Assignment 2

CSI211—02 Programming Practice, Fall 2011

Due date:

11:59pm, October 18 (Tuesday) 2011.

Introduction

The purpose of this second assignment is to get more experience on parallel programming with POSIX threads. Specifically, we will focus on communication and synchronization between threads, such as

- passing arguments to threads during thread creation,
- communicating results back to the main thread upon thread termination (the main thread is the thread that executes your C **`main()`** function), and
- synchronization using mutexes, semaphores and barriers.

You should hand in your programs by the due date as explained in Section "Deliverables and Submission" below.

Please note that Assignments 1 and 2 are highly relevant for the midterm exam.

Environment

You can solve this assignment on any computer that has Linux, GNU GCC and the Pthreads librarary installed. A convenient way will be to do the assignment on our server (`elc1.cs.yonsei.ac.kr`).

This assignment will be tested and marked on the server; it is therefore highly recommended that you test your programs on the server before submitting (see also the submission instructions below).

Example 1: Monte-Carlo Pi computation

Monte Carlo methods are a class of computer algorithms that use random numbers to compute their results. We will implement a well-known Monte-Carlo method that computes an approximation of pi (http://en.wikipedia.org/wiki/Pi). This Monte-Carlo method lends itself nicely to parallelization. It is in fact an *embarrassingly parallel* programming problem. Recall from the lecture on "Parallelism" that an embarrassingly parallel problem can be split easily into many tasks, with no or only a few dependencies between tasks.

As usual, we implement a sequential version of the algorithm first, to get the functional requirements right. In a second step, we will parallelize the sequential version to speed up the computation.

Sequential version

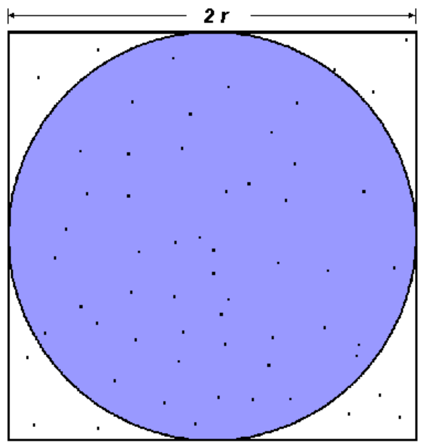

Figure 1

One way to approximate the value of Pi is based on the ratio of the area of a circle of radius r to the area of a square of length 2*r as depicted in Figure 1. The area of the square is $A_{\text{square}} = (2r)^2 = 4r^2$. The area of the circle is $A_{\text{circle}} = r^2\pi$. From the ratio of the area of the circle to the area of the square we can derive pi as depicted in the following equation.

$$\frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi}{4} \quad (a)$$

$$\pi = \frac{4 \cdot A_{\text{circle}}}{A_{\text{square}}} \quad (b)$$

To compute pi, let's place n points *at random* inside the square. We can then count the points that are inside the circle. If the distribution of points is truly random, then the ratio of the number of points inside the circle to the total number of points in the square should be according to Equation (a). The value of pi can then be approximated by

$$\pi = \frac{4 \cdot \text{NumberOfPointsInCircle}}{\text{NumberOfPointsInSquare}} \qquad (1)$$

You need to perform the following steps in your program:

1) The total number of random points that your program shall process will be provided on the command-line. Read this command-line argument (from now on referred to as "total_number_of_points") from the command line. (See the slides on command-line arguments from the C tutorial.) You need to convert the command-line string argument provided in **`argv[]`** to int, e.g., by using **`atoi()`** (see the Linux manpage on atoi, "man atoi"). Note: do not use scanf to read in arguments, or your program will fail our automated testing!
2) Inscribe a circle in a square (i.e., decide on a radius r that determines the size of the square and the circle). Note: Equation (1) shows that the approximation of pi is independent of the value of r. You are free to pick any value for r.

