CS 35L- Software Construction Laboratory

Fall 18

TA: Guangyu Zhou

Multithreading/Parallel Processing

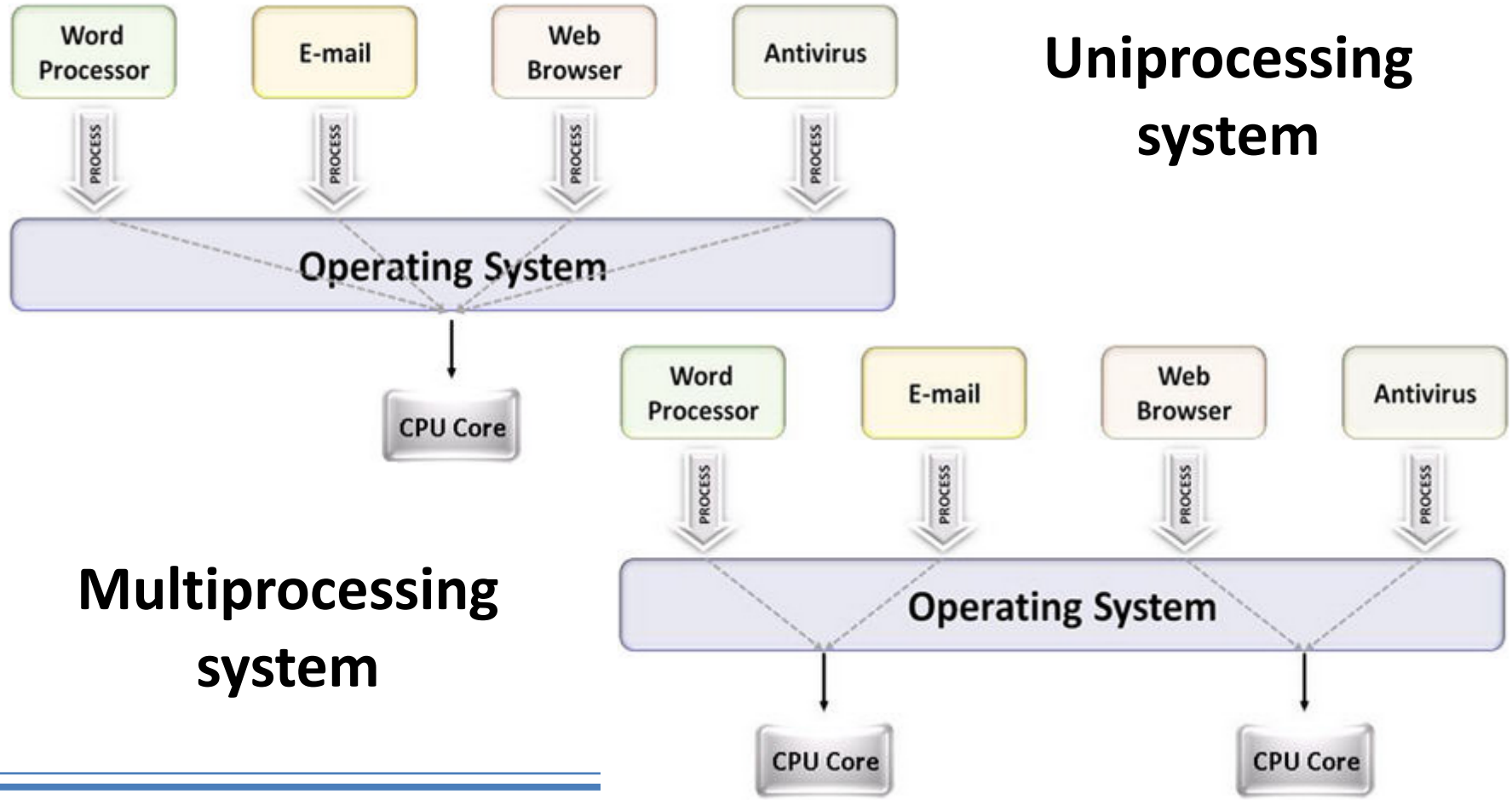
Week 7

Outline

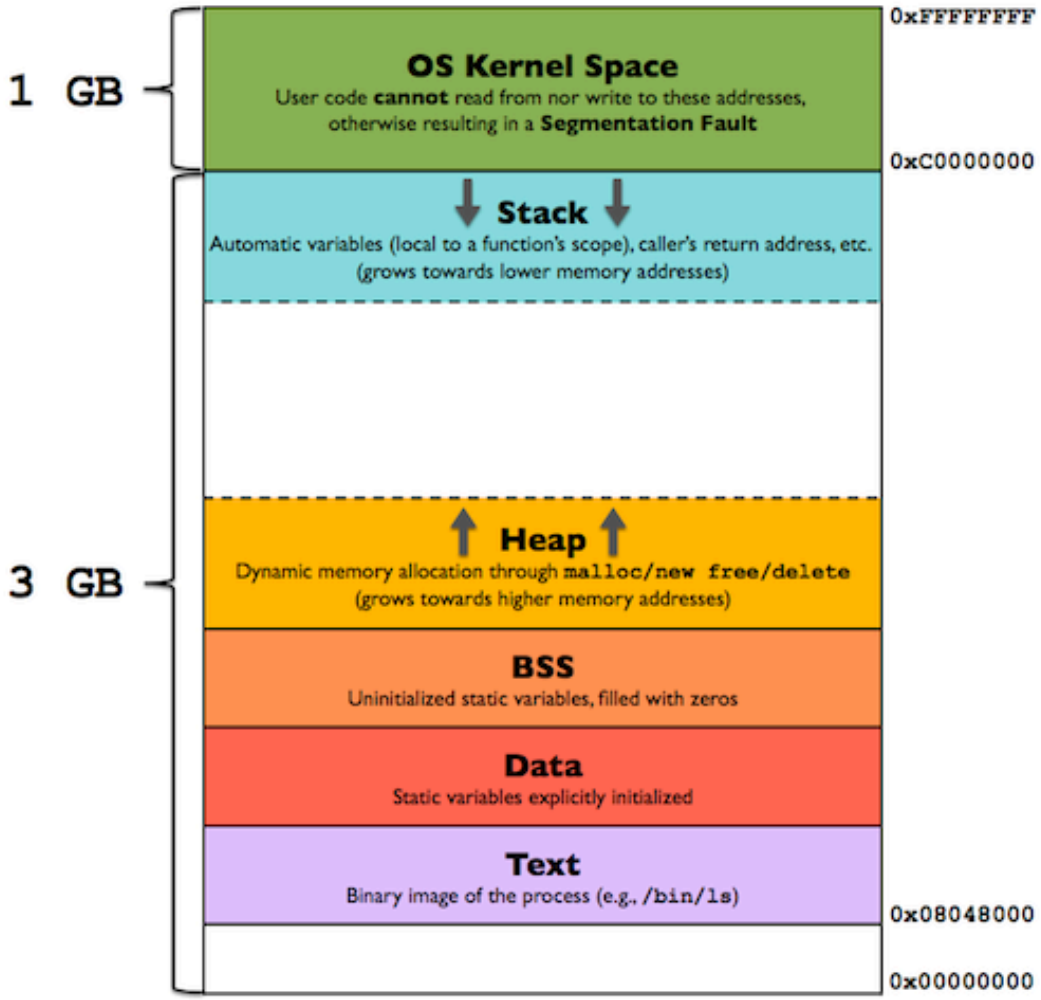
- Multitasking
- **Multi-Thread Processing**
