

CS 3650 Computer Systems – Spring 2023

# Concurrency (2)

Week 9

# Recap on threads

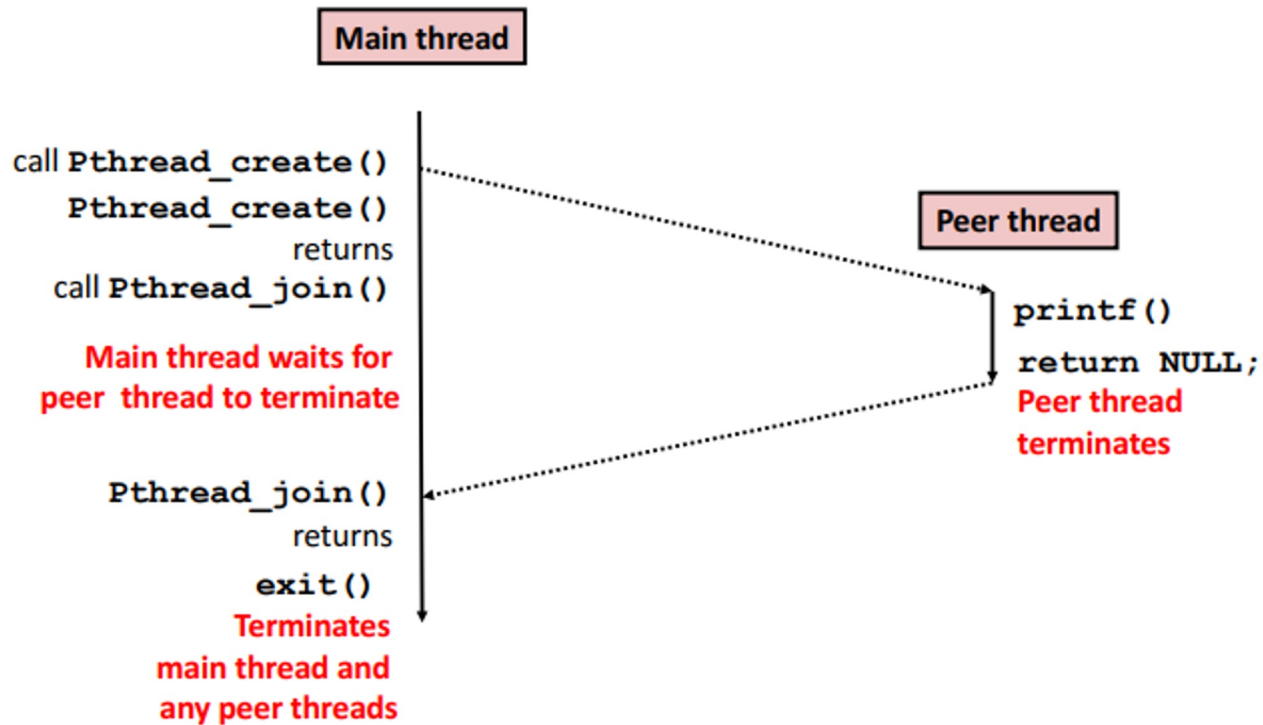
- Light-weight processes that share the same memory and state
- Every process has at least one thread
- Benefits:
  - Resource sharing, no need for IPC
  - Economy: faster to create, faster to context switch
  - Scalability: simple to take advantage of multi-core CPUs

# Hello Thread

- The thread that is “launched” is a function in the program
  - This is done when the thread is created
  - Different attributes can be sent to threads (in this case the first NULL)
  - Arguments can also be passed to the function (second NULL)
- `pthread_join` is the equivalent to “wait” for threads
- What if we don't call join?

```
1 // Compile with:
2 //
3 // clang -lpthread thread1.c -o thread1
4 //
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <pthread.h>
8
9 // Thread with variable arguments
10 void *thread(void *vargp){
11     printf("Hello from thread\n");
12     return NULL;
13 }
14
15 int main(){
16     // Store our Pthread ID
17     pthread_t tid;
18     // Create and execute the thread
19     pthread_create(&tid, NULL, thread, NULL);
20     // Wait in 'main' thread until thread executes
21     pthread_join(tid, NULL);
22     // end program
23     return 0;
24 }
```

# Visual execution of “Hello Thread”



# Launching multiple threads

- This time launch 10000 threads
- counter is shared between threads
- What is wrong with this program?
- What is the final output?

```
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9999
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
```

```
1 // Compile with:
2 //
3 // clang -lpthread thread3.c -o thread3
4 //
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <pthread.h>
8
9 #define NTHREADS 10000
10
11 int counter = 0;
12
13 // Thread with variable arguments
14 void *thread(void *vargp){
15     counter = counter +1;
16     return NULL;
17 }
18
19 int main(){
20     // Store our Pthread ID
21     pthread_t tids[NTHREADS];
22     printf("Counter starts at: %d\n",counter);
23     // Create and execute multiple threads
24     for(int i=0; i < NTHREADS; ++i){
25         pthread_create(&tids[i], NULL, thread, NULL);
26     }
27     // Create and execute multiple threads
28     for(int i=0; i < NTHREADS; ++i){
29         pthread_join(tids[i], NULL);
30     }
31
32     printf("Final Counter value: %d\n",counter);
33     // end program
34     return 0;
35 }
```

# Example with lock

- Included a `pthread_mutex_lock`
- lock and unlock protects
- Locks in other words enforce, that we have exclusive access to a region of code.

```
1 // Compile with:
2 // clang -lpthread thread4.c -o thread4
3 // This program fixes a problem with thread3.c
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <pthread.h>
7
8 #define NTHREADS 10000
9
10 int counter = 0;
11 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
12
13 // Thread with variable arguments
14 void *thread(void *vargp){
15     pthread_mutex_lock(&mutex1);
16     counter = counter + 1;
17     pthread_mutex_unlock(&mutex1);
18     return NULL;
19 }
20
21 int main(){
22     // Store our Pthread ID
23     pthread_t tids[NTHREADS];
24     printf("Counter starts at: %d\n",counter);
25     // Create and execute multiple threads
26     for(int i=0; i < NTHREADS; ++i){
27         pthread_create(&tids[i], NULL, thread, NULL);
28     }
29
30     // Create and execute multiple threads
31     for(int i=0; i < NTHREADS; ++i){
32         pthread_join(tids[i], NULL);
33     }
34     printf("Final Counter value: %d\n",counter);
35     // end program
36     return 0;
37 }
```

# What was happening?

Thread 1 (counter = counter + 1)

`pthread_mutex_lock`

Read "counter": 10

Add 1 to "counter": 11

Write to "counter": 11

`pthread_mutex_unlock`

Thread 2 (counter = counter + 1)

`pthread_mutex_lock`

// Lock is held by thread 1 so

// thread 2 has to wait until

// thread 1 unlocks

// Now acquires the lock and runs

Read "counter": 10

Add 1 to "counter": 11

Write to "counter": 11

`pthread_mutex_unlock`

BLOCKS  
EXECUTION

# Posix Threads API (PThreads Interface)

- Sample functions
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
      - `exit()` - Terminates all threads
      - `return` - terminates current thread
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
      - `pthread_mutex_lock` and `pthread_mutex_unlock`



# Bank Transactions

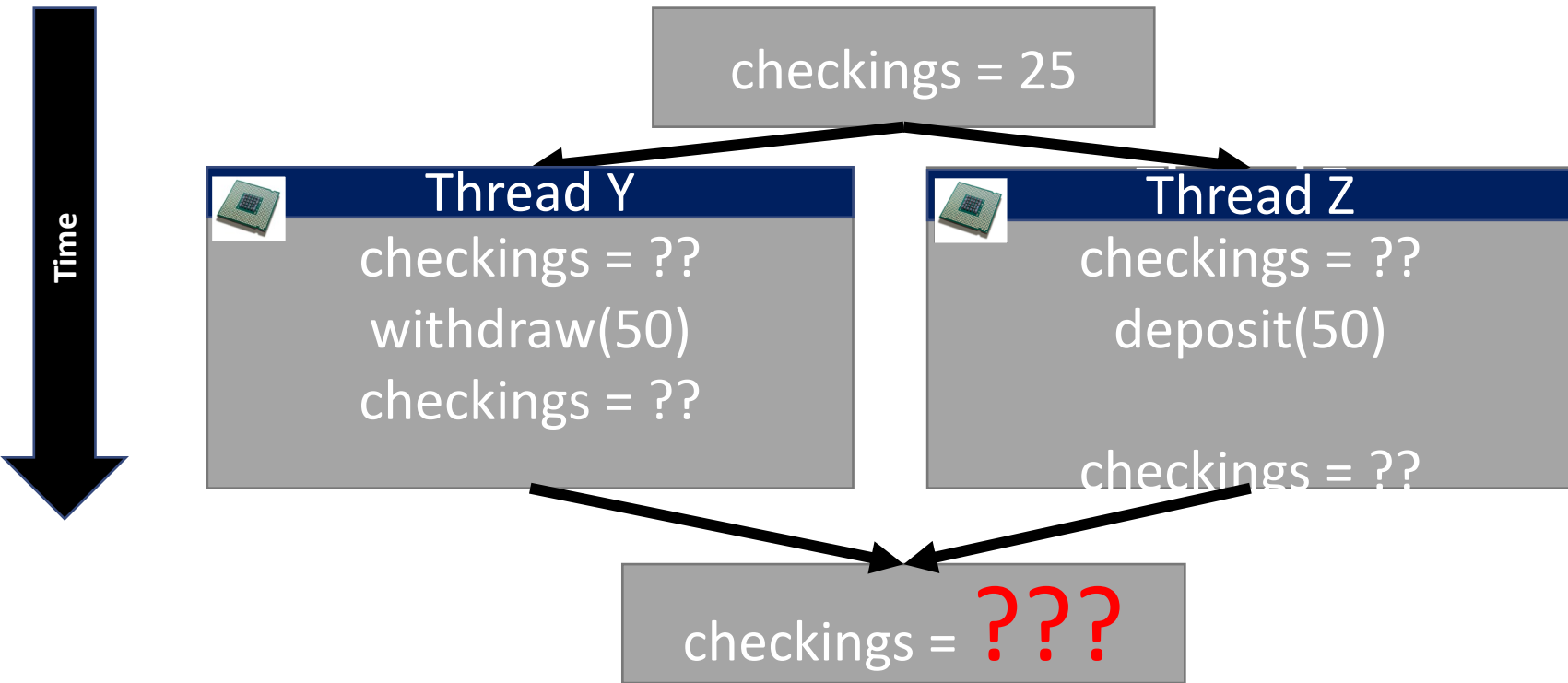
# A series (i.e. serial) of Bank Transactions

1. If I start with **\$25** in my checking account.
2. Then I deposit \$50, I have \$75.
3. If I then withdraw \$50, I now have \$25.
4. My final balance is **\$25**.
5. There is a variable *checkings* that monitors our balance.

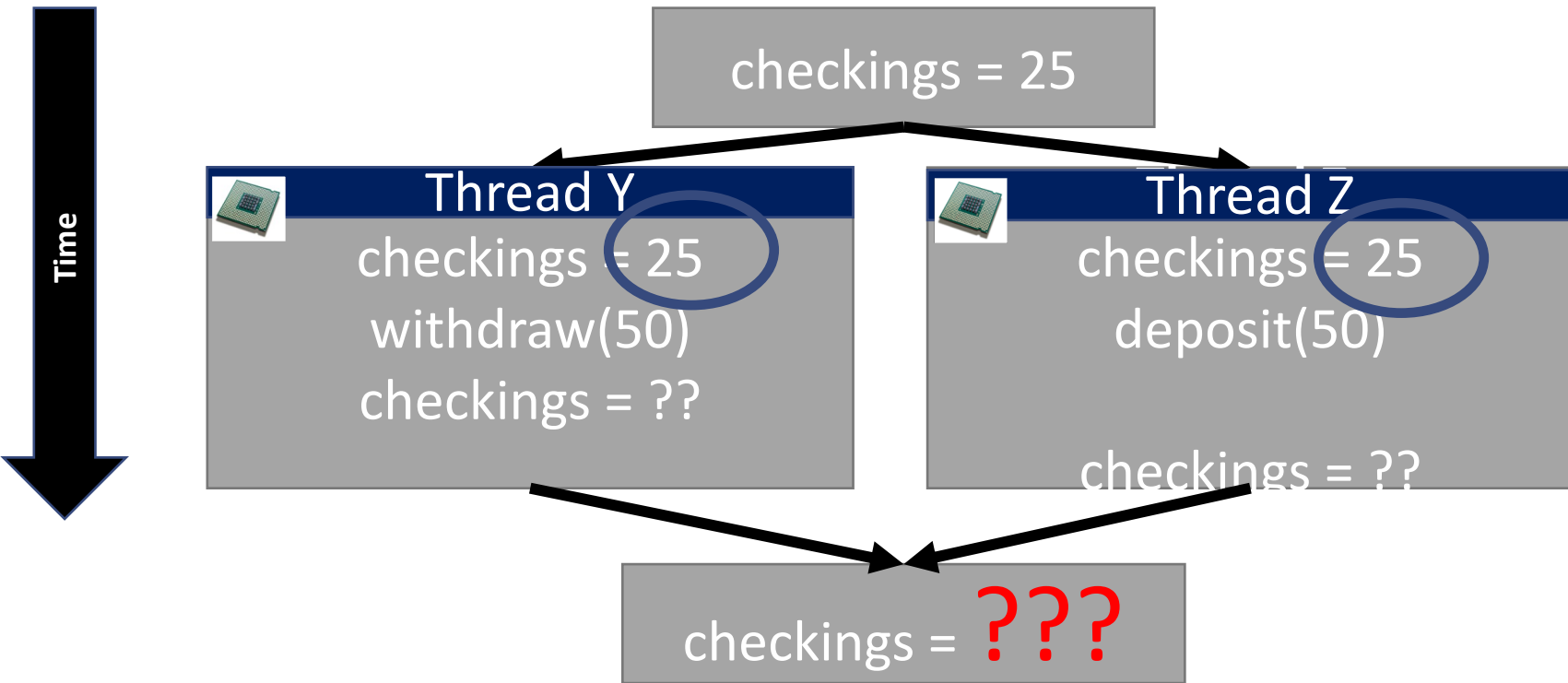
# Concurrent Bank Transaction

1. If I start with **\$25** in my checking account.
2. Then I deposit \$50 and withdraw \$50 at the same time (concurrently)
3. My final balance should still be **\$25**.
4. There is a **shared variable** checkings in each thread that monitors our balance.