3) Randomly generate points in the square. Each point (x,y) consists of a random x-coordinate and a random y-coordinate.
4) Determine the number of points in the square that are also in the circle.
5) Let x be the number of points in the circle divided by the number of points in the square.
6) Pi = 4 * x

Hint 1: Your program should work as depicted in the following pseudo-code.

```
nr_points_in_circle = 0;
loop j = 1, total_nr_of_points
       x = random_number between 0 and r;
       y = random_number between 0 and r;
       if (x,y) within circle then
              nr_points_in_circle++;
       end if;
end loop;
pi = 4 * nr_points_in_circle/total_nr_points;
```

Hint 2: As depicted in Figure 2, Pythagoras' theorem can be used to decide whether a point with coordinates (x, y) is inside or outside of the circle of radius r.

Decide Pixel Location:

Inside : $x^2+y^2 \le r^2$

Outside : $x^2+y^2 > r^2$

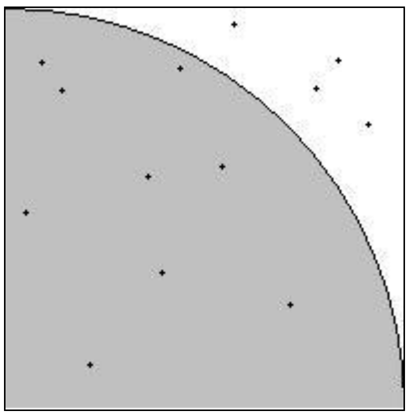
Figure 2

Hint 3: you may use the **rand_r()** function (see "man rand") from <stdlib.h> to generate random numbers. Note that unlike **rand_r()**, the **rand()** function is not thread-safe! (You can use it for the sequential version, but you should not use it for the parallel version.)

Parallel version

1) The parallel version shall get two arguments from the command-line: (1) the number of threads to be used ("number_of_threads"), and the number of random points per thread ("number_of_points_per_thread").
2) Create the required number_of_threads.
3) Each thread must create number_of_points_per_thread random points and count the number of points that are inside the circle.

4) Each thread must tell the main thread about the number of points it found to be inside the circle.
5) The main thread must collect the results from the threads and calculate the value for pi (Please note that with the parallel version the total number of points investigated is number_of_threads * number_of_points_per_thread).

Hint 4: every thread must compute the number of random points located inside the circle. You need to communicate this value back to the main thread. One way to accomplish this is to assign each thread a thread ID. The thread ID is essentially an integer value, 0 for the first thread, 1 for the second thread aso. Pass this thread ID as an argument to the thread during thread creation (see lecture slides on POSIX threads). You can then provide a global integer array variable were each thread can deposit the number of points found to be inside the circle. The main thread can read the values from there and compute pi. Note that there is a task dependency: the main thread can compute pi only *after* the threads have provided their values. Alternatively, you can pass back the number of points found via pthread_join (again, see the lecture slides).

- Input Format

```
Sequential version:   ./monte_seq   <total_number_of_points>
Parallel version:     ./monte_par   <number_of_threads>  <number_of_points_per_thread>
```

- Output Format

Value of Pi: 1 digit whole number part, floating point ("."), 8 digits fractional part, newline ("\n").

- Sample Output

```
[elc1: ~]$ ./monte_seq 100000
3.14130100

[elc1: ~]$ ./monte_par 4 25000
3.14135100
```

- Constraints.

You may assume the following constraints:

- The number_of_threads that your program must handle is between 1 and 4. If a different number is specified on the command line, your program must issue an error message and terminate, returning -1.
  Example:
  ```
  [elc1: ~]$ ./monte_par 5 25000
  Error: number of threads must be between 1 and 4.
  ```
  To exit your program and return -1, use **`exit()`** (see "man 3 exit").
- You can assume that the number_of_points_per_thread are between 1 and 100000000.

- You need not care about command-line arguments that are not valid integer numbers.