- **Synchronization**
- **Pthread API**

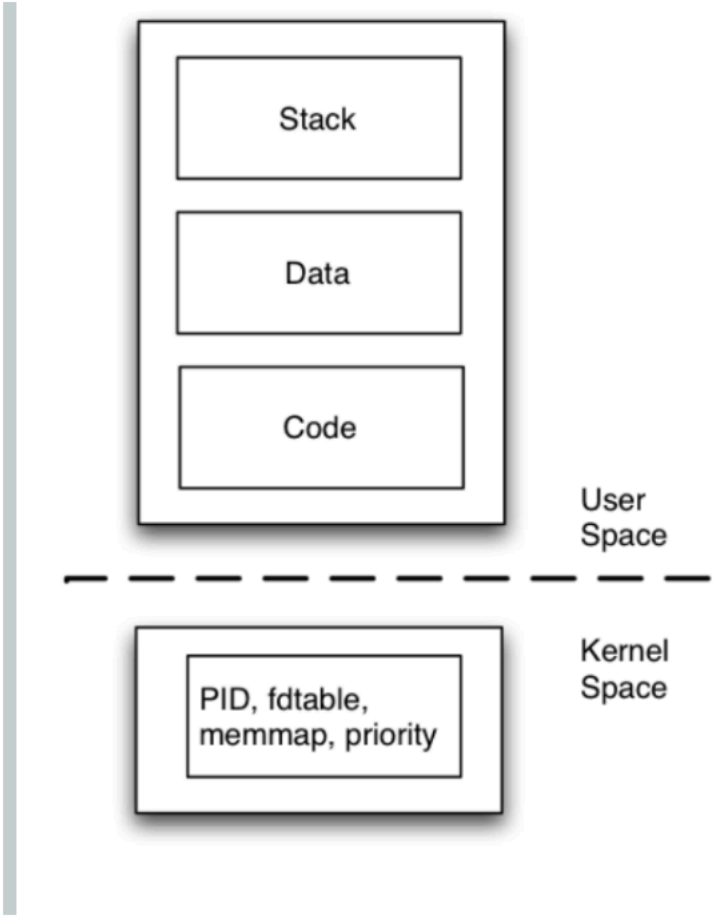
Review: Multitasking, single core vs. multi-core

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



Review: Memory Layout of a process (detailed)





