Exercises to work out and turn in:

Grading Guidelines (See Appendix):

In general, a right answer will get full credit when:

1. It is right (worth 25%)
2. It is right **AND** neatly presented making it easy and pleasant to read. (worth an **extra** 15%)
3. There is an **obvious and clear link[[1]](#footnote-1)** between 1) the information provided in the exercise and in class and 2) the final answer. A clear link is built by properly writing, justifying, and documenting an answer (worth an **extra** 60%).
4. Calculation mistakes will be minimally penalized (2 to 5% of full credit) while errors on units will be more heavily penalized.

**Late Submission** : as specified in the syllabus. Day counting starts one minute after the deadline.

**Check Your Submission:**  after submitting, download your submission to check whether it is the right version and it is complete.

You are welcome/encouraged to discuss exercises with other groups or the instructor. But, ultimately, **personal** writing is expected.

* USE THIS FILE AS THE STARTING DOCUMENT YOU WILL TURN IN. **KEEP IN THE QUESTIONS** AND **INSERT** YOUR ANSWERS **RIGHT AFTER THE QUESTIONS**.
* IF USING HAND WRITING (STRONGLY DISCOURAGED), REWRITE THE QUESTIONS.
* FAILING TO FOLLOW TURN IN DIRECTIONS /GUIDELINES WILL COST A 30% PENALTY.

Objectives of this assignment:

* to work on a Unix based system
* to become familiar with the notion of a *process*
* to assess/evaluate the *fairness* and *reliability* of the CPU scheduler on Tux machines

What you need to do:

* to work on a Unix based system
* Review a few Unix commands
* Examine C code
* to understand the fork() function to create a ”child” process
* to understand the relationship (or lack of) between parent and child process
* to "experience" the ***data shared*** problem

**Important:**

* *One submission per team.* ***You must complete all tasks on an Engineering Unix Tux machine.***
* *Writing and presentation of your report are considered to grade your hands-on lab. Your conclusions* ***must be supported*** *by the data/measurements you collect. Your conclusions must be correct.*
* ***Questions about this lab must be posted on Piazza if you need timely answers****.*
* ***Work ahead. Do not wait until the last minute.***

Task 1: Basic Unix Commands (30 points)

(Well written short answers are acceptable for this assignment)

The objective of this exercise is to get familiar with basic frequently used commands.

**Task**:

**In order to save space, for this assignment and future ones, clip out the screenshots to contain only the relevant information. Make sure that the screenshots are easily readable.**

Consider these basic frequently used Unix commands:

echo, date, time, more, rm, rmdir

Be careful with the commands rm, rmdir.

For each of the above commands,

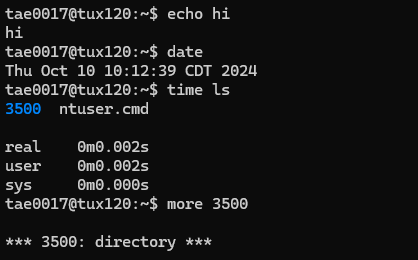
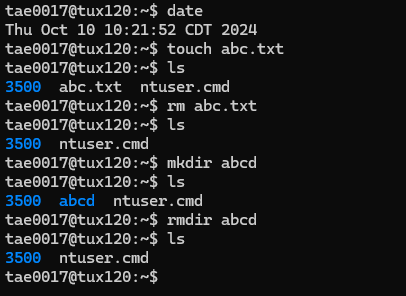
1) Provide a brief description (you may use the **man** command, but use your own words ultimately)

2) Execute the command on a Tux machine. Provide a screenshot[[2]](#footnote-2).

3) Briefly comment the results/outcomes of the execution

4) Report any unexpected behavior.

No need to provide a screenshot

**echo** – The echo command echoes or displays lines of text and outputs any argument that is being passed. For example, running *echo hi* will just display the output *hi*.

**date** – The date command displays the current date and time. For example, running the *date* command will give us today’s date (ex: Thu Oct 10 11:35:24 CDT 2024). If the system’s clock isn’t right, it will display the wrong time.