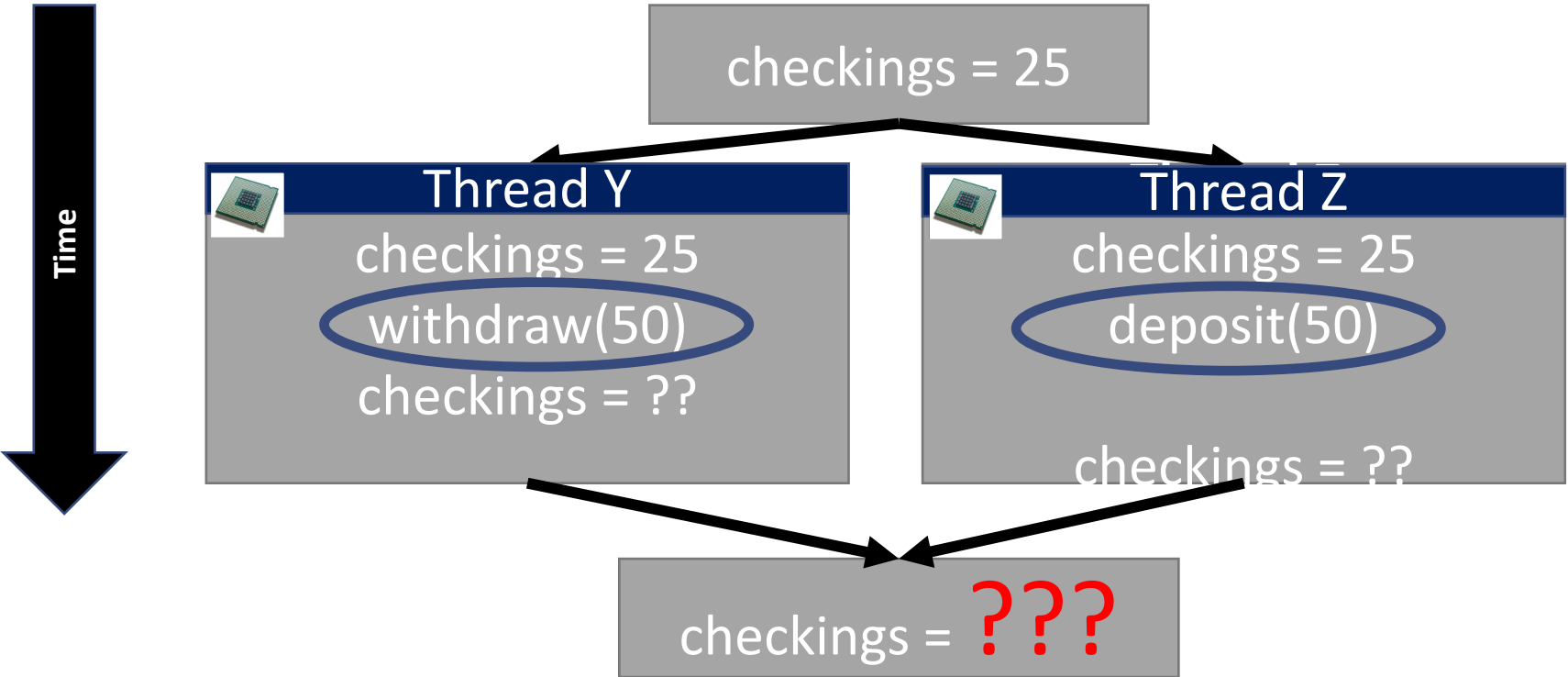
# Read our initial balance



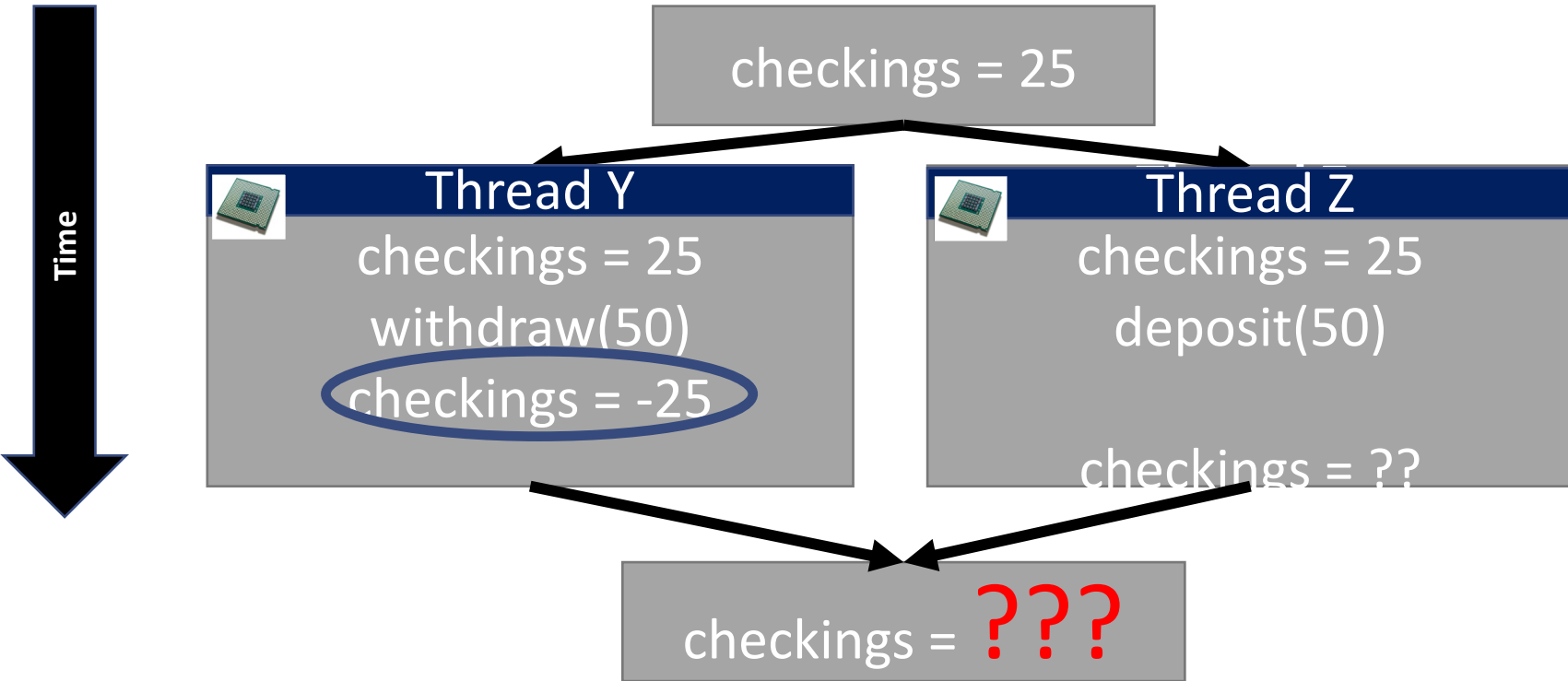
Okay, we have \$25 – now move on



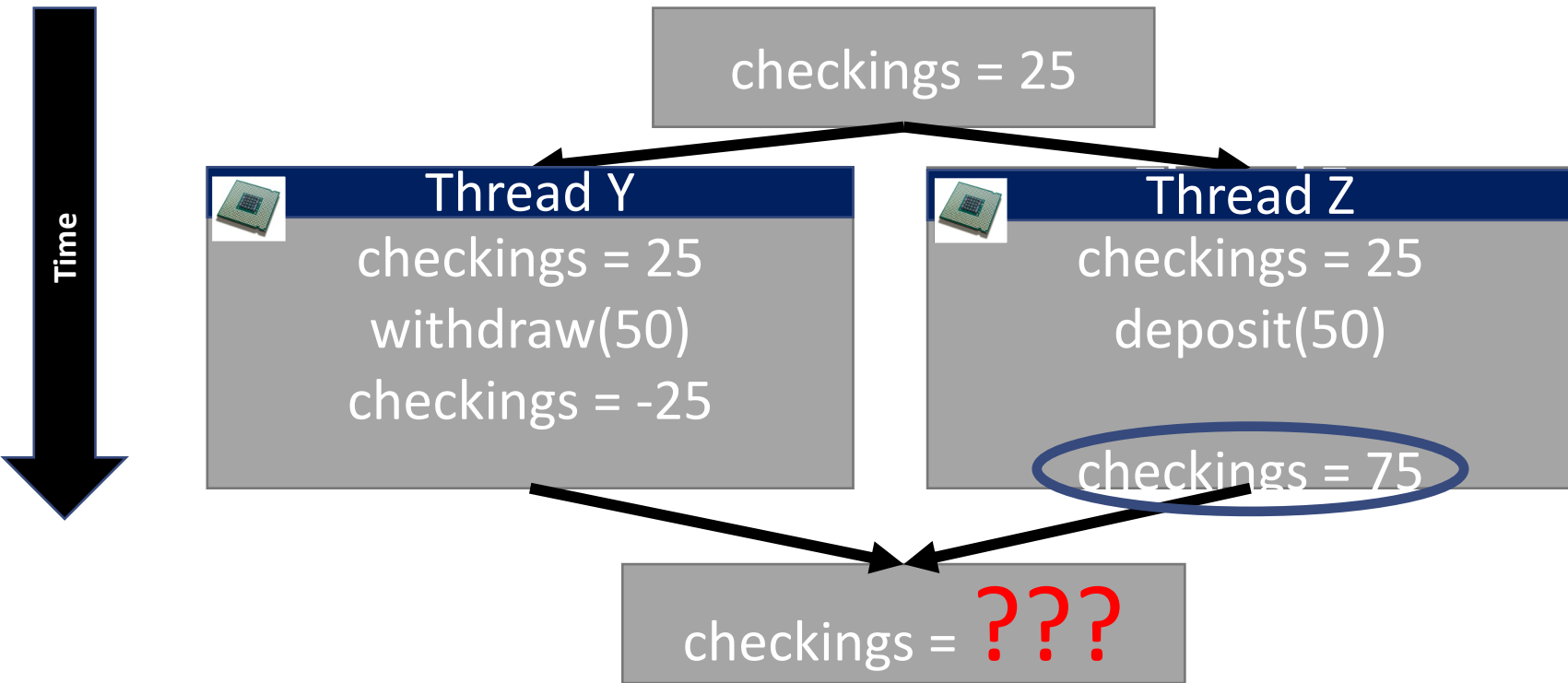
withdraw and deposit occur (Thread Y and Z)