Parallelism

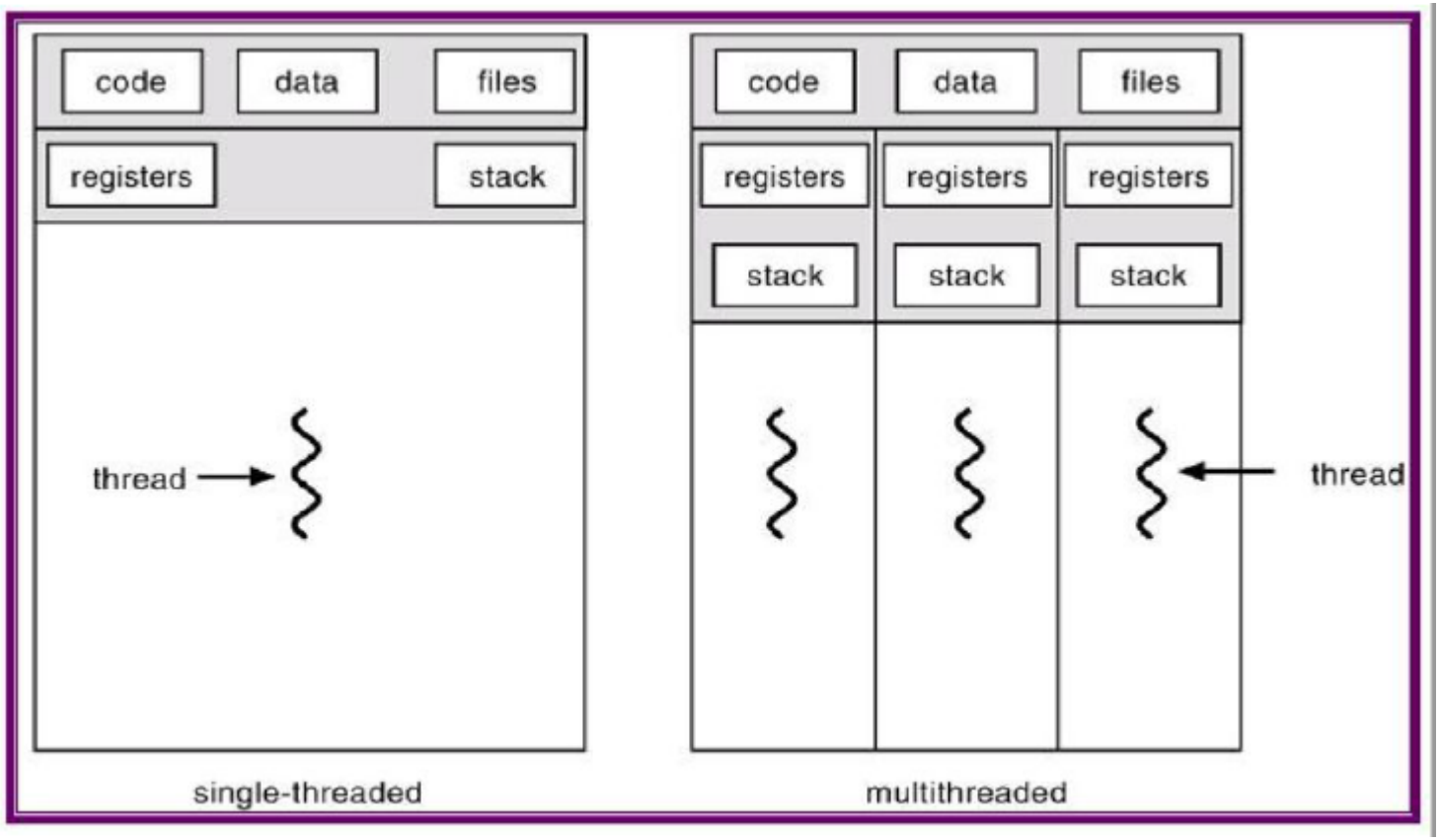
- Executing several computations simultaneously to gain performance
- Different forms of parallelism
  - **Multitasking**
    - Several processes are scheduled alternately or possibly simultaneously on a multiprocessing system
  - **Multithreading**
    - Same job is broken logically into pieces (threads) which may be executed simultaneously on a multiprocessing system

What is a thread?

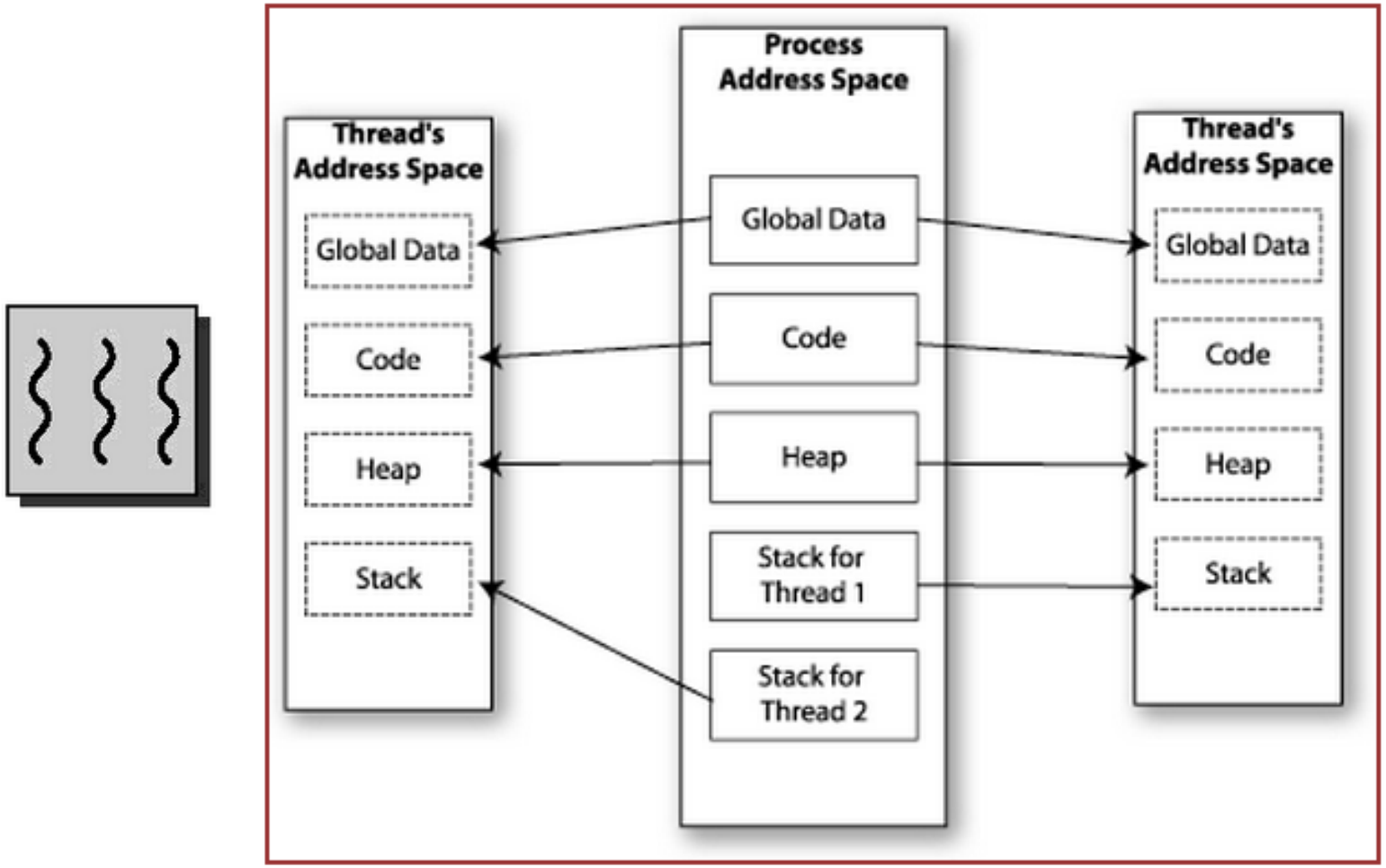
- A flow of instructions, path of execution within a process
- The smallest unit of processing scheduled by OS
- A process consists of at least one thread
- Multiple threads can be run on:
  - **A uniprocessor (time-sharing)**
    - Processor switches between different threads
    - Parallelism is an illusion
  - **A multiprocessor**
    - Multiple processors or cores run the threads at the same time
    - True parallelism

