SYSC4001 Assignment 2:

Process Scheduling, Memory Management: Part I, II & III

Group 8

Antoine Hickey,

ID: 101295764

Enzo Chen,

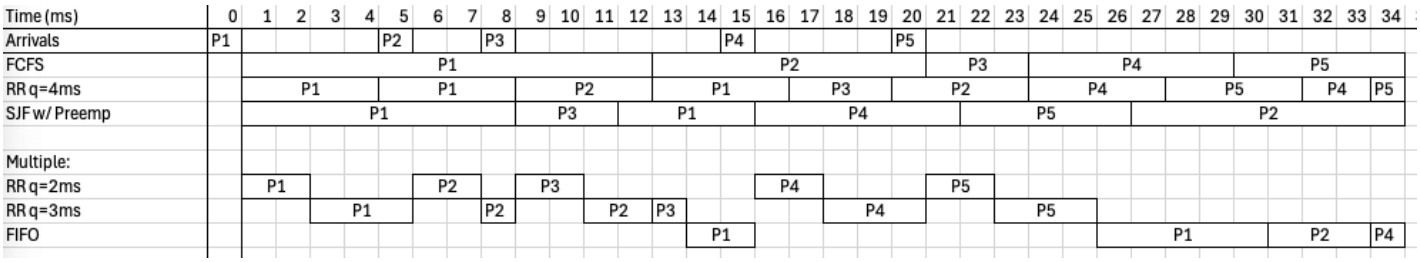
ID: 101306299

Lecture: SYSC4001A

Lab: SYSC4001L2

Part I – Concepts

a. Single CPU burst, no I/O



		FCFS		RR		STR		MULT	
Process	AT	CT	TT	CT	TT	CT	TT	CT	TT
P1	0	12	12	16	16	15	15	30	30
P2	5	20	15	23	18	34	29	33	28
P3	8	23	15	19	11	11	3	13	5
P4	15	29	14	33	18	21	6	34	19
P5	20	34	14	34	14	26	6	25	5
Mean			14		15		12		17

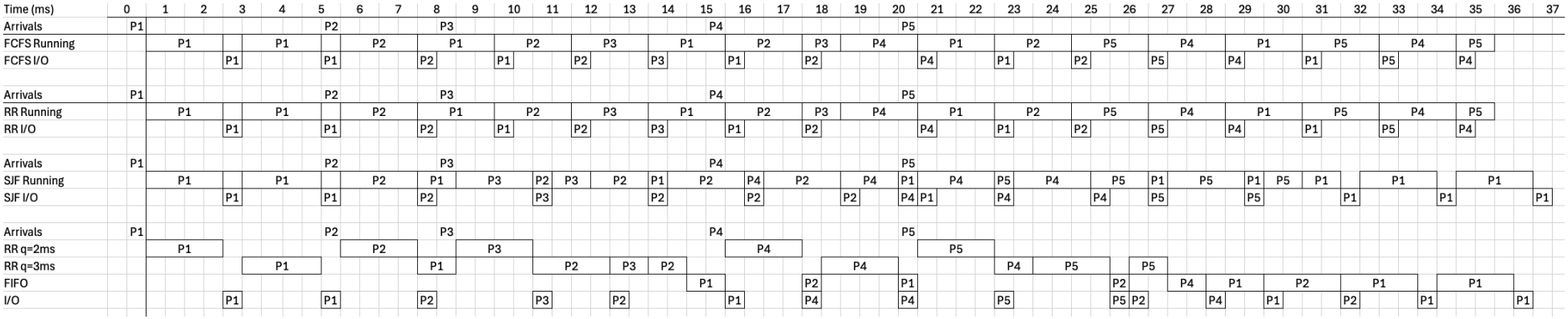
Let AT be the arrival time, CT be the completion time, and TT be the turnaround time of each process.

Turnaround time is calculated as CT – AT.

FCFS (First Come First Serve) or FIFO (First In, First Out), is one of the simplest scheduling algorithms which will accord a higher priority to the CPU bursts that arrive first. This algorithm is therefore non-preemptive: The process must give up control voluntarily (i.e. going into wait for I/O or System Calls). Once the event has finished, it will interrupt the CPU, adding the process to the back of the ready queue (from wait). Similarly, if a new process arrives while another process is running, the new process will be added to the back of the ready queue. FCFS is also considered a “fair” algorithm since it won’t prioritize a process with a shorter burst time. This also means that starvation is not a problem: With some algorithms, it is possible that longer processes never execute since they repeatedly get kicked to the back of the queue (for having low priority due to long process time as an example).As a result, aging is often introduced (where the more time a process spends in the ready queue, the higher priority it becomes). This, however, is not necessary for FCFS since there is no starvation to begin with.

