Discussion 1

Aniket Shivam

Quick reminder

- Lab Assignment 1 is posted!

- Office Hours:
  - Both Lab Sections are treated as Office Hours.
  - Via Zoom
  - Plus post questions on Piazza

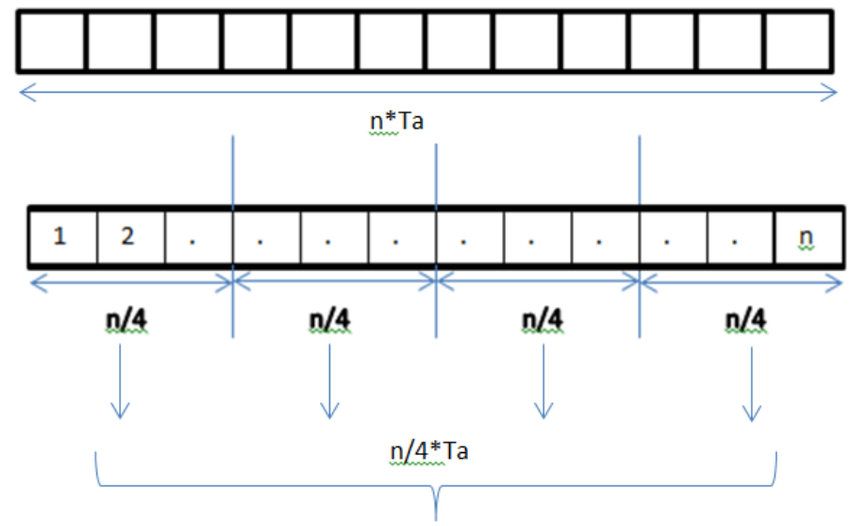
Parallelism

- Data Parallelism
  - Same task(code) on different data
    - Ex: Find a word in a document.
- Task Parallelism
  - Different task(code) on same/different data
  - Ex: Calculate Mean, Median, Mode of some data

Data Parallelism

- Addition of two vectors:

for i = **1** to **N**
    S: A[i] = B[i] + C[i]
end for



- Workload balancing and Data Shared
- Machine: Multi-core architecture with 4 cores
- Approach: Create 4 threads, divide dataset into 4 (No shared memory).
- SIMD(Single Instruction Multiple Data)

Task Parallelism

- Four tasks to carry out on same dataset

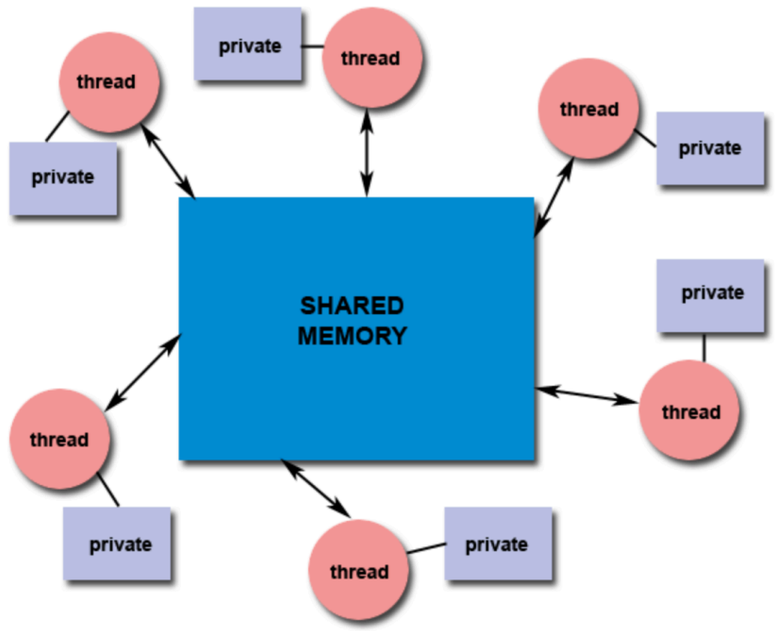
- Data may be shared/copied between threads

