PART II - CONCURRENT PROCESSES IN UNIX

Repository Link: https://github.com/akshavib/SYSC4001-A2-P2.git

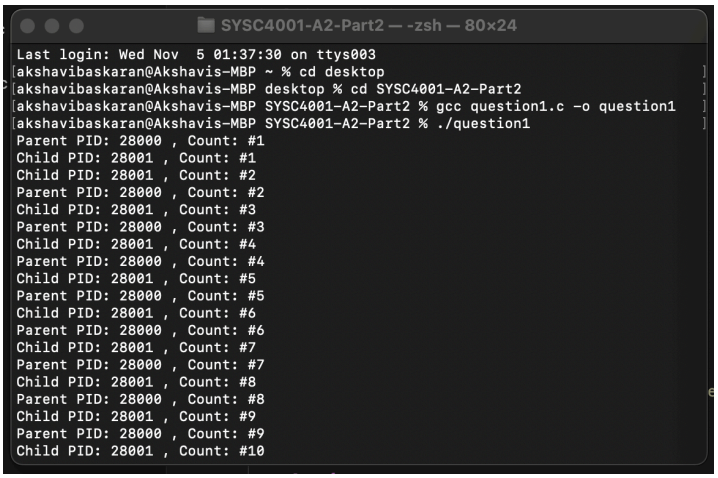
1. Run two concurrent processes in C/C++ under Linux

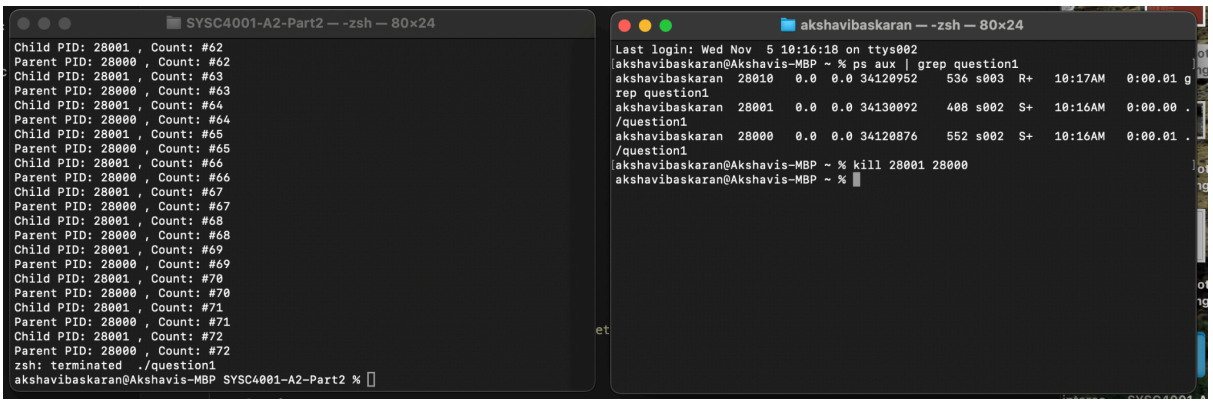
Using the fork system call (page 472 of the Linux book), create two independent processes, which run indefinitely. Process 1 will run forever, will initialize a counter at 0, and will increment it in each cycle of an infinite loop. Process 2 will run forever, will initialize a counter at 0, and will increment it in each cycle of an infinite loop. Use delay functions to slow the display speed.

To finish the program, use the kill command (man pages), find the PID of both processes (ps) and kill them.

Two concurrent processes were successfully created and executed in C *(question1.c)* using the **`fork()`** system call under Linux. Process IDs (PIDs) for both the parent and child were identified using the ps command, and the processes were safely terminated using the kill command. The figures below show the execution and termination of both processes. The parent count begins very large because when the process forks, the child copies the parent's memory. This leads to both processes sharing the same print buffer which causes their outputs to overlap and mix together, making it look like the parent's count starts from a much larger number than it really does.

The **`fork()`** system call is used in Linux systems to create a new process. The new process is considered as the child process, while the one that invoked **`fork()`** is the parent process. When **`fork()`** is called by the parent process, the resulting child process receives its own Process ID, a copy of the parent's code, data, stack, and a separate memory space. The parent process will resume its execution, while the child process will start executing from the same point. The **`fork()`** function returns 0 to the child and provides the child's PID to the parent. This is the fundamental process that enables UNIX systems to launch new programs and support multitasking





2. Extend the Processes. Now, process 1 will display only multiples of 3. That is, increment the counter in an infinite loop, calculate if it's a multiple of 3, and display that number. Also, display the number of cycles in each iteration. Example:

Cycle number: 0 – 0 is a multiple of 3
Cycle number: 1
Cycle number: 2
Cycle number: 3 – 3 is a multiple of 3
...