# Checkings from Thread Y updates first

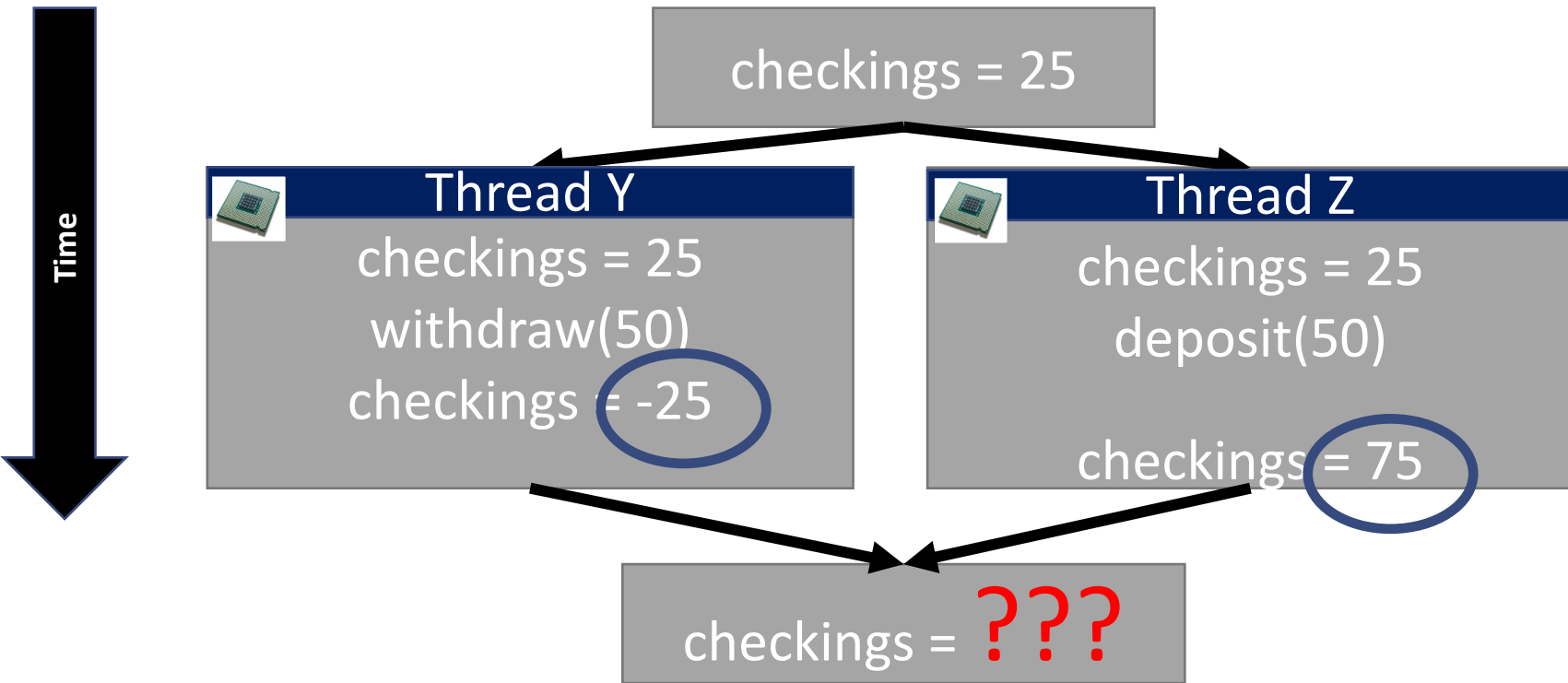


(Thread Z) updates its checkings value shortly after

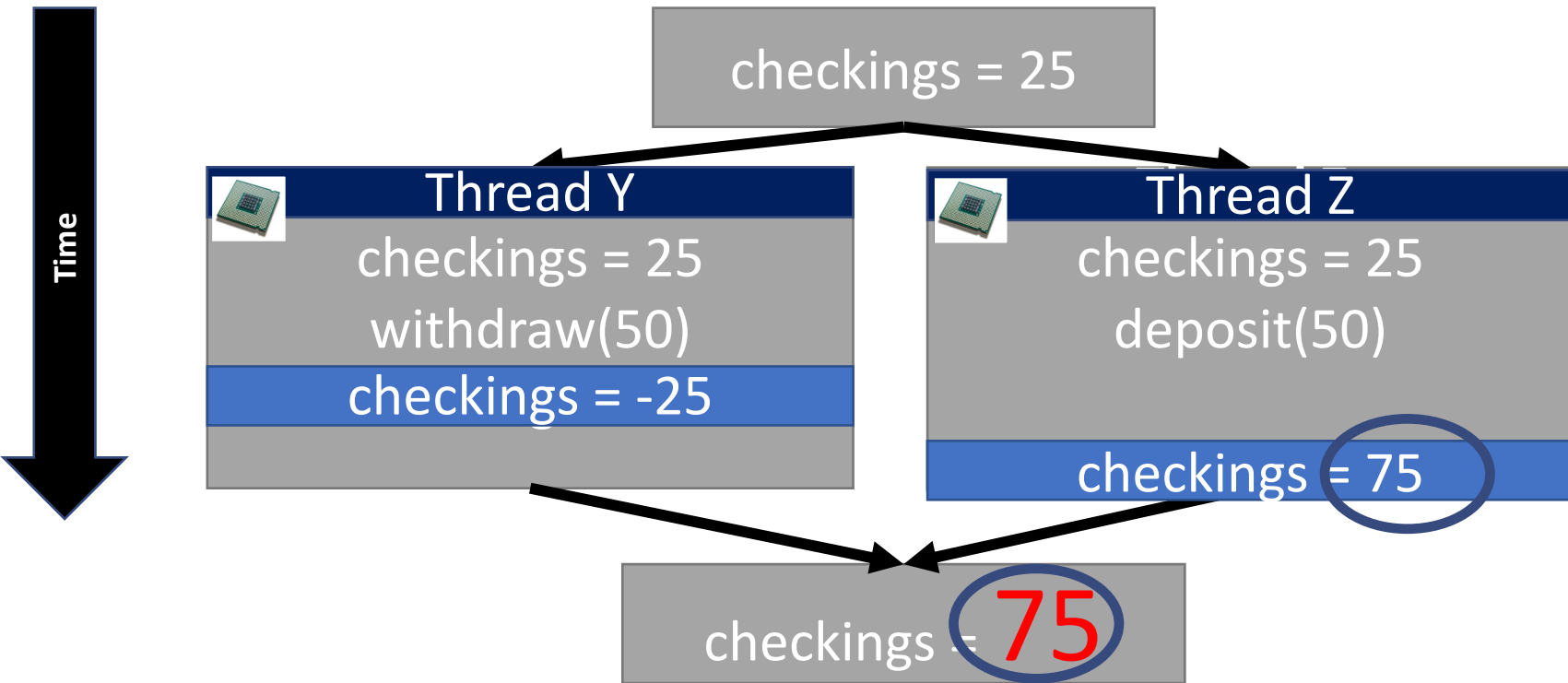




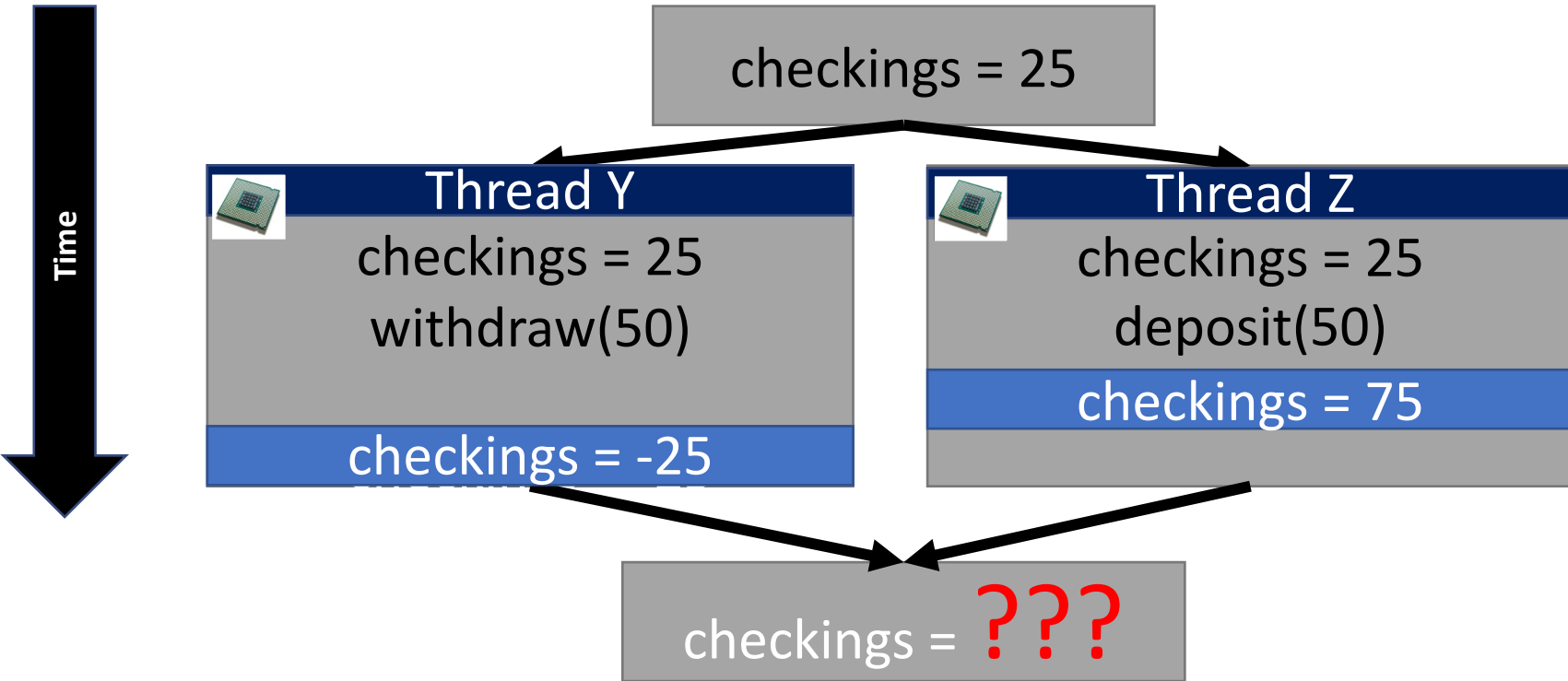
# Now we have conflicting information



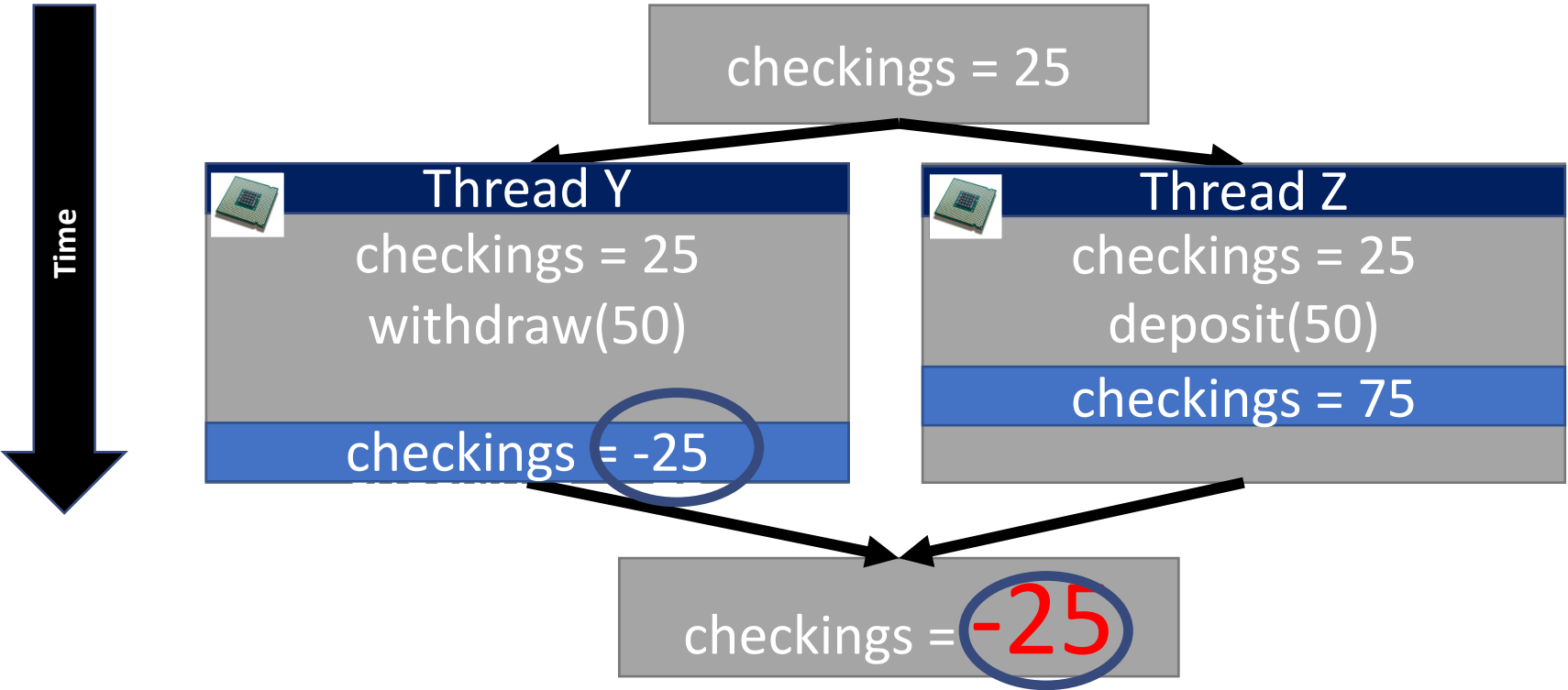
checkings stores the last value of 75 (Thread Z)



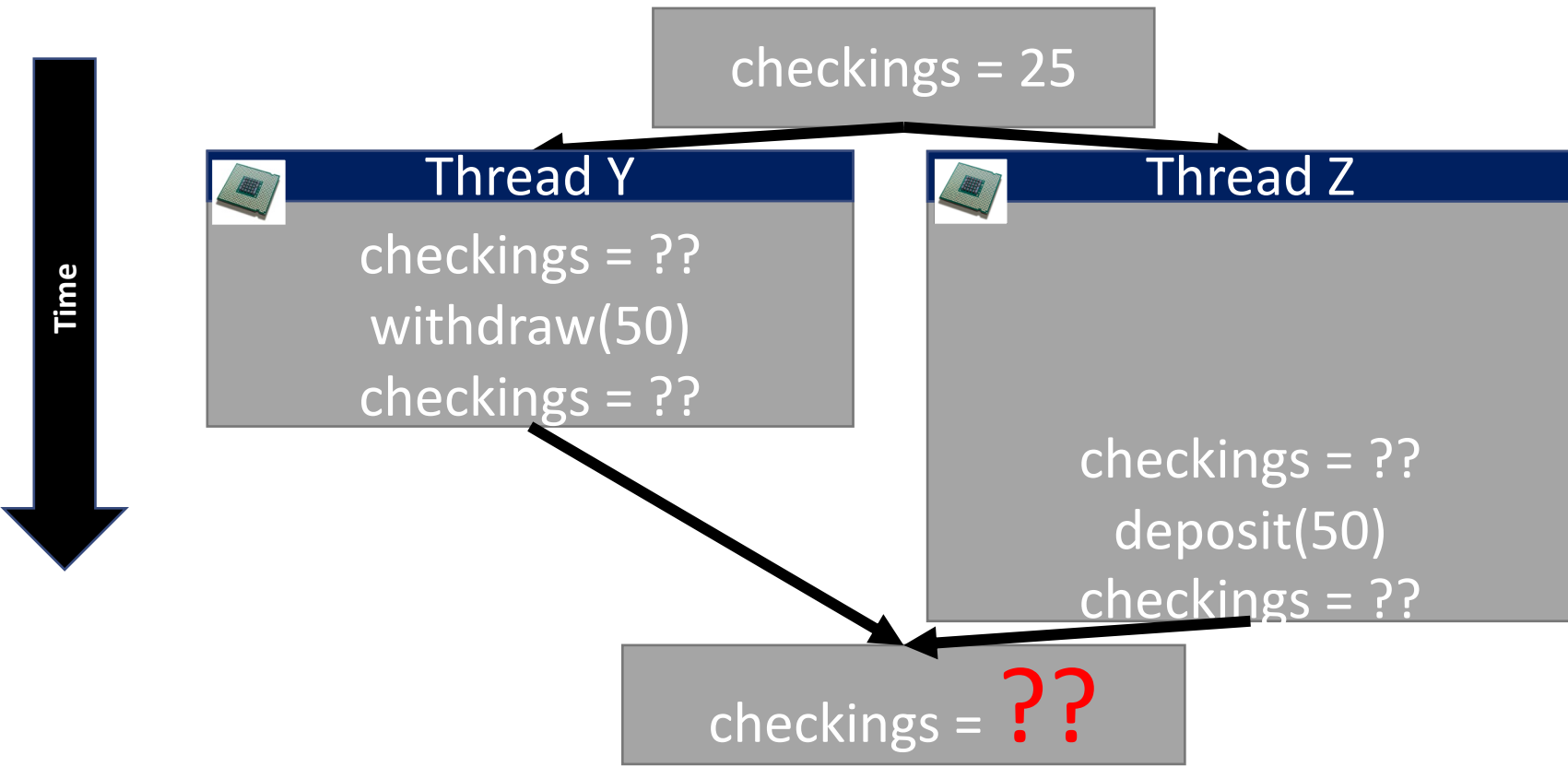
# What if these operations had swapped!