- MIMD(Multiple Instruction Multiple Data)
- MISD(Multiple Instruction Single Data)

```
Create 4 threads
If thread # == 1:
    Execute Task 1
If thread # == 2:
    Execute Task 2
If thread # == 3:
    Execute Task 3
If thread # == 4:
    Execute Task 4
end if
```

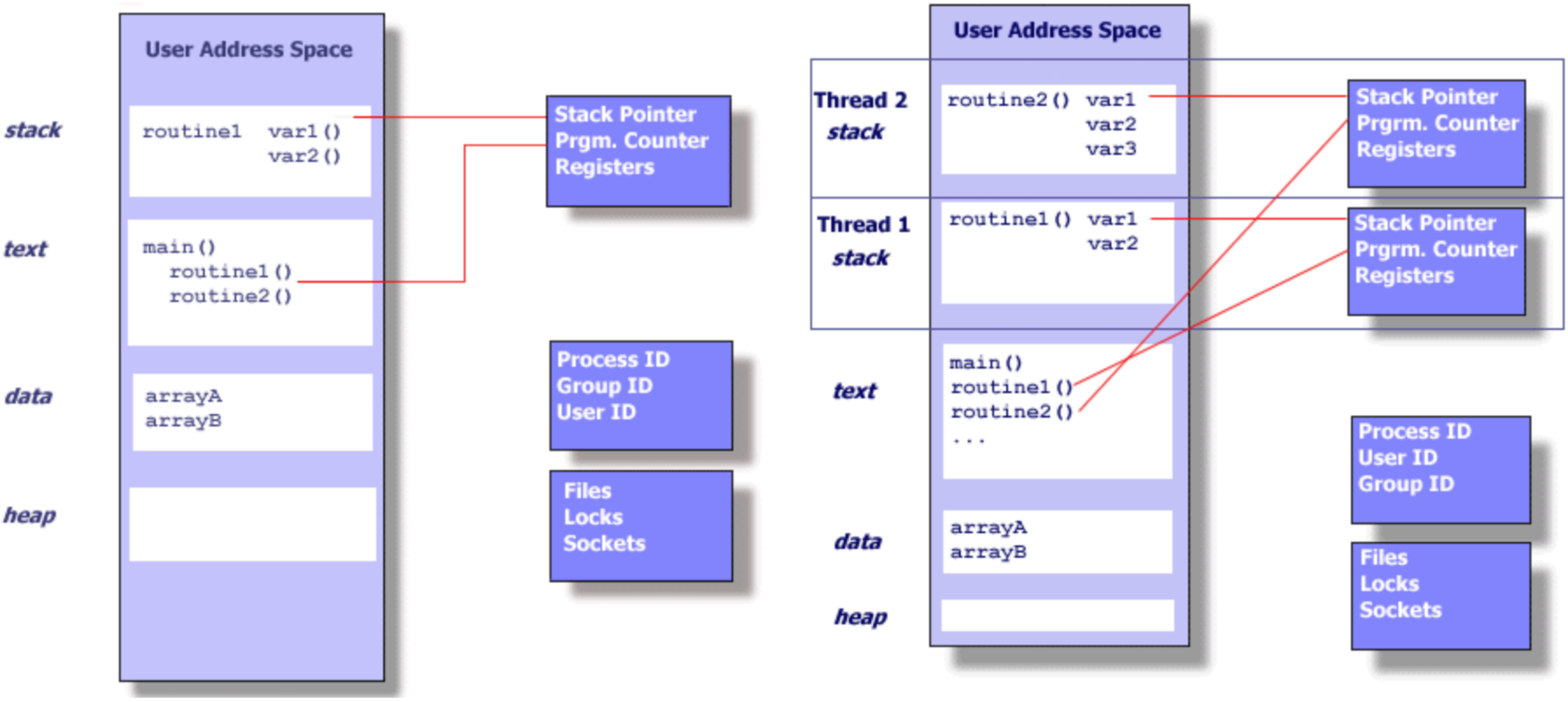
Scope Rules

- Each thread has both shared and private variables.
- Shared global variables
- Private copies of local variables