Round Robin, or RR scheduling, is a preemptive algorithm that alternates between process, giving each one a quantum of CPU processing at a time. RR uses the timer to interrupt processes at the end of their allocated quanta. The algorithm works by cycling through the ready queue, running each process for a quantum, and then putting it at the back of the ready queue. New processes are also added to the back of the ready queue as they arrive. RR tends to have a long wait time, as processes have to keep waiting for their turn to receive more quanta instead of finishing processing. This algorithm is similar to first come first serve, or FCFS, since it is fair, and it assigns quanta in the order of arrival of the processes. It is identical to FCFS if the quanta is longer than every CPU burst. If the quanta is too short, the overhead caused by context switches becomes high, which prevents the CPU from doing useful work.

b. Each process requests I/O every 2ms, for a duration of 0.5ms



		FCFS		RR		SJF		MULT	
Process	AT	CT	TT	CT	TT	CT	TT	CT	TT
P1	0.0	30.5	30.5	30.5	30.5	36.5	36.5	36.0	36.0
P2	5.0	24.5	19.5	24.5	19.5	18.5	13.5	31.5	26.5
P3	8.0	18.0	10.0	18.0	10.0	11.5	3.5	13.0	5.0
P4	15.0	34.5	19.5	34.5	19.5	25.0	10.0	28.0	13.0
P5	20.0	35.0	15.0	35.0	15.0	30.0	10.0	26.5	6.5
Mean			18.9		18.9		14.7		17.4

Assume that every time a process returns to the ready queue from I/O and a new process appears at the same time, the new process enters the ready queue before the one returning from I/O.

c. Memory Management

Position	Hole Size (KB)
1	85
2	340
3	28
4	195
5	55
6	160
7	75
8	280

Job	Memory (KB)
J1	140
J2	82
J3	275
J4	65
J5	190

First Fit Allocation Table	
Job	Partition
J1	2
J2	1
J3	8
J4	2
J5	4
Remaining free	466
Total internal fra	0
Total external fra	466
Utilization perce	62%

Best Fit Allocation Table	
Job	Partition
J1	6
J2	1
J3	8
J4	7
J5	4
Remaining free	466
Total internal fra	0
Total external fra	466
Utilization perce	62%

Worst Fit Allocation Table	
Job	Partition
J1	2
J2	8
J3	None
J4	2
J5	8
Remaining free	741
Total internal fra	0
Total external fra	741
Utilization perce	39%

First Fit	
J1 -> P2	340-140=200
Position	Hole Size (KB)
1	85
2	200
3	28
4	195
5	55
6	160
7	75
8	280
J2 -> P1	85-82=3
Position	Hole Size (KB)
1	3
2	200
3	28
4	195
5	55
6	160
7	75
8	280
J3 -> P8	280-275=5
Position	Hole Size (KB)
1	3
2	200
3	28
4	195
5	55
6	160
7	75
8	5
J4 -> P2	200-65=135
Position	Hole Size (KB)
1	3
2	135
3	28
4	195
5	55
6	160
7	75
8	5
J5 -> P4	195-190=5
Position	Hole Size (KB)
1	3
2	135
3	28
4	5
5	55
6	160
7	75
8	5

Best Fit	
J1 -> P6	
Position	Hole Size (KB)
1	85
2	340
3	28
4	195
5	55
6	20
7	75
8	280
J2 -> P1	85-82=3
Position	Hole Size (KB)
1	3
2	340
3	28
4	195
5	55
6	20
7	75
8	280
J3 -> P8	280-275=5
Position	Hole Size (KB)
1	3
2	340
3	28
4	195
5	55
6	20
7	75
8	5
J4 -> P7	75-65=10
Position	Hole Size (KB)
1	3
2	340
3	28
4	195
5	55
6	20
7	10
8	5
J5 -> P4	195-190=5
Position	Hole Size (KB)
1	3
2	340
3	28
4	5
5	55
6	20
7	10
8	5