What is a thread?(cont.)

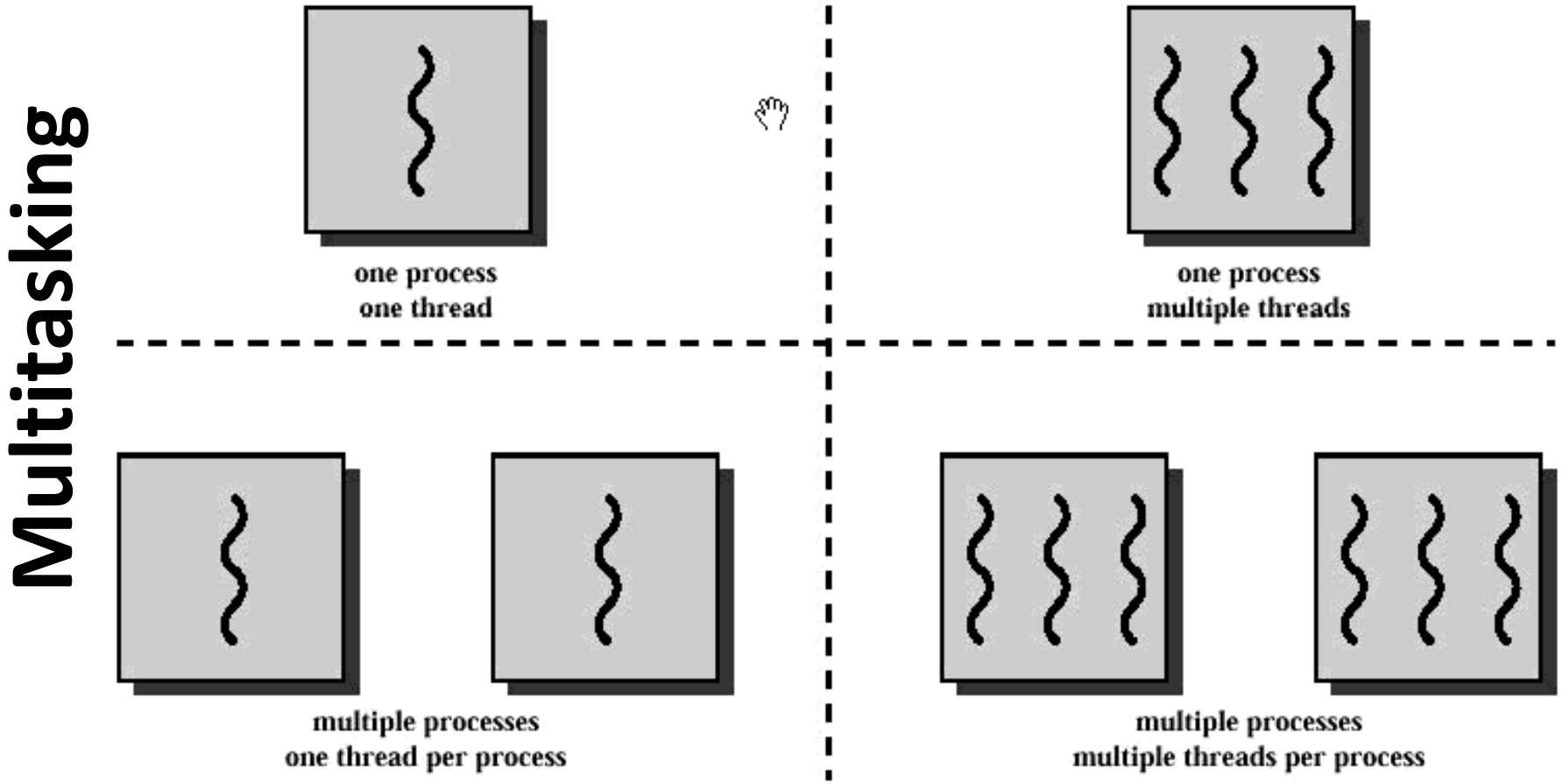
- Each thread has its own:
  - Stack
  - Registers
  - Thread ID
- Each thread shares the following with other threads belonging to the same process:
  - Code
  - Global Data
  - OS resources



