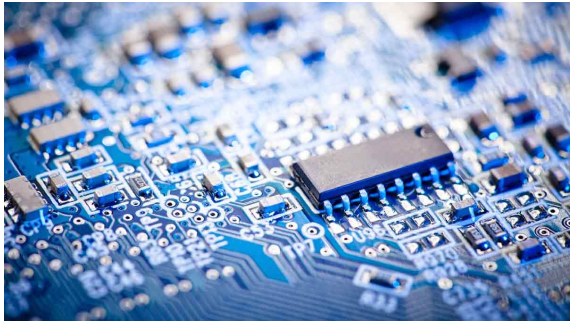
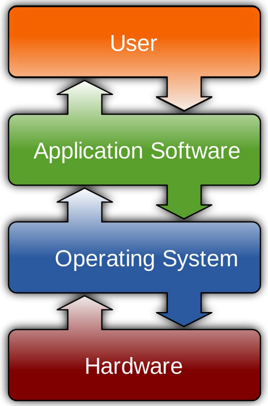
Please do not redistribute these slides without prior written permission



CS 3650
Computer Systems

Alden Jackson / Ferdinand Vesely



Intro	Virtualization		Concurrency	Persistence	Appendices
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	
		21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs	
		22 Swapping: Policies		45 Data Integrity and Protection	
		23 Case Study: VAX/VMS		46 *Summary*	
		24 *Summary*		47 *Dialogue*	
				48 Distributed Systems	
				49 Network File System (NFS)	
				50 Andrew File System (AFS)	
				51 *Summary*	