Example 2: RGB to grayscale conversion

In this example we start working with graphic images. Algorithms on graphic images (e.g. filters) frequently show potential to apply parallelism. In this example, we convert color images to grayscale. Our color images use the RGB color model (http://en.wikipedia.org/wiki/RGB_color_model). RGB is an additive color model where the primary colors Red (R), Green (G) and Blue (B) are added to produce any desired color. An RGB raster image (or pixel image) provides a triple of three values <R,G,B> for each pixel in the image. We will use the PPM file format for images. **The PPM file format has been discussed in class, please refer to the lecture slides provided in YSCEC.**

For the following discussion, assume three values <R,G,B> for the red, green and blue component of a pixel. We can convert a pixel to grayscale by using the following formula:

$$\textit{(GrayScale Value)} = \textit{(R value)}*0.30 + \textit{(G value)}*0.59 + \textit{(B value)}*0.11 \qquad (1)$$

Example: If one pixel's RGB value is <100, 200, 100>, it will convert to a grayscale pixel value of 159:
159 = 100*0.30 + 200*0.59 + 100*0.11

Applying this algorithm to all pixels of an image, we can convert the image from color to grayscale.

(Before Conversion)

(After Conversion)

As usual, we will develop a sequential version for this programming problem first; We will parallelize the sequential version in a second step.

Sequential version

Your program must perform the following steps:

1) Read the filename of the input image file from the command line.
2) Read the filename of the output image file from the command line.
3) Read in the input image file; the image file will be provided in PPM P3 format.
4) Convert the image to grayscale.
5) Write the image to the requested output file. The output file must be in PPM P2 format.

Notes:

- You must not make any assumption on the size of the image provided as input. Your program must read in the header of the input file and allocate enough memory to accommodate the whole image. You should use **`malloc()`** to allocate memory (see 'man malloc' for a description on how to dynamically allocate memory).
- You can assume that the R,G,B components of a pixel are in the range between 0 and 255. You should specify 255 as the maximum pixel value in the header of your output file.
- The formula in Equation (1) above uses floating point computations. You must type-cast the final result to type int, because the PPM file format requires decimal values (and not floating-point values).

Parallel version

The parallel version of your program should accept an additional command-line parameter that denotes the **number of threads** to be used for grayscale conversion. You must not assume a limit on the number of threads. This means that you need to allocate thread IDs dynamically (malloc).

Each pixel in the image can be converted independently from the other pixels, which means that grayscale conversion is again an embarrassingly parallel programming problem (with enough processors/cores, you could convert the image in one step, by having each processor/core work on a single pixel!).

You must divide your image into parts and assign each thread a part to work on (data-parallelism). You should provide an argument with pthread_create() to tell each thread on which part of the image to work on. The actual division of work between threads is up to you. Note: reading of the input image and writing of the output image should still be done by the main thread. Only grayscale conversion is parallelized!

Correctness Criteria

In order for this exercise to be graded, the output of your file must be formatted in PPM P2 format (see our lecture slides on the PPM image format). You must format your file according to the following guidelines:

1) The first line in your file MUST follow this format:
   `P2 [a] [b] [c]`
   where
   - P2 is the ASCII character "P" followed by the ASCII character "2".
   - [a] is the width of the image in pixels (determined by the width of the input file)
   - [b] is the height of the image in pixels (determined by the height of the input file)
   - [c] is the maximum grayscale value. For this assignment, we will use 255 as the maximum grayscale value.
2) Subsequent lines contain the grayscale values for 4 pixels each. If the number of pixels in an image is not evenly divisible by 4, then the last line of the file contains less than 4 pixel values.
3) You must have exactly one blank (" ") between all values (see the example below).
4) You must not have any extra whitespace characters (blanks, tabs, ...) between values or at the end of a line of output. Your rows must end in "<value>\n".
5) Your image file must not contain comments or empty lines.

Example:

Assume the grayscale image depicted in Figure 3. (We assume that white pixels have the value 255, black pixels have the value 0, and the gray pixels in this image have the value 128.

