CS 35L- Software Construction Laboratory

Fall 2016

TA: Jin Wang

November 7, 2016

Course Information

- Course Schedule
  - Week 8: dynamic linking
  - Week 9: SSH + mutex in multi-thread processing
  - Week 10: final review

  Note: Submit the assignment 10 on CCLE not later then Friday of week 10

  Slides of week 10 will not be posted!

  DO COME TO THE CLASS!

Multithreading/Parallel Processing

Week 7

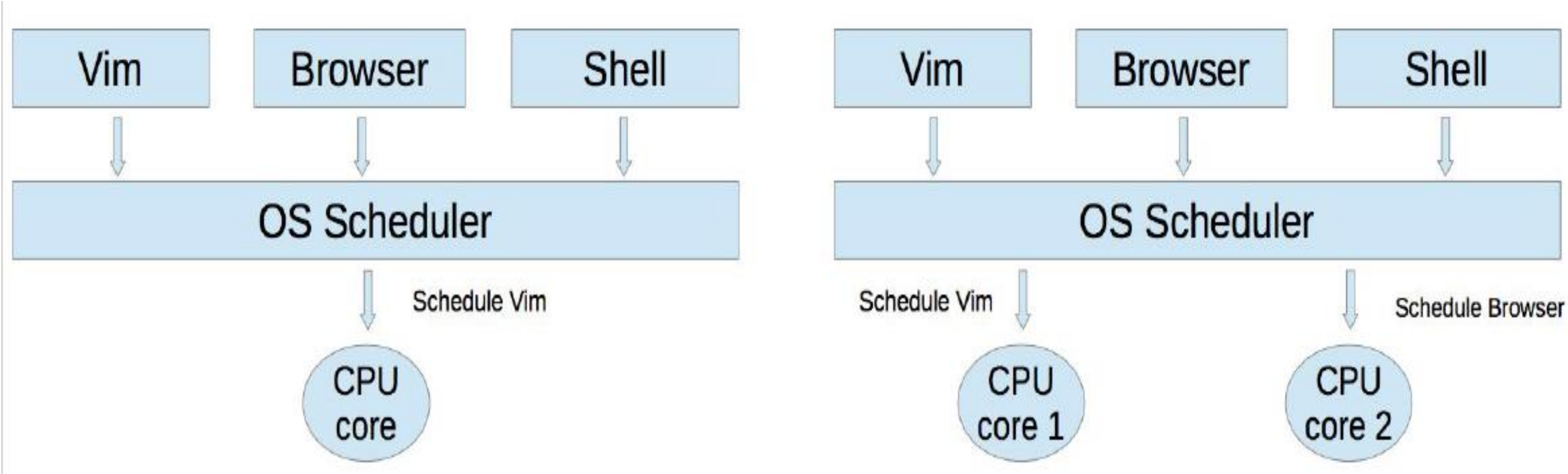
Outline

- Basic Knowledge about Multi-Thread Processing
- Pthread API
- Hints for Assignment 7

