CS 471 Operating Systems

Yue Cheng

George Mason University
Spring 2019

Review: Threads

Threads

- **Processes** vs. **threads**
  - Parent and child processes do not share address space
  - Inter-process communication w/ message passing or shared memory

  - Threads created by one process share address space, open files, global variables, etc.
  - Much cheaper and more flexible inter-thread communication and cooperation

A Simple Example Using Pthread

```
1   #include <stdio.h>
2   #include <assert.h>
3   #include <pthread.h>
4
5   void *mythread(void *arg) {
6       printf("%s\n", (char *) arg);
7       return NULL;
8   }
9
10  int
11  main(int argc, char *argv[]) {
12      pthread_t p1, p2;
13      int rc;
14      printf("main: begin\n");
15      rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16      rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17      // join waits for the threads to finish
18      rc = pthread_join(p1, NULL); assert(rc == 0);
19      rc = pthread_join(p2, NULL); assert(rc == 0);
20      printf("main: end\n");
21      return 0;
22  }
```

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

What would a 3rd thread trace look like?

Synchronization

- Race Conditions
- The Critical Section Problem
- Synchronization Hardware
- Semaphores

Threaded Counting Example

```c
#include <stdio.h>
#include "common.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char*) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

git clone https://github.com/remzi-arpacidusseau/ostep-code.git

Back-to-Back Runs

Run 1…
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 10706438)

Run 2…
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 11852529)

What Exactly Happened??

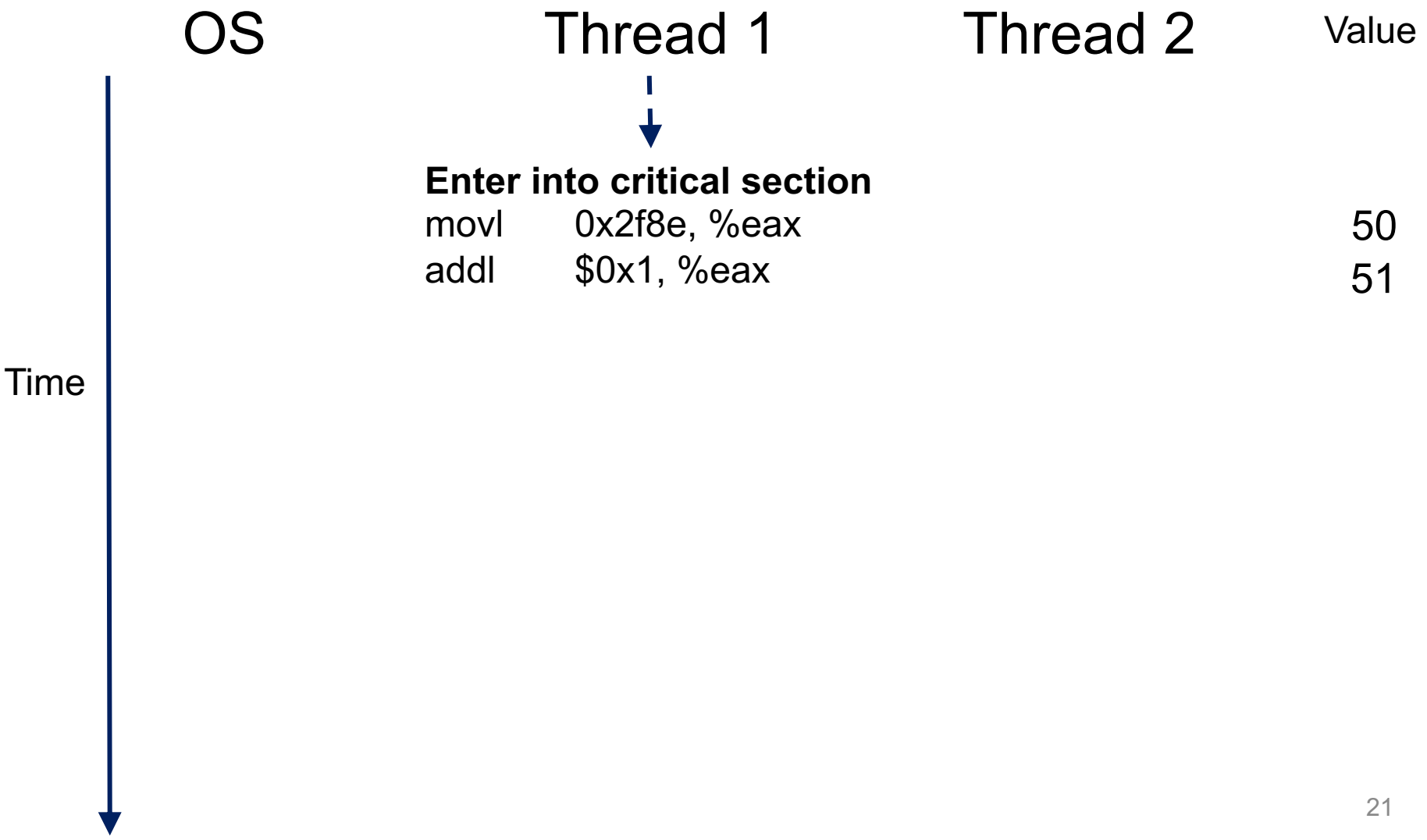
What Exactly Happened??

% otool -t -v thread_rc [Mac OS X]
% objdump -d thread_rc [Linux]

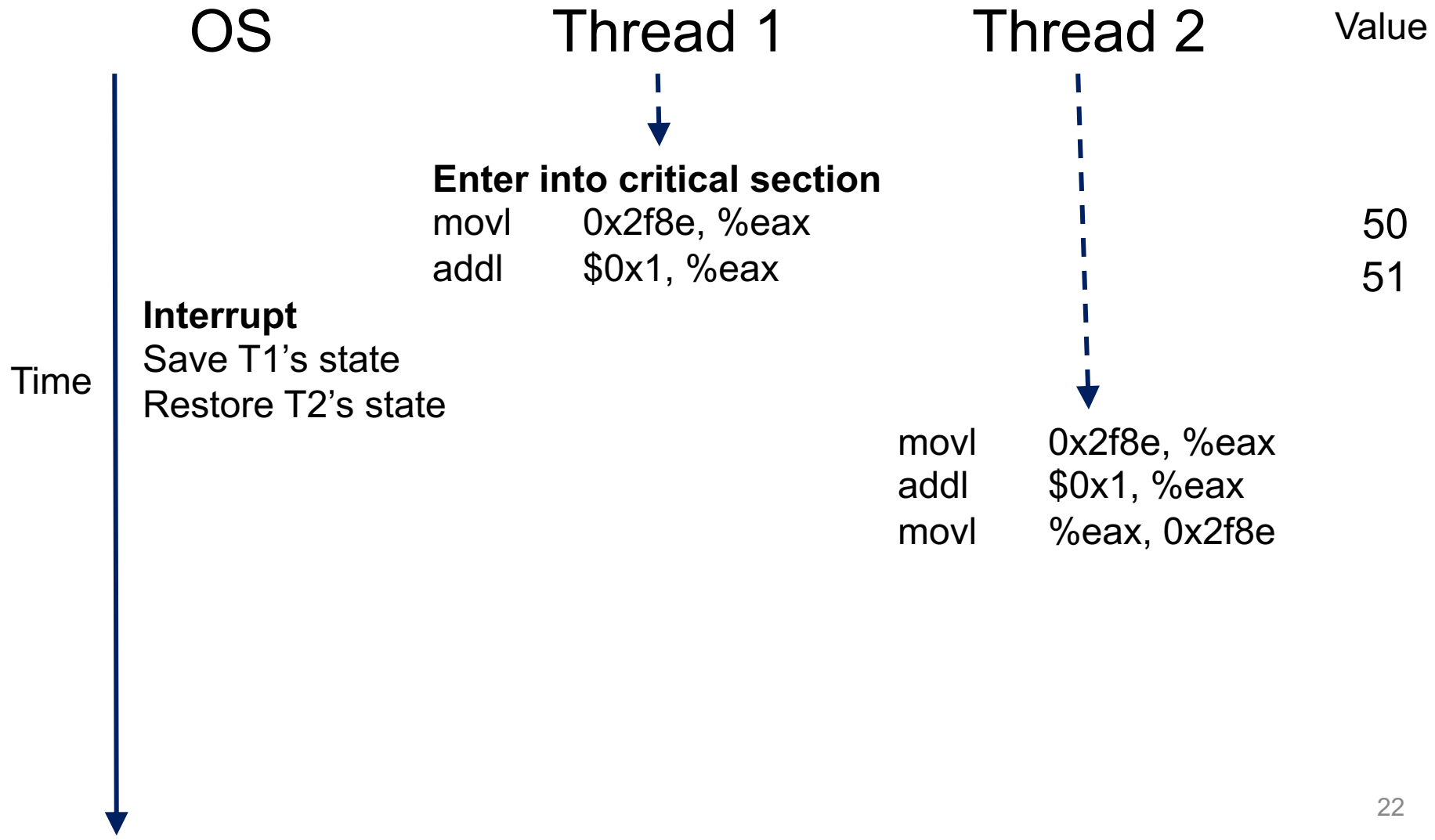
```
...
0000000100000d52    movl  0x2f8e %eax
0000000100000d58    addl  $0x1,  %eax
0000000100000d5b    movl  %eax,  0x2f8e
...
```

