CS 33: Computer Organization

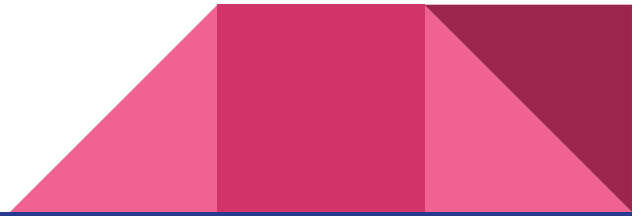
Dis 1B: Week 8 Discussion

Agenda

- Performance Lab
- Race Condition
- Deadlock

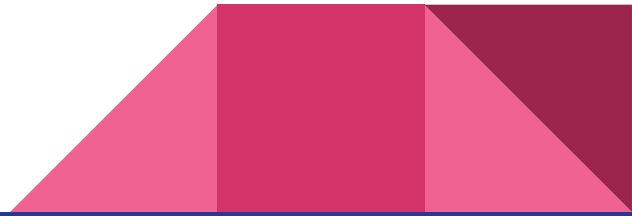
Performance Lab

- Last lab of the quarter!
- Should be the easiest lab of the quarter
- Or… is it?



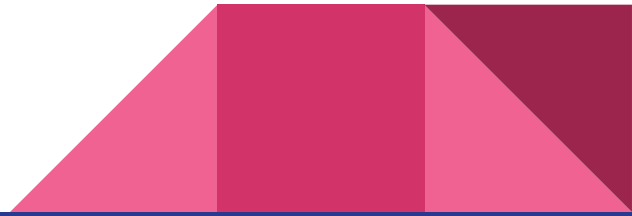
Optimization

- Single-threaded optimization
- Multithreading
- You will need both to reach 6x optimization!



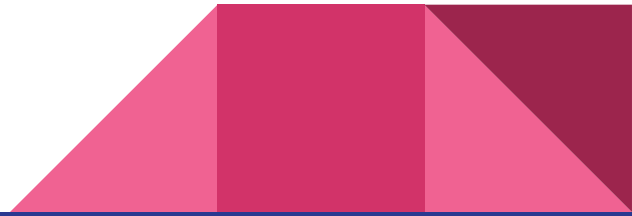
Single-threaded Optimization

- Week 5 Lecture Slides on Optimization
- Week 6 Discussion Slides on Optimization



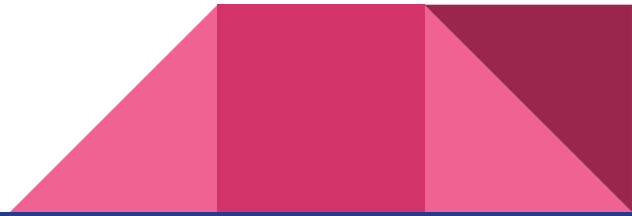
Multithreading

- Week 7 Discussion Slides
- Today’s Discussion



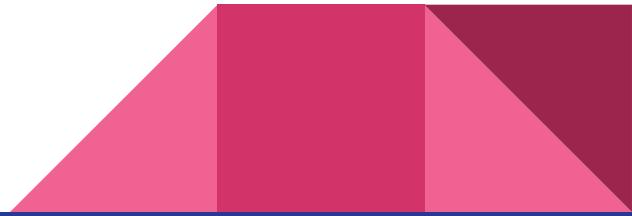
Good News

- You won’t have to worry about synchronization (deadlock, mutex, semaphore, etc.) for this lab



Getting the Lab

- Login to lnxsrv06.seas.ucla.edu
- mkdir perflab
- cd perflab
- cp /w/class.1/cs/cs33/cs33w17/perflab-handout.tar ./perflab-handout.tar
- tar -xvf perflab-handout.tar

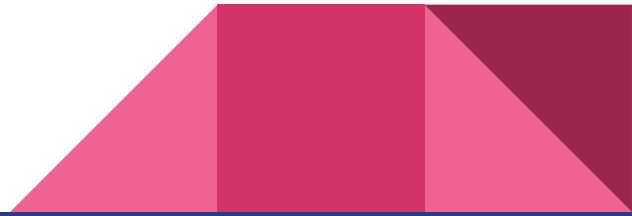


Lab Logistics

- Your code MUST compile on lnxsrv06. Otherwise, you will get a zero
  - Warnings are okay for this lab
  - Beware, even if your code works on lnxsrv02, it might not work on lnxsrv06
  - So work on lnxsrv06
- We are going to grade everything using “driver” file

