Evan Schipellite
Robert Bethune
CSI 385 - 02
1 February 2015

PA1: Report

Simple Forking

The parent process begins by creating the desired number of child processes. Each child process, after being created, will then proceed to a function that allows them to loop until the global count reaches the maximum. Each iteration, the child process will increment the global count and output information regarding its PID and the current count. After completing, the children exit the program, and the parent will also exit after creating the children. This means that parent process may be complete before the children finish incrementing the global count.

However, there were some issues encountered when executing this algorithm as it currently is. First, we needed to create a static int pointer to the count, utilizing mmap to effectively create the global variable that would be shared between the processes. Otherwise, the count would be copied to each child process, but not updated and shared between them. This would simply output each child process incrementing their own count to the max. Instead, this alternative implementation allows for the fork processes to properly increment a global count.

The only subtle issue remaining relates to the speed of any given fork. Most of the time, a single fork only has its output displayed, simply because the process completes the incrementing procedure before another forked child process is created. This means that there is no way to determine when a fork will activate or how long it will last before swapping to another process. Sometimes it would swap between two or three, while other times it would utilize one child process.

Pipes

Utilizing the pipe method, it is possible to have each child process read and write to the pipe data, therefore allowing them to acquire, alter, and save the current count. In many ways, this has similar aspects to the previous forking process, as it will still often only utilize one or two of child processes at any given time. However, there is one important peculiarity that usually occurs.

While the processes are supposed to increment up to a maximum, most of the time the pipe method will cause the increment to go a bit beyond the maximum. This occurs because each process needs to read the pipe during the looping section of the program. Before reading from the pipe, it is only aware of the last known count, which would be less than the maximum. However, after reading it notes that the count is beyond, increments, and finishes the loop. So, the program will always go beyond the maximum, since each process will read the count, alter, and save before actually completing.

Shared Memory

As far as can currently be examined, the shared memory essentially provides an equivalent method of managing the program as utilized when mmap was applied for the first fork simulation. Shared memory allows for an int pointer to be allocated, with each child process access the same memory space to acquire the value of the count. Similar to the fork method, the program may either choose one child processes, or alternate between a few depending on how the system chooses to manage the processes. This method also requires additional allocation and address checks to ensure that the memory space is available, applying a unique key to distinguish the location.

Conclusion

Out of all the methods, the mmap and shared memory provided the most effective means for managing the job distribution for child processes. Overall, it is crucial to note that the operating system can swap between child processes in an unpredictable manner. This means that it is important to ensure that the child processes are created and conducted in such a way that allows for them to run in any order. This is why the shared memory implementation works the best, since the child processes can access the current count at any time, making it a reliable method for managing constantly altering data.