Worst Fit	
J1->P2	340-140=200
Position	Hole Size (KB)
1	85
2	200
3	28
4	195
5	55
6	160
7	75
8	280
J2->P8	280-82=198
Position	Hole Size (KB)
1	85
2	200
3	28
4	195
5	55
6	160
7	75
8	198
J3->None	
Position	Hole Size (KB)
1	85
2	200
3	28
4	195
5	55
6	160
7	75
8	198
J4->P2	200-65=135
Position	Hole Size (KB)
1	85
2	135
3	28
4	195
5	55
6	160
7	75
8	198
J5->P8	198-190=8
Position	Hole Size (KB)
1	85
2	135
3	28
4	195
5	55
6	160
7	75
8	8

Scenario 1: Multiple processes are allowed per partition

Position	Hole Size (KB)
1	85
2	340
3	28
4	195
5	55
6	160
7	75
8	280

Job	Memory (KB)
J1	140
J2	82
J3	275
J4	65
J5	190

First Fit Allocation Table	
Job	Partition
J1	2
J2	1
J3	8
J4	4
J5	None
Remaining free	656
Total internal fra	338
Total external fra	318
Utilization perce	46%

Best Fit Allocation Table	
Job	Partition
J1	6
J2	1
J3	8
J4	7
J5	4
Remaining free	466
Total internal fra	43
Total external fra	423
Utilization perce	62%

Worst Fit Allocation Table	
Job	Partition
J1	2
J2	8
J3	None
J4	4
J5	None
Remaining free	931
Total internal fra	528
Total external fra	403
Utilization perce	24%

First Fit		
J1 -> P2	340-140=200	
Position	Hole Size (KB)	Status
1	85	Empty
2	200	Filled
3	28	Empty
4	195	Empty
5	55	Empty
6	160	Empty
7	75	Empty
8	280	Empty
J2 -> P1	85-82=3	
Position	Hole Size (KB)	Status
1	3	Filled
2	200	Filled
3	28	Empty
4	195	Empty
5	55	Empty
6	160	Empty
7	75	Empty
8	280	Empty
J3 -> P8	280-275=5	
Position	Hole Size (KB)	Status
1	3	Filled
2	200	Filled
3	28	Empty
4	195	Empty
5	55	Empty
6	160	Empty
7	75	Empty
8	5	Filled
J4 -> P4	195-65=130	
Position	Hole Size (KB)	Status
1	3	Filled
2	200	Filled
3	28	Empty
4	130	Filled
5	55	Empty
6	160	Empty
7	75	Empty
8	5	Filled
J5 -> None		
Position	Hole Size (KB)	Status
1	3	Filled
2	200	Filled
3	28	Empty
4	130	Filled
5	55	Empty
6	160	Empty
7	75	Empty
8	5	Filled

Best Fit		
J1 -> P6		
Position	Hole Size (KB)	
1	85	Empty
2	340	Empty
3	28	Empty
4	195	Empty
5	55	Empty
6	20	Filled
7	75	Empty
8	280	Empty
J2 -> P1	85-82=3	
Position	Hole Size (KB)	
1	3	Filled
2	340	Empty
3	28	Empty
4	195	Empty
5	55	Empty
6	20	Filled
7	75	Empty
8	280	Empty
J3 -> P8	280-275=5	
Position	Hole Size (KB)	
1	3	Filled
2	340	Empty
3	28	Empty
4	195	Empty
5	55	Empty
6	20	Filled
7	75	Empty
8	5	Filled
J4 -> P7	75-65=10	
Position	Hole Size (KB)	
1	3	Filled
2	340	Empty
3	28	Empty
4	195	Empty
5	55	Empty
6	20	Filled
7	10	Filled
8	5	Filled
J5 -> P4	195-190=5	
Position	Hole Size (KB)	
1	3	Filled
2	340	Empty
3	28	Empty
4	5	Filled
5	55	Empty
6	20	Filled
7	10	Filled
8	5	Filled

