CSCI 447 - Operating Systems, Spring 2024
Homework 2 : histories, synchronization
Due Date: See Canvas

~~Collaboration~~ Policy

Your homework submissions must be yours. You may chat with other students on a high-level about topics and concepts, but you cannot share, disseminate, co-author, or even view, other students' code. If any of this is unclear, please ask for further clarification.
The use of the textbook as help is allowed. But, do not use prose from the textbook verbatim. Paraphrase. Rearticulate in your own words.

1 Free Response Questions

Upload a pdf document to Canvas for the *Homework 2 - questions* submission.

Q1 : 10 points

Assume you are an employee of We-Tolerate-Errors, a software company that specializes in writing programs with multiple threads/processes. You are working on a program for which you have the executable but not the source code (you cannot modify the program, not even via reverse engineering funny business). All that you've been told is that the program has 3 user-level threads:

- **Thread A** : $a1 < a2$
- **Thread B** : $b1 < b2 < b3$
- **Thread C** : $c1 < c2 < c3$
- **Thread D** : $d1 < d2$

Threads B and C have 3 instructions, and Threads A and D have 2 instructions each. Each thread's instructions are executed sequentially (where $<$ specifies the ordering that we've seen in lecture). However the four threads are run concurrently, so the orderings among instructions from different threads is unpredictable from one invocation of the program to another. For example the following 2 are instruction histories that might be realized if the program is run concurrently:

- **History 1** : $a1 < a2 < b1 < c1 < c2 < b2 < b3 < d1 < d2 < c3$
- **History 2** : $a1 < b1 < b2 < c1 < c2 < a2 < c3 < b3 < d1 < d2$

Also assume that dependencies among variables in use by the threads necessitate that the following constraints (instruction histories) be maintained :

- **Constraint 1** : $a1 < b3$
- **Constraint 2** : $c2 < a2$
- **Constraint 3** : $d1 < b2$

We-Tolerate-Errors guarantees their software to be correct to a certain fault tolerant level. Your task is to assess the program's fault tolerant level. The question you must answer : **what percentage of all of the possible invocations of the software (when run concurrently) will give an incorrect result?** An Incorrect result is achieved when an execution history violates one of the constraints. **For full credit, you must show your work. Provide your answer with three decimal digits**. If you write code (your choice of language) to answer this question, insert your code into the submission document.

Q2 : 7 points

Assume three threads, `A`, `B`, and `C`, with two instructions each, as shown in Figure 1.

Thread A	Thread B	Thread C
IA1 : x = x+1	IB1 : x = x-2	IC1 : x = x+2
IA2 : print(x)	IB2 : print(x)	IC2 : print(x)

Figure 1: Three Threads, with two instructions each

All three threads are sharing the integer variable `x`, and the instruction pipeline on the computer where these threads are concurrently executed contains *fetch*, *execute*, and *write back* stages. As a first step, rewrite the instructions for the threads into a register/ALU view (as was shown in lecture). For the update, each thread fetches the value of `x` into a private (non-shared) memory space, and the write back stage writes to the shared variable. Assume that the `print` instruction of each thread cannot be decomposed further and the `x` used by a `print` is the value of the shared `x`.

Under these conditions, which of the following outputs (output to the screen) is/are possible when all three threads run to completion? If the output is NOT possible, then explain **why**. If the output IS possible, then **provide the order** of execution of the register/ALU instructions, as well as the starting value of the shared variable.

A. 545

B. 232

C. 151

D. 223

E. 536

F. 446

G. 789

Q3 : 3 points

What is the main advantage of the layered approach to system design? What are the disadvantages of using the layered approach?

Q4 : 3 points

Including the initial parent process, how many processes are created by the following program:

```
#include <stdio.h>
#include <unistd.h>

int main(){
   fork();
   fork();
   fork();
   return 0;
}
```

Q5 : 3 points

When a process creates a new process using `fork()`, which of the following states is/are shared between the parent process and the child process:

A. Stack

B. Heap

C. Shared memory segments

Q6 : 3 points

Using Amdahl's Law, calculate the speedup gain of an application that has a 65% parallel component for (a) two processing cores, and (b) four processing cores.

2 Coding : 6 pts for setup & output, 15 pts for the program

The coding tasks are an extension to Lab 2. **Complete lab 2 prior to starting the coding portions of this homework**.

1. Create and checkout a new branch on your gitlab for this assignment. Name the branch *homework2*. **When creating a new branch, ALWAYS do that from the master branch.** In the *homework2* branch, create a new directory, *homework2*. All work for this homework assignment should be done in the *homework2* branch and *homework2* directory.

2. Download the *homework2Files.tgz* file from the course website, and save it to the *homework2* directory. Unzip and untar. Yes – the files are the SAME as those for Lab 2.

3. Implement the Dining Philosopher's Solution using a monitor. Philosophers should think, get hungry, and eat, in that order. A starting framework for your solution is provided in *Main.k*. Each philosopher is modeled with a thread and the code you are being provided sets up these threads. The synchronization will be controlled by a "monitor" called `ForkMonitor`. The code for each thread/philosopher is provided for you. Look over the `PhilosophizeAndEat` method; you should not need to change this code. Note that each philosopher will eat 7 times. The monitor to control synchronization between the threads is implemented with a class called `ForkMonitor`. The following class specification of ForkMonitor is provided:

```
class ForkMonitor
  superclass Object
  fields
    status: array [5] of int      -- For each philosopher: HUNGRY,
                                  -- EATING, or THINKING
  methods
    Init ()
    PickupForks (p: int)
    PutDownForks (p: int)
    PrintAllStatus ()
endClass
```

   Write code for the `Init`, `PickupForks` and `PutDownForks` methods. You'll also need to add additional fields and perhaps even add another method.

   The code for `PrintAllStatus` is provided. You should call this method whenever you change the status of any philosopher. This method will print a line of output, so you can see what is happening.

   How can you proceed? You'll need a mutex to protect the monitor itself. There are two main methods (`PickupForks` and `PutDownForks`) which are called by the philosopher threads. Upon beginning each of these methods, the first thing is to lock the monitor mutex. This will ensure that only one thread at a time is executing within the monitor. Just before each of these methods returns, it must unlock the monitor (by unlocking the monitor's mutex) so that other threads can enter the monitor code.

   The file *DiningPhilosopherSampleOutput.pdf* on the course website contains a sample output of a correct solution.

4. Create a transcript of a terminal session in which you demonstrate the running of Dining Philosophers. Name the file *homework2-script.txt*.

5. `add`, `commit`, and `push` your new file, and changes to existing files, along with the script, to your *homework2* branch of your gitlab. **Make sure NOT to push to your git any .o or executable files. That can be achieved via the use of .gitignore, or by adding and committing files one-by-one.**

3 Rubric

Custom and book questions	29
Programming Task : setup (git), including branch	1
Programming Task : .o nor executable file(s) have NOT been pushed to your git branch	1
Programming Task : script, *assignment2-script.txt*, submitted to the homework2 branch	4
Programming Task : solution to Dining Philosophers Each philosopher eats 7 times No adjoining philosophers eating at the same time Sometimes two philosophers eating at the same time Eating duration varies among philosophers and for each philosopher Philosophers proceed from thinking, to hungry, to eating, in that order	15
Total	50 points