- Need for mutex(if writing global variables).
- In C/C++:
  - Global: outside all functions.
  - Local: Inside a function or block({…})

Memory Management



Labs related Info

- Before submitting code, please test on Openlab.
- Compile using flags suggested in the assignment.
- Submit on Canvas
- Only submit essential files (one file per problem)

Login

- ssh to <username>@openlab.ics.uci.edu
- From inside the campus (or housing)
- From outside use the VPN
  - http://www.lib.uci.edu/connect-campus
- You will be assigned to any machine that is available

Modules

- Openlab uses the SGE module system
- When you login you only have the basic things
- If you need something you must load the module and it will be available
- More info
  https://www.ics.uci.edu/computing/linux/modules.php

Using modules

- See what modules are available

```
$ module avail
```

- Load a module

```
$ module load <name>
   Use: gcc/8.2.0
```

C++ threads

- Exist since the C++11 standard came out
- Class that represents a thread of execution
- Is part of the Thread support library
- Underneath the hood uses Pthreads

Creation

- `std::thread` defined in the header `<thread>`
  - Receives a function that will be executed in a different thread
  - Function must return **`void`**

Example

```
#include <thread>
void
someFunction()
{
    // Function code ...
}
int
main()
{
    // Some code ...
    std::thread t(someFunction);
    // Main code ....

    return 0;
}
```

Two threads

- You now have two threads
  - Thread `main` executes the rest of main
  - Thread `t` executes some function

Just creating threads is not enough!

- The created thread may finish after the main thread
- The operating system will throw an error
- How do we wait for the thread to finish?
- You must wait for the other threads before finishing the main thread

Use `join` to wait for a thread

```
#include <thread>
void
someFunction()
{
    // Function code ...
}
int
main()
{
    // Some code ...
    std::thread t(someFunction);
    // Main code ....

    t.join();

    return 0;
}
```

How to pass arguments

- Arguments are passed as part of the creation process

```
void
someFunction(int a)
{
    // Does something with a
}

int
main()
{
    int mainA;
    std::thread t(someFunction, mainA);
    return 0;
}
```

Passing references doesn't work!

```
void
someFunction(int& a)
{
    // Does something with a
}

int
main()
{
    int mainA;
    std::thread t(someFunction, mainA);
    return 0;
}
```

You have to use std::ref

```
void
someFunction(int& a)
{
    // Does something with a
}

int
main()
{
    int mainA;
    std::thread t(someFunction, std::ref(mainA));
    return 0;
}
```

Vectors of threads

- To create a vector of threads is simple
- There are two equivalent options
  - push_back(std::thread(funcNam, arg...))
  - emplace_back(func_name, args...
- Remember the threads will start as soon as you create them!

Example of vector push_back

```
void
someFunction(int a)
{
    // Does something with a
}

int
main()
{
    std::vector<std::thread> threads;
    for(int i = 0; i < 10; ++i)
       threads.push_back(std::thread(someFunction, i));
    for(int i = 0; i < 10; ++i)
       threads[i].join();
    return 0;
}
```

Example of vector emplace_back

```
void
someFunction(int a)
{
    // Does something with a
}

int
main()
{
    std::vector<std::thread> threads;
    for(int i = 0; i < 10; ++i)
       threads.emplace_back(someFunction, i);
    for(int i = 0; i < 10; ++i)
       threads[i].join();
    return 0;
}
```

C++ Atomic

- Each instantiation and full specialization of the std::atomic template defines an atomic type.
- If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined.
- Use hardware support to perform operations.