Multitasking

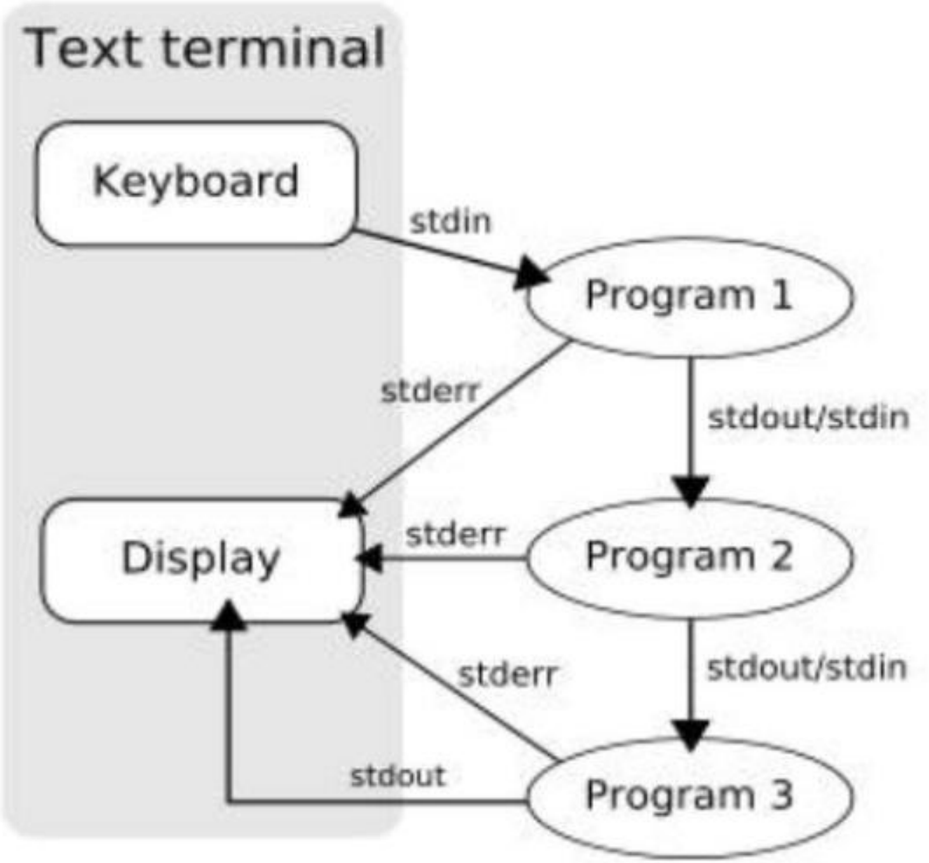
- Run multiple processes **simultaneously** to increase performance
- Processes do not share internal structures (stacks, globals, etc)
- Single core: Illusion of parallelism by switching processes quickly (**time-sharing**)
- **Multi-core: True** parallelism. Multiple processes execute **concurrently** on different CPU cores

Architecture: Single core vs. Multi-core



Multitasking

- tr -s ‘[:space:]’ ‘\n’ | sort -u | comm -23 – words
- Three separate processes spawned simultaneously
  - P1 – tr
  - P2 – sort
  - P3 - comm
- Common buffers (pipes) exist between 2 processes for communication
- ‘tr’ writes its stdout to a buffer that is read by ‘sort’
- ‘sort’ can execute, as and when data is available in the buffer
- Similarly, a buffer is used for communicating between ‘sort’ and ‘comm’