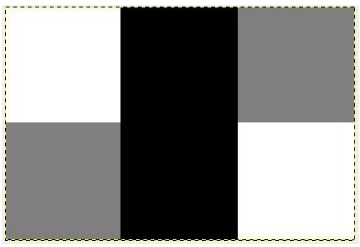
Figure 3

Given the image in Figure 3, your output file should be formatted as follows.

```
P2 3 2 255
255 0 128 128
0 255
```

Note: if your file does not follow the above conventions, it will fail the automated grading procedure and receive 0 points!

We provide you with sample images and sample solutions in directory /opt/pp/assignment2/example2 on the elc1 server. You can use the Unix diff command to ensure that the output of your program is the same as the sample solution:

```
[elc1: ~]$ cp /opt/pp/assignment2/example2/* .
```
(copies sample images and solutions to your working directory)

```
[elc1: ~]$ ./conv_seq sample1.ppm output.ppm
```
(your program will convert the sample image file sample.ppm to grayscale and store the output in file output.ppm)

```
[elc1: ~]$ diff output.ppm sample1_solution.ppm
[elc1: ~]$ echo $?
 0
[elc1: ~]$
```
(The **`diff`** utility will compare the file contents of your file output.ppm and the provided solution in file sample1_solution.ppm. If **`diff`** exits with exit code 0 (checked with the command "**`echo $?`**"), the files are the same. This means that your program produced the correct output. If the output of your program differs from the sample solution, **`diff`** will point out the differences.)

- Sample invocations

```
Sequential version:   ./conv_seq   <input filename>   <output filename>
Parallel version:     ./conv_par   <input filename>  <output filename>   <number of threads>
```

- Sample Output

```
[elc1: ~]$ ./conv_seq test1.ppm output1.ppm
[elc1: ~]$ ./conv_par test2.ppm output2.ppm 4

[elc1: ~]$ cat output1.ppm
P2 1024 1024 255
125 123 142 111
...(more lines)...

[elc1: ~]$ cat output2.ppm
P2 1024 1024 255
125 123 142 111
...(more lines)...
```

The above example outputs assume that the input file was 1024x1024 pixels!

- Utilities

- You can use the 'ppmtojpeg' command to check your output (to get more information, refer to the lecture slides on the ppm image format.) Note that gimp (www.gimp.org/) can display ppm files directly. Gimp is available for Windows and Linux.
- You can create additional test data by converting jpg files to ppm P3 format. For example:

```
[elc1:~]$ cp /opt/pp/assignment2/example2/eisbaer.jpg .
[elc1:~]$ jpegtopnm eisbaer.jpg > eisbaer.pnm
[elc1: ~]$  P6toP3 eisbaer.pnm eisbaer.ppm
```

Note that the source code for the P6toP3 utility is provided in /opt/pp/assignments2/example2. P6toP3 converts a ppm file from P6 (binary) to P3 (ASCII) format.

Example 3: Understanding C pointers

What does the following program print? You should be able to figure out the answer without running the program on a computer! NB: this material is covered in the tutorial notes on C.

```c
#include <stdio.h>

int x = 1, y = 2, z = 3, x1 = 4;
int *u, *v, *w;
int A[6] = {5, 4, 3, 2, 1, 0};

void foo (int a, int * b, int ** c) {
   a  = 2 * (*b);
   *b = 10;
   **c = 20;
}

int * foo1 (int a) {
   return &A[a];
}

void foo2 (int * a, int b) {
   a = u;
}

int main(void) {
   u = &y;
   foo (x, &y, &u);
   printf("%d %d %d %d\n", x, x1, y, z);
   *u = 40;
   v = u;
   u = &z;
   w = foo1(*u);
   v = foo1(2);
   *v = 30;
   printf("%d %d %d %d %d %d %d %d\n", *w, x1,
           A[0], A[1], A[2], A[3], A[4], A[5]);
   foo2(v, 22);
   printf("%d\n", *v);
   foo(22, A, &u);
   printf("%d %d %d %d %d %d %d\n", z, A[0], A[1], A[2], A[3], A[4], A[5]);
   return 0;
}
```

Example 4: Ping Pong Pung

In file /opt/pp/assignment2/pingpongpung_template.c you find a simple C program that generates three threads. Every thread contains an endless loop with a printf statement. One thread outputs "ping", the second thread outputs "pong", and the third thread outputs "pung". It is your task to establish thread synchronization such that the output will be

```
ping
pong
pung
ping
pong
pung
....
```

You are allowed to use three binary semaphores only. You should reason about your program to ensure that the above execution order is enforced. A test run that shows the correct result does not prove anything (your program might still be wrong in subtle ways).

