Week 7
Multithreading

20 February 2019

CS 35L Lab 4

Jeremy Rotman

Announcements

- ➔ Assignment #6 is due Saturday by 11:55pm
- ➔ For Assignment #10
  - ◆ **Email me to tell me what story you are choosing**
  - ◆ Here is the link to see what stories people have signed up for already
    - Choose a story at least one week before you present
    - Try to avoid political topics

Outline

- ➔ Multithreading and Multitasking
- ➔ POSIX threads
- ➔ Assignment 6

Questions?

Multiprocessing

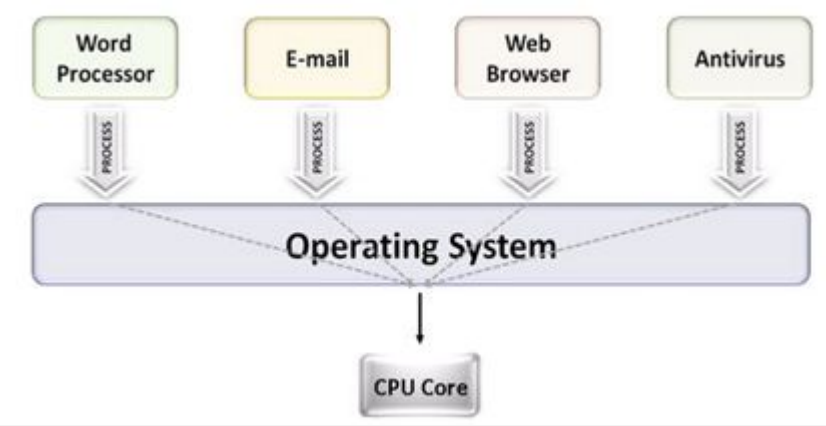
What is Multiprocessing?

Multiprocessing

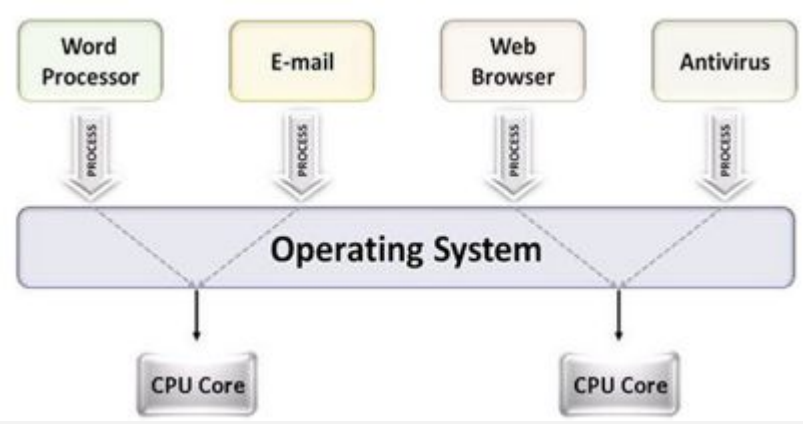
What is Multiprocessing?

The use of multiple CPUs/cores to run multiple tasks simultaneously

Uniprocessing System



Multiprocessing System



Parallelism

Parallelism

➔ Executing several computations simultaneously to gain performance

Parallelism

- Executing several computations simultaneously to gain performance
- Two primary forms of parallelism
  - Multitasking
    - Several processes are scheduled in an alternating pattern
    - Potentially, a simultaneous pattern if on a multiprocessing system
  - Multithreading
    - The same job is split into logical pieces (threads)
    - Threads may be executed simultaneously in a multiprocessing system

Threads

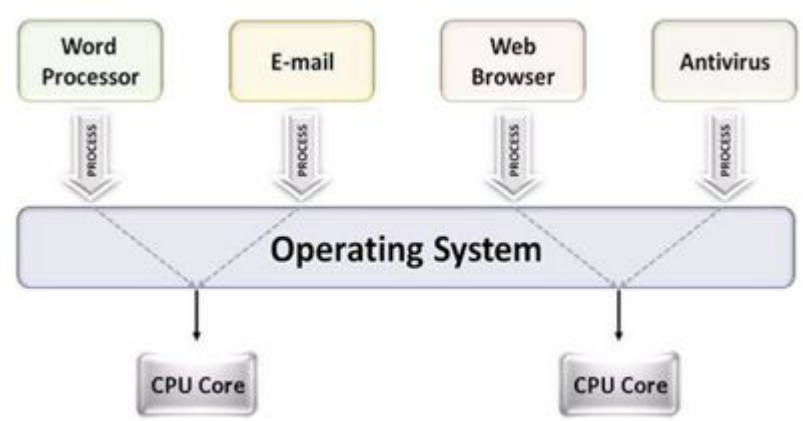
- ➔ A flow of instructions, or path of execution within a process
- ➔ Smallest unit of process scheduled by the OS
- ➔ A process consists of at least one thread

