Semaphores

CMPU 334 – Operating Systems
Jason Waterman
Semaphore

- Created by Dijkstra to be a single primitive for synchronization
  - Can be used as both locks and condition variables
- An object with an integer value associated with it
- We can manipulate with two routines
  - `sem_wait()`
  - `sem_post()`
- Must initialize before use

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

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

- `sem_wait(sem_t *s)`
  - **Decrement**s the integer value of the semaphore by 1
  - If the value is **negative** the semaphore will wait
    - It will cause the caller to suspend execution waiting for a subsequent post
    - Similar to a `cond_wait()`
  - If the value of the semaphore (after the decrement) is positive or zero, return immediately

- `sem_post(sem_t *s)`
  - **Increment**s the value of the semaphore by 1
  - If there is any threads waiting on the semaphore, **wake** one of them up

- When negative, the value of the semaphore is the number of threads waiting on the semaphore

- **Both operations happens atomically**
Using a Semaphore as a Lock

• Semaphores can be used to provide mutual exclusion

```c
1    sem_t m;
2    sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4    sem_wait(&m);
5    //critical section here
6    sem_post(&m);
```

• What should the semaphore above be initialized to?
  • The semaphore should be initialized to 1

• This is known as a binary semaphore
  • Works the same as a lock

<table>
<thead>
<tr>
<th>Value of Semaphore</th>
<th>Thread 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>call sem_wait()</td>
</tr>
<tr>
<td>1</td>
<td>sem_wait() returns</td>
</tr>
<tr>
<td>0</td>
<td>(crit sect)</td>
</tr>
<tr>
<td>0</td>
<td>call sem_post()</td>
</tr>
<tr>
<td>1</td>
<td>sem_post() returns</td>
</tr>
</tbody>
</table>
### Thread Trace: Two Threads Using A Semaphore

<table>
<thead>
<tr>
<th>Value</th>
<th>Thread 0</th>
<th>State</th>
<th>Thread 1</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>call sem_wait()</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td>sem_wait() returns</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td>(crit set: begin)</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td>Interrupt; Switch → T1</td>
<td>Ready</td>
<td></td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Ready</td>
<td>call sem_wait()</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Ready</td>
<td>decrement sem</td>
<td>Running</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>Ready</td>
<td>(sem &lt; 0)→sleep</td>
<td>sleeping</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>Running</td>
<td>Switch → T0</td>
<td>sleeping</td>
</tr>
<tr>
<td>-1</td>
<td>(crit sect: end)</td>
<td>Running</td>
<td></td>
<td>sleeping</td>
</tr>
<tr>
<td>-1</td>
<td>call sem_post()</td>
<td>Running</td>
<td></td>
<td>sleeping</td>
</tr>
<tr>
<td>0</td>
<td>increment sem</td>
<td>Running</td>
<td></td>
<td>sleeping</td>
</tr>
<tr>
<td>0</td>
<td>wake(T1)</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td>sem_post() returns</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td>Interrupt; Switch → T1</td>
<td>Ready</td>
<td></td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td>Ready</td>
<td></td>
<td>sem_wait() returns</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td>Ready</td>
<td></td>
<td>(crit sect)</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td>Ready</td>
<td></td>
<td>call sem_post()</td>
<td>Running</td>
</tr>
<tr>
<td>1</td>
<td>Ready</td>
<td></td>
<td>sem_post() returns</td>
<td>Running</td>
</tr>
</tbody>
</table>
Semaphores as Condition Variables

• What should $X$ be?
  • The value of semaphore should be set to is 0

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

A Parent Waiting For Its Child

The execution result

parent: begin
child
parent: end

10/24/2023
CMPU 334 -- Operating Systems
The Producer/Consumer (Bounded-Buffer) Problem

- This works for a single producer and consumer
- What is we have multiple producers or consumers?
  - We have a race condition
  - Need to add mutual exclusion for the calls to `put()` and `get()`

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

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

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    return tmp;
}

sem_t empty; /* empty slots */
sem_t full;  /* items in buffer */

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

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

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX);
    sem_init(&full, 0, 0);
    // ...
}
```
A Solution: Adding Mutual Exclusion (Incorrectly)

