**Late Submission** : as specified in the syllabus. Late day counting starts one millisecond after the deadline.

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

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

**Instructions** For this assignment, you must implement a ***task*** to evaluate the ***fairness*** and ***reliability*** of the CPU scheduler. The instructor will provide you with a *shell* in C. Coding the *task* inside the provided shell will require about 100 lines. Your C program must compile and execute on the Unix machines on the Engineering Network System (tux machines). Otherwise, no credit will be awarded. These machines can be accessed with *ssh* through the host ***gate.eng.auburn.edu***. The instructor will provide videos about how to log to Tux machine and move files to/from your local machine.

Objectives of this assignment:

* to work on a Unix based system
* to “*dust off*” your programming skills in C
* to become familiar with the notion of a *process*
* to assess the *fairness* and *reliability* of the CPU scheduler

What You Need To Do:

* Examine and understand a provided C source program (shell)
* Design a task that meets specific requirements
* Implement the task inside the shell
* Run experiments of the shell augmented with your task .
* Collect and process data about the experiment
* Build plots
* Interpret and discuss the plots

**IMPORTANT:**

1. *Your code will be tested and graded* ***REMOTELY*** *on the Engineering Unix (Tux) machines. If the code does not compile or run on those machines, you will not get any credit.*
2. *One submission per group.*
3. *Writing and presentation of your report are considered to grade your lab (60%). Your conclusions* ***must be supported*** *by the data/measurements you collect. Your conclusions must be correct.*
4. *The quality of your code will be evaluated (20%).*
5. *The quality of the task you design (20%)*
6. ***Questions about this lab must be posted on Piazza if you need a timely answer****. Do not post your code*
7. ***Work ahead to get early feedback from your instructor to improve your code and your report to achieve the best score. Based on experience, if you write the report at the last minute and do not get any feedback, the score is often in the low 40%.***

**Lab Assignment (Turned in by one group mate)**

It is assumed that by the 10th "class day",

**first**, 1) you have an engineering Unix account, 2) you can edit text files, 3) you can compile C programs, and 4) you can execute C programs on the Unix (Tux) machines. You can use any personal computer or computing lab to remotely access the Engineering Unix (Tux) machines. You can use **ssh** to access **gate.eng.auburn.edu** or **secureCRT** on a Windows machine or on **rdp.eng.auburn.edu**. Ask for help if you do not know how to use these tools.

**second,** you joined a Canvas group (even alone):

**Look at the “How to get started?” section at the very end of the lab.**

This lab has multiple parts: 1) Compile, execute and understand the provided *shell*, 2) design and implement a task meeting specific requirements, 3) run experiments, 4) collect data, and 5) analyze and report your results. An efficient code is a code that 1) is correct, , 2) is concise, 3) does not waste memory, and 4) is well documented.

The instructor designed and implemented in C a shell in which you will insert the task you must design and implement. Look for the word **"\_Student**" in the C code, you will find where you must insert your code.

**The Shell**

Below is the pseudocode of the shell:

**Inputs**:

**np**: number of concurrent processes to execute

**nm**: number of minutes each process runs

**Output**:

Displays when each process starts executing the student's task

(student's responsibility)

Displays "fraction/amount" of task executed by each process

Collects (writes to a file) (done by each process)

Displays the amount of time the process "ran" on the CPU (done by each process)

Process the data collected by each process. (Done by parent process)

Displays mean and variance of (Done by Parent process)

**Code (Parent)**:

for i=1 to ***np***

create (fork) a process Pi

for i=1 to ***np***

"green light" Pi to start its task

Compute and displays mean and variance of collected by children processes.

