Homework 1

Assignment due:

9:00 pm, Tuesday, September 12th.

Problems 1, 2, 3:

Submit typed (not hand-written) answers to the non-programming problems. Prepare a **`answers.pdf`** file containing your answers, and submit it to the D2L dropbox. For these three exercises, explain your answers clearly and succinctly. Problems 1-3 are worth 36 points (12 points each).

Programs:

Submit the program solutions electronically. The programming exercises and associated report are worth 64 points.

A reminder: Homework assignments are individual efforts. The two projects can be done in teams of two, not the homework assignments. You may discuss the meanings of questions with classmates, but the answers and programs you turn in are to be yours alone.

Problem 1: Exercise 2.17, page 86

Consider the following program:

```
co <await (x >= 3) x = x - 3;>
// <await (x >= 2) x = x - 2;>
// <await (x == 1) x = x + 5;>
oc
```

a.) For what initial value(s) of **`x`** is the program guaranteed to terminate, assuming scheduling is weakly fair? What are the corresponding final values? Explain.
"Guaranteed to terminate" means that all three arms of the **`co`** will complete.

b.) Are there additional initial value(s) of **`x`** for which the program might terminate? If so, show how those values might result in termination, and the final values that might result. Explain.
"Might terminate" means that there is at least one scenario in which all three arms of the **`co`** will complete, but they are not guaranteed to do so.

Problem 2: Exercise 2.20, pages 86-87

(Only three of the parts)
Let **`a[1:m]`** and **`b[1:n]`** be integer arrays, **`m > 0`** and **`n > 0`**. Write predicates to express the following properties.

b.) Either **`a`** or **`b`** contains a single zero, but not both.

d.) The values in **`b`** are the same as the values in **`a`**, except they are in reverse order. (Assume for this part that **`m == n`**.)

f.) Some element of **`a`** is larger than some element of **`b`**, and vice versa.

As an example, here is a solution to part e.:
e.) Every element of **`a`** is an element of **`b`**.

Solution:

e.) $\forall i: 1 \le i \le m, \exists j: (1 \le j \le n : a[i] = b[j])$

Problem 3: Exercise 2.33

Consider the following program:

```
int x = 10, c = true;
co <await x == 0)>; c = false;
// while (c) <x = x - 1>;
oc
```

a.) Will the program terminate if scheduling is weakly fair? Explain.

b.) Will the program terminate if scheduling is strongly fair? Explain.

Now, add the following as a third arm of the **co** statement:

```
while (c) {if (x < 0) <x = 10>;}
```

I.e., the program would be:

```
int x = 10, c = true;
co <await x == 0)>; c = false;
// while (c) <x = x - 1>;
// while (c) {if (x < 0) <x = 10>;}
oc
```

c.) Will the program terminate if scheduling is weakly fair? Explain.

d.) Will the program terminate if scheduling is strongly fair? Explain.

Programming Exercises:

(Variations on Exercise 1.9, page 37)

The textbook's description:

Write a sequential recursive program to sort an array of strings. Use quicksort for this sequential sort.

Then, modify your program to use recursive parallelism. Be careful to ensure that the parallel calls are independent. Implement both programs and compare their performance.

Write three versions:

a.) Sequential version

b.) Process version

c.) Thread version

Use C for each of the programs.

The course will start covering synchronization techniques (busy-waiting, semaphores) while you are working on this assignment. Do NOT use them on this assignment. The idea here is to create processes or threads to complete a task, wait for them to finish the task (i.e., wait for them to terminate). Repeat for each task.

a.) Sequential Version

Write a sequential quicksort (i.e., no processes and no threads) that will read strings from a file, sort the strings alphabetically using a recursive implementation of quicksort, and then write the sorted strings to **`stdout`**.

Using **`gettimeofday()`**, print to **`stderr`** the time the program needs to perform the quicksort (do not include the times needed to read in the file and to print the sorted file to stdout).

Name this file **`sortSeq.c`**. Your **`Makefile`** should produce an executable named **`sortSeq`**.

The name of the file to sort will be the first command-line argument.

You may use additional command-line arguments, if you wish. These additional arguments should be optional. Thus, we should be able to run your program by typing:

```
sortSeq <fileName>
```

Wikipedia (among other sources) has a good article on quicksort. Take note of better ways to choose the pivot — in particular, the median-of-three approach (use this!!).

Avoid using **`strcpy`** and **`memcpy`** during the sorting process! This can significantly increase the run time of the sort. Either can be used during the reading of the file.

A common technique for speeding up quicksort is to stop the recursion when there is a small (20 to 40) number of lines remaining to sort. The idea is to use an N-squared sort to finish this last piece of the sorting to avoid the recursive calls that would be needed to finish the sort. Do **not** do that here. Use the quicksort algorithm to do all the sorting.

b.) Process Version

Use processes (not threads).
Name your program: **`sortProcess.c`**

The number of processes will be a command-line argument, along with the name of the file to sort. You may use additional command-line arguments if you wish, but they should be optional. Thus, we should be able to run your program with the command:

```
sortProcess <numProcesses> <fileName>
```

You can assume that the number of processes will be a power of two, with **`16`** as the largest possible value.