Lecture 9 - Concurrency

Part 2

Bank Transactions

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

1. If I start with **$25** in my checkings 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.

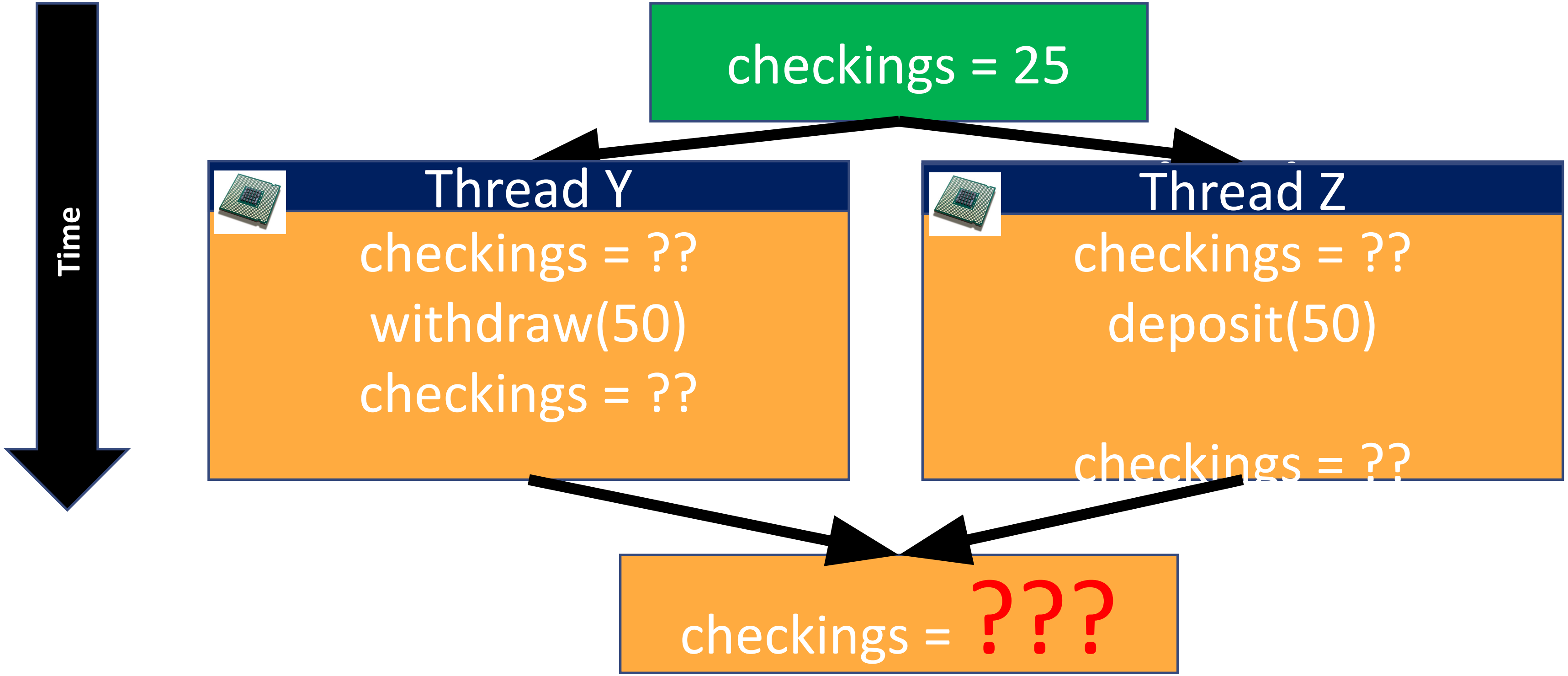
Concurrent Bank Transaction

1. If I start with **$25** in my checkings 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.

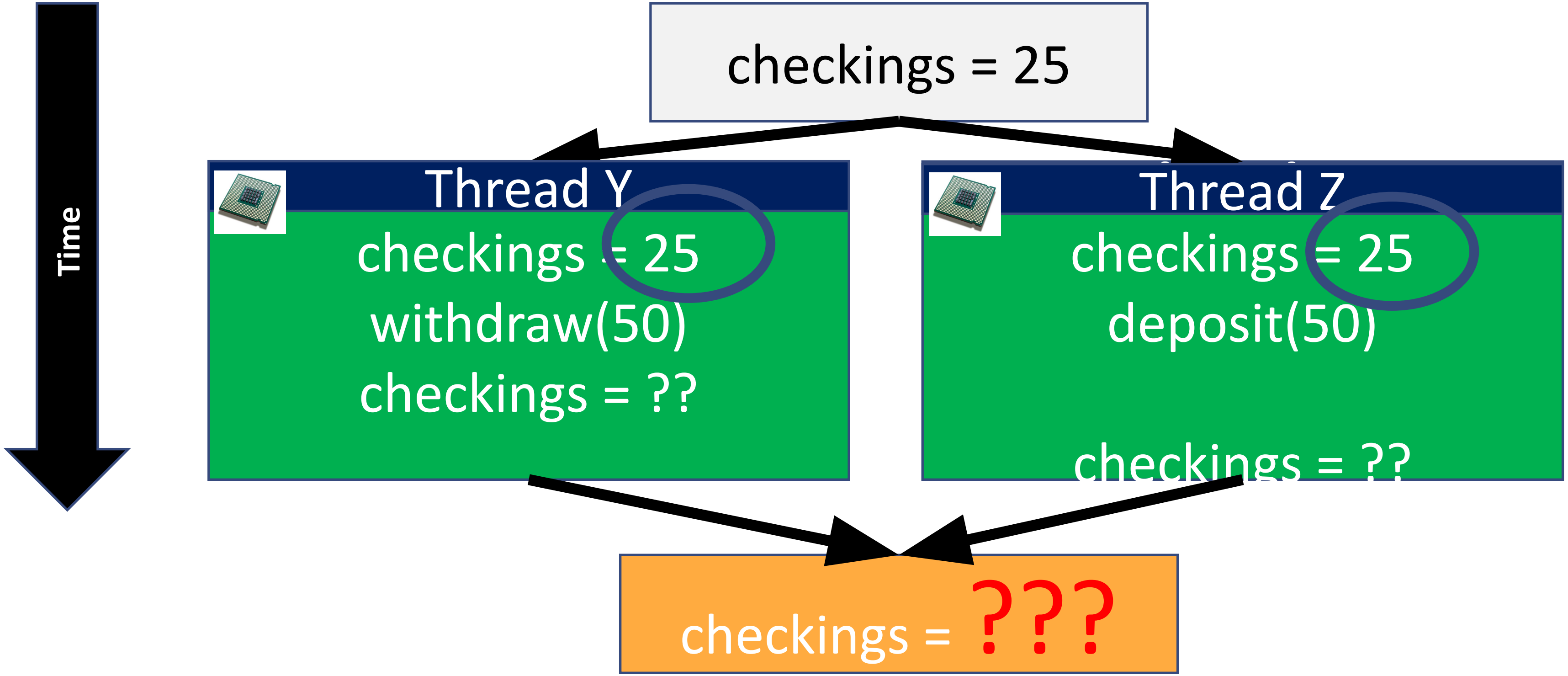
Concurrent Bank Transaction

1. If I start with **$25** in my checkings 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.

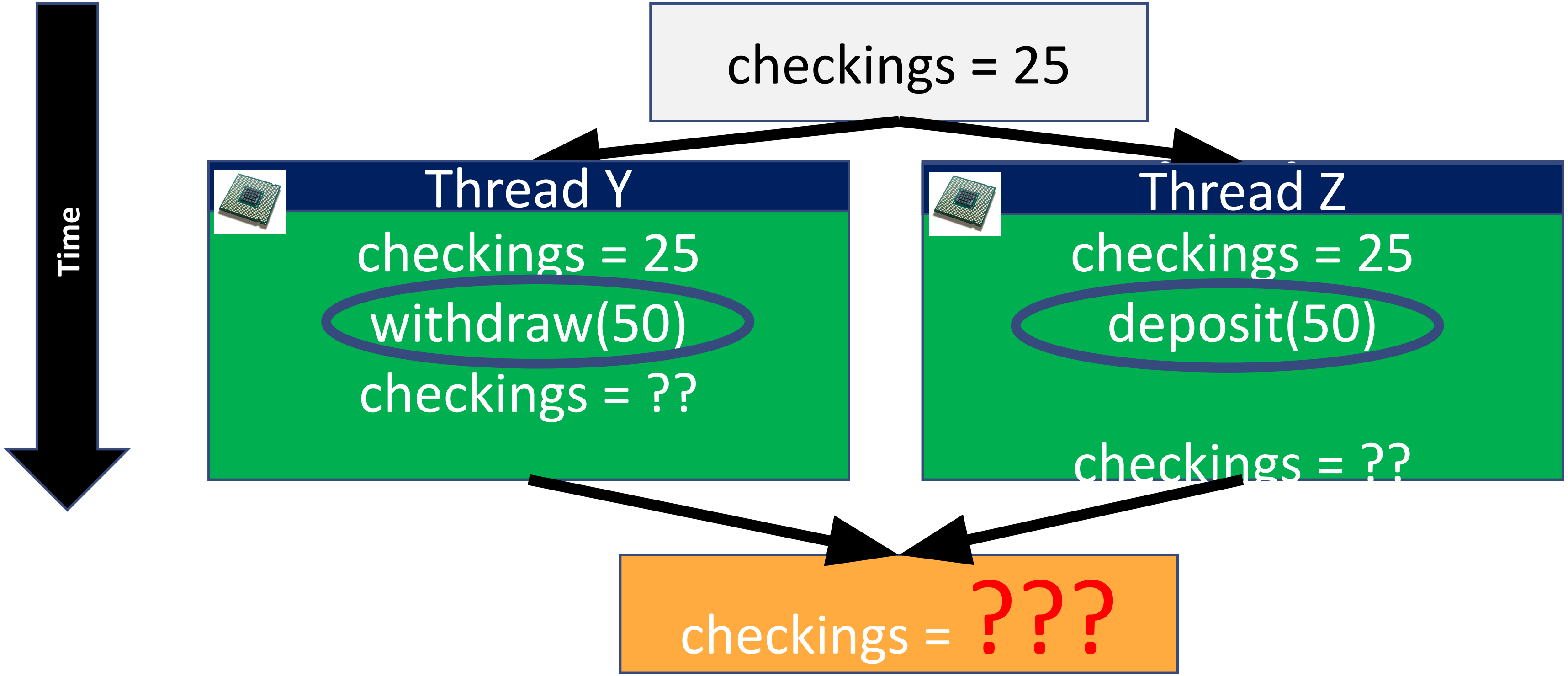
Read our initial balance



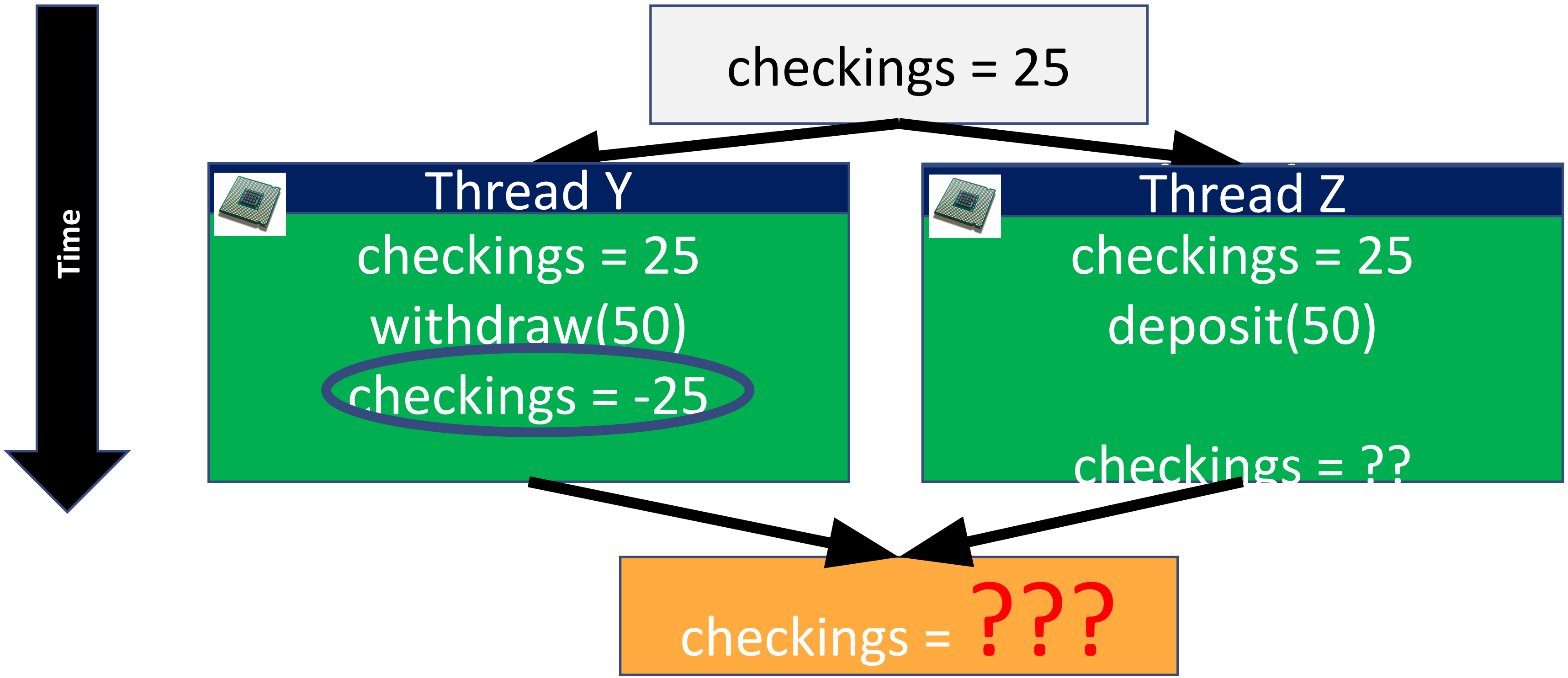
Okay, we have $25 – now move on



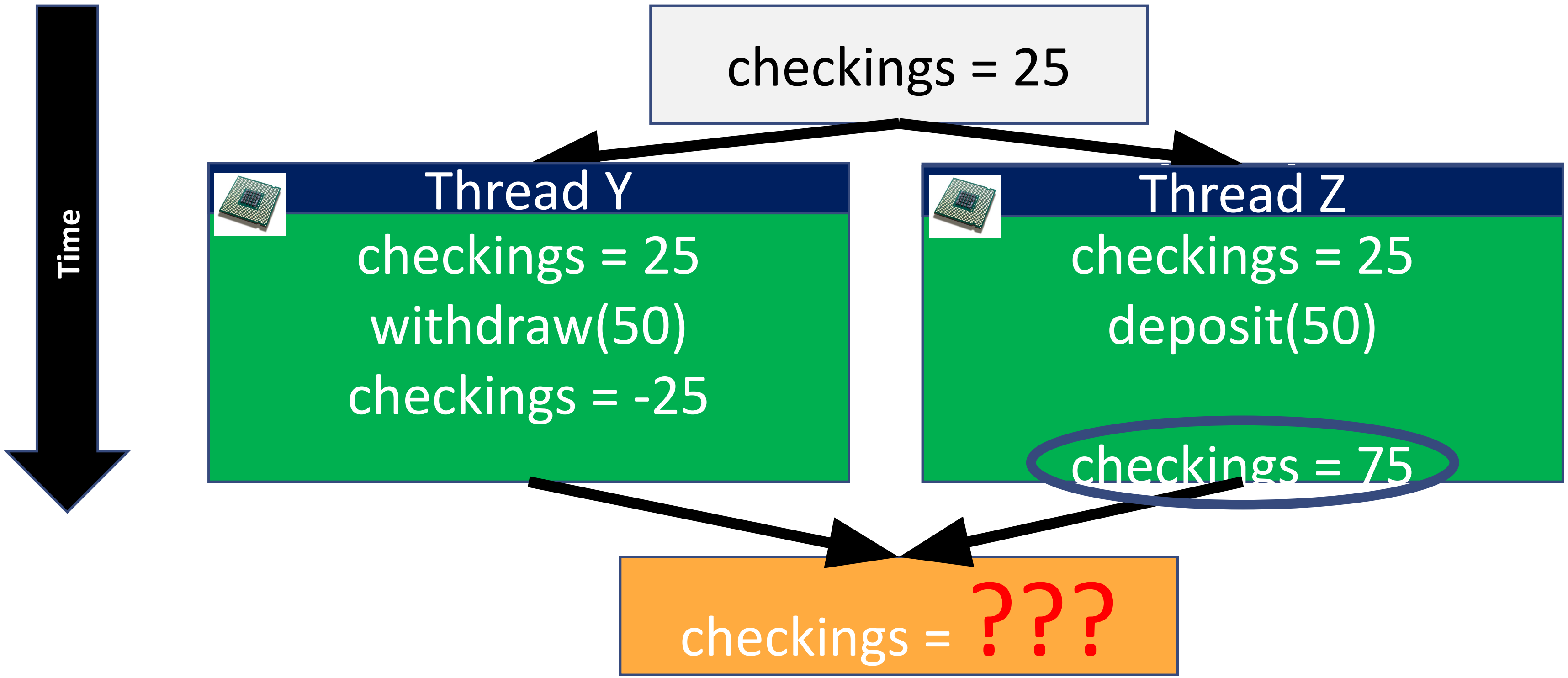
withdraw and deposit occur (Thread Y and Z)