Example 5: I told you that... (Pieces of advice from famous computer scientists)

In file /opt/pp/assignment2/studypp_template.c you find a simple C program that generates four threads. Each of these threads belongs to a famous computer scientist that we already encountered during the lectures. These computer scientists got together to share some of their most important insights. Consider the example below.

```
[bburg@elc1 sems]$ ./studypp
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
You shall abide   by the   locking   hierarchy.
Hennessy: Now we're into the explicit parallelism multiprocessor area!
Moore: The amount of transistors on a single chip doubles every 18 months.
Dijkstra: Goto considered harmful!
Amdahl: Your potential speedup is limited by the fraction of sequential code in your program.
http://en.wikipedia.org/wiki/Amdahl%27s_law
http://www.intel.com/pressroom/kits/bios/moore.htm
http://en.wikipedia.org/wiki/Edsger_Dijkstra
http://view.eecs.berkeley.edu/wiki/Main_Page
[bburg@elc1 sems]$
```

Essentially, those computer scientists proceed in 3 phases, depicted in blue, green and red above.

- In the first phase, they collaborate to pronounce the sentence "You shall abide by the locking hierarchy". Since they consider this really important, they repeat this 10 times. If you consider the code of each computer scientist's thread, you find printf statements that output parts of the above message.
- In the second phase, each computer scientist gives his opinion on an important programming topic. The order of these messages is arbitrary.
- In the third phase, each computer scientist tells us a link where we can find more insights. The order of these insights is again arbitrary.

It is your duty to use the semaphores defined at the beginning of the program to ensure the correct output of the messages "You shall abide by the locking hierarchy" in Phase 1. You are not allowed to introduce additional semaphores.

You must use the barrier defined at the beginning of the program to ensure that no computer scientists enters Phase 2 before his colleagues have finished Phase 1.

You must use this barrier for a second time to ensure that no computer scientists enters Phase 3 before his colleagues have finished Phase 2. Note again that the computer scientists are allowed to speak in any order during Phases 2 and 3 (the output shown above is only one possible example)!

Marking

With Example 1, we will check that your approximation of pi is within reasonable bounds given the number of random points. With Example 2, we will run your program on several ppm files and compare the solutions computed by your program to our solutions. With examples 4 and 5, we will check the correctness of your locking schemes. **Testing will be done on the elc1 server. Please make sure that your programs compile and execute there. Programs which fail to compile, crash, or produce no or the wrong output will receive no points.**

Deliverables and Submission

Your submission should contain the following files:

- monte_seq.c
- monte_par.c
- conv_seq.c
- conv_par.c
- pingpongpung.c
- studypp.c

Please note that Example 3 is meant as an additional preparation for the midterm exam. Therefore no points will be given for Examples 3 and you need not hand it in.

To submit the above files, please log in on elc1.cs.yonsei.ac.kr, create a submission directory, e.g., ~/myassignment2, and copy the above files to this directory (the directory should contain only those files and nothing else!).

After copying the files to your submission directory, you must cd to the submission directory and type 'submit' to submit your Assignment 2. The submit command will collect all files from the directory and copy them to our repository.

```
[you@elc1 ~] cd ~/myassignment2
[you@elc1 myassignment2] submit
```

Please note: There is no submission of Assignment 2 unless you issue the 'submit' command! Failing to issue this command will result in 0 points. The timestamp of your submission will be recorded by the 'submit' command. Please note also that if you re-submit, then a previous submission will be deleted (i.e., all files from the previous submission will be deleted) and a new timestamp will be recorded for your submission.

If you have questions, don't hesitate to contact the TAs. ☺

Good luck!