Worst Fit		
J1->P2	340-140=200	
Position	Hole Size (KB)	
1	85	Empty
2	200	Filled
3	28	Empty
4	195	Empty
5	55	Empty
6	160	Empty
7	75	Empty
8	280	Empty
J2->P8	280-82=198	
Position	Hole Size (KB)	
1	85	Empty
2	200	Filled
3	28	Empty
4	195	Empty
5	55	Empty
6	160	Empty
7	75	Empty
8	198	Filled
J3->None		
Position	Hole Size (KB)	
1	85	Empty
2	200	Filled
3	28	Empty
4	195	Empty
5	55	Empty
6	160	Empty
7	75	Empty
8	198	Filled
J4->P4	195-65=130	
Position	Hole Size (KB)	
1	85	Empty
2	200	Filled
3	28	Empty
4	130	Filled
5	55	Empty
6	160	Empty
7	75	Empty
8	198	Filled
J5->None		
Position	Hole Size (KB)	
1	85	Empty
2	200	Filled
3	28	Empty
4	130	Filled
5	55	Empty
6	160	Empty
7	75	Empty
8	198	Filled

Scenario 2: Only one process is allowed per partition

In the first scenario, it is assumed that multiple processes could be assigned to the same partition. In this case, the total internal fragmentation for all three algorithms is 0 KB. This is because all the processes are allocated the exact amount of memory they ask for. The external fragmentation, therefore, is equal to the total amount of free memory, which is the sum of all the remaining hole sizes. The memory utilization percentage is calculated as the sum of memory taken by the stored processes divided by the sum of the holes when all are empty. Here, both best fit and first fit allocate memory to all the processes, meaning they both have the same memory utilization of 62%, which is the maximum possible in this scenario. Worst fit, however, immediately allocates processes to the two largest gaps, Positions 2, and 8, leaving nowhere for J3 to be allocated. This lowers its memory utilization percentage to only 39%. In terms of hole sizes, worst fit produced the fewest small holes, followed by first fit and then best fit. For this set of jobs specifically, first fit was the most appropriate, since it showed the maximum possible memory utilization, while having fewer uselessly small holes.

In the second scenario, it is assumed that only a single process could be assigned to the same partition. In this case, the total internal fragmentation is calculated as the sum of the remaining memory in the filled holes, while the external fragmentation is the sum of the empty holes. As for memory utilization, only best fit allocates memory to all the processes, giving it the best memory utilization of 62%. First fit does not find room for J5, so it uses only 42%, and worst fit does not find room for J3 or J2, so it uses 24%. Similarly to the first scenario, worst fit produced the fewest small holes, followed by first fit and then best fit. This caused worst fit to have the most internal fragmentation, as the entirety of the largest possible holes were allocated to each process, causing the largest waste of extra allocated memory. For this set of jobs, in this scenario, best fit was the most appropriate, since it was the only algorithm to allocate memory to all the processes, maximizing overall memory usage.

For a system with frequent small allocations, the best fit algorithm is most appropriate, as it maximizes the overall memory usage by minimizing the size of leftover holes after allocation. As shown in both scenarios, best fit always has the highest memory utilization percentage.

In theory, for a system with mixed workload size, worst fit should be the best algorithm, as it leaves the largest possible empty holes (assuming you can allocate more than one process to one partition). This ensures there is still room for large processes. In practice however, based on the calculations, best fit appears to be the best algorithm for mixed workload sizes. Best fit maximizes the overall memory usage, minimizes internal fragmentation, and leaves the large holes open as long as possible for big processes.

Part II – Concurrent Processes in Unix

https://github.com/e95400411-cmd/SYSC4001_A2_P2

The fork() system call creates a new child process, which has a copy of the memory address space of the calling process. Both processes continue executing concurrently from the same point after the fork, however the return value of fork() is different. The parent receives the PID of the child as a return value, indicating it is the parent, and the child receives 0, indicating it is the new child.

The exec() system call will replace the current process (code, data, stack, heap are all replaced) with the image of another binary, while keeping the same process ID. In our case, we choose to fork the main program into a parent and child process, while manually sharing the same memory (shm) by passing in a shmid. Then, we choose to make the child process execute differently from the parent by looking at it's PID. In order to specify what this new execution will look like, we use the system call execl("pro2", buffer, NULL), where "pro2" is the binary image that is being copied and run (if the process is the child). The subsequent arguments are a list of null terminated string arguments (buffer containing a shared memory ID). These will be passed the pro2's main argv argument.

