# ASSIGNMENT 2

---

**Seth Lunders**
**Professor Bolden**
**CS 240**
**May 10, 2021**

---

## Program Design

First I familiarized myself with the existing code. Part 1 of the assignment was helpful here, as I just had to modify the code. In addition to the ExtractRGB RED, GREEN, and BLUE functions, I also got the LUMINANCE and SEPIA functions working. Here is that code:

### ImageToGrayScale();

```
// Inside the ImageToGrayScale function:
      else if (m == SEPIA)
      {
      // This converts to grayscale, then adds some red and green back
      // to get the sepia color.
        double dGray = (r + g + b) / 3.0;
        gray = (int)dGray;
        image[i][j].red = gray + ((255 - gray) / 3);
        image[i][j].green = gray + ((255 - gray) / 4),
        image[i][j].blue = gray;
      }
      else if (m == LUMINANCE)
      {
        double dLuminance = (r * 0.2126 + g * 0.7152 + b * 0.0722);
        luminance = (int)dLuminance;
        image[i][j].red = luminance;
        image[i][j].green = luminance,
        image[i][j].blue = luminance;
      }
```

### Design for Forking

This has taken a lot more thought than modifying the existing code. The first thing I want to do is split the image into 2 parts. The image is a 2d array of Pixel structs, but a 2d array can be represented as a 1d array. So, it would make sense to me to fork and edit the image this way:

- Access the image as a 1d array
- Make it accessible to all the children with mmap
- Get the length of the array using width*height*3, then divide that by 2 (or however many forks you want).
- Fork and run each half of the array through the image-modification function.
- Exit the child process.
- Convert the 1d array back into a 2d Pixel array.

### Update May 9th

After thinking a lot about how to get my previous forking design to work, I came up with a much simpler idea.

- First, use mmap in the loading function to make the image modifiable by all children.
- Modify the image-processing function to allow specifying starting and ending rows.
- fork() a desired number of times, sending each one a different section of the image to process. Make the parent wait until all children have returned using wait(NULL).
- Save image as normal.

This is the process that ended up working, I will go over the code for it in the Source Code section.

---

# Programming Log

- May 6 | Familiarized myself with existing code
    - Looked over assignment requirements (~5 min)
    - Compiled sample code and tested functionality (10 min)
    - Modified existing ExtractRGB/RED to work with BLUE and Green also (10 min)
    - Modified ImageToGrayScale to work with SEPIA and LUMINANCE (30 min)
- May 7 | Designed forking methods
    - Now that the image modification functions are working, I can start planning how to speed up the processing with forking.
- May 9 | Redesigned forking method, completed most of assignment.
    - Using the System Monitor of Pop!_OS, I found that while processing very large images (the one I used was about 200megapixels) the program was using upwards of

5gb of memory. This didn't seem right, so I found a program called valgrind that allowed me to pinpoint the memory leak issue. I hadn't been freeing the memory allocated by malloc in the LoadImage function.
    - I changed it to use mmap instead of malloc, and this solved the problem, as well as made it so the memory was shared. Before, I had used a seperate function that copied the image from LoadImage into an mmap-ed section of memory.
- May 10 | Timing processes today
    - Turns out there's more bugs to fix. Today I tried running my code on the CS server, which helped me find a bug that I didn't notice on my own machine. It was a pretty simple fix, and I outline it in the Output section. It had to do with my understanding of the wait(NULL) function.
    - Finished the timing section of the assignment
    - Finished putting this paper together.

---

# Source Code

The source code is almost entirely the same as the provided code, with a few additions. I'll list just those here.