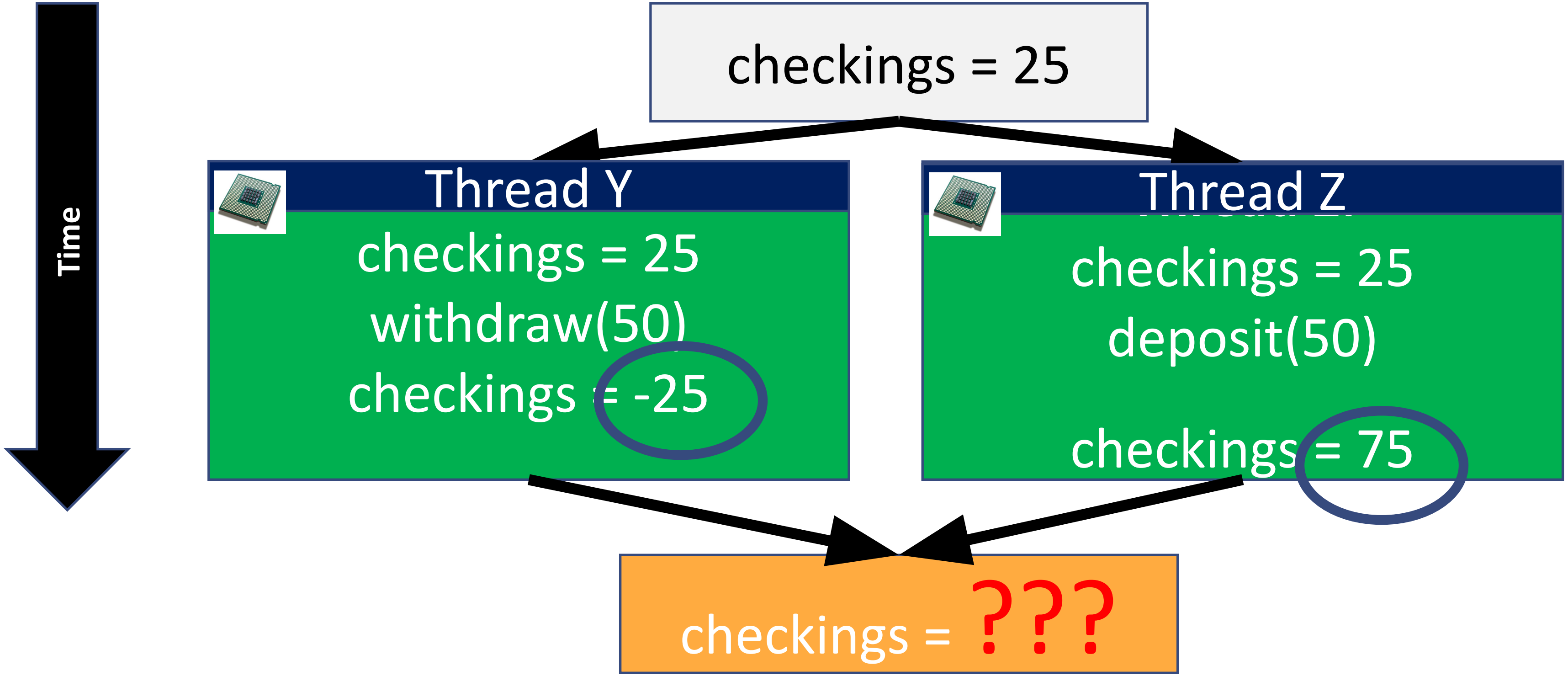
Checkings from Thread Y updates first



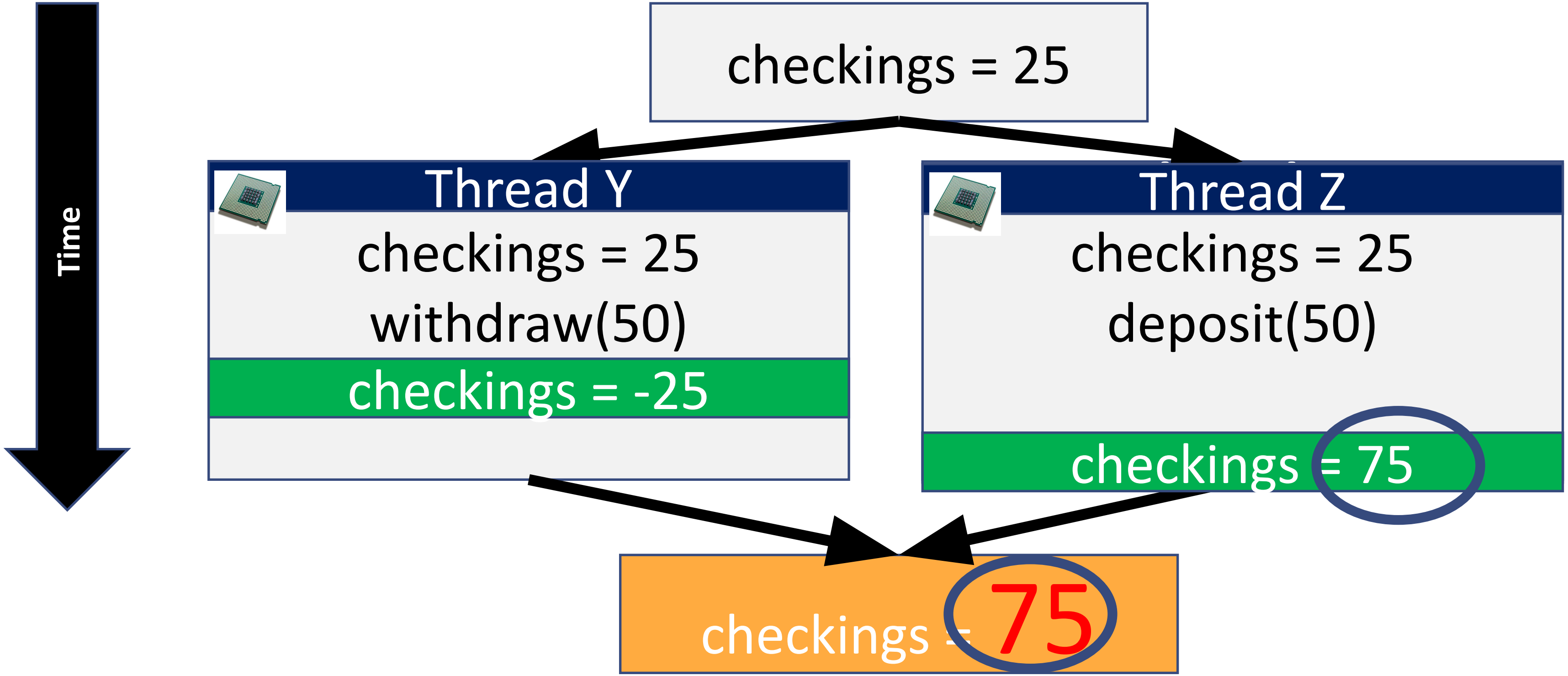
(Thread Z) updates its checkings value shortly after



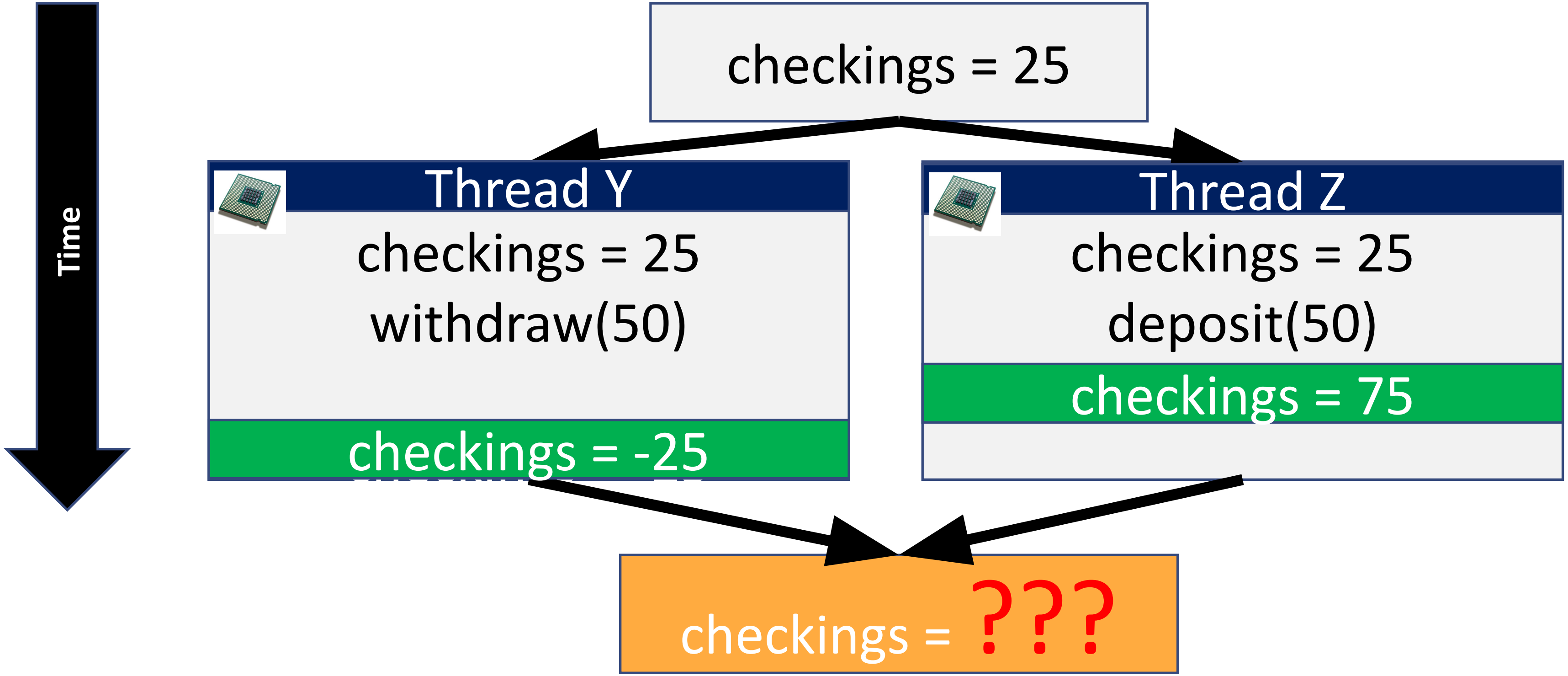
Now we have conflicting information



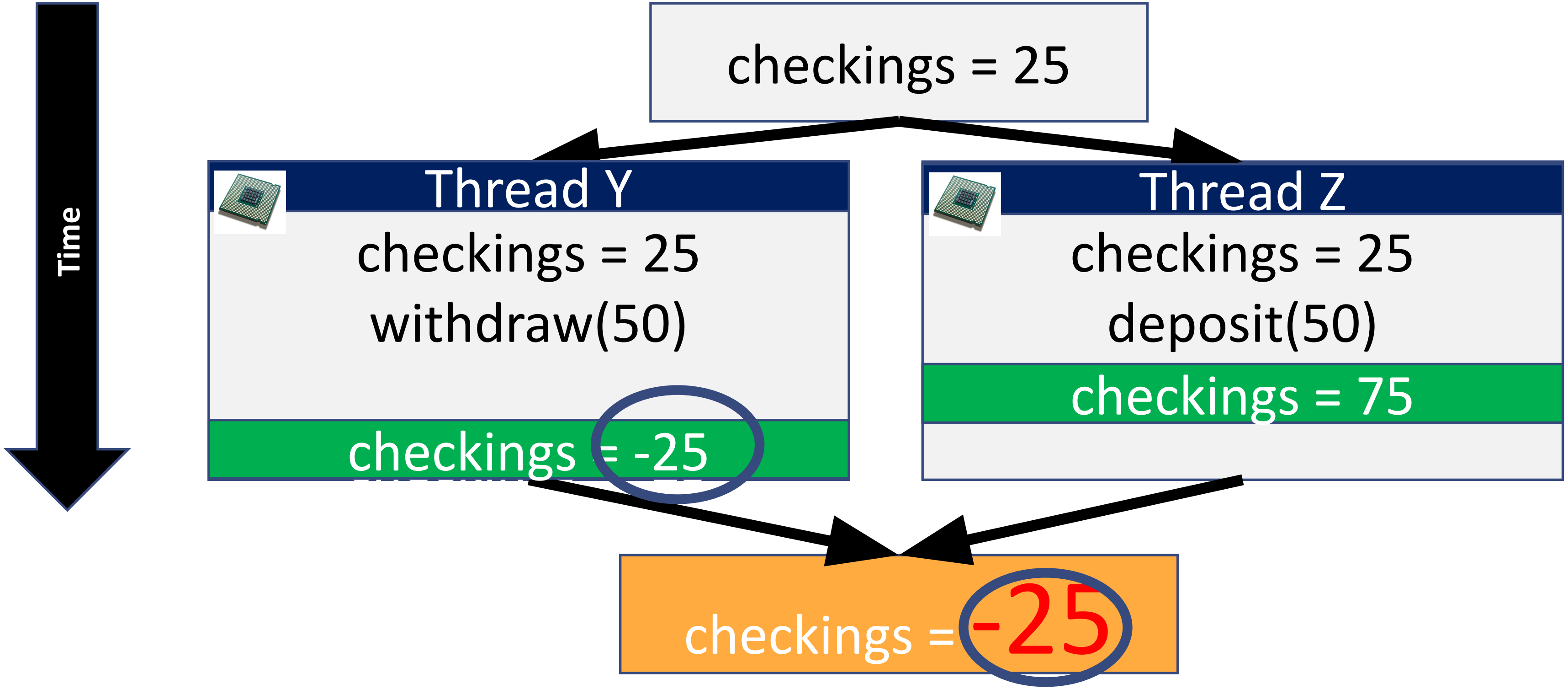
checkings stores the last value of 75 (Thread Z)



