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**

Description: The echo command is used to display lines of text or variables. It's commonly used in scripting to show the value of variables or to provide a message to the user. When you run echo followed by a string, it will simply print that string to the terminal. For example, echo "Hello, World!" will display "Hello, World!".

A black screen with white text

Description automatically generated with low confidence

**Date**

Description: This command displays the current date and time. When you execute date, it shows the current date and time based on your system's time zone settings. The format usually includes day, month, date, time, time zone, and year.

A black screen with white text

Description automatically generated with low confidence

**Time**

Description: The time command is used to measure the time taken by a command to execute. When you prefix a command with time, it executes the command and then displays the time taken to execute it. For example, time ls will list the contents of the directory and then show how long it took.

A screen shot of a computer

Description automatically generated with low confidence

**More**

Description: This command is a file pager that lets you view the contents of a file one screen at a time. When you run more followed by a filename, it displays the contents of the file. You can navigate through the file with the enter key or space bar. If the file is short enough to fit on one screen, more will display it all at once, similar to the cat command.

A screenshot of a computer program

Description automatically generated with medium confidence

**Rm**

Description: This command is used to delete files. Running rm followed by a filename will delete that file. Be cautious, as this deletion is permanent and cannot be undone. Without flags, rm won't delete directories or write-protected files, prompting for confirmation.

**Rmdir**

Description: The rmdir command is used to delete empty directories. When you execute rmdir followed by a directory name, it will delete the directory if it's empty. If the directory is not empty, rmdir will fail and return an error.

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.

Answer here ... (Include a screenshot)

**A screen shot of a computer

Description automatically generated with low confidence**

The expected behavior of the output should be showing the counter increment in values of 2 and 20 respectively by the child and processes. When the fork() happens it creates a separate process with its own space in memory. “\*countptr” is put onto the heap, which in turns means that each child process will have its own version of the counter variable. This also means that the increments made by the child process will not be visible to the parent process and vice versa. The child and parent processes would need a shared variable to correctly point to.

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.

Answer here ...

In lab41.c the variable “\*countptr” is not a true shared variable between the parent and child processes. Since each process has its own version/copy of “\*countptr”, the counter will never reflect the correct value. Thus, even though \*countptr appears to be a shared pointer in the code, in reality, after the fork, there are two distinct \*countptr variables - one for each process.

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

Answer here ...

Since the parent and child processes work on separate copies of \*countptr, there is no shared data problem in the traditional sense as in there are no created race conditions. Each process will independently modify its own copy of the variable without affecting the other. The shared data problem typically associated with concurrent processes accessing and modifying the same memory location does not occur in this scenario due to the separate memory spaces post-fork.

**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?

Answer here ...

Based on the observed execution, it can be seen that the variable “\*countptr” is now a genuine shared variable between the two processes. The child and parent processes have access to the same memory locations, pointed at the variable. When a change is made by the child process to the variable “\*countptr”, the updated version of the variable can be seen by the parent process and vice versa.

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.

Answer here ... (Include a screenshot)

**A screen shot of a computer screen

Description automatically generated with low confidence**

No, the program still does not fully work as intended by the description given. The counter does not increase exclusively in increments of 2 or 20. The busy wait loop introduces an unpredictable delay, and the increments performed by the processes can be overlapped or be mixed up at different points of execution.

**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.

Answer here ... (Include a screenshot)

**A screen shot of a computer screen

Description automatically generated with low confidence**

No, the program still does not work as intended. The parent and child processes do not cooperate correctly to increase the variable “\*countptr” by either jumps of 2 or 20 respectively. The issue still lies with how the Peterson Solution is implemented in the code. The problem is with the entry and exit points of the critical section. The entry code tries to enforce mutual exclusion between the two processes by setting the “Interested0” and “Interested1” variables and entering a while loop for busy waiting. However, the condition in the while loop is incorrect and does not properly handle the turn variable. The exit code for releasing the critical section is also incorrect. It does not correctly release the ownership of the critical section back to the parent process.

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.

Answer here ...

By making these changes below, the program should execute as intended. The parent and child processes will cooperate correctly to increase “\*countptr” by jumps of either 2 or 20 while maintaining mutual exclusion.

First the condition is the while loop of the entry code needed to be added to in order to properly deal with other factors to maintain mutual exclusion.

Add the below code to the entry code.