Process 2 will do the same but decrementing the value of the counter.

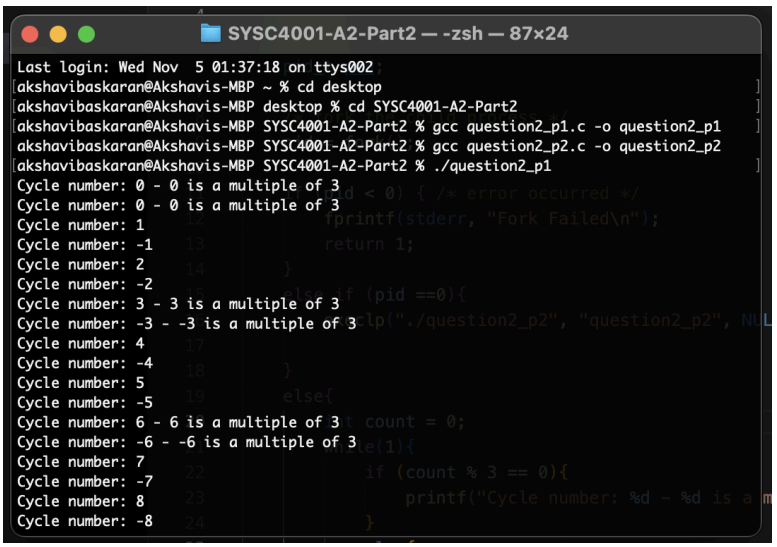
Use the exec system call to launch Process 2 (i.e., Process 2 should be a different program/executable).

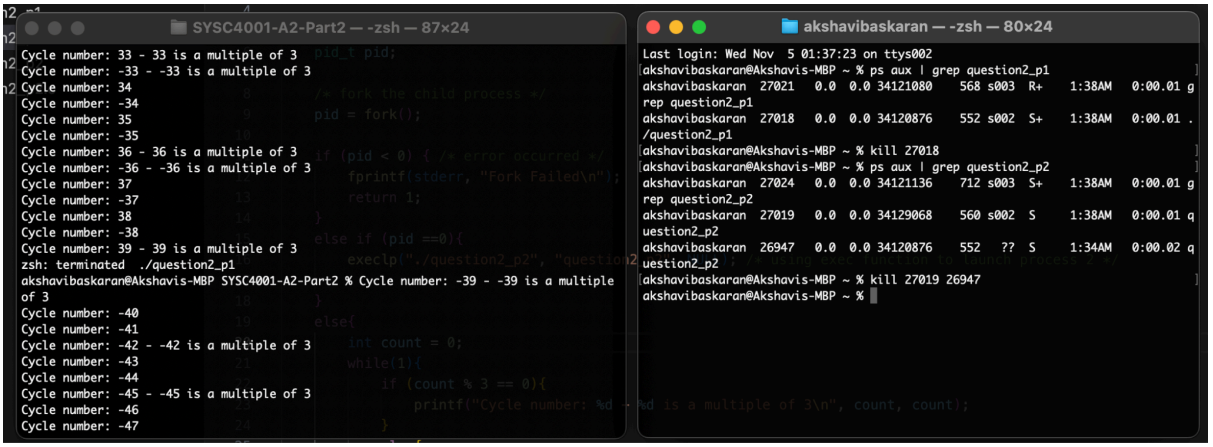
Use delay functions to slower the display speed.