Use the following two techniques to perform the sort:

The first stage:

> Divide the strings to be sorted into **`numProcesses`** sections. Create a process to use quicksort on each section. Wait for all of the processes to finish.

Subsequent stages:

> Use a series of steps that use merging to join pairs of sections until only one section is left.

For example, with 8 processes:

The first stage (see above) will create the 8 processes with each process performing quicksort on one-eighth of the array of strings.

Then, create 4 processes to merge pairs of sections, resulting in 4 sorted sections. Each thread will merge two (of the eight) sections. I.e., the first process will merge the first two sections, etc.

Then, create 2 processes to merge the 4 sections remaining into two sorted sections.

Finally, merge sort the last two sections to produce the fully-sorted result.

Print to **`stderr`** the time needed to do the quicksorting and merging. Do not include the time needed to read the original file and do not include the time needed to print the sorted file to **`stdout`**.

`mmap` parameters:
The lecture notes state that **`MAP_LOCKED`**. is optional on Linux and should not be used on OSX.

It turns out that asking the OS to keep the shared memory in real memory, rather than allowing it to be swapped to virtual memory, will limit the maximum size of the shared memory region on OSX. The maximum size is then too small for the larger files that you will use in testing.

The simple solution is to not use **`MAP_LOCKED`** at all on both OSX and Linux.

c.) Thread Version
Use threads (not processes).
Name your program: **`sortThread.c`**

The number of threads will be a command-line argument, along with the name of the file to sort. You may use additional command-line arguments if you wish, but they should be optional. Thus, we should be able to run your program with the command:

```
sortThread <numThreads> <fileName>
```

You can assume that the number of threads will be a power of two, with **`16`** as the largest possible value. Divide the strings to be sorted into **`numThreads`** sections. Create a thread to use quicksort on each section.

Use the following two techniques to perform the sort:

The first stage:

Divide the strings to be sorted into **`numThreads`** sections. Create a thread to use quicksort on each section. Wait for all of the threads to finish.

Subsequent stages:

Use a series of steps that use merging to join pairs of sections until only one section is left.

For example, with 8 threads:

> The first stage (see above) will create the 8 threads with each thread performing quicksort on one-eighth of the array of strings.
>
> Then, create 4 threads to merge pairs of sections, resulting in 4 sorted sections. Each thread will merge two (of the eight) sections. I.e., the first process will merge the first two sections, etc.
>
> Then, create 2 threads to merge the 4 sections remaining into two sorted sections.
>
> Finally, merge sort the last two sections to produce the fully-sorted result.

Print to **`stderr`** the time needed to do the quicksorting and merging. Do not include the time needed to read the original file and do not include the time needed to print the sorted file to **`stdout`**.

Text files:
Your sorting code will read from a text file that has no more than 200 characters per line (including the newline character).

Assume there will be up to 1,000,000 lines in the input file.

There are several text files in **`/home/cs422/fall17/hw1`** that you can use in testing your programs.

Patrick's executables:
My three executables are available in **`/home/cs422/fall17/hw1`**. They are named **`patrickSortSeq`**, **`patrickSortThread`**, and **`patrickSortProcess`**. Each executable takes two parameters: the number of threads/processes and the name of the file to be sorted. The number of threads/processes should be 1, 2, 4, 8, or 16. Note that **`patrickSortSeq`** also takes the number of threads/processes, even though it uses a sequential sort.

There is one additional parameter that can be used with these executables. If **`-d`** is the first parameter, the program will produce (many!) diagnostics messages to stdout during execution. This is an optional feature; you are not required to do this in your code (though you may if you wish!)

Report:
Name the report: **`report.pdf`**.
Prepare a short report that describes the test cases you ran and the results you obtained. Be specific about the test cases you ran. We should be able to run your test cases and see similar results.

You are required to do your testing on machines in the CS department. This way, we can duplicate your results. You may, of course, develop on your own machine, and run tests there if you wish. However, this cannot replace running programs on the departmental machines.

Do **not** use lectura to do performance tests.

The machines oxford and cambridge are available, and can be reached via ssh from off-campus.

You can also use any of the Mac's in 228 or 930. These can be reached from off-campus by first ssh'ing to lectura/oxford/cambridge, and then ssh'ing to the Mac you choose. Check the load on your chosen Mac before starting tests that take a long time.

Turnin:

Note: We are asking for submission of programs via D2L only.

- For the written answers to questions 1, 2, and 3, submit **`answers.pdf`**.
  - Do NOT submit in any other format!
- For the three programs, submit **`sortSeq.c`**, **`sortProcess.c`** and **`sortThread.c`**. Include a **`Makefile`** that has a target named **`all`**. By typing **`make all`**, we should be able to create **`sortSeq`**, **`sortProcess`** and **`sortThread`** executables.
- For the report submit **`report.pdf`** to the dropbox.
- Logon to D2L. Select CSc 422. Click on the "Add a File" button. A pop-up window will appear. Click on the "Choose File" button. Use the file browser that will appear to select your file(s). Click on the "Upload" button in the lower-right corner of the pop-up window.
- You are now back at "Submit Files - Program1". Click on "Upload" in the lower-right of this window. You should now be at the "File Upload Results" page, and should see the message **`File Submission Successful`**.