While (\*Interested1ptr == 1 && \*turnptr == 1) {

If (\*turnptr != 1) {

\*Interested0ptr = 0;

While (\*turnptr !=0);

\*Interested0ptr = 1;

}

}

Change the while loop condition in the parent process entry code:

while (\*Interested0ptr == 1 && \*turnptr == 0);

updated version:

while (\*Interested0ptr == 1 && \*turnptr == 0) {

if (\*turnptr != 0) {

\*Interested1ptr = 0;

while (\*turnptr != 1);

\*Interested1ptr = 1;

}

}

Now the exit code in the child and parent processes must be adjusted for these changes to reflect.

Change the exit code in the child process:

\*Interested0ptr = 0;

Updated version:

\*turnptr = 1;

\*Interested0ptr = 0;

Now change the exit code in the parent process:

\*turnptr = 0;

\*Interested1ptr = 0;

These changes should reflect the proper handling of the turn variable and also performs the necessary waiting and “letting go” of the critical section of the program. Below I will show the complete program with the updated versions of the while loops for both the entry code and exit code for the program to run as intended.

Updated COMPLETE code for lab43.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

int nloop = 240;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

\* Function: increment a counter by some amount one by one \*

\* argument: ptr (address of the counter), increment \*

\* output : nothing \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void add\_n(int \*ptr, int increment){

int i,j;

for (i=0; i < increment; i++){

\*ptr = \*ptr + 1;

for (j=0; j < 7000000;j++);

}

}

int main(){

int pid; /\* Process ID \*/

int \*countptr; /\* pointer to the counter \*/

int \*turnptr, \*Interested0ptr, \*Interested1ptr; // Peterson's Solution variables

int fd; /\* file descriptor to the file "containing" my counter \*/

int fd\_turn, fd\_Interested0, fd\_Interested1;

int zero = 0; /\* a dummy variable containing 0 \*/

system("rm -f counter");

system("rm -f turn");

system("rm -f Interested0");

system("rm -f Interested1");

/\* create files which will "contain" my shared variables \*/

fd = open("counter",O\_RDWR | O\_CREAT);

write(fd,&zero,sizeof(int));

fd\_turn = open("turn",O\_RDWR | O\_CREAT);

write(fd\_turn,&zero,sizeof(int));

fd\_Interested0 = open("Interested0",O\_RDWR | O\_CREAT);

write(fd\_Interested0,&zero,sizeof(int));

fd\_Interested1 = open("Interested1",O\_RDWR | O\_CREAT);

write(fd\_Interested1,&zero,sizeof(int));

/\* map my files to memory \*/

countptr = (int \*) mmap(NULL, sizeof(int),PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd,0);

turnptr = (int \*) mmap(NULL, sizeof(int),PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd\_turn,0);

Interested0ptr = (int \*) mmap(NULL, sizeof(int),PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd\_Interested0,0);

Interested1ptr = (int \*) mmap(NULL, sizeof(int),PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd\_Interested1,0);

if ((!countptr) || (!turnptr) || (!Interested0ptr) || (!Interested1ptr)) {

printf("Mapping failed\n");

exit(1);

}

\*countptr = 0;

\*turnptr = 0;

\*Interested0ptr = 0;

\*Interested1ptr = 0;

close(fd);

close(fd\_turn);

close(fd\_Interested0);

close(fd\_Interested1);

setbuf(stdout,NULL);

pid = fork();

if (pid < 0){

printf("Unable to fork a process\n");

exit(1);

}

if (pid == 0) {

/\* The child increments the counter by two's \*/

while (\*countptr < nloop){

// Entry to Critical Region

\*Interested0ptr = 1;

\*turnptr = 1;

while (\*Interested1ptr == 1 && \*turnptr == 1) {

if (\*turnptr != 1) {

\*Interested0ptr = 0;

while (\*turnptr != 0);

\*Interested0ptr = 1;

}

}

// Critical Region

add\_n(countptr,2);

// Exit code out of Critical Region

\*turnptr = 1;

\*Interested0ptr = 0;

printf("Child process -->> counter= %d\n",\*countptr);

}

close(fd);

close(fd\_turn);

close(fd\_Interested0);

close(fd\_Interested1);

}

else {

/\* The parent increments the counter by twenty's \*/

while (\*countptr < nloop){

// Entry to Critical Region

\*Interested1ptr = 1;