**Code of Process Pi (Child Process):**

wait for green light

Display start time

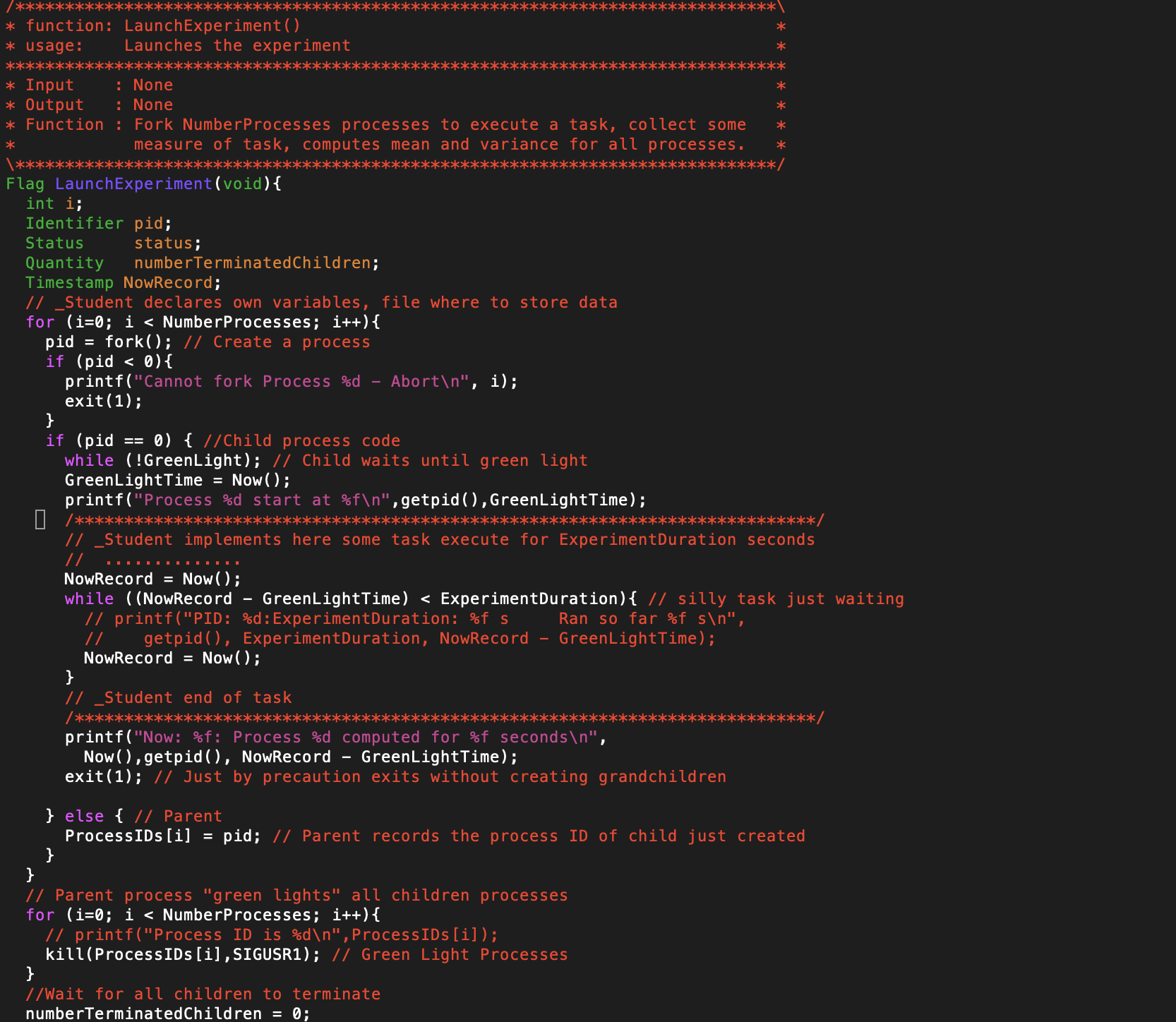
Execute Student's Task for ***nm*** minutes