Multithreading

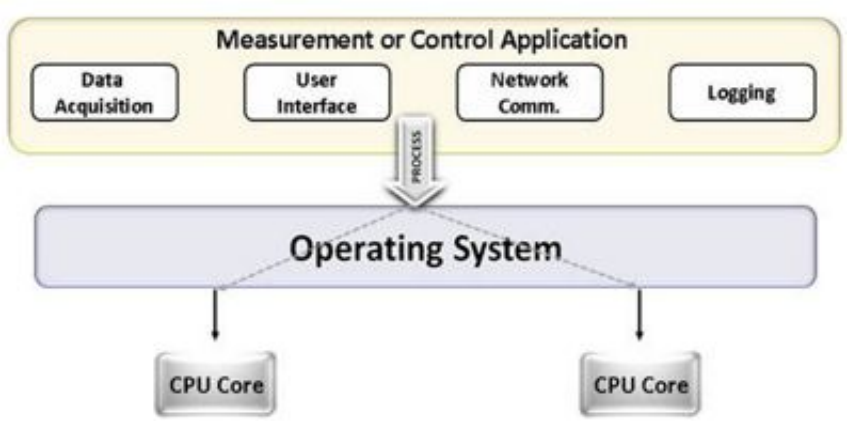
- Multiple threads can be run on:
  - A Uniprocessor
    - Processor switches between threads
    - An illusion of parallelism
  - A Multiprocessor
    - Multiple processes or cores run the threads at the same time
    - True parallelism

