Concurrency #5:

Semaphores

(Lecture-Semaphores-Intro)

-> give def (handout)

-> show [README]

lock
join
zemaphore
reader-writer

(next time: PC + deadlock)

HAND THIS OUT

```
//
// SEMAPHORE: PSEUDO-CODE
//
sem_init(sem_t *s, int initvalue) {
    s->value = initvalue;
}

sem_wait(sem_t *s) {
    while (s->value <= 0)
        put_self_to_sleep(); // put self to sleep
    s->value--;
}

sem_post(sem_t *s) {
    s->value++;
    wake_one_waiting_thread(); // if there is one
}

//
// IMPORTANT: each is done atomically
// (i.e., body of post() and wait() happen all at once)
//
```

```
semaphores
- sem-def.c          : how a semaphore works

using semaphore as a lock (BINARY semaphore):
- sem-lock.c         : code w/o locks
- sem-lock-works.c   : code w/ semaphore as lock

using semaphore to signal:
- sem-join.c         : join problem again
- sem-join-works.c   : join problem using semaphores

writing your own semaphore
- zemaphore.c        : using cond vars + locks
- sem-myown-lock.c   : using zemaphores as locks

bounded buffer problem:
- sem-pc.c           : same old problem
- sem-pc-signals.c   : uses signaling to solve problem
- sem-pc-locks.c     : adds a lock too
- sem-pc-works.c     : actually works
- sem-myown-pc.c     : using zemaphores in PC problem

read/write lock (if time)
- rwlock.c           : working thing
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "mythreads.h"

#define PMAX (100)

volatile static int counter = 0;

sem_t lock;

void *
worker(void *arg) {
    int i;
    Sem_wait(&lock);
    for (i = 0; i < 1e6; i++)
        counter++;
    Sem_post(&lock);
    return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: sem-lock <numthreads>\n");
        exit(1);
    }
    int threads = atoi(argv[1]);
    if (threads > PMAX) {
        fprintf(stderr, "%d threads is the max\n", PMAX);
        exit(1);
    }

    pthread_t pid[PMAX];

    Sem_init(&lock, 1);

    printf("begin\n");

    int i;
    for (i = 0; i < threads; i++)
        Pthread_create(&pid[i], NULL, worker, NULL);

    for (i = 0; i < threads; i++)
        Pthread_join(pid[i], NULL);

    printf("counter: %d\n", counter);

    return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include "mythreads.h"

sem_t done;

void *
child(void *arg) {
    sleep(5);
    printf("child\n");
    Sem_post(&done);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    Sem_init(&done, 0);
    Pthread_create(&p, NULL, child, NULL);
    Sem_wait(&done);

    printf("parent: end\n");
    return 0;
}
```

```
#ifndef __ZEMAPHORE_h__
#define __ZEMAPHORE_h__

#include "mythreads.h"

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

void
Zem_init(Zem_t *z, int value)
{
    z->value = value;
    Cond_init(&z->cond);
    Mutex_init(&z->lock);
}

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

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


#endif // __ZEMAPHORE_h__
```

```
typedef struct _rwlock_t { sem_t writelock; sem_t lock; int readers;
                         } rwlock_t;
void rwlock_init(rwlock_t *L) {
    L->readers = 0;
    sem_init(&L->lock, 1);
    sem_init(&L->writelock, 1); }
void rwlock_acquire_readlock(rwlock_t *L) {
    sem_wait(&L->lock);                       // ra1
    L->readers++;                             // ra2
    if (L->readers == 1)                      // ra3
        sem_wait(&L->writelock);              // ra4
    sem_post(&L->lock); }                     // ra5
void rwlock_release_readlock(rwlock_t *L) {
    sem_wait(&L->lock);                       // rr1
    L->readers--;                             // rr2
    if (L->readers == 0)                      // rr3
        sem_post(&L->writelock);              // rr4
    sem_post(&L->lock); }                     // rr5
void rwlock_acquire_writelock(rwlock_t *L) { sem_wait(&L->writelock);}
void rwlock_release_writelock(rwlock_t *L) { sem_post(&L->writelock);}
```