Semaphores

```
#include <semaphore.h>
```

- a semaphore holds an integer count
  - represents how many resources are available
  - all operations are atomic

this follows the pthreads library API documentation – the book's alternative (`wait` decrements, blocking when value < 0; `post` increments, unblocking a waiting thread) has the same public semantics

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict abs_timeout);
```

  - *wait* attempts to decrement the value
    - `sem_wait` – if the current value is 0, the calling thread blocks until the decrement is possible
    - `sem_trywait` – if the current value is 0, return an error (`EAGAIN`)
    - `sem_timedwait` – if the current value is 0, block until the value becomes > 0 or the timeout is reached

```
int sem_post(sem_t *sem);
```

  - *post* increments the value
    - if this makes the value > 0, unblock one of the waiting threads

Producer/Consumer

```
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;
}
```

```
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
    // ...
}
```

does this work?

- for `MAX=1`
- for `MAX>1`

the bodies of `put` and `get` are critical sections and with `MAX` > 1, the semaphores only provide ordering (not mutual exclusion)

Second Try

add a binary semaphore as a lock for mutual exclusion

```
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;
}
```

```
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);
    }
}
```

does this work?

if a thread blocks waiting on either `empty` or `full`, it is with the `mutex` lock held – no one else can acquire it to rectify the buffer condition

→ deadlock

Third Try

adjust the order of the `wait`s

```
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;
}
```

```
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);
    }
}
```

does this work?

limit the `mutex` lock to just the critical section – a thread waiting on `empty` or `full` does not hold any locks, and a thread holding the `mutex` lock will release it itself before possibly blocking on anything else

Reader-Writer Locks

- support concurrent reads but exclusive writes
  - many readers at once but only one writer
  - common access pattern for data structures
- idea
  - write lock to provide mutual exclusion for read vs write
    - lock is acquired when a reader or writer enters
    - lock is released when writer or last reader exit
  - a counter to keep track of how many readers so the last reader to leave can release the write lock
  - a mutex lock to make read-write lock operations atomic

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);
}
```

does this work?

but readers can starve writers

Best Practices

- reader-writer locks add overhead compared to single mutex lock for access and may not end up speeding up performance

> TIP: SIMPLE AND DUMB CAN BE BETTER (HILL'S LAW)
> You should never underestimate the notion that the simple and dumb approach can be the best one. With locking, sometimes a simple spin lock works best, because it is easy to implement and fast. Although something like reader/writer locks sounds cool, they are complex, and complex can mean slow. Thus, always try the simple and dumb approach first.