Multitasking vs. Multithreading

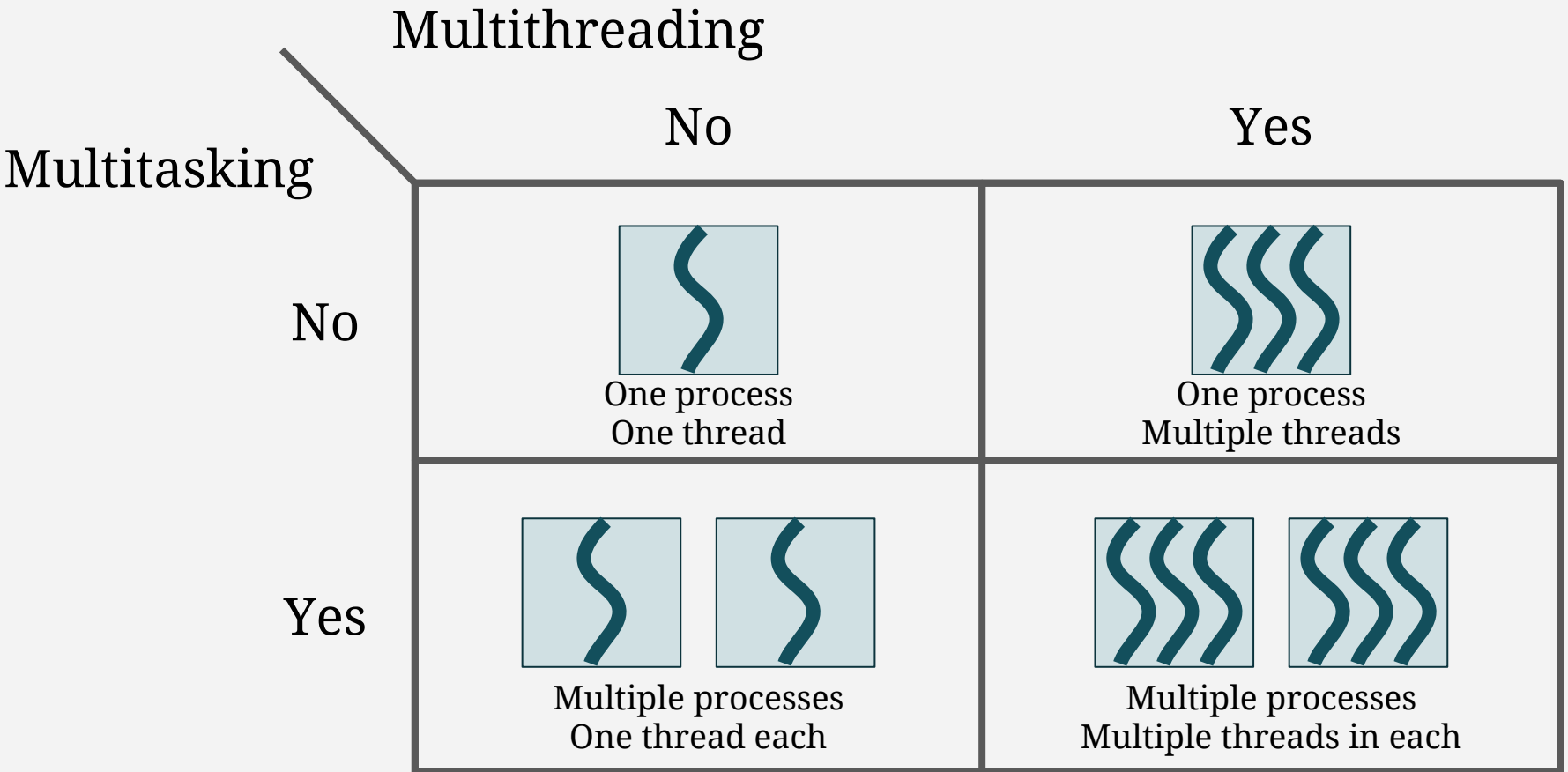
Multitasking



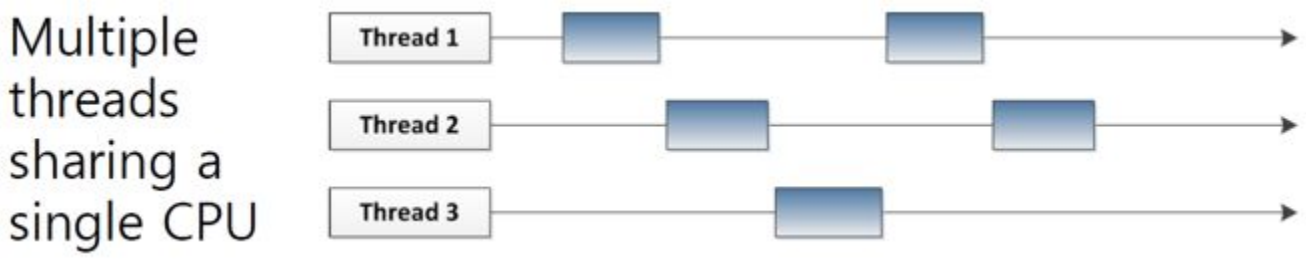
Multithreading



Using Both

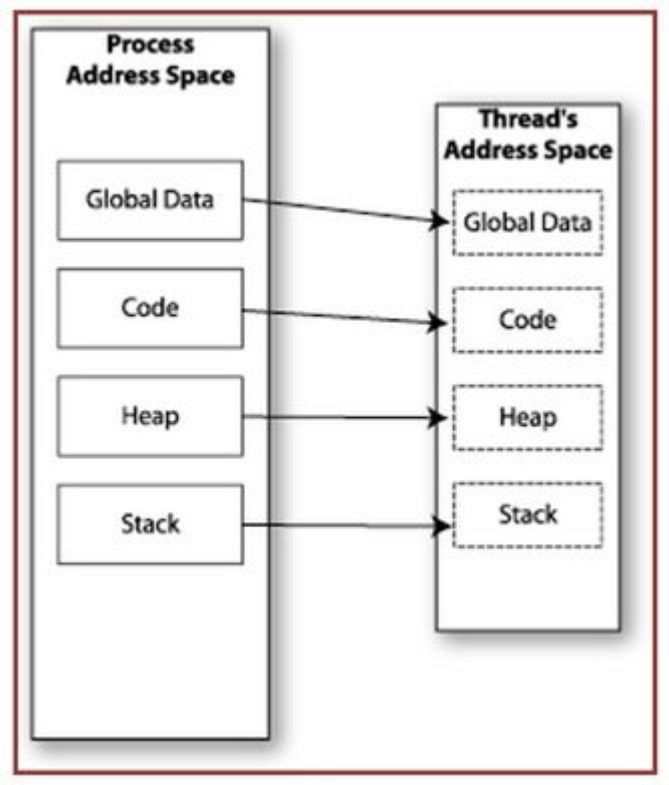

Multithreading
No
Yes
Multitasking
No
Yes
One process
One thread
One process
Multiple threads
Multiple processes
One thread each
Multiple processes
Multiple threads in each

Scheduling Threads



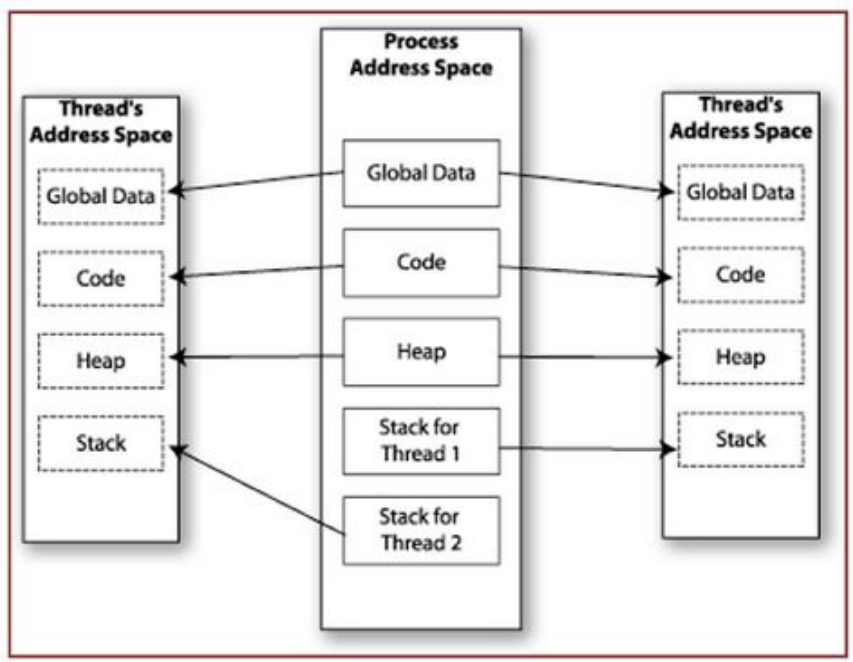
Memory Layout

- ➔ Single-threaded program
- ➔ The process allocates space for the thread



Memory Layout

- ➔ Multi-threaded program
