Concurrency

University of Stavanger

Concurrency

1. What is a thread?

   General idea and problems (chap 26)

2. The thread API (chap 27)

   How to use them in c or python and go

   - Creation, completion, synchronisation with locks and condition variables.

3. How Locks are implemented ? (chap 28)
4. Concurrent data structures (chap29)
5. How condition variables are implemented?(chap30)
6. Semaphores and some general concurrency patterns (chap 31)
7. Concurrency bugs (chap 32)
8. ~~Event-based concurrency (chap 33)~~

What is a Thread

What are the differences between threads and processes?

What is a Thread

What are the differences between threads and processes?

Similarities

- A Process has one or many threads
- PCB for process state, TCB (Thread/Task Control Block) for thread state
- The scheduler schedules threads just like it does with a process.
- The possible states of a threads are the same as for processes (ready, blocked, running)

So, what are the differences?

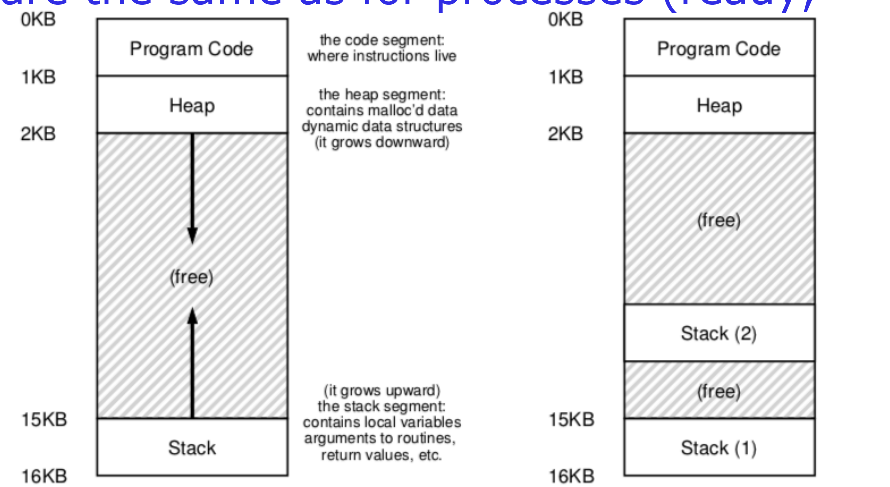


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

What is a Thread

What are the differences between threads and processes?

Differences

- A single threaded process is just a process with one thread, thus one stack. A multi threaded process can have many stacks; one per thread.
- Threads of a process share the same address space!

- Heap and program code are accessible by all threads within a process!! **Major concern**
- When switching between threads the OS does not need to switch the Page Table. Why?

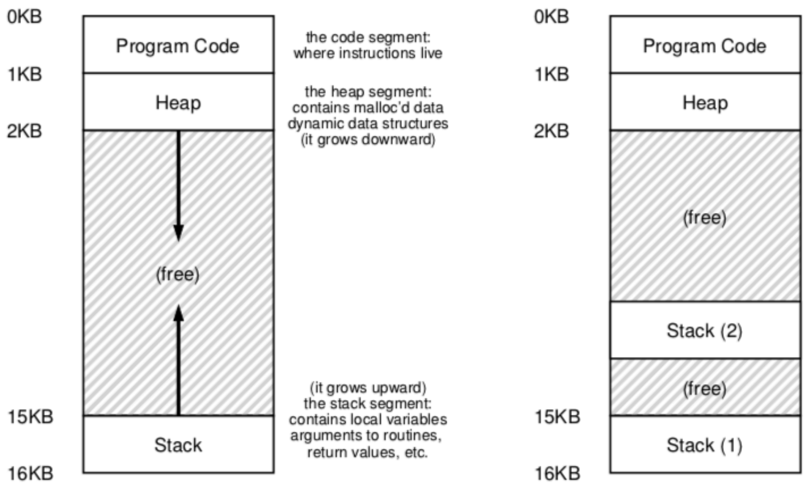


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

What is a Thread

Why Threads?

Parallelism

Spread work over multiple CPUs, and thus finish the job faster!

Examples?

What is a Thread

Why Threads?

Parallelism

Spread work over multiple CPUs, and thus finish the job faster!

Examples?

Note!

Scheduling multi-threaded processes is addressed in chapter 10, we will come back to this.

What is a Thread

Why Threads?

Avoid blocking progress

Threading enables overlapping I/O operations with other activities within a single program.

Examples?

What is a Thread

Why Threads (26.1)?

Discussion

What is the difference between parallelism and Concurrency?

What is a Thread

Why Threads?

Discussion

What is the difference between parallelism and Concurrency?

- Concurrency is not parallelism

What is a Thread

Why Threads?

Discussion

What is the difference between parallelism and Concurrency?

- Concurrency is not parallelism
- Concurrency is about dealing with lots of things at once, while parallelism is about doing lots of things at once. Concurrency enables parallelism, but parallelism is not the (only) goal of concurrency (Rob Pike, Go programming language)

What is a Thread

Why Threads?

Discussion

What is the difference between parallelism and Concurrency?

- Concurrency is not parallelism
- Concurrency is about dealing with lots of things at once, while parallelism is about doing lots of things at once. Concurrency enables parallelism, but parallelism is not the (only) goal of concurrency (Rob Pike, Go programming language)
- Concurrency is about making correct systems, while parallelism is about making systems run fast by exploiting the available resources

What is a Thread

Why Threads?

Discussion

What is the difference between parallelism and Concurrency?

- Concurrency is not parallelism
- Concurrency is about dealing with lots of things at once, while parallelism is about doing lots of things at once. Concurrency enables parallelism, but parallelism is not the (only) goal of concurrency (Rob Pike, Go programming language)
- Concurrency is about making correct systems, while parallelism is about making systems run fast by exploiting the available resources
- Parallelism is a subset of currency?

What is a Thread

Why Threads?

Discussion

What is the difference between parallelism and Concurrency?

- Concurrency is not parallelism
- Concurrency is about dealing with lots of things at once, while parallelism is about doing lots of things at once. Concurrency enables parallelism, but parallelism is not the (only) goal of concurrency (Rob Pike, Go programming language)
- Concurrency is about making correct systems, while parallelism is about making systems run fast by exploiting the available resources
- Parallelism is a subset of currency?
- You can understand it, but you can’t explain it ☺

What is a Thread

Why Threads?

So why multiple threads when we can have multiple processes?

What is a Thread

Why Threads?

So why multiple threads when we can have multiple processes?

Threads share the same address space within a process. This gives more possibilities for application development, but at the cost of additional complexity.

What gets complex?

What is a Thread

Why Threads?

So why multiple threads when we can have multiple processes?

Threads share the same address space within a process. This gives more possibilities for application development, but at the cost of additional complexity.

What gets complex?

What is a Thread

Run this program t0.c

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A");
    assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B");
    assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL);
    assert(rc == 0);
    rc = pthread_join(p2, NULL);
    assert(rc == 0);
    printf("main: end\n");
    return 0;
```

What is a Thread

And this program t1.c

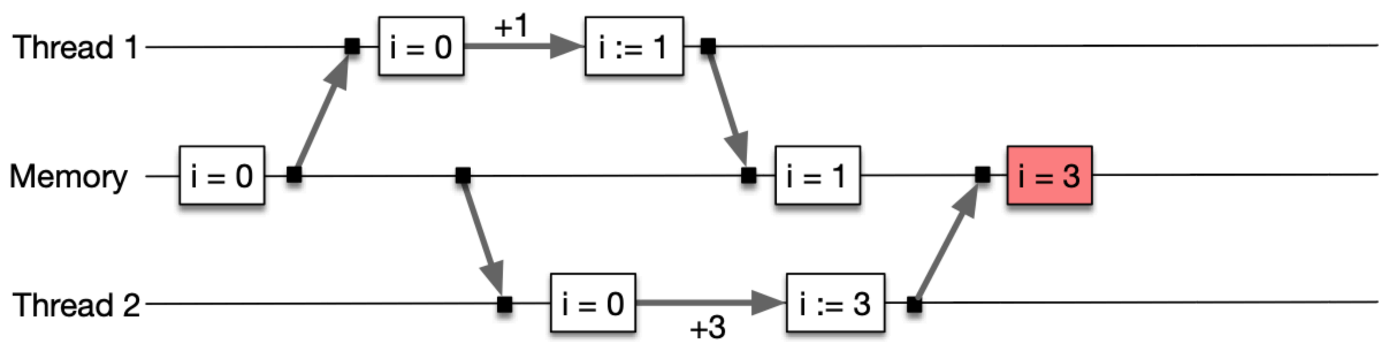
```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

static volatile int counter = 0;