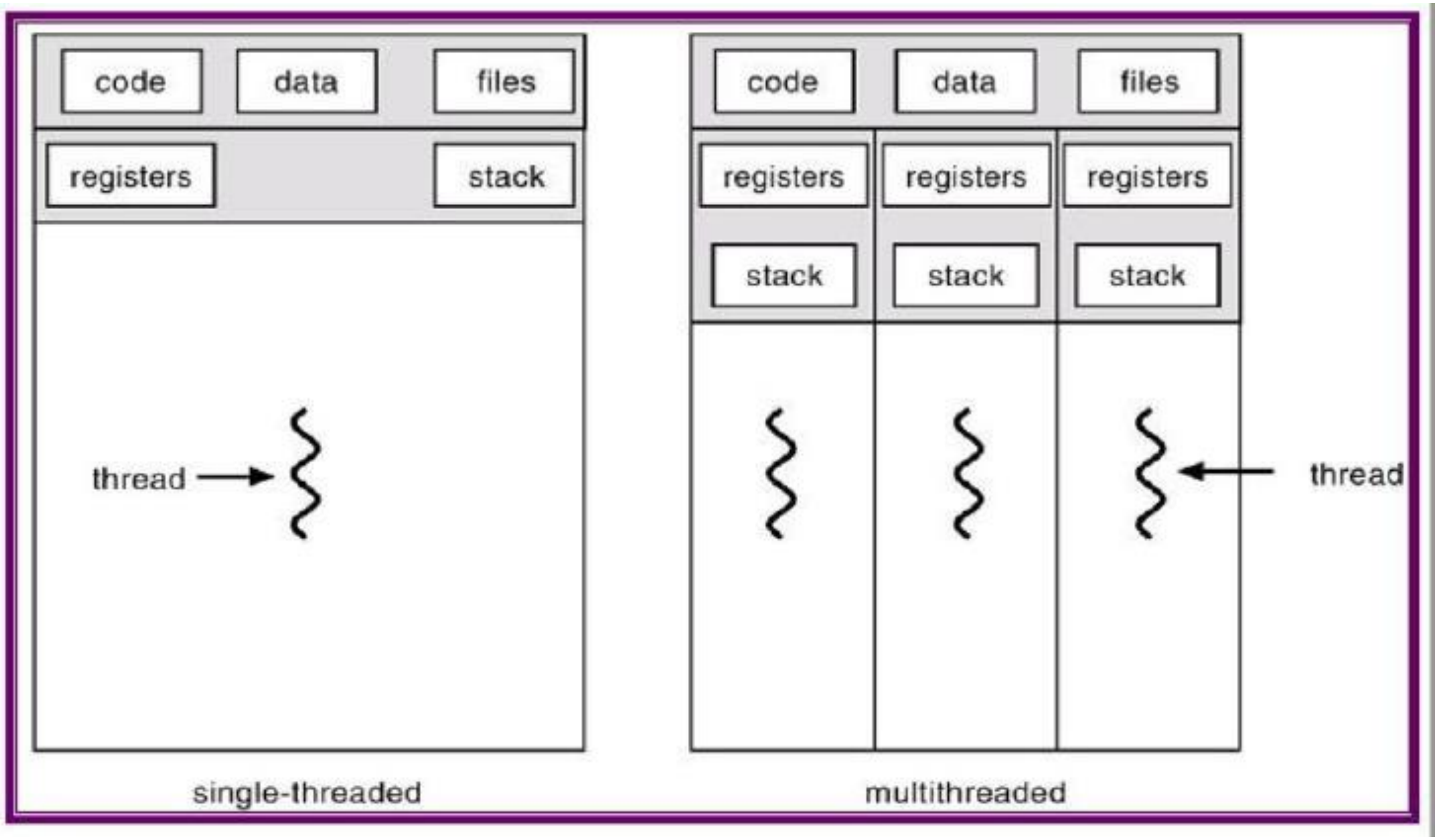


Thread

- A process can be
  - Single-threaded
  - Multi-threaded
- Threads in a process can run in parallel
- A thread is a lightweight process
- It is a **basic unit** of CPU utilization

Thread(cont.)

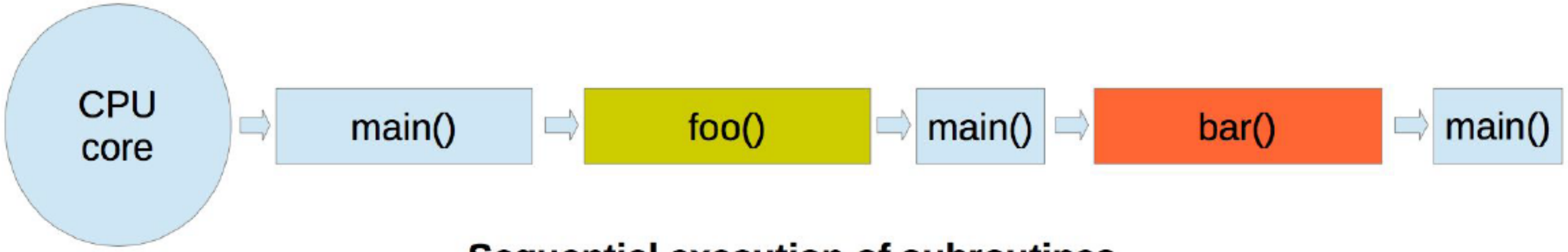
- Each thread has its own:
  - Stack
  - Registers
  - Thread ID
- Each thread shares the following with other threads belonging to the same process:
  - Code
  - Global Data
  - OS resources



Single threaded execution

```
int global_counter = 0
int main()
{
    …
    foo(arg1,arg2);
    bar(arg3,arg4,arg5);
    …
    return 0;
}
```

```
void foo(arg1,arg2)
{
    //code for foo
}
void bar(arg3,arg4,arg5)
{
    //code for bar
}
```


CPU core

main()

foo()

main()

bar()