- ➔ The process allocates individual space for each thread's stack
- ➔ Each thread shares the rest of the process's space

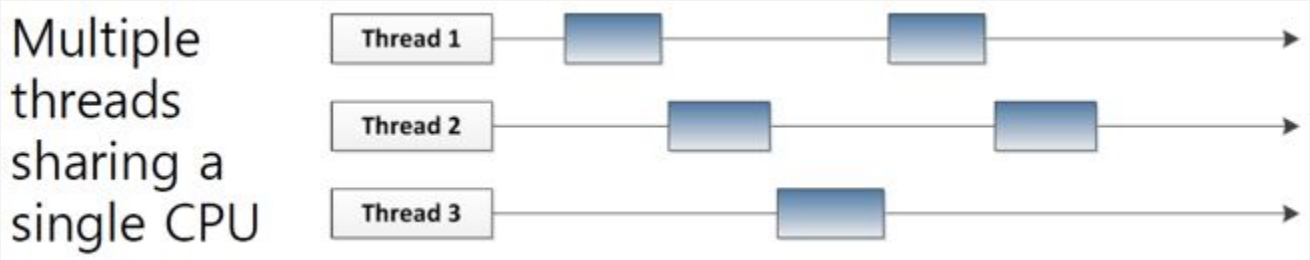


Shared Memory

- Every process receives its own memory
- Every thread receives part of its parent process's memory
  - Threads from the same parent process share memory
- Data sharing requires no extra work
  - Processes must interact with other processes with system calls, pipes, etc.
- Makes Multithreading efficient
  - Also less expensive
- Also makes Multithreading difficult
  - Race Conditions

