resources that are being requested by the next thread in the chain

Preventing Hold and Wait

<u>Thread 1</u>

```
Lock(L1);
Lock(L2);
```

Deadlock Version

<u>Thread 2</u>

```
Lock(L2);
Lock(L1);
```

Preventing Hold and Wait

- Acquire **all** locks **at once atomically**

Deadlock Version

Thread 1

```
Lock(L1);
Lock(L2);
```

Thread 2

```
Lock(L2);
Lock(L1);
```

Preventing Hold and Wait

- Acquire **all** locks **at once atomically**

Thread 1

```
Lock(L1);
Lock(L2);
```

Deadlock Version

Thread 2

```
Lock(L2);
Lock(L1);
```

Preventing Hold and Wait

- Acquire **all** locks **at once atomically**

Deadlock Version

Thread 1

Lock(L1);
Lock(L2);

Thread 2

Lock(L2);
Lock(L1);

Non-deadlock Version

Thread 1

Lock(ALL)
Lock(L1);
Lock(L2);
…..
Unlock(ALL);

Thread 2

Lock(ALL)
Lock(L1);
Lock(L2);
…
Unlock(ALL)

Preventing Hold and Wait

- Acquire **all** locks **at once atomically**

Thread 1
Lock(ALL)
Lock(L1);
Lock(L2);
…..
Unlock(ALL);

Non-deadlock Version

Thread 2
Lock(ALL)
Lock(L1);
Lock(L2);
…
Unlock(ALL)

Preventing Hold and Wait

- Acquire **all** locks **at once atomically**
- Cons

```
Thread 1
Lock(ALL)
Lock(L1);
Lock(L2);
.....
Unlock(ALL);
```

Non-deadlock Version

```
Thread 2
Lock(ALL)
Lock(L1);
Lock(L2);
...
Unlock(ALL)
```

Preventing Hold and Wait

- Acquire **all** locks **at once atomically**
- Cons
  - Requires us to know which all locks will be required ahead of time

<u>Thread 1</u>
Lock(ALL)
Lock(L1);
Lock(L2);
.....
Unlock(ALL);

Non-deadlock Version

<u>Thread 2</u>
Lock(ALL)
Lock(L1);
Lock(L2);
...
Unlock(ALL)

Preventing Hold and Wait

- Acquire **all** locks **at once atomically**
- Cons
  - Requires us to know which all locks will be required ahead of time
  - Reduction of concurrency

Thread 1
Lock(ALL)
Lock(L1);
Lock(L2);
…..
Unlock(ALL);

Non-deadlock Version

Thread 2
Lock(ALL)
Lock(L1);
Lock(L2);
…
Unlock(ALL)

Concurrency Bugs — Deadlock
Dependency Graphs

Concurrency Bugs — Deadlock Dependency Graphs

Condition	Description

Concurrency Bugs — Deadlock Dependency Graphs

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.

Concurrency Bugs — Deadlock Dependency Graphs

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

Concurrency Bugs — Deadlock Dependency Graphs

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.

Concurrency Bugs — Deadlock Dependency Graphs

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 or more resources that are being requested by the next thread in the chain

Pre-emption

Thread 1

```
Lock(L1);
Lock(L2);
```

Deadlock Version

Thread 2

```
Lock(L2);
Lock(L1);
```

Non-deadlock Version

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

Pre-emption

Thread 1

Lock(L1);
Lock(L2);

Deadlock Version

Thread 2

Lock(L2);
Lock(L1);

Non-deadlock Version

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

Pre-emption

- Livelock: Both threads running this sequence repeatedly

Deadlock Version

<u>Thread 1</u>

Lock(L1);

Lock(L2);

<u>Thread 2</u>

Lock(L2);

Lock(L1);

Non-deadlock Version

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

Pre-emption

- Livelock: Both threads running this sequence repeatedly
- How to solve?

<u>Thread 1</u>

```
Lock(L1);
Lock(L2);
```

Deadlock Version

<u>Thread 2</u>

```
Lock(L2);
Lock(L1);
```

Non-deadlock Version

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

Pre-emption

- Livelock: Both threads running this sequence repeatedly
- How to solve?
  - Add random delay

Thread 1

```
Lock(L1);
Lock(L2);
```

Deadlock Version

Thread 2

```
Lock(L2);
Lock(L1);
```

Non-deadlock Version

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

Concurrency Bugs — Deadlock Dependency Graphs

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 or more resources that are being requested by the next thread in the chain

Prevention — Mutual exclusion

Prevention — Mutual exclusion

- Use atomic instructions!