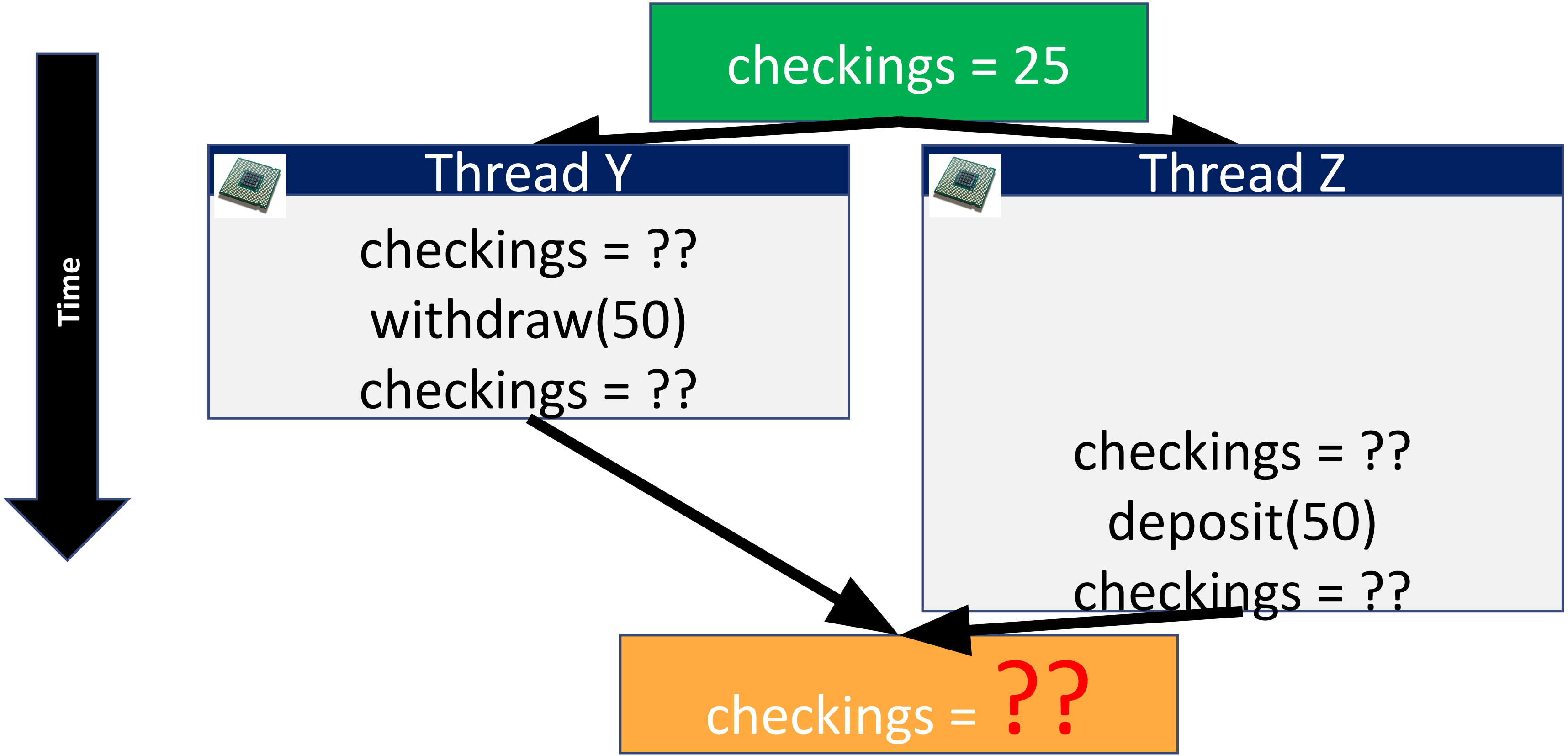
What if these operations had swapped!



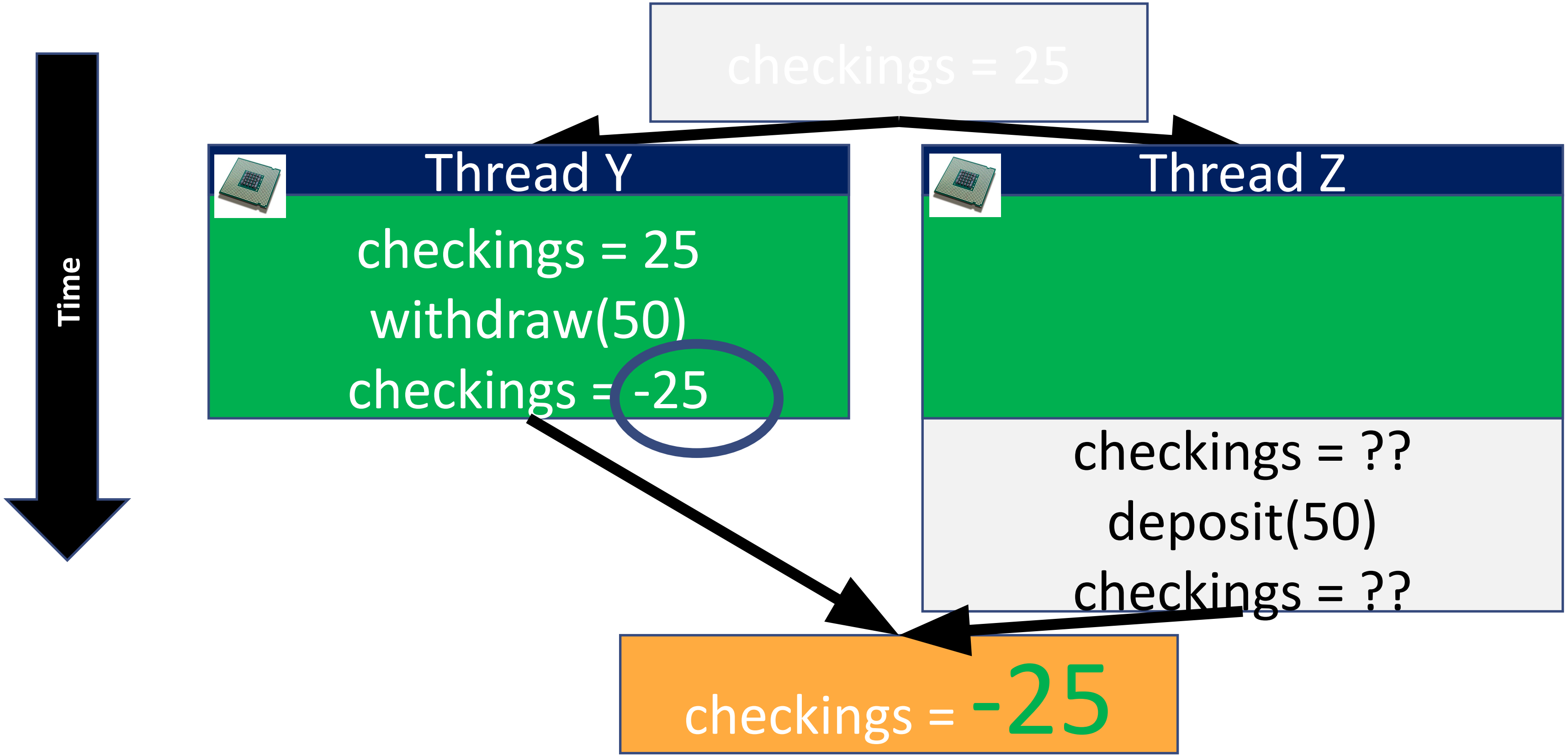
This time our balance is -25! (Thread Y)



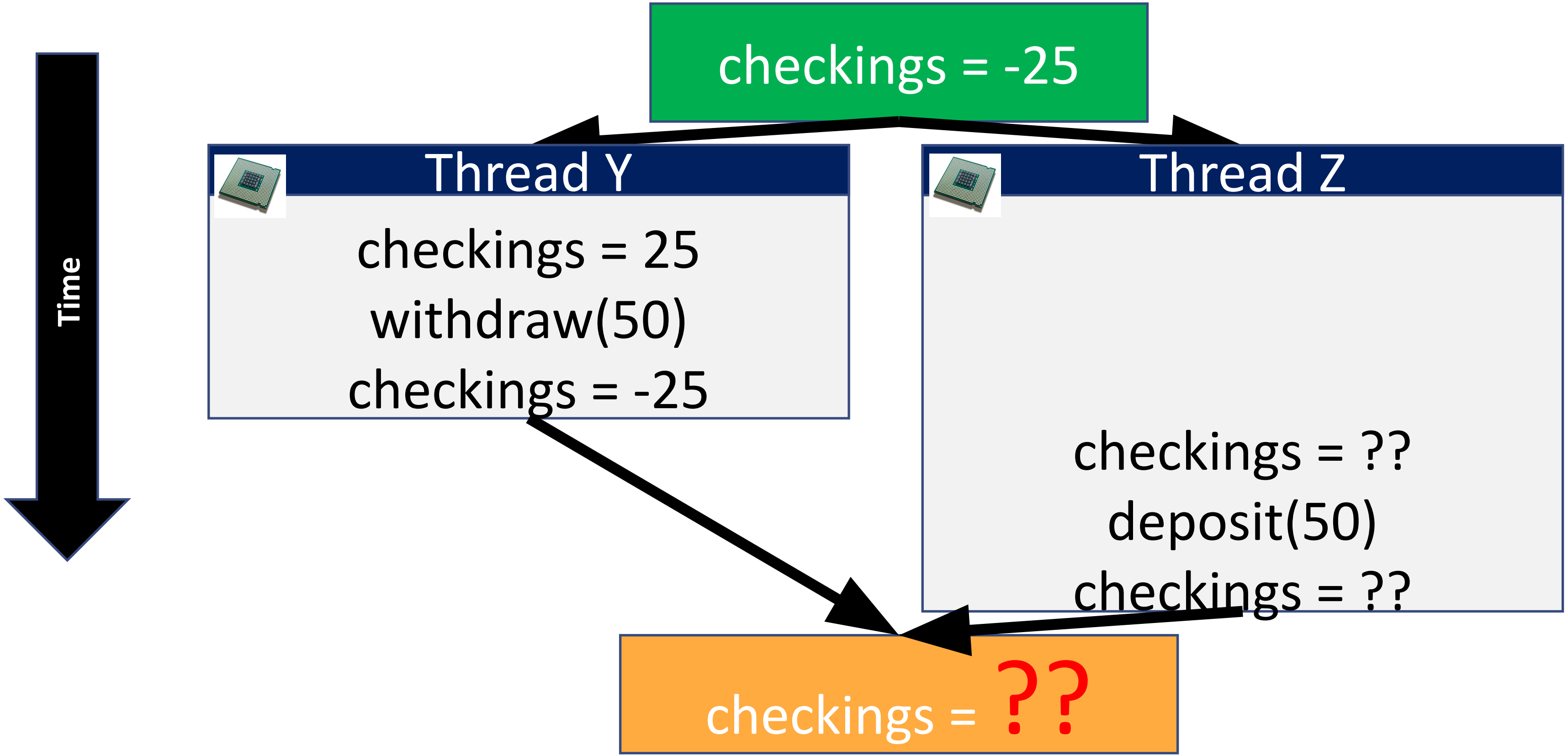
How about if Thread Z lags behind Thread Y?