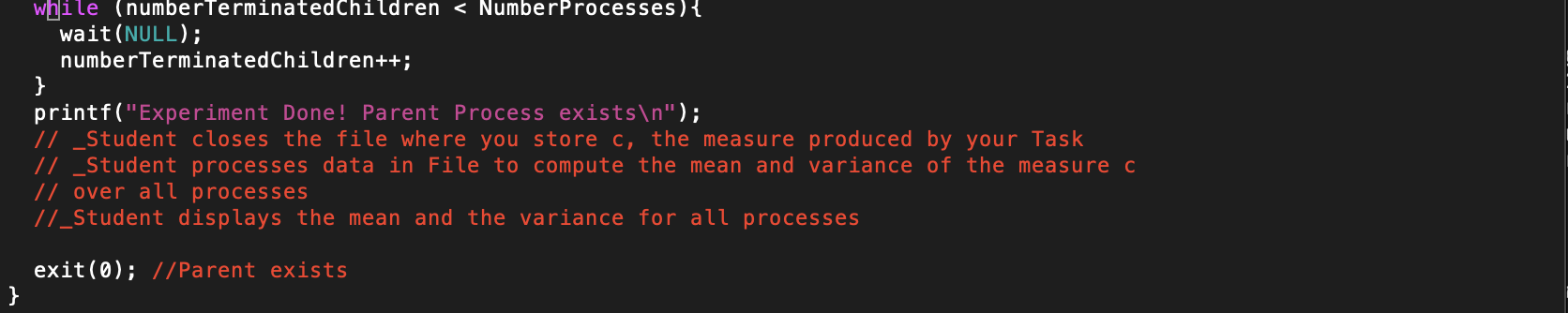
(student's responsibility) Displays "fraction/amount" of task executed by each process

Displays the amount of time the process competed for the CPU (should be ***nm*** minutes)

Collects (writes to a file)

Below is the C implementation (Do not hesitate to ask questions):





Task Requirements

The objective is to design a **simple** Task that allows to somewhat produce a number that measures how much Process Pi used the CPU during **nm** minutes. When the task ends, it must display the number and write it to a file.

Your task will be graded based on the following requirements/criteria assuming that the CPU scheduler is fair and reliable under a constant (computing) load:

1) If we run one process multiple times for the same number of **nm** minutes, the number should be the same each time within

2) If we run **np** processes multiple times for ***k\* nm*** minutes, the number displayed should be times larger than the number displayed for running **nm** minutes within . For example, if I run 20 processes for minutes, the number displayed should be times larger than the number displayed for running 20 processes for minutes within .

3) If we run **100**processes multiple times for ***2*** minutes, the number displayed should be times larger than the number displayed for running **1000** processes for **2**minutes within . For this requirement, your code will be tested with these values **100** and **1000**.

To compile the provided shell program (named evaluatingCPUScheduling.c), you must type:

***cc -o eval evaluatingCPUScheduling.c***

where

***eval***  is the executable (produced if the compilation is successful).

***evaluatingCPUScheduling.c*** is the shell source file in which you must *insert* your task.

To execute eval for 3 processes during 2 minutes, you must type:

./eval 3 2

**YOU CAN ONLY ADD*:***

***1) your task where indicated inside the procedure LaunchExperiment***

***2) your variables for your task where indicated inside the procedure LaunchExperiment***

**YOU CANNOT add, delete, or modify anywhere outside LaunchExperiment. YOU CANNOT delete or modify original variables and instructions of the shell inside LaunchExperiment. If you do not follow these guidelines, there will be a systematic 50 points penalty.**

2) **CPU Scheduling Evaluation:**

The objectives are :

a) to check whether your task meets the requirements

b) to evaluate the **fairness** and the **reliability.**

We describe below the experiments you must conduct to achieve the objectives.

a) Checking whether your task meets the requirements

Requirement 1 (6 points):

- Run eval 5 times during 2 minutes with np = 1 (one process). For each run of eval, collect and report here the 5 values you collected for . Are the five values within of each other? If not, your task does not meet Requirement 1.

Requirement 1I 1 (7 points):

- **First**, Run eval once during 2 minutes with np = 20 (20 processes). Collect the mean (for all 20 processes) and report it. **Second**, Run eval once during 3\*2 (6) minutes with np = 20 (20 processes). Collect the mean (for all 20 processes) and report it Is within? If not, your task does not meet Requirement 1I.

