Operating System (OS)
CS232

Concurrency: Introduction to Threads and Threading API

Dr. Muhammad Saeed &
Dr. Muhammad Mobeen Movania

Outlines

- Multitasking vs Multithreading
  - What is a thread?
  - Threads vs Processes
- Why Multithreading
  - Why use threads?
- Example Code
- Issues that need to be address in Multithreading
  - Accessing shared global variable
  - Why output is different?
  - Race condition and atomicity
- Thread APIs
- Summary

Multitasking vs Multithreading

What is a thread?

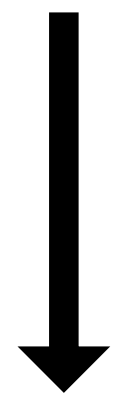
- Thread is an **independent path of execution**
- In a multi-threaded program, each thread has its own PC, register (context), stack, but it **shares the process’s address space**

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("hello world\n");

   return 0;
}
```

main thread



Threads vs Processes

- Threads are like processes:
  - They can execute independently
  - Each thread has a separate PC and set of registers (context) while executing
  - If two threads T1 & T2 are running on a single processor then:
    - Only one can run at any given time
    - Switching from on thread to other requires a context switch
  - Each thread has its own stack (thread local storage)

Threads vs Processes

- Threads are different than processes:
  - Threads share the same address space
  - Context switch b/w threads results in switching of stacks but not of page tables!
  - A single process can have one or more threads
  - Thread states are saved in Thread Control Blocks (TCBs) instead of PCBs

Single vs Multi-threaded Address Spaces

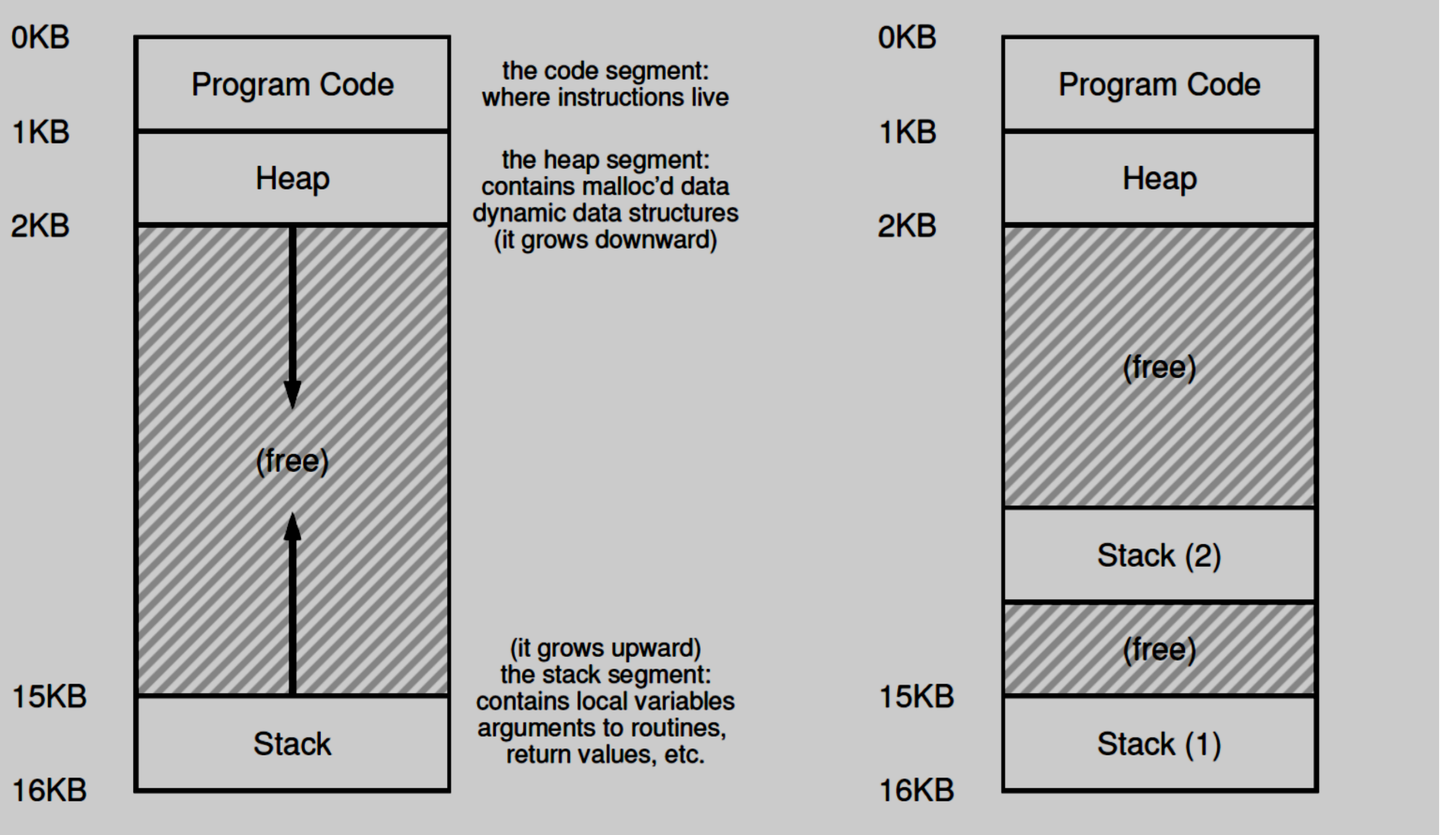
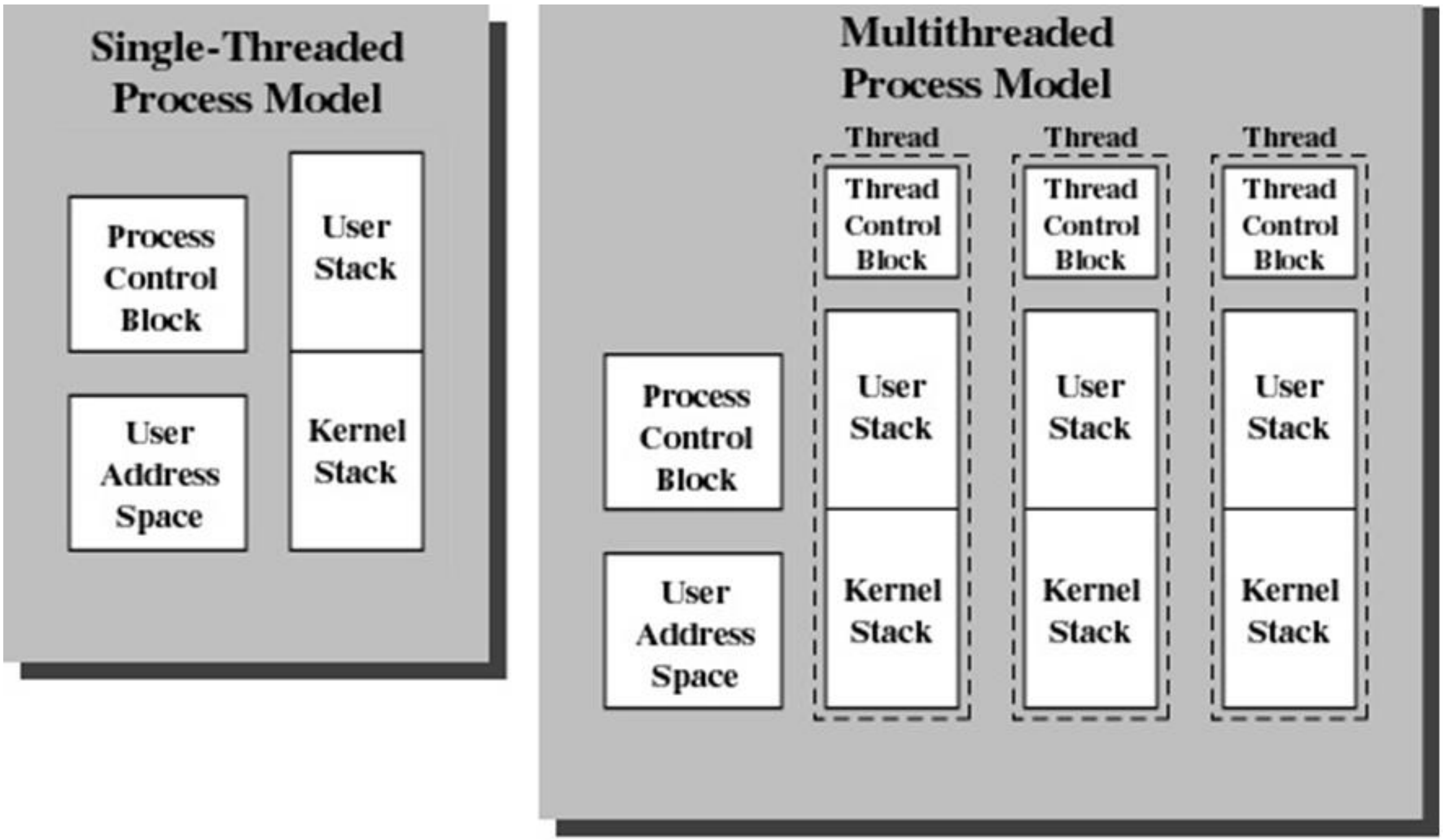


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

Multitasking vs Multithreading



Why Multithreading

Why use threads?

- Parallelism
  - Better exploit multi-core CPUs
- Blocking I/O
  - When doing a blocking I/O operation, only one thread gets blocked!
  - Overlap I/O with other activities within a process
- Same could be achieved via multiple processes but this approach is costly in terms of memory and time.
