CPSC 457 – Fall 2025
Assignment 1
Processes in Linux
Due: September 19th @ 11:59 PM

Objective: To work with processes in Linux.

Outcomes: In this assignment, you will write parallel C programs that create Linux processes. You will also get exposure to Inter-Process Communication (IPC) through the wait system call and through shared memory.

Programming Language: Use C for this assignment.

Note: You must ensure that your code runs on the University Linux servers, you can remote ssh to the cslinux.ucalgary.ca load balancer. Code that does not run or compile on the university servers will not receive any credit. You cannot run system calls like fork() on an operating system that is not UNIX-like.

WARNING: You must make sure that all created processes exit (with the exit() system call). Failure to do so may result in overloading the servers and may bring them down. System admin can determine whose user is responsible for such behaviour.

I – Basic Parallel Program with Fork
Programming: Write a program that forks processes to do a treasure hunt in a matrix. The input to this program is a 100x1000 matrix. Each cell in the matrix is either a 0 (empty) or 1 (treasure). There is one treasure in the matrix. Your program will fork 100 processes where each process searches a row of 1000 cells for the treasure. The parent waits for the child processes to finish the search. Child processes communicate with the parent through the exit() and wait() system calls. The child that finds the treasure sends a success signal to the parent. The parent should determine the exact row and column where the treasure is located. Use an appropriate integer value for a failed search.
Note: You are not allowed to use shared memory for this part.

Sample Output
```
Child 0 (PID 2338): Searching row 0
…
Child 99 (PID 2438): Searching row 99
Parent: The treasure was found by child with PID 2411 at row 73 and column 652
```

Reflection: Write a paragraph discussing the most challenging aspects to you of this program and how you tackled them.

II – Parallel Prime Number Finder
Programming: Write a program that forks N children to work in parallel on finding prime numbers in a range of values. The ranges assigned to child processes are mutually exclusive. The input is collected through the command line:
```
LOWER_BOUND UPPER_BOUND N
```
Where the lower and upper bounds are simply the range where the search for prime numbers is taking place and N is the total number of processes.

The parent process:

- Allocates shared memory. (Use the system calls shmget(), shmat(), shmdt(), and shmctl())
- Spawns N child processes using fork().
- Waits for all children using wait().
- After all children complete, reads results from shared memory.
- Frees shared memory and prints all collected prime numbers.

Each child process:

- Computes a non-overlapping subrange of the total range.
- Finds prime numbers in that subrange.
- Stores them in the shared memory in a thread-safe manner (e.g., using an offset scheme).
- Exits.

Note: If the number of processes (N) is greater than the number of values in the input range, make sure each child process is assigned at least one value to check.

Use shmget() and shmat() to create and attach a shared memory segment:

```
int shmid = shmget(IPC_PRIVATE, SIZE, IPC_CREAT | 0666);
int *shm_ptr = (int *) shmat(shmid, NULL, 0);
```

Memory Layout:

- Use a shared array of integers large enough to hold all found primes.
- Use a shared index counter (e.g., the first integer in shared memory) to store how many primes have been stored so far. Each child process:
    - Finds primes.
    - Uses this counter to know where to write its primes.

Note: In general, we need to use locking to avoid race conditions. However, this will be covered in the second half of the course. For simplicity, this assignment will avoid race conditions by avoiding overlapping memory access by assigning each child a separate offset as follows.

Suggested Memory Layout

Instead of concurrent writes to the same buffer, each child writes to its **own block** in shared memory:

- Total array size = N * MAX_PRIMES_PER_CHILD
- Child i writes to indices [i * MAX_PRIMES_PER_CHILD ... (i+1)*MAX_PRIMES_PER_CHILD - 1]
- Parent reads all segments

This avoids the need for locking and is appropriate for this assignment's level of complexity.

Primality testing

Use the following function to test if a number is prime:

```
int is_prime(int num) {
    if (num < 2) return 0;
    for (int i = 2; i <= sqrt(num); i++) {
        if (num % i == 0) return 0;
    }
    return 1;
}
```

Sample Output (for input `100 249 3`):

```
Child PID 2345 checking range [100, 149]
Child PID 2346 checking range [150, 199]
Child PID 2347 checking range [200, 249]

Parent: All children finished. Primes found:
101 103 107 109 113 127 131 137 139 149 ...
```

Reflection: Write a paragraph discussing:

- How you made sure the program does not create more than the required number of processes.
- How the work was divided among child processes.
- How you made access to shared memory by child processes safe.

Deliverables

Submit a GIT link to your source code and upload your source code to d2l (this is only required for archival purposes.) **Your TA will check the version history on GIT.** Remember that assignments with no version history on GIT will not receive any credit.

Upload a PDF report to D2L that contains your reflection.

Input Commands

You will compile and run your code in the specified format for part 1:

```
gcc -O2 -Wall a1p1.c -o a1p1
./a1p1 < inputfilep1.txt
```

And the following for part 2:

```
gcc -O2 -Wall a1p2.c -o a1p2 -lm
./a1p2 <LOWER> <UPPER> <NPROCS>
```

Output Format

Your program should produce the output format specified in each part above.

Marking Scheme

A spreadsheet detailing the marking scheme will be posted on D2L.

Assignment Policies

For more information, such as teams and late submission policies, refer to the document titled **Assignment Policies** on D2L.

Happy Coding :)
CPSC 457 Team