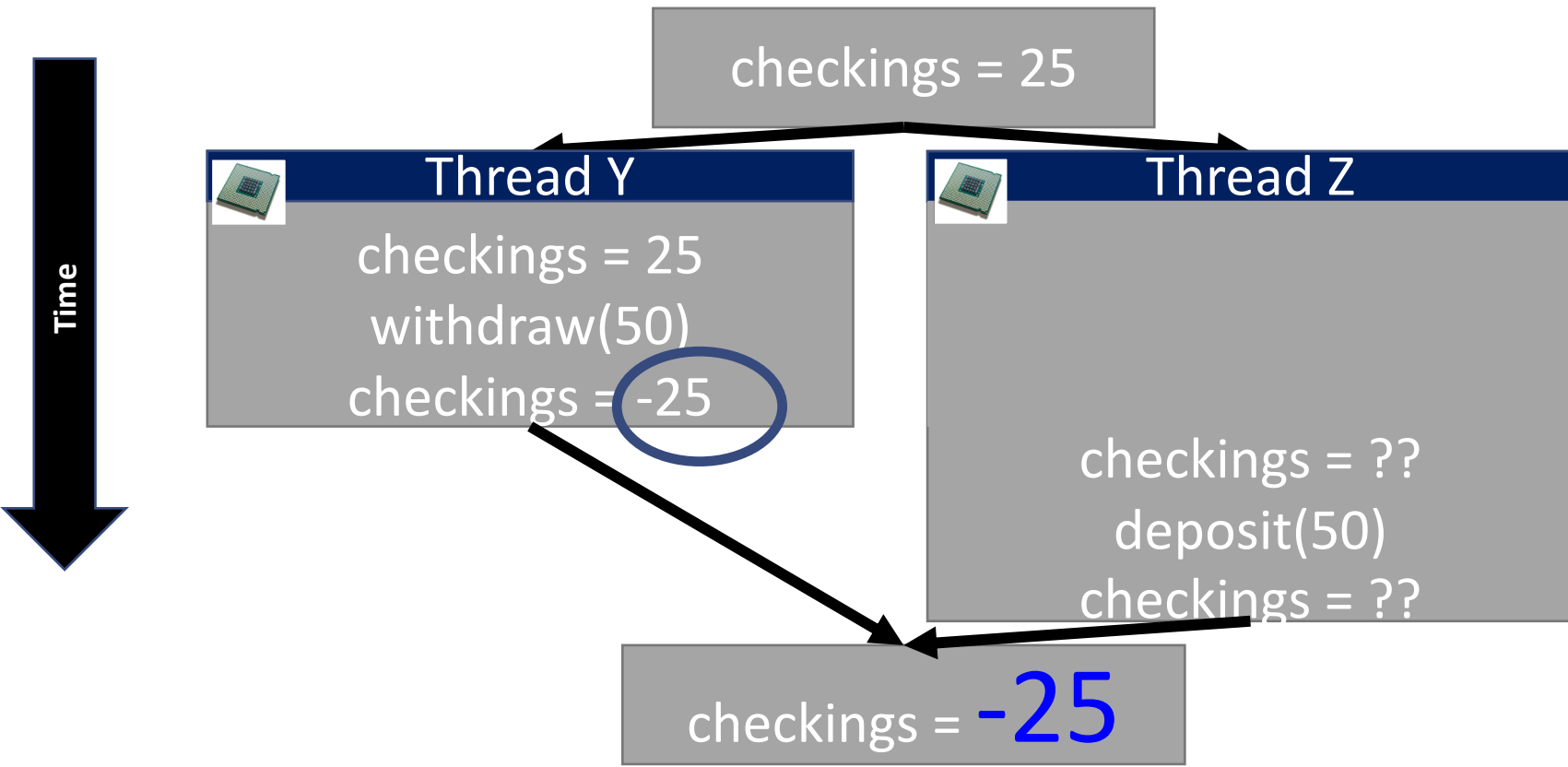
# This time our balance is -25! (Thread Y)



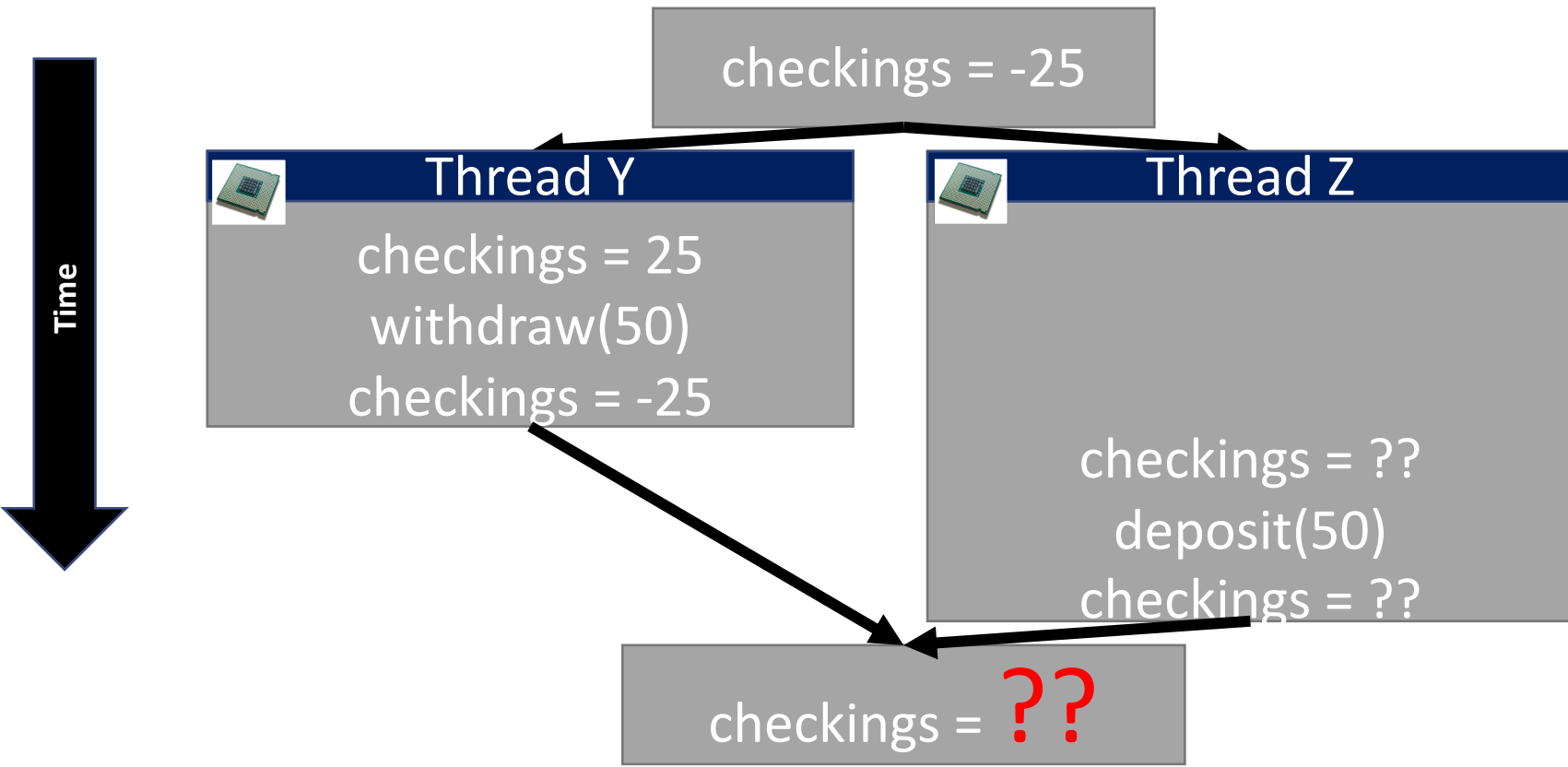
# How about if Thread Z lags behind Thread Y?



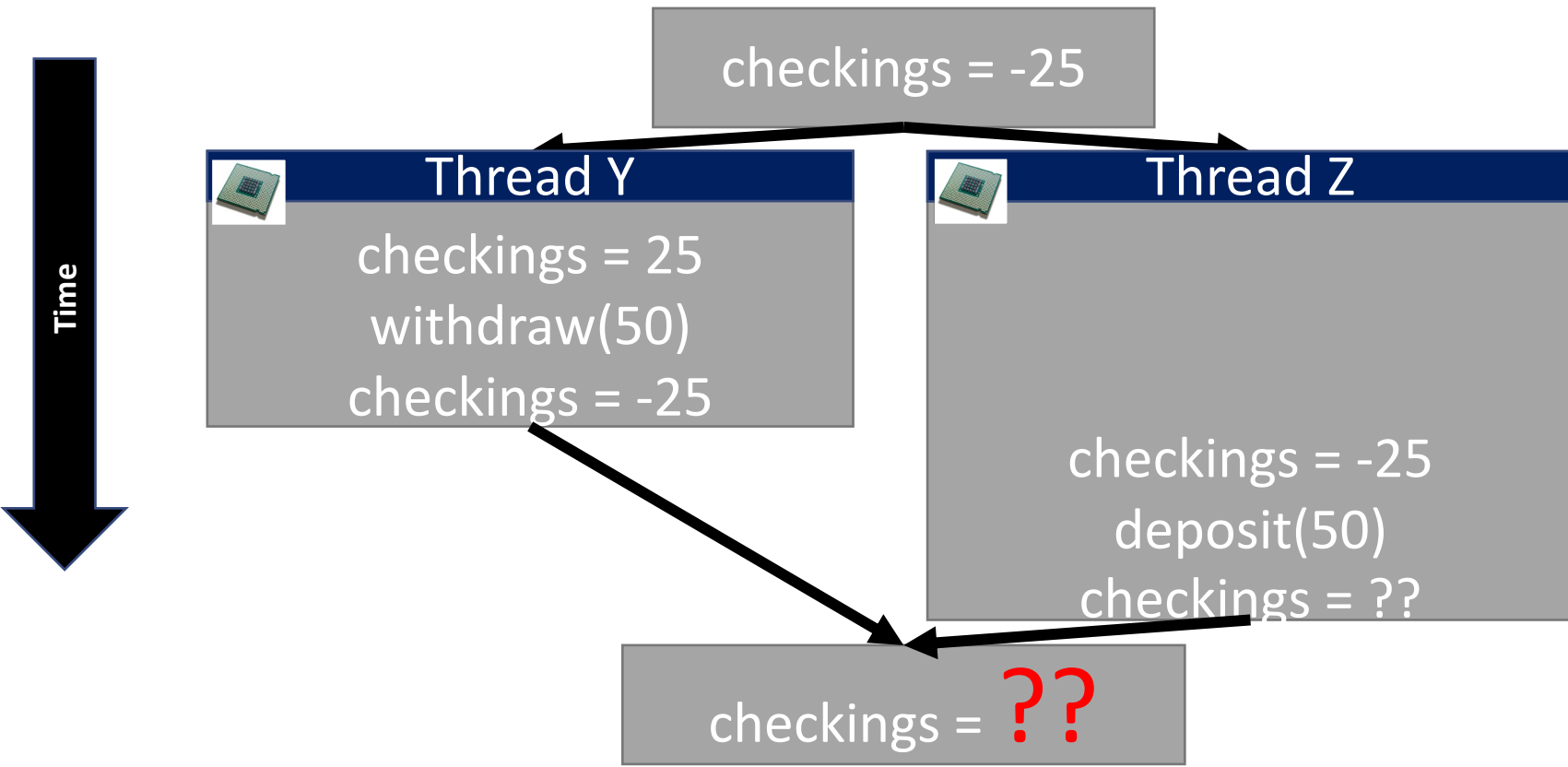
# How about if Thread Z lags behind Thread Y?