**time** – The time command measures the time it takes for commands to execute. For example, running *time ls* will report how long it took for the *ls* command to execute, splitting into real, user, and sys (system) times.

**more** – The more command displays more contents of a file or directory one screen at a time. For example, running *more 3500* will display an output of *\*\*\* 3500: directory \*\*\**.

**rm** – The rm command will remove files from the system. For example, running *rm test* will remove the file called *test* if it exists. However, it will not explicitly tell you it removed the file unless you use the *ls* command after.

**rmdir** – The rmdir command will remove empty directories from the system. For example, running *rmdir abcd* will remove the directory called *abcd* if it is empty. However, it will not explicitly tell you it removed the directory unless you use the *ls* command after.

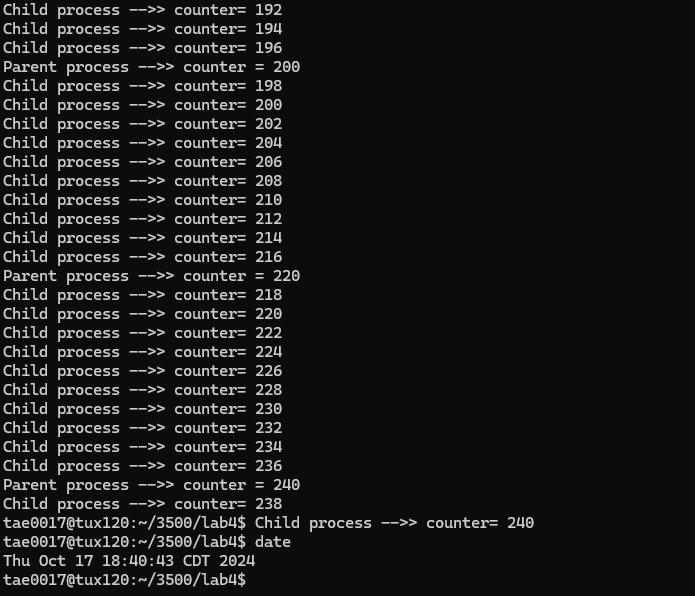
Task 2 (70 points)

The objective of this task is to show you how the shared data problem can lead to incorrect results. Recall that the shared data problem arises when concurrent processes modify shared variables. This task is designed to highlight these conditions. The instructor wrote three programs lab41.c, lab42.c, and lab43.c. These three programs illustrate these conditions, problems, and solution. You must work these hands-on laboratory exercise on a Tux machine.

# Exercise 1 (20 points): For this exercise, you must work with the lab41.c program. This program has a parent and child processes supposedly *sharing* a variable. This program is *intended* to increment the shared (common) variable counter named *\*countptr*. The parent process is *supposed* to increment *\*countptr* by increments of 20 while the child increments by 2s. A satisfactory execution of this program may be: the child increments the counter *\*countptr* twice (reaching 4), then the parent increments the counter *\*countptr* thrice to reach 64 ...and so on. When these processes print, we should see the variable always increasing by jumps of 2 or 20. The instructor proposes the program lab41.c and pretends that his program meets the above requirements. Your task is to decide whether this is true or not. You must provide sound arguments to support your decision. Download the program *lab41.c*. Compile it (cc -o lab41 lab41.c) and execute it (./lab41). Examine the C code and observe the output. Read about the fork() system call. You may consider watching this short [Youtube video](https://www.youtube.com/watch?v=xVSPv-9x3gk) about fork(). *In short, the fork() system call duplicates the parent. The duplicated process is called the child process. This child process is an exact copy of the parent, except that it has a different process ID. Furthermore, the parent and child processes have completely independent memory space. If the parent modifies any of its variables, the "same" variables of the child do not change. Similarly, if the child modifies any of its variables, the "same" variables of the parent do not change*. Answer the following questions:

1) (6 points) Does the program lab41.c really execute as supposed (or intended)? Describe the outputs observed. Justify and explain your observations.

When running the program lab41.c, the program does not execute as intended. The numbers produced in these outputs are correct, but the parent and child processes do not modify the \*countptr variable in a synchronizing way. Instead, there are random jumps in the output, such as how the parent process reads 200 after the child process reads 196 but before it reads 198. I believe it’s because of the fork() system call and how it has both the parent and child processes operating in independent memory spaces.



