# CS 537
# Lecture 18
# Semaphores

Michael Swift





## Threading Review

```
int value = 0;

/* the thread */
void * runner(void * param)
{
  value = 5;
  pthread_exit(0);
}

int main(int argc, char * argv[])
{
  int pid;
  pthread_t tid;

  pid = fork();
  if (pid == 0) { /* child process */
    pthread_create(&tid, NULL /* attributes */, runner, NULL /* arg */);
    pthread_join(tid, NULL);
    printf("Child: value = %d\n",value); /* LINE A */
  } else { /* parent process */
    wait(NULL); /* wait for child */
    printf("Parent: value = %d\n",value); /* LINE B */
  }
}
```

What gets printed?





## Lock Granularity

Locks protect data, not code!

**Global lock**

```
Lock g_lock;

Withdraw(account, amount) {
  acquire(g_lock);
  balance = account.balance;
  balance -= amount;
  account.balance = balance
  release(g_lock);
}
Deposit(account, amount) {
  acquire(g_lock);
  balance = account.balance;
  balance += amount;
  account.balance = balance
  release(g_lock);
}
```

**Fine grain locks**

```
Struct account_s {
  int balance;
  Lock a_lock;
}

Withdraw(account, amount) {
  acquire(account.a_lock);
  balance = account.balance;
  balance -= amount;
  account.balance = balance
  release(account.a_lock);
}
```





## Problems with fine grained locks

```
Struct account_s {
  int balance;
  Lock a_lock;
}

transfer(acc1,acc2, amount) {
  acquire(acc1.a_lock);
  acquire(acc2.a_lock);
  acc1.balance -= amount;
  acc2.balance += amount;
  release(acc1.a_lock);
  release(acc2.a_lock);
}
```

- Problem:
  - T1: transfer(mike,jill)
  - T2: transfer(jill,mike)

T1: acquire(mike.a_lock);
T2: acquire(jill.a_lock);
T1: acquire(jill.a_lock)
T2: acquire(mike.a_lock);

## Disabling Interrupts

- An alternative:

```
struct lock {
}
void acquire(lock) {
   cli();   // disable interrupts
}
void release(lock) {
  sti();    // reenable interupts
}
```

- Can two threads disable interrupts simultaneously?
- What's wrong with interrupts?
  - only available to kernel (why? how can user-level use?)
  - insufficient on a multiprocessor
    - back to atomic instructions
- Like spinlocks, only use to implement higher-level synchronization primitives





## Problems with spinlocks

- Horribly wasteful!
  - if a thread is spinning on a lock, the thread holding the lock cannot make process
- How did lock holder yield the CPU in the first place?
  - calls yield( ) or sleep( )
  - involuntary context switch
- Only want spinlocks as primitives to build higher-level synchronization constructs
- SOLUTION: **blocking locks**
  - suspend thread on a wait queue until lock released
  - More later…





## Blocking in Semaphores

- Each semaphore has an associated queue of processes/threads
  - when wait() is called by a thread,
    - if semaphore is "available", thread continues
    - if semaphore is "unavailable", thread blocks, waits on queue
  - signal() opens the semaphore
    - if thread(s) are waiting on a queue, one thread is unblocked
    - if no threads are on the queue, the signal is remembered for next time a wait() is called
- In other words, semaphore has history
  - this history is a counter
  - if counter falls below 0 (after decrement), then the semaphore is closed
    - wait decrements counter
    - signal increments counter





## Example: bounded buffer problem

- AKA producer/consumer problem
  - there is a buffer in memory
    - with finite size N entries
  - a producer process inserts an entry into it
  - a consumer process removes an entry from it
- Processes are concurrent
  - so, we must use synchronization constructs to control access to shared variables describing buffer state

## Producer/Consumer: Single Buffer

- Simplest case:
  - Single producer thread, single consumer thread
  - Single shared buffer between producer and consumer
- Requirements
  - Consumer must wait for producer to fill buffer
  - Producer must wait for consumer to empty buffer (if filled)
- Requires 3 semaphores
  - emptyBuffer: Initialize to ???
  - fullBuffer: Initialize to ???
  - mutex: Initialize to ???

Producer

```
While (1) {
       wait(&emptyBuffer);
       wait(&mutex);

       Fill(&buffer);
       signal(&mutex);
       signal(&fullBuffer);
}
```

Consumer

```
While (1) {
       wait(&fullBuffer);
       wait(&mutex);
       Use(&buffer);
       signal(&mutex);
       signal(&emptyBuffer);
}
```





## Example: Readers/Writers

- Basic problem:
  - object is shared among several processes
  - some read from it
  - others write to it
- We can allow multiple readers at a time
  - why?
- We can only allow one writer at a time
  - why?





## Readers/Writers using Semaphores

```
semaphore mutex  ; controls access to readcount
semaphore wrt    ; control entry to a writer or first reader
int readcount    ; number of readers

write process:
   wait(wrt)     ; any writers or readers?
     <perform write operation>
   signal(wrt)   ; allow others

read process:
   wait(mutex)   ; ensure exclusion
        readcount = readcount + 1 ; one more reader
        if (readcount == 1) wait(wrt) ; if we're the first, synch with
   writers
   signal(mutex)
        <perform reading>
   wait(mutex)   ; ensure exclusion
        readcount = readcount - 1 ; one fewer reader
        if (readcount = 0) signal(wrt) ; no more readers, allow a
   writer
   signal(mutex)
```





## Readers/Writers notes

- Note:
  - the first reader blocks if there is a writer
    - any other readers will then block on mutex
  - if a writer exists, last reader to exit signals waiting writer
    - can new readers get in while writer is waiting?
  - when writer exits, if there is both a reader and writer waiting, which one goes next is up to scheduler

## Example: implement join

- Goal: use semaphores to wait for a thread to complete.
  - What should sem be initialized to?

```
Sempahore sem;

main() {
  sem_init(sem, ??);

  create_thread(myfunc);
  wait(sem);
}
```

```
myfunc() {
  do_work ();
  signal(sem);
}
```