# How about if Thread Z lags behind Thread Y?

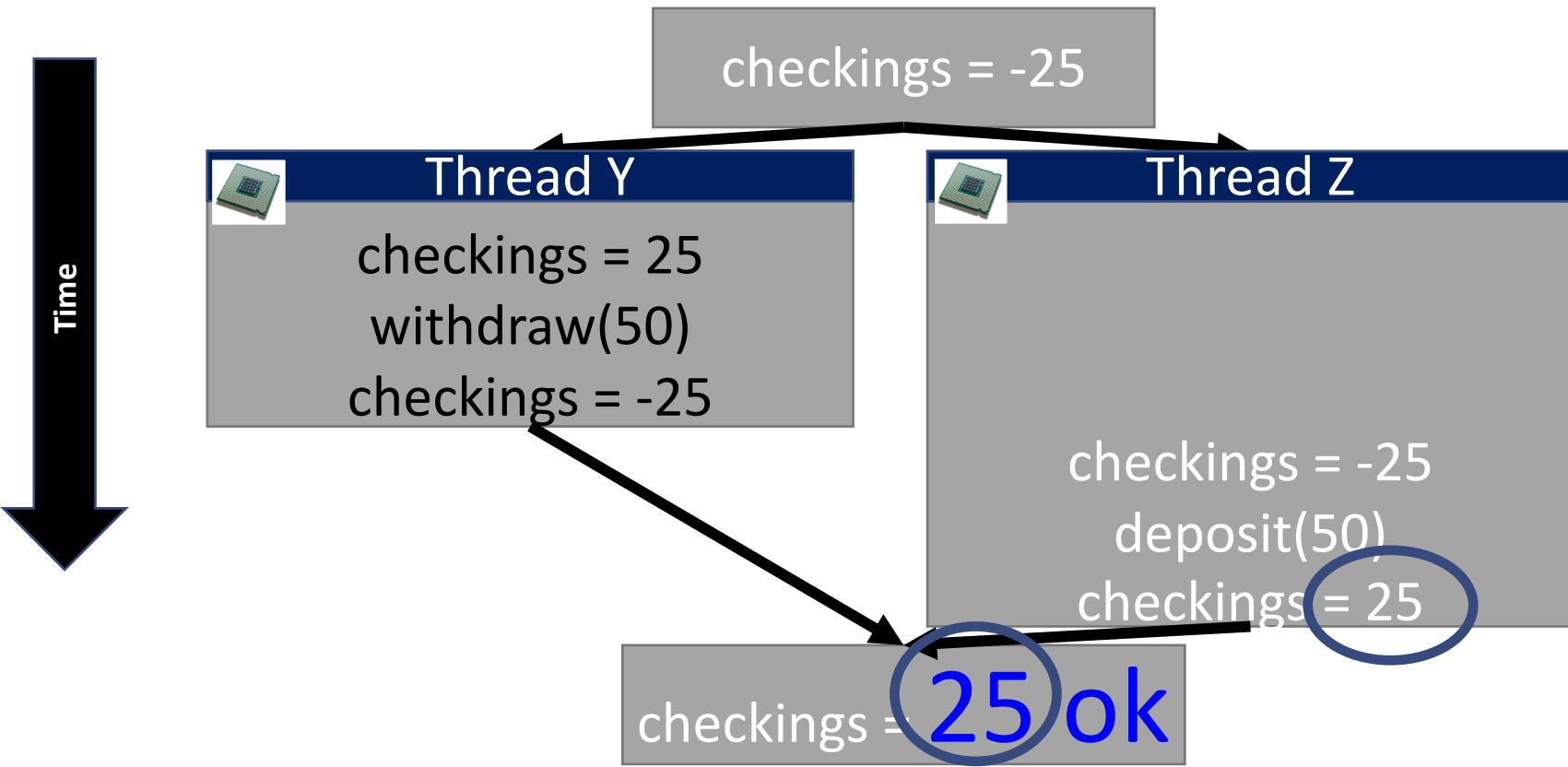


# How about if Thread Z lags behind Thread Y?





Okay—this time we happen to get 25



We have witnessed a data race

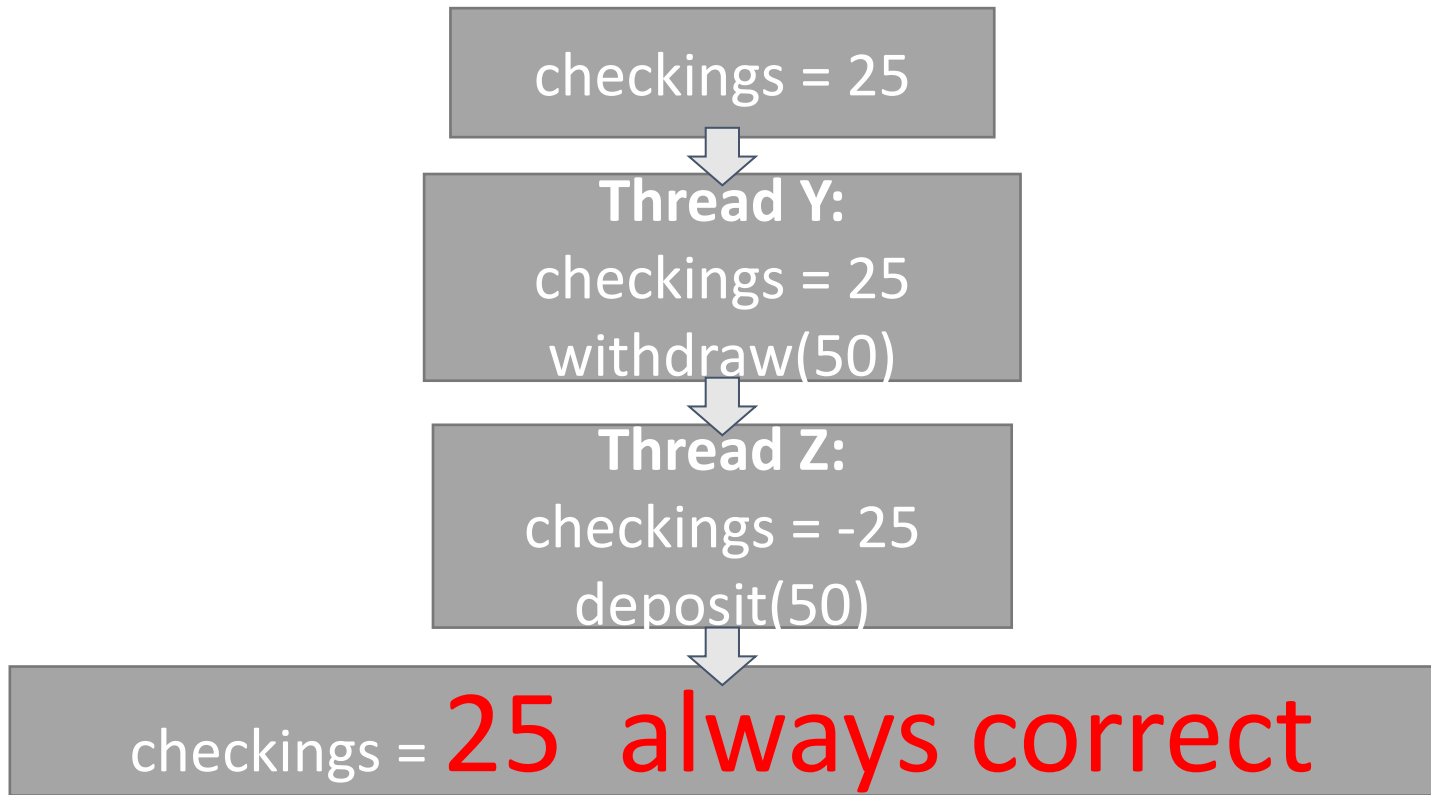
A common concurrency problem

checkings = 75

checkings = -25

checkings = 25 ok

We need to **synchronize** – enforce ordering



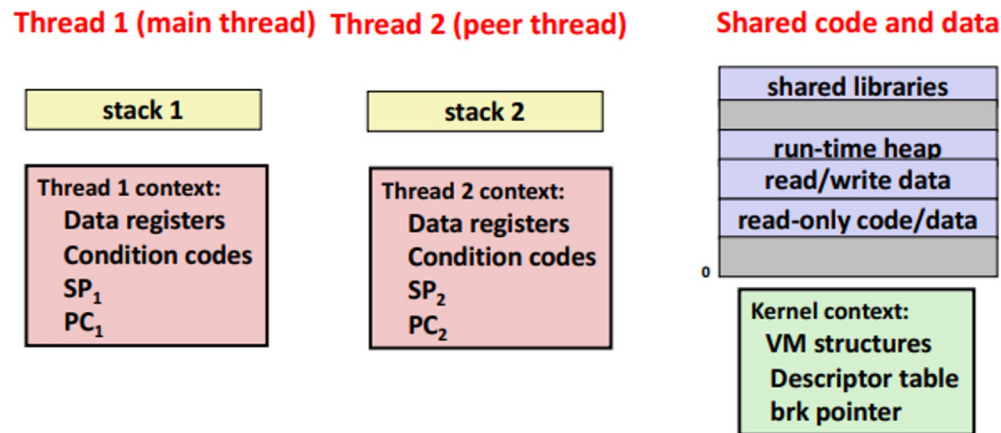
# (The Bug!)

- What is wrong with this program?
  - The problem is we have a global “counter” that is shared
  - There is an interleaving of instructions here.
  - Any possible interleaving can occur!
- Solution is to add locks!

```
1 // Compile with:
2 // clang -lpthread thread4.c -o thread4
3 // This program fixes a problem with thread3.c
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <pthread.h>
7
8 #define NTHREADS 10000
9
10 int counter = 0;
11 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
12
13 // Thread with variable arguments
14 void *thread(void *vargp){
15     pthread_mutex_lock(&mutex1);
16     counter = counter + 1;
17     pthread_mutex_unlock(&mutex1);
18     return NULL;
19 }
20
21 int main(){
22     // Store our Pthread ID
23     pthread_t tids[NTHREADS];
24     printf("Counter starts at: %d\n",counter);
25     // Create and execute multiple threads
26     for(int i=0; i < NTHREADS; ++i){
27         pthread_create(&tids[i], NULL, thread, NULL);
28     }
29
30     // Create and execute multiple threads
31     for(int i=0; i < NTHREADS; ++i){
32         pthread_join(tids[i], NULL);
33     }
34     printf("Final Counter value: %d\n",counter);
35     // end program
36     return 0;
37 }
```

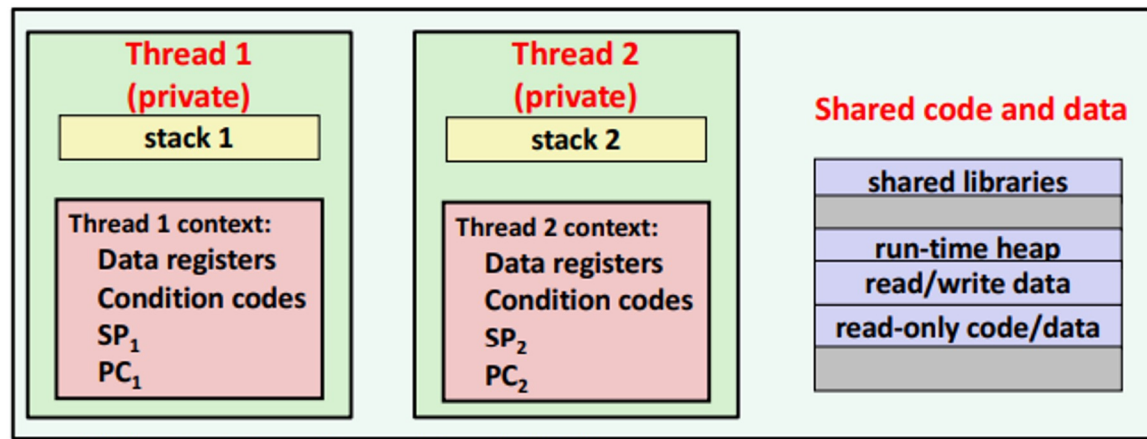
# What Data is Shared in Threaded C Programs?

- Global variables are shared
  - We just saw an example with counter.
  - (Note: the compilers can be smart)
    - (“counter” is only shared if it is referenced within the thread, otherwise do not copy it.)



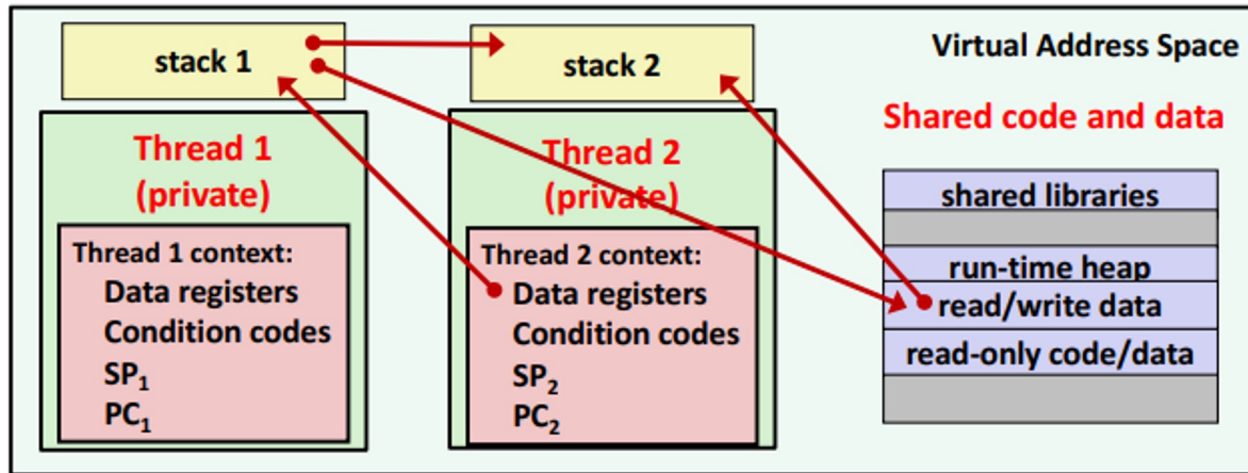
# Threads Memory Model: Conceptual

- Multiple threads run within the context of a single process
- Each thread has its own separate thread context
  - Thread ID, stack, stack pointer, PC, condition codes, and General Purpose Registers
- All threads share the remaining process context
  - Code, data, heap, and shared library segments for virtual address space
  - Open files



# Threads Memory Model: Actual

- Separation of data is not strictly enforced:
  - Register values are truly separate and protected
  - Any thread however, can read and write the stack of any other thread



# Mapping Variable Instances to Memory

- Global Variables
  - Definition: Variable declared outside of a function
  - Virtual Memory contains exactly one instance of any global variable
- Local Variables
  - Definition: Variable declared inside function without static attribute
  - Each thread stack contains one instance of each local variable
- **Local static variables**
  - Definition: Variables declared inside function with the static attribute
  - Virtual memory contains exactly one instance of any local static variable.



# Mapping Variable Instances to Memory

- 1 main thread “m” and two threads “p0” and “p1”

**Global var:** 1 instance (ptr [data])

```
char **ptr; /* global var */
```

**Local vars:** 1 instance (i.m, msgs.m)

```
int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

**Local var:** 2 instances (myid.p0 [peer thread 0's stack], myid.p1 [peer thread 1's stack])

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}
```

**Local static var:** 1 instance (cnt [data])

sharing.c

# Shared Variable Analysis

- 1 main thread “m” and two threads “p0” and “p1”

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr			
cnt			
i.m			
msgs.m			
myid.p0			
myid.p1			

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}

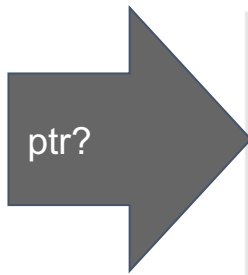
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

```

# Shared Variable Analysis



<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr			
cnt			
i.m			
msgs.m			
myid.p0			
myid.p1			

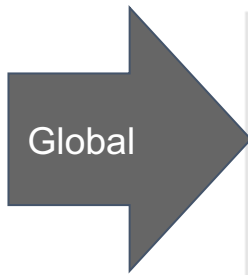
```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

# Shared Variable Analysis



<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt			
i.m			
msgs.m			
myid.p0			
myid.p1			

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}

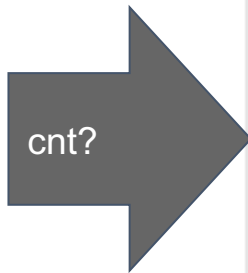
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

```

# Shared Variable Analysis



<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt			
i.m			
msgs.m			
myid.p0			
myid.p1			

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

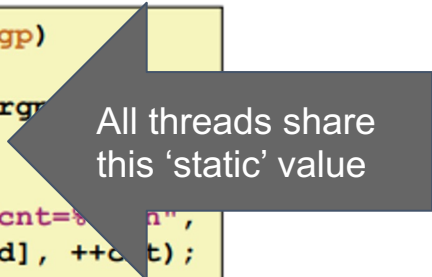
# Shared Variable Analysis

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
<code>ptr</code>	yes	yes	yes
<code>cnt</code>	no	yes	yes
<code>i.m</code>			
<code>msgs.m</code>			
<code>myid.p0</code>			
<code>myid.p1</code>			

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

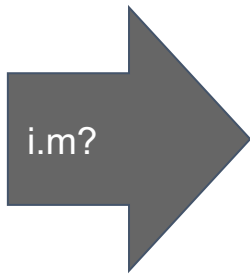
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```



All threads share this 'static' value

# Shared Variable Analysis



<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt	no	yes	yes
i.m			
msgs.m			
myid.p0			
myid.p1			

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}

```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

```

# Shared Variable Analysis

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
<code>ptr</code>	yes	yes	yes
<code>cnt</code>	no	yes	yes
<code>i.m</code>	yes	no	no
<code>msgs.m</code>			
<code>myid.p0</code>			
<code>myid.p1</code>			

Shared?

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```



# Shared Variable Analysis

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	<b>YES</b>	<b>YES</b>
msgs.m			
myid.p0			
myid.p1			

This example is abusing "i".  
If a pointer to an object is passed to pthread\_create then it should be shared.

Shared?  
Yes.

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

# Shared Variable Analysis

msgs?  
(careful)

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m			
myid.p0			
myid.p1			

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

# Shared Variable Analysis

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0			
myid.p1			

We have a 'ptr' to msg, so effectively shared

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid] ++cnt);
    return NULL;
}
```

# Shared Variable Analysis

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0			
myid.p1			

myid.p0?

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}

```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

```

# Shared Variable Analysis

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1			

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

Local to peer  
thread 0 only

# Shared Variable Analysis

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1			

So for myid.p1?