Race Conditions

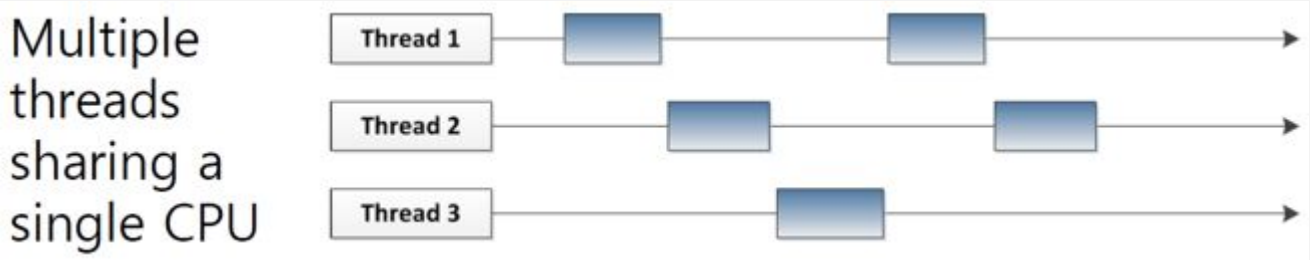
➔ Threads are often scheduled in small chunks



➔ What happens if two threads both rely on specific values of data in memory?

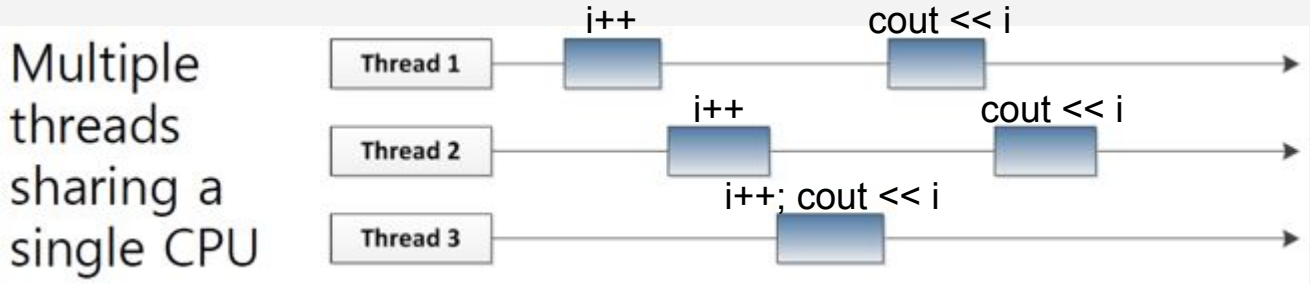
Race Conditions

```
i = 0;
while(i < 10) {
    i++;
    cout << i;
}
```



Race Conditions

```
i = 0;
while(i < 10) {
    i++;
    cout << i;
}
```



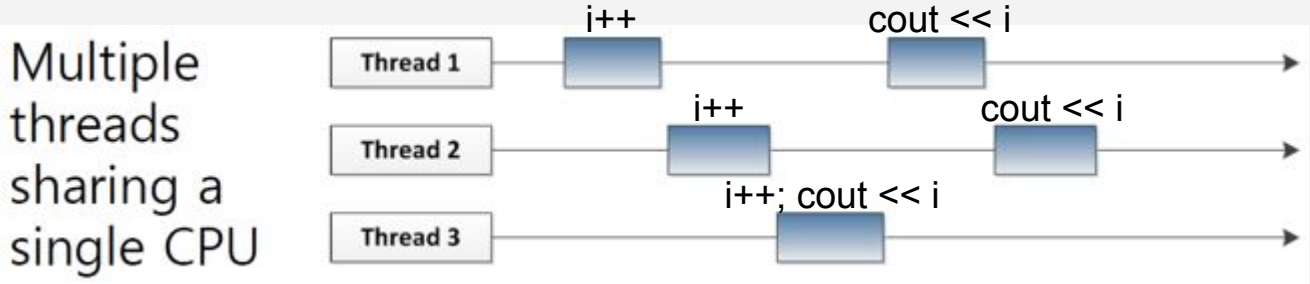
Race Conditions

Output:

3

3

3



Multithreading and Multitasking

- Multithreading
  - Threads share the same address space
    - Lightweight creation/destruction
    - Easy inter-thread communication
    - An error in one thread can bring all of the threads down
- Multitasking
  - Processes are insulated from each other
    - Expensive creation/destruction
    - Expensive IPC
    - An error in one process cannot bring down another process

POSIX Threads (pthreads)

Built in functions to create and manage threads:

- pthread_create
  - Create a new thread within a process
- pthread_join
  - Waits for another thread to terminate
- pthread_equal
  - Compare thread IDs to check if they are equal
- pthread_self
  - Return ID of the calling thread
- pthread_exit
  - Terminates the currently running thread

pthread_create

- Create a new thread and make it executable
- Can be called any number of times from anywhere in the code
- Return
  - Success: 0
  - Failure: Error Code

pthread_create

```
int pthread_create(pthread_t* tid, const pthread_attr_t*
                   attr, void* (myfunction)(void*), void* arg);
```

- tid: unique identifier for new thread
- attr: object that holds attributes
  - NULL for default
- my_function: function the thread will execute
- arg: a *single* argument that can be passed to my_function
  - NULL for no argument

pthread_join

- Makes originating thread wait for the completion of all of its spawned reads
- Without join, the originating thread would exit as soon as it’s done with the jon
  - A spawned thread can be aborted even if it is in the middle of its chore
- Return
  - Success: 0
  - Failure: Error Code

pthread_join

int pthread_join(pthread_t tid, void** status);

- ➔ tid: ID of thread to wait on
- ➔ status: Exit status of target thread is stored in the location pointed to by *status
  - ◆ NULL if no status needed

The job to be done by the thread, one void* argument

Loop to create threads, thread IDs are just ints

Loop to wait for each thread to close

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 5

void *perform_work(void *arguments){
  int index = *((int *)arguments);
  int sleep_time = 1 + rand() % NUM_THREADS;
  printf("THREAD %d: Started.\n", index);
  printf("THREAD %d: Will be sleeping for %d seconds.\n", index, sleep_time);
  sleep(sleep_time);
  printf("THREAD %d: Ended.\n", index);

}

int main(void) {
  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS];
  int i;
  int result_code;

  //create all threads one by one
  for (i = 0; i < NUM_THREADS; i++) {
    printf("IN MAIN: Creating thread %d.\n", i);
    threads[i] = i;
    thread_args[i] = i;
    result_code = pthread_create(&threads[i], NULL, perform_work, &thread_args[i]);
    assert(!result_code);
  }

  printf("IN MAIN: All threads are created.\n");

  //wait for each thread to complete
  for (i = 0; i < NUM_THREADS; i++) {
    result_code = pthread_join(threads[i], NULL);
    assert(!result_code);
    printf("IN MAIN: Thread %d has ended.\n", i);
  }

  printf("MAIN program has ended.\n");
  return 0;
}
```

Lab 6

- Evaluate multithreaded sort
  - Make sure to use the path /usr/local/cs/bin
- Generate a file with 2^{24} random floating point numbers
  - This may show up as 2^2^4 in the spec on your laptop
- Get this by reading bytes from /dev/random as single-precision floats with **od**
  - -t: selects output format
  - -N *count*: formats no more than *count* bytes
- Use **sed** and **tr** to format this to be one number per line
  - Also remove whitespace

Homework 6

- Ray Tracing
  - Mimics propagation of light through objects
  - Simulates effects of a single light ray as it's reflected or absorbed by objects in the images
  - You'll be given code that does this
- Run the original file
- Then, make it multithreaded
- Run the multithreaded version and compare the images

Homework 6

- Modifying the Program
  - Include <pthread.c> in main.c
  - Use pthread_create and pthread_join in main.c
  - Link with -lpthread flag (LDLIBS target)
- Make clean check
  - Outputs “1-test.ppm”
  - To see “1-test.ppm”
    - Ubuntu: sudo apt-get install gimp
    - lnxsrv: X Forwarding
    - gimp 1-test.ppm
    - Alternatively, transfer to your own computer

Questions?