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e5; i++) {
        counter++;
        // counter = counter + 1;
    }
    printf("%s: end\n", (char *)arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    int rc;
    rc = pthread_create(&p1, NULL, mythread, "A");
    assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B");
    assert(rc == 0);
    rc = pthread_join(p1, NULL);
    assert(rc == 0);
    rc = pthread_join(p2, NULL);
    assert(rc == 0);
    printf("main: end (counter = %d)\n", counter);
    return 0;
}
```

What is a Thread

The heart of the problem (26.4)?

Sharing data leads to problems.

1. Race Condition /Data Race



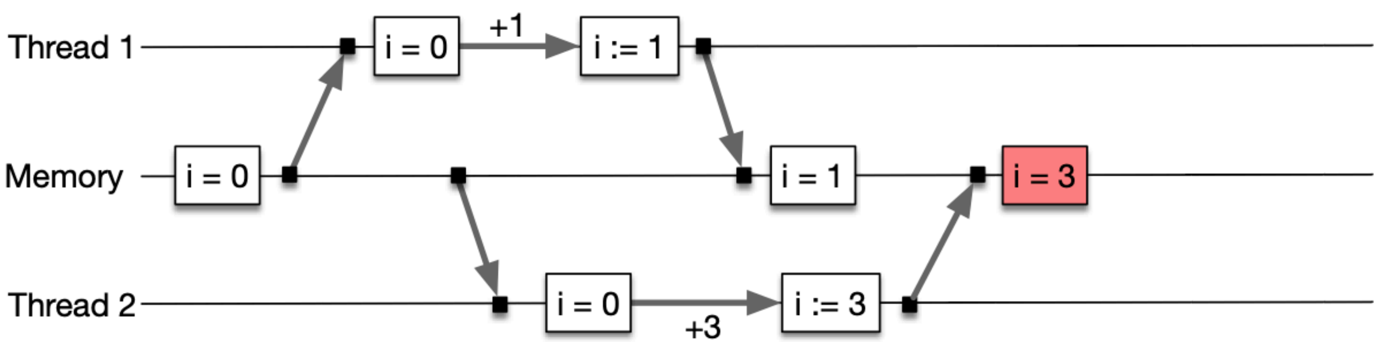
A critical section is that piece of code that when executed by multiple threads may results in a **race condition. Solution?**

What is a Thread

The heart of the problem (26.4)?

Sharing data leads to problems.

1. Race Condition /Data Race



A critical section is that piece of code that when executed by multiple threads may results in a **race condition. Solution? Mutual Exclusion**

What is a Thread

Demo:

Race condition again in Python and go

What is a Thread

Edsger Dijkstra

- coined these terms
- Pioneer in the field

Def. Critical Section: a piece of code that *accesses* (read or write) a shared variable or resource and must not be concurrently executed by more than one thread.

A critical section is code that when executed by multiple threads may result in a **race condition**.

What we want:

Def. Mutual Exclusion: guarantee that if one thread is executing within the critical section, other threads are prevented from doing so!

What is a Thread

The Wish for Atomicity

We need some basic operations that we know are not going to be interrupted. Atomic operations!

What is a Thread (chap26)

Summary

Thread API Concurrency with Go

Critical section and mutex.

Exercise 1:

the program **lock-task1.go** can increment and decrement a counter. It is not thread safe, can you modify the code to make it thread safe.

Thread API Concurrency with Go

Producer Consumer

Exercise 2:

Locks (Chapter 28)

We need a synchronization primitive to make sure that a piece of code (critical section) can only be accessed by one thread at the time.

This primitive is called lock and provides mutual exclusion, also referred to as mutex.

Locks (Chapter 28)

We need a synchronization primitive to make sure that a piece of code (critical section) can only be accessed by one thread at the time.

This primitive is called lock and provides mutual exclusion, also referred to as mutex.

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

In C

Locks (Chapter 28)

We need a synchronization primitive to make sure that a piece of code (critical section) can only be accessed by one thread at the time.

This primitive is called lock and provides mutual exclusion, also referred to as mutex.

```
func (sc *SafeCounter) increment() {
    sc.Mu.Lock()
    sc.Val++
    sc.Mu.Unlock()
}
```

In Go: visit the info repo for the complete code "atomic-counter.go"

Locks (Chapter 28)

28.4 Evaluating Locks

Evaluation criteria

Correctness: Does the lock provide mutual exclusion?

Locks (Chapter 28)

28.4 Evaluating Locks

Evaluation criteria

Correctness: Does the lock provide mutual exclusion?

Fairness: Are the threads waiting for the lock fairly treated?

Locks (Chapter 28)

28.4 Evaluating Locks

Evaluation criteria

Correctness: Does the lock provide mutual exclusion?

Fairness: Are the threads waiting for the lock fairly treated?

Performance: Time overhead due to lock. Consider few threads waiting, many threads waiting, single CPU and multiple CPUs.

Locks (Chapter 28)

28.5 Controlling Interrupts

Idea: Disable interrupt when a thread enters the critical section.

Correctness: Does the lock provide mutual exclusion?

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Locks (Chapter 28)

28.5 Controlling Interrupts

Idea: Disable interrupt when a thread enters the critical section.

Correctness: Does the lock provide mutual exclusion?

On Single CPU Yes

Easy to see that disabling interrupts means no other thread can access the shared variables.

On Multiple CPUs No

Every CPU has its own interrupt, but the memory is still shared!

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Locks (Chapter 28)

28.5 Controlling Interrupts

Idea: Disable interrupt when a thread enters the critical section.

Fairness: Are the threads waiting for the lock fairly treated?

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

University of Stavanger

Locks (Chapter 28)

28.5 Controlling Interrupts

Idea: Disable interrupt when a thread enters the critical section.

Fairness: Are the threads waiting for the lock fairly treated?

No

Relies on well behaving threads/ user programs. Wrong assumption

Many things can go wrong with this approach

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Locks (Chapter 28)

28.5 Controlling Interrupts

Idea: Disable interrupt when a thread enters the critical section.

Performance: Time overhead due to lock. Consider few threads waiting, many threads waiting, single CPU and multiple CPUs.

Poor solution anyway no need to answer this question

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Locks (Chapter 28)

28.6 Locks using load and store

Correctness: Does the lock provide mutual exclusion?

This is Spin-waiting

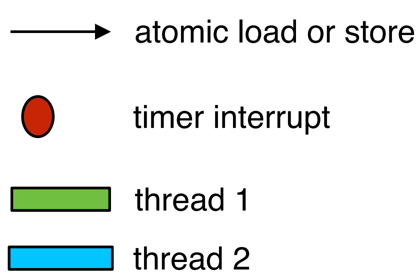
```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Figure 28.1: **First Attempt: A Simple Flag**

Locks (Chapter 28)