- Threads are extensively used in Web servers, DBMSs, etc.

Creation Time

- 50,000 processes or threads creation time in seconds

Platform	fork()	pthread_create()
Intel 2.6 GHz Xeon E5-2670 (16cpus/node)	8.1	0.9
Intel 2.8 GHz Xeon 5660 (12cpus/node)	4.4	0.7
AMD 2.3 GHz Opteron (16cpus/node)	12.5	1.2
AMD 2.4 GHz Opteron (8cpus/node)	17.6	1.4
IBM 4.0 GHz POWER6 (8cpus/node)	9.5	1.6
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	1.7
IBM 1.5 GHz POWER4 (8cpus/node)	104.5	2.1
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.6
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	2

Benefits of Threads

- **Less time to**
  - create a new thread than a process
  - terminate a thread than a process
  - switch between two threads within the same process
- **Low Memory Requirements than Multiprogramming**
- Since threads within the same process share memory and files, they can **communicate with each other without invoking the kernel**

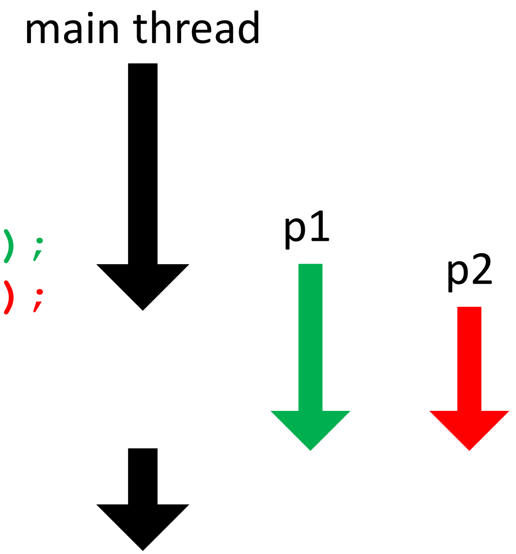
Example Code

Threads: Example Code

