COMP536 | Homework One: System Calls

Course: COMP536 Secure and Cloud Computing
Name: Hua-Yu Cheng (hc105)
Email: hc105@rice.edu
Date: 09/10/2025

Part 1: The Speed of System Calls vs. Library Calls

Task 1

The script is under the `/part1` folder named `program.sh`. It is a shell script that compiles the code, runs the program 1000 times, and records the pattern of `f1` and `f2` calls in a `run.txt` file.

To run the script:

```
chmod +x program.sh
./program.sh
```

Task 2

I ran the script 5 times and the results can be found in `/part1/patterns`. Across all 5 runs, I see that `f1` is always before `f2`, i.e., 1000 times an `f1` before an `f2` in a run. Hence, I always see the same pattern.

Task 3

Though I always see the same pattern, this behavior is not guaranteed to happen across runs.

The two threads perform different I/O operations:

`f1` uses `fprintf`, which is a high-level buffered library call which is more efficient as it reduces the number of expensive system calls required.

On the other hand, `f2` uses the `write` system call, which does not use a user-space buffer and each call crosses into the kernel immediately. This makes `write()` less efficient for many small writes, since each call is a system call.

Though `f1` thread is more efficient than `f2`, the order is not guaranteed. This is because the scheduler of the operating system is responsible for deciding which thread get to run on the CPU as well as the CPU time each thread gets. The scheduler schedules the thread based on the current state of each thread, the overall system workload, and the scheduling policy.

Hence, since the order depends on how the operating system schedules the threads, the order could change based on the operating system and workload.

Part 2: Understanding the Differences

Task 1

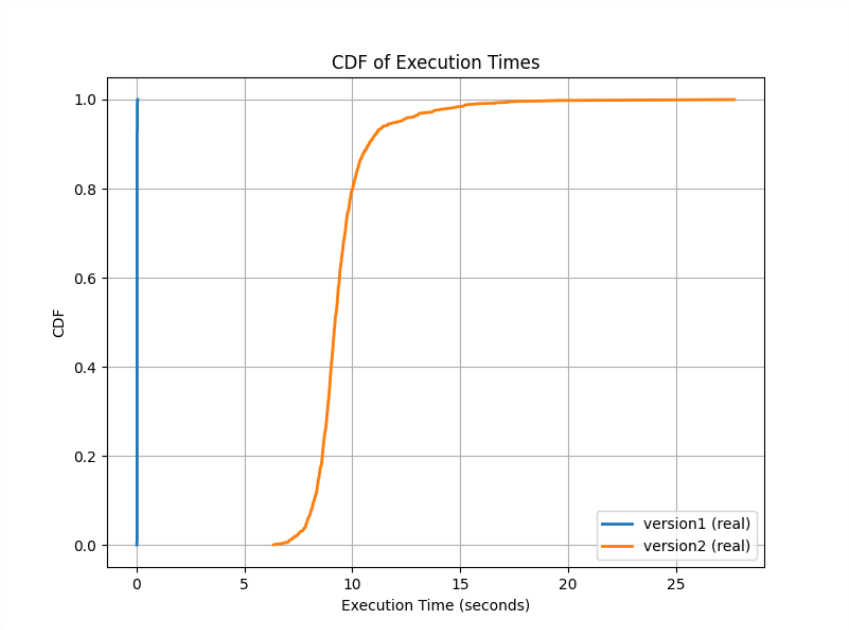
The script is under the `/part2` folder named `run.sh`. It is a shell script that compiles the code, runs `version1.c` and `version2.c` each for 1000 times, and records the time in `/part2/v1_times.txt` and `/part2/v2_times.txt` respectively.

To run the script:

```
chmod +x run.sh
./run.sh
```

Task 2

The Python script `/part2/plot_time.py` is used to parse the result and plot the CDF figure. Below is the CDF curves for two programs' execution times:



We can see that the CDF of version1's execution time rises almost vertically in 0 seconds. This means that all runs of version1 completed in a very **short** and **consistent** time (with mean 0.031s and median 0.030s).

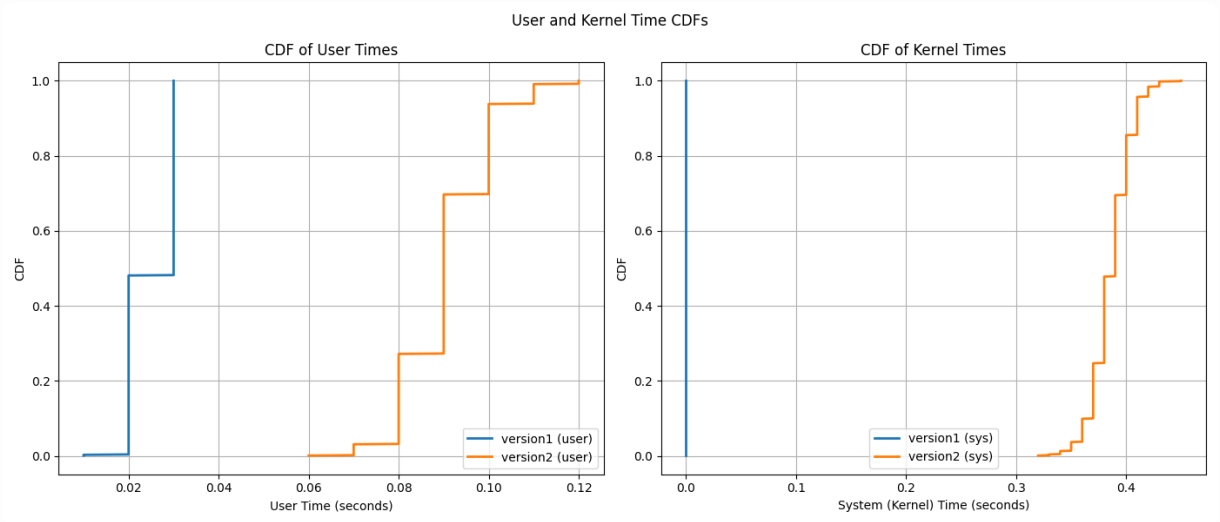
By contrast, the CDF of version2's execution time starts rising around 6 seconds and slowly approaches 1 by around 25 seconds. This means that version2 runs **slower** (with mean 9.49s and median 9.2s). There's also **high variability** between runs since the CDF has a gradual slope. This spread shows that execution time is not stable.