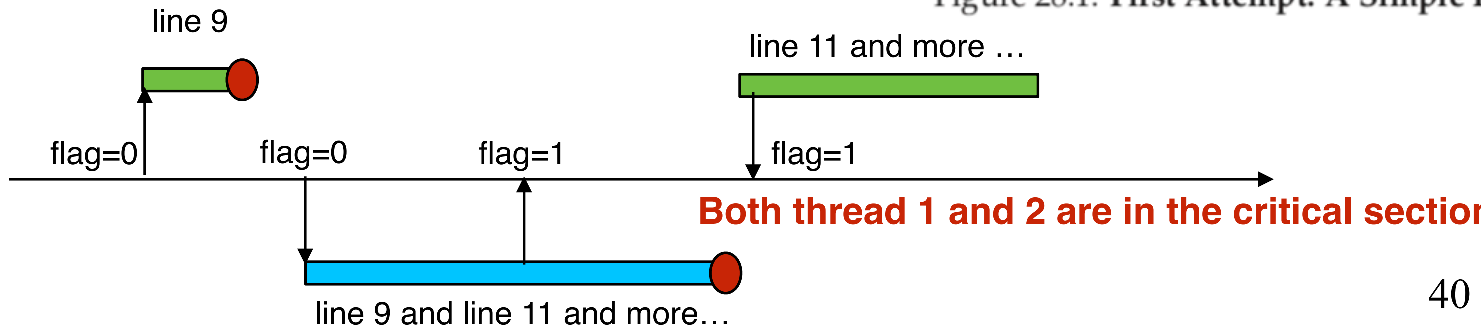
28.6 Locks using load and store

Correctness: Does the lock provide mutual exclusion? No, even if we assume atomic load and store.




```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10          ; // spin-wait (do nothing)
11      mutex->flag = 1;          // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

Figure 28.1: **First Attempt: A Simple Flag**



Locks (Chapter 28)

28.6 Locks using load and store

Correctness: Does the lock provide mutual exclusion? No, even if we assume atomic load and store.

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Figure 28.1: **First Attempt: A Simple Flag**

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: **Trace: No Mutual Exclusion**

Locks (Chapter 28)

28.7 Spin Locks with Test-And-Set

We need useful atomic instructions to implement locks: test-and-set instruction / atomic exchange instruction

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Allows to return the old value while setting the new value in ONE big operation (atomic). Also referred to as transaction.

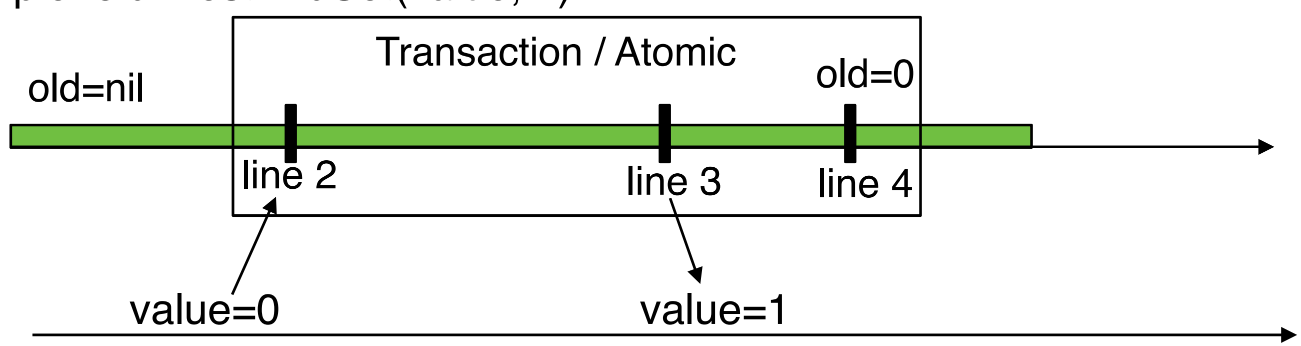
Locks (Chapter 28)

28.7 Spin Locks with Test-And-Set

We need useful atomic instructions to implement locks:
test-and-set instruction / atomic exchange instruction

