Lecture 11

Jan 29, 2024

Threads & Concurrency

So, far we have studied single threaded programs

- Recall: Process execution

–PC points to current instruction being run

–SP points to stack frame of current function call

- A program can also have multiple threads of execution
- What is a thread?

Why do we need ?

Consider the case of a word processor :

It has several actions including a) respond to user's keystrokes and b) Spell check

Don’t want to slow response to user's keystrokes because of spell checking

=> We need concurrency

These concurrent work-items must share the same data (text buffer), OS resources (eg, files, display functions etc)

Why do we need ?

Consider the case of a web server:

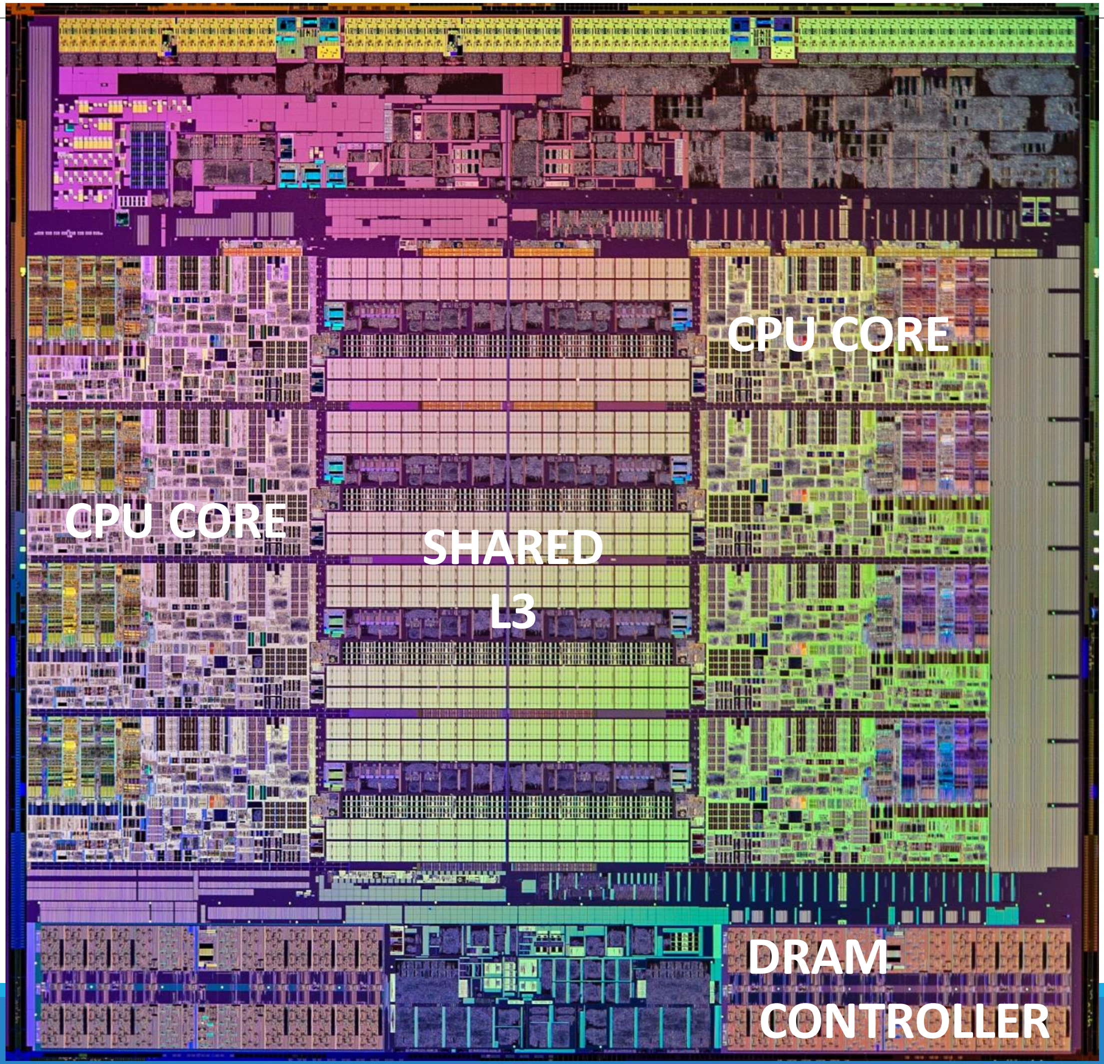
Multiple requests need to be serviced concurrently

Especially important to maximize performance as one request may be held upon an I/O call

These request handlers share the same code, OS resources (eg. Sockets), and perhaps persistent stores like databases etc

Multicores

Intel® Core™ i7-5960X



Remember : What is an OS?

