COMPE571 – Embedded Operating Systems – Fall 2025 – Programming Assignment 1

NOTE: You should form groups of 2 or 3 for this assignment.

NOTE: You have to work in a UNIX environment (e.g. Ubuntu, MacOS, Raspbian OS, etc.) to implement this assignment. You have to use C programming.

Assignment Description

In this programming assignment, you are going to compare sequential programming against multithreading and multitasking. Your multithreading and multitasking implementations will feature task parallelism, task synchronization (and depending on your implementation, shared resource protection), and inter-process communication.

You have a main WORKLOAD, where you have to calculate the sum of variables from 0 up to `N` (not inclusive). You have to implement this WORKLOAD in the following cases:

- **Case 1: Baseline:** This case implements the WORKLOAD in a sequential way. In this case, implement a C file that performs the WORKLOAD without any parallelism.
- **Case 2: Multithreading:** This case implements the WORKLOAD in a multithreading way. The WORKLOAD should be divided evenly among different number threads (`NUM_THREADS`). Refer to the report section to see the values of `N` and `NUM_THREADS` (we made sure that `NUM_THREADS` evenly divides `N`). The main C file should create multiple threads, distribute the WORKLOAD among those threads, synchronizes with respect to the completion of all threads, accumulates their results, and reports the final result. You are to use the pthread library for this case, specifically, the following functions:
  - `pthread_create()`
  - `pthread_join()`
- **Case 3: Multitasking:** This case implements the WORKLOAD in a multitasking way. The WORKLOAD should be divided evenly among different number tasks (`NUM_TASKS`). Refer to the report section to see the values of N and NUM_TASKS (we made sure that `NUM_TASKS` evenly divides `N`). The main C file should create multiple tasks, distribute the WORKLOAD among those tasks, synchronizes with respect to the completion of all tasks, accumulates their results, and reports the final result. You have two options to implement the multitasking case (you can choose either one):
  - **Option 1: Creating multiple tasks via `fork()`:** In this option, your main C file creates multiple tasks via `fork()`. You have to keep track of the child and parent tasks across different `fork()` calls. You have to keep in mind that when a new task is created via `fork()`, the address space of the child task is separate than the parent task, i.e. you have to use inter-process communication for sending/receiving intermediate results. The recommended method for inter-process communication is **pipes**. You are to use the following functions in your implementations:
    - `fork()`
    - `wait() or waitpid()`
    - `pipe()`
    - `dup()`

    - Please take a look at the man pages for the `pipe()` and `dup()`.
  - **Option 2: Creating multiple tasks via `popen()`:** In this option, your main C file creates multiple tasks via `popen()`. `popen()` function both starts a new task and opens a pipe, in the form of a file descriptor, so that the main C file can receive information from the newly created task. You are to use the following functions in your implementations:
    - `popen()`
    - `fscanf()` (this is to read from a file descriptor created via `popen()`)
    - Please see the man page for `popen()`.

Important Notes:

1. Your baseline implementation should be very simple, around 10 lines of code.
2. You can have a single C file for multithreading option where `NUM_THREADS` can be a variable in your single C file. Your code can be modular in the sense that your code should not need any change when `NUM_THREADS` changes. If you use a common global variable among multiple threads to accumulate the sum, make sure that you implement appropriate shared variable protection mechanisms (e.g. semaphores). However, you can implement your code without a shared variable among threads. HINT: make your threads *return* the calculated sum to the main task.
3. Multitasking Option 1 requires careful code writing and you will possibly need multiple versions of your code with respect to different values of `NUM_TASKS`. This is because you have to keep track of parent and child tasks across different `fork()` calls, as well as different pipes among different parent/child couples.
4. Multitasking Option 2 is expected to be easier than Multitasking Option 1. However, it is expected to work slower (why?). In this case, you can parametrize the baseline implementation in a way that it receives a lower and an upper limit as command line arguments (and then calculates the sum between the lower and upper values). Then, your main C file calls `popen()`, which creates new tasks running the parametrized baseline code with different upper and lower limits. Then the main C file obtains the partial results from the newly generated tasks via inter-process communication (i.e. using the pipes created via `popen()`).
5. You HAVE TO make sure that your multithreading and multitasking implementations work correctly, i.e. the final result of your multithreading and multitasking should match the baseline code. If your multithreading and multitasking implementations do not work correctly, you will NOT receive ANY credit for those implementations.

Measuring timing: In the report, you are to compare the execution time for your different implementations. Thus, you need to measure how long each case takes to execute. You have two options for measuring timing:

- **Option 1: Using `time` command via command line:** This option uses the `time` command in the command line when starting a new task. It reports the elapsed time in real, user, and system time when the task finishes. Please check the man page for time command. Note that you have to figure out which value among the reported values (real, user, and system) you should use in your report. One limitation of this method is that it measures the duration of the entire task, however, you might be interested in measuring a portion of the respective task for more accurate and granular time measuring (see Option 2 for this).

- **Option 2: Using `time.h` library or `time()` function within your C file:** This method gives you the option of measuring the time duration of specific parts of a C file, rather than the entire program. In this option, you can designate two points of interest, obtain timestamps at those two points, and calculate the difference between those timestamps. It enables you to have high precision time measurements. Please see the following page for different ways to implement this option. This is the *recommended* way to measure time in this assignment.

25% Additional Credit: If you implement both options for multitasking and include comparison/analysis with respect to both of them.

Report

Your report should include the following:

- Detailed explanation of implementation of different cases
  - What your baseline code is
  - How you are implementing multithreading
  - How you are implementing multitasking (which option you chose)
  - Comparison of the code size between multithreading and multitasking
- Results
  - Total execution time for each case (baseline, multitask, multithread)
    - NOTE: This should not include just a single experiment for each case. You need to make sure that you are repeating each case enough times to have statistically significant results.
    - How many times did you repeat each case?
    - Report average and standard deviation and compare how standard deviation changes among these different cases. Explain why you observe higher standard deviation in some cases, if applicable.
  - Report the execution time statistics with the following values of `N`, `NUM_THREADS`, and `NUM_TASKS`:
    - `N = {100000000,1000000000,10000000000}`
    - `NUM_THREADS = {2, 4, 8}`
    - `NUM_TASKS = {2, 4, 8}`
  - Explain what you observe when you increase `N` and why it happens that way.
  - Explain what you observe when you increase `NUM_THREADS`. Explain whether what you observe has any connection with an *underlying hardware property* of the system you are using.
  - Explain what you observe when you increase `NUM_TASKS`. Explain whether what you observe has any connection with an *underlying hardware property* of the system you are using.
  - Compare multithreading vs. multitasking. Is there a clear winner between these two? And why? Would the answer to this question change in a different system with different hardware characteristics?

        - Hint: Consider what hardware property would have a direct effect on multitasking/multithreading.
    - The values of `NUM_THREADS` and `NUM_TASKS` (with respect to the value of `N`) are selected to make your implementation easier. Discuss how your implementation would change if `NUM_THREADS` and `NUM_TASKS` would not evenly divide `N`. How would your results change?
        - Note: For this part, you do not need to implement code, but only discuss how your current implementation would change.

Deliverables

Your assignment submission must include TWO **separate** files:

- A **zip** file containing a set of functional C files representing different cases (baseline, multitask, multithread) along with a README file that shows which C file corresponds to which part of the assignment,
- Project report containing your implementation details and discussing your results. Note that you should include the main results in the report as well (as described above). Please submit your report in DOCX or PDF (preferred) format.

Your final score will be determined by the combination of your report and turning in functional C files on time. **Note that the report should be at most 4 pages long.**