```
1    int TestAndSet(int *old_ptr, int new) {
2        int old = *old_ptr; // fetch old value at old_ptr
3        *old_ptr = new;     // store 'new' into old_ptr
4        return old;         // return the old value
5    }
```

Example: old=TestAndSet(value, 1)



University of Stavanger

Locks (Chapter 28)

28.7 Spin Locks with Test-And-Set

So how to use this to implement a lock?

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: A Simple Spin Lock Using Test-and-set

Locks (Chapter 28)

28.8 Evaluating Spin Locks

Correctness, fairness, performance?

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: A Simple Spin Lock Using Test-and-set

Locks (Chapter 28)

28.8 Evaluating Spin Locks

Correctness: Yes

Fairness: Starvation

Performance on signal CPU: Poor. A thread holding a lock under context switch, other threads will spin the entire time slice hoping for a lock that is not going to be released soon. (wasting an entire time slot)

Locks (Chapter 28)

28.8 Evaluating Spin Locks

Correctness: Yes

Fairness: Starvation

Performance on multiple CPUs: better if the number of threads roughly equals the the number of CPUs. **Not the case though**

Lookup the notes and read page 323 for explanation.

Remark

The idea is too simplistic, since every thread is on different CPU, using a time slice on one cpu does not penalize another thread on another CPU. But…

Locks (Chapter 28)

28.9 Compare-And-Swap

Slightly more powerful primitive than Test-And-Set

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Figure 28.4: Compare-and-swap

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Locks (Chapter 28)

28.9 Compare-And-Swap (CAS)

What to modify to implement a lock with CAS?

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: A Simple Spin Lock Using Test-and-set

University of Stavanger

Locks (Chapter 28)

28.9 Compare-And-Swap (CAS)

What to modify to implement a lock with CAS?

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag,0,1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: A Simple Spin Lock Using Test-and-set

Locks (Chapter 28)

28.9 Evaluating Spin Locks with CAS

Correctness: Yes

Fairness: Starvation

Performance on signal CPU: Poor. A thread holding a lock under context switch, other threads will spin the entire time slice hoping for a lock that is not going to be released soon. (wasting an entire time slot)

Locks (Chapter 28)

28.9 Compare-And-Swap (CAS)

Why is it considered as a lock free synchronization?

Hint: Try to reason about a concurrent counter with CompareAndSwap and with TestAndSet

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Figure 28.4: Compare-and-swap

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Locks (Chapter 28)

28.9 Compare-And-Swap (CAS)

Why is it considered as a lock free synchronization?

Hint: https://www.youtube.com/watch?v=8XJ8r_n7POY&ab_channel=RobEdwards

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Figure 28.4: Compare-and-swap

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Locks (Chapter 28)

28.9 Compare-And-Swap (CAS)

With CAS it is possible to implement a lock free safe counter, but not with TestAndSet (TAS)

See atomic-counter.go in info repos under demos

Locks (Chapter 28)

28.10 Load-Linked and Store-Condition

Not syllabus

Locks (Chapter 28)

28.11 Fetch-And-Add

Another atomic operation that can be used to implement a lock.

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Locks (Chapter 28)

28.11 Fetch-And-Add

Lock with Fetch-And-Add

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Figure 28.7: Ticket Locks

Locks (Chapter 28)

28.11 Fetch-And-Add

Lock with Fetch-And-Add

Correctness: Yes

Fairness: Yes

Performance: No, still spin-waiting

Locks (Chapter 28)

28.12 Spin locks are too much spinning

Waste of resources

Locks (Chapter 28)

28.13 Just Yield, (Baby)

Now, we need the OS!

Assumes an OS primitive yield()—> a thread can ask to go from running to ready state.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Locks (Chapter 28)

28.13 Just Yield, (Baby)

What issues can arise?

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Locks (Chapter 28)

28.13 Just Yield, (Baby)

What issues can arise? Unnecessary context switch.

100 threads, 1 holding the lock, all others will try to find out the the lock is taken and yield again. Can we avoid this

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

University of Stavanger

Locks (Chapter 28)

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup

28.14 Using Queues

- park(): puts the thread to **blocked**
- unpark(id): puts the thread to **ready**
- access to CS is when flag=1(the lock)
- acquire the guard to manipulate the queue (adding to queue and park, remove from queue and unpark)

unlock():

- unpack() systematically leaves flag=1 for the thread just woken up, and sets guard=0

Locks (Chapter 28)

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

Thread 1

lock_t

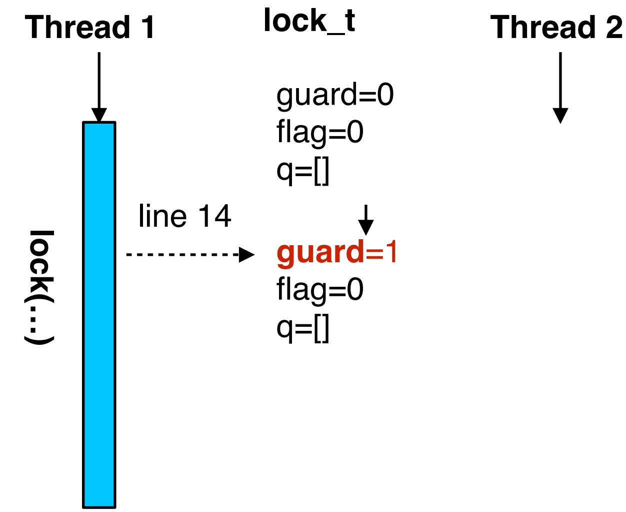
guard=0
flag=0
q=[]

Thread 2

Locks (Chapter 28)

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

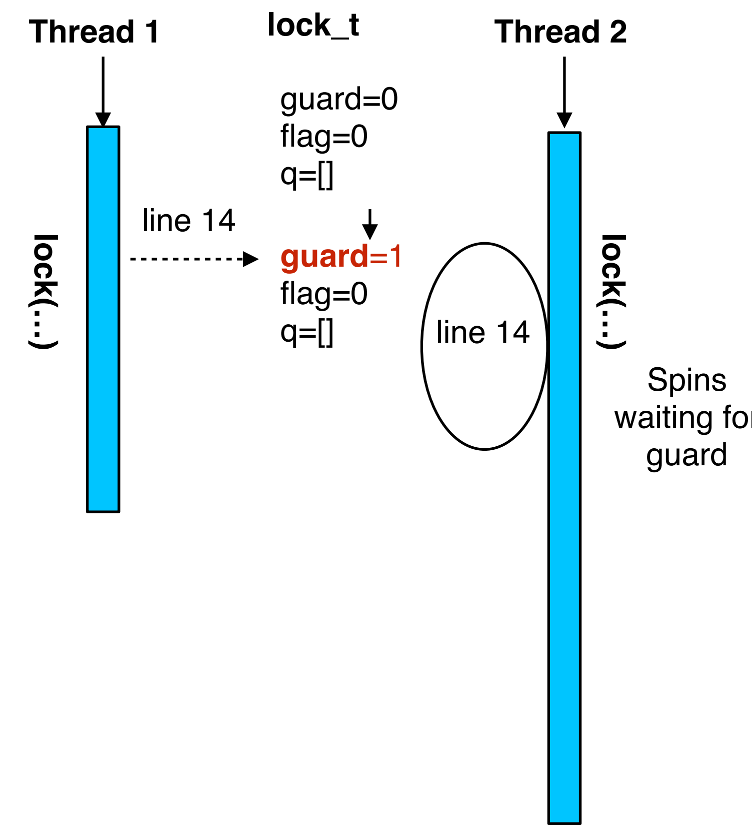
Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup



Locks (Chapter 28)

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

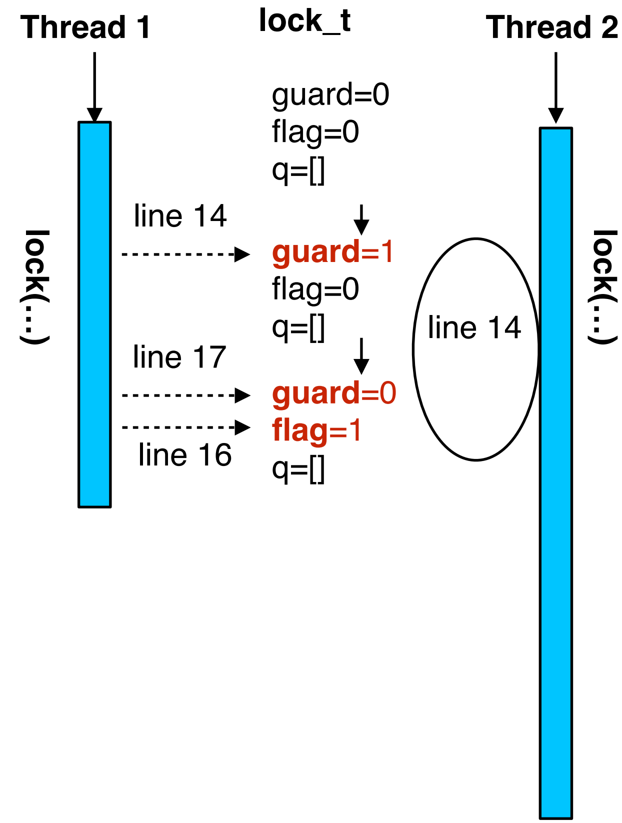


University of Stavanger

Locks (Chapter 28)

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

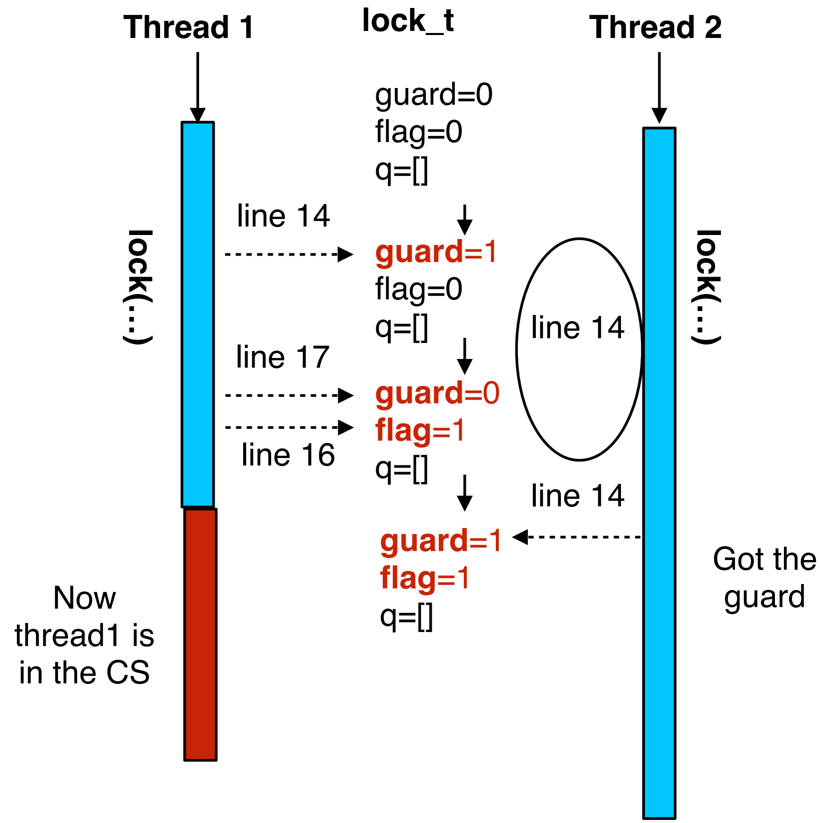
Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**



Locks (Chapter 28)

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

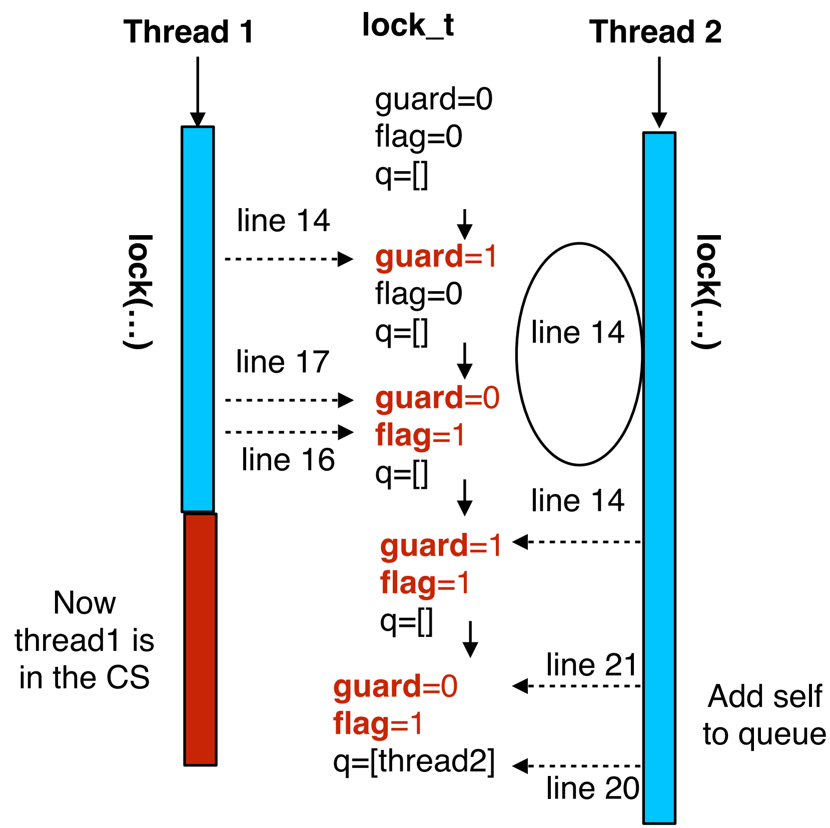
Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**



Locks (Chapter 28)