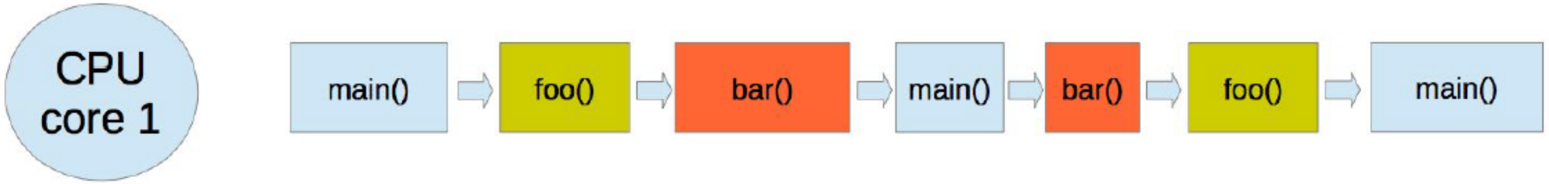
main()

Sequential execution of subroutines

Multi threaded execution (single core)

```
int global_counter = 0
int main()
{
    …
    foo(arg1,arg2);
    bar(arg3,arg4,arg5);
    …
    return 0;
}
```

```
void foo(arg1,arg2)
{
    //code for foo
}
void bar(arg3,arg4,arg5)
{
    //code for bar
}
```

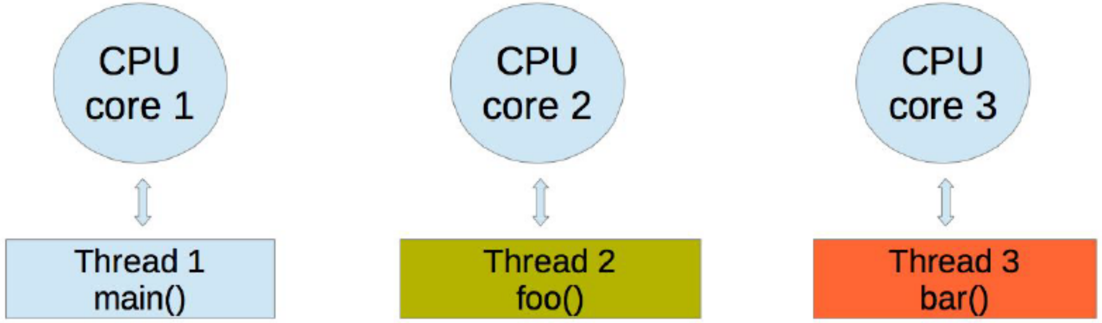


Time Sharing – Illusion of multithreaded parallelism
(Thread switching has less overhead compared to process switching)

Multi threaded execution (multiple cores)

```
int global_counter = 0
int main()
{
    …
    foo(arg1,arg2);
    bar(arg3,arg4,arg5);
    …
    return 0;
}
```

```
void foo(arg1,arg2)
{
    //code for foo
}
void bar(arg3,arg4,arg5)
{
    //code for bar
}
```



True multithreaded parallelism

Multithreading properties

- Efficient way to parallelize tasks
- **Thread switches are less expensive** compared to process switches (context switching)
- Inter-thread communication is easy, via **shared global** data
- Need **synchronization** among threads accessing same data