Part III - Simulation Report

https://github.com/hicked/SYSC4001_A2_P3

Simulation 1

Trace 1 consists of a main program “init” that gets forked. Both the parent and the child will then exec two other programs (program1 and program 2 respectively):

A note: While FORK and EXEC are SYSCALLs 2 and 3 respectively, “SYSCALL, 2/3” will not work in the trace files, as 0-20 have been reserved specifically for external I/O devices. We should not be able to interact with I/O using the SYSCALL command, but this is just how it was implemented in the previous assignment. Therefore, while FORK and EXEC are system calls 2 and 3, they must be parsed in using their own arguments:

FORK, <delay>

EXEC <program_name>, <delay>

Additionally, after every FORK and EXEC, the currently active PCBs are saved, and can be seen in the system status. This can be helpful to get a more general sense of what those two commands do.

Lines 0-23 (FORK operation):

Similar to any other system call, this starts with a switch to kernel mode, saving context, and finding the FORK ISR vector (vector 2 at 0x0004-due to the defined vector ISR size). However, what is specific to FORK (inside its ISR) is that it will then clone its own PCB to create a child process (PID 1 in this case). It then calls the scheduler and returns via IRET.

Lines 24-246 (Child EXECs program1):

The child will be immediately scheduled to execute program1 since it has higher priority than the parent. The system switches to kernel mode again, finds the EXEC ISR (vector 3 0x0006), then loads the program to copy from disk (15ms per Mb * 10Mb = 150ms). It will then find a new partition (partition 4 in this case) for the new process (updating the PCB in the meantime). This involves checking the size first, to make sure that it can fit. Because of the way the algorithm is structured, it will use the best-fit method for this. Note that the parent process is currently in the ready queue, but if we were to have more than one processor, they would both run at the same time (which could present some problems related to concurrent real time systems such as race conditions). But once again, in this scenario, init would just wait for its child to finish (which is about to run program1).

Line 247-347 (Child runs program1):

Program1 contains a single CPU burst of 100ms. The child executes it fully, then terminates. This allows the parent (PID 0) to retake control at time 347.

Lines 347-619 (Parent EXEC program2):

The parent (still init, PID 0) now executes or copies what is in program2, similarly to how child did it with program1: kernel mode, find EXEC ISR, validate 15 Mb size, load (15ms each = 225ms), assign partition 3 (best fit), update PCB, scheduler, IRET. Two things to mention however, is that when exec runs, the programs on non-volatile memory (hard drive, SSD). This is why the loader is there: It loads the program from the slower device into memory, so that it can replace what is in child (or in this case now, the parent). Also, the partitions that are occupied are being kept track of, to ensure that the same partition is not allocated to two different processes.

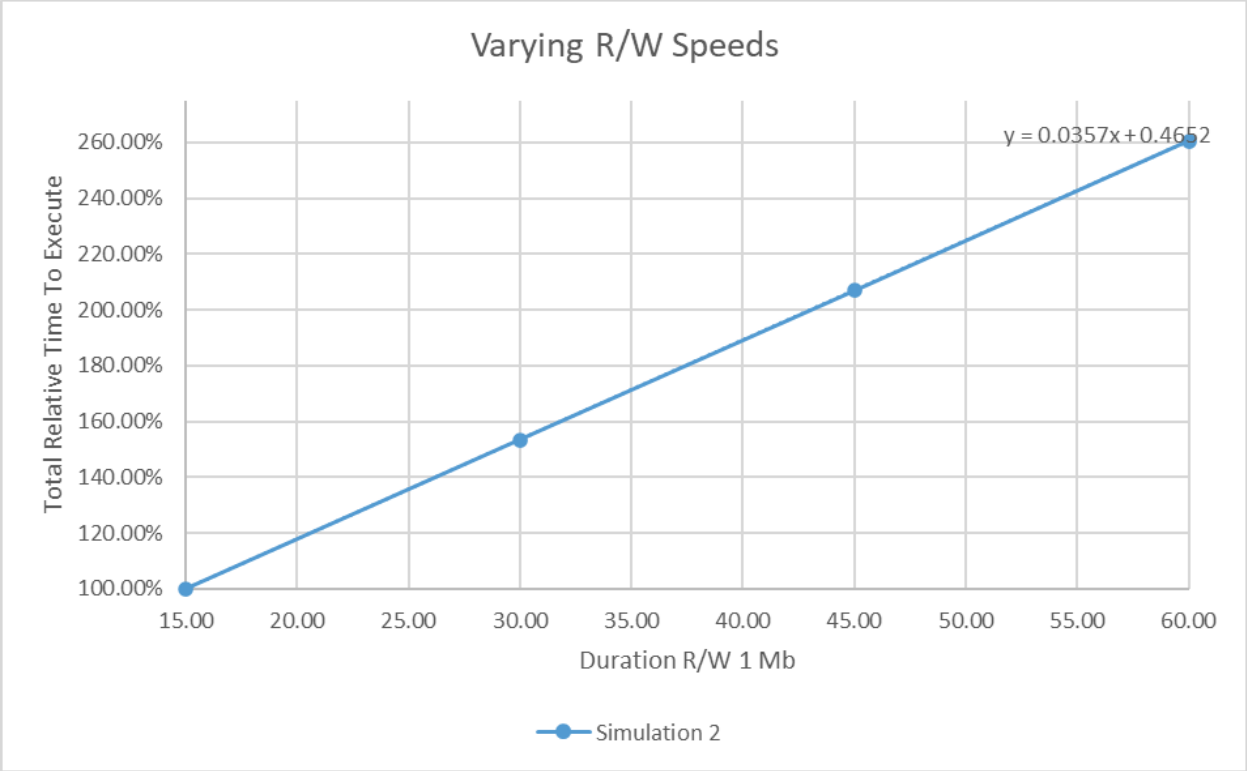
Lines 620-924 (Parent runs program2 and terminates):

