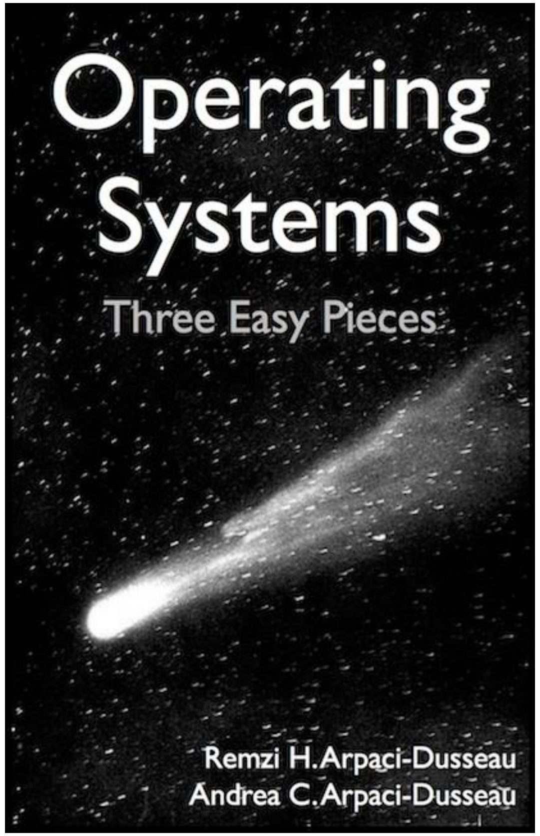
CMSC 125: Operating Systems

- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org

Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

26. Concurrency: An Introduction

Operating System: Three Easy Pieces

Thread

- A new abstraction for <u>a single running process</u>

- Multi-threaded program
  - A multi-threaded program has more than one path of execution
  - Multiple PCs (Program Counter)
  - They share the same address space

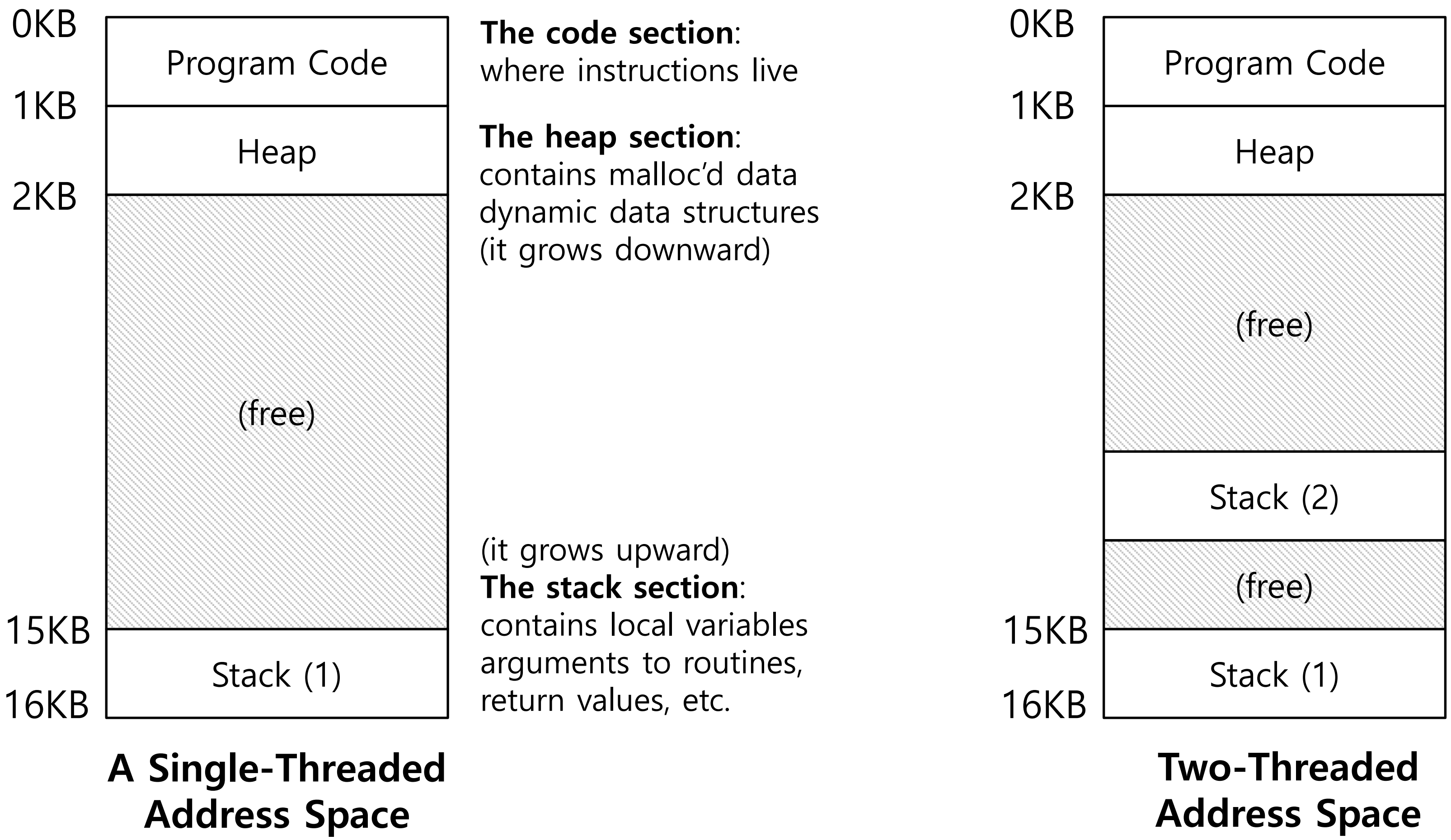
Context switch between threads

- Each thread has its own program counter and set of registers
  - One or more **thread control blocks(TCBs)** are needed to store the state of each thread

- When switching from running one (T1) to running the other (T2)
  - The register state of T1 be saved
  - The register state of T2 restored
  - The address space remains the same

The stack of the relevant thread

- There will be one stack per thread



Why Use Threads?

1. Parallelism
    - Speeds up computations
2. Avoid blocking a process’ progress due to slow I/O
    - Enables the overlap of I/O and computations within a process
    - Ex. Spell-checking using a background thread in a word processor

Thread Creation Example

```
#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;
}
```

Thread Creation Example: Possible Execution Path 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 Creation Example: Possible Execution Path 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 Creation Example: Possible Execution Path 3

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"		

Why So Many Possible Execution Paths?

- Depends on the thread scheduler!

Shared Data Example

- Further complicates things because data inconsistencies can happen

```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
// No, this is not how you would add 10,000,
// a counter, but it shows the problem nicel
//
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;
}
```

Critical Section

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

Shared Data (Cont..)

```
// 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;
}
```

Race Condition

- The result depends on the execution path
- Example with two threads
  - counter = counter + 1 (initial is 50)
  - We expect the result is 52. However,

Part of thread state

OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	**51**

Critical Section

- A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread because executing it might result to a race condition and thus to incorrect results
- Solution: Need to support **atomicity**(all instruction executions in the critical section will not be preempted) for critical section access (**mutual exclusion**)
  - Will need special hardware instructions and OS support

Locks

- Another abstraction such that a thread will not be able to execute in the critical section unless it holds a lock
- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

Condition Variables

- Some problems will require “waiting for another thread” – **Synchronization**
  - Ex. Threads for I/O processing
  - Other synchronization problems: Bounded-Buffer, Producer-Consumer, Readers-Writers, Dining Philosophers Problem