2) (8 points) Is the variable \****countptr*** really a shared (common) variable? In other words, are the changes made to \**countptr* by the child visible by the parent, and *vice versa*?  Describe/Justify/Explain.

No, the variable \*countptr is not really a shared variable. The fork() system call duplicates the parent and creates a child process, and they each have their own independent memory spaces. There is no way to synchronize these variables and the changes that are being made are not visible in each of these processes.

3) (6 points) Can the shared data problem arise with the program lab41.c? Answer/Justify/Explain

No, the shared data problem does not arise with the program lab41.c because they don’t have any shared data. In other words, each process is independent from one another and any updates to \*countptr from one process won’t affect the other. The shared data problem would only happen if both processes shared the same memory space.

**Exercise 2 (30 points)**:

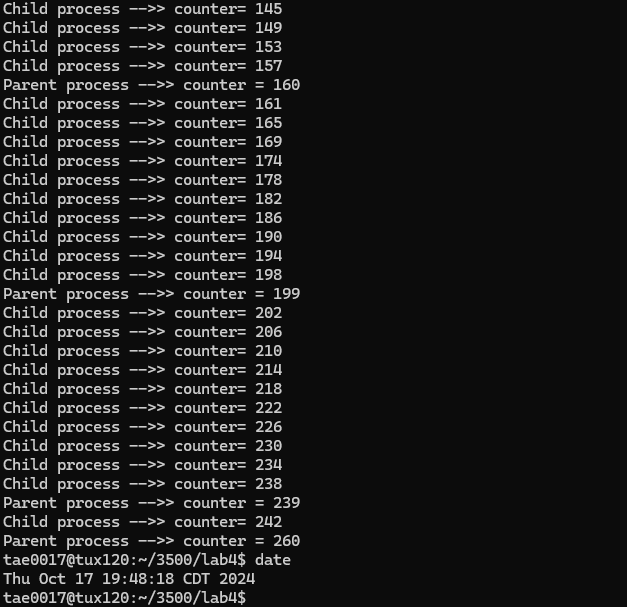
For this exercise, the instructor modified lab41.c to create a genuine **shared** variable \**countptr*. The variable \**countptr* is now shared by the parent and the child. If the parent process modifies \**countptr*, the child process "sees" the change. Similarly, if the child process modifies \**countptr*, the parent process "sees" the change. By modifying lab41.c, the instructor created a new program lab42.c. You do not need to understand the details about how the shared variable is created. Download, compile **(cc -o lab42 lab42.c)**, and execute **(./lab42)** the program lab42.c.

1)   (6 points)    Based on the execution, show that \**countptr* is now a genuine shared variable (*countptr* points to a zone shared by the parent and the child). Now, are the changes to \**countptr* made by the child visible by the parent?

Yes, \*countptr is now a genuinely shared variable and the changes made to \*countptr by the child are now visible to the parent. There is now a shared memory space that is accessible by both processes and they can both see each other’s process updates.

2) (24 points)  Does the program really execute as supposed (or intended), i.e, the counter increases exclusively in increments of 2 or 20? Explain what is happening.

No, and despite the shared variable, the program still does not execute as intended because the counter does not increase in increments of 2 or 20. Instead, the child process will increase in increments of 4 (except for one increase by 5) and the parent processes are incremented by completely different numbers.



**Exercise 3 (20 points)**:

The instructor decided to use the *Peterson Solution* to synchronize the parent and the child processes such that the CPU scheduler does not context switch when the procedure (method) add\_n is executed. The idea is to set the method add\_n as the critical region. For this, the instructor modified the program lab42.c to produce the program lab43.c. In the program lab43.c, the instructor created the variables \*turnptr, \*Interested0ptr, and Interested1ptr. In this exercise, your mission is to decide whether the instructor did a good job. If he did not implement correctly the Peterson Solution, you must correct the entry code or the exit code that may be buggy. Download, compile **(cc -o lab43 lab43.c)**, and execute **(./lab43)** the program lab43.c.