Program2 contains a SYSCALL to device 4 (reminder that SYSCALL 0-19 is reserved for I/O). The system switches to kernel mode, finds vector 4, runs the I/O driver ISR (40ms), then polls the device for 250ms (from the device table) until I/O completes. After IRET at time 924, the parent process terminates (as well as the entire trace). Note that this differs slightly from the lab manual since we assumed that the ISR activity time (40ms) was NOT included in the device table (assuming the device table was only for I/O device delay).

The next simulations are a similar flow, so to avoid repetitiveness, I will only emphasize what is particular, or what stands out about each simulation or scenario.

Simulation 2

Trace 2 introduces nested forking and demonstrates the wait queue behavior with multiple processes. The initial fork creates child PID 1, which then EXECs program1. However, program1 itself contains another FORK call, creating a second child (PID 2). This results in three processes in the system: the original parent (PID 0) waiting, the first child (PID 1) waiting, and the child's child (PID 2) running. Both children eventually EXEC program2 (the “grandchild” PID 2 first, then PID 1 after it finishes). The parent then simply performs a CPU burst. The system status shows the PCBs and how the wait queue grows to multiple waiting processes, with PIDs 0 and 1 both in waiting state while PID 2 runs. We then wanted to find the impact of disk I/O performance, since EXECs require reading from non-volitive memory, and then writing to memory: We varied the read/write speed across four scenarios (15ms, 30ms, 45ms, and 60ms per Mb) to observe how process turnaround times scale with slower storage devices: It demonstrated that the EXEC overhead grows linearly with the loader speed while the rest of the execution remains unchanged. However, the loader is still one of the slowest steps in the flow, meaning that the impact was significant (execution went up 260% by the time it reached 60ms). This emphasizes the importance of having a fast device to read and write from, as we cannot have all programs in memory. We want to avoid it being the bottleneck so that the CPU is not wasted.




```
time: 31; current trace: FORK, 17
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   1 |        init |               5 |    1 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------+

time: 220; current trace: EXEC program1, 16
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   1 |    program1 |               4 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------+

time: 249; current trace: FORK, 15
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   2 |    program1 |               3 |   10 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+-------------------------------------------------------+

time: 530; current trace: EXEC program2, 33
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   2 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |    program1 |               4 |   10 | waiting |
+-------------------------------------------------------+

time: 864; current trace: EXEC program2, 33
+-------------------------------------------------------+
| PID |program name |partition number | size |   state |
+-------------------------------------------------------+
|   1 |    program2 |               3 |   15 | running |
|   0 |        init |               6 |    1 | waiting |
+-------------------------------------------------------+
```