```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q;
5   } lock_t;
6
7   void lock_init(lock_t *m) {
8       m->flag  = 0;
9       m->guard = 0;
10      queue_init(m->q);
11  }
12
13  void lock(lock_t *m) {
14      while (TestAndSet(&m->guard, 1) == 1)
15          ; //acquire guard lock by spinning
16      if (m->flag == 0) {
17          m->flag = 1; // lock is acquired
18          m->guard = 0;
19      } else {
20          queue_add(m->q, gettid());
21          m->guard = 0;
22          park();
23      }
24  }
25
26  void unlock(lock_t *m) {
27      while (TestAndSet(&m->guard, 1) == 1)
28          ; //acquire guard lock by spinning
29      if (queue_empty(m->q))
30          m->flag = 0; // let go of lock; no one wants it
31      else
32          unpark(queue_remove(m->q)); // hold lock (for next thread!)
33      m->guard = 0;
34  }
```

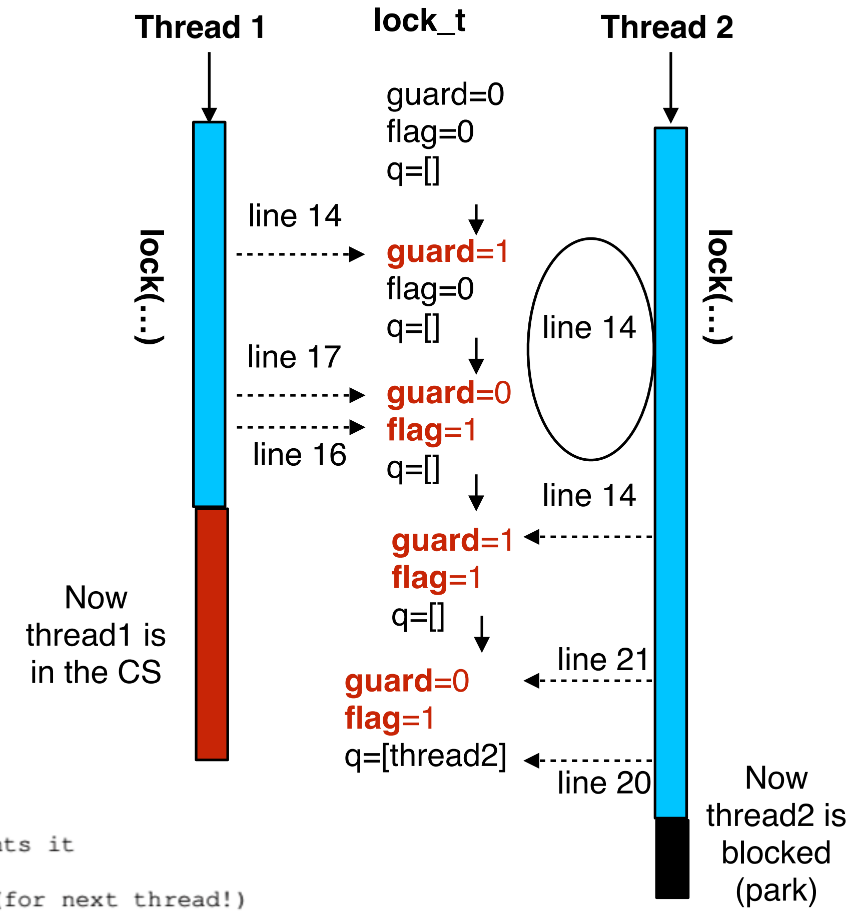
Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**



Locks (Chapter 28)

```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q;
5   } lock_t;
6
7   void lock_init(lock_t *m) {
8       m->flag  = 0;
9       m->guard = 0;
10      queue_init(m->q);
11  }
12
13  void lock(lock_t *m) {
14      while (TestAndSet(&m->guard, 1) == 1)
15          ; //acquire guard lock by spinning
16      if (m->flag == 0) {
17          m->flag = 1; // lock is acquired
18          m->guard = 0;
19      } else {
20          queue_add(m->q, gettid());
21          m->guard = 0;
22          park();
23      }
24  }
25
26  void unlock(lock_t *m) {
27      while (TestAndSet(&m->guard, 1) == 1)
28          ; //acquire guard lock by spinning
29      if (queue_empty(m->q))
30          m->flag = 0; // let go of lock; no one wants it
31      else
32          unpark(queue_remove(m->q)); // hold lock (for next thread!)
33      m->guard = 0;
34  }
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

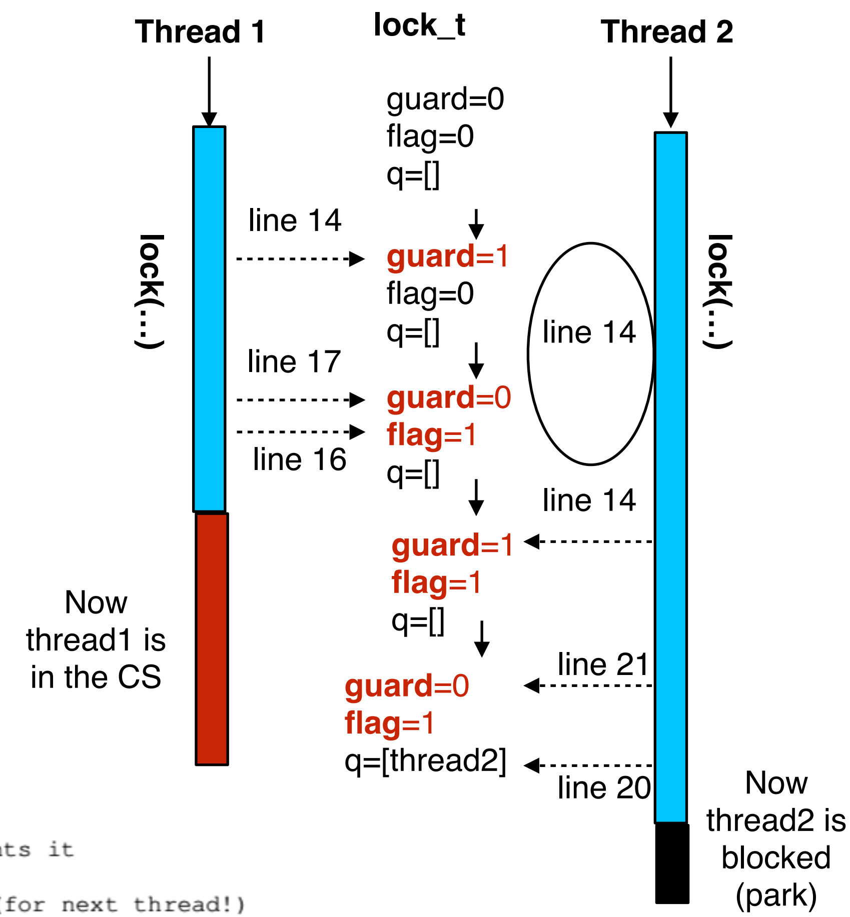


Locks (Chapter 28)

Discuss unlock(lock_t *m) !

```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q;
5   } lock_t;
6
7   void lock_init(lock_t *m) {
8       m->flag  = 0;
9       m->guard = 0;
10      queue_init(m->q);
11  }
12
13  void lock(lock_t *m) {
14      while (TestAndSet(&m->guard, 1) == 1)
15          ; //acquire guard lock by spinning
16      if (m->flag == 0) {
17          m->flag = 1; // lock is acquired
18          m->guard = 0;
19      } else {
20          queue_add(m->q, gettid());
21          m->guard = 0;
22          park();
23      }
24  }
25
26  void unlock(lock_t *m) {
27      while (TestAndSet(&m->guard, 1) == 1)
28          ; //acquire guard lock by spinning
29      if (queue_empty(m->q))
30          m->flag = 0; // let go of lock; no one wants it
31      else
32          unpark(queue_remove(m->q)); // hold lock (for next thread!)
33      m->guard = 0;
34  }
```

Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup



Locks (Chapter 28)

28.15 Different OS, Different Support

Not syllabus

Locks (Chapter 28)

28.16 Two-Phase Locks

Not syllabus

Locks (Chapter 28)

Summary

- Locks are synchronisation primitives
- Locks ensure safe manipulation of the critical section
- To implement a lock we need atomic operations (hardware support)
- Some atomic operations such as CompareAndSwap enable some lock-free synchronization
- To avoid unnecessary spinning we need OS support Yield, Park, Unpark for example

Condition Variables (Chapter 30)

Locks don’t let waiting threads to know when a condition is fulfilled, unless they have to check it them selfs.

