2.2.3 Synchronization

- In real life, "synchronization" means that you want to do things at the same time
- In computer science, "synchronization" could meant the above, *OR*, it means that you want to *prevent* do things at the same time

Mutual Exclusion

Also see https://en.wikipedia.org/wiki/Therac-25

Threads and Mutual Exclusion

Thread 1:	Thread 2:
`x = x+1;`	`x = x+1;`

- looks like it doesn't matter how you execute, x will be incremented by 2 in the end
  - choices are
    - thread 1 executes x = x+1 then thread 2 executes x = x+1
    - thread 2 executes x = x+1 then thread 1 executes x = x+1
  - are there other choices?

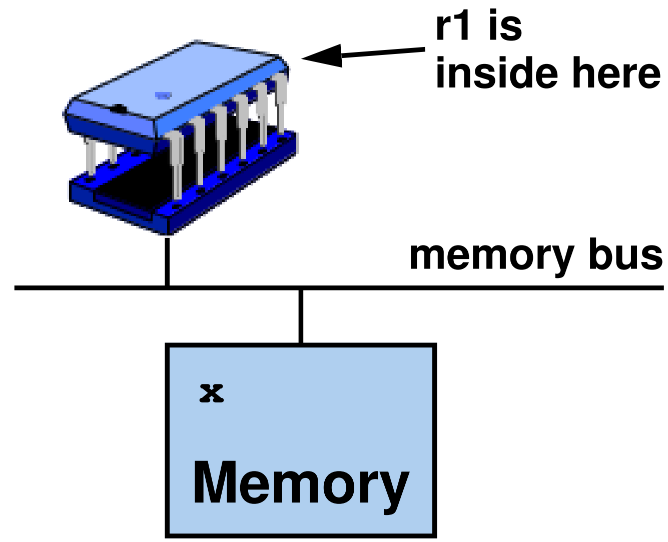
Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
 /*
  ld   r1,x
  add  r1,1
  st   r1,x
  */
```

Thread 2:

```
x = x+1;
 /*
  ld   r1,x
  add  r1,1
  st   r1,x
  */
```



- **Unfortunately, machines do not execute high-level language statements**
  - they execute machine instructions
  - now if thread 1 executes the first (or two) machine instructions
  - context switch can happen (to run a different thread)
    - this *can* happen if you have a *preemptive scheduler*
  - then thread 2 executes all 3 machine instructions
  - then later thread 1 executes the remaining machine instructions
  - `x` would have only increased by 1

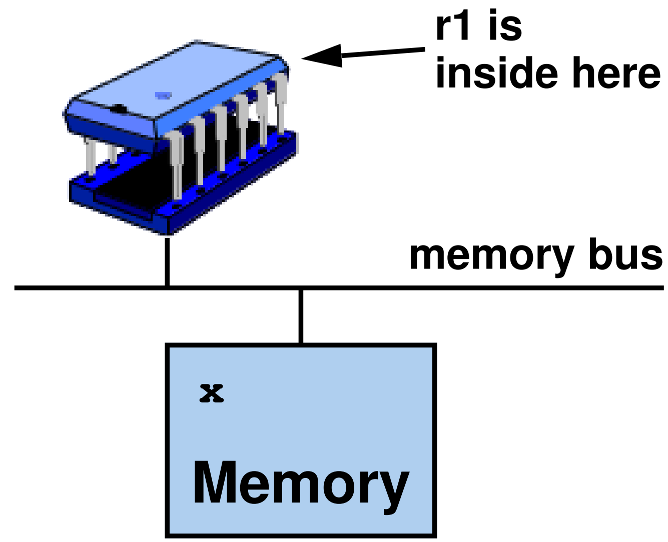
Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
 /*
   ld    r1,x
   add   r1,1
   st    r1,x
   */
```

Thread 2:

```
x = x+1;
 /*
   ld    r1,x
   add   r1,1
   st    r1,x
   */
```



- We want `x=x+1` to be executed *atomically*
  - *atomically* means that the 3 machine instructions are *locked* together
    - if you execute the first machine instruction, you must execute all 3 *without interruption*
  - *atomicity* is an *abstraction*
    - it's important to understand *exactly* what it means to be *atomic*

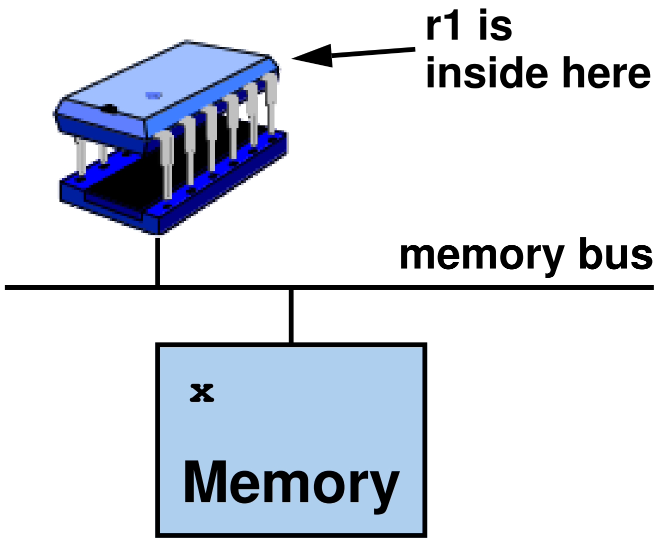
Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
 /*
  ld   r1,x
  add  r1,1
  st   r1,x
  */
```

Thread 2:

```
x = x+1;
 /*
  ld   r1,x
  add  r1,1
  st   r1,x
  */
```



- ***Atomic*** **operation**
  - if you execute the first machine instruction, can the CPU go do something else (e.g., handle a hardware interrupt)?
    - yes!
  - what does atomic really mean if you can go do something else?
    - it means *atomic*, *with respect to* the *variables involved*
      - in this example, it's just `x`
      - you *can* do something else that does not involve `x`
    - more involved in general as we will see soon

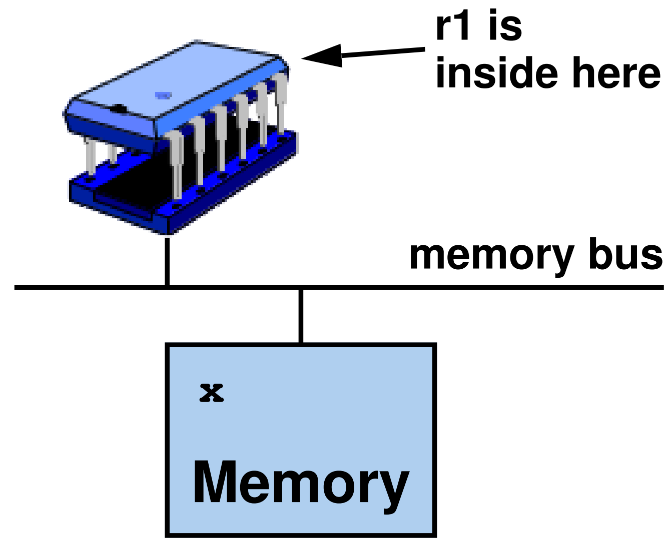
Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
 /*
  ld    r1,x
  add   r1,1
  st    r1,x
  */
```

Thread 2:

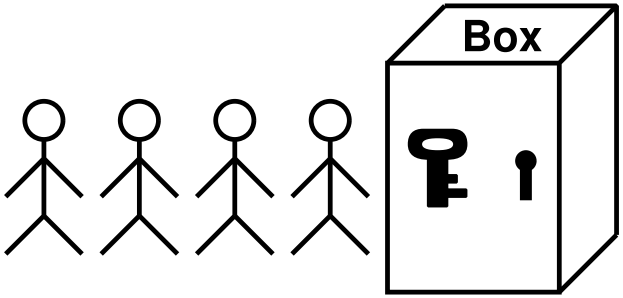
```
x = x+1;
 /*
  ld    r1,x
  add   r1,1
  st    r1,x
  */
```



- *Atomic* operation
  - every time you talk about an *atomic operation*, you need to be very clear about exactly what it is *with respect to*
  - we will use a visual aid

Threads and Synchronization

- Solution: put `x` in an (abstract) safe-deposit box under lock and key
  - whoever has the key gets to use `x`
    - what if you fall asleep while inside the box?
      - no problem, others will just have to *wait*
    - isn't that inefficient?
      - correctness is more important
      - if you know you will fall asleep inside the box, you should be nice to others (and be more efficient) by getting out of the box and get in line later



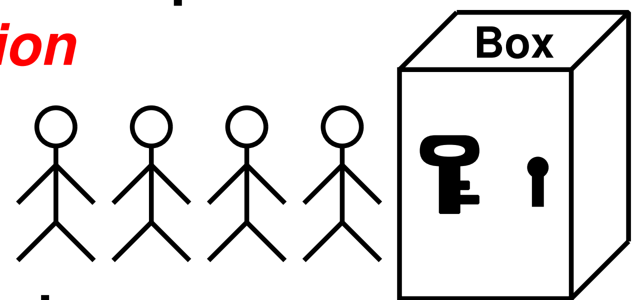
- Rule for accessing `x` from now on:
  - you can only access `x` using an atomic operation
- What if you have more than one variable (e.g., `x`, `y`, `z`)?
  - put all of them inside one safe-deposit box
  - you can only access `x`, `y`, `z` *atomically*, *with respect to* the *operation of the "box"*

Threads and Synchronization

```
// shared by both threads
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int x;
...
pthread_mutex_lock(&m);
x = x+1;
pthread_mutex_unlock(&m);
```

critical section

Locking a mutex is like getting the key to a safe-deposit box



- code between `pthread_mutex_lock()` and `pthread_mutex_unlock()` for a particular *mutex* is called a *critical section with respect to that mutex*
  - *all* the critical sections *with respect to a particular mutex* are "*mutually exclusive*"
    - the system (not necessarily the OS) guarantees that only *one* critical section can be executing at any point in time *with respect to a particular mutex*
  - how it's really done will be covered in Ch 5

