University of Windsor
School of Computer Science

COMP 3300: Operating Systems Fundamentals

Assignment 1 (Winter 2025). Due: February 14, 2025

Problem #1: Develop a hybrid application that demonstrates both process creation (via fork()) and thread creation (via pthreads). (30 points)

Requirements:

- The parent process must first create a single thread that prints a message including the parent process ID and thread ID.
- The parent process then creates a child process using fork().
- **Child Process:** Creates two threads using pthreads, where each thread prints a message same format as respective parent process ID and thread ID.
- **Parent Process:** Implement two variations:
  - **Variant 1 (With wait()):** The parent process waits (wait()) for the child process to complete before terminating.
  - **Variant 2 (Without wait()):** The parent process does not use wait(), allowing the child process and parent's thread execution to overlap.
- Compare the two variants by analyzing (run each variant at least two times):
  - Differences in output order and execution timing.
  - The impact of allowing concurrent execution of parent and child processes.
  - Observed variations in process execution behavior.

Deliverables:

- **Source Code:** p1_YourName.c (Include both variants, with wait() commented while submitting).
- **Execution Output:** Screenshot demonstrating full executions, including compilation commands.
- **Diagram:** A simple process/thread execution hierarchy illustrating (by pend and paper/word tool, but non any online tool) the parent process, child process, and their respective threads.(Based on the output)

Problem #2: Investigate and compare the performance of processes and threads for executing a computationally intensive task. (40 points)

Requirements:

- Implement a computationally intensive task (e.g., computing a large Fibonacci number recursively without memoization).

  Workload Details:
  - For **Variants 1, 2, and 3**, perform **6 iterations** (i.e., compute fib(42) six times).
  - For **Variant 4**, perform **8 iterations** (i.e., compute fib(42) eight times) so that the workload can be evenly split across processes and threads.
- Implement four configurations:
  - **Variant 1:** Single Process, Single Thread. Execute all 6 iterations sequentially within a single process.
  - **Variant 2:** Two Processes, Each with a Single Thread (split workload equally, each process computes fib(42) three times).
  - **Variant 3:** Single Process with Two Threads (computation divided between two threads within the same process, each thread computes fib(42) three times).
  - **Variant 4:** Two Processes, Each with Two Threads (split workload across two processes, each spawning two threads, each process handles 4 iterations, with each thread performing 2 iterations).
- Measure performance using the ***time*** command (record real, user, and system time).

Deliverables:

- **Source Code Files:** p2_v1_yourName.c, p2_v2_yourName.c, p2_v3_yourName.c p2_v4_yourName.c
- **Execution Output:** Screenshot showing timing results for each variant, including compilation commands.
- **Written Brief Summary (100–200 words):**
  (a) Compare single vs. multiple instances.
  (b) Processes vs. threads (isolation, memory sharing, context switching).
- **Performance Table:** Summarizing timing results.

Problem #3: Implement a simple multi-threaded solution to manage computational tasks that compute different Fibonacci numbers with ordered output. (30 points)

Requirements:

- **Task Definition:**Predefine a set of tasks where each task computes a different Fibonacci number. For example:
  Task 1: Compute fib(10)
  Task 2: Compute fib(11)
  Task 3: Compute fib(12)
  Task 4: Compute fib(13)
  Task 5: Compute fib(14)
  Task 6: Compute fib(15)
  Task 7: Compute fib(16)
  Task 8: Compute fib(17)
  Task 9: Compute fib(18)
  Task 10: Compute fib(19)
- **Ordered Output:** Ensure that the results are printed in the original task order (i.e., Task 1's result is printed first, Task 2's result second, etc.), regardless of the order in which the threads complete.
- **Implementation:**
  -- Create one thread per task.
  -- Each worker thread computes its assigned Fibonacci number and stores its result in a shared or global array at an index corresponding to the task order.
  -- After all threads have completed, the main thread prints the results in the correct order.

Deliverables:

- **Source Code:** problem3_YourName.c (include clear implementation comments).
- **Execution Output:** Screenshots showing execution results and throughput statistics, including compilation commands.
- **Written Analysis (150–250 words):**
  (a) Discuss the efficiency improvements (if any, in terms of time) over sequential execution (you need to implement a equivalent sequencial solution for the comparison, but do not need to submit that code, just show the screenshot for output and timming in the PDF).
  (b) Explain how ordered output is achieved despite concurrent processing.
  (c) Analyze potential scalability challenges as the number of tasks increases.

Submission Instructions

- **Total Point: 100 [Weight: 10% of the final grade]**

- All deliverables must be submitted as a **single zipped file** containing:
  - A **PDF file** that includes:
    - Include the **configuration (cpu clock/core info/ram etc.)** of the machine on which you've implemented the programs.
    - All required output **screenshots** on your computer / nomachine (online compiler is not allowed).
    - Each screenshot must clearly show the **compilation command** immediately before executing the program.
    - Screenshots, diagram, written explanations tables and analysis, **correctly labeled** for each problem.
  - All **C source** code files.
    - Submit all source code files (.c) with **appropriate names** (as stated in the deliverables).
    - Ensure your program **compiles and runs** without errors; if it does not compile or run, it will **not be considered** for grading.
    - There will be **no consideration** for incorrect file names or submissions that fail to run.
    - If you submit a runnable program, partial marks can be given if some parts are correct.