Middleware between user programs and system hardware that provides the following:

Abstraction

OS makes HW easy-to-use and program by providing interfaces

Virtualization

OS creates an illusion of dedicated HW for each user and application

Concurrency

OS enables controlled interaction between multiple applications

Do we know how to do that?

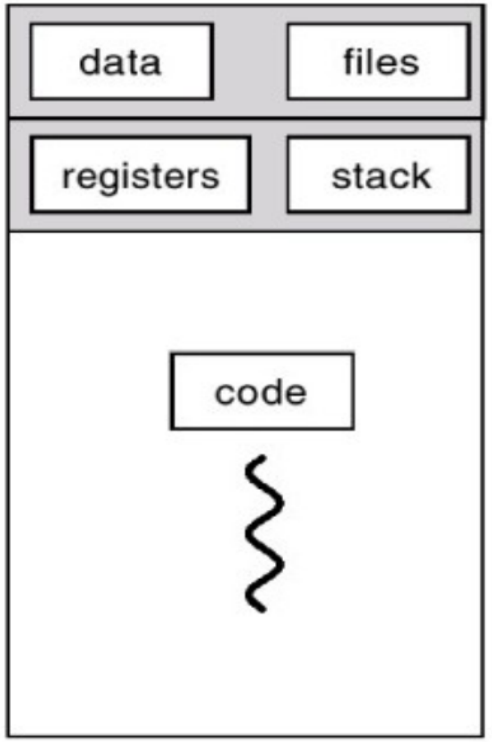
Till now, We know processes and IPCs

For the word processor application, we can create a new process that does spell check and communicates by files/signals

For the webserver, whenever we get a new request, we can fork a child that services the request

Process Execution

- Separate streams of execution
- Each process isolated from the other
- Process state contains
  - Process ID
  - Environment
  - Working directory.
  - Program instructions
  - Registers
  - Stack
  - Heap
  - File descriptors
- Created by the OS using fork
  - Significant overheads



Process is an overkill

In essence a process tracks three things

Address Space : Code, data, heap, (PT)

Execution State : PC, SP, register values …

OS State : ID, open files, user permissions, directory, network connections, signals etc

Which of these do we need to share amongst the concurrent units from the earlier examples ?

Process is an overkill

Address Space : Code, data, heap, (PT) -- SHARED

Execution State : PC, SP, register values -- SEPARATE

OS State : ID, open files, user permissions, directory, network connections, signals etc

Need a 'lightweight process' as the unit of execution

"THREADS"

