Lecture Notes

Semaphores are more powerful than locks due to their counters and FIFO waiting queues that prioritize fairness over progress

Semaphore Operations

- Operations work with both parts
    - an integer counter (initial value unspecified)
    - a FIFO waiting queue
- P - (test): “wait”
    - decrement counter, if count >= 0, return
    - if counter > 0, add process to waiting queue
- V - (raise): “signal”
    - increment counter
    - if counter >= 0, add process to waiting queue
- If you use the counter as values of 0 or 1, they are locks
- If the number is negative, it represents the number of threads waiting

Definition of serialize: turn the object into 1’s and 0’s so that they can be transmitted over to another computer
*Take something that is an object in memory and turn it into 1’s and 0’s so that it can be transmitted over to another device

Using Semaphores for Exclusion

- initialize semaphore to one
- use P/wait operation to take the lock
    - the first will succeed
    - subsequent attempts will block
- Use V/post operation to release the lock
    - restore semaphore count to non-negative
    - if any threads are waiting, unblock the first in line

Using semaphores for notifications

- initialize semaphore count to zero
    - count reflects # of completed events
- use P/wait operation to await completion
    - if already posted, it will return immediately
    - else all callers will block until post is called
- use post operation to signal completion
    - increment the count
    - if any threads are waiting, unblock the first in line
- one signal per wait: no broadcasts

Object Level Locking

- mutexes protect code critical sections
    - brief duration
    - other threads operating on the same data
    - all operating in a single address space
- persistent objects are more difficult
    - critical sections are likely to last much longer
    - many different programs can operate on them
    - may not even be running on a single computer
- solution: lock objects (rather than code)

Whole File Locking

- int flock(fd, operation)
    - supported operation
        - LOCK_SH : shared lock
        - LOCK_EX : exclusive lock (one at a time)
        - LOCK_UN : release a lock
    - lock is associated with an open file descriptor
        - lock is released when that file descriptor is closed
    - locking is purely advisory
        - does not prevent reads, writes, and unlinks

Advisory vs Enforced Locking

- Enforced Locking
    - done within the implementation of object methods
    - guaranteed to happen, whether or not user wants it
    - may sometimes be too conservative
    - stop read/write operations
- Advisory Locking
    - a convention that “good guys” are expected to follow
    - users expected to lock object before calling methods
    - gives users flexibility in what to lock
    - gives users more freedom to do it wrong
    - mutexes are advisory locks

Ranged File Locking

- int lockf(fd, cmd, offset, len)
    - enforced but are limited in scope
    - F_LOCK: get/wait for an exclusive lock
    - F_ULOCK: release a lock
    - F_TEST/F_TLOCK: test or non-blocking request
- lock is associated with a file descriptor
    - lock is released when file descriptor is closed

- locking may or may not be enforced
    - depending on the underlying file system

Chapter 29: Lock-Based Concurrent Data Structures

- Adding locks to a data structure to make it usable by threads make the structure **thread safe**

29.1: Concurrent Counters

- One of the simplest data structures is the counter

```
1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Figure 29.1: **A Counter Without Locks**

```
1  typedef struct __counter_t {
2      int             value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Figure 29.2: **A Counter With Locks**

- The concurrent counter follows a design pattern common to the simplest and most basic concurrent data structures
    - implements a single lock, which is acquired when calling a routine that manipulates that data structure -> released when returning from the call
- However, the implementation that we have of this current counter scales poorly (performance drop off) -> the more threads you have the longer the execution becomes
- Scalable Counting
    - Sloppy Counter
        - Works by representing a single logical counter via numerous local physical counters, one per CPU core, as well as a single global counter

- Ex: a machine with four CPUs has four local counters and a global counter
- There is also a lock for each local counter as well as one for the global counter
- Basic idea: Because each CPU can update their own counter without contention from other competing threads, they can efficiently increment/decrement this counter
- The local values are periodically transferred over to the global counter -> given core acquires the lock and increments the global counter's value while setting the local counter to zero
- Ex: what happens if we do the local-to-global transfer too often?
    - Not very scalable
- Ex: what happens if we do the local-to-global transfer not too often?
    - Very scalable but accuracy of the counter may be off (global counter)
- This allows us to scale sloppy counters very well