Lets revisit the python and go examples from the info repo

Condition Variables (Chapter 30)

Condition Variable definition

Def. Condition Variable: an explicit queue that threads can put themselves on when some **condition** is not satisfied, to **wait** for that condition to become true. Another thread can change said condition, and then **signal** (to wake up one or more) waiting threads allowing them to continue.

Go doc: sync.Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.

Condition Variables (Chapter 30)

func NewCond(mutex Locker) *Cond
    func (*Cond) Wait()
    func (*Cond) Signal()

Wait():

- Wait() assumes that the associated mutex is locked when Wait() is called
- Wait() will atomically:
  - Release the mutex lock, then
  - Put calling goroutine to sleep
- When goroutine wakes up after being signaled by some other goroutine
  - Must re-acquire mutex lock before resuming after the line of Wait()

Signal(): Always hold the lock when calling

Rule: Hold the lock when calling Signal() and Wait().

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Producer about to produce

Consumer got the lock to access “items”

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Has produced but cannot put the item in "items" yet. Needs the lock!

No item to consume, must wait

Condition Variables (Chapter 30)

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine producer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Go routine consumer

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Producer checks if “items” is filled (it is not)

Wait() puts the consumer in block state, and releases the lock (we don’t see that)

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Producer adds an item to "items"

Wait() puts the consumer in block state, and releases the lock (we don't see it)

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Producer signals to wake up the consumer and still holds the lock

Wait() puts the consumer in block state, and releases the lock (we don't see it)

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Producer signals to wake up consumer and still holds the lock

Consumer gets woken up, but needs to get the lock again! (we don't see that)

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Producer signals to wake up consumer and still holds the lock

Consumer still waits the lock to get released (still in wait)

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Producer releases the lock

Consumer gets the lock inside wait() (we don't see that)

Condition Variables (Chapter 30)

Go routine producer

```go
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```go
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```go
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

The producer can now access "items"

The consumer releases the lock

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

The producer can now add produced items to “items”

The consumer needs the lock to check if there are items to consume

Condition Variables (Chapter 30)

```go
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine producer

```go
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Go routine consumer

```go
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

The producer signals, but no one is waiting anyways

The consumer needs the lock to check if there are items to consume

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

The producer releases the lock

The consumer needs the lock to check if there are items to consume

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

The producer tries to acquire the lock again

The consumer consumes and signal, but no one is waiting anyways

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

Now if the items is filled again!

Imagine: The producer happens to be fast !

Imagine:

The consumer happens to be slow

Condition Variables (Chapter 30)

Go routine producer

```go
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```go
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```go
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

The producer waits, and within wait releases the lock

The consumer can get the lock and consume

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

The producer goes to wait, and within wait releases the lock

The consumer can get the lock, consume, and signal again.

Condition Variables (Chapter 30)

Go routine producer

```
var cond = sync.NewCond(&Mu)
var items = make([]int, 0)
```

Go routine consumer

```
func produce() {
    prod_count := 0
    for prod_count < 100 {
        item := rand.Intn(100)
        cond.L.Lock()
        if len(items) == MAX_ITEMS {
            //producer waiting
            cond.Wait()
        }
        items = append(items, item)
        prod_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

```
func consume() {
    consume_count := 0
    for consume_count < 100 {
        cond.L.Lock()
        if len(items) == 0 {
            //consumer waits
            cond.Wait()
        }
        item := items[0]
        items = items[1:]
        consume_count += 1
        cond.Signal()
        cond.L.Unlock()
        time.Sleep(some periode)
    }
    Wg.Done()
}
```

The point: if items is neither filled or empty, they just compete on accessing “items”. Otherwise they can wait() (sleep/block) until they can continue working.

More efficient than just using locks

Condition Variables (Chapter 30)

Discuss this code

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

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

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

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

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

Semaphores (Chapter 31)

Definition

Aim: Single primitive for all things related to synchronization called **Semaphore.**

Replacement for Locks and Condition variables

Semaphores (Chapter 31)

Definition

Locks and Condition variables capabilities in one abstraction called **Semaphore**

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

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

Invariant

The negative number tells us about the number of waiting threads

Semaphores (Chapter 31)

31.2 Binary Semaphore (Locks)

The usage depends on how the semaphore is initiated

What **X** should be if we only want 1 thread at the time in the critical section?

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

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

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

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

Semaphores (Chapter 31)

31.2 Binary Semaphore (Locks)

The usage depends on how the semaphore is initiated

What **X** should be if we only want 1 thread at the time in the critical section? **X=1**

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

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

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

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

Semaphores (Chapter 31)

31.2 Binary Semaphore (Locks)

Example with 1 thread

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

Figure 31.4: **Thread Trace: Single Thread Using A Semaphore**

Semaphores (Chapter 31)

31.2 Binary Semaphore (Locks)

Example with 2 threads

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: 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()` returns	Running
0		Ready	`(crit sect)`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`sem_post()` returns	Running

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

Semaphores (Chapter 31)

31.3 Semaphore for Ordering

Usage pattern (Similar to condition variables):

- One thread waiting for something to happen
- Another thread making that something happen

Semaphores (Chapter 31)

31.3 Semaphore for Ordering

Example: parent waiting for child. What should **X** be?

```
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;
}
```

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

Semaphores (Chapter 31)

31.3 Semaphore for Ordering

Example: parent waiting for child. What should **X** be? **X=0**

```
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;
}
```

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

Semaphores (Chapter 31)

31.4 Semaphore for Producer/Consumer Problem

Study this code!

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);        // Line P1
        sem_wait(&mutex);        // Line P1.5 (MUTEX HERE)
        put(i);                  // Line P2
        sem_post(&mutex);        // Line P2.5 (AND HERE)
        sem_post(&full);         // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);         // Line C1
        sem_wait(&mutex);        // Line C1.5 (MUTEX HERE)
        int tmp = get();         // Line C2
        sem_post(&mutex);        // Line C2.5 (AND HERE)
        sem_post(&empty);        // Line C3
        printf("%d\n", tmp);
    }
}
```

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

Semaphores (Chapter 31)

31.5 Reader-Writer locks

Is another general problem that can be solved by semaphores.

The idea is to exploit the nature of the operations.

Insert modifies a list

delete item modifies a list

read does not modify a list

Basically a lock for reading and a lock for writing and a careful in the way to acquire them and release them.

Semaphores (Chapter 31)

31.5 Reader-Writer locks

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int   readers;   // #readers 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);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->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);
}
```

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

Initiate two locks and a reader count

acquire basic lock and increase the number of readers. If there is a reader (==1) disable writing by acquiring write lock

acquire basic lock and decrease the number of readers. If there is no reader (==0) enable writing by releasing write lock

Point:

Disable writing if reading

Semaphores (Chapter 31)