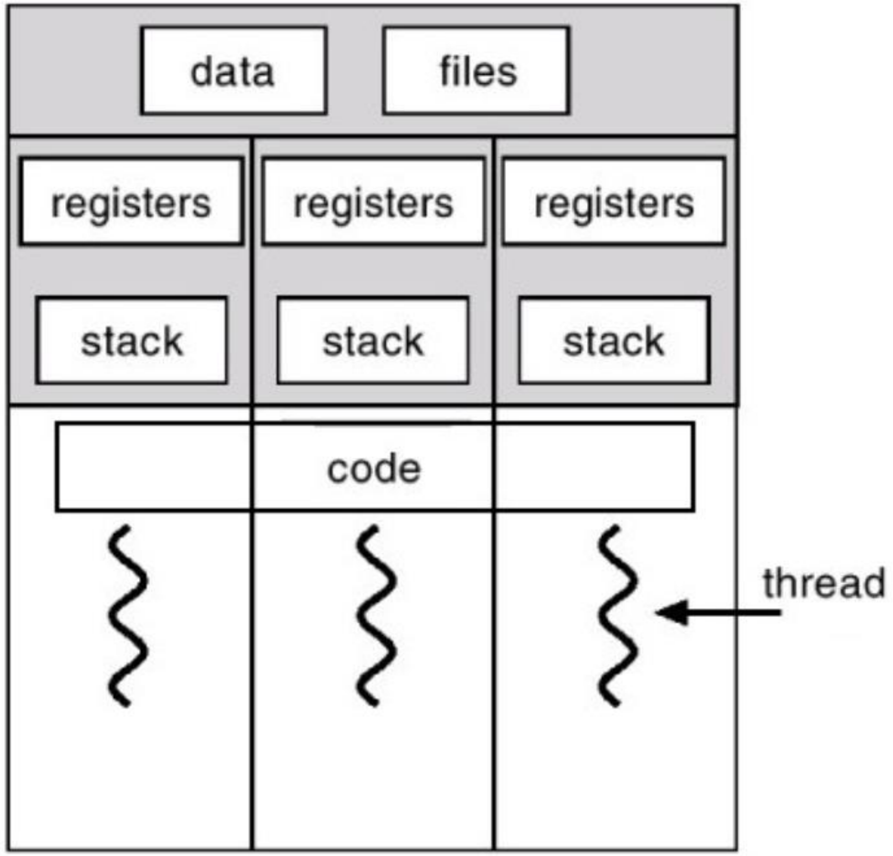
Threads

A thread is like another copy of a process that executes independently

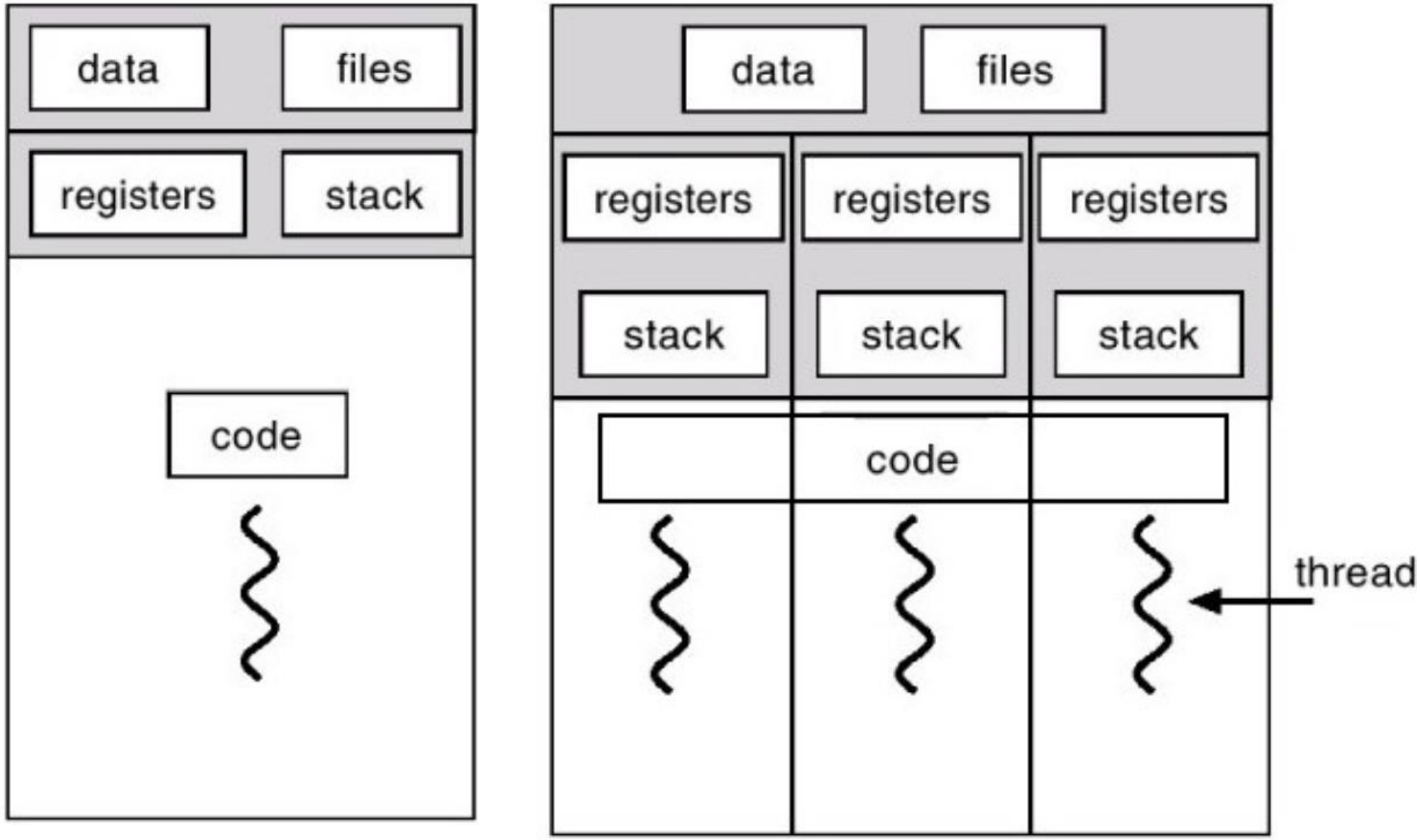
- Threads shares the same address space (code, heap)
- Each thread has separate PC
  - Each thread may run over different part of the program
- Each thread has separate stack for independent function calls