Simulation 3

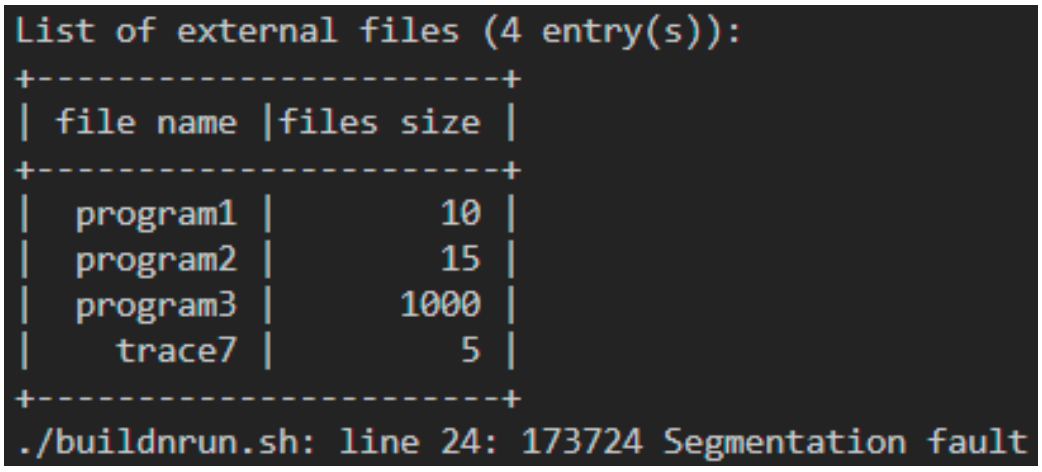
Trace 3 showcases the FORK logic the simulator enforces (as per the instructions): After FORK, the child skips everything inside IF_PARENT until ENDIF, then continues with its own code. That's why the child still runs its 10ms CPU burst (it lies outside the skipped region), even though it comes after the IF_PARENT trace. The parent then gets control, and it will perform a SYSCALL device 6, poll, and then END_IO. Note that our output differs slightly from the manual because we assumed END_IO was a fixed 1ms restore (ported over from assignment 1).

Simulation 4

Trace 4 is supposed to showcase that as soon as EXEC is called, the rest of the trace is replaced, meaning that in this example, the CPU burst and SYSCALL 8 will not be run, even though they are present in the child's trace: As soon as it hits EXEC program1, everything is replaced/lost.

Simulation 5

Trace 5 is a self-EXEC loop: After fork, the child EXECs program1, whose trace immediately EXECs program2, whose trace immediately EXECs program1, and so on... This creates an endless re-exec cycle with no CPU or I/O instructions, so the child never finishes and the parent's "CPU, 205" in IF_PARENT is never reached. In the real world, each EXEC would repeatedly free and reassign partitions (re-run the loader) forever, thrashing the memory and disk utilization. For our simulation, however, the program will just hang, until a segmentation fault happens (when stack runs out). Therefore, no execution/system status files are produced for this case.



Simulation 6

Simulation 6 shows a fork where the child tries to exec an oversized program and fails: In the execution, it prints “program is 1000Mb large,” and then “ERROR: EXEC failed: no suitable partition for program3.” This is because the largest partition that is available is 40Mb.

Simulation 7

Simulation 7 shows the system status filling partition by partition with each fork/exec of the program until all six partitions are occupied. The next fork then fails saying that there is “no suitable partition.”

```
time: 508; current trace: EXEC trace7, 5
+--------------------------------------------------+
| PID |program name |partition number | size |   state |
+--------------------------------------------------+
|   4 |      trace7 |               2 |    5 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |      trace7 |               5 |    5 | waiting |
|   2 |      trace7 |               4 |    5 | waiting |
|   3 |      trace7 |               3 |    5 | waiting |
+--------------------------------------------------+

time: 532; current trace: FORK, 10
+--------------------------------------------------+
| PID |program name |partition number | size |   state |
+--------------------------------------------------+
|   5 |      trace7 |               1 |    5 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |      trace7 |               5 |    5 | waiting |
|   2 |      trace7 |               4 |    5 | waiting |
|   3 |      trace7 |               3 |    5 | waiting |
|   4 |      trace7 |               2 |    5 | waiting |
+--------------------------------------------------+

time: 635; current trace: EXEC trace7, 5
+--------------------------------------------------+
| PID |program name |partition number | size |   state |
+--------------------------------------------------+
|   5 |      trace7 |               1 |    5 | running |
|   0 |        init |               6 |    1 | waiting |
|   1 |      trace7 |               5 |    5 | waiting |
|   2 |      trace7 |               4 |    5 | waiting |
|   3 |      trace7 |               3 |    5 | waiting |
|   4 |      trace7 |               2 |    5 | waiting |
+--------------------------------------------------+
```