Requirement 1I1 (7 points):

- **First**, Run eval once during 2 minutes with np = 100 (100 processes). Collect the mean (for all 100 processes) and report it. **Second**, Run eval once during 2 minutes with np = 1,000 (1000 processes). Collect the mean (for all 1,000 processes) and report it Is within? If not, your task does not meet Requirement 1II.

a) (30 points) **Fairness**: If the CPU scheduler is **fair**, the measure **c** (number displayed by your task) must be close for all processes you run within the **same** experiment. We call an *experiment* the fact to run the program *eval* **once**.

In order to evaluate fairness, you must conduct one experiment during 2 minutes for each of the following values of **np** (number of processes): 2, 5, 10, 20, 40, 60, 80, 100, 200, 400, 500, 1000. For each experiment with value , compute the variance of the measure c your task displays for each process. *See the appendix about how to compute the variance*. Plot the variance of c versus the number of processes (put the np values 2, 5, 10, 20, 40, 60, 80, 100, 200, 400, 500, 1000 on the x-axis and the variance on the y-axis) Discuss this plot and draw reasonable conclusions regarding fairness. In other words, is the CPU scheduler of the Tux machines fair based on the data you collected?

b)(30 points) **Reliable**: If the CPU scheduler is **reliable**, the average measure **c** (number displayed by your task) for **np** processes must remain the same (within .) for successive experiments running for the same number of processes **np** each time. In other words, if you execute the same process on a machine, the execution time should not widely vary: the execution time should not be one time 2 minutes, then 2 hours, then 1 minute, then 30 minutes.....

In order to evaluate reliability, run 10 times eval with np=100 (100 process) during 2 minutes (./eval 100 2). Report here the mean of c for each run. Plot the mean versus the run # (put the values 1, 2, ....10 on the x-axis) and the mean on the y-axis. Discuss this plot and draw reasonable conclusions regarding reliability. In other words, is the CPU scheduler of the Tux machines reliable based on the data you collected?

**Get Started**

1. compile the code I provided you by typing:

cc -o eval evaluatingCPUScheduling.c

1. Execute the code (evaluate 1 process during 2 minutes): ./eval 1 2
2. Observe what is displayed
3. Open and examine the shell evaluatingCPUScheduling.c
4. Focus on the routine LaunchExperiment and try to understand what it does and how it works
5. Insert some task where indicated (search for \_Student)
6. Think about a simple task to insert that would meet the requirements outlined.
7. Copy evaluatecpuscheduling.c onto MyevaluateCPUScheduling.c. MyevaluateCPUScheduling.c is the shell augmented with the task you designed and implemented. This is the file you will turn in.
8. Compile and execute to check whether your task meets the requirements
9. Make sure to complete all required tasks.
10. Do not hesitate to ask questions on Piazza or during office hours.

**Common mistakes**

1) Finish the code at the last minute: you will not have enough time to collect data and write a good report

2) Finish the la/reportb at the last minute without any feedback from the instructor: expect a grade below 50%.

