Synchronization

- Why do you need synchronization?
  - Data race: Multiple threads/processes try to read/write the same shared data
- When do you need synchronization?
  - Accessing shared data
- What is shared data?
- How to synchronize?
  - Critical section
- What is needed to guarantee critical sections
  - Four conditions
A real life example for Synchronization

- Roommate A comes back, checks the refrigerator. If there is no milk, he then goes out to buy milk
- Roommate B also does the same thing
- If the two are not synchronized, what’s the result?
- How can you solve this problem?
Review: Critical Section

Process {
    while (true) {
      ENTER CRITICAL SECTION
      Access shared variables; // Critical Section;
      LEAVE CRITICAL SECTION
      Do other work
    }
}

Which four conditions?
A turn-based implementation

```java
int turn = 1;

while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}

while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called alternation
(1) Does it satisfy mutex?
(2) Does it work (satisfy all 4 conditions)?
Using Locks

withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}

- What happens when blue tries to acquire the lock?
- Why is the “return” outside the critical section? What if not?
- What happens when a third thread calls acquire?
What could be the result?

- Initially x=0
- x is a global variable

**Thread 1**

```c
int i; //private
for (i=0; i<3; i++)
    x++;   
```

**Thread 2**

```c
int j; //private
for (j=0; j<3; j++)
    x++;   
```

After thread 1 and 2 finish, what can be the value of x
Implementing Locks (1)

- How do we implement locks?
- Here is one attempt:

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

- This is called a **spinlock** because a thread spins waiting for the lock to be released
- Does this work?
Using Test-And-Set

- Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
};
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- Does it work?
- Does it work on multiprocessors?
Other Similar Hardware Instruction

- Swap = TSL

```c
void Swap (char* x, *y);
\ All done atomically
{
    char temp = *x;
    *x = *y;
    *y = temp
}
```
Do it yourself

- Use Swap to implement lock acquire and release

```c
struct lock {
    int held = 0;
}

void acquire (lock) {
    Fill in code here
}

void release (lock) {
    Fill in code here
}
```
Semaphore Questions

- Are there any problems that can be solved with counting semaphores that cannot be solved with mutex semaphores?

- If a system provides only mutex semaphores, can you use it to implement a counting semaphores?
  - Do it yourself