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

# Shared Variable Analysis

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                    "Hello from bar" };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread, (void *)i);
    Pthread_exit(NULL);}
}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

Local to peer  
thread 1 only

# Synchronization of Threads

- Shared variables are thus handy for moving around data
- But if we do not share properly, we can have synchronization errors!
  - There is a solution however!
  - (recap below)



=

```
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9999
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
```



# We need a tool to protect shared resources

```
void deposit (float amount)
```

```
{
```



```
    checkings += amount;
```

```
}
```



# What to be careful with locks

# Correctness (can be) Easy

## Performance Hard

**Simply add locks!**

**lock**      withdraw(...) {...}

**lock**      deposit(...) {...}

**lock**      addInterest(...) {...}

**lock**      checkMinBalance(...) {...}

**lock**      chargeFee(...) {...}

**lock**      printBalance(...) {...}

# Correctness (can be) Easy

## Performance Hard

**Simply add locks!**

**lock**      `withdraw(...) {...}`  
**lock**      `deposit(...) {...}`  
**lock**      `addInterest(...) {...}`  
**lock**      `checkMinBalance(...) {...}`  
**lock**      `chargeFee(...) {...}`  
**lock**      `printBalance(...) {...}`

Good job—  
no data races  
here!



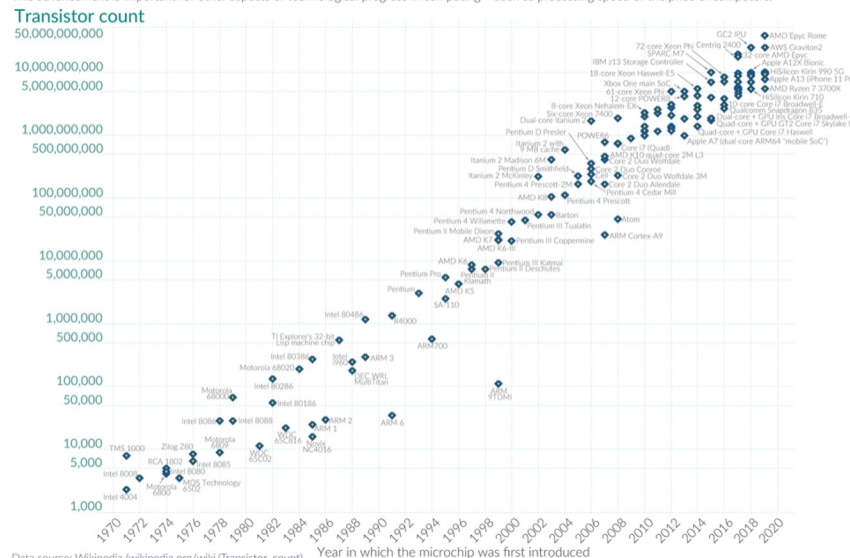
# Correctness (can be) Easy Performance Hard

Your program runs sequentially— did you forget about Amdahl's law?



## Moore's Law: The number of transistors on microchips doubles every two years.

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor\_count)  
OurWorldInData.org - Research and data to make progress against the world's largest problems. Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

By Max Roser, Hannah Ritchie - <https://ourworldindata.org/uploads/2020/11/Transistor-Count-over-time.png>, CC BY 4.0, <https://commons.wikimedia.org/w/index.php?curid=98219918>

# Where should we place locks?

- Suppose we have a shared counter which we increment by some precomputed value

```
int cumulative_time = 0; // global variable

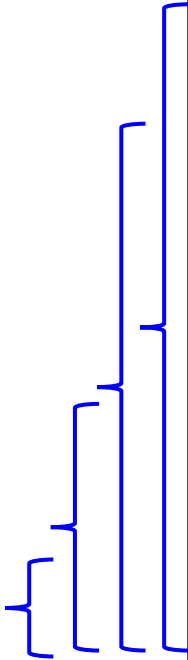
int main(void) {
    ...
    for (int i = 0; i < 100; i++)
        pthread_create(&tid[i], NULL,
                      thread, NULL);
    ...
    // joins all threads
    ...
    printf("cumulative time %d\n",
           cumulative_time);
    return 0;
}
```

```
void *thread(void *argv) {
    int start = get_current_time_in_int();

    int tmp = 0;
    for (int i = 0; i < 100000; i++) {
        tmp += 1;
    }

    int end = get_current_time_in_int();
    int elapsed = end - start;

    cumulative_time += elapsed;
}
```



# Critical Sections

- These examples highlight the **critical section problem**
- Classical definition of a critical section:  
*“A piece of code that accesses a shared resource that MUST NOT be concurrently accessed by more than one thread of execution.”*
- Unfortunately, this definition is somewhat misleading
  - Implies that the **piece of code** is the problem
  - In fact, the **shared resource** is the root of the problem

# Concurrent queue example

```
typedef struct node {
    int value;
    struct node *next;
} node_t;
```

```
typedef struct queue {
    node_t *head;
    node_t *tail;
} queue_t;
```

```
queue_t *queue_new() {
    queue_t *q = malloc(sizeof(queue_t));
    node_t *tmp =
        malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    return q;
}
```

```
void queue_enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));

    tmp->value = value;
    tmp->next = NULL;

    q->tail->next = tmp;
    q->tail = tmp;
}
```

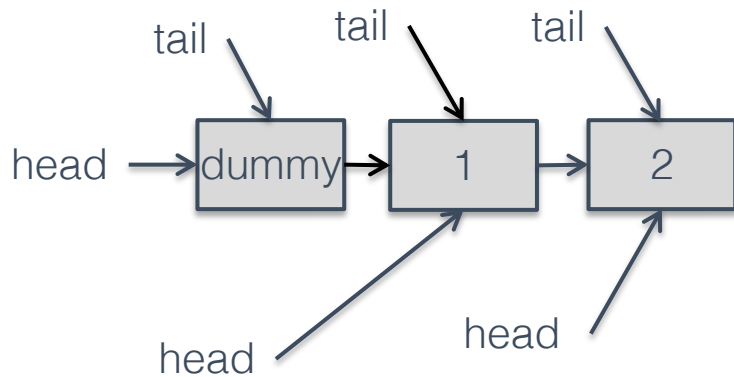
```
int queue_dequeue(queue_t *q, int *value) {
    node_t *tmp = q->head;
    node_t *new_head = tmp->next;

    if (new_head == NULL)
        return -1; // queue was empty

    *value = new_head->value;
    q->head = new_head;
    free(tmp);
    return 0;
}
```



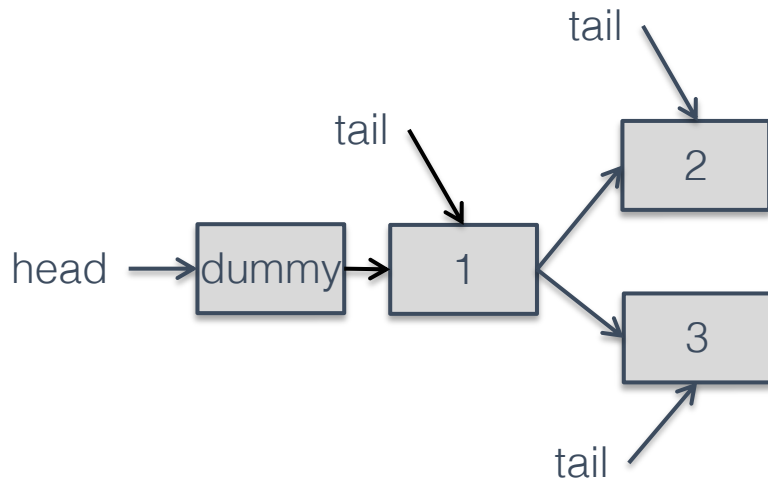
# Queue



```
void queue_enqueue(queue_t *q, int value) {  
    node_t *tmp = malloc(sizeof(node_t));  
  
    tmp->value = value;  
    tmp->next = NULL;  
  
    q->tail->next = tmp;  
    q->tail = tmp;  
}
```

```
int queue_dequeue(queue_t *q, int *value) {  
    node_t *tmp = q->head;  
    node_t *new_head = tmp->next;  
  
    if (new_head == NULL)  
        return -1; // queue was empty  
  
    *value = new_head->value;  
    q->head = new_head;  
    free(tmp);  
    return 0;  
}
```

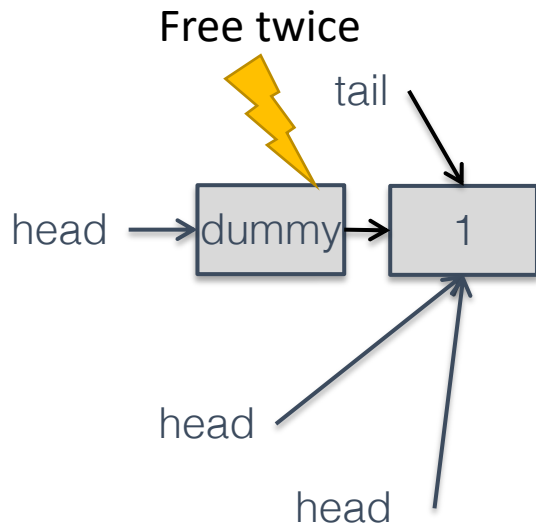
# Queue (enqueue race)



```
void queue_enqueue(queue_t *q, int value) {  
    node_t *tmp = malloc(sizeof(node_t));  
  
    tmp->value = value;  
    tmp->next = NULL;  
  
    q->tail->next = tmp;  
    q->tail = tmp;  
}
```

```
int queue_dequeue(queue_t *q, int *value) {  
    node_t *tmp = q->head;  
    node_t *new_head = tmp->next;  
  
    if (new_head == NULL)  
        return -1; // queue was empty  
  
    *value = new_head->value;  
    q->head = new_head;  
    free(tmp);  
    return 0;  
}
```

# Queue (dequeue race)

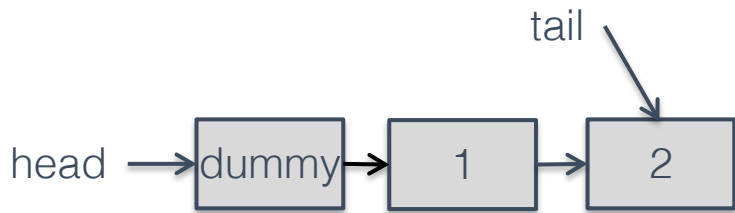


```
void queue_enqueue(queue_t *q, int value) {  
    node_t *tmp = malloc(sizeof(node_t));  
  
    tmp->value = value;  
    tmp->next = NULL;  
  
    q->tail->next = tmp;  
    q->tail = tmp;  
}
```

```
int queue_dequeue(queue_t *q, int *value) {  
    node_t *tmp = q->head;  
    node_t *new_head = tmp->next;  
  
    if (new_head == NULL)  
        return -1; // queue was empty  
  
    *value = new_head->value;  
    q->head = new_head;  
    free(tmp);  
    return 0;  
}
```

# Queue (fixes)

- Use a lock
  - Problems?
- Can use two different locks
  - Tail lock, head lock



```
void queue_enqueue(queue_t *q, int value) {  
    node_t *tmp = malloc(sizeof(node_t));  
  
    tmp->value = value;  
    tmp->next = NULL;  
  
    q->tail->next = tmp;  
    q->tail = tmp;  
}
```

```
int queue_dequeue(queue_t *q, int *value) {  
    node_t *tmp = q->head;  
    node_t *new_head = tmp->next;  
  
    if (new_head == NULL)  
        return -1; // queue was empty  
  
    *value = new_head->value;  
    q->head = new_head;  
    free(tmp);  
    return 0;  
}
```

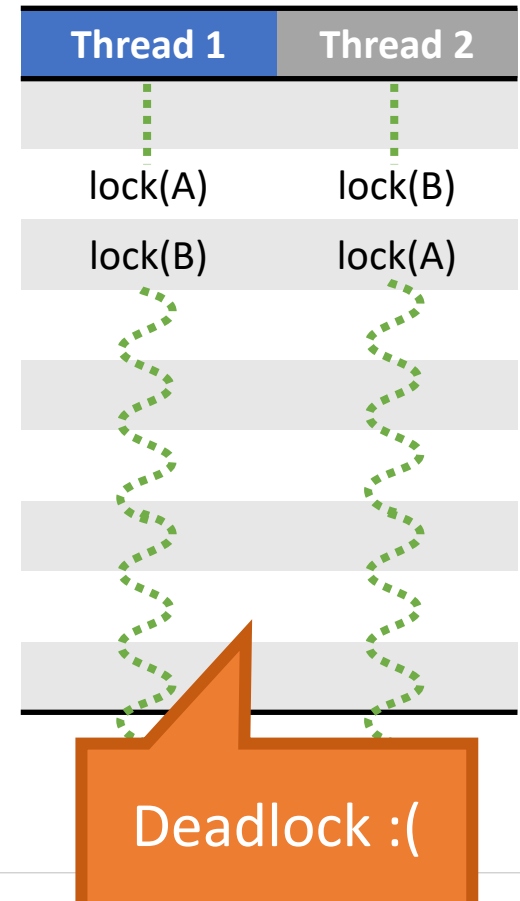
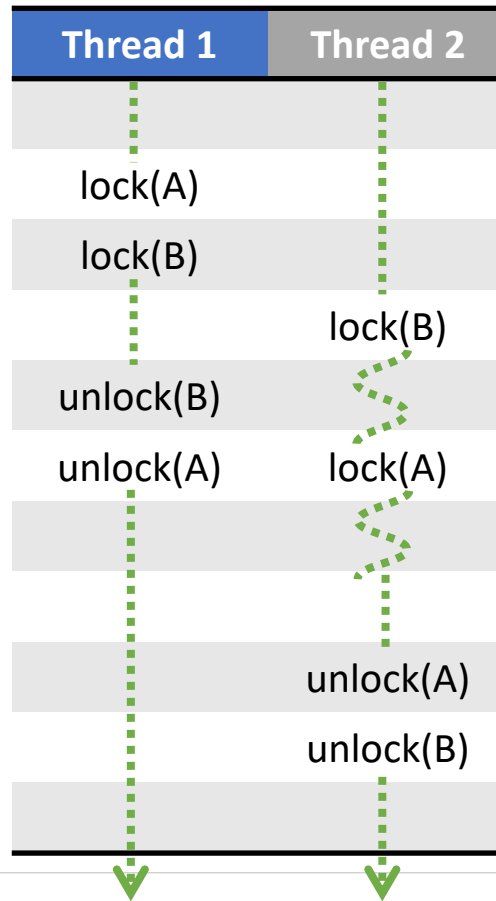
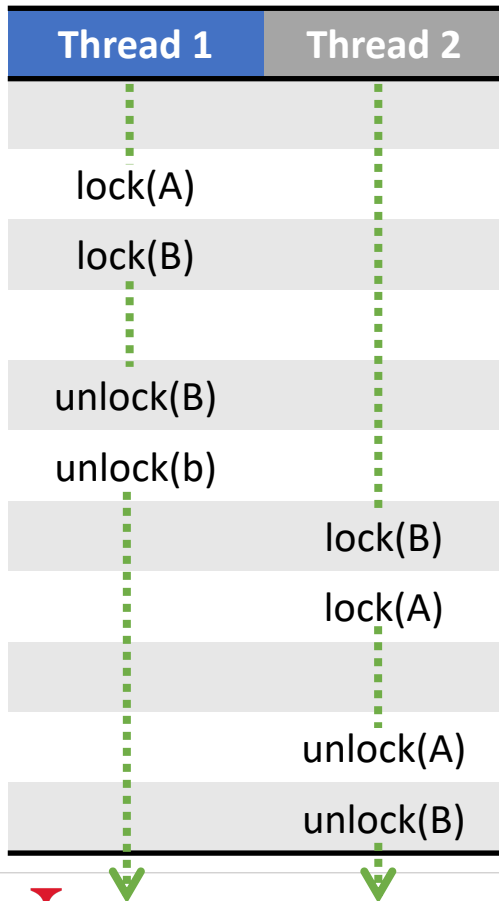
# What can go wrong with locks?