29.2: Concurrent Linked Lists

```
// basic node structure
typedef struct __node_t {
    int                 key;
    struct __node_t         *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
    node_t                *head;
    pthread_mutex_t     lock;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; // fail
    }
    new->key  = key;
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}

int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; // success
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; // failure
}
```

Figure 29.7: **Concurrent Linked List**

- A simple case would be to acquire the lock when an insert routine is called and to release the lock when an exit routine is called or if malloc fails (rare case)
- Scaling Linked Lists
    - **Hand-over-hand locking (lock coupling)**
        - Instead of having a single lock for the entire list, you instead add a lock per node of the list

- When traversing the list, the code first grabs the next node's lock and then releases the current node's lock

```
1  void List_Init(list_t *L) {
2      L->head = NULL;
3      pthread_mutex_init(&L->lock, NULL);
4  }
5
6  void List_Insert(list_t *L, int key) {
7      // synchronization not needed
8      node_t *new = malloc(sizeof(node_t));
9      if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // just lock critical section
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head   = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }
```

Figure 29.8: **Concurrent Linked List: Rewritten**

-
- There is still a significant amount of overhead for acquiring and releasing locks
- This makes the concurrency scalable, meaning that more threads can be supported but the increased overhead from locking and unlocking causes performance drop. Therefore, it is important to remember that concurrency doesn't necessarily ensure speed

29.3: Concurrent Queues

```
1  typedef struct __node_t {
2      int                 value;
3      struct __node_t    *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t             *head;
8      node_t             *tail;
9      pthread_mutex_t     headLock;
10     pthread_mutex_t     tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
20
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next  = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }
```

Figure 29.9: **Michael and Scott Concurrent Queue**

- two locks that are represented as head and tail locks
- The goal of these locks are to enable concurrency of enqueue and dequeue operations
    - enqueue will almost always access the tail lock (to add to the queue)
    - dequeue will almost always access the head of the lock (to pop)
- Michael and Scott added a dummy node (allocated in the queue initialization code) which enables the separation of tail and head operations

29.4: Concurrent Hash Table

- This implementation of a hash table does not resize

```
1   #define BUCKETS (101)
2
3   typedef struct __hash_t {
4       list_t lists[BUCKETS];
5   } hash_t;
6
7   void Hash_Init(hash_t *H) {
8       int i;
9       for (i = 0; i < BUCKETS; i++) {
10          List_Init(&H->lists[i]);
11      }
12  }
13
14  int Hash_Insert(hash_t *H, int key) {
15      int bucket = key % BUCKETS;
16      return List_Insert(&H->lists[bucket], key);
17  }
18
19  int Hash_Lookup(hash_t *H, int key) {
20      int bucket = key % BUCKETS;
21      return List_Lookup(&H->lists[bucket], key);
22  }
```

Figure 29.10: **A Concurrent Hash Table**

-
- Uses a lock per hash bucket to enable concurrent operations to take place
- Tip: avoid premature optimization (Knuth's Law)
    - When building a concurrent data structure, start with the most basic approach -> add a single big lock to provide synchronized access
    - Be sure that the basic approach is correct before making optimizations

Chapter 30.2: Producer/Consumer (Bounded Buffer) Problem

- The producer/consumer problem was posed by Djikstra
- Imagine one or more consumer threads and one or more producer threads
    - Producers generate data items and place them in a buffer, consumers grab said items from the buffer and consume them in some way
    - Ex: In a multi-threaded web server, a producer puts HTTP requests into a work queue, consumer threads take request out of this queue and process them
    - grep foo file.txt | wc -l (producer is the grep and consumer is the wc)
    - Because this bounded buffer is a shared resource, we must require synchronized access to it, or else a race condition will arise
- A Broken Solution
    - Simply putting a lock around the critical section where we put and get stuff from the queue doesn't work -> we need a condition variable
    - When a producer wants to fill the buffer, it waits for it to be empty while the consumer waits for the fullness

```
1   cond_t  cond;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);            // p1
8           if (count == 1)                        // p2
9               Pthread_cond_wait(&cond, &mutex); // p3
10          put(i);                                // p4
11          Pthread_cond_signal(&cond);            // p5
12          Pthread_mutex_unlock(&mutex);          // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);            // c1
20          if (count == 0)                        // c2
21              Pthread_cond_wait(&cond, &mutex); // c3
22          int tmp = get();                       // c4
23          Pthread_cond_signal(&cond);            // c5
24          Pthread_mutex_unlock(&mutex);          // c6
25          printf("%d\n", tmp);
26      }
27  }
```

