TCSS 422: OPERATING SYSTEMS

Three Easy Pieces:
Ch. 26: Concurrency Introduction

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

April 16, 2018 | TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Concurrency: Introduction – Ch. 26

April 16, 2018 | TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma | L6b.2

CHAPTER 26 - CONCURRENCY: AN INTRODUCTION

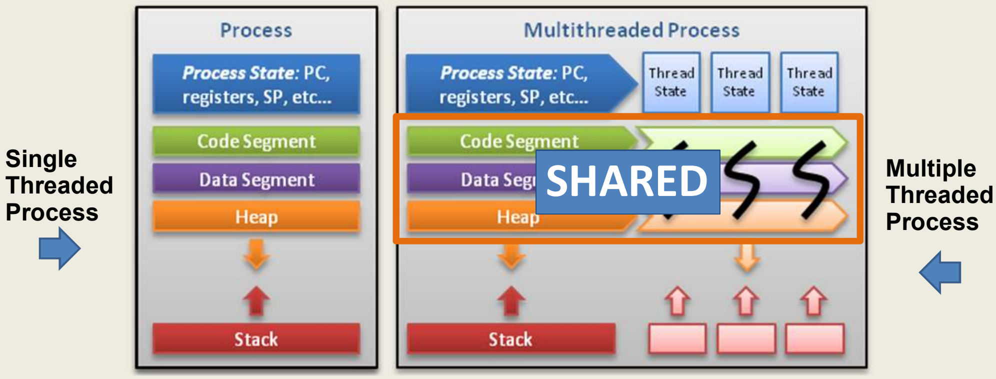
April 16, 2018 | TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma | L6b.3

OBJECTIVES

- Introduction to threads
- Race condition
- Critical section
- Thread API

April 16, 2018 | TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma | L6b.4

THREADS



©Alfred Park, http://randu.org/tutorials/threads

THREADS - 2

- Enables a single process (program) to have multiple "workers"
- Supports independent path(s) of execution within a program *with shared memory …*
- Each thread has its own Thread Control Block (TCB)
  - PC, registers, SP, and stack
- Code segment, memory, and heap are shared

PROCESS AND THREAD METADATA

- Thread Control Block vs. Process Control Block

Thread identification
Thread state
CPU information:
- Program counter
- Register contents

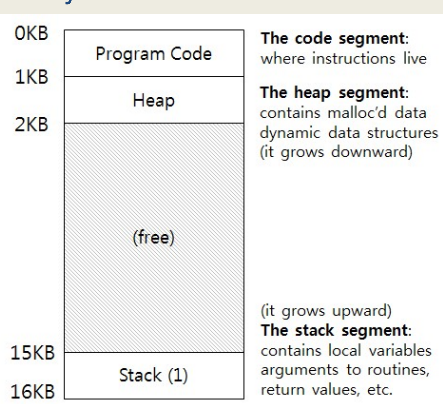
Thread priority
Pointer to process that created this thread
Pointers to all other threads created by this thread

Process identification
Process status
Process state:
- Process status word
- Register contents
- Main memory
- Resources
- Process priority

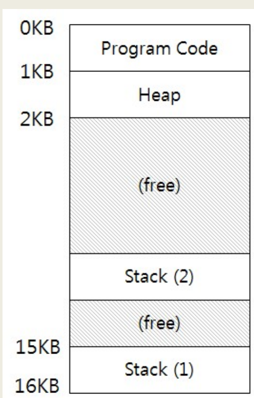
Accounting

SHARED ADDRESS SPACE

- Every thread has it's own stack / PC



A Single-Threaded Address Space



Two threaded Address Space

THREAD CREATION EXAMPLE

```
#include <stdio.h>
#include <assert.h>
#include <pthread.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");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```


POSSIBLE ORDERINGS OF EVENTS

int main()	Thread 1	Thread 2
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'		

POSSIBLE ORDERINGS OF EVENTS - 2

int main()	Thread 1	Thread 2
Starts running		
Prints 'main: begin'		
Creates Thread 1		
	Runs	
	Prints 'A'	
	Returns	
Creates Thread 2		
		Runs
		Prints 'B'
		Returns
Waits for T1	*Returns immediately*	
Waits for T2		*Returns immediately*
Prints 'main: end'		

April 16, 2018 | TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma | L6b.11

POSSIBLE ORDERINGS OF EVENTS - 3

int main()	Thread 1	Thread 2
Starts running		
Prints 'main: begin'		
Creates Thread 1		
Creates Thread 2		
Waits for T		
	Runs	
	Prints 'A'	
	Returns	
Waits for T2		*Immediately returns*
Prints 'main: end'		

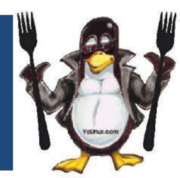
What if execution order of events in the program matters?

April 16, 2018 | TCSS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma | L6b.12

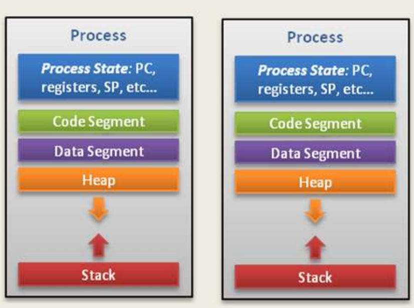
COUNTER EXAMPLE

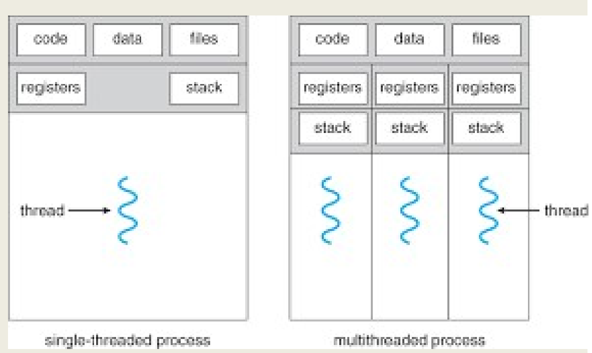
- Counter example
- A + B : ordering
- Counter: incrementing global variable by two threads

PROCESSES VS. THREADS



- What's the difference between forks and threads?
  - Forks: duplicate a process
  - Think of ***CLONING*** - There will be two identical processes at the end
  - Threads: no duplicate of code/heap, lightweight execution threads





RACE CONDITION

- What is happening with our counter?
  - When counter=50, consider code: counter = counter + 1
  - If synchronized, counter will = 52

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

- Code that accesses a shared variable must not be ***concurrently*** executed by more than one thread
- Multiple *active* threads inside a ***critical section*** produce a ***race condition***.
- ***Atomic execution*** (*all code executed as a unit*) must be ensured in ***critical*** sections
  - These sections must be ***mutually exclusive***

LOCKS

- To demonstrate how critical section(s) can be executed "atomically-*as a unit*" Chapter 27 & beyond introduce locks

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

Critical section

- Counter example revisited

April 16, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L6b.17

QUESTIONS