Grading

- If your code is at least 6x faster than the original code, you will get a full credit
- If not, you will get a grade = (your speedup - 1) / 5 * 100%
- Top 3 solutions will get 20% bonus points
- 4th ~ 10th solutions will get 10% bonus points



What you need to do to get an A

- The only file that you need to modify is kernel.c
- Fill out team_t team struct with your UID, full name, e-mail address
- Modify smooth function
  - You are only going to modify smooth function
  - Don’t worry about rotate
  - kernels.c will have a comment that says “IGNORE EVERYTHING AFTER THIS POINT!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!111”
  - If you see that comment, please ignore everything that comes after that comment

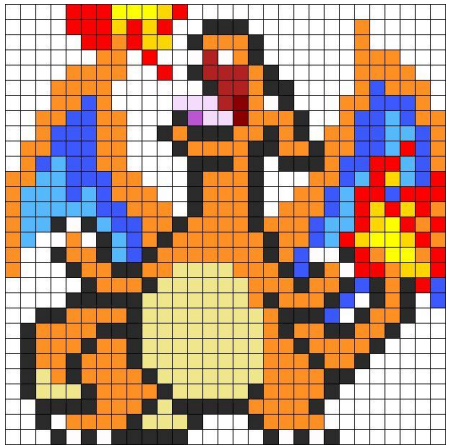
What you are actually doing

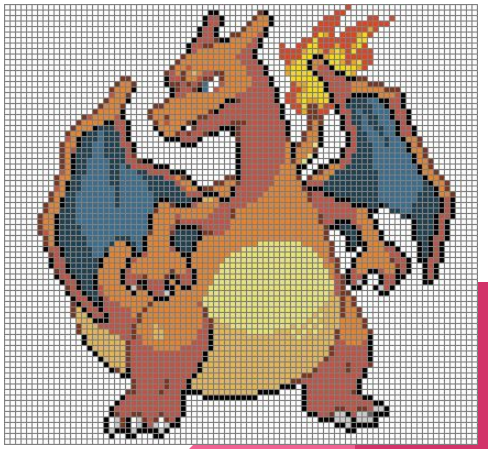
- Image processing!
- You are going to optimize a function that blurs an image

Image Blurring

How an Image is Represented

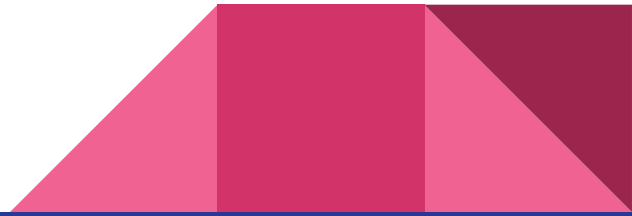
- Image is represented as a 2D array of pixels
- Pixel values are a triplet of (Red, Green, Blue)
- Bigger pixels (image size) = Better pictures!





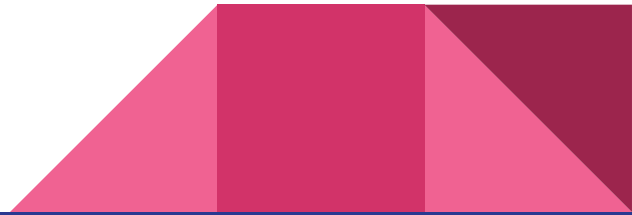
Bigger Pixels, Bigger Problems?

