Memory: For the purpose of our simulator, we have 6 fixed partitions for memory (40, 25, 15, 10, 8, 2) this memory stores the programs being run.

PCB: To be able to properly simulate we require a PCB. The PCB for our simulator is very basic and contains PID, PPID (parent PID), program name, partition number, program size, and the current state.

External files: This contains all the programs and there size (MB) that are external to our INIT.

Program: Programs are run and contain the following commands

FORK: command triggers a system call that clones the current PCB to create a child PCB.

EXEC name: command triggers a system call that copies a program from memory (found in external files) into the child. This takes time so we display it through the program size * 15.

IF_CHILD, IF_PARENT, ENDIF: IF_CHILD and IF_PARENT are used to determine if the child or parent should be executing. ENDIF displays that both parent and child should be executing.

Outputs: Contains all the output of the program being run.

Execution: All the execution steps are logged into the execution file.

System Status: All the system status updated like PCB changes that display current running and waiting tasks are logged in the system_status file.

Example 1:

Explanation:

FORK, 10

- init creates child process

IF_CHILD, 0

- beginning of child only code

EXEC program1, 50

- Child becomes program 1 and executes
    - Free old partition, new partition allocated with program 1 and pid of child
- CPU, 100 runs

IF_PARENT, 0

- Parent process is running from now on and child is done executing

EXEC program2, 25

- Program 2 is stored in PID 0
- Program 2 runs
    - SYSCALL, 4

ENDIF, 0

- Parent and child would now execute
- The rest of the program does not matter as both parent and child called EXEC the init that was running is now gone. The program never returns to the trace

Example 2:

Explanation:

FORK, 17

- Init creates child process

IF_CHILD, 0

- Beginning of child only running

EXEC program1, 16

- Child replaces image with program 1
- Program 1 executes
    - FORK, 15
        - Creates a child
    - IF_CHILD, 0
        - Child runs from here
    - IF_PARENT, 0
        - Child is done running, parent now runs
    - ENDIF, 0
        - Child and parent run the rest
    - EXEC program2, 33
        - Both parent (child of init) and child (grand child of init) run exec and replace image with program2
            - Both run this as ENDIF occurred before this line
        - CPU, 53
            - Runs by both of them

ENDIF, 0

- Parent and child will run the rest
- Only initial parent left to run

CPU, 205

- Run by init

Example 3:

Explanation:

FORK, 20

- init forks and creates a new child process (PID 1)
- The child now starts to execute.

IF_CHILD, 0

- Child runs the current code until a IF_PARENT or END_IF is found

IF_PARENT, 0

- Child is done executing now.
- Parent will execute until END_IF

EXEC program1, 60

- Parrent runs EXEC and copies program1 data into its PCB
- Parent process executes program1
- This can be seen in the output as the only program left over has the PID of the parent and its program name is program1

```
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   0 |    program1 |               4 |   10 | running |
+-------------------------------------------------------+
```

ENDIF, 0

- Both parent and child will run the code below

CPU, 10

- CPU burst runs, child is already done though.
- Parent is the only process to run CPU burst

Why is break important?

The break statement is important because the exec() replaces the current process entirely with the contents of the new program. So the compiler now needs to focus on servicing the new program. If there was no break statement, i would increment and the compiler would continue to read the next few instructions in the same program, which, after replacing the old program, is an irrelevant operation

Take the process in init.txt file for example. When exec() is run, the current program is replaced with the new one - referring to the contents of program1.txt. That means the compiler should now loop through the contents of the program1.txt, which requires the reading loop occurring in the current trace file (init.txt) to be broken first. If the break was not placed there, the compiler would ignore the contents of the newly replaced program1 and continue with process the next commands in init.txt (i.e. ENDIF, 0, and CPU, 10), which is not needed. Essentially, that break statement acts as a redirection of instructions to the new program, allowing the compiler to process the new set of instructions in the newly replaced program.