\*turnptr = 0;

while (\*Interested0ptr == 1 && \*turnptr == 0) {

if (\*turnptr != 0) {

\*Interested1ptr = 0;

while (\*turnptr != 1);

\*Interested1ptr = 1;

}

}

// Critical Region

add\_n(countptr,20);

// Exit code out of Critical Region

\*turnptr = 0;

\*Interested1ptr = 0;

printf("Parent process -->> counter = %d\n",\*countptr);

}

close(fd);

close(fd\_turn);

close(fd\_Interested0);

close(fd\_Interested1);

}

}

**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".

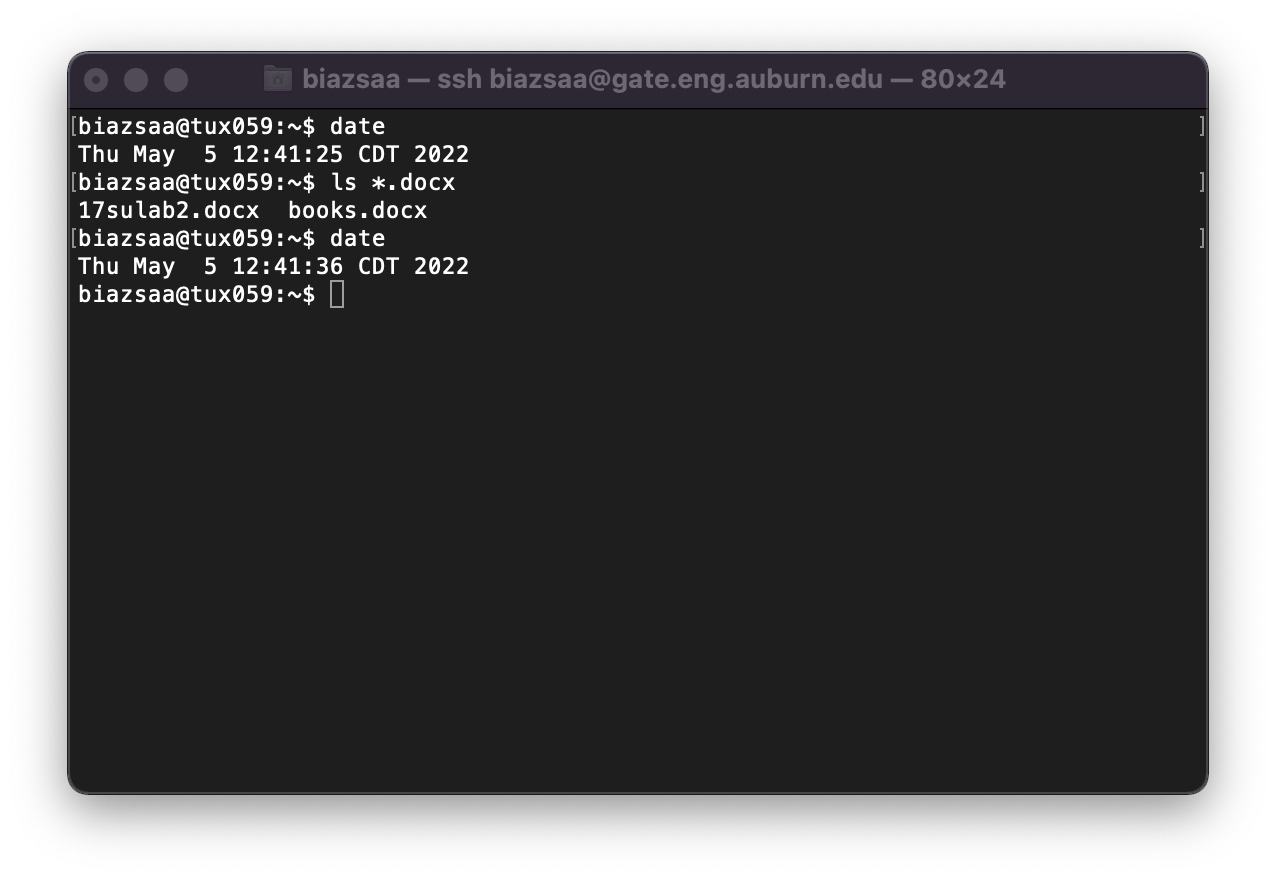
**Student 3** will get 25 + 15 + 60 = 100 points

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)