31.6 The Dining Philosophers Problem

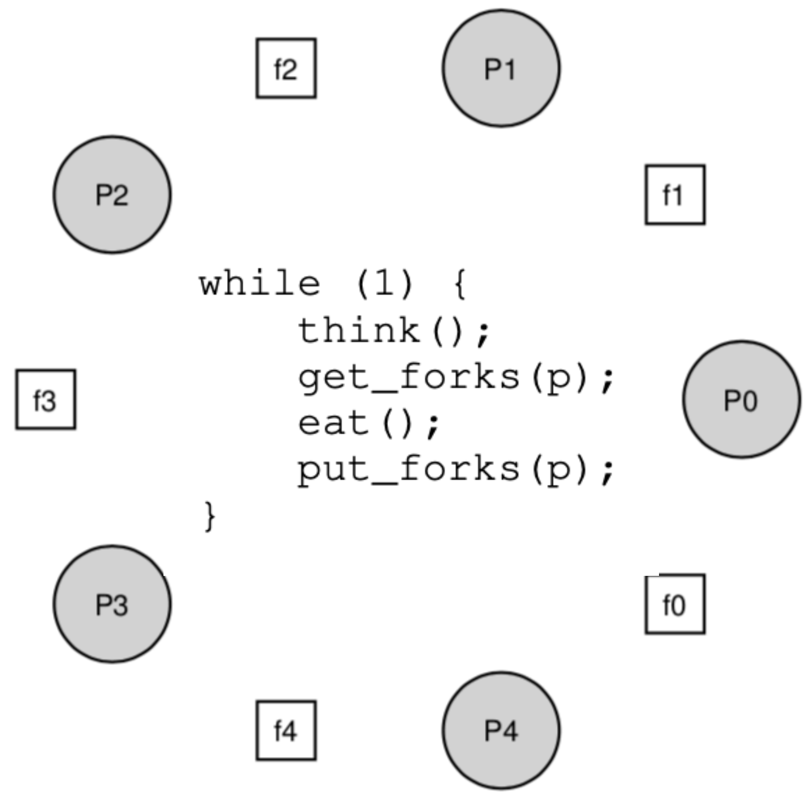
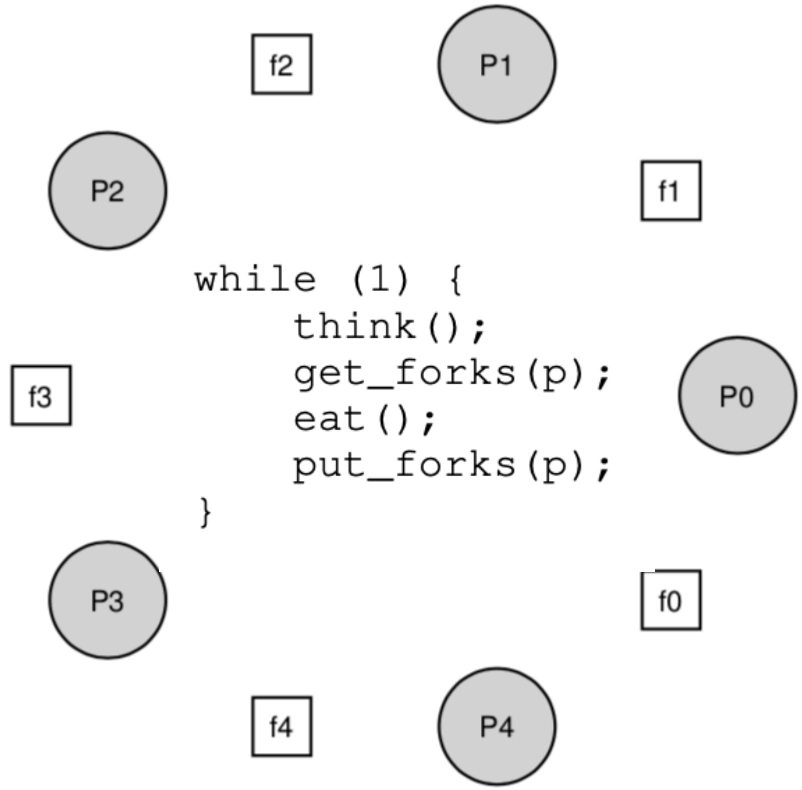


Figure 31.14: **The Dining Philosophers**

Semaphores (Chapter 31)

31.6 The Dining Philosophers Problem

```
void get_forks(int p) {
    sem_wait(&forks[left(p)])
    sem_wait(&forks[right(p)]
}

void put_forks(int p) {
    sem_post(&forks[left(p)])
    sem_post(&forks[right(p)]
}
```




```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

This solution does not work, **why?**

Figure 31.14: **The Dining Philosophers**

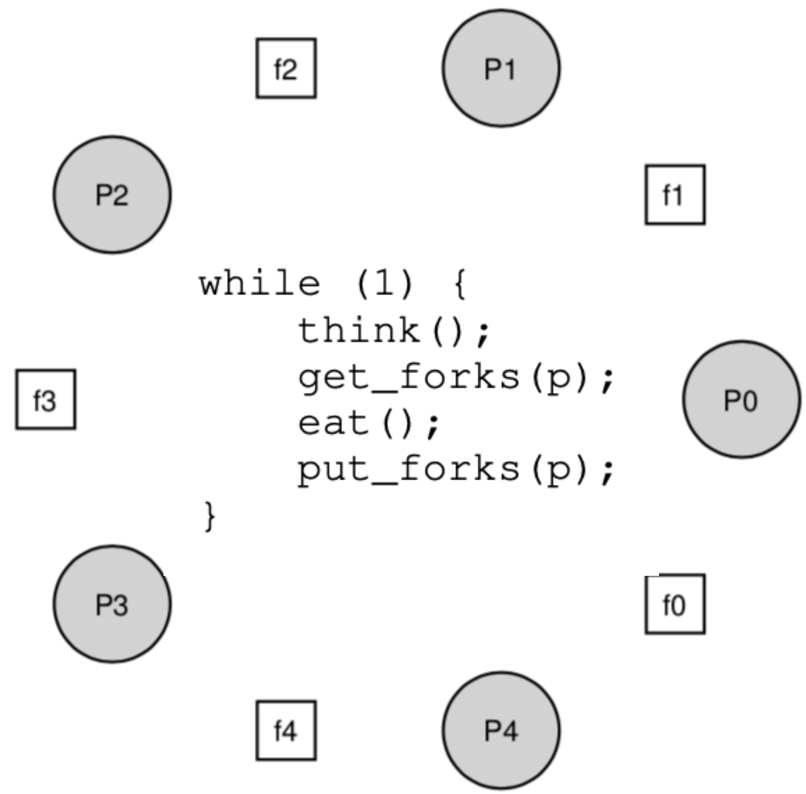
Semaphores (Chapter 31)

31.6 The Dining Philosophers Problem

```
void get_forks(int p) {
    sem_wait(&forks[left(p)])
    sem_wait(&forks[right(p)]
}

void put_forks(int p) {
    sem_post(&forks[left(p)])
    sem_post(&forks[right(p)]
}
```

This solution does not work, **why?**

Deadlock: Every p holds the left fork waiting for **the right one**




```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

Figure 31.14: **The Dining Philosophers**

Semaphores (Chapter 31)

31.6 The Dining Philosophers Problem

```
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]
        sem_wait(&forks[left(p)])
    } else {
        sem_wait(&forks[left(p)])
        sem_wait(&forks[right(p)]
    }
}
```

P=4 is just an agreed on number.

Break the cycle

This solution avoids deadlock. Starvation?

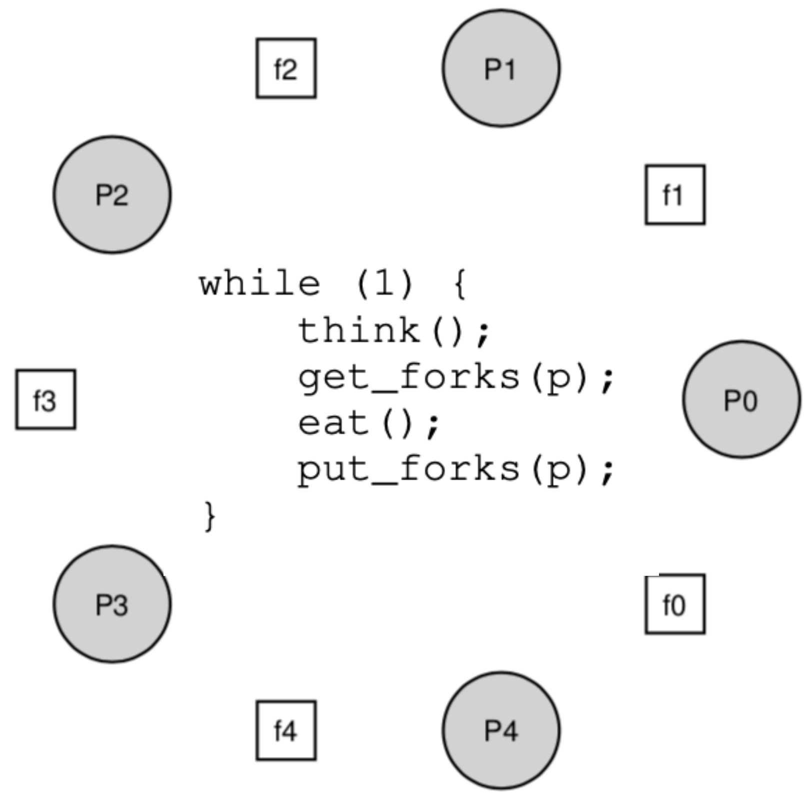


Figure 31.14: **The Dining Philosophers**

University of Stavanger

Semaphores (Chapter 31)