Figure 30.6: **Producer/Consumer: Single CV And If Statement**

- With just a single consumer and producer, the above code works
- Ex: Two consumers (Tc1 and Tc2) and one producer (Tp1)
  - After the producer wakes Tc1 but before Tc1 ever ran, the state of the bounded buffer changed due to Tc2 because signaling a thread only wakes them up
  - The second consumer sneaks in and takes the lock before the first consumer can and therefore Tc1 tries to consume but cannot
  - We need a way to let the first consumer thread know that the resource is not available

- A Better, But Still Broken Solution
  - Fix is easy: change the if statement to a while loop -> always remember to use while loops with condition variables
  - Bug could occur in which all three threads from the previous example could be sleeping at the same time
  - Signaling is clearly needed, but must be more directed as the consumer should not wake other consumers, only producers, and vice versa
- The Single Buffer Producer/Consumer Solution
  - The solution here is once again a small one: use two condition variables
  - producer threads wait on the condition **empty and signals fill**
  - consumer threads wait on the condition **fill and signal empty**

```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Figure 30.10: **Producer/Consumer: Two CVs And While**

- The Correct Producer/Consumer Solution
  - We want to enable more concurrency and efficiency by adding more buffer slots so that multiple values can be produced before sleeping and multiple values can be consumed before sleeping
  - This approach is more efficient because it reduces context switches


```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Figure 30.11: **The Correct Put And Get Routines**

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        while (count == MAX)                   // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                // p4
        Pthread_cond_signal(&fill);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            Pthread_cond_wait(&fill, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&empty);           // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.12: **The Correct Producer/Consumer Synchronization**

  - The buffer structure itself and the corresponding put and get methods are different

- A producer only sleeps if all buffers are currently filled and a consumer sleeps only when all buffers are currently empty

30.3: Covering Conditions

- Example: Multi-threaded memory allocation library

```
1   // how many bytes of the heap are free?
2   int bytesLeft = MAX_HEAP_SIZE;
3
4   // need lock and condition too
5   cond_t  c;
6   mutex_t m;
7
8   void *
9   allocate(int size) {
10      Pthread_mutex_lock(&m);
11      while (bytesLeft < size)
12          Pthread_cond_wait(&c, &m);
13      void *ptr = ...; // get mem from heap
14      bytesLeft -= size;
15      Pthread_mutex_unlock(&m);
16      return ptr;
17  }
18
19  void free(void *ptr, int size) {
20      Pthread_mutex_lock(&m);
21      bytesLeft += size;
22      Pthread_cond_signal(&c); // whom to signal??
23      Pthread_mutex_unlock(&m);
24  }
```

Figure 30.13: **Covering Conditions: An Example**

-
- When a thread calls into the memory allocation code, it might have to wait in order for more memory to become free
- Conversely, when a thread frees memory, it signals that more memory is free -> which waiting thread should be woken up?
- Solution: call pthread_cond_broadcast() which wakes up all waiting threads

Chapter 31: Semaphores

31.1: Semaphores: A Definition

- **A semaphore is an object with an integer value** that we can manipulate with two routines: sem_wait() and sem_post()
- Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value
- sem_init(&s, 0, 1)
    - The second value 0 indicates that is shared across threads, and the third value 1 indicates the initial semaphore value

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

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

-

- sem_wait() will either return right away or it will cause the caller to suspend the execution waiting for a subsequent post
- sem_post() does not wait for some particular condition to hold like sem_wait() -> increments semaphore value and if there are one or more threads waiting, wakes one of them up
- The value of the semaphore, when negative, is equal to the number of waiting threads

31.2: Binary Semaphores (Lock)

- The initial value of the semaphore should be one
    - imagine a scenario with two threads -> call sem_wait() -> wait if the value is not greater than or equal to 0, since its not -> resume execution -> thread 0 is now free to enter the critical section, if no other thread attempts to call the critical section, when it exits the thread will call sem_post and increment the counter back to 1.
- Because locks only have two states, held and not held, this is sometimes called a binary semaphore