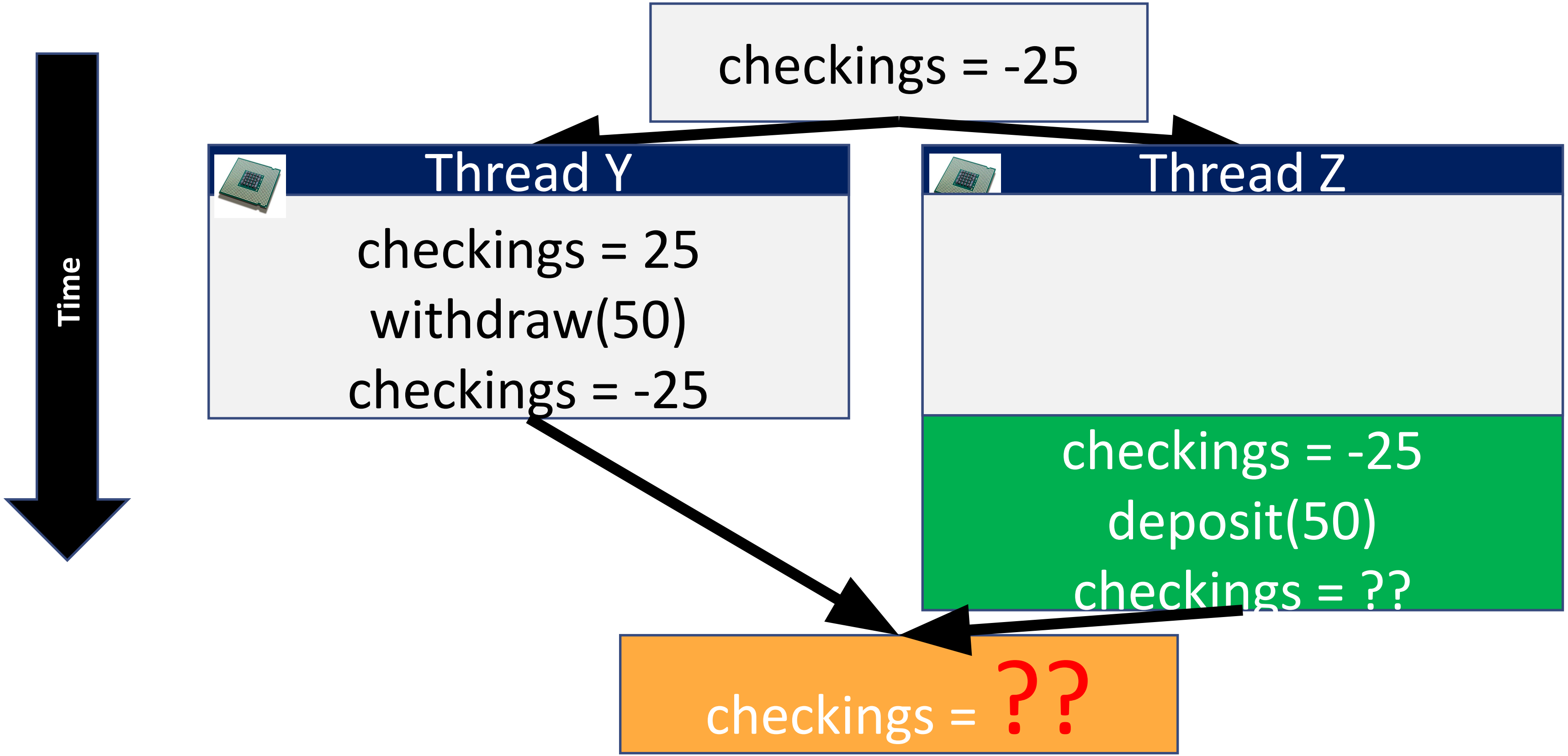
How about if Thread Z lags behind Thread Y?



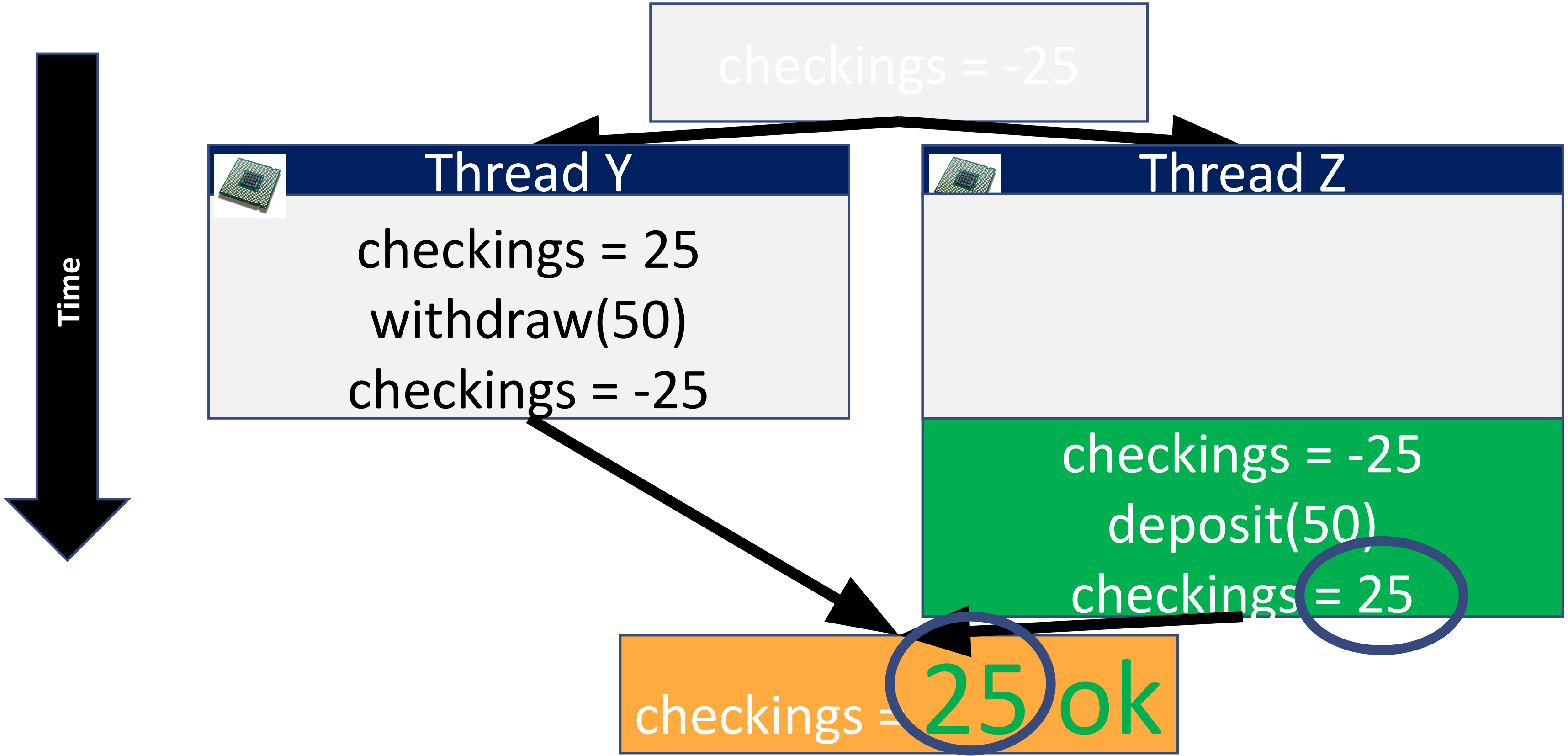
How about if Thread Z lags behind Thread Y?



How about if Thread Z lags behind Thread Y?