Memory Layout: Multithreaded Program



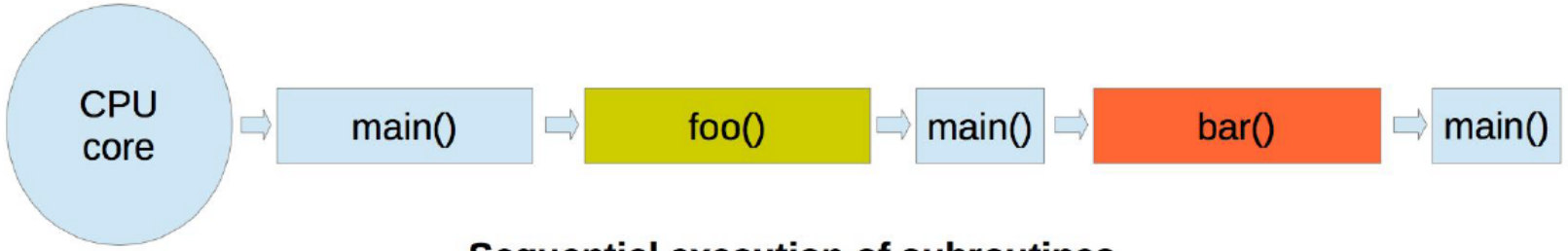
Multithreading



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
}
```

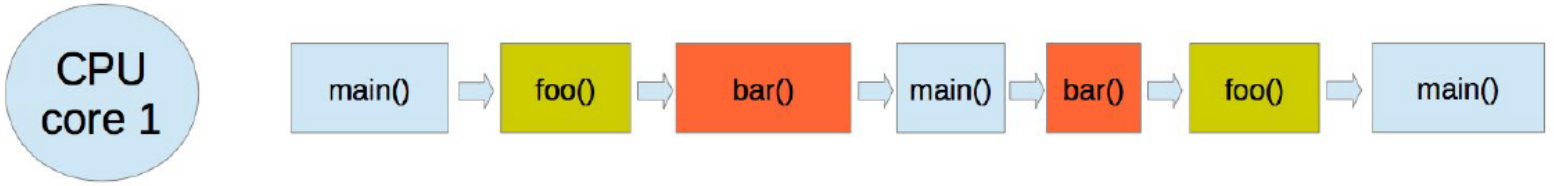


Sequential execution of subroutines

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
}
```

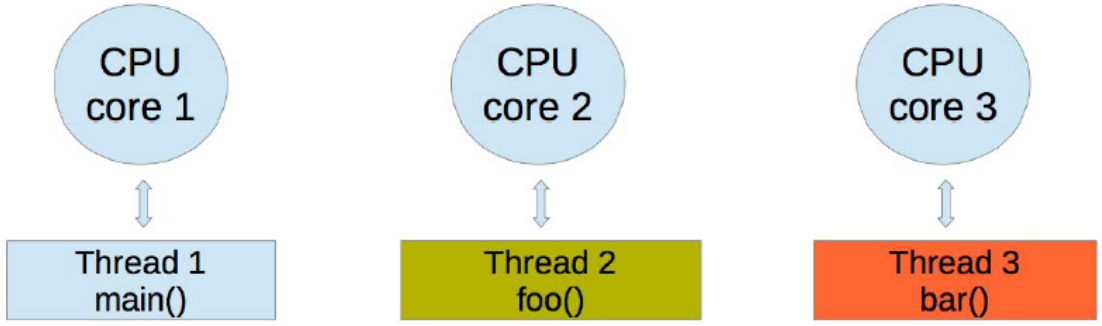


Time Sharing – Illusion of multithreaded parallelism
(Thread switching has less overhead compared to process switching)

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
}
```



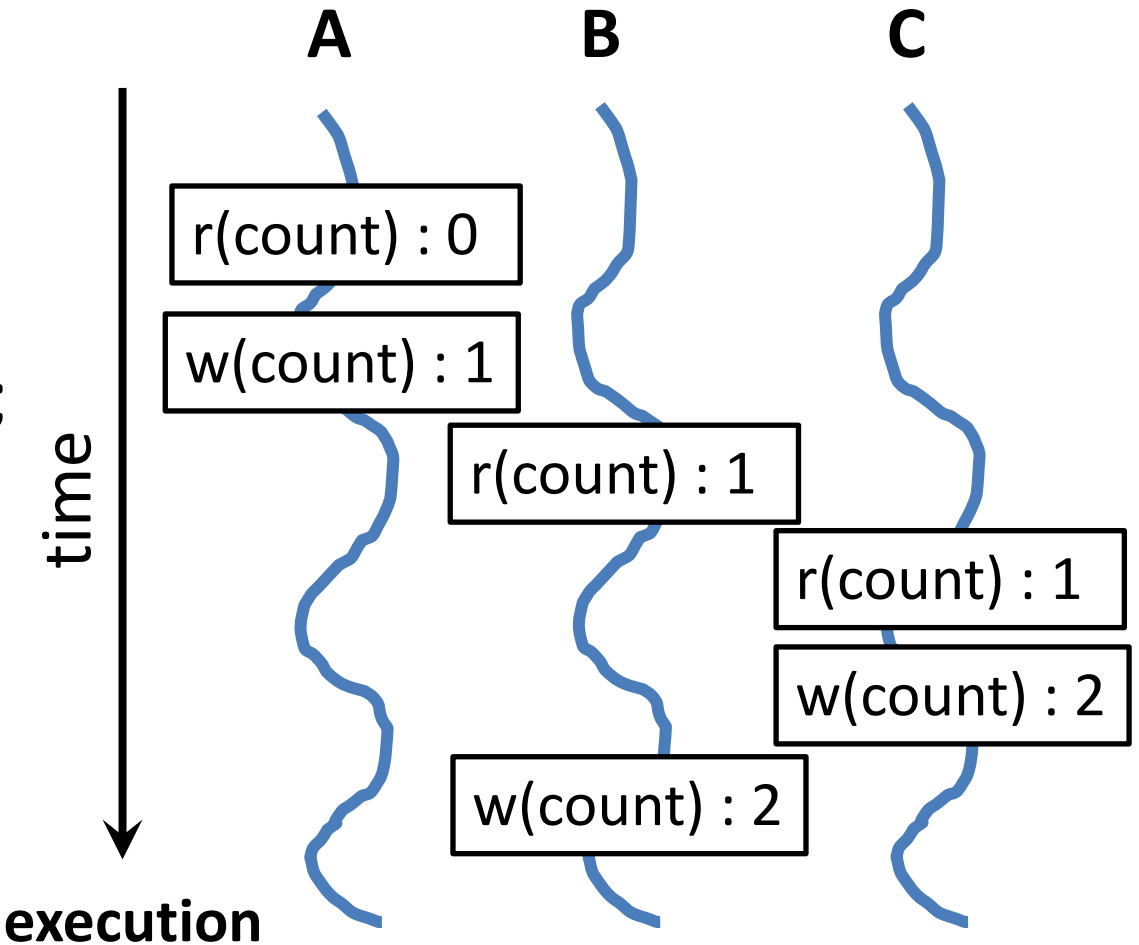
True multithreaded parallelism

Multithreading properties

- Advantages:
  - 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
- Disadvantages?
  - Shared Memory => ***Race condition***
  - Need **synchronization** among threads accessing same data

Race Condition