31.3: Semaphores For Ordering

- Example: A thread may wish to wait for a list to become non-empty so it can delete an element from it -> we must use the semaphores as an ordering primitive

```
1   sem_t s;
2
3   void *
4   child(void *arg) {
5       printf("child\n");
6       sem_post(&s); // signal here: child is done
7       return NULL;
8   }
9
10  int
11  main(int argc, char *argv[]) {
12      sem_init(&s, 0, X); // what should X be?
13      printf("parent: begin\n");
14      pthread_t c;
15      Pthread_create(&c, NULL, child, NULL);
16      sem_wait(&s); // wait here for child
17      printf("parent: end\n");
18      return 0;
19  }
```

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

- The implementation of this is fairly simple -> the parent thread calls sem_wait() and the child thread calls sem_post when it is done executing the critical section
    - However, unlike binary semaphores (locks), we should not set the original value of the semaphore to 1 -> set to 0 instead

    - Assume parent creates the child but the child does not run yet -> we want the parent to wait for the child to run first -> the only way this will happen is if the value of the semaphore is initially zero so when we call sem_wait() it will be negative.

31.4: The Producer/Consumer Problem

- Possible race condition: If there are two producer threads, if one thread stores one value in the buffer, and before the counter is incremented another thread can store the value at the same location, essentially overwriting the data
- Solution: Adding Mutual Exclusion
    - If we simply add locks around the put/get part of the critical section -> deadlock can occur
- **Avoiding deadlock**
    - From previous example: imagine two threads
        - consumer gets to run first -> acquires the mutex, then calls sem_wait() on the full semaphore -> because there is no data yet, this call causes the consumer to block and thus yield the CPU -> while consumer still holds the lock
        - Producer then runs -> has data to produce -> calls sem_wait() on the binary mutex semaphore -> but the lock is already held so the thread must wait
    - Classic deadlock: The consumer holds the mutex and is waiting for someone to signal full -> however, the producer is waiting for the mutex although it can signal that it is full
- **Working Solution**
    - To solve this problem, we simply must reduce the scope of the lock
    - Move the mutex acquire and release to be just around the critical section -> full and empty wait and signal code is left outside

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&empty);           // Line P1
9          sem_wait(&mutex);           // Line P1.5 (MOVED MUTEX HERE...)
10         put(i);                     // Line P2
11         sem_post(&mutex);           // Line P2.5 (... AND HERE)
12         sem_post(&full);            // Line P3
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);            // Line C1
20         sem_wait(&mutex);           // Line C1.5 (MOVED MUTEX HERE...)
21         int tmp = get();            // Line C2
22         sem_post(&mutex);           // Line C2.5 (... AND HERE)
23         sem_post(&empty);           // Line C3
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33     // ...
34 }
```

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

-

31.5: Reader-Writer Locks

- We want a flexible locking primitive that admits that different data structure accesses might require different kinds of locking
- Imagine a concurrent list operations that includes inserts and simple lookups
    - while insert actually changes the state of the list, lookups simply just read the data structure
    - As long as we ensure that no writing is going on, we can concurrently run lookups

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

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

-
- use rwlock_acquire_writelock() to acquire a write lock, and rwlock_release_writelock() to release it
- Once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too
    - however any thread that wishes to acquire the write lock must wait until all of the reading is done
    - the last one that exits the critical section calls sem_post()
- In this implementation, it will be relatively easy for the readers to starve the writers
- Reader-writer locks should be used carefully because they tend to add overhead to a system

31.6: The Dining Philosophers

- Assume there are five “philosophers” sitting around a table
    - Between each pair of philosophers is a single fork (and thus, five total)
    - The philosophers each have times where they think and they don’t need forks and times where they eat and need the fork
    - In order to eat, the philosophers need to have two forks, the one on the left and right

- Deadlock: If each philosopher happens to grab the fork on their left before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another
- Solution: Breaking the Dependency
    - Change how forks are acquired by at least one of the philosophers
    - Assume that one philosopher always acquires the forks in a different order from the rest

31.7: How to Implement Semaphores

```
1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```

Figure 31.16: **Implementing Zemaphores With Locks And CVs**

- Key difference between this implementation and Dijkstra's
    - We don't maintain that the value of the semaphore, when negative describes how many threads are waiting

31.8: Summary

- Semaphores are a powerful and flexible primitive for writing concurrent programs. Some programmers use them exclusively, shunning locks and condition variables