- Check out links in Lab Document for more details

Atomic Compare and Exchange/Swap

- Compare-And-Swap (CAS) used to achieve synchronization.
- C++11 supports this atomic operation on language level to help us write portable multithreaded code that run on all platforms that are standard-compliant.
- It loads the value from memory, compares it with an expected value, and if equal, store a predefined desired value to that memory location.
- All these are performed in an atomic way, i.e., if the value has been changed in the meanwhile by another thread, CAS will fail.
- On x86, performed using a single *cmpxchg* instruction.

Atomic Compare and Exchange/Swap

```
bool compare_exchange_weak( T& expected, T desired,
                            std::memory_order order =
                                std::memory_order_seq_cst ) noexcept;

bool compare_exchange_strong( T& expected, T desired,
                              std::memory_order order =
                                  std::memory_order_seq_cst ) noexcept;
```

Parameters

expected - reference to the value expected to be found in the atomic object.

Gets stored with the actual value of *this if the comparison fails. (NOTE!)

desired - the value to store in the atomic object if it is as expected

order - the memory synchronization ordering for both operations

Return value

`true` if the underlying atomic value was successfully changed, `false` otherwise.

Atomic Compare and Exchange/Swap

- For the dining philosopher problem, we use to get hold of the forks.

- Use ***compare_exchange_weak*** to occupy the forks one at a time. (Strong version is fine too, for x86 its the same.)

  bool first_fork = false;
  if((phil.fork).compare_exchange_weak(first_fork, true)) {…..}

  - Use it inside a ***while*** loop, since it will fail a lot.

  - Two forks = Two CAS, but just one while loop.

    - Remember to drop the first fork, if thread can't get the second fork.

Mutual exclusion

- To enable serial access to data/streams we have to use mutual exclusion
- We will see three ways:
  - mutex
  - lock_guard
  - unique_lock (we won't use it)

Mutex

- A primitive object to ensure mutual exclusion
- Basic usage

```
std::mutex m;
int sharedData;

while(!m.try_lock())
{
    // manipulate shared data
    sharedData++;
    m.unlock();
}
```

Problems

- What happens if you forget to unlock?
- What happens if there is an exception in the critical section?
- Not recommended to use it in this direct manner

lock_guard

- A better solution than mutex
- An object that holds the mutex and locks it
- Will hold the lock while it is in the scope
- When it is destroyed it releases the lock
- No way to hold the lock forever

lock_guard usage

```
std::mutex m;
int sharedData;

{
    std::lock_guard<std::mutex> lock(m)
    // manipulate shared data
    sharedData++;
} // mutex is released here
```

Compiling C++11 Threads

- Load module
  - Use the '`-std=c++11`' and '-pthread' flags
  - Add '-Wall' to see warnings