31.7 How to Implement Semaphores

```
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);
}

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

Making semaphore with a lock, a condition variable and a value, is easy.

Curiously, making condition variable from a semaphore is difficult.

Maybe not so general as we might think.

Semaphores (Chapter 31)

Summary

- Can be used instead of locks with an inherent mechanism of waking up and putting to sleep threads
- Can be used for the producer consumer problem rather than lock and condition variable
- Reader writer lock: basically a two locks application (with tricks)
- Dinning Philosophers: Basically a multi lock application (with tricks)
- Easy to make Semaphore from condition variable
- Hard to make Condition Variable from Semaphore

Common Concurrency Problems (Chapter 32)

Read it on your own, discuss and search.

Questions

Under which conditions deadlocks occurs?

Under which conditions starvation occurs?

What is a race condition?

What is atomicity violation?

What is order violation?

Go Channels

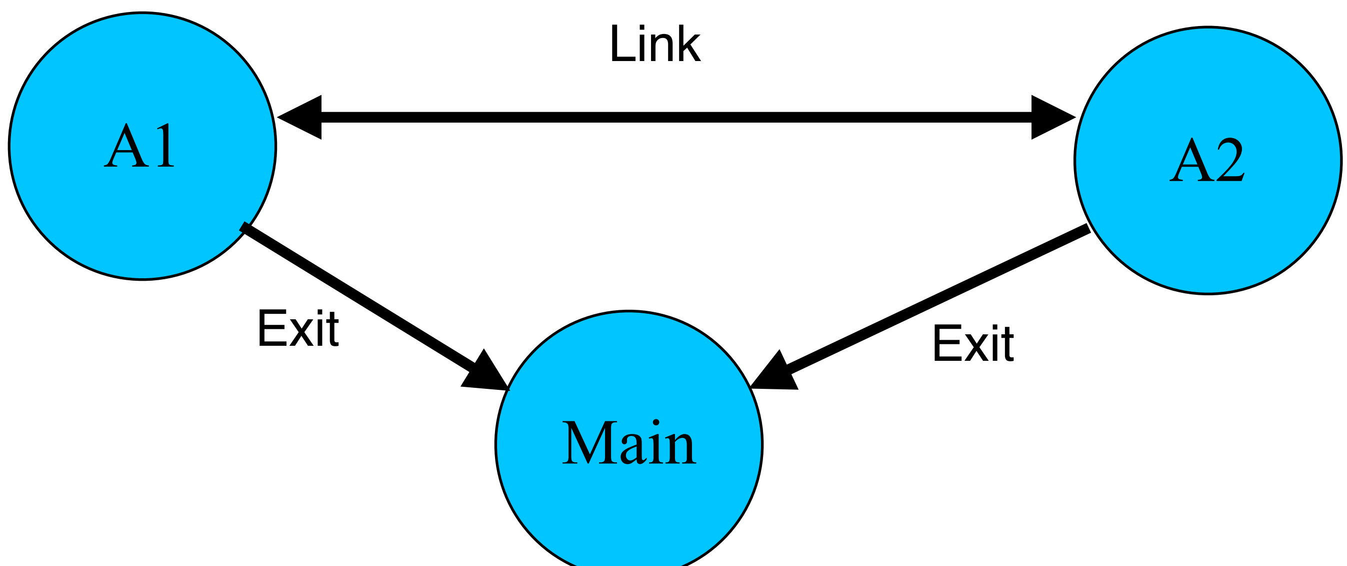
https://www.youtube.com/watch?v=f6kdp27TYZs&ab_channel=GoogleforDevelopers

Some exercises based on channels.

Go Channels

https://www.youtube.com/watch?v=f6kdp27TYZs&ab_channel=GoogleforDevelopers

Agents A1 and A2, will concurrently search for a value that is multiple of some X. The value must be less than some Max and the search should not last more than Y iterations



Go Channels

```
package demos
import (
    "fmt"
    "math/rand"
    "time"
)
type SearchingAgent struct {
    Link       chan int
    Multiple   int
    Iterations int
    Max        int
    Name       string
    Exit       chan int
}
func (ag *SearchingAgent) Search() {
   Implement this function!!!        }
func RunAgentDemo() {
    link := make(chan int)
    exit := make(chan int)
    agent1 := SearchingAgent{Name: "A1", Multiple: 110, Max: 3000, Iterations:
300, Link: link, Exit: exit}
    agent2 := SearchingAgent{Name: "A2", Multiple: 110, Max: 3000, Iterations:
300, Link: link, Exit: exit}
    go agent1.Search()
    go agent2.Search()
    <-exit
    <-exit
    fmt.Println("Done playing")
```

Go Channels

https://www.youtube.com/watch?v=f6kdp27TYZs&ab_channel=GoogleforDevelopers

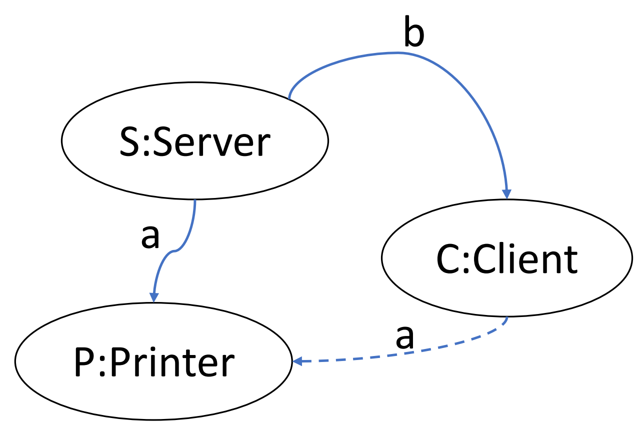
Some exercises based on channels.

Go Channels

Translation between Process Algebra and GO

Server (S): Send channel a on channel b
Client (C): Receive channel a and send through it
Printer (P): Receive in a and print content

System: $\underbrace{b<a>}_{S} \mid \underbrace{b(a).a<x>}_{C} \mid \underbrace{a(x)}_{P}$

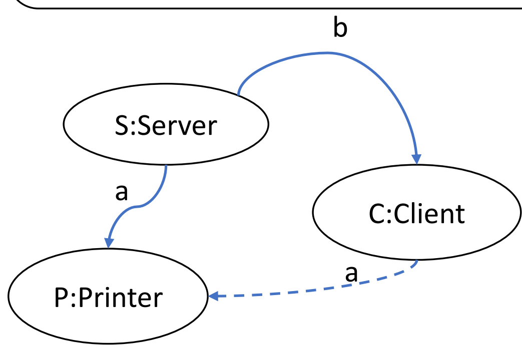


Go Channels

Translation between Process Algebra and GO

Server (S): Send channel a on channel b
Client (C): Receive channel a and send through it
Printer (P): Receive in a and print content

System: $\underbrace{b\langle a\rangle}_{S} \mid \underbrace{b(a).a\langle x\rangle}_{C} \mid \underbrace{a(x)}_{P}$




```go
func main() {
    S := func(a chan int, b chan chan int) {
        b <- a}
    P := func(a chan int) {
        fmt.Println("Printing ", <-a)}
    C := func(b chan chan int) {
        a := <-b
        a <- 11}
    system := func() {
        a := make(chan int)
        b := make(chan chan int)
        go P(a)
        go C(b)
        go S(a, b)}
    system()}
```