The difference is because the two programs, though both create a single worker thread and write a characters to a file, use different I/O operations:

- Version1 uses `fprintf`, a buffered `stdio` library call. Glibc collects bytes into an 4–8 KB user-space buffer and only calls the kernel occasionally (when the buffer is full or when we explicitly call `fclose`). Since it is buffered, `fprintf` return to the thread very quickly after writing to its internal buffer in memory, without waiting for the data to be written to the disk. As a result, version1 only has tens of kernel writes instead of hundreds of thousands, allowing it to execute relatively fast and stable.

- Version2 uses the `write` system call. The call is likely to **block and wait** for the operating system to complete the actual disk write operation. Version2's thread uses one syscall per byte, so there's 100,000 kernel calls in total, which is a much slower process. The extent of this blocking depends on the system load, disk cache, and timing of background flushers, which is why version2 not only runs slower but also shows much greater variability across runs.

Task 3



The CDF for version1's userspace time clustered around 0.02-0.03 seconds, and the kernel space time is effectively 0.00s (to two decimals) across runs because we have very few syscalls thanks to buffering. Since user ~0.02–0.03s, sys ~0.00s, real ~0.03s, version1's CPU time and real time match, which means it is compute-bound and efficient.

On the other hand, version2's userspace time clustered at around 0.09–0.12s, which is a bit higher due to copying arguments into the kernel for many writes. It's kernel space time is also higher, around 0.32–0.45s, because we have lots of syscalls. Additionally, user (~0.1s) + sys (~0.4s) ≈ 0.5s, yet the real time is ~9–25s. This mismatch shows version2 spends most of its real time waiting for `write()`, not actually using the CPU.

Task 4

The execution times are not the same across runs, and this effect is especially visible in version2. Its variability comes from issuing many small `write()` system calls without user-space buffering. Each call goes straight into the kernel's page cache, and the operating system decides later when to flush data to disk. These flushes depend on background load, cache state, and kernel policies, which are outside the program's control. As a result, version2 sometimes finishes quickly and sometimes stalls, leading to high variability across runs.

Below are some methods to make the execution times more stable across runs.

1. **Reduce I/O Operations**
   Execution time variability often comes from the operating system's handling of disk writes. The most effective method is to reduce the number of I/O operations, which is the key difference between version1 and version2. By using buffered I/O functions like `fprintf` or `fwrite`, the program writes data to an in-memory buffer, and only performs a costly system call to the kernel when the buffer is full or the file is closed. This reduces syscall overhead and the chance of being throttled by the kernel, hence makes the execution time more stable.

2. **Use `fdatasync()` or `fsync()`**
   Another method is to explicitly flush file data to disk using `fdatasync()` or `fsync()`. Normally the operating system delays writes by keeping data in memory and writing it out later in the background. This improves performance, but it also makes execution time less predictable because the flush can happen at different moments in different runs. By calling `fdatasync()` (flushes data only) or `fsync()` (flushes data and metadata) at controlled points, we can remove this uncertainty and make runs more consistent.

3. **Use RAM-backed filesystem**
   Another method to minimized I/O delay and variability is to use RAM-backed filesystem. On a typical disk, writes are cached in memory and flushed to the device later, sometimes unpredictably. This means two runs of the same program may hit the flush at different points and finish at different times. By using RAM-backed filesystems such as `tmpfs`, files are stored in memory, so writes are effectively just memory copies. Since there is no actual disk involved, there are no unpredictable flushes, making timing much more consistent.

4. **Pin the Process to a CPU Core**
   Another reason that the execution time varies is because of process scheduling. To avoid this, we can assign the process to a specific CPU core using `sched_setaffinity` or `taskset`. This prevents the scheduler from moving it between cores, which reduces cache and TLB misses caused by migration and avoids unnecessary context switches, which leads to more stable execution time.

5. **Use Memory-Mapped I/O**
   By using memory-mapped I/O (e.g., using `mmap`, a file is mapped directly into the program's virtual address space. This allows the program to access the file through normal memory operations, avoiding the overhead of many system calls. By reducing syscall overhead (which is the main reason of variability), memory-mapped I/O can make execution times faster and more stable.