```
int count = 0;
void increment()
{
  count = count + 1;
}
```



Result depends on order of execution

=> Synchronization needed

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**

Multithreading & Multitasking: Comparison

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

Hints for Lab 6

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 6

- Evaluate the performance of multithreaded ‘sort’ command
- Delete the empty line
- Add /usr/local/cs/bin to PATH (export)
- Generate a file containing 2^24 random **single precision floating point numbers, one per line with no white space**
  - /dev/urandom: pseudo-random number generator

Lab 6

- od
  - Write the contents of its input files to standard output in a user-specified format
  - Options
    - -t fF: single-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 6

- 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

Pthread API

- To use **pthreads** you will need to include <pthread.h> AND you need to compile with **-lpthread** compiler option. This option tells the compiler that your program requires threading support

- There are 5 basic pthread functions:

1. **pthread_create: creates a new thread within a process**
2. **pthread_join: waits for another thread to terminate**
3. pthread_exit: terminates the currently running thread
4. pthread_equal: compares thread ids to see if they refer to the same thread
5. pthread_self: returns the id of the calling thread

pthread_create

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

- Function: creates a new thread and makes it executable
- Can be called any number of times from anywhere within code
- Return value:
  - Success: zero
  - Failure: error number
- **PTHREAD_THREADS_MAX:** Constant for max number of threads that can be created

Parameters

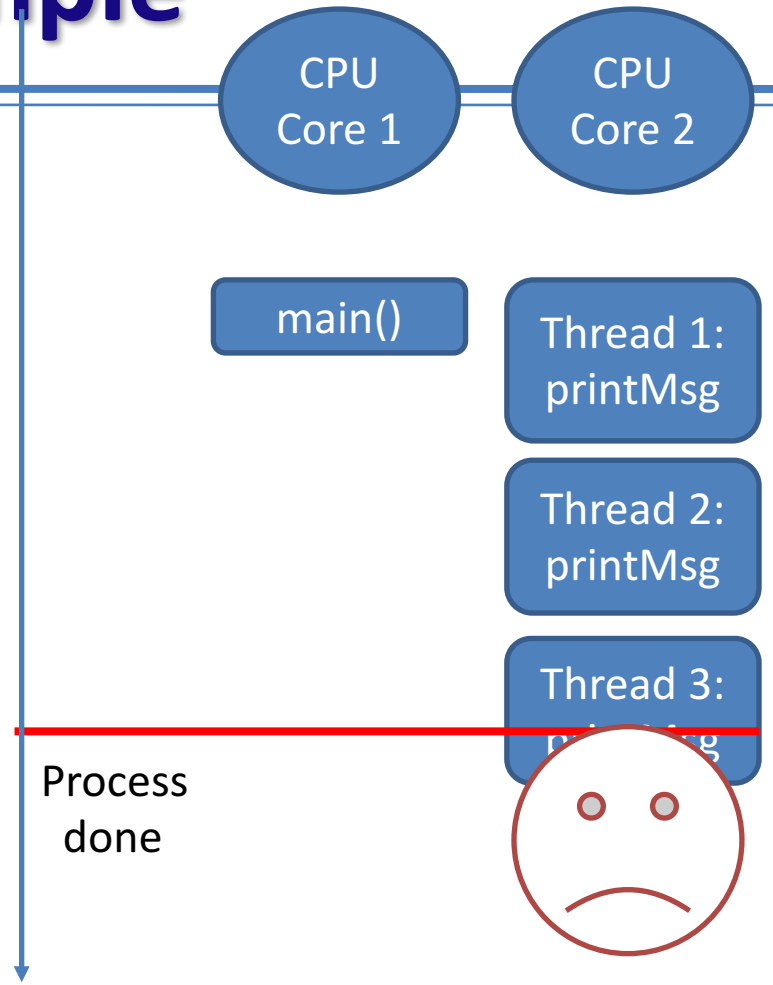
- int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(my_function)(void *), void *arg);
  - tid: unique identifier for newly created thread
  - attr: object that holds thread attributes (priority, stack size, etc.)
    - Pass in NULL for default attributes
  - my_function: function that thread will execute once it is created
  - arg: a single argument that may be passed to my_function
    - Pass in NULL if no arguments

Question

- If I call **pthread_create** twice, at least how many stacks does your process have?
- A. 1
- B. 2
- C. 3
- Your process will contain three stacks - one for each thread. The first thread is created when the process starts, and you created two more. Actually there can be more stacks than this, but let's ignore that complication for now.

pthread_create Example

