OS Assignment-2

MemOS Group

January 2024

PART 1

Question 1