- If the pixels are bigger, it takes up more space
- Since it takes up more space, operation on a bigger image is usually slower
- A lot of applications (Facebook, Messenger, etc.) that are sensitive to users' internet speed compresses images for that reason



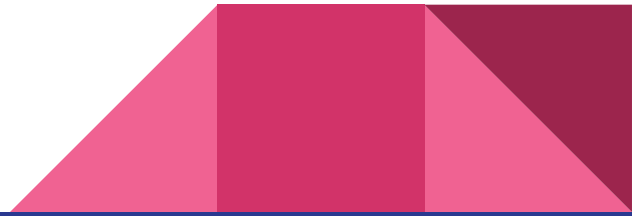
Performance Lab Hints

- READ THE HANDOUT CAREFULLY
  - There are so many hints on the handout itself, just like the previous labs
  - We are literally spoiling you guys by telling exactly what you need to do…
  - Wait until you take CS 35L or CS 111 then you'll understand
- SEASNET servers have 8 cores
- Use a different number of threads for each stage
  - For smaller images, creating threads might create overhead instead of speed-up
- Divide up the problem
  - Currently, 1 function does all the work. Is that efficient?
  - Locality?



Quick Review: Threads

- How to create a thread:
  - int pthread_create(pthread_t *tid, pthread_attr_t const *attr, func *f, void *args)
    - The argument tid is a thread id that is a pointer to a pthread_t passed in and assigned when the thread is created. A pthread_t is essentially a number
    - attr specifies options. By default, 0
    - This creates a thread, assigns the thread id to tid. When the thread starts, it will call f(args)

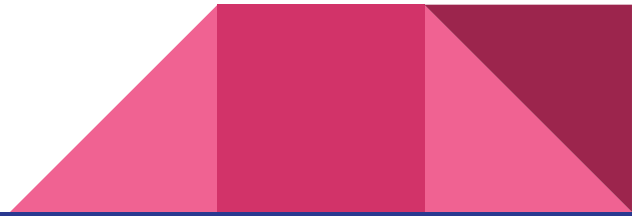


Quick Review: Threads

- int pthread_join(pthread_t tid, void ** thread_return)
  - One thread waits for thread tid to complete before continuing

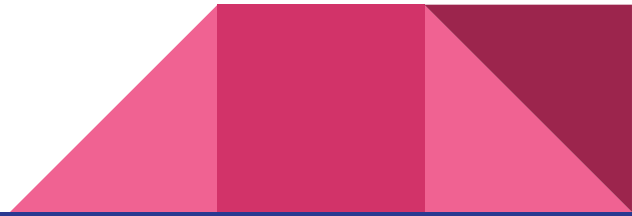
Creating threads that works on the same function

```
pthread_t threads[NUM_THREADS];

void (*funptr) (int);

funptr = &someFunctionThatTakesInAnInt;

for (i = 0; i < NUM_THREADS; i++) {

        pthread_create(&threads[i], NULL, funptr, (void*) i);

}

for (i = 0; i < NUM_THREADS; i++) {

        pthread_join(threads[i], NULL);

}
```



What if My Function takes in Multiple Arguments?

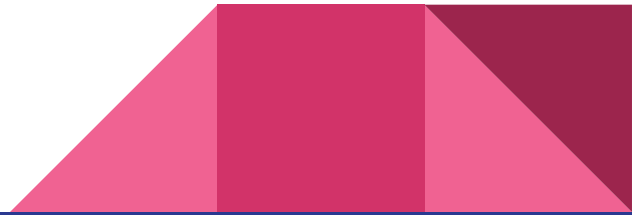
- Change your function to take in a single argument
  - Simple and easy. Right?
  - Wait… what?



What if My Function takes in Multiple Arguments?

- Change your function to take in a single argument
  - Simple and easy. Right?
  - Wait… what?
- Change your function to take in a pointer to a struct of argument

```
typedef struct {
   int i1;
   int i2;
   char* s;
} fun_args;

fun_args* args = malloc(sizeof(fun_args));
args->i1 = i1;
args->i2 = i2;
args->s = s;
pthread_create(&thread, NULL, funptr, (void*) args);
```

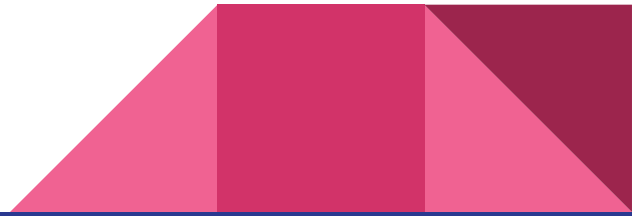


Key Idea

- Divide up the problem
- Create multiple threads and keep their thread IDs
- Let each thread work on independent subset of the entire problem
- Join them all using the thread IDs that we saved

Join

- If you do not call pthread_join on all of your threads, your code will work sometimes
- However, your code will also fail sometimes
- Why would this happen?



