25. Semaphores

semaphore : a single primitive for all things related to synchronization
one can use semaphore as both locks and condition variables.

Semaphores : a Definition

semaphore : object with an integer value that we can manipulate with two routines

- sem_wait() / sem_post()

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

Figure 31.1: **Initializing A Semaphore**

- 2nd : 0 , semaphore is shared between threads in the same process
- 3rd : 1 , 1 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
}
```

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

Binary Semaphores (Locks)

```
sem_t m;
sem_init(&m, 0, X); // init to X; what should X be? : 1

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

Figure 31.3: **A Binary Semaphore (That Is, A Lock)**

Setting the Value of a Semaphore

consider the number of resources you are willing to give away immediately after initializaion

Semaphores For Ordering

```
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?    : 0
    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**

using a semaphore as an ordering primitive (similar to use of condition variables)

The Producer / Consumer (Bounded Buffer) Problem

First Attempt

```
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;
}
```

Figure 31.9: **The Put And Get Routines**

what if 2 threads run put() at the same time before one increase fill? : overwritten

```
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 tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);          // Line C1
        tmp = get();              // Line C2
        sem_post(&empty);         // Line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    // ...
}
```

Figure 31.10: **Adding The Full And Empty Conditions**

A solution: adding mutual exclusion

```
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)
    }
}

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);
    }
}
```

Deadlock
: consumer holds lock first
=> wait forever

Figure 31.11: **Adding Mutual Exclusion (Incorrectly)**

At Last, a Working Solution

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);       // Line P1
        sem_wait(&mutex);       // Line P1.5 (lock)
        put(i);                 // Line P2
        sem_post(&mutex);       // Line P2.5 (unlock)
        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 (lock)
        int tmp = get();        // Line C2
        sem_post(&mutex);       // Line C2.5 (unlock)
        sem_post(&empty);       // Line C3
        printf("%d\n", tmp);
    }
}
```

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

mutex acquire and release to be just around the critical section
*better put that in the put() & get() function

Simple and Dum can be better

with locking, sometimes a simple spin lock works best, because it is easy to implement and fast.
Complex can mean slow. => Alwasy try the simple & dumb approach first

Reader-Writer Locks

different data structure accesses might require different kinds of locking
as long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

```
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);
}
```

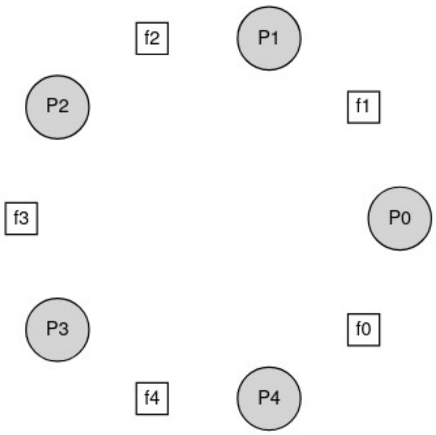
Figure 31.13: **A Simple Reader-Writer Lock**

- Once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too.
- any thread that wishes to acquire the write lock will have to wait until all readers are finished
- the last one to exit the critical section calls sem_post() on “writelock” and thus enables a waiting writer to acquire the lock.

Negatives : fairness. It would be relatively easy for readers to starve writers.

Reader-writer locks often add more overhead (especially with more sophisticated implementations)
=> thus do not end up speeding up performance as compared to just using simple and fast locking primitives

The Dinning Philosophers




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

Figure 31.14: **The Dining Philosophers**

Broken solution

```
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)]);
}
```

Figure 31.15: **The `get_forks()` And `put_forks()` Routines**

what if everyone grab their left fork? => deadlock

A solution : Breaking the Dependency

change how forks are acquired by at least one of the philosophers
: last philosopher grabs right fork before left.

```
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)]);
    }
}
```

Figure 31.16: **Breaking The Dependency In `get_forks()`**

Thread Throttling (admission control)

how can a programmer prevent “too many” threads from doing something at once and bogging the system down? (ex. All threads enter the memory-intensive region at the same time)
: decide a threshold for “too many”
=> use a semaphore to limit the number of threads concurrently executing.

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);
}
```

Figure 31.17: **Implementing Zemaphores With Locks And CVs**

building condition variables out of semaphores is a much trickier proposition
Semaphore wait/post can only “hint” that something happened, but they can’t directly enforce *when* or *under what condition* a thread should wake without extra machinery.

Be Careful with Generalization

: Don’t generalize. Generalizations are generally wrong.
One could view semaphores as a generalization of locks and condition variables.
However, given the difficulty of realizing a condition variable on top of a semaphore, perhaps this generalization is not as general as you might think.