- Forgetting to unlock
  - Other threads wait indefinitely and program can freeze
- Unlocking more than once
  - Undefined behavior
- Locking more than once
  - Thread blocks at the second call

# Deadlocks

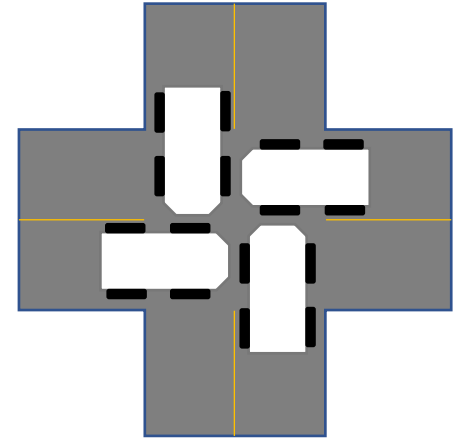
# Layers of Locks

		Thread 1	Thread 2
mutex	A	lock A	lock B
mutex	B	lock B	lock A
		// do something	// do something
		unlock B	unlock A
		unlock A	unlock B



# Deadlock

- Four necessary conditions
  - Mutual exclusion
    - Only one owner is allowed for the resource
  - Hold and wait
    - Holding on one or more resources and waiting to acquire more
  - No preemption
    - Resources cannot be taken away
  - Circular wait
    - Holding on a resource and waiting for others in circular manner
- Removing one or more conditions will resolve deadlocks
  - Use of `try_lock` and releasing existing resources upon trying to lock
  - Carefully ordering lock function call orders to avoid circular waits





# pthread\_mutex\_trylock

- Tries to acquire lock
  - If successful, return true and proceed with exclusive access
  - Else return false and proceed without exclusive access

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

```
int counter = 0;
```

```
void *thread (void *argv) {  
    for (int i = 0; i < 10; i++) {  
        if (pthread_mutex_trylock(&mtx)) {  
            counter = counter + 1;  
            pthread_mutex_unlock(&mtx);  
        }  
    }  
}
```

- Why is unlock() called only inside if statement?
- What is the final counter value if 10 thread execute concurrently?

# Thread safety

# Thread Safety

- Functions called from a thread need to be ‘thread-safe’
- A Function is thread-safe if it:
  - Always produces correct results
  - When called repeatedly from multiple concurrent threads.

# Thread-Safety Classes

- Class 1: Functions that do not protect shared variables
- Class 2: Functions that keep state across multiple invocations
- Class 3: Functions that return a pointer to a static variable
- Class 4: Functions that call thread-unsafe functions

# Thread-Unsafe Functions Class 1

- Functions that do not protect shared variables

```
// Thread with variable arguments
void *thread(void *vargp){
    counter=counter+1;
    return NULL;
}
```

# Thread-Unsafe Functions Class 1 - Fix

- Functions that do not protect shared variables
- The solution: Ensure locks are around everything

```
// Thread with variable arguments
void *thread(void *vargp){
    pthread_mutex_lock(&mutex1);
    counter = counter + 1;
    pthread_mutex_unlock(&mutex1);
    return NULL;
}
```

# Thread-Unsafe Functions Class 2

- Functions that keep state across multiple invocations

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

# Thread-Unsafe Functions Class 2

- Functions that keep state across multiple invocations

rand() is a classic example. In fact, why might we not want a race condition in our random number generator?

```
static unsigned int next = 1;
```

```
/* rand: return pseudo-random integer on 0..32767 */  
int rand(void)
```

```
{  
    next = next*1103515245 + 12345;  
    return (unsigned int)(next/65536) % 32768;  
}
```

```
/* srand: set seed for rand() */  
void srand(unsigned int seed)
```

```
{  
    next = seed;  
}
```



# Thread-Unsafe Functions Class 2

- Functions that keep state across multiple invocations

Ans: May want repeatability for testing. So since rand is deterministic, we don't want multiple threads returning the same value

```
static unsigned int next = 1;
```

```
/* rand: return pseudo-random integer on 0..32767 */  
int rand(void)
```

```
{  
    next = next*1103515245 + 12345;  
    return (unsigned int) (next/65536) % 32768;  
}
```

```
/* srand: set seed for rand() */  
void srand(unsigned int seed)
```

```
{  
    next = seed;  
}
```

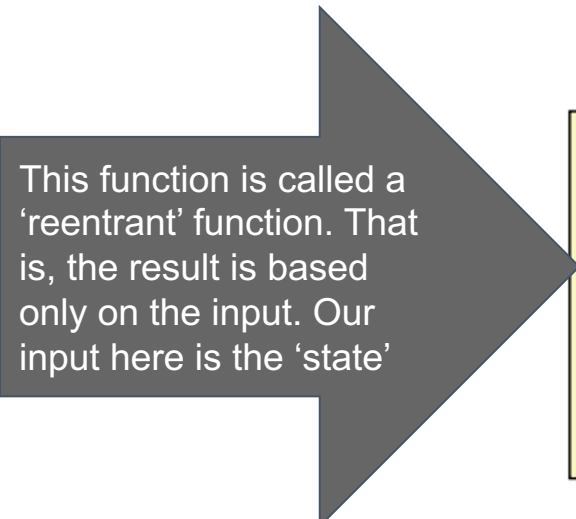
# Thread-Unsafe Functions Class 2 - Fix

- Functions that keep state across multiple invocations
- The solution: Pass state as part of an argument so 'static' can be removed

```
/* rand_r - return pseudo-random integer on 0..32767 */  
  
int rand_r(int *nextp)  
{  
    *nextp = *nextp*1103515245 + 12345;  
    return (unsigned int)(*nextp/65536) % 32768;  
}
```

# Thread-Unsafe Functions Class 2 - Fix

- Functions that keep state across multiple invocations
- The solution: Pass state as part of an argument so 'static' can be removed



This function is called a 'reentrant' function. That is, the result is based only on the input. Our input here is the 'state'

```
/* rand_r - return pseudo-random integer on 0..32767 */  
  
int rand_r(int *nextp)  
{  
    *nextp = *nextp*1103515245 + 12345;  
    return (unsigned int)(*nextp/65536) % 32768;  
}
```

# Thread-Unsafe Functions Class 3

- Functions that return a pointer to a static variable

```
/* Convert integer to string */  
char *itoa(int x)  
{  
    static char buf[11];  
    sprintf(buf, "%d", x);  
    return buf;  
}
```

# Thread-Unsafe Functions Class 3 - Fix

- Functions that return a pointer to a static variable
- The solution: Use locks, and rewrite function to return address of variable.
  - Extra mutex's can generally be used to make things thread-safe
  - May cost extra, in terms of performance.

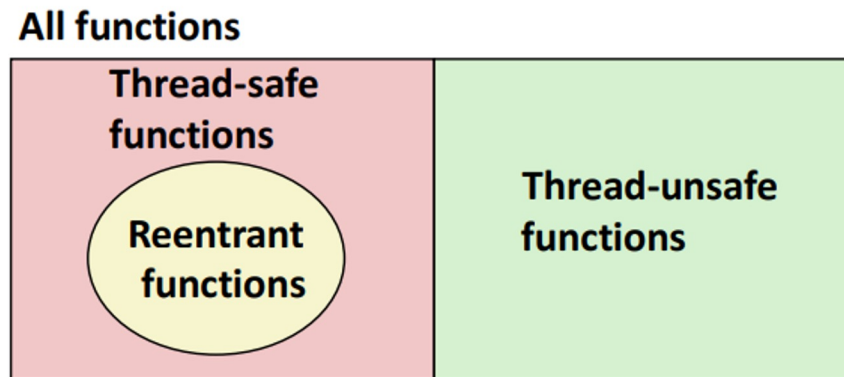
```
char *lc_itoa(int x, char *dest)
{
    P(&mutex);
    strcpy(dest, itoa(x));
    V(&mutex);
    return dest;
}
```

# Thread-Unsafe Functions Class 4

- Functions that call thread-unsafe functions
- Any function that calls a thread-unsafe function is now unsafe!
- The solution: do not call thread-unsafe functions
- Document your functions if they are thread-unsafe to prevent yourself from making errors!

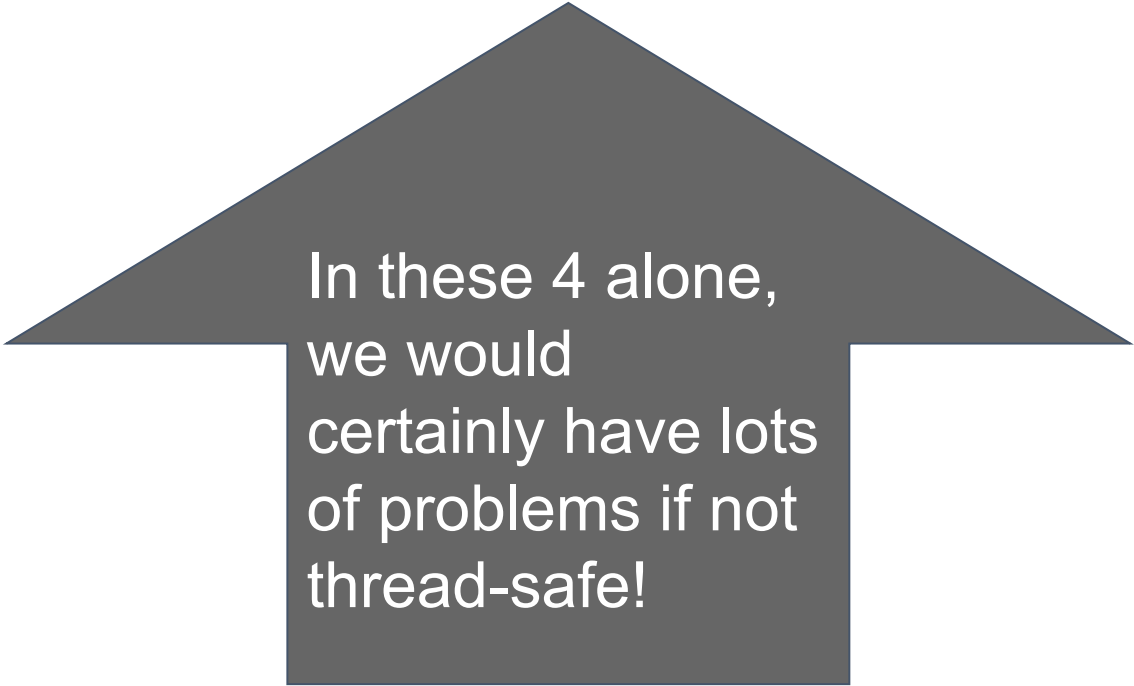
# Reentrant Functions - Recap

- A function is reentrant if it accesses no shared variables when called by multiple threads
  - Important to note because:
    - These functions require no synchronization
    - (It is the only way to fix Class 2 functions and make them thread-safe)



# Example thread-safe functions?

- What do you think, are the following thread-safe?
  - e.g. malloc, free, printf, scanf



In these 4 alone,  
we would  
certainly have lots  
of problems if not  
thread-safe!