```
counter = counter + 1;
```

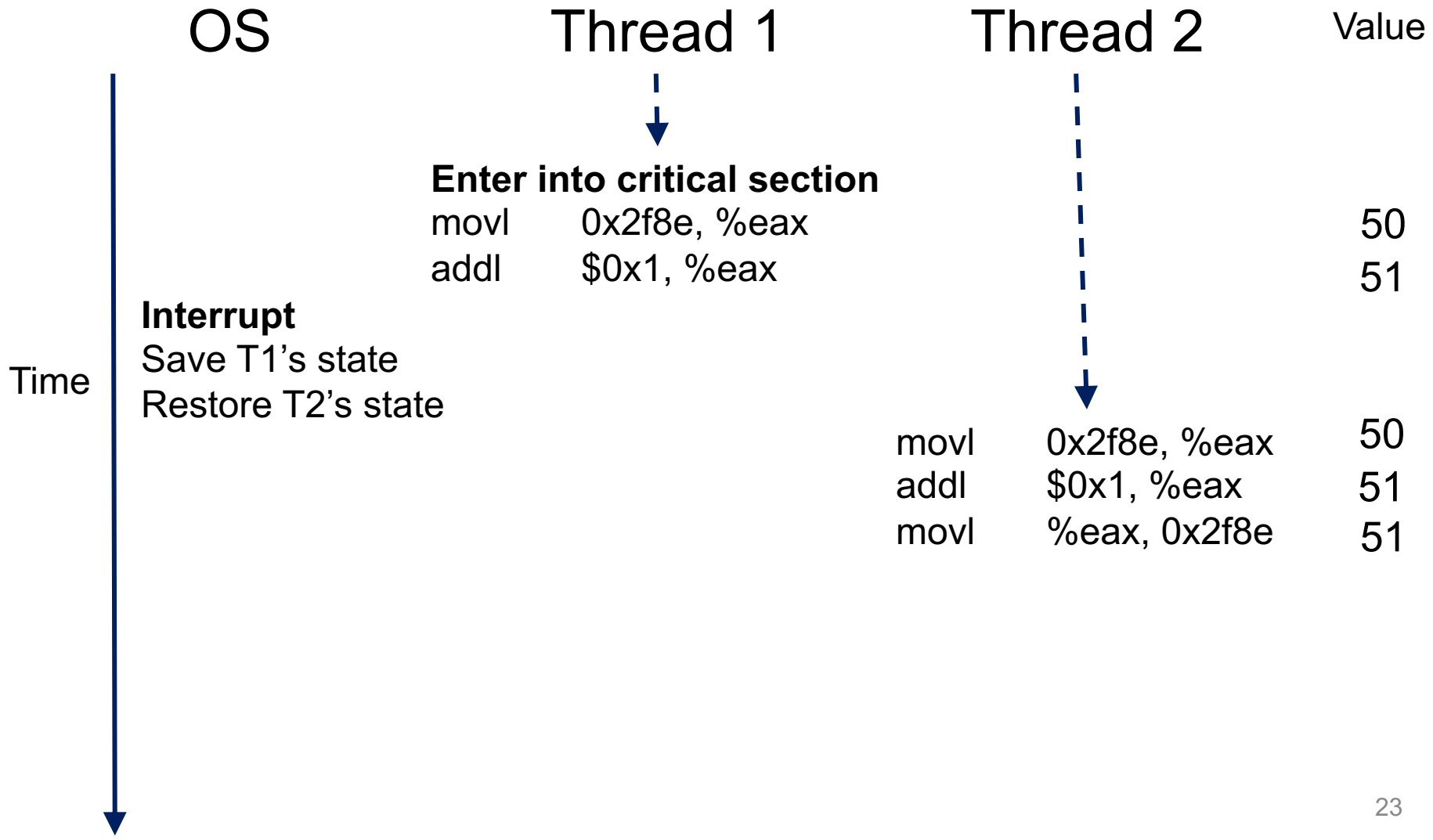
Concurrent Access to The Same Memory Address



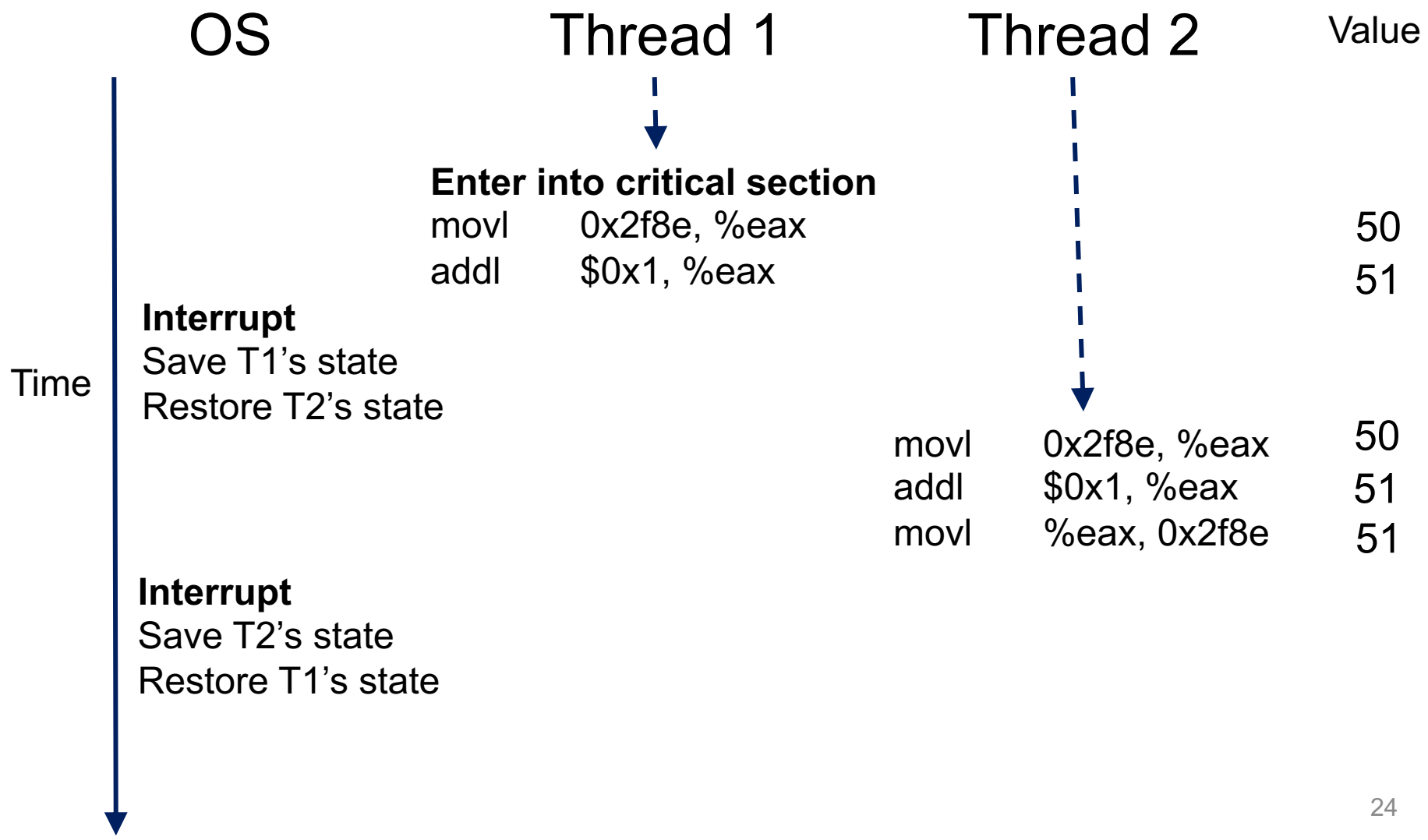
Concurrent Access to The Same Memory Address



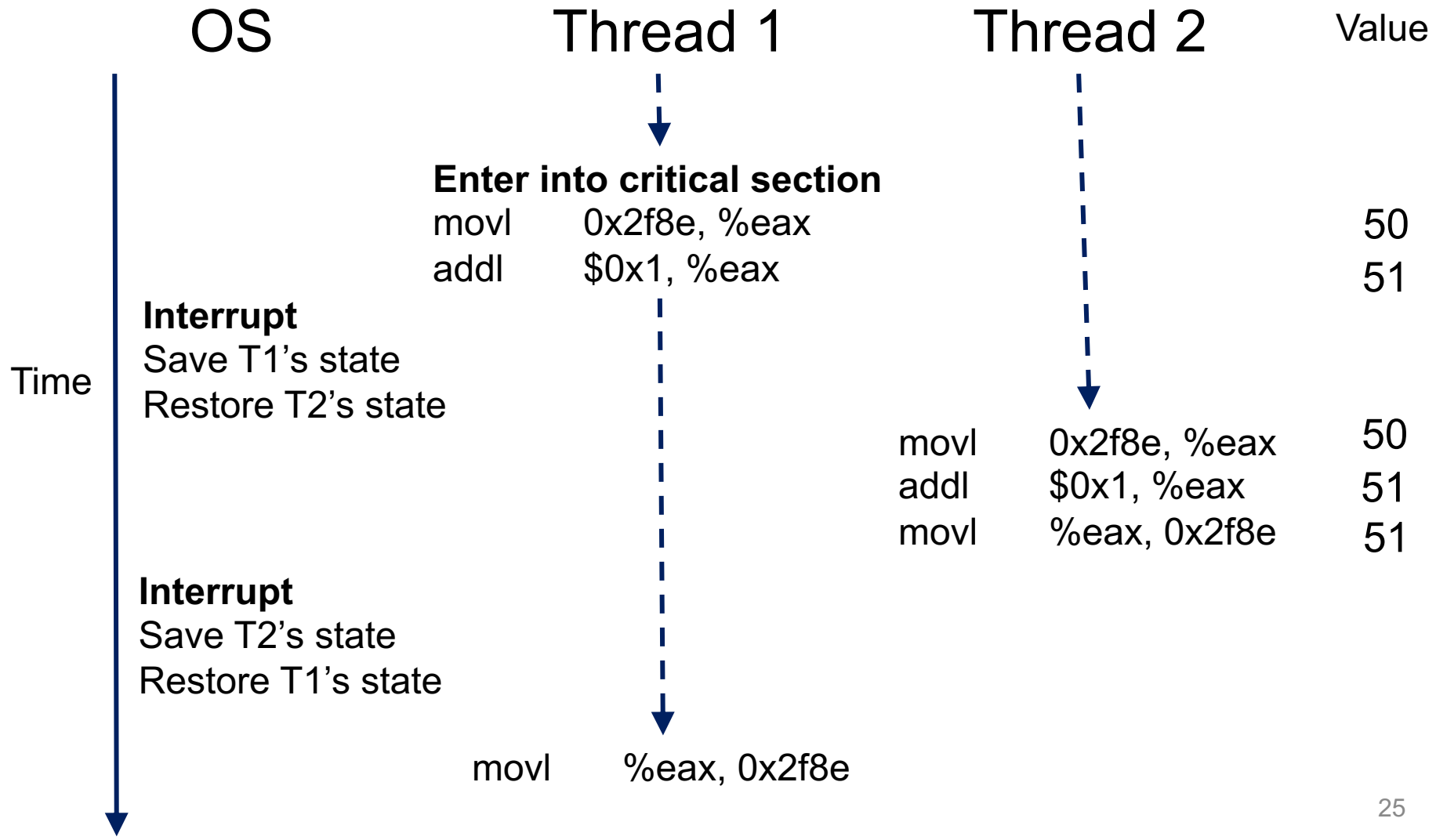
Concurrent Access to The Same Memory Address



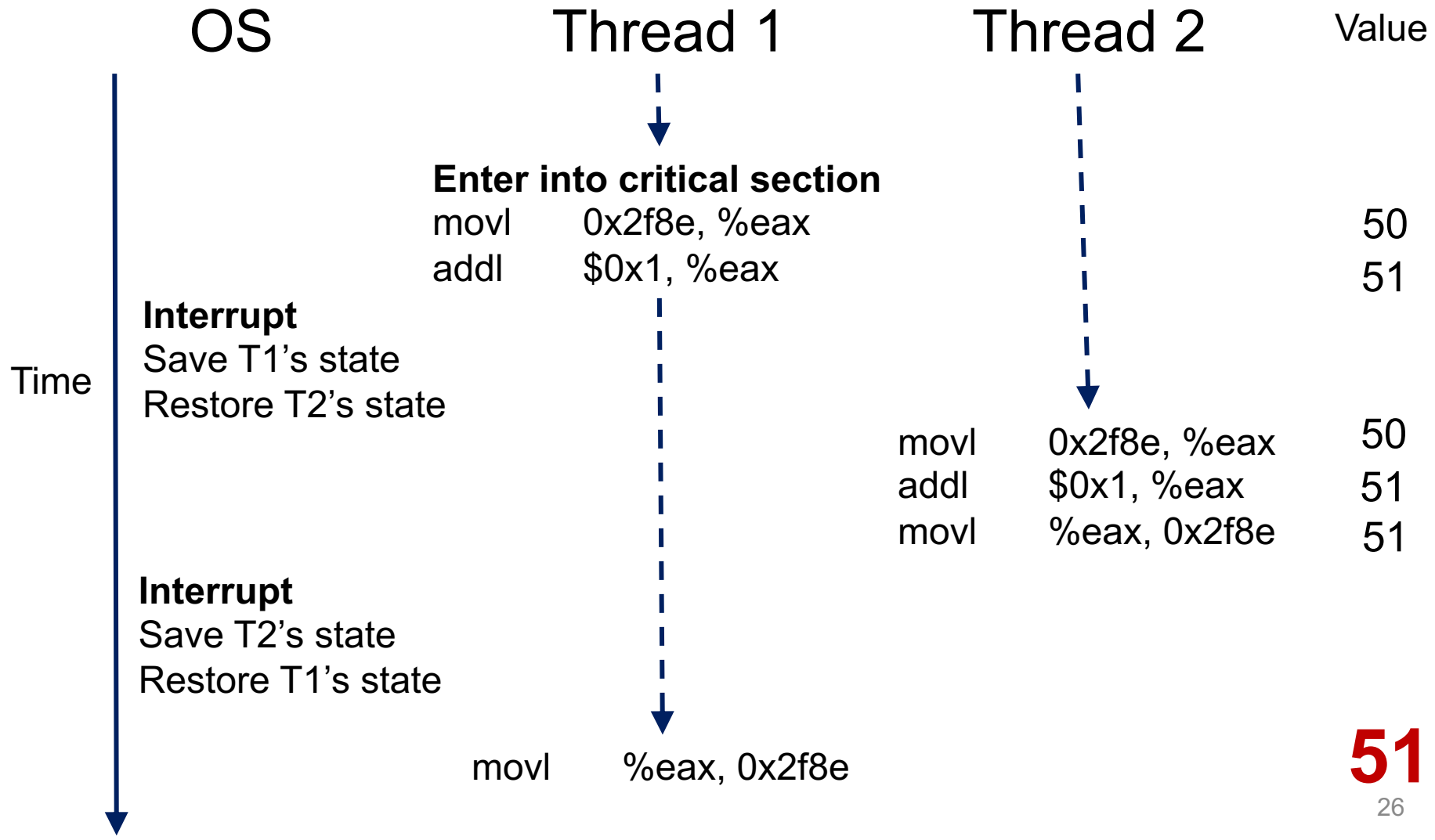
Concurrent Access to The Same Memory Address



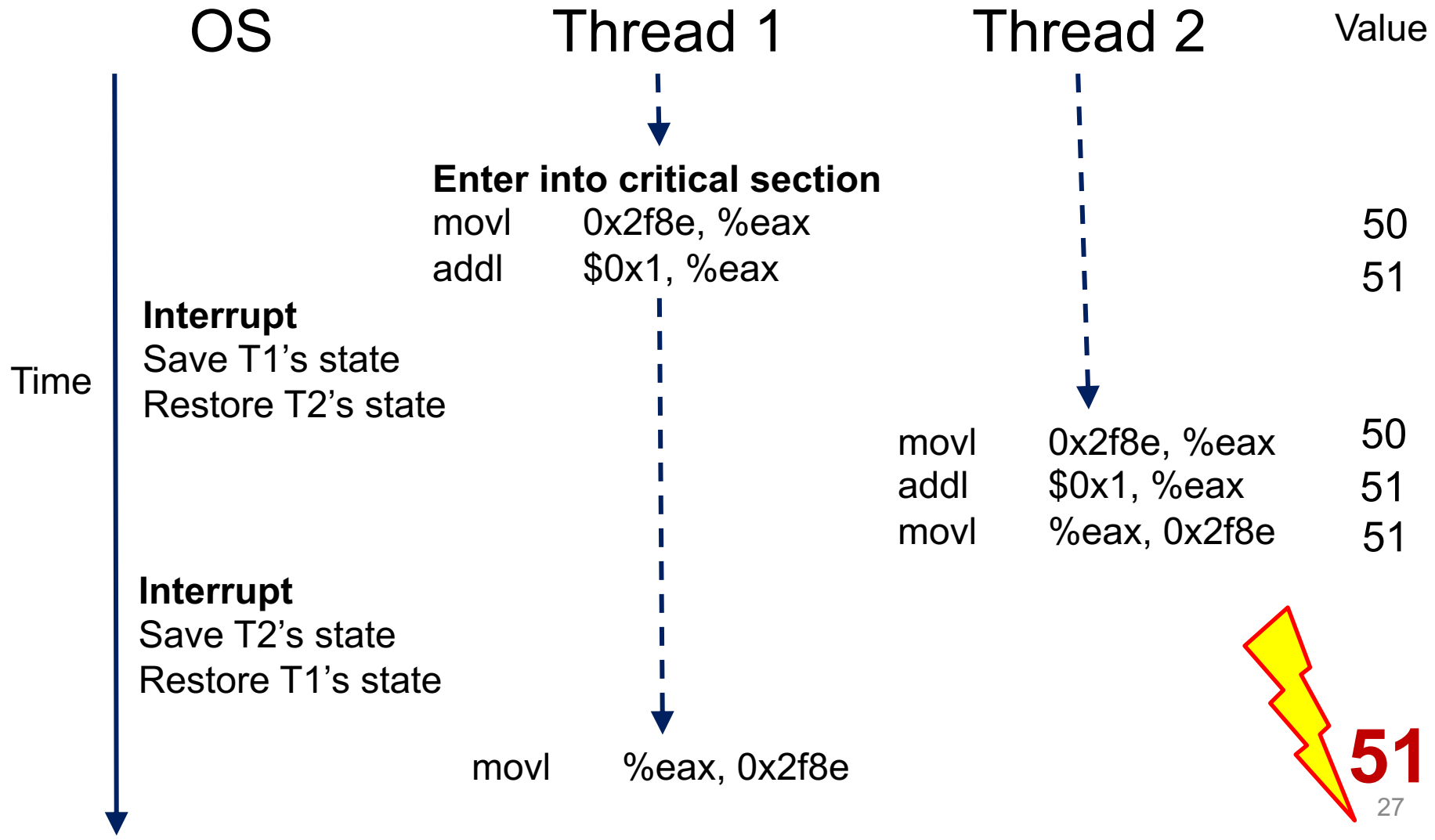
Concurrent Access to The Same Memory Address



Concurrent Access to The Same Memory Address



Concurrent Access to The Same Memory Address



Race Conditions