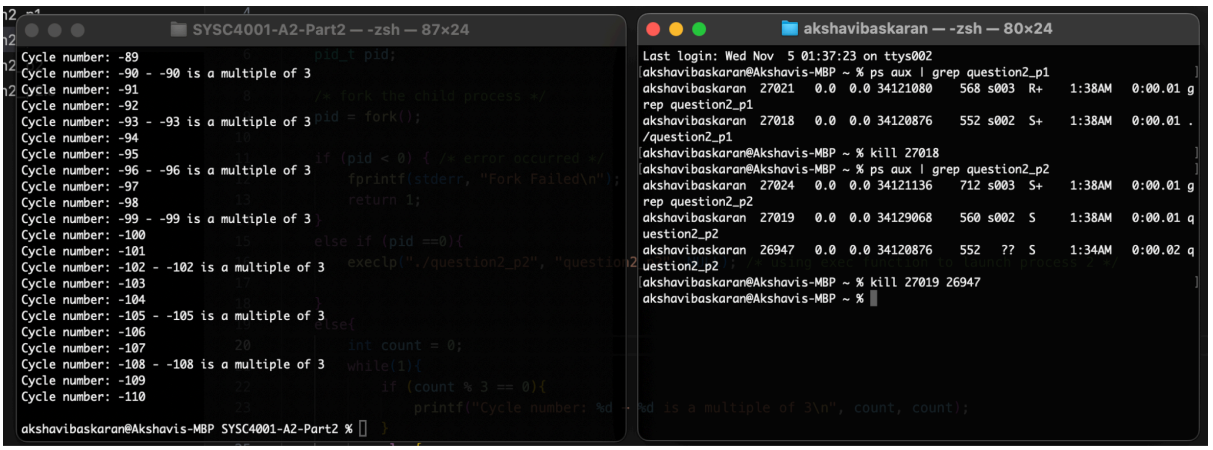
To finish the program execution, use the kill command (man pages), find the pid of both processes (ps) and kill them.

Both processes were successfully extended. The first program, *question2_p1.c*, implements an incrementing counter that displays multiples of three and uses the exec system call to launch the second program. The second program, *question2_p2.c*, executes as a separate process and prints multiples of three while decrementing its counter. As expected, when the parent process was terminated, the child process continued running independently, as shown in the second screenshot. The child process was then safely terminated using the appropriate commands.

The **`exec()`** system call function is used to replace and load a process' memory space with a new program and execute it without creating a new one. In general, it's called after **`fork()`** to overwrite the memory image that was copied over from the parent process to the child one.

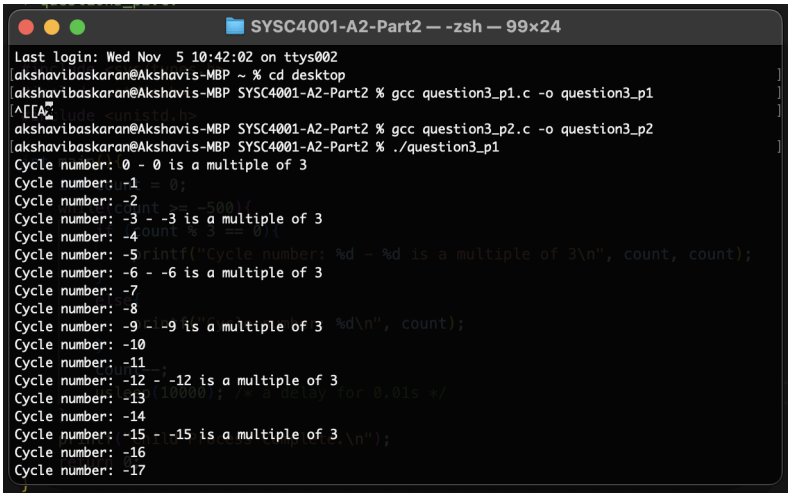


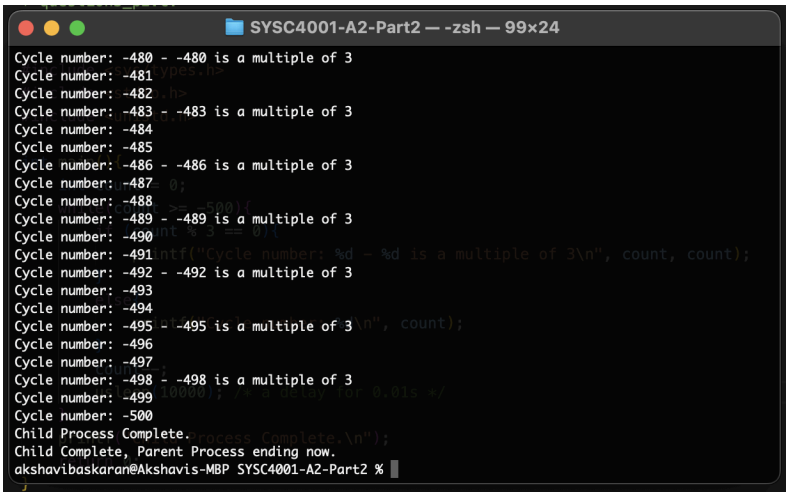




3. Extend the processes above once more. Use the wait system call. Process 1 starts as in 2, and when Process 2 starts, it waits for it. Process 2 runs until it reaches a value lower than -500. When this happens, Process 1 should end too.