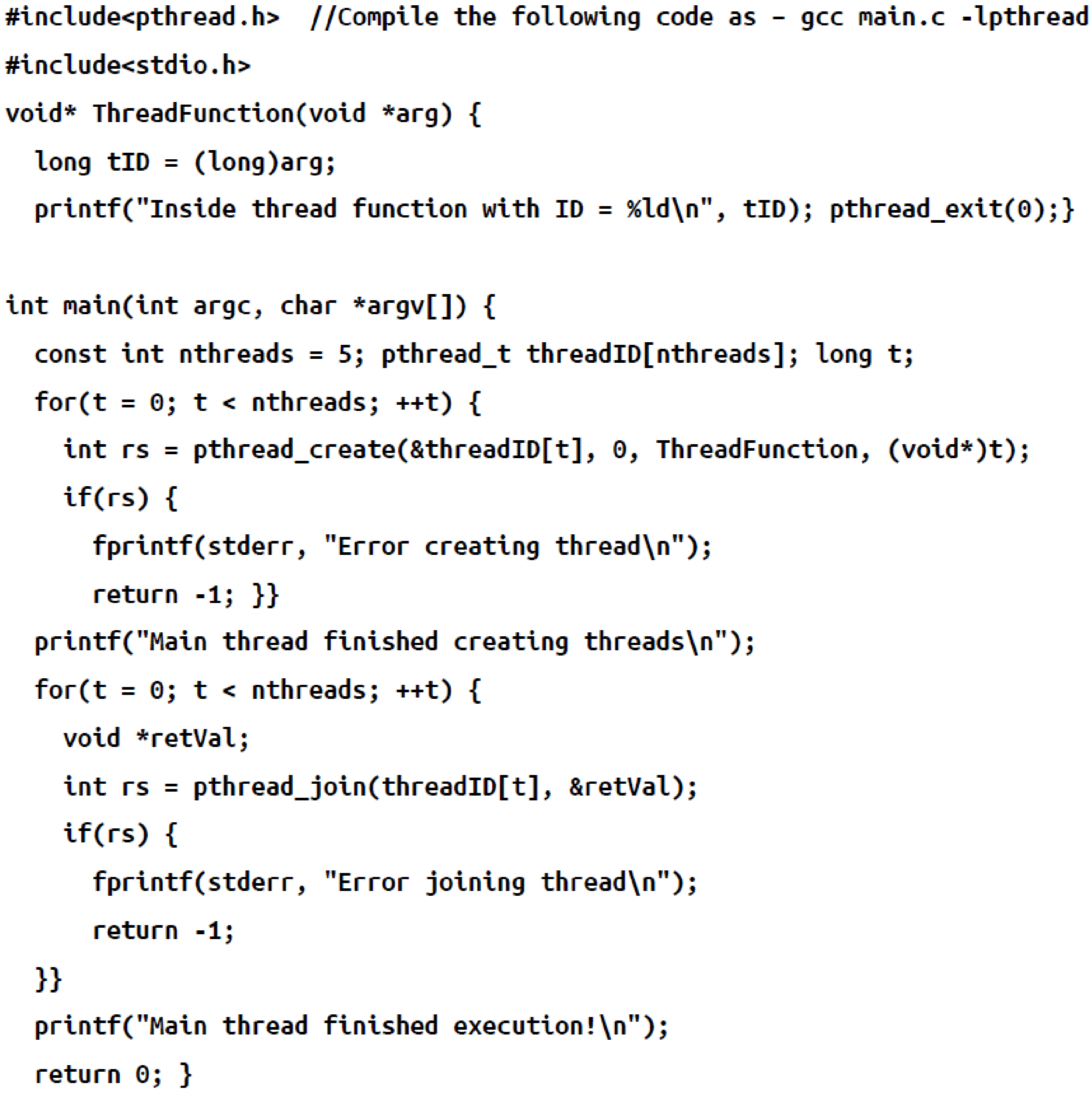
Pthread API

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

    Returns 0 on success, otherwise returns non-zero number
  - void **pthread_exit**(void *retval);
  - int **pthread_join**(pthread_t thread, void **retval);

    Returns 0 on success, otherwise returns non zero error number

Pthread API



```
#include<pthread.h>  //Compile the following code as - gcc main.c -lpthread
#include<stdio.h>
void* ThreadFunction(void *arg) {
  long tID = (long)arg;
  printf("Inside thread function with ID = %ld\n", tID); pthread_exit(0);}

int main(int argc, char *argv[]) {
  const int nthreads = 5; pthread_t threadID[nthreads]; long t;
  for(t = 0; t < nthreads; ++t) {
    int rs = pthread_create(&threadID[t], 0, ThreadFunction, (void*)t);
    if(rs) {
      fprintf(stderr, "Error creating thread\n");
      return -1; }}
  printf("Main thread finished creating threads\n");
  for(t = 0; t < nthreads; ++t) {
    void *retVal;
    int rs = pthread_join(threadID[t], &retVal);
    if(rs) {
      fprintf(stderr, "Error joining thread\n");
      return -1;
  }}
  printf("Main thread finished execution!\n");
  return 0; }
```

Thread safety/synchronization

- **Thread safe function** - safe to be called by multiple threads at the same time. Function is free of ‘race conditions’ when called by multiple threads simultaneously
- **Race condition** - the output depends on the order of execution
    - Shared data changed by 2 threads
        - int balance = 1000
    - Thread 1
        - T1 - read balance
        - T1 - Deduct 50 from balance
        - T1 - update balance with new value
    - Thread 2
        - T2 - read balance
        - T2 - add 150 to balance
        - T2 - update balance with new value

