CS 341

#18 Barriers.
Deadlock. The Reader-Writer Problem

Challenge 1: "Make a barrier using only one mutex lock() and unlock() call!"

"Impossible! Line 2 is a Critical Section, if a thread has locked the mutex..."

But here is an awful solution. (Why is this a 'poor' solution?)

```
01  void barrier() {
        lock
02     count ++
       unlock
03     while( count != N) {}
04
05  }
```

→ this will only work if CPU update the cache for this threads when others update it.

2. When is disabling interrupts a solution to the Critical Section Problem?

```
pthread_mutex_lock     => { disable interrupts on the CPU }
pthread_mutex_unlock  => {enable interrupts on the CPU }
```

Are there limitations to this approach? may work for 1 CPU PC?

3. Challenge II: Create a barrier using each of the following lines once.
All 5 threads must call barrier before they all continue.

```
int remain =5;  earlier... sem_init(&s,0,_0_?)
void barrier() { ... Rearrange the following!
   sem_wait(&s);
   sem_post(&s);
   remain --;
   pthread_mutex_lock(&m);
   pthread_mutex_unlock(&m);
   if(remain)
}
```

avoid hold & wait ...

lock m
remain --
unlock
if (remain)
    sem_wait
sem_post → the last thread add one pizza, wake up one, and the new thread calls another thread and so on

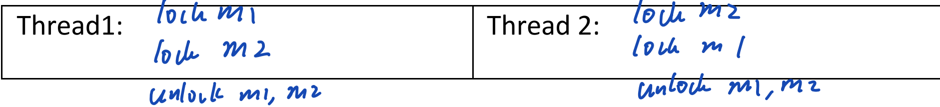
4. Is there a Race condition?

lock pleaseStop = 1 ② p_cond_broadcast(&cv) ③ unlock	lock ① while(!pleaseStop) p_cond_wait(&cv,&m) ④ unlock

fix

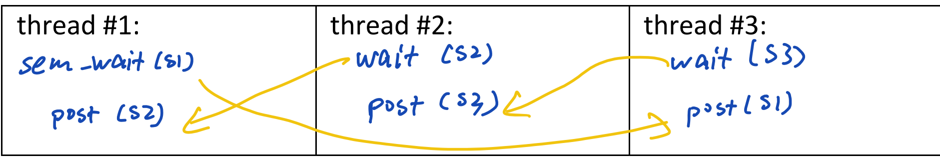
5. Deadlock: " waiting for an event that never happens "

Use two mutex locks and two threads to create an example of deadlock



Thread1: lock m1 lock m2 unlock m1, m2	Thread 2: lock m2 lock m1 unlock m1, m2

Use three counting semaphores and three threads to deadlock 3 threads



thread #1:	thread #2:	thread #3:
sem_wait (s1)	wait (s2)	wait (s3)
post (s2)	post (s3)	post (s1)

- Must deadlock involve threads? What about single-threaded processes?

6. What is the Resource Allocation Graph for deadlock detection?

7. The Reader Writer problem

A common problem in many different system applications

read_database(table, query) {...}	update_row(table, id, value) {...}
cache_lookup(id) {...}	cache_modify(id, value) {...}

8. ReaderWriter locks are useful primitives & included in the pthread library!

```
01  pthread_rwlock_t lock;

02  p_rwlock_init
03  p_rwlock_wrlock  -> exclusive
04  p_rwlock_rdlock
05  p_rwlock_unlock
```

```
01  cache_lookup(id) {
02       p...rdlock(...)
03       read from resource     } -> lots of readers
04       p...unlock(...)
05       return result
06  }
```

while writers writing, will readers simply wait?

CS241: synch. skills and the ability to *build* these! Along the way, also learn to reason about, develop and fix multi-threaded code

9. ~~ Welcome to the *Reader Writer* Game Show! ~~

Contestant #1

```
p_mutex_t *readlock,*writelock
readlock=malloc(sizeof p_mutex_t)
writelock=malloc(sizeof p_mutex_t)
p_m_init(readlock,NULL)
P_m_init(writelock,NULL)

read() {
lock(readlock)        ) too slow since
// do read            ) no sychronous
unlock(readlock)      ) reading
}
```

```
write() {
lock(writelock)
lock(readlock)
// do writing
unlock(readlock)
unlock(writelock)
}
```

Is #1 a Solution? Problems?

Contestant #2

never starts writing if multiple reading happens

```
bool reading=0, writing=0
```

```
read() {
  while(writing) {} <- race

  reading = true
   // do reading here
  reading = false
}
```

```
write() {
 ->while(reading||writing) {}

  writing = true
  // do writing here
  writing = false
}
```

Is #2 a Solution? Problems?

Contestant #3

```
read(){
 lock(&m)
 while (writing)
    cond_wait(cv,m)

 reading++

/* Read here! */

 reading--
 cond_signal(cv)
 unlock(&m)
```

```
write(){
 lock(&m)
 while (reading||writing)
    cond_wait(cv,m)

 writing++

/* Write here! */

 writing--;
 cond_signal(cv)
 unlock(&m)
```

Is #3 a Solution? Problems?