Semaphore

Semaphore

- A semaphore is fundamentally an integer whose value is never allowed to fall below 0.
- There are two operations on a semaphore: `wait` and `post`. The `post` operation increment the semaphore by 1, and
- the `wait` operations does the following:
  - ✓ If the semaphore has a `value > 0`, the semaphore is decremented by `1`.
  - ✓ If the semaphore has value `0`, the caller will be blocked (busy-waiting or more likely on a queue) until the semaphore has a value larger than `0`, and then it is decremented by `1`.
- We declare a semaphore as:

```
sem_t sem;
#include<semaphore.h>
```

Semaphore

```
int sem_init(sem_t * sem, int pshared, unsigned int
value);
```

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

- This initializes the semaphore `*sem`. The initial value of the semaphore will be `value`.
- If the `pshared` argument has a `non-zero` value, then the semaphore is shared between processes.
- If the `pshared` argument is `zero`, then the semaphore is shared between threads of the process
- On success, the return value is `0`, and on failure, the return value is `-1`.
- An attempt to initialize a semaphore that has already been initialized results in undefined behavior.

Semaphore

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

- The `sem_wait()` function locks the semaphore only if the semaphore is currently **not locked** and its value is **non-zero**.
- If the semaphore value is currently **zero**, then the calling thread will not return from the call to `sem_wait()` until it either locks the semaphore or the call is interrupted by a signal.
- The `sem_trywait()` function locks the semaphore referenced by `sem` only if the semaphore is currently **not locked**; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.
- On success, the return value is `0`, and on failure, the return value is `-1`.

Semaphore

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

- The `sem_post()` function unlocks the semaphore and the semaphore value is simply incremented.
- If successful, the `sem_post()` function returns `zero`; otherwise the function returns `-1`.

Semaphore

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

- The `sem_destroy()` function is used to destroy semaphore indicated by `sem`.
- It is safe to destroy an initialised semaphore upon which no threads are currently blocked. The effect of destroying a semaphore upon which other threads are currently blocked is undefined.
- Upon successful completion, a value of `zero` is returned. Otherwise, a value of `-1` is returned.

Semaphore

```
int sem_getvalue(sem_t *sem, int *sval);
```

- The `sem_getvalue()` function updates the location referenced by the `sval` argument to have the value of the semaphore referenced by `sem` without affecting the state of the semaphore.
- Upon successful completion, the function returns a value of `zero`. Otherwise, the function returns a value of `-1`.

Example: Thread printing its name

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>
#define THREAD_NUM 4
sem_t semaphore;
void* routine(void* args) {
    sem_wait(&semaphore);
    sleep(1);
    printf("Thread %d\n", (int)args);
    sem_post(&semaphore);
    free(args);
}
```

```
int main(int argc, char *argv[]) {
pthread_t th[THREAD_NUM];
sem_init(&semaphore, 0, 4);
int i;
for (i = 0; i < THREAD_NUM; i++) {
    int* a = malloc(sizeof(int));
    *a = i;
     if(pthread_create(&th[i], NULL, &routine, a)!=0)
    {
        perror("Failed to create thread");
     }
}
    for (i = 0; i < THREAD_NUM; i++) {
      if (pthread_join(th[i], NULL) != 0) {
         perror("Failed to join thread");
      }
    }
sem_destroy(&semaphore);
return 0;
}
```

Example: Semaphores for ordering

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>

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;
}
```

Example: Producer-Consumer

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>

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

```
Void* consumer (void *arg)
{
    int i;
    for (i=0; i < loops; i++) {
        sem wait (&full);
        sem_wait(&mutex);
        int tmp get ();
        sem_post(&mutex);
        sem_post(& empty);
        printf("%d\n", tmp);
    }
}
```

Producer-Consumer:

```
#define THREAD_NUM 8
sem_t semEmpty;
sem_t semFull;
pthread_mutex_t mutexBuffer;
int buffer[10];
int count = 0;
void* producer(void* args) {
    while (1) {
        // Produce
        int x = rand() % 100;
        sleep(1);
        // Add to the buffer
        sem_wait(&semEmpty);
pthread_mutex_lock(&mutexBuffer);
        buffer[count] = x;
                    count++;
pthread_mutex_unlock(&mutexBuffer);
        sem_post(&semFull);
    }
}
```

```
void* consumer(void* args) {
    while (1) {
        int y;
        // Remove from the buffer
        sem_wait(&semFull);
pthread_mutex_lock(&mutexBuffer);
        y = buffer[count - 1];
        count--;
        pthread_mutex_unlock(&mutexBuffer);
        sem_post(&semEmpty);

        // Consume
        printf("Got %d\n", y);
        sleep(1);
    }
}
int main(int argc, char* argv[]) {
    srand(time(NULL));
    pthread_t th[THREAD_NUM];
    pthread_mutex_init(&mutexBuffer, NULL);
    sem_init(&semEmpty, 0, 10);
    sem_init(&semFull, 0, 0);
    int i;
    for (i = 0; i < THREAD_NUM; i++) {
        if (i > 0) {
            if (pthread_create(&th[i], NULL, &producer, NULL) != 0) {
                perror("Failed to create thread");
            }
        } else {
            if (pthread_create(&th[i], NULL, &consumer, NULL) != 0) {
                perror("Failed to create thread");
            }
        }
    }
    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    sem_destroy(&semEmpty);
    sem_destroy(&semFull);
    pthread_mutex_destroy(&mutexBuffer);
    return 0;
}
```

Dining philosophers problem:

```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N–1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      1                   /* philosopher is trying to get forks */
#define EATING      2                   /* philosopher is eating */

typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */


void philosopher(int i)                 /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}
```

Dining philosophers problem:

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```