1. (10 points) Does the program really execute as supposed (or intended), i.e, the parent and the child processes cooperate to increase \*counter by jumps 2 or 20? Explain what is happening.

No, the program doesn’t execute as intended; even though the variable is now shared under the same memory space, the child and parent processes don’t cooperate to increase \*countptr by jumps of 2 or 20. However, this new program is working better than expected, as the child process is only being implemented once before and after the parent processes instead of being implemented 10 times in the other programs.

**A screen shot of a computer

Description automatically generated**

2) (10 points) The instructor did not implement correctly the *Peterson’s* *solution* to correct the program ***lab42.c***. to execute as intended: the variable should increase by 2’s or twenty’s. Locate in C program the exit and entry codes in the parent and child processes. Try to correct, if necessary, these entry and exit codes (Refer to the lecture). Describe the changes you made to the program lab43.c. For this spell out each buggy instruction and spell out the change you made.

* Interested1ptr = 1 (original); Interested1ptr = 0 (changes for exit code of parent function)
* \*Interested0ptr = 0 (original); \*Interested0ptr = 1 (changes for entry code of child function)
* Added another loop to both child and parent functions that makes \*Interested0ptr = 0 when \*countptr >= nloop and added break statements that stop both processes after the parent function hits 240 on the counter.

**Do not hesitate to ask questions on Piazza if you have any doubt.**

**Common mistake**

Starting the hands-on lab at the last minute.

**What to turn in?**

Two files:

a) **Electronic copy** of **this** file that includes your answers. I repeat: you must insert your answers in **this** file. Do not delete anything from this file. This file with your answers must be put posted **separately** on Canvas (not in a zipped folder).

Good writing and presentation are expected.

b) **Modified C Program lab43.c.** This is the C program containing your modifications of the Peterson's Solution.

**In case of doubt, do not hesitate to ask questions on Piazza.**

What you need to turn in:

* Electronic copy of this file (including your answers) (standalone). Submit the file as a Microsoft Word or PDF file.
* Recall that answers must be well written, documented, justified, and presented to get full credit.
* How this assignment will be graded:
* A right answer will get full credit when:
* It is right (worth 25%)
* It is right AND neatly presented making it easy and pleasant to read. (worth 15%)
* There is an obvious and clear link between 1) the information provided in the exercise and in class and 2) the final answer. A clear link is built by properly writing, justifying, and documenting an answer (worth 60%).
* Calculation mistakes will be minimally penalized (2 to 5% of full credit) while errors on units will be more heavily penalized.
* You are welcome/encouraged to discuss exercises with other students or the instructor. But, ultimately, personal writing is expected.

**Appendix**: Grading: What is an OBVIOUS and CLEAR LINK?

Here is an example to explain what an **obvious and clear link** is and how we grade your work.

Consider the following problem:

"(100 points) John travels from Auburn to Atlanta in his car at a speed of 50 mph. Leaving at 8am, at what time will John reach Atlanta".

Here are the answers of three students and their scores:

**Student 1** answers: "10am". Student 1 will get 25 points.

**Student 2**answers : "John will reach Atlanta at 10am". Student 2 will get 25+15 = 40 points

**Student 3** answers: "The time t to travel a distance d at speed v is equal to d/v = d/50mph. The problem does not provide the distance d from Auburn to Atlanta. Based on Google, the distance from Auburn to Atlanta is approximately 100 miles (**document is here**). Therefore, the time t = 100 miles/50mph = 2 hours. Since John left at 8am, he will then reach Atlanta at 8am + 2 hours = 10 am".