- Observe: In a **time-shared** system, **the exact instruction execution order** cannot be predicted
  - Deterministic vs. **Non-deterministic**

- Any possible orders can happen, which result in different output across runs

Race Conditions

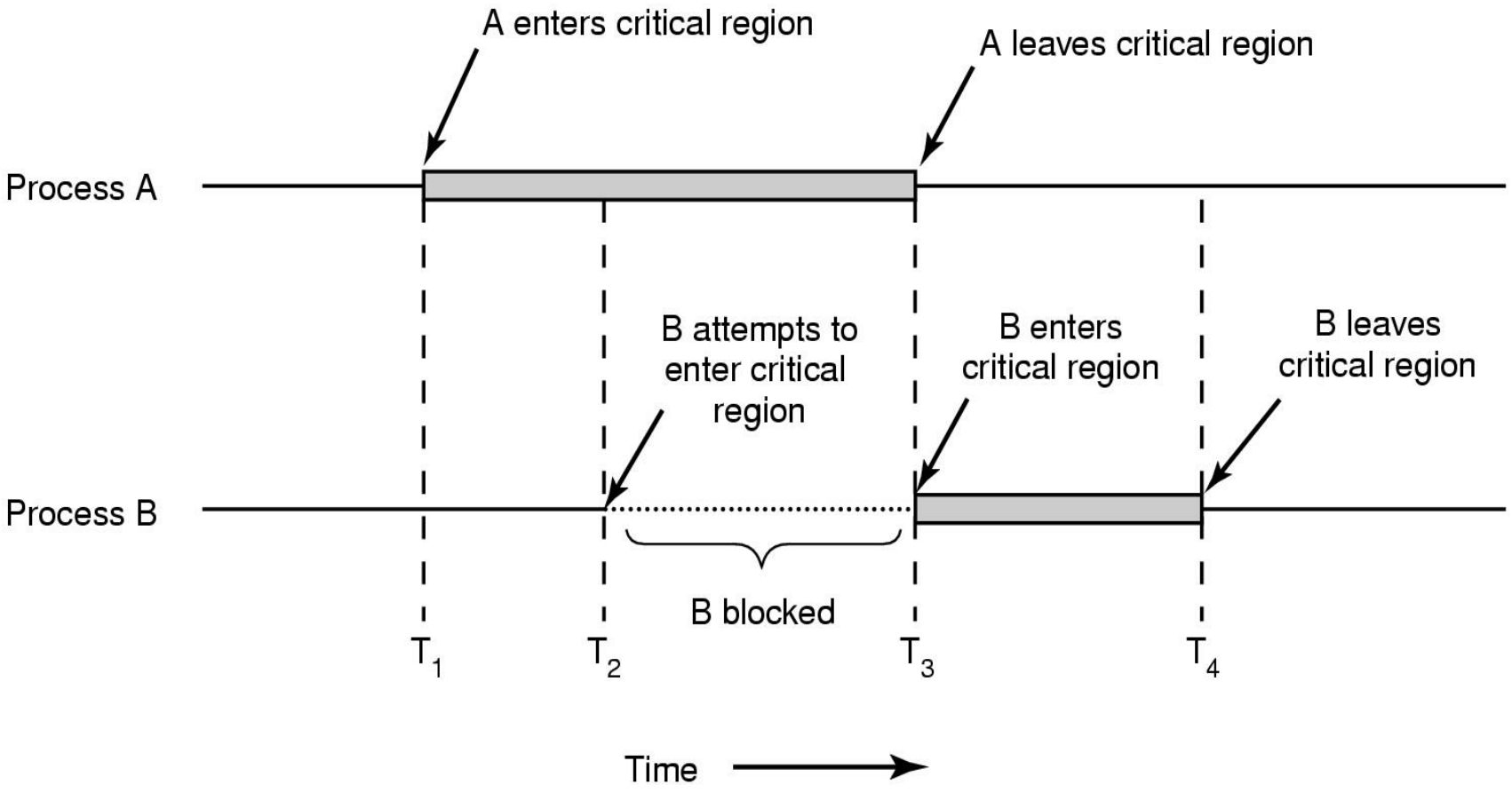
- Situations like this, where multiple processes are writing or reading some shared data and the final result depends on who runs precisely when, are called **race conditions**
  - A serious problem for any concurrent system using shared variables

- Programmers must make sure that some **high-level** code sections are executed **atomically**
  - Atomic operation: It completes in its entirety without worrying about interruption by any other potentially conflict-causing process

The Critical-Section Problem

- *N* processes/threads all competing to access the shared data
- Each process/thread has a code segment, called critical section (critical region), in which the shared data is accessed
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in that critical section
- The execution of the critical sections by the processes must be mutually exclusive in time

Mutual Exclusion



Solving Critical-Section Problem

Any solution to the problem must satisfy **four conditions**!

Mutual Exclusion:

No two processes may be simultaneously inside the same critical section

Bounded Waiting:

No process should have to wait forever to enter a critical section

Progress:

No process executing a code segment unrelated to a given critical section can block another process trying to enter the same critical section

Arbitrary Speed:

No assumption can be made about the relative speed of different processes (though all processes have a non-zero speed)

Using **Lock** to Protect Shared Data

- Suppose that two threads A and B have access to a shared variable "`balance`"

Thread A:

```
balance = balance + 1
```

Thread B:

```
balance = balance + 1
```

```
1   lock_t mutex; // some globally-allocated lock 'mutex'
2   ...
3   lock(&mutex);
4   balance = balance + 1;
5   unlock(&mutex);
```

Locks

- A lock is a **variable**

- Two states
  - Available or free
  - Locked or held

- `lock()`: tries to acquire the lock
- `unlock()`: releases the lock that has been acquired by caller

Building a Lock

- Needs help from hardware + OS
- A number of hardware primitives to support a lock
- Goals of a lock
  - Basic task: **Mutual exclusion**
  - Fairness
  - Performance

Getting Help from the Hardware

- One solution supported by hardware may be to use interrupt capability

```
do {
    lock()
       critical section;
    unlock()
       remainder section;
} while (1);
```

```
1   void lock() {
2       DisableInterrupts();
3   }
4   void unlock() {
5       EnableInterrupts();
6   }
```

Getting Help from the Hardware

- One solution supported by hardware may be to use interrupt capability

```
do {
    lock()
       critical section;
    unlock()
       remainder section;
} while (1);
```

```
1   void lock() {
2       DisableInterrupts();
3   }
4   void unlock() {
5       EnableInterrupts();
6   }
```

Are we done??

First Attempt: A Simple Flag

- How about just using `Loads/Stores` instructions?

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10          ; // spin-wait (do nothing)
11      mutex->flag = 1;          // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

First Attempt: A Simple Flag

- How about just using `Loads/Stores` instructions?

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)   // TEST the flag
10          ; // spin-wait (do nothing)
11      mutex->flag = 1;           // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

A spin lock

First Attempt: A Simple Flag

- How about just using `Loads/Stores` instructions?

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)   // TEST the flag
10          ; // spin-wait (do nothing)
11      mutex->flag = 1;           // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

A spin lock

What's the problem?

First Attempt: A Simple Flag

Flag is 0 initially

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	

First Attempt: A Simple Flag

Flag is 0 initially

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)

First Attempt: A Simple Flag

Flag is set to 1 by T2

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1

First Attempt: A Simple Flag

Flag is set to 1 again! Two threads both in Critical Section

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

First Attempt: A Simple Flag

Flag is set to 1 again! Two threads both in Critical Section

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Reason: No mutual exclusion!

Synchronization Hardware

- Many machines provide special hardware instructions to help achieve mutual exclusion
- The `TestAndSet(TAS)` instruction tests and modifies the content of a memory word **atomically**
- TAS returns old value pointed to by old_ptr and updates said value to new

