SYSC4001 - Assignment 2 Part III Report

Sohaila Haroun - 101297624
Zaineb Ben Hmida - 101302936

https://github.com/sohaila-cu/SYSC4001_A2_P3.git

Things to discuss in the report:

- The examples from the assignment
    - As can be seen, our first 3 simulations are the same as the 3 examples from the assignment and the outputs are identical.
- The examples I made
    - We added more programs and played around with their sizes (more explanations on results below)
    - Some of the examples I made included doubly nested EXEC calls where init forked and exec a program that also forked and execed, twice, (simulation 5&6, program6). The output, especially the system status, shows this being done very clearly.
- The example with not enough memory with and without error handling
    - In our simulations 5.1 and 5.2, we created a program6, that is very large and could either never fit in any partition, or it could fit once in a partition but if more than one process EXEC program6, there will be a problem with the memory allocation. In 5.1, we did not add error handling and you can see that the PCB shows that program6 is in partition -1, which does not exist. If this was not a simulation this would cause many issues, possibly segment faults, where the process is trying to access and/or overwrite memory that does not exist, or it does not have access to. We decided to add error handling in simulation 5.2 and when a memory allocation error occurs, an error is displayed in the terminal, as well as in the execution file and the EXEC gets cancelled and that program does not get loaded into the process, as seen in the PCB.
- Why there is a break at end of exec
    - break; //Why is this important? (answer in report)
        - We need a break since the current trace was replaced or completed by whatever was in exec, because it is implemented with recursion. So you do not need the loop to continue once you are done all of the execution steps in exec.

- In example/simulation 2, in program1 (in my simulation it is called program 3), after the ENDIF, there is EXEC program2, 33 //which process executes this? Why?
    - Both the child and the parent execute this, this is because when FORK was called, both the child and the parent have the same program. The child will then execute every line in the trace file under 'IF_CHILD, 0' until IF_PARENT, 0' or 'ENDIF, 0' is encountered and the parent will execute every line under 'IF_PARENT, 0' until an 'ENDIF, 0' is encountered. In this program, the conditions were empty, so as can be seen in the execution file, nothing happens, but every line after 'ENDIF, 0' should be executed by both the parent and the child. In this

case, the exec will be run by both the parent and child, as can be seen by the system status and execution file.

- It is noteworthy to mention, that if a tracefile forked, and the if_child came before the if_parent, the child will run whatever is in if_child and skips if_parent first, because it has priority as noted in the below assumption.

  Assumptions made:

  • The child process has a higher priority than the parent process, and no preemption.

  Therefore, the child process should be executed until completion before resuming the parent process. This means that the simulator does not account for orphaned processes yet; the child must finish execution before the parent process can terminate. HINT: As soon as you hit a FORK, start executing the child. When you hit an IF_PARENT, skip all the lines of the trace file till you get to ENDIF. Once you get to ENDIF, execute till the end of the trace file. This marks the end of the child process; go back to the IF_PARENT line, execute the rest as parent. Do NOT worry about scheduling.
- //rest of the trace after ENDIF doesn't really matter (why?)
  - Because ENDIF marks the end of the mutually-exclusive child/parent blocks, the simulator's fork/if semantics mean the interesting difference between child and parent happens inside the IF_CHILD / IF_PARENT region. After ENDIF both flows continue from the same place, so for the purpose of distinguishing what each process does immediately after the fork the contents after ENDIF don't affect which branch did what (they'll both execute the same remaining tr
-