Threads

- Separate streams of execution within a single process
- Threads in a process <u>not</u> isolated from each other
- Each thread state (thread control block) contains
  - Registers (including EIP, ESP)
  - stack



Threads vs Processes



Key Idea: Separate **process info** (address space, OS resources) from **execution thread** (PC, regs, stack)

Why threads ?

• Parallelism: a single process can effectively utilize multiple CPU cores

–Understand the difference between concurrency and parallelism

–Concurrency: running multiple threads/processes at the same time, even on single CPU core, by interleaving their executions

–Parallelism: running multiple threads/processes in parallel over different CPU cores

• Even if no parallelism, concurrency of threads ensures effective use of CPU when one of the threads blocks (e.g., for I/O)

Threads - Advantages

Communication between threads is very efficient – Same address space

Context switch between threads of same process is faster: Eg. No page table change

Each thread can work on its own data: Independent stack and registers

Threads can advantage of multi-core systems : Schedulable entity is a thread

Why threads?

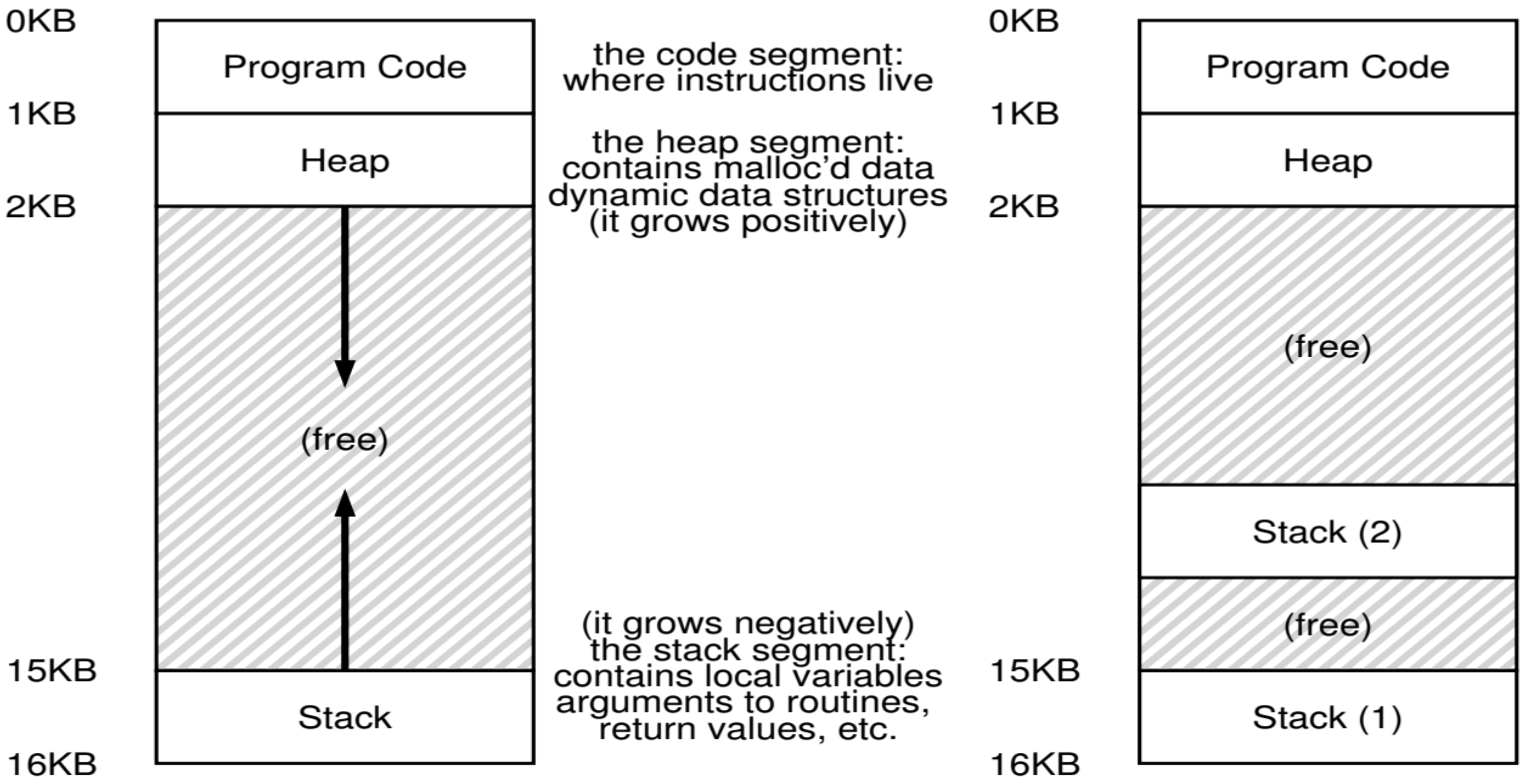
- Lightweight

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Cost of creating 50,000 processes / threads
(https://computing.llnl.gov/tutorials/pthreads/)

- Efficient communication between entities
- Efficient context switching

Single-Threaded And Multi-Threaded Address Spaces



Threads vs Processes

Parent P forks a child C

–P and C do not share any memory

–Need complicated IPC mechanisms to communicate

–Extra copies of code, data in memory

• Parent P executes two threads T1 and T2

–T1 and T2 share parts of the address space

–Global variables can be used for communication

–Smaller memory footprint

• Threads are like separate processes, except they share the same address space

Comparison

- A thread has no data segment or heap
- A thread cannot live on its own. It needs to be attached to a process
- There can be more than one thread in a process. Each thread has its own stack
- If a thread dies, its stack is reclaimed

- A process has code, heap, stack, other segments
- A process has at-least one thread.
- Threads within a process share the same I/O, code, files.
- If a process dies, all threads die.

Trivia

Do the following statements apply to Processes (P), Threads (T) or both (B)

Can share a virtual address space

Take longer to context switch

Have an execution context

Usually result in hotter caches when multiple exist

Make use of some communication mechanisms

Scheduling Threads

- OS schedules threads that are ready to run independently, much like processes
- The context of a thread (PC, registers) is saved into/restored from thread control block (TCB)
  - Every PCB has one or more linked TCBs
- Threads that are scheduled independently by kernel are called kernel threads
  - E.g., Linux pthreads are kernel threads