Okay—this time we happen to get 25

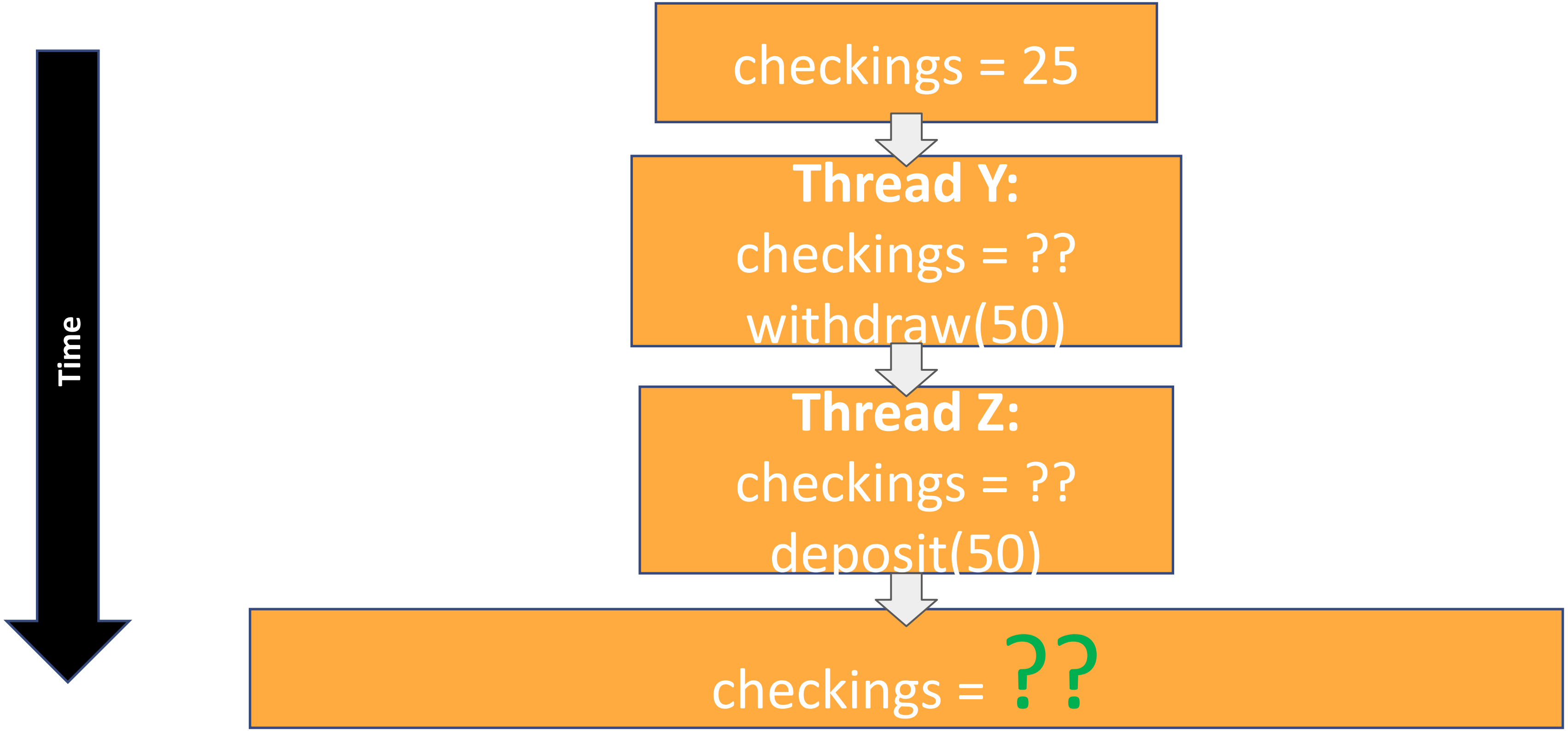


checkings = 75	checkings = -25	checkings = 25 ok

We have witnessed a data race

a common concurrency problem

We need to **synchronize** – enforce ordering



Read our checkings

Time

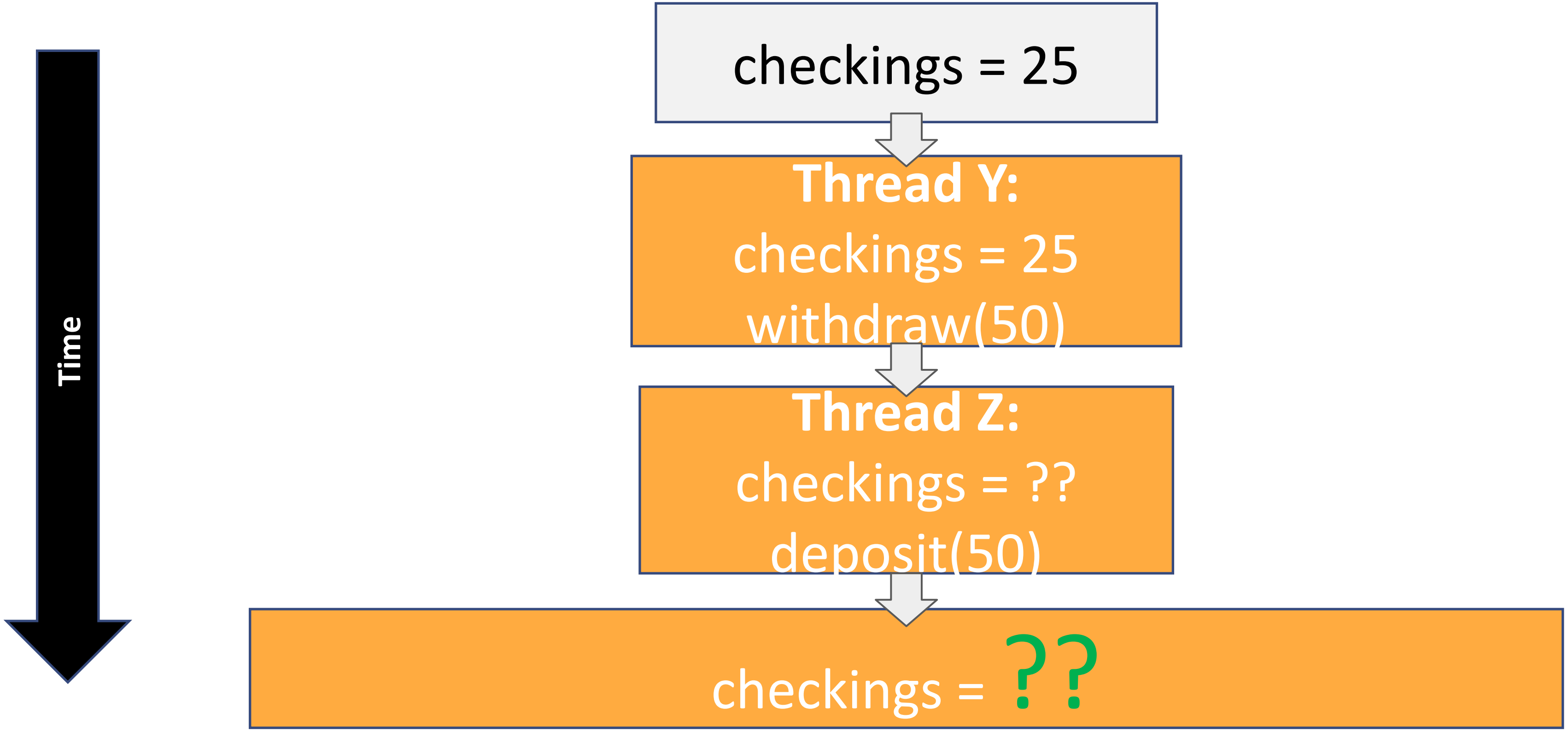
checkings = 25

Thread Y:
checkings = ??
withdraw(50)

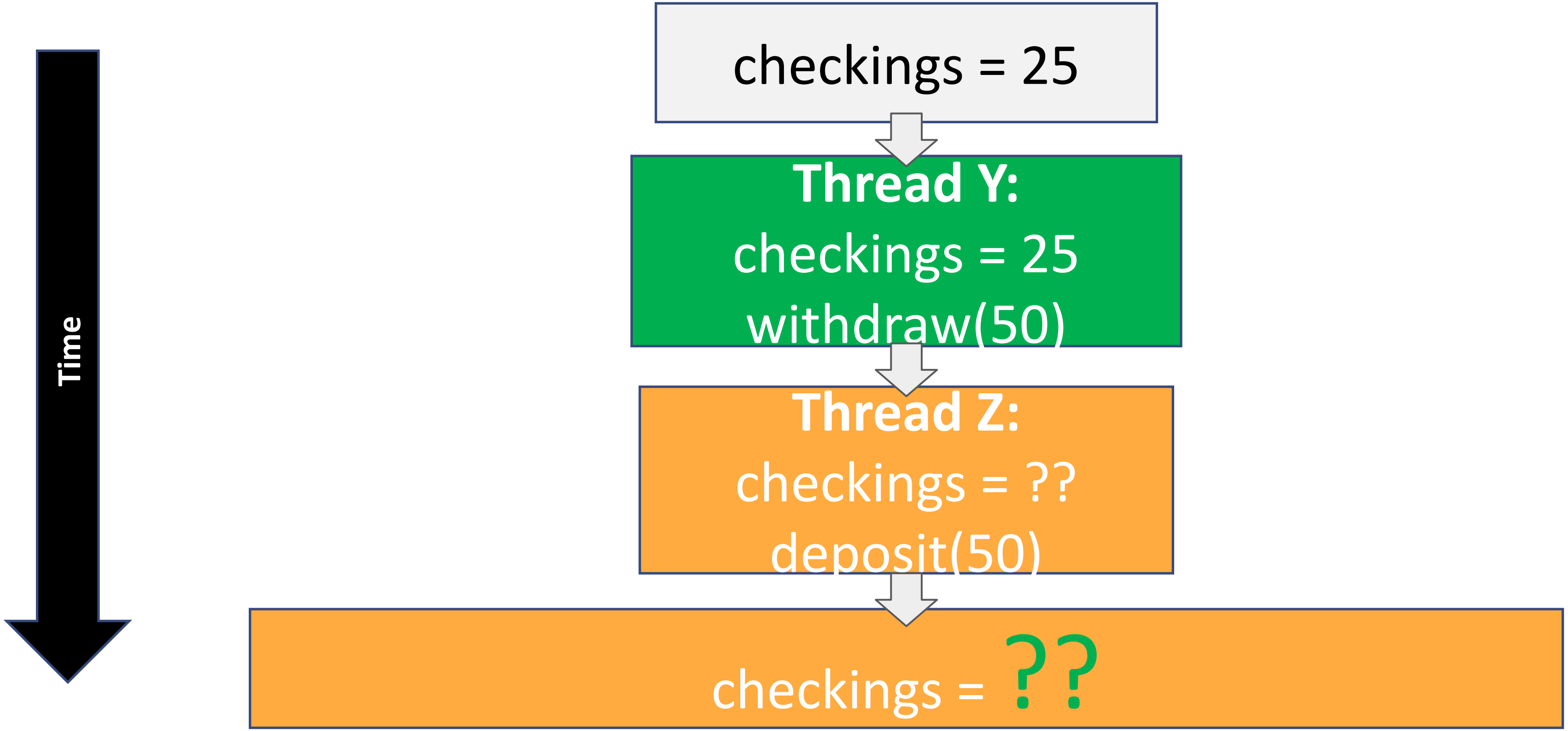
Thread Z:
checkings = ??
deposit(50)

checkings = ??

