# CS31 Worksheet: Week 12: Synchronization

If you call `thread_create()` on a modern OS (Linux/Mac/Windows), which type of thread would you expect to receive? (Why? Which would you pick?)

A. Kernel threads

B. User threads

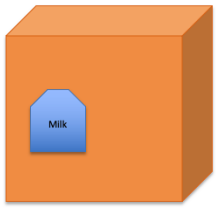
C. Some other sort of threads

If one CPU core can run a program at a rate of X, how quickly will the program run on two cores? Why?

A. Slower than one core (<X)
B. The same speed (X)
C. Faster than one core, but not double (X-2X)
D. Twice as fast (2X)
E. More than twice as fast(>2X)

## How many cartons of milk can we have in this scenario? (Can we ensure this somehow?)

| Time | You | Your Roommate |
|---|---|---|
| 3.00 | Arrive home | |
| 3.05 | Look in fridge, no milk | |
| 3.10 | Leave for the grocery store | |
| 3.15 | | |
| 3.20 | Arrive at the grocery store | |
| 3.25 | Buy Milk | |
| 3.30 | | |
| 3.35 | Arrive home, put milk in fridge | Arrive Home |
| 3.40 | | Look in fridge, find milk |
| 3.45 | | Cold Coffee (nom) |




A. One carton (you)
B. Two cartons
C. No cartons
D. Something else

## Credit/Debit Problem: Race Condition

**Thread $T_0$**

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

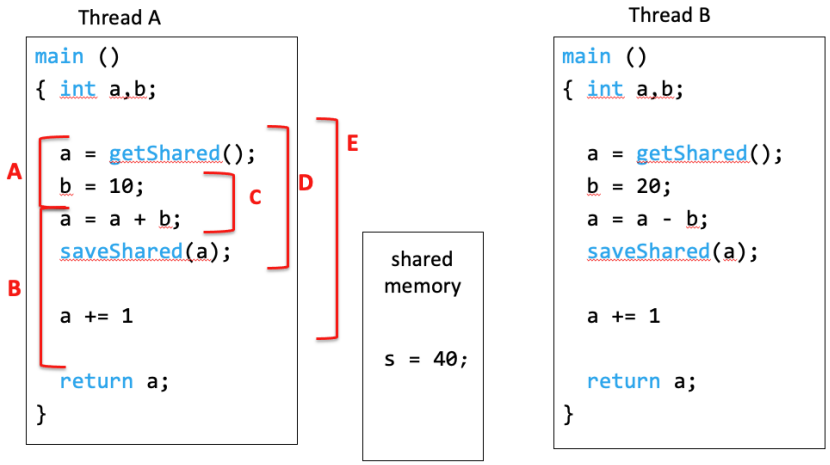
**Thread $T_1$**

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

# Which code region is a critical section?




# Is there a race condition?

Suppose `count` is a global variable, multiple threads increment it:

`count++;`

A. Yes, there's a race condition (`count++` is a critical section).
B. No, there's no race condition (`count++` is not a critical section).
C. Cannot be determined

How about if compiler implements it as:

```
movq (%rdx), %rax    // read count value
addq $1, %rax        // modify value
movq %rax, (%rdx)    // write count
```

How about if compiler implements it as:

```
incq (%rdx)          // increment value
```

# Semaphore Operations

```
sem s = n;                                          // declare and initialize

wait (sem s)                                        // Executes atomically(*)
    decrement s;
    if s < 0:
        block thread (and associate with s);

signal (sem s)                                      // Executes atomically(*)
    increment s;
    if blocked threads:
        unblock (any) one of them;
```

Based on what you know about semaphores, should a process be able to check beforehand whether wait(s) will cause it to block?

A. Yes, it should be able to check.
B. No, it should not be able to check.