SYSC 4001A L3
Assignment 2 Part III
Report

Caroline Twelves: 101301400
Salma Khai: 101301357

In this assignment there were two fundamental system calls implemented; fork and exec. When each trace file is being run, it is important to know the function of these calls.

The fork system call is a system call, in which the current process creates a copy of itself, referred to as a child process. It does this by firstly going into kernel mode and saving the context of the CPU, before the call was implemented. It then creates a new PCB for the child process, and copies the parents address space and execution context into it. The difference between these two processes is nearly identical, differing only in their Process identifier numbers, with the child also holding the parent process identifier number in its PPID field. When forking a process, it is vital to be conscious to avoid making the child a zombie, which occurs when the child terminates itself, but the parent does not receive its exit status. This can be avoided by having the parent process use the system call wait(), which will stop the parent from executing until the child terminates, or it terminates the child. It then has to call IRET to return to user mode.

The exec family of functions replaces the current process image with a new program. Unlike fork(), which will ensure the creation of a new process, exec keeps the same process ID but discards the old program's code, data, and stack, loading the new executable into memory. This means that the current program is completely replaced by the new one. If an exec succeeds, then it never returns. Any open file descriptors, environment variables, and process attributes (like working directory and signal masks) are preserved unless flagged to close on exec. This is why in this assignment at the end of the exec code, there is a break statement. This allows the exec to stop executing the trace file. In practice, exec is often used after fork(): the parent continues the running of its own code,while the child calls on exec to run a different program. Variants of this call such asexeclp() or execvp() search the system's path for the executable while execve() directly loads the specified file.

In this assignment,outlined more below, the execution file and system status files are updated after each execution of both exec and fork.

In the first test case given the first execution is fork which takes 10 milliseconds to execute. The first thing that must happen is to switch to kernel mode. This allows us to manipulate the hardware of the computer, which includes the main memory. We also need to save the context, which is the previous CPU state. Fork is called here by the initial PCB. What fork does each time it's called is create a copy of itself, referred to as a child process. It does this by creating a copy of itself, a new PCB, and assigning the PPID of this child as the PID of the parent, to classify the child as a child to this parent. The child then must be allocated memory. This is done through partitions. In this assignment, there are 6 partitions, each with a different amount of memory. The child is given the first one that it fits into based on its size. The parent must then 'wait' for its child to terminate so as to not create a zombie process. The implementation in this assignment is to push the parent to the back of the queue, as to allow the child to run, as well as loading the child's trace code into the program. IRET is then called, to switch back to user mode.

The child is now the process executing the lines of code in a recursive fashion, the next one in the first test case being EXEC program1, 50. The first thing that must happen again is switching to kernel mode and saving context. This line indicates that the child must run the external file named program1. The length of this file is pulled from the external_files.txt file, and tracked. This is how long it will take to load the program into the memory, as the complete size of the program must be uploaded. The process is loaded into one of the memory partitions and allocated this memory. The PCB is also updated to reflect this, and that the child is currently running a process. The program 1 is then run in its entirety, with its trace file being uploaded and run through recursively. After its completion, IRET must happen to switch back to user mode, the scheduler must then be called again to determine which process will run next, and its outputs and times recorded. The memory will also be deallocated. The exec file must also break after the program is finished recursively, as to not have it continue running the program again.

The child then executes code in a recursive fashion, line by line until it reaches the IF_PARENT line (the next one in this case), caught by a loop called each time the child is recursively called. Once this line is met, the output files the child created and added to, as well as the time the child took is appended to the main variables holding this data and the memory of the child is deallocated, effectively terminating the process. Once this happens, the parent process will run its execution from where it left off.

In the first test case, the parent then executes the second program. It is loaded in the exact same method that the first program was, the only difference is that it is now the parent executing it.

Endif, ends the parent process, or child if the child were to be running.

The second test case runs exactly the same as the first, except the second program is called within the first. This execution is largely the same, except the child runs both programs and the first program does not terminate until it sees the second terminate.

The other cases run in the same way, forking when a child process runs, having the parent wait until the child is finished execution and any time an exec call is run to run that in the same way.