- In contrast, some libraries provide user-level threads
  - User program sees multiple threads
  - Library multiplexes larger number of user threads over a smaller number of kernel threads
  - Low overhead of switching between user threads (no expensive context switch)
  - But multiple user threads cannot run in parallel

pthread library

- Create a thread in a process

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                    void *arg);
```

Thread identifier (TID) much like

Pointer to a function, which starts execution in a different thread

Arguments to the function

- Destroying a thread

```
void pthread_exit(void *retval);
```

Exit value of the thread

Pthread contd..

- Join : Wait for a specific thread to complete

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

TID of the thread to wait for

Exit status of the thread

what is the difference with wait()?

```
#define Pthread_create(thread, attr, start_routine, arg) assert(pthread_create(thread, attr, start_routine, arg) == 0);
#define Pthread_join(thread, value_ptr)                  assert(pthread_join(thread, value_ptr) == 0);
```

Example

```c
#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;
}
```

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

void *thread_fn(void *arg){
  long id = (long) arg;
  printf("Starting thread %ld\n", id);
  sleep(5);
  printf("Exiting thread %ld\n", id);
  return NULL;
}

int main(){
  pthread_t t1, t2;

  pthread_create(&t1, NULL, thread_fn, (void *)1);
  pthread_create(&t2, NULL, thread_fn, (void *)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Exiting main\n");
  return 0;
}
```

Thread Trace

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"		

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"		

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"		

Example: With 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;
}

// 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;
}
```

With shared data – What happens?

```
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)
```

What do we expect? Two threads, each increments counter by 10^7, so 2X10^7

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

Sometimes, a lower value. Why?

Assembly code

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	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**

Assembly code of
`counter = counter + 1`

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

Race condition & Synchronization

What just happened is called a race condition –Concurrent execution can lead to different results (functional output depends on order of execution)

Non-deterministic scheduling can change results

• **Critical section:** portion of code that can lead to race conditions

Usually critical sections access shared resources (eg. Variable on heap)

• What we need: mutual exclusion–Only one thread should be executing critical section at any time

• What we need: atomicity of the critical section –**The critical section should execute like one uninterruptible instruction**