Thread safety/synchronization

- Order 1
  - balance = 1000
  - T1 - Read balance (1000)
  - T1 - Deduct 50: 950 in temporary result
  - T2 - read balance (1000)
  - T1 - update balance: 950 at this point
  - T2 - add 150 to balance: 1150 in temporary result
  - T2 - update balance: balance is 1150 at this point
- **The final value of balance is 1150**

- Order 2
  - balance = 1000
  - T1 - read balance (1000)
  - T2 - read balance (1000)
  - T2 - add 150 to balance: 1150 in temporary result
  - T1 - Deduct 50: 950 in temporary result
  - T2 - update balance: balance is 1150 at this point
  - T1 - update balance: balance is 950 at this point
- **The final value of balance is 950**

Thread synchronization

- **Mutex (mutual exclusion)**

  Threads start with “Mutex.lock()” and end with “Mutex.unlock()”
  - Thread 1
    - Read balance
    - Deduct 50 from balance
    - Update balance with new value
  - Thread 2
    - Read balance
    - Add 150 to balance
    - Update balance with new value
- Only one thread will get the mutex. Other thread will **block in Mutex.lock()**
- Other thread can start execution only when the thread that holds the mutex calls **Mutex.unlock()**

Summary of Multi-Thread Programming

- Multithreads is an efficient way to parallelize tasks
- **Thread switches are less expensive** compared to process switches (context switching)
- Inter-thread communication is easy, via **shared global** data
- Need **synchronization** among threads accessing same data
    - e.g. Mutex.lock(), Mutex.unlock()

Assignment 7 is available

- Visit:

  http://web.cs.ucla.edu/classes/fall16/cs35L/assign/assign7.html
- Deadline: 11:55 PM, **Nov. 13**

Hints for Assignment 7

The grade break down

- Lab 25%
  - Lab log 25% (manually)
- Homework 75%
  - Output of make clean check 10% (automatically + manually)
  - Homework report 15% (manually)
  - The Multi-thread program 50% (automatically)

Lab 7

- Evaluate the performance of multithreaded ‘sort’ command
- Delete the empty line
- Add /usr/local/cs/bin to PATH (export)
- Generate a file containing 10M random **double precision floating point numbers, one per line with no white space**
  - /dev/urandom: pseudo-random number generator
  - od -An -t f8 -N 10000000 < /dev/urandom
  - Question: what’s the meaning of these options ?

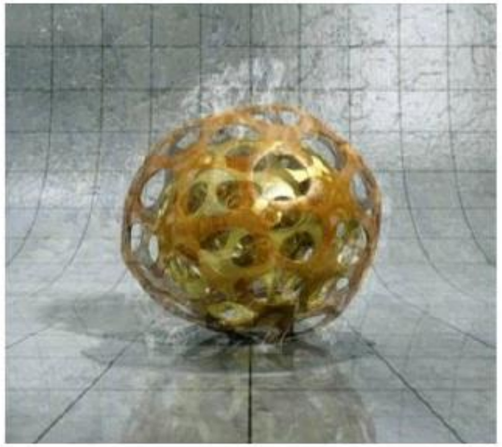
Lab 7

- od
  - Write the contents of its input files to standard output in a user-specified format
  - Options
    - -t f: Double-precision floating point
    - -N <count>: Format no more than *count* bytes of input
- sed, tr
  - Remove address, delete spaces, add newlines between each float
  - [generate random numbers]| tr -s ' ' '\n‘ >[yout .txt file]

Lab 7

- Use time -p to time the command sort -g on the data you generated
- Send output to /dev/null
- Run sort with the --parallel option and –g option: compare by general numeric value
    - Use time command to record the real, user and system time when running sort with 1, 2, 4, and 8 threads
    - Record the times and steps in log.txt
    - e.g. time -p /usr/local/cs/bin/sort -g --parallel=2 [your text file] > /dev/null