```
1  int TestAndSet(int *old_ptr, int new) {
2      int old = *old_ptr; // fetch old value at old_ptr
3      *old_ptr = new;     // store 'new' into old_ptr
4      return old;         // return the old value
5  }
```

Operations performed atomically!

Mutual Exclusion with TAS

- Initially, lock's flag set to **0**

```
1   typedef struct __lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *lock) {
6       // 0 indicates that lock is available, 1 that it is held
7       lock->flag = 0;
8   }
9
10  void lock(lock_t *lock) {
11      while (TestAndSet(&lock->flag, 1) == 1)
12          ; // spin-wait (do nothing)
13  }
14
15  void unlock(lock_t *lock) {
16      lock->flag = 0;
17  }
```

→ **A correct spin lock**

Busy Waiting and Spin Locks

- This approach is based on **busy waiting**
  - If the critical section is being used, waiting processes loop continuously at the entry point
- A binary "lock" variable that uses busy waiting is called a **spin lock**
  - Processes that find the lock unavailable "spin" at the entry
- It actually works (mutual exclusion)
- Disadvantages?
  - Fairness?
  - Performance?

Busy Waiting and Spin Locks

- This approach is based on **busy waiting**
  - If the critical section is being used, waiting processes loop continuously at the entry point
- A binary "lock" variable that uses busy waiting is called a **spin lock**
  - Processes that find the lock unavailable "spin" at the entry
- It actually works (mutual exclusion)
- Disadvantages?
  - Fairness? (A: No. Heavy contention may cause starvation)
  - Performance? (A: Busy waiting wastes CPU cycles)

A Simple Approach: Just Yield ~~(Win)~~!

- When you are going to Spin, just **give up** the CPU to another process/thread

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

One Step Further: Semaphores

- Introduced by **E. W. Dijkstra**
- Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied
- Two operations are defined on a semaphore variable s:

  `sem_wait(s)` (also called `P(s)` or `down(s)`)

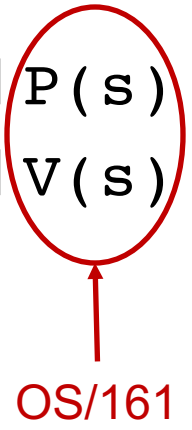
  `sem_post(s)` (also called `V(s)` or `up(s)`)

One Step Further: Semaphores

- Introduced by **E. W. Dijkstra**
- Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied
- Two operations are defined on a semaphore variable s:

  `sem_wait(s)` (also called `P(s)` or `down(s)`)

  `sem_post(s)` (also called `V(s)` or `up(s)`)



Semaphore Operations

- Conceptually, a semaphore has an integer value. This value is greater than or equal to 0

- ```
  sem_wait(s):
    s.value-- ;  /*  Executed atomically */
    /* wait/block if s.value < 0 (or negative) */
  ```

- A process/thread executing the wait operation on a semaphore with value < 0 being blocked until the semaphore's value becomes greater than 0
  - **No busy waiting**

- ```
  sem_post(s):
    s.value++;  /* Executed atomically */
    /* if one or more process/thread waiting, wake one */
  ```

Semaphore Operations (cont.)

- If multiple processes/threads are blocked on the same semaphore '`s`', only one of them will be awakened when another process performs post(s) operation

- Who will have higher priority?

Semaphore Operations (cont.)

- If multiple processes/threads are blocked on the same semaphore '`s`', only one of them will be awakened when another process performs post(s) operation

- Who will have higher priority?
  - A: FIFO or whatever queuing strategy

Attacking Critical Section Problem with Semaphores

- Declare and define a semaphore:

```
sem_t s;
sem_init(&s, 0, 1);   /* initially s = 1 */
```

Binary semaphore, which is a lock

- Routine of Thread 0 & 1:

```
do {
    sem_wait(s);
      critical section
    sem_post(s);
      remainder section
} while (1);
```

Attacking Critical Section Problem with Semaphores

- **Single** thread using a **binary** semaphore

Value of Semaphore	Thread 0	Thread 1
1		

Attacking Critical Section Problem with Semaphores

- **Single** thread using a **binary** semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	

Attacking Critical Section Problem with Semaphores

- **Single** thread using a **binary** semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	

Attacking Critical Section Problem with Semaphores

- **Single** thread using a **binary** semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	
1	`sem_post()` returns	

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect: end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect: end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running

Attacking Critical Section Problem with Semaphores

- **Two** threads using a **binary** semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect: end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	`sem_wait()` returns	Running
0		Ready	`(crit sect)`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`sem_post()` returns	Running

Worksheet