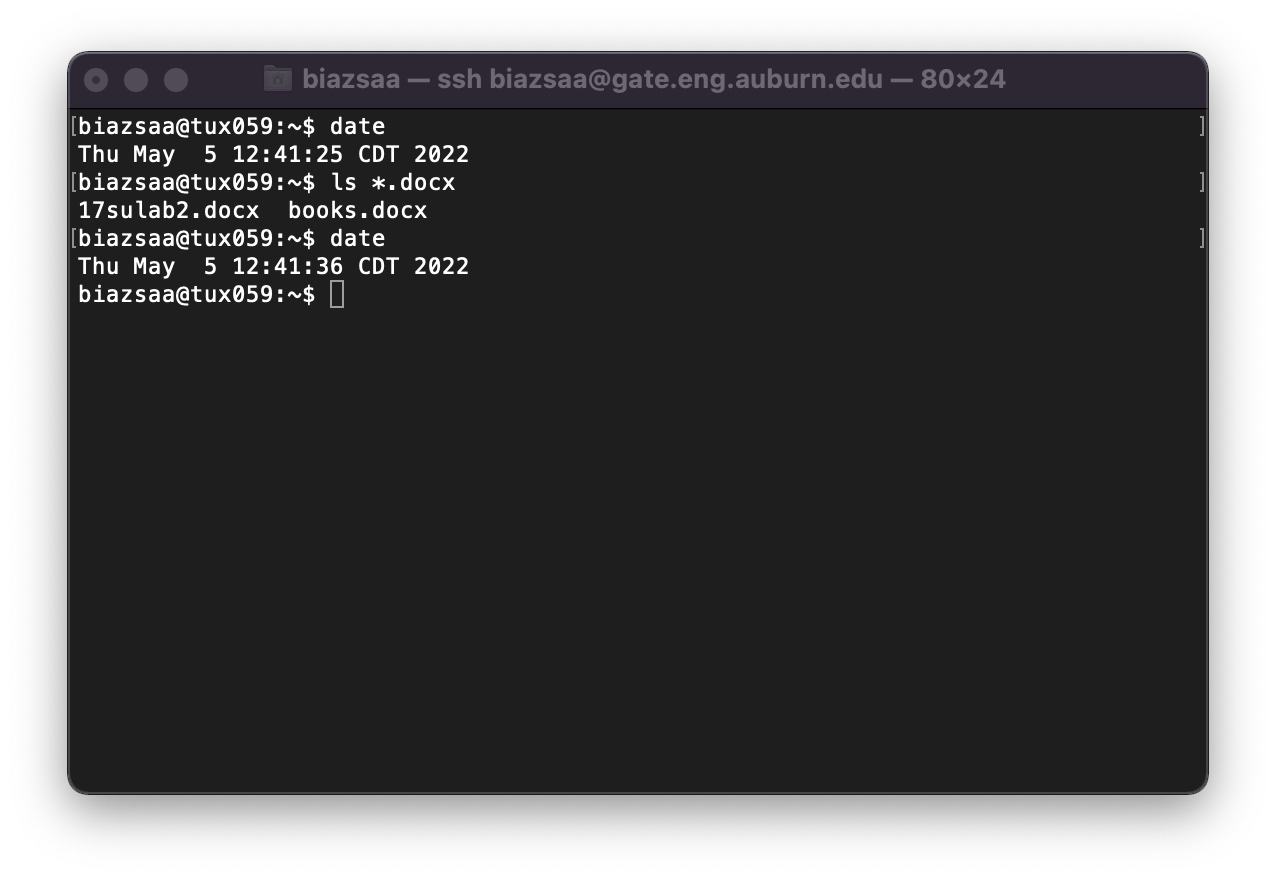
**Student 3** will get 25 + 15 + 60 = 100 points

Do you see the **direct** **link** going from the data provided in the question to the final answer, using general knowledge/formula and documents?.... Can you now solve the following problem and get 100 points?

"(100 points) Alice travels from Auburn to Atlanta in her car at a speed of 50 mph. Leaving at 8am, at what time will Alice reach Atlanta assuming that she had a flat tire that delayed her 30 minutes".

**Screenshot: Required Information**

**In order to save space, for this assignment and all *FUTURE* ones, clip out the screenshots to contain only the relevant information. *When applicable, ALL screenshots must show the date, the Tux machine you are using for the exercise and the Auburn username of one of the team mates*. Make sure that the screenshots are easily readable. Below is template screenshot:**

****

1. See on the appendix what an obvious and clear link is. [↑](#footnote-ref-1)
2. Recall the required information for ALL screenshots. See Appendix. [↑](#footnote-ref-2)