D2L will send you a confirmation email after each item is placed in the Dropbox. This email is sent to your UA email (which is <yourUANetId>@email.arizona.edu). Retain these emails at least until you have received a grade for the assignment; it is your confirmation that you did submit the program(s).

You may submit the various items more than once. We will grade the last one that you put in the Dropbox.

Homework 1 Solutions

Problem 1:

Consider the following program:

```
co <await (x >= 3) x = x - 3;>
// <await (x >= 2) x = x - 2;>
// <await (x == 1) x = x + 5;>
oc
```

a.) For what initial value(s) of **x** is the program guaranteed to terminate, assuming scheduling is weakly fair? What are the corresponding final values? Explain. "Guaranteed to terminate" means that all three arms of the **co** will complete.

b.) Are there additional initial value(s) of **x** for which the program might terminate? If so, show how those values might result in termination, and the final values that might result. Explain. "Might terminate" means that there is at least one scenario in which all three arms of the **co** will complete, but they are not guaranteed to do so.

a.) Only the initial values **1** and **6**.

For **x = 1**: **C** has to execute first. Then, there are two possible orders for the other two statements. **C**, **A**, **B** will result in **1**. **C**, **B**, **A** will result in **1**.

For **x = 6**: Either **A** or **B** has to execute first, followed by the other of **A** or **B**. **C** is executed last. **A**, **B**, **C** will result in **5**. **B**, **A**, **C** will result in **5**.

b.) There are two possible results:

For **x = 4**: The sequence **A**, **C**, **B** will result in **4**. (Note: The other possibility with **x** starting as **4** would be if **B** goes first. This leads to: **B**, then deadlock. **x** will be **2** after **B** executes; **A** and **C** will never get past their **await** clauses.

For **x = 3**: The sequence **B**, **C**, **A** will result in **3**. (Note: The other possibility with **x** starting as **3** would be if **A** goes first. This leads to: **B**, **A**, then deadlock. **x** will be **-2** after **A** executes and **C** will never get past the **await** clause..)

Problem 2: Exercise 2.20, pages 86-87

(Only three of the parts)

Let **a[1:m]** and **b[1:n]** be integer arrays, **m > 0** and **n > 0**. Write predicates to express the following properties.

b.) Either **a** or **b** contains a single zero, but not both.

d.) The values in **b** are the same as the values in **a**, except they are in reverse order. (Assume for this part that **m == n**.)

f.) Some element of **a** is larger than some element of **b**, and vice versa.

b.) $((\exists i: 1 \le i \le m : a[i] = 0) \wedge (\forall j : 1 \le j \le m, i \ne j : a[j] \ne 0) \wedge (\forall j : 1 \le j \le m : b[j] \ne 0)) \vee ((\exists i : 1 \le i \le m : b[i] = 0) \wedge (\forall j : 1 \le j \le m, i \ne j : b[j] \ne 0) \wedge (\forall j : 1 \le j \le m : a[j] \ne 0))$

d.) $(\forall i : 1 \le i \le m : a[i] = b[m - i + 1])$

f.) $(\exists i,j : 1 \le i \le m, 1 \le j \le n : a[i] > b[j]) \wedge (\exists i,j : 1 \le i \le n, 1 \le j \le m \;\; b[i] > a[j])$

Problem 3: Exercise 2.33

Consider the following program:

```
int x = 10, c = true;
```

```
co <await x == 0)>; c = false;
// while (c) <x = x - 1>;
oc
```

a.) Will the program terminate if scheduling is weakly fair? Explain.
b.) Will the program terminate if scheduling is strongly fair? Explain.

Now, add the following as a third arm of the **co** statement:

```
while (c) {if (x < 0) <x = 10>;}
```

I.e., the program would be:

```
int x = 10, c = true;
co <await x == 0)>; c = false;
// while (c) <x = x - 1>;
// while (c) {if (x < 0) <x = 10>;}
oc
```

c.) Will the program terminate if scheduling is weakly fair? Explain.
d.) Will the program terminate if scheduling is strongly fair? Explain.

a.)
Assuming no wrap around of the values of **x** in the second branch of the **co** statement: No guarantee. **x** becomes zero only once, and the first branch of the **co** statement would need to check at exactly the right point.

b.)
Assuming no wrap around of the values of **x** in the second branch of the **co** statement: No guarantee. **x** becomes zero only once, and the first branch of the **co** statement would need to check at exactly the right point.

Assuming wrap around does happen, then the program is guaranteed to finish under strongly fair scheduling. The value of **x** will infinitely often be true (note: it will not be true nearly as often as it is false, but the only requirement of strongly fair is that it be true infinitely often — no requirement as to the percentage of time it is true).

c.)
No. Weakly fair requires that the condition become true and remain true. The condition does not remain true with this change, since it again becomes false when **x**'s value changes.

d.)
Repeating part b.) Yes. The condition becomes true infinitely often. Strongly fair scheduling will then guarantee that process 1 finds its condition true and sets **c** to stop the second and third processes.