Prevention — Mutual exclusion

- Use atomic instructions!

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

Prevention — Mutual exclusion

- Use atomic instructions!

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

- Use above code to implement atomic increment: x = x+k

Prevention — Mutual exclusion

- Use atomic instructions!

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

- Use above code to implement atomic increment: x = x+k

```
void AtomicIncrement(int *value, int amount){
...Fill Here
}
```

Prevention — Mutual exclusion

- Use atomic instructions!

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

- Use above code to implement atomic increment: x = x+k

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

Prevention — Mutual exclusion

- List insertion using atomic instructions

Prevention — Mutual exclusion

- List insertion using atomic instructions

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

Prevention — Mutual exclusion

- List insertion using atomic instructions

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

- Where is the race condition?

Prevention — Mutual exclusion

- Mutex based solution

Prevention — Mutual exclusion

- Mutex based solution

```
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
}
```

Prevention — Mutual exclusion

- Atomic instruction based?

HINT

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

Prevention — Mutual exclusion

- Atomic instruction based?

HINT

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

```
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

Only one of T1 & T2 will
run at a given time …

Deadlock Avoidance via Scheduling

- Global knowledge about which threads might be acquired is needed ahead of time

Only one of T1 & T2 will run at a given time …

Deadlock Avoidance via Scheduling

- Global knowledge about which threads might be acquired is needed ahead of time
- Assume 2 processors and 4 threads and following lock requirements. How will you schedule to avoid deadlocks?

Only one of T1 & T2 will run at a given time …

Deadlock Avoidance via Scheduling

- Global knowledge about which threads might be acquired is needed ahead of time
- Assume 2 processors and 4 threads and following lock requirements. How will you schedule to avoid deadlocks?
  - Hint: you can think serial!

Only one of T1 & T2 will run at a given time …

Deadlock Avoidance via Scheduling

- Global knowledge about which threads might be acquired is needed ahead of time
- Assume 2 processors and 4 threads and following lock requirements. How will you schedule to avoid deadlocks?
  - Hint: you can think serial!

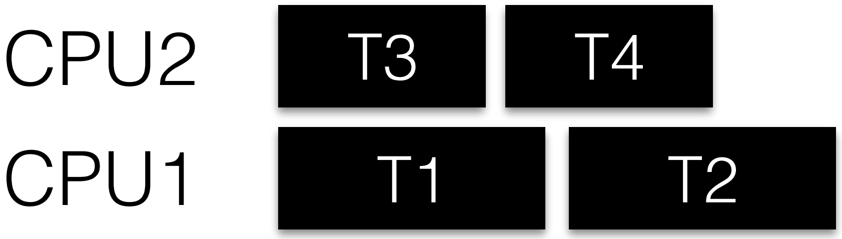
	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

Only one of T1 & T2 will run at a given time …

Deadlock Avoidance via Scheduling

- Global knowledge about which threads might be acquired is needed ahead of time
- Assume 2 processors and 4 threads and following lock requirements. How will you schedule to avoid deadlocks?
  - Hint: you can think serial!

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



CPU2 T3 T4

CPU1 T1 T2

Only one of T1 & T2 will run at a given time …

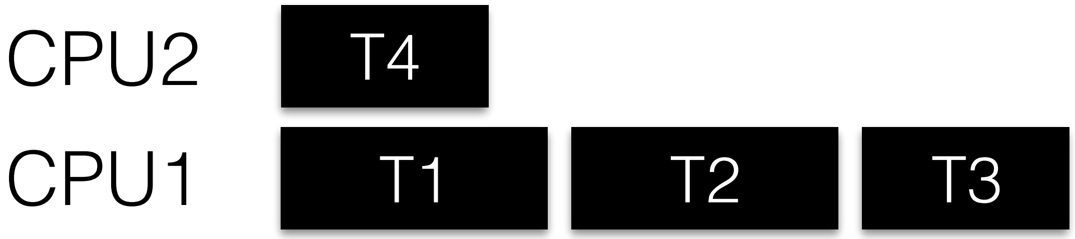
Deadlock Avoidance via Scheduling

Deadlock Avoidance via Scheduling

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

Deadlock Avoidance via Scheduling

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



CPU2 T4

CPU1 T1 T2 T3