Accounting 101

- DJ currently has $1,000 in the bank
- DJ had stock shares of Snap that he bought last month for $500 and sold all of his shares today for $30,000
- DJ paid a rent of $5,000 today for a studio in Westwood
- How much money should DJ have in his account after these series of events?

Accounting 101

- However, after all these events happened, DJ’s account is in deficit of $4,000 and is charged with bank service fee of $100,000
- DJ is now bankrupt
- WTF happened?

Race Condition

- Suppose a USC alumni programmed the following code for DJ's bank to update someone's bank account

```
bool updateBankAccount(int accountNumber, int transactionType, int amount) {
    // withdrawal
    if (transactionType == 0)
    {
        amountInAccount[accountNumber] -= amount;
    }
    else if (transactionType == 1)
    {
        amountInAccount[accountNumber] += amount;
    }
    else
        return false;
    return true;
}
```

How to prevent Race Condition

- Use locks!

```
bool updateBankAccount(int accountNumber, int transactionType, int amount) {
   // withdrawal
   if (transactionType == 0)
   {
      lock(amountInAccount[accountNumber]);
      amountInAccount[accountNumber] -= amount;
      unlock(amountInAccount[accountNumber]);
   }
   else if (transactionType == 1)
   {
      lock(amountInAccount[accountNumber]);
      amountInAccount[accountNumber] += amount;
      unlock(amountInAccount[accountNumber]);
   }
   else
      return false;
   return true;
}
```

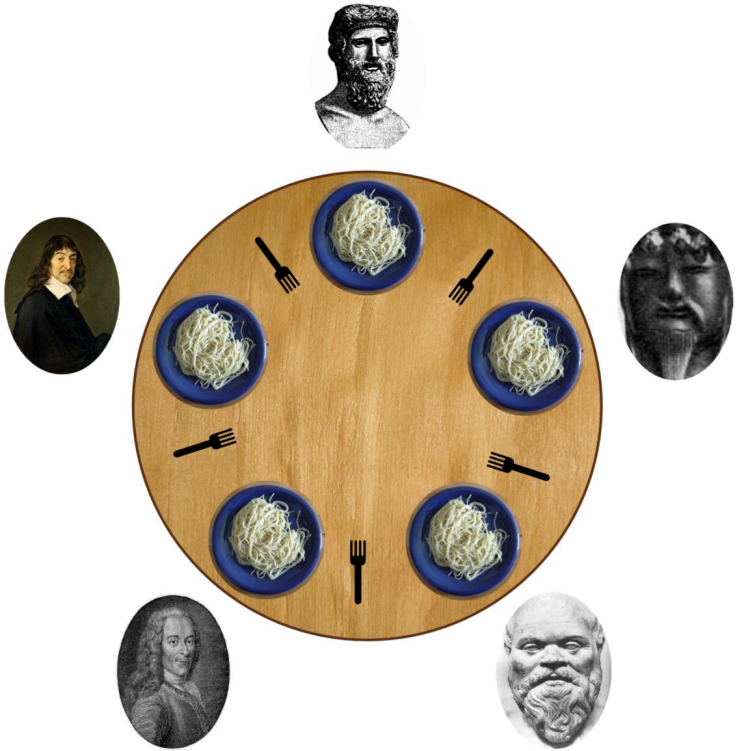
Lock

- A lock ensures that only one thread can access the data at a time when the data is locked
- Different locking mechanisms
  - Mutex
  - Semaphores
  - And many more!
- With locks, our code will have no problems!
  - Or will it?
    - Quick side-note: This is exactly how CS 111 is. A different mechanism will be introduced to prevent a problem, which will introduce another problem, resulting in coming up with another mechanism that will be introduced to prevent the newly created problem by the mechanism that prevents the original problem, which will introduce another problem

Deadlock

Dining Philosophers Problem

- Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers
- Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks
- Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher
- After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others
- A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks



Deadlock Code Example