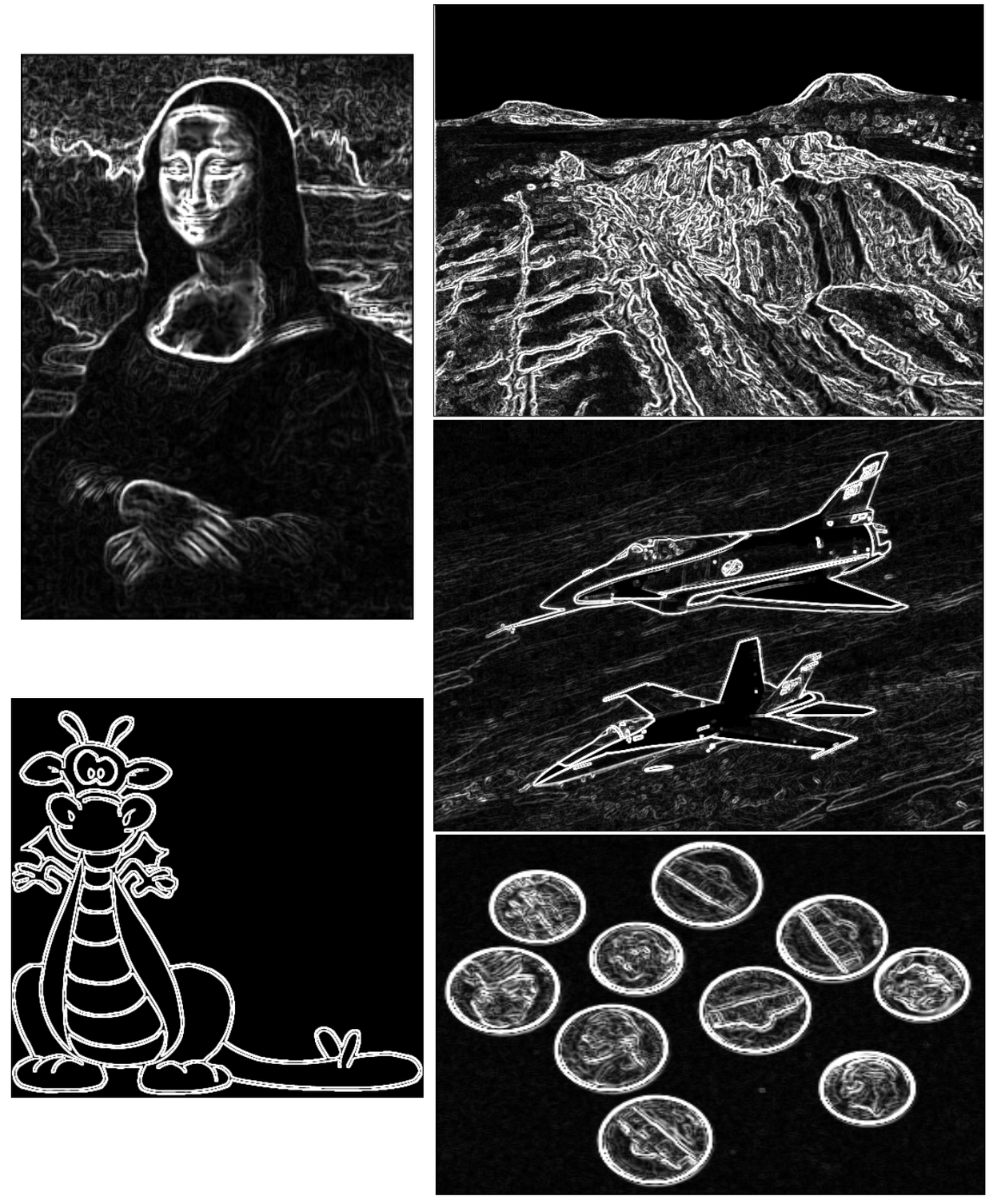
```
$ g++ -std=c++11 -pthread example.cpp -o Example
```

Comments

- We need the shared variables (nextAvailableChunk, outputImage)
- It is better to separate into a function where you are going to update it
    - Assure access is mutually exclusive
- Maybe reduce conflict while writing the image
    - Create temporary partial output image.
    - Reduce chances of Race condition, if error in implementation

Should look like this!

- Solution files are provided for reference.
- Use Linux ***diff*** to check if your solution is correct.
- Usual mistakes:
  - Missing rows around chunks



Implementation

- If you want to change something in skeleton?
    - You can. But do not change CL args structure.
- Seeing some extra variables or structs or globals?
    - Ignore them if you don't need them.

Grading

- 50 pts for each implementation
- 0 pts, if program doesn't even compile.
    - Before submission, check if the code compiles and works on ***openlab*** server.
- 0 pts, if it just compiles but does nothing useful at all.
- Partial grading, only if there is a minor bug/design error.

Questions?

You can post on Piazza
Or
Attend Office hours

Discussion Slides will be posted on Class Website