# Lecture 13: Concurrency: Threads and Locks

Operating Systems

**Content taken from:** https://pages.cs.wisc.edu/~remzi/OSTEP/

# Till now

- CPU virtualization
  - Mechanism
  - Policy
- Memory virtualization
  - Mechanism
    - Segmentation
    - Paging
    - Swapping
  - Policy
    - LRU
    - Random

# Concurrency

- Till now, we assumed that each process has a single point of execution
  - A single PC where instructions are being fetched from and executed
- Now, we introduce a new abstraction: **thread**
- Each process can have multiple threads i.e. multiple points of execution
  - Multiple PCs
  - Such a process is called **multi-threaded** process

# What is a thread?

- Each thread is nothing but a separate process
- All the threads belonging to the same process share the same address space
  - Code, Data, Heap is shared but not the stack
- Each thread has a its own PC, private set of registers and stack
- Context switch between threads is similar to context switch between processes
  - OS needs to save thread’s state to **thread control block (TCB)**
  - OS does not need to switch the page table. Why?

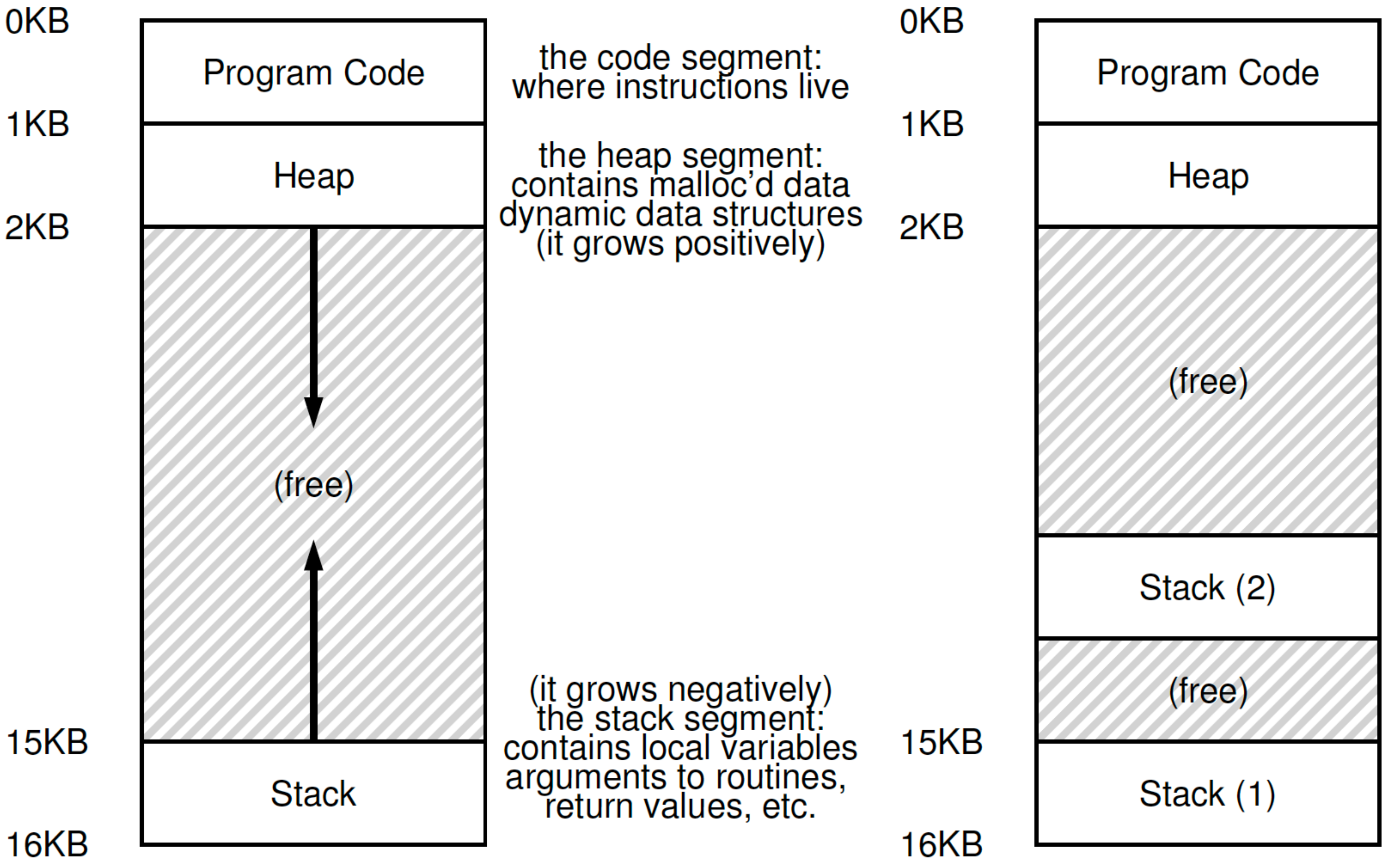


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

# Why use Threads?

- **Parallelism**
  - If you are running a program on a multi-processor system, you may want to utilize all the processors simultaneously
  - E.g. Adding two large arrays
- **Avoiding blocking**
  - Threading enables overlap of I/O with other activities ***within*** a single program
  - E.g. While a thread is doing I/O, another thread can do some computations or processing

# Example: Thread Creation

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    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: end\n");
    return 0;
}
```

Figure 26.2: **Simple Thread Creation Code (`t0.c`)**

| 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" | | |

Figure 26.3: **Thread Trace (1)**

| **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” | | |

Figure 26.4: **Thread Trace (2)**

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

Figure 26.5: **Thread Trace (3)**

# Example: Shared Data

```c
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.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;
}
```

```c
// 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;
}
```

Figure 26.6: **Sharing Data: Uh Oh (`t1.c`)**

# Indeterminate Result

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

# Uncontrolled Scheduling

- In the previous example, assembly code which updates the counter looks like the following:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