Homework 7: Ray-Tracing



Homework 7: Motivation

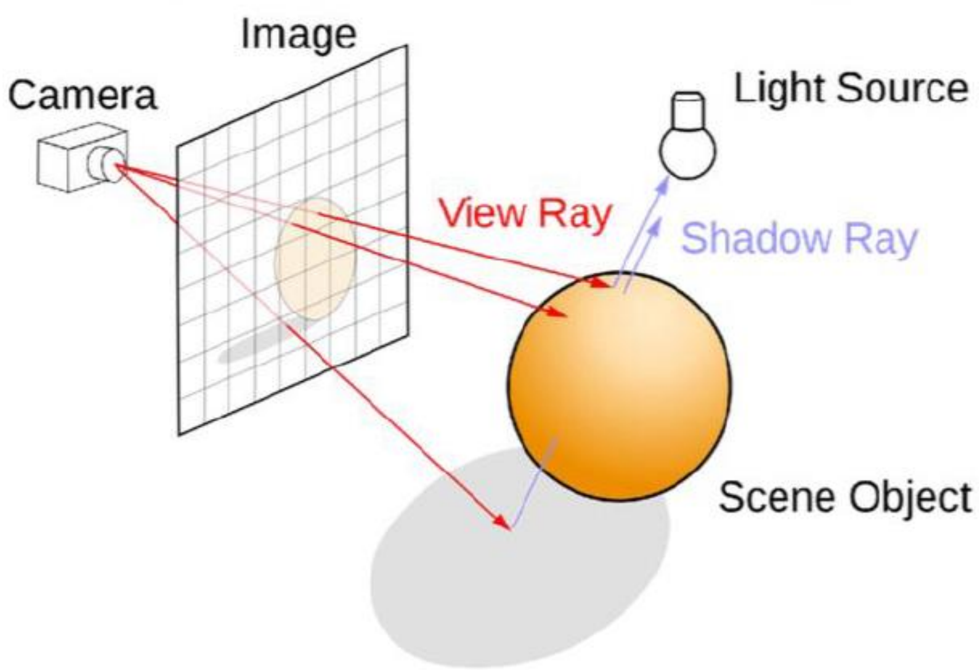
- SIGGRAPH 2015 technical paper: https://www.youtube.com/watch?v=XrYkEhs2FdA

Homework 7: Ray-Tracing

- **Powerful rendering technique in Computer Graphics**
- **Yields high quality rendering**
  - Suited for scenes with complex light interactions
  - Visually realistic
  - Trace the path of light in the scene
- **Computationally expensive**
  - Not suited for real-time rendering (e.g. games)
  - Suited for rendering high quality pictures (e.g. movies)
- **Embarrassingly parallel**
  - Good candidate for **multi-threading**
  - Threads need **not synchronize** with each other, because each thread works on a different pixel

Homework 7: Ray-Tracing

- Trace the path of a ray from eyes
  - One ray per pixel in the view window
  - The color of the ray is the color of corresponding pixel
- Check for intersection of ray
- Lighting
  - Flat shading: the whole object has uniform brightness
  - Lambertian shading: cosine of angle between surface normal and light direction



Recall Pthread API

- pthread_create
- pthread_exit
- pthread_join
- Tip: no need to consider mutex in the homework

Example of pthread_join

```
#include <pthread.h> …
#define NUM_THREADS 5
void *PrintHello(void *thread_ num) {
printf("\n%d: Hello World!\n", (int) thread_num); }
int main() {
pthread_t threads[NUM_THREADS];
int ret, t;
for(t = 0; t < NUM_THREADS; t++) {
printf("Creating thread %d\n", t);
ret = pthread_create(&threads[t], NULL, PrintHello, (void *) t);
// check return value }
for(t = 0; t < NUM_THREADS; t++) {
ret = pthread_join(threads[t], NULL);
// check return value }
}
```