```
1.  #include <pthread.h> …
2.  void *printMsg(void *thread_num) {
3.      int t_num = (int) thread_num;
4.      printf("It's me, thread #%d!\n", t_num); }
5.
6.  int main() {
7.      pthread_t tids[3];
8.      int t;
9.      for(t = 0; t < 3; t++) {
10.         ret = pthread_create(&tids[t], NULL, printMsg, (void *) t);
11.         if(ret) {
12.             printf("Error creating thread. Error code is %d\n", ret");
13.             exit(-1); }
14.     }
15. }
```



Possible problem with this code?

If main thread finishes before all threads finish their job -> incorrect results

pthread_join

- Function: makes originating thread wait for the completion of all its spawned threads' tasks
- Without join, the originating thread would exit as soon as it completes its job
  - A spawned thread can get aborted even if it is in the middle of its chore
- Return value:
  - Success: zero
  - Failure: error number

Arguments

- int pthread_join(pthread_t tid, void **status);
- tid: thread ID of thread to wait on
- status: the exit status of the target thread is stored in the location pointed to by *status
  - Pass in NULL if no status is needed

What is the purpose of pthread_join?

- Wait for a thread to finish
- Clean up thread resources
- Grabs the return value of the thread

pthread_join Example

```
1.  #include <pthread.h> …
2.  #define NUM_THREADS 5
3.
4.  void *PrintHello(void *thread_ num) {
5.          printf("\n%d: Hello World!\n", (int) thread_num); }
6.
7.  int main() {
8.          pthread_t threads[NUM_THREADS];
9.          int ret, t;
10.         for(t = 0; t < NUM_THREADS; t++) {
11.                 printf("Creating thread %d\n", t);
