Semaphores, Advanced Locks, and Synchronization Problems

Semaphore Definition

- An object with an integer value
  - We can manipulate with two routines; `sem_wait()` and `sem_post()`
  - Initialization

```
#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 Interfaces

- `sem_wait()`

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

- If the value of the semaphore was *one* or *higher* when called `sem_wait()`, **return right away**
- It will cause the caller to suspend execution waiting for a subsequent post
- When negative, the value of the semaphore is equal to the number of waiting threads

Semaphore Interfaces

- sem_post()

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

  - Simply **increments** the value of the semaphore
  - If there is a thread waiting to be woken, **wakes** one of them up

Binary Semaphores (Locks)

- What should **x** be?
  - The initial value should be **1**

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

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

Single Thread Using a Semaphore

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

Two Threads Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit set: 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

Semaphores As Condition Variables

```c
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;
}
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- What should **x** be?
  - The value of semaphore should be set to is **0**

Parent Waiting For Child (Case 1)

- The parent call `sem_wait()` before the child has called `sem_post()`

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*`(Child exists; is runnable)`*	Ready
0	`call sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem < 0)→sleep`	sleeping		Ready
-1	*`Switch→Child`*	sleeping	`child runs`	Running
-1		sleeping	`call sem_post()`	Running
0		sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post() returns`	Running
0		Ready	*`Interrupt; Switch→Parent`*	Ready
0	`sem_wait() retruns`	Running		Ready

Parent Waiting For Child (Case 2)

- The child runs to completion before the parent call `sem_wait()`

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() retruns`	Running		Ready

The Producer/Consumer (Bounded-Buffer) Problem

- **Producer**: `put()` interface
  - Wait for a buffer to become *empty* in order to put data into it
- **Consumer**: `get()` interface
  - Wait for a buffer to become *filled* before using it

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

void put(int value) {
    buffer[fill] = value;      // line f1
    fill = (fill + 1) % MAX;  // line f2
}

int get() {
    int tmp = buffer[use];    // line g1
    use = (use + 1) % MAX;    // line g2
    return tmp;
}
```

The Producer/Consumer (Bounded-Buffer) Problem

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&empty);          // line P1
            put(i);                    // line P2
            sem_post(&full);           // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
            sem_wait(&full);           // line C1
            tmp = get();               // line C2
            sem_post(&empty);          // line C3
            printf("%d\n", tmp);
    }
}
…
```

First Attempt: Adding the Full and Empty Conditions

The Producer/Consumer (Bounded-Buffer) Problem

```
int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX);       // MAX buffers are empty to begin with…
    sem_init(&full, 0, 0);          // … and 0 are full
    // …
}
```

First Attempt: Adding the Full and Empty Conditions (Cont.)

- Imagine that `MAX` is greater than 1
  - If there are multiple producers, race condition can happen at line p*1*
  - It means that the old data may be overwritten
- We've forgotten **mutual exclusion**
  - Filling of a buffer and incrementing of the index into the buffer is in a critical section

Adding Mutual Exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&mutex);          // line p0 (NEW LINE)
            sem_wait(&empty);          // line p1
            put(i);                    // line p2
            sem_post(&full);           // line p3
            sem_post(&mutex);          // line p4 (NEW LINE)
    }
}

(Cont.)
```

Adding Mutual Exclusion (Incorrectly)

Adding Mutual Exclusion

```
(Cont.)
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);          // line c0 (NEW LINE)
        sem_wait(&full);           // line c1
        int tmp = get();           // line c2
        sem_post(&empty);          // line c3
        sem_post(&mutex);          // line c4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Adding Mutual Exclusion (Incorrectly)

Adding Mutual Exclusion

- Imagine two thread: one producer and one consumer
  - The consumer **acquire** the `mutex` (line c0)
  - The consumer **calls** `sem_wait()` on the full semaphore (line c1)
  - The consumer is **blocked** and **yield** the CPU
    - The consumer <u>still holds the mutex</u>!
  - The producer **calls** `sem_wait()` on the binary `mutex` semaphore (line p0)
  - The producer is now **stuck** waiting too
  - This is a classic deadlock situation