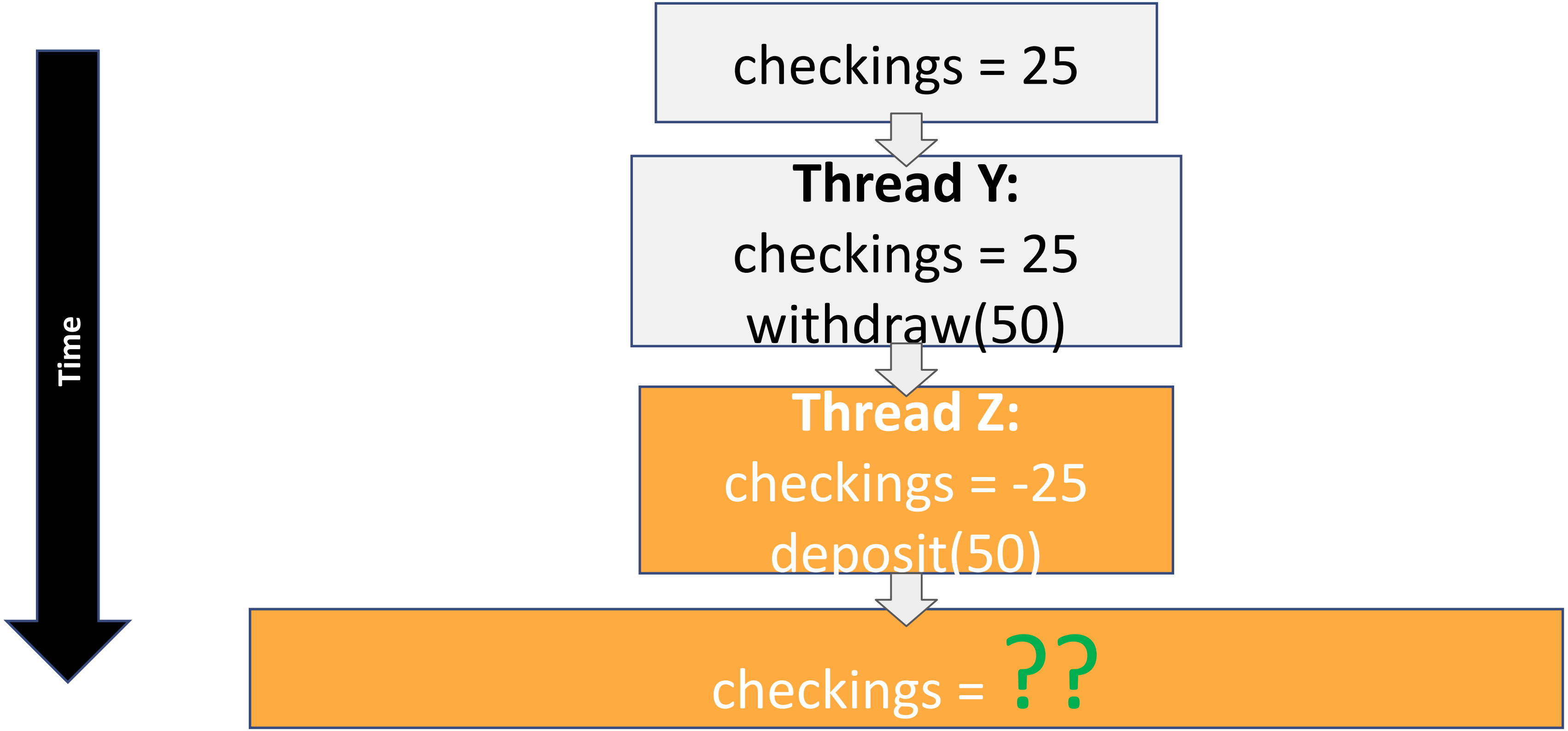
Thread Y uses checkings=25



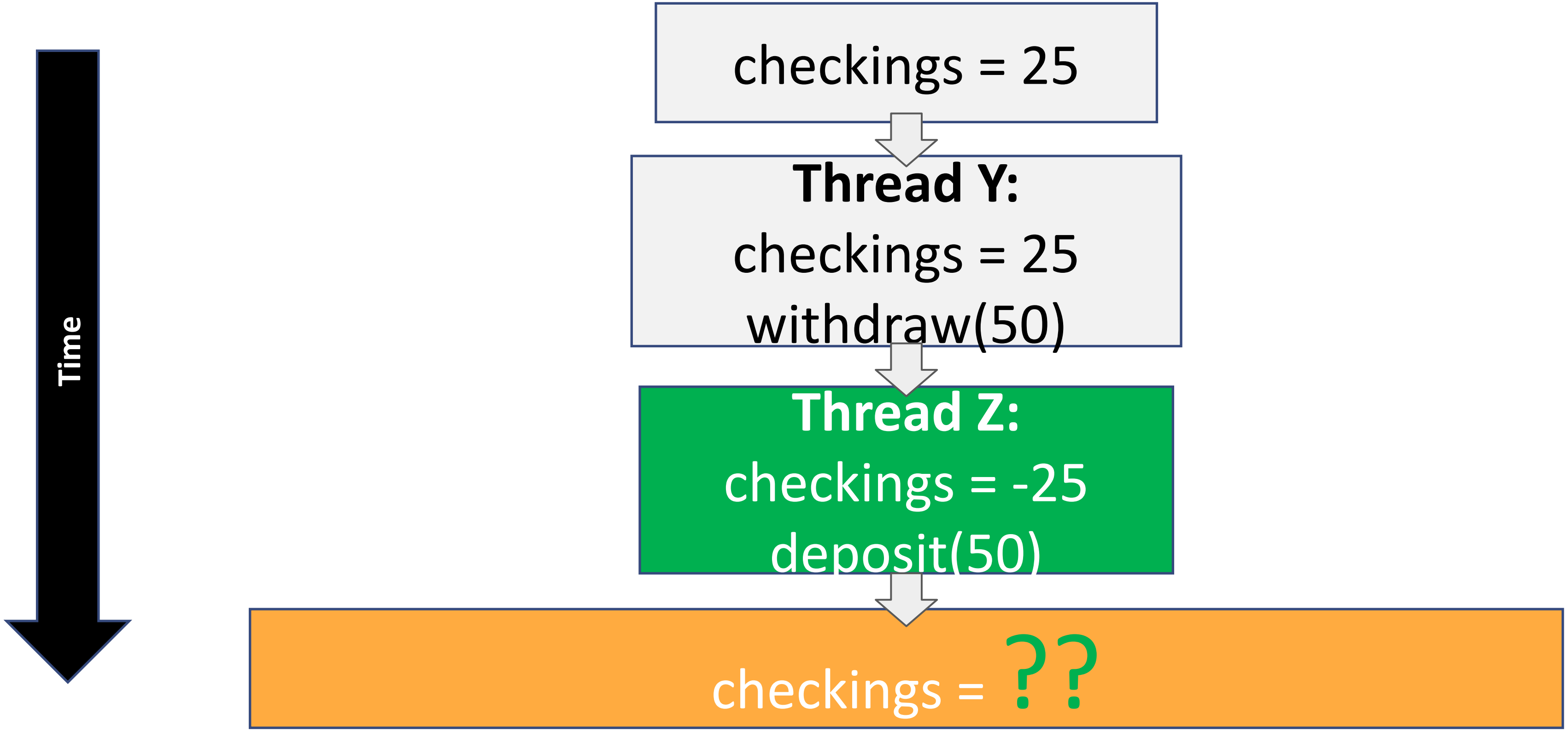
Thread Y withdraws(50)



Thread Z reads in checkings



Thread Z deposits(50)



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

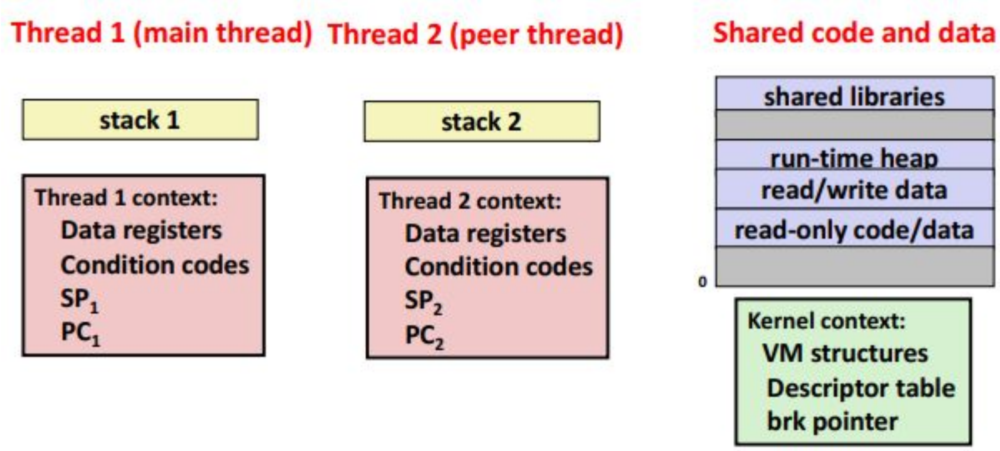
(The Bug!)

- This time launch 10000 threads
- counter is shared between threads
- 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!

```c
// Compile with:
//
// clang -lpthread thread3.c -o thread3
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;

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

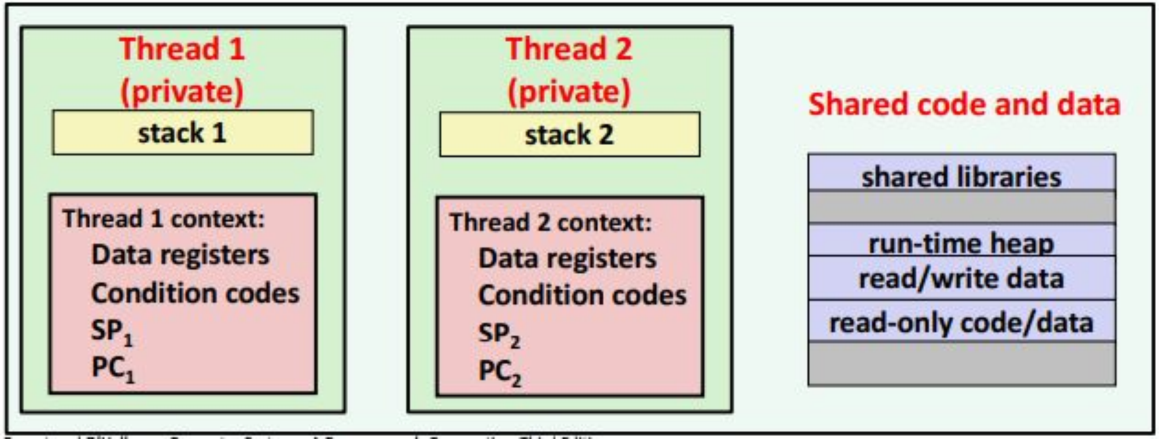
What Data is Shared in Threaded C Programs?

- 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.)



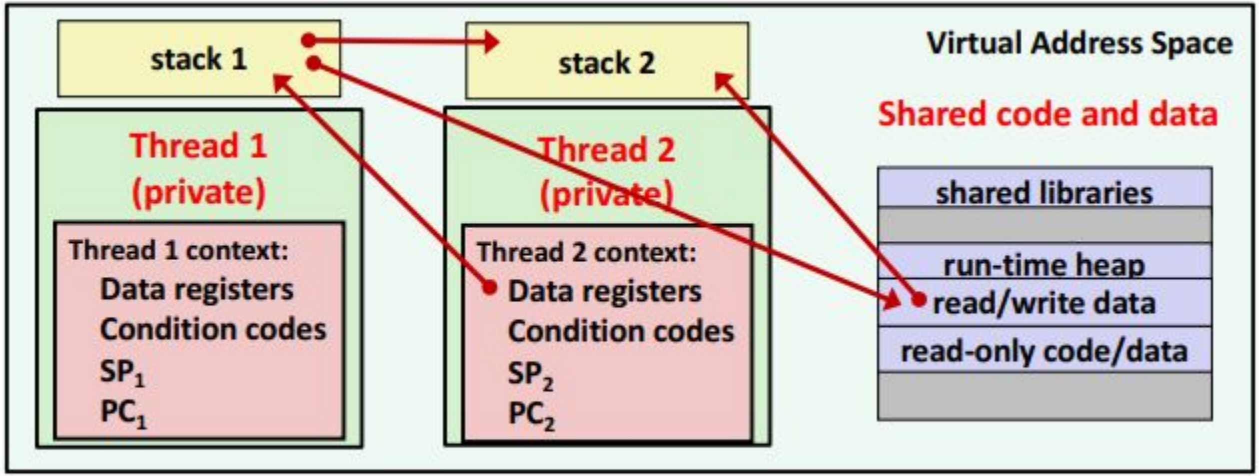
Threads Memory Model: Conceptual

- 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 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

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]
)

```
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

```
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

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`			

```
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`			
`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)varg
  static int cnt = 0;

  printf("[%ld]:  %s (cnt=%   n",
         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`			


```
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)

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			

```
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			
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

myid.p0?

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			

```
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			

```
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

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?

```
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	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)

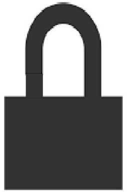
=

```
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;

}

Correctness (can be) Easy
Performance Hard

withdraw(…) {…}

deposit(…) {…}

addInterest(…) {…}

checkMinBalance(…) {…}

chargeFee(…) {…}

printBalance(…) {…}

Correctness (can be) Easy
Performance Hard

Simply add locks!

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

Example with lock

(thread4.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;
}
```

Example with lock

- Included a pthread_mutex_lock

```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;
}
```

Example with lock

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

```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;
}
```

Example with lock

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

```
mike:8$ gcc thread4.c -o thread4 -lpthread
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
^[[AFinal Counter value: 10000
```

```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;
}
```

Example with lock

- Also, don't forget to join!