# Example thread-safe functions

- All of the functions in the Standard C Library are thread-safe
  - e.g. malloc, free, printf, scanf
- Most Unix system calls are thread-safe. Below are a selection of exceptions. See **man pthreads** for the full list

Thread-unsafe function	Class	Reentrant version	
asctime	3	asctime_r	Time
ctime	3	ctime_r	
gethostbyaddr	3	gethostbyaddr_r	Networking
gethostbyname	3	gethostbyname_r	
inet_ntoa	3	(none)	
localtime	3	localtime_r	Time
rand	2	rand_r	Random

# Lock implementations

# Implementing Mutual Exclusion

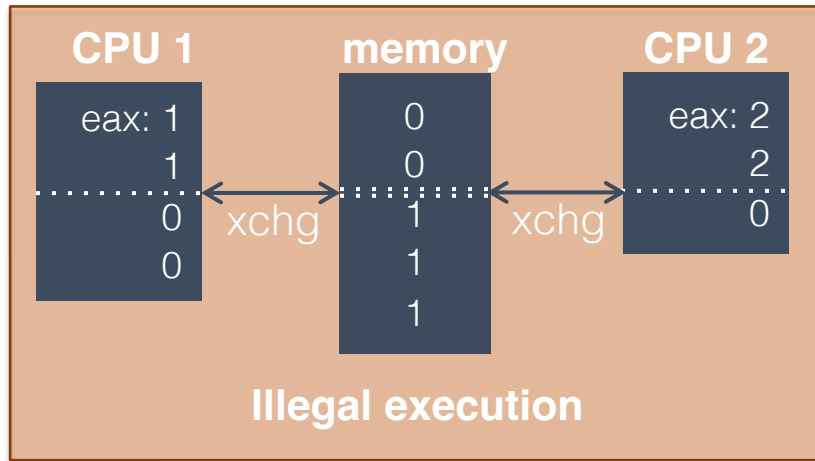
- Typically, developers don't write their own locking-primitives
  - You use an API from the OS or a library
- Why don't people write their own locks?
  - Much more complicated than they at-first appear
  - Very, very difficult to get correct
  - May require access to privileged instructions
  - May require specific assembly instructions
    - Instruction set architecture dependent

# Instruction-level Atomicity

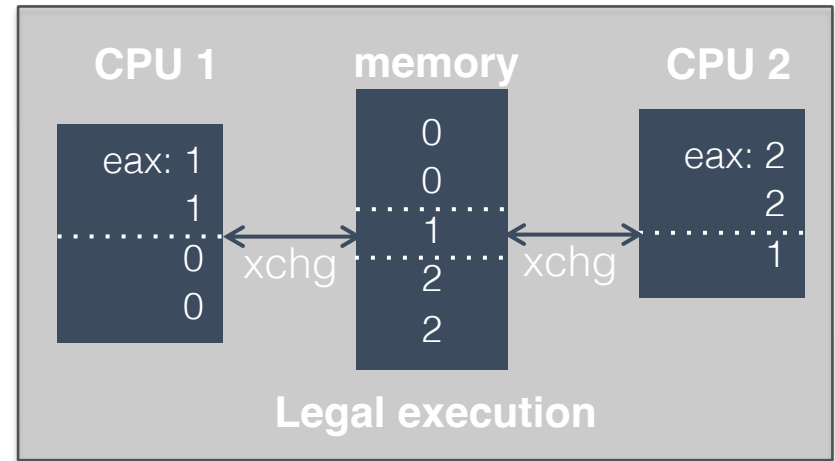
- Modern CPUs have atomic instruction(s)
  - Enable you to build high-level synchronized objects
- On x86:
  - The lock prefix makes an instruction atomic
    - `lock inc eax ; atomic increment`
    - `lock dec eax ; atomic decrement`
  - Only legal with some instructions
  - The `xchg` instruction is guaranteed to be atomic
    - `xchg eax, [addr] ; swap eax and the value in memory`

# Behavior of xchg

## Non-Atomic xchg



## Atomic xchg



- Atomicity ensures that each xchg occurs before or after xchg's from other CPUs

# Building a Spin Lock with xchg

spin\_lock:

```

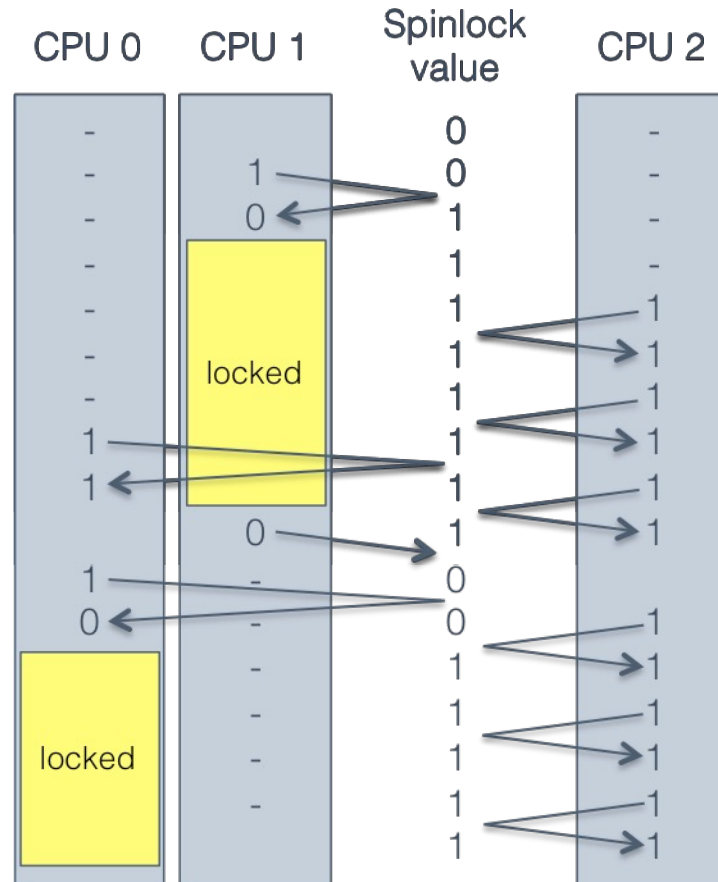
mov eax, 1
xchg eax, [lock_addr]
test eax, eax
jnz spin_lock
    
```

If (1<sup>st</sup> & 2<sup>nd</sup>) == 0 then ZF=1  
 else ZF=0  
 ...

spin\_unlock:

```

mov [lock_addr], 0
    
```



CPU 1 locks.

CPUs 0 and 2 both try to lock, but cannot.

CPU 1 unlocks.

CPU 0 locks, simply because it requested it *slightly* before CPU 2.

Do you see any problem with spinlocks?

# Building a Multi-CPU Mutex (avoids extensive spinning)

```
typedef struct mutex_struct {
    int spinlock = 0; // spinlock variable
    int locked = 0;   // is the mutex locked? guarded by spinlock
    queue waitlist;  // waiting threads, guarded by spinlock
} mutex;

void mutex_lock(mutex * m) {
    spin_lock(&m->spinlock);
    if (!m->locked) {
        m->locked = 1;
        spin_unlock(&m->spinlock);
    }
    else {
        m->waitlist.add(current_process);
        spin_unlock(&m->spinlock);
        yield();
        // wake up here when the mutex is acquired
    }
}
```

# Building a Multi-CPU Mutex (avoids extensive spinning)

```
typedef struct mutex_struct {
    int spinlock = 0; // spinlock variable
    int locked = 0;   // is the mutex locked? guarded by spinlock
    queue waitlist;  // waiting threads, guarded by spinlock
} mutex;

void mutex_unlock(mutex * m) {
    spin_lock(&m->spinlock);
    if (m->waitlist.empty()) {
        m->locked = 0;
        spin_unlock(&m->spinlock);
    }
    else {
        next_thread = m->waitlist.pop_from_head();
        spin_unlock(&m->spinlock);
        wake(next_thread);
    }
}
```



# Semaphores

# Semaphores

- Generalization of a mutex
  - Invented by Edsger Dijkstra
  - Associated with a positive **integer N**
  - May be locked by up to N concurrent threads
- Semaphore methods
  - `sem_wait()`: **N--; if N < 0 then sleep;**
    - Wait/aquire/lock
    - Also commonly known as P (proberen – test) operation
  - `sem_post()`: **N++; if waiting threads > 0, wake one up;** // a.k.a. V()
    - Unlock
    - Also commonly known as V (verhogen – increment) operation
- Depending on the initial value N, interesting features can be implemented

# Semaphore

```
sem_t s;
```

```
sem_init(&s, 0, 1);
```

```
int sem_wait(sem_t *s) {
```

```
    // executes atomically
```

```
    decrement the value of semaphore s by one
```

```
    wait if value of semaphore s is negative
```

```
}
```

```
int sem_post(sem_t *s) {
```

```
    // executes atomically
```

```
    increment the value of semaphore s by one
```

```
    if there are one or more threads waiting, wake one
```

```
}
```

# C semaphore programming example

- API
  - `#include <semaphore.h>`
  - `int sem_init(sem_t *s, 0, unsigned int val)`
    - Second argument: shared among threads (0) vs processes (non-zero)
    - Third argument: initial value of N
  - `int sem_wait(sem_t *s);`
  - `int sem_post(sem_t *s);`
  - `int sem_destroy(sem_t *sem);`

# Using semaphores for mutual exclusion

- Basic Idea:
  - Associate a unique semaphore  $S$ , initially 1
    - (i.e. 1 spot open for a thread to enter)
  - Surround corresponding critical sections with  $P(S)$  and  $V(S)$  operations
- Binary semaphore: Semaphore whose value is always 0 or 1
  - P operation: “locking” the mutex
  - V operation: “unlocking” or “releasing” the mutex
  - “Holding” a mutex: locked and not yet unlocked
- Counting semaphore: Used as a counter for set of available resources.

# The Bounded Buffer Problem

- We want to keep the buffer size to a limit
- Multiple threads puts and gets from the buffer

```
list      buffer
put(item):
    if len(buffer) >= N
        return ERROR
    else
        buffer.add_tail(item)

Get():
    if len(buffer) == 0
        return NULL
    else
        return buffer.remove_head()
```

```
list      buffer
mutex     m
put(item):
    m.lock()
    if len(buffer) >= N
        m.unlock()
        return ERROR
    else
        buffer.add_tail(item)
        m.unlock()

Get():
    m.lock()
    if len(buffer) == 0
        m.unlock()
        return NULL
    else
        tmp = buffer.remove_head()
        m.unlock()
        return tmp
```

# The Bounded Buffer Problem

- Use of semaphore can limit the number of threads that can put/get at the same time

```
class semaphore_bounded_buffer:
```

```
    mutex      m
```

```
    list      buffer
```

```
    semaphore S_space = semaphore(N)
```

```
    semaphore S_items = semaphore(0)
```

```
    put(item):
```

```
        S_space.wait()
```

```
        m.lock()
```

```
        buffer.add_tail(item)
```

```
        m.unlock()
```

```
        S_items.post()
```

```
    get():
```

```
        S_items.wait()
```

```
        m.lock()
```

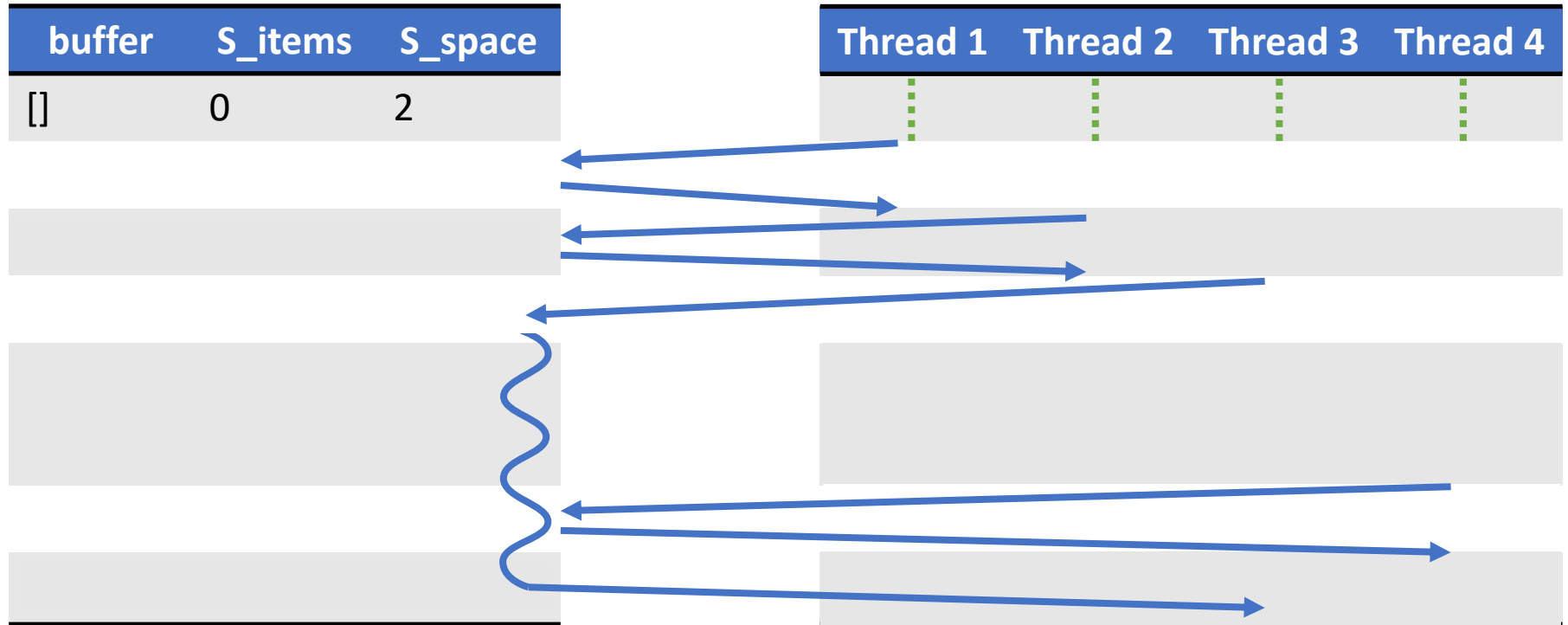
```
        result = buffer.remove_head()
```

```
        m.unlock()
```

```
        S_space.post()
```

```
        return result
```

# Example Bounded Buffer





# Signaling and condition variables

# Signaling

- Goal: Once something happens in one thread, then another thread may proceed

```
sem_init(&s, 0, 0)
```

Thread A

```
statement A1  
sem.post(&s)  \\ send signal
```

Thread B

```
sem.wait(&s)  \\ wait until  
post  
statement B1
```

# The problem...

- Suppose a thread wants to check a condition is TRUE before continuing
- Say the main thread wants to see if a child thread has finished some operation, how could the wait be implemented?
  - parent: begin
  - child: completes a task and moves on to the next step
  - parent: does something according to the completion
- All we have is join...
- Could we use the state of a shared variable?

# The problem...

- Yes, but performance is NOT good: inefficient and wastes CPU time

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    pthread_create(&p, NULL, child, NULL);
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    done = 0;
    pthread_create(&p, NULL, child, NULL);
    while (done == 0)
        ; // spin
    ...
    return 0;
}
```

```
int done = 0;

void *child(void *arg) {
    printf("child\n");
    sleep(5);
    done = 1;
    sleep(10);
    return NULL;
}
```

Note: this code is unsafe

# Use a condition variable

- condition variable: an explicit queue that threads can put themselves on to wait for some state/condition to change
- when it changes: wake one of waiting threads and allow them to continue
  
- Two operations: wait() and signal()
  - wait(): a thread wishes to put itself to sleep
    - pthread\_cond\_wait()
  
  - signal(): when a condition has changed and a thread needs to be waken from sleeping
    - pthread\_cond\_signal()

# Use a condition variable

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    pthread_create(&p, NULL, child, NULL);
    pthread_mutex_lock(&m);
    while (done == 0) {
        // releases lock when going to sleep
        pthread_cond_wait(&c, &m);
        // when woken up it automatically
        // acquires the lock
    }
    pthread_mutex_unlock(&m);
    printf("parent: end\n");
    return 0;
}
```

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int done = 0;
```

```
void *child(void *arg) {
    printf("child\n");
    sleep(1);

    pthread_mutex_lock(&m);

    done = 1;

    pthread_cond_signal(&c);

    pthread_mutex_unlock(&m);

    sleep(10);
    return NULL;
}
```

# Summary of Synchronization

- Programmers need a clear model of how variables are shared by threads
- Variables shared by multiple threads must be protected to ensure mutually exclusive access
- Deadlocks must be prevented
  
- Synchronization primitives
  - Mutex
  - Semaphores
  - Condition variables