### imageUtils.h

I added a 'rowStart' int to these functions, to make it so I could give them a starting position in the image.

```
/* Convert image to grayscale using specified mode. */
void ImageToGrayScale(Pixel **image, int height, int width, Mode m, int
rowStart);

/* Extract specific colors */
void ExtractRGB(Pixel **image, int height, int width, Mode m, int rowStart);
```

### imageUtils.c

```
else if (m == SEPIA)
{
    double dGray = (r + g + b) / 3.0;
    //gray = MIN_MACRO( (int)dGray, 255 );
    gray = (int)dGray;
    image[i][j].red = gray + ((255 - gray) / 3.5);
```

```
    image[i][j].green = gray + ((255 - gray) / 5),
    image[i][j].blue = gray;
} else if (m == LUMINANCE)
{
    double dLuminance = (r * 0.2126 + g * 0.7152 + b * 0.0722);
    //gray = MIN_MACRO( (int)dGray, 255 );
    luminance = (int)dLuminance;
    image[i][j].red = luminance;
    image[i][j].green = luminance,
    image[i][j].blue = luminance;
}
```

### colorUtils.h

Added GREEN, BLUE, LUMINANCE, and SEPIA to 'Mode' options

```
typedef enum {
  AVERAGE,
  RED,        /* extract  */
  GREEN,
  BLUE,
  LUMINANCE,
  SEPIA
} Mode;
```

### in testImage.c

Here is the final code for this forking section, using the ImageToGrayScale SEPIA function, though the same format is used for all the other functions:

```
int forkCount = 8; // Number of total threads you want
int division = h / forkCount; // How many rows to process
int childCount = 0;
  .
  .
  .
if (forkCount == 1) // For running in single-threaded mode
{
  ImageToGrayScale(image, division, w, SEPIA, 0);
```

```
}
else
{
  // While there are less than the desired number of threads, keep forking
  for (int i = 0; i < (forkCount - 1); i++) // Subtract 1 to compensate for the
parent thread
  {
    int childPID = fork();
    childCount++;
    if (childPID != 0)
    {
      if (childCount == 1) // Run first time only
      {
        ImageToGrayScale(image, division, w, SEPIA, 0);
      }
    }
    else
    {
      ImageToGrayScale(image, division * (childCount + 1), w, SEPIA, division *
childCount);
      // printf("I am child #%i with PID: %i\n", childCount, getpid());
      exit(EXIT_SUCCESS);
    }
  }
  while (childCount > 0) // Loop until all children have exited
  {
    wait(NULL);
    childCount--;
  }
}
```

# Output

Here's an issue I was having while running on the CS server:


Using some printf commands to do some debugging, I found that the Parent process was not waiting for *all* the children to exit. Turns out, I had originally used wait(NULL) somewhat incorrectly, as it only waits for *one* child to exit, and I only used it one time. This was easily fixed with a while statement:

```
while(childCount > 0) // Loop until all children have exited
{
  wait(NULL);
  childCount--;
}
```

Here is the image after running with the updated code:


With the added while loop, the parent waits until all children have finished before saving the image.

---

# Results & Observations

## Results from 'time ./testImage l16Sample.jpg l16Sample_s.jpg 6'

Here is the output for the time command with 1, 2, 8, then 16 threads, using the large photo I used above.

1 Thread:

```
real    0m3.342s
user    0m3.054s
sys     0m0.134s
```

2 Threads:

```
real    0m3.166s
user    0m3.000s
sys     0m0.187s
```

8 Threads:

```
real    0m3.258s
user    0m3.100s
sys     0m0.197s
```

16 Threads:

```
real    0m3.177s
user    0m2.965s
sys     0m0.194s
```

Multiprocessing didn't give as much as a speed boost as I expected—at first glance. Based just on these times, it's hard to tell that there's any difference in speed; in fact, some of the multiprocessed ones are slower than the single thread. So, I found a program called gprof that let me see how much time the program spends executing individual functions. Here are the times for just the parent function to execute the ImageToGrayScale function:

```
1 Thread:  0.33s
2 Threads: 0.16s
4 Threads: 0.08s
8 Threads: 0.04s
32 Threads: 0.01s
64 Threads: 0.00s (Too small for gprof to measure)
```

When just looking at the actual image-processing function, we can see that the multiprocessesing is making a difference. Previously, the loading and saving of the image took most of the total time, so small changes like this were difficult to see. It would be interesting to see if you could apply multiprocessing to the actual loading and saving of the image to speed up the process even further.

---