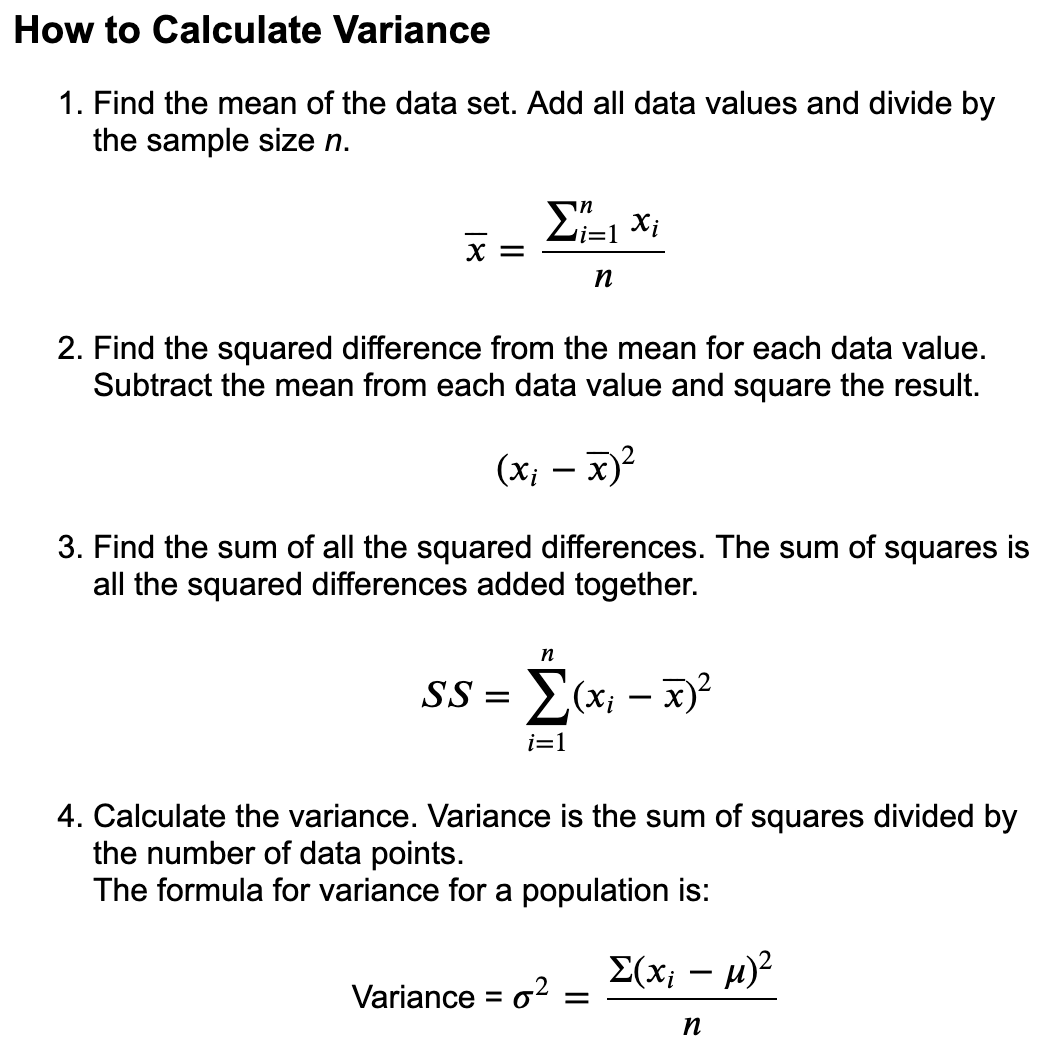
**What to turn in?**

Two files a) **Electronic copy** of **this** file that includes your answers and b) the C source code **MyevaluateCPUScheduling.c**. I repeat: you must insert your answers and plots in **this** file. Do not delete anything from this file. This file and the C source code must be put posted **separately** on Canvas (not in a zipped folder). **A penalty of 30 points will be applied if these instructions are not followed.**

Good writing and presentation are expected.

In case of doubt, do not hesitate to ask questions on Piazza or during office hours.

**Appendix**

****

**Answers:  
  
Chart, line chart

Description automatically generated**

**Something has fairness if its results are independent of given variables (https://en.wikipedia.org/wiki/Fairness\_(machine\_learning)). That is, our inputs should produce similar outputs no matter what the inputs are. In our line graph, the points are close to creating a straight line, which would imply uniformity. Furthermore, if you would look at each number c on its own, you would find that they are each within +-10% of each other. Therefore, since the numbers are within that percentage of each other and the graph is close to straight, then we can conclude that the Tux machine CPU scheduler is fair.**

**Something is reliable if its measure is consistent over many repetitions (https://opentextbc.ca/researchmethods/chapter/reliability-and-validity-of-measurement/). That is, no matter how many times we perform the same experiment, our values should remain close to one another. In our line graph, the points are close to creating a straight line, which would imply uniformity. Furthermore, if you would look at each number c on its own, you would find that they are each within +-10% of each other. Therefore, since the numbers are within that percentage of each other and the graph is close to straight, then we can conclude that the Tux machine CPU scheduler is reliable.**