Set Up

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

- Mutex initialization
  - mutex is unlocked
  - initialize data structure (initially empty) used to keep track of *waiting threads*

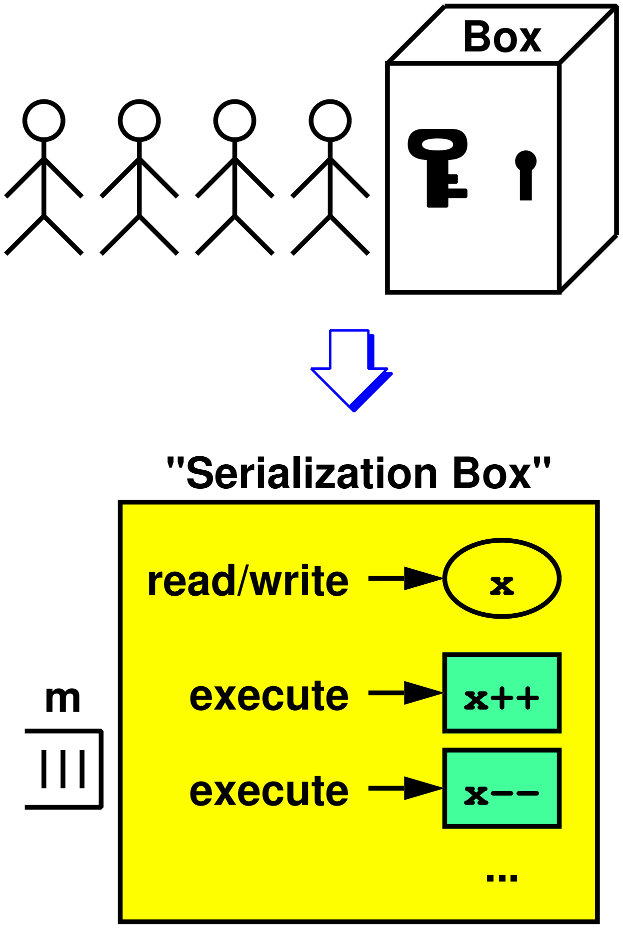
- If a mutex cannot be initialized statically, do:

```
int pthread_mutex_init(
        pthread_mutex_t *mutexp,
        pthread_mutexattr_t *attrp)

int pthread_mutex_destroy(
        pthread_mutex_t *mutexp)
```

- Usually, mutex attributes are not used

One Mutex, Multiple Critical Sections

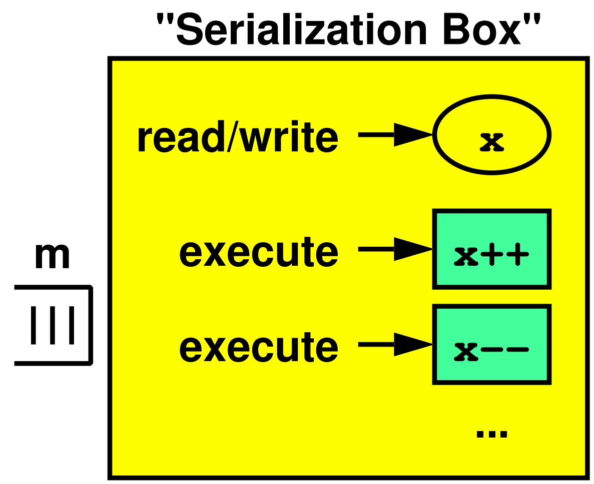
```
f1() {
  pthread_mutex_lock(&m);
  x++;        } critical section
  pthread_mutex_unlock(&m);
}

f2() {
  pthread_mutex_lock(&m);
  x--;        } critical section
  pthread_mutex_unlock(&m);
}
```



- only one thread can be running inside the "Serialization Box" at a time (access to the "box" is "serialized" or "synchronized")
  - you should *only* access shared variables using *critical section code*
- the "Serialization Box" is not a real box, it's conceptual

One Mutex, Multiple Critical Sections

```
f1() {
  pthread_mutex_lock(&m);
  x++;            } critical section
  pthread_mutex_unlock(&m);
}

f2() {
  pthread_mutex_lock(&m);
  x--;            } critical section
  pthread_mutex_unlock(&m);
}
```



- By calling `pthread_mutex_lock(&m)`, a thread can be placed into a *queue* and *wait* there *indefinitely* for mutex `m` to become available
  - multiple threads would join this queue
  - queue is served one at a time, like a supermarket checkout
  - when it's your thread's turn, `pthread_mutex_lock()` returns with the mutex locked, your thread can execute critical section code, and then release the mutex

Taking Multiple Locks

- Mutex is not a cure-all
  - when you have more than one locks, you may get into trouble

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

Taking Multiple Locks

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

- Graph representation ("wait-for" graph) for the entire process
  - draw an arrow from a mutex you are holding to another mutex you are waiting for