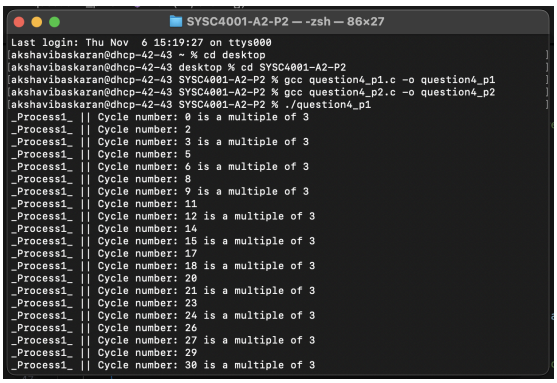
The wait() system call was implemented in Process 1 to ensure it pauses execution until Process 2 completes its 500 iterations. Once Process 2 reached the specified termination condition (counter value below -500), Process 1 resumed and both processes were successfully terminated.

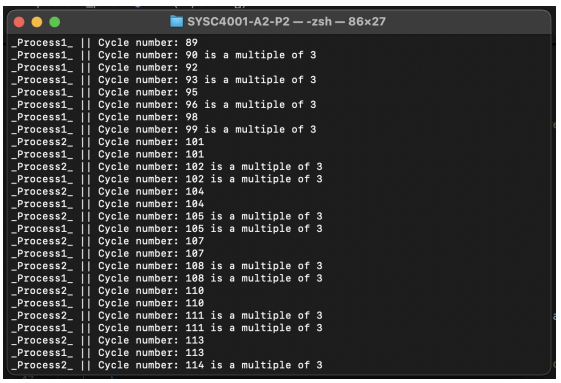


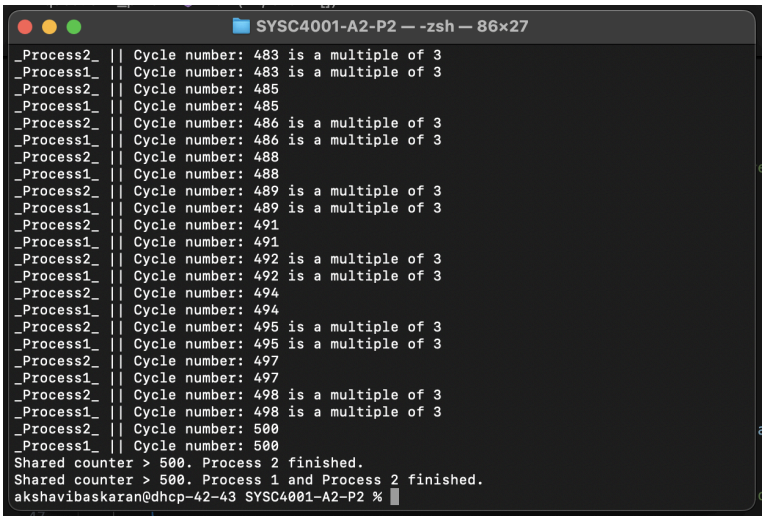


4. Extend the processes above once more. They should now share memory. The primary functions are listed in the book and course materials. Using shmget, shmctl, shmat, and shmdt, add two common variables shared between the two processes. The first variable contains the value of the multiple (in the example above: 3; that number can be changed and then the program should adapt and display the multiples of the chosen number). The second variable contains the value of the counter used by Process 1, which is now shared with Process 2. Process 2 starts only when the value of this variable is larger than 100. Each of the processes should now react to the value of the shared variable and display a message identifying themselves and numbers in shared memory. Both processes finish when the value of the shared variable is larger than 500.

All required functionality has been successfully implemented. The two processes now share memory through the use of shmget, shmat, shmdt, and shmctl. Process 1 updates the counter stored in shared memory, while Process 2 monitors this value and begins execution only after the counter exceeds 100.







5. Extend the processes above once more. They should now protect concurrent access to the shared memory positions. On top of the shm instructions, you should protect the shared memory access using semaphores. Use semget, semop, semctl to protect the shared memory section.

The implementation has been successfully extended to include protection of concurrent access to the shared memory. The semaphores ensure that only one process accesses or modifies the shared variables at a time, preventing race conditions and ensuring data consistency. The program functions correctly, with both processes synchronizing properly through the use of semaphores.