- The above piece of code is called as **critical section**
- A critical section is a piece of code that accesses a shared variable and must not be concurrently executed by more than one thread.

# Race Condition

| OS | Thread 1 | Thread 2 | PC | eax | counter |
|---|---|---|---|---|---|
| | | | (after instruction) | | |
| | *before critical section* | | 100 | 0 | 50 |
| | `mov 8049a1c,%eax` | | 105 | **50** | 50 |
| | `add $0x1,%eax` | | 108 | **51** | 50 |
| **interrupt** | | | | | |
| *save T1* | | | | | |
| *restore T2* | | | 100 | 0 | 50 |
| | | `mov 8049a1c,%eax` | 105 | **50** | 50 |
| | | `add $0x1,%eax` | 108 | **51** | 50 |
| | | `mov %eax,8049a1c` | 113 | 51 | **51** |
| **interrupt** | | | | | |
| *save T2* | | | | | |
| *restore T1* | | | 108 | 51 | 51 |
| | `mov %eax,8049a1c` | | 113 | 51 | **51** |

Figure 26.7: **The Problem: Up Close and Personal**

# How to avoid race conditions?

- We want **mutual exclusion**
- If one thread is executing within the critical section, the others should be prevented from doing so.
- In other words, we should write multi-threaded code that accesses critical sections in a **synchronized** and **controlled** manner
  - This avoids race conditions
  - Results in deterministic program outputs

# Locks and Pthread Locks

- Programmers annotate source code with locks, putting them around critical sections, and thus ensure that any such critical section executes as if it were a **single atomic** instruction

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

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper; exits on failure
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

# Evaluating Locks

- **Correctness:** Does the lock ensure mutual exclusion?
- **Fairness:** Does each thread contending for the lock get a fair shot at acquiring it once it is free?
- **Performance:** How much time overhead does the lock add?

# How can we build a lock?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Figure 28.1: **First Attempt: A Simple Flag**

# Lock using a simple flag does not work

- Mutual Exclusion is not ensured

| 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!) | |

Figure 28.2: **Trace: No Mutual Exclusion**

- Performance overhead: **Spin-waiting**

# Building Working Spin Locks with Test-And-Set

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

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

# Evaluating Spin Locks

- Correctness?
- Fairness?
- Performance?