```c
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(&mutex);
9          sem_wait(&empty);
10         put(i);
11         sem_post(&full);
12         sem_post(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);
20          sem_wait(&full);
21          int tmp = get();
22          sem_post(&empty);
23          sem_post(&mutex);
24          printf("%d\n", tmp);
25      }
26  }
```

• What is wrong with the above implementation?
  • Consider a consumer thread trying to consume when the buffer is empty
  • It will be blocked on `sem_wait()` but it still holds the lock
    • Deadlock!
A Working Semaphore Solution

```c
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);
9           sem_wait(&mutex);
10          put(i);
11           sem_post(&mutex);
12           sem_post(&full);
13       }
14   }
15
16   void *consumer(void *arg) {
17       int i;
18       for (i = 0; i < loops; i++) {
19           sem_wait(&full);
20           sem_wait(&mutex);
21           int tmp = get();
22           sem_post(&mutex);
23           sem_post(&empty);
24           printf("%d\n", tmp);
25       }
26   }
27
28   int main(int argc, char *argv[]) {
29       // ...
30       sem_init(&empty, 0, MAX);
31       sem_init(&full, 0, 0);
32       sem_init(&mutex, 0, 1);
33       // ...
34   }
```
Reader-Writer Locks

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

This special type of lock is known as a **reader-writer lock**
A Reader-Writer Locks

- Only a single writer can acquire the lock
- Once a reader has acquired a read lock
  - More readers will be allowed to acquire the read lock too
  - A writer will have to wait until all readers are finished
- What about fairness?
  - It would be relatively easy for reader to starve writer
  - A more sophisticated scheme could prevent this

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

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

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->writelock); // first reader acquires writelock
  sem_post(&rw->lock);
}

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

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

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

• Assume there are five “philosophers” sitting around a table
  • Between each pair of philosophers is a single fork (five total)
  • The philosophers each have times where they think, and don’t need any forks, and times where they eat
  • In order to eat, a philosopher needs two forks, both the one on their left and the one on their right
The Dining Philosophers (Cont.)

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

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

---

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

---

// helper functions
int left(int p) { return p; }
int right(int p) {
    return (p + 1) % 5;
}
The Dining Philosophers (Cont.)

• We need some **semaphores**, one for each fork: `sem_t forks[5]`

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

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

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

• Deadlock occurs
  • If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right
  • Each will be stuck **holding one fork** and waiting for another, **forever**
A Solution: Breaking The Dependency

• Change how forks are acquired
  • Let’s assume that philosopher 4 acquires the forks in a different order

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

• There is no situation where each philosopher grabs one fork and is stuck waiting for another
• The cycle of waiting is broken
Thread throttling

• Used to prevent “too many” threads from doing something all at once
• Limit the number of concurrent threads with a threshold semaphore
  • Throttling, a form of admission control
• Example:
  • Hundreds of threads solving a parallel problem
  • One area of the code is memory-intensive
  • If all threads are allowed into this area, machine will start swapping and thrashing
• Solution:
  • Add a semaphore initialized to the maximum number of threads allowed in the memory-intensive area
  • Put a sem_wait() and sem_post() around the memory-intensive area
How To Implement Semaphores

- Build our own version of semaphores called **Zemaphores**
- Doesn't maintain the invariant that a negative value is a count of threads waiting on the semaphore
  - The value is never lower than zero
  - This behavior is easier to implement and matches the current Linux implementation

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

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

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

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

• We need to synchronize for correctness
  • Unsynchronized code can cause incorrect behavior
  • But too much synchronization means threads spend a lot of time waiting, not performing useful work

• Getting synchronization right is hard
  • Testing isn’t enough
  • Need to assume worst case: all interleavings are possible

• How to choose between locks, semaphores and condition variables?
  • Locks are very simple and suitable for many cases
    • Issues: Maybe not the most efficient solution
    • E.g., can’t allow multiple readers but one writer inside a standard lock
  • Condition variables allow threads to sleep until an event occurs
    • Just remember the state of the world might have changed since the signal was called
  • Semaphores provide general functionality
    • But can be tricky to get correct