```c
// Compile with:
//
// clang -lpthread thread4_fixed.c -o thread4_fixed
//
#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;
}
```

State is mutated in a deposit and withdraw

synchronized

1.) checkings = 25

2.) deposit(50)

3.) withdraw(50)

4.) checkings = **25**

not synchronized

1.) checkings = 25

2.) withdraw(50)

3.) deposit(50)

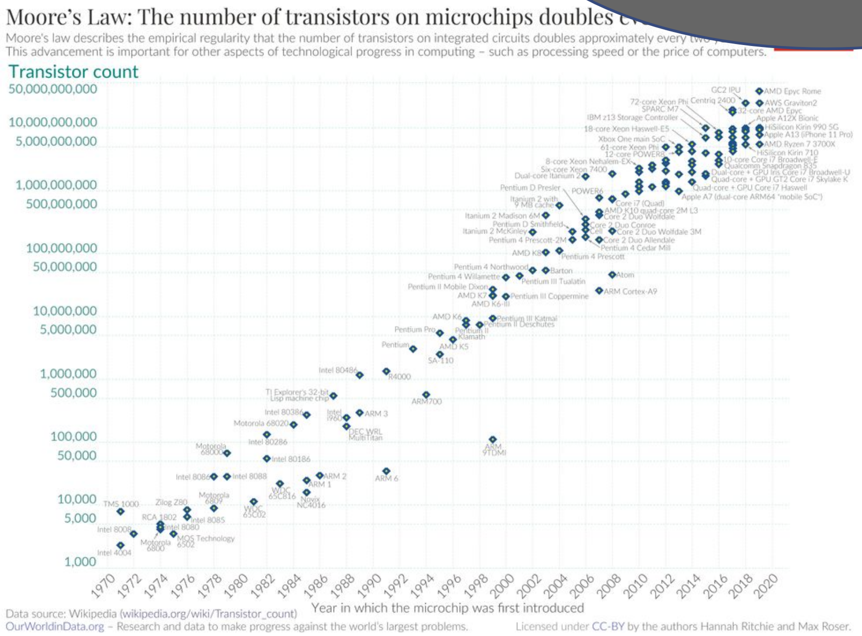
4.) checkings = 75? -25? 25?

Correctness (can be) Easy
Performance Hard

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?



Enforcing Mutual Exclusion

- Question: How can we guarantee we will not execute shared regions of code unsafely.
- Answer: We synchronize the execution of the threads
  - That is, we make sure regions of code have mutually exclusive access to each critical section
    - A critical section is a section of code that is shared and should only have one thread access it at a time.
- Classic solution:
  - Semaphores from the late Edsger Dijkstra
  - http://www.cs.toronto.edu/~demke/2227/S.14/Papers/p341-dijkstra.pdf

The Structure of the "THE"-Multiprogramming System

Edsger W. Dijkstra
Technological University, Eindhoven, The Netherlands

Semaphores

Binary Semaphores

- Mutex, which we have previously seen, is a special case of semaphore
  - Value is 0 or 1 (locked or unlocked)
- Recommended to use these over general semaphores when appropriate
  - Simpler abstraction
  - easier to read

General Semaphores

- Semaphore: non-negative global integer synchronization variable
  - Manipulated by P and V operations
- P(s) ("wait", "acquire", or "lock")
  - If s is nonzero, then decrement by 1 and return immediately
    - Test and decrement operations occur atomically (indivisibly)
  - If s is zero, then suspend thread until s becomes nonzero and the thread is restarted by a V operation
  - After restarting, the P operation decrements s and returns control to the caller
- V(s) ("unlock")
  - Increment s by 1
    - Increment operation occurs atomically
  - If there are any threads blocked in a P operation waiting for s to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing.
- Semaphore invariant: (s >= 0)

Semaphores continued

- OS Kernel guarantees code between brackets [] is guaranteed to execute indivisibly
  - Only one P(lock) or V(unlock) operation at a time can modify s
  - When while loop terminates, only P(lock) can decrement s.

P(s): [`while (s == 0) wait(); s--;`]

- Dutch for "Proberen" (test)

V(s): [`s++;`]

- Dutch for "Verhogen" (increment)

C semaphore programming example

- API
  - #include <semaphore>
  - int sem_init(sem_t *s, 0, unsigned int val)
  - int sem_wait(sem_t *s);
  - int sem_post(sem_t *s);

- Programming example
  - http://greenteapress.com/thinkos/html/thinkos012.html

Using semaphores for mutual exclusion

- Basic Idea:
  - Associate a unique semaphore *mutex*, initially 1, with each shared variable
    - (i.e. 1 spot open for a thread to enter)
  - Surround corresponding critical sections with P(mutex) and V(mutex) operations
- Terminology
  - Binary semaphore: Semaphore whose value is always 0 or 1
  - Mutex: Binary semaphore used for mutual exclusion
    - 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.

Pros and Cons of Thread-Based Designs

- Pros
  - Easy to share data structures between threads
    - e.g. logging information, file cache, etc.
  - Threads are more efficient than processes
- Cons
  - Unintentional sharing can introduce subtle and hard-to-reproduce errors
  - The ease with which data can be shared is both the greatest strength and greatest weakness of threads
  - Hard to know which data is being shared and what is private
  - Hard to find errors by testing
    - Often data races do not always show up!
      - (The probability is not zero!)

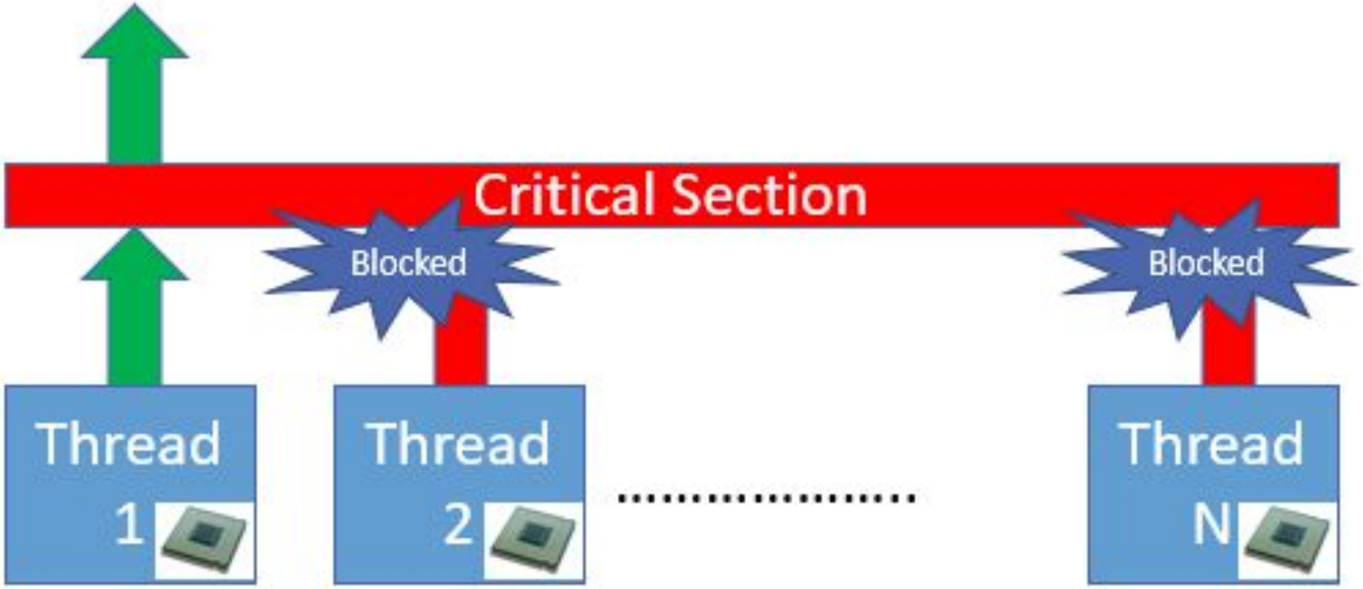
Summary of Synchronization

- 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
- Semaphores are a fundamental mechanism for enforcing mutual exclusion
  - Use MUTEX when possible

Concurrency Continued

Critical Section

- Code protected between a lock or semaphore



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.

Lack of Thread Safety

Datarace

Deadlock

Lack of Thread Safety

```c
// Compile with:
//
// clang -lpthread race.c -o race
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter =0;

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

int main(){
        // Store our Pthread ID
        pthread_t tid[NTHREADS];
        // Create and execute the thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tid[i], NULL, thread, NULL);
        }
        // Wait in 'main' thread until thread executes
        for(int i =0; i < NTHREADS; ++i){
                pthread_join(tid[i],NULL);
        }

        printf("Final value of counter %d\n",counter);

        // end program
        return 0;
}
```

```c
// Compile with:
//
// clang -lpthread deadlock.c -o deadlock
//
#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;
        // Lock is never released!! deadlock!
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tid[NTHREADS];
        // Create and execute the thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tid[i], NULL, thread, NULL);
        }
        // Wait in 'main' thread until thread executes
        for(int i =0; i < NTHREADS; ++i){
                pthread_join(tid[i],NULL);
        }

        printf("Final value of counter %d\n",counter);

        // end program
        return 0;
}
```

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.

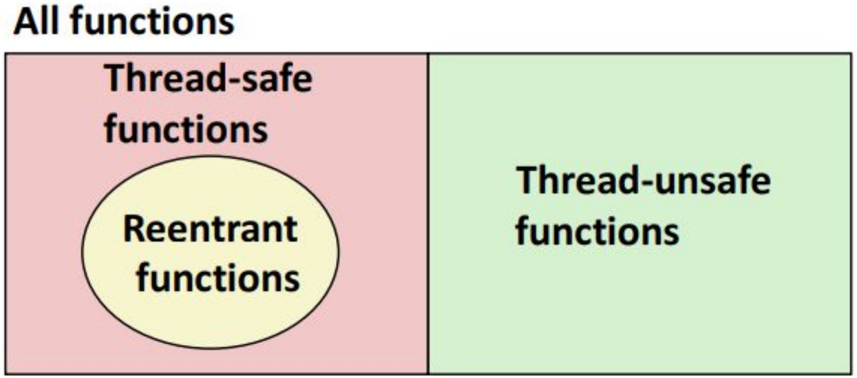
```
char *lc_itoa(int x, char *dest)
{
    P(&mutex);
    strcpy(dest, itoa(x));
    V(&mutex);
    return dest;
}
```

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

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! Uh-oh, should we rewrite our memory allocators for a future homework??

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 with the following exceptions

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

Semaphore Example

- Sometimes you may want to allow more than one thread through at once.
  - This is known as barrier synchronization
  - Here is an example of barrier synchronization using a semaphore to allow 3 threads to run simultaneously

```c
// gcc -lpthread semaphore.c -o semaphore
//
// Barrier Synchronization example
//
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h> // new library!

#define NTHREADS 9

sem_t barrier;

void *thread(void *vargp){
        // Barrier for threads to enter
        sem_wait(&barrier); // Wait and post are our lock/unlock equivalents
                printf("Hello from a thread\n");
                sleep(1); // Sleep is used to simulate some amount of work
        sem_post(&barrier);

        return NULL;
}

int main(){
        pthread_t tids[NTHREADS];
        // Initialize a barrier which allows 3 threads in
        sem_init(&barrier,0,3);
        // Create our threads
        int i;
        for(i =0; i < NTHREADS; ++i){
                pthread_create(&tids[i],NULL,thread,NULL);
        }
        // Join threads
        for(i =0; i < NTHREADS; ++i){
                pthread_join(tids[i],NULL);
        }
        // Destroy our semaphore
        sem_destroy(&barrier);
}
```

Other common concurrency patterns

- Signaling
- Producer-Consumer
- Readers-Writers

Signaling

Signaling

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

Thread A

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

Thread B

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

Signaling - c example

Thread A

```
statement A1
sem.signal() // sem_post
```

```c
// gcc -lpthread semaphore2.c -o semaphore2
// Signal example
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h> // new library!

sem_t sem;

void *threadA(void *vargp){
        printf("Hello from thread A\n");
        sem_post(&sem);
        return NULL;
}

void *threadB(void *vargp){
        sem_wait(&sem);
        printf("Hello from thread B\n");
        return NULL;
}

int main(){

        pthread_t tids[2];
        // Initialize a binary semaphore
        sem_init(&sem,0,1);
        // Create our threads
        pthread_create(&tids[0],NULL,threadA,NULL);
        pthread_create(&tids[1],NULL,threadB,NULL);
        // Join threads
        pthread_join(tids[0],NULL);
        pthread_join(tids[1],NULL);
        // Destroy our semaphore
        sem_destroy(&sem);
}
```

More Examples or End