```
#include <stdio.h>
#include <pthread.h>
#include “common.h”
#include “common_threads.h”

void *mythread(void* args) {
   print(“%s\n”, (char*) args);
   return NULL;
}

int main(int argc, char *argv[])
{
   pthread_t p1, p2;
   printf(“main begin\n");
   Pthread_create(&p1, NULL, mythread, “A”);
   Pthread_create(&p2, NULL, mythread, “B”);
   Pthread_join(p1, NULL);
   Pthread_join(p2, NULL);
   printf(“main end\n");
   return 0;
}
```



CPU Scheduling Output

- Output is dependent on which thread is created and which is scheduled first on the CPU

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)**

Issues that need to be addressed in Multithreading

- Race Condition
- Atomicity
- Mutual Exclusion
- Synchronization

Threads – accessing shared global variable

```
#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;
}
```

Threads – accessing shared global variable

```
//
// 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;
}
```

Expected Output

```
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)
```

Actual Output

- Non-deterministic output
  - On multiple runs, counter is ≠ 20000000

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

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Why is the output different

- Bone of contention (race condition/data race)
  - `counter=counter+1`

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

  - During execution, an interrupt might preempt a running thread
    - On resumption, the thread might get a stale state of shared variable and may modify it wrongly

Why is the output different?

OS	Thread 1	Thread 2	(after instruction) PC	%eax	counter
	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	51
	mov %eax, 0x8049a1c		113	51	**51**

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

Race conditions

- The result of a program is ***indeterminate***
  - It depends on the sequence in which the instructions are scheduled!

- ***Critical section:***
  - A portion of code which, if executed by multiple threads, can result in race condition
  - Usually accesses a shared resource
  - Should not be executed ***concurrently*** by more than one thread (**mutual exclusion**)

- Solution
  - Atomic operations or atomicity
  - Synchronization

Atomicity

- Mutual exclusion can be achieved if critical sections are executed ***atomically***.
- Atomic operations are executed as “all or nothing”
  - i.e. they cannot be interrupted half way.
  - Databases have similar concepts – transactions.

Operating System – the first concurrent program

- Multiple processes run concurrently
- Let’s say two processes try to append to a file
- Possible problems?
  - Interrupts!!

- All OS data structures are shared b/w processes
- Codes that update these shared data structures are critical sections

Thread APIs

- Thread Management
- Mutual Exclusion
- Synchronization

pthread API

- Simple API that allows creation of threads
- To use
  - #include <pthread.h>
  - add –pthread option at compilation to gcc
- Provides many useful thread functions and synchronization primitives (locks, condition variables etc.)

Thread functions

- Thread creation

```
#include <pthread.h>
int
pthread_create(pthread_t          *thread,
         const pthread_attr_t *attr,
               void           *(*start_routine)(void*),
               void           *arg);
```

- thread completion (thread joining to main thread)

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Thread - Passing arguments

```
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Thread – Returning values

```
typedef struct { int a; int b; } myarg_t;
typedef struct { int x; int y; } myret_t;

void *mythread(void *arg) {
    myret_t *rvals = Malloc(sizeof(myret_t));
    rvals->x = 1;
    rvals->y = 2;
    return (void *) rvals;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myret_t *rvals;
    myarg_t args = { 10, 20 };
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &rvals);
    printf("returned %d %d\n", rvals->x, rvals->y);
    free(rvals);
    return 0;
}
```

Thread – Avoid returning values of local variables

```
void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    myret_t oops; // ALLOCATED ON STACK: BAD!
    oops.x = 1;
    oops.y = 2;
    return (void *) &oops;
}
```

Locks

- Initialization function

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

- Deletion function

```
pthread_mutex_destroy()
```

- Lock Usage

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

Condition Variables

- Provide signaling between threads so if one thread is waiting on some shared data, it may signal the other threads when it releases the shared resource
- Works with locks
- Usage

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

Summary

- We have seen what threads are and how they relate to processes
- We saw how we may create multithreaded programs using the pthreads API
- We saw some issues that might pop up during multithreaded programs
  - Race conditions
- We looked at how these are avoided through synchronization (locks, condition variables) and atomic operations