Finally, A Working Solution

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);        // line p1
        sem_wait(&mutex);        // line p1.5 (MOVED MUTEX HERE…)
        put(i);                  // line p2
        sem_post(&mutex);        // line p2.5 (… AND HERE)
        sem_post(&full);         // line p3
    }
}

(Cont.)
```

Adding Mutual Exclusion (Correctly)

Finally, A Working Solution

```
(Cont.)
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&full);            // line c1
            sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE…)
            int tmp = get();            // line c2
            sem_post(&mutex);           // line c2.5 (… AND HERE)
            sem_post(&empty);           // line c3
            printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with …
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // …
}
```

Adding Mutual Exclusion (Correctly)

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-write 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

```
typedef struct _rwlock_t {
    sem_t lock;       // binary semaphore (basic lock)
    sem_t writelock;  // used to 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);
    …
```

A Reader-Writer Locks

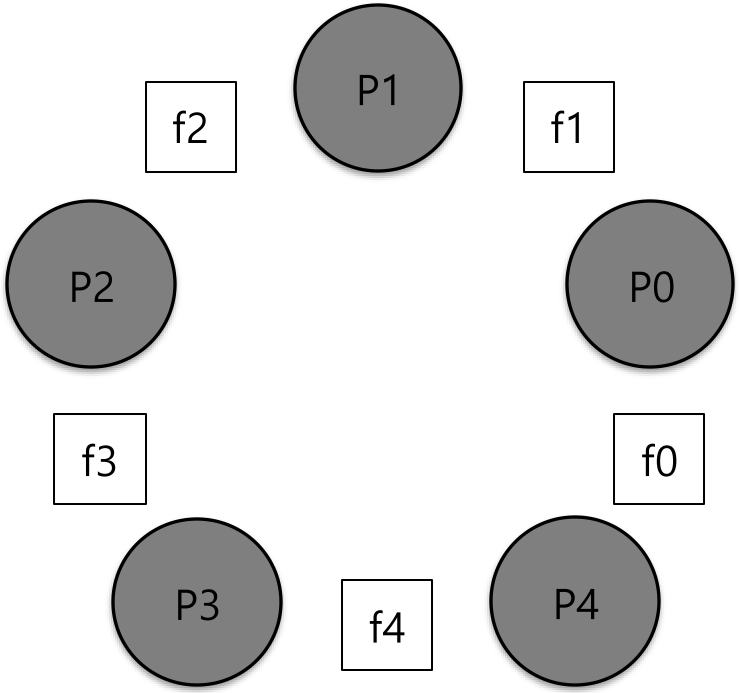
```
        rw->readers++;
        if (rw->readers == 1)
                sem_wait(&rw->writelock); // first reader acquires writelock
        sem_post(&rw->lock);
    }

    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);
    }
```

A Reader-Writer Locks

- The reader-writer locks have fairness problem
  - It would be relatively easy for reader to **starve writer**
  - How to prevent more readers from entering the lock once a writer is waiting?

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 contention for these forks**



The Dining Philosophers

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

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

Basic loop of each philosopher

```
// helper functions
int left(int p) { return p; }

int right(int p) {
        return (p + 1) % 5;
}
```

Helper functions (Downey's solutions)

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

The Dining Philosophers

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

```
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 occur!
  - 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 acquire the forks in a *different order*

```
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**

How To Implement Semaphores

- Build our own version of semaphores called Zemaphores

```
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);
}
…
```

How To Implement Semaphores

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

- Zemaphore don't maintain the invariant that the value of the semaphore reflects the number of waiting threads
  - The value never be lower than zero
  - This behavior is **easier** to implement and **matches** the current Linux implementation