12.                 ret = pthread_create(&threads[t], NULL, PrintHello, (void *) t);
13.          }
14.
15.         for(t = 0; t < NUM_THREADS; t++) {
16.                 ret =  pthread_join(threads[t], NULL);
17.         }
18. }
```

pthread_exit

- pthread_exit(void *) only stops the **calling** thread i.e. the thread never returns after calling pthread_exit.
- The pthread library will automatically finish the process if there are no other threads running.

pthread_join vs. pthread_exit

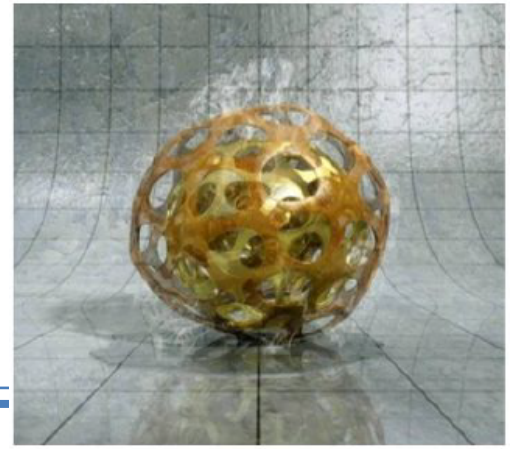
- Both pthread_exit and pthread_join will let the other threads finish on their own (even if called in the main thread).
- However, only pthread_join will return to you when the specified thread finishes. pthread_exit does not wait and will immediately end your thread and give you no chance to continue executing.

Hints for Assignment 6

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)

Homework 6: Ray-Tracing

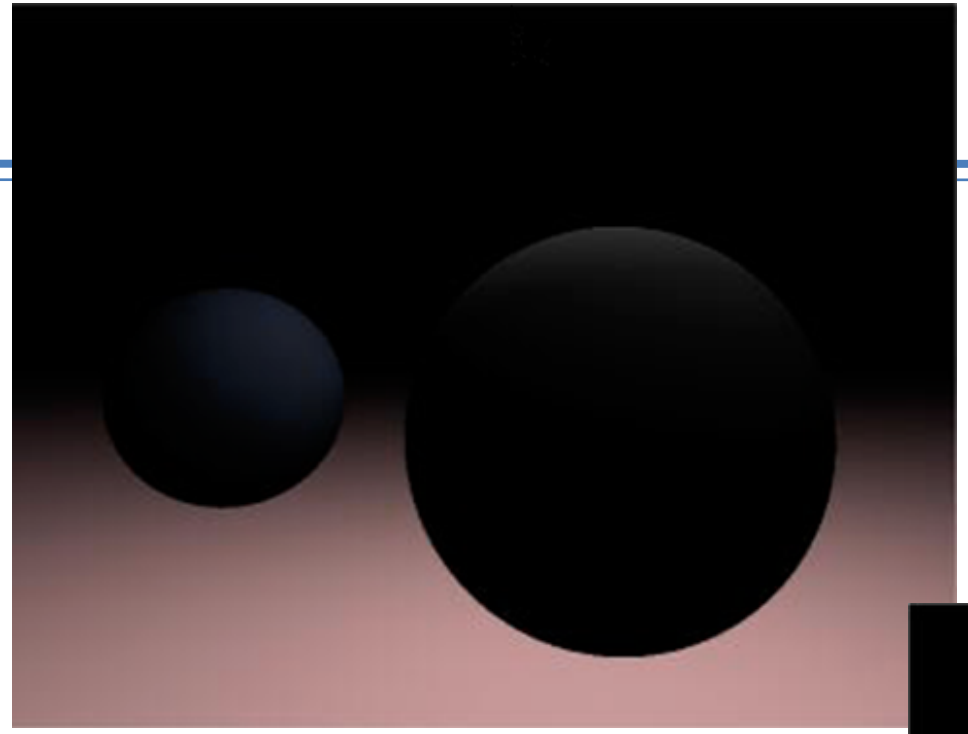


Homework 6: Motivation

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

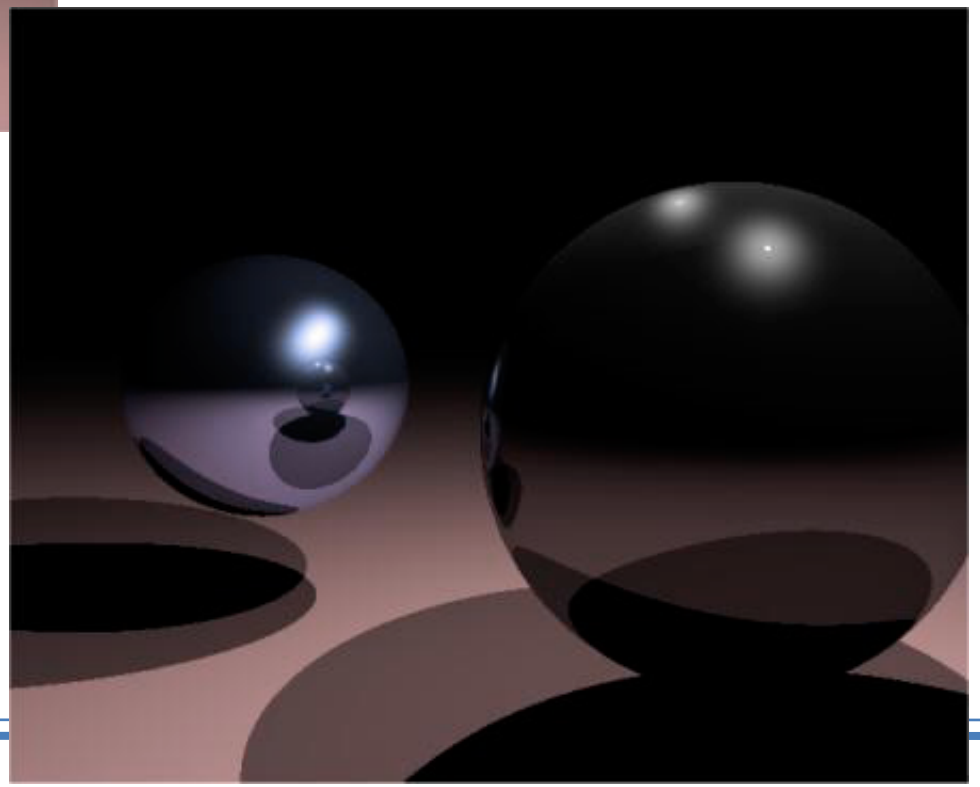
Homework 6: Ray-Tracing

- **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



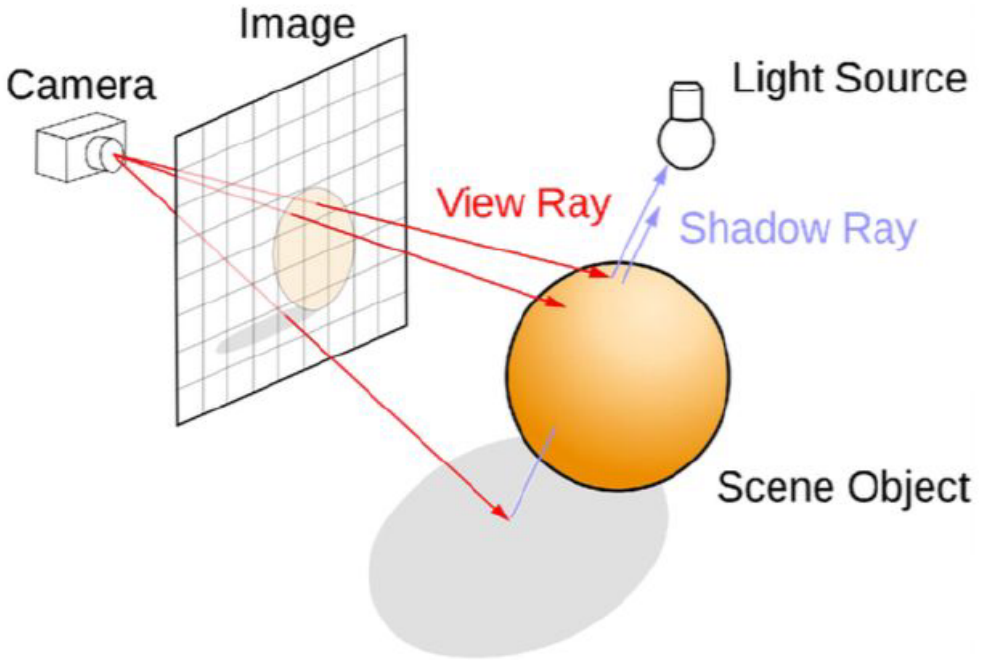
Without ray tracing

With ray tracing



Homework 6: 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_join
- Tip: no need to consider synchronization issue (mutex) in the homework

Homework 6: 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 6: 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

baseline.ppm

Homework 6: 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
- **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

Questions?