Homework 7: tips

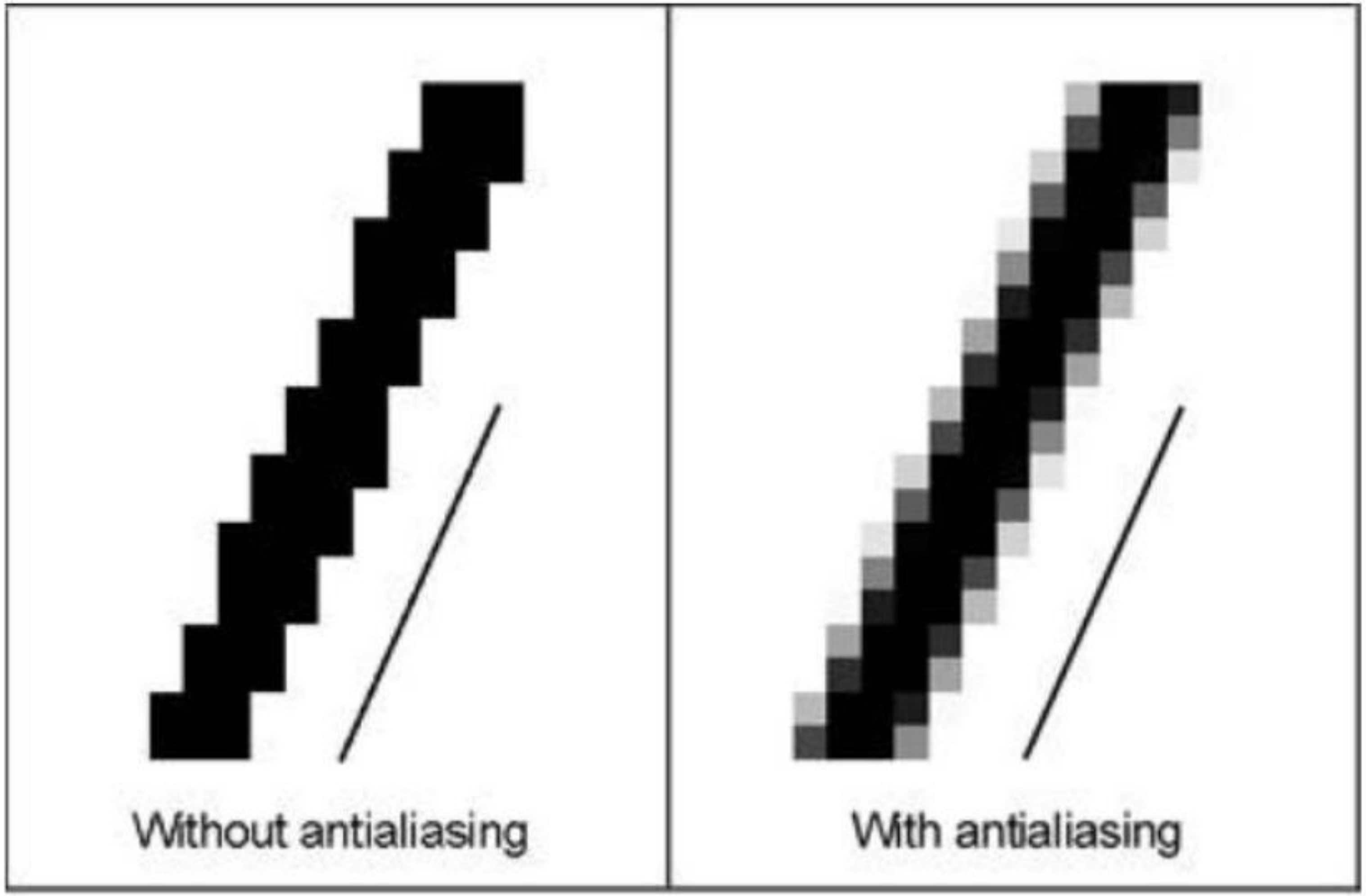
- Download the single-threaded ray tracer implementation
- Run it to get output image
- Multithread ray tracing
    - Modify **main.c** and **Makefile**
- Run the multithreaded version and compare resulting image with single-threaded one

Homework 7: tips

- Build a multi-threaded version of Ray tracer
- Modify “main.c” & “Makefile”
  - Include <pthread.h> in “main.c”
  - Apply “pthread_create” & “pthread_join” in “main.c”
  - Link with –lpthread flag (**LDLIBS target**)
- make clean check
  - Outputs “1-test.ppm”
  - Can see “1-test.ppm”
    - sudo apt-get install gimp (Ubuntu)
    - X forwarding (lnxsrv)
    - gimp 1-test.ppm

1-test.ppm

Homework 7-antialiasing



Homework 7: tips

- Make sure that there is no compile error!
- Read the source code to understand the task
- But do not modify other functions in the original code
- Make sure your submission is a **gzipped file** .tgz
- Key point: how to divide the task to run multiple threads
- Difficulty: the 3rd and 4th arguments of pthread_create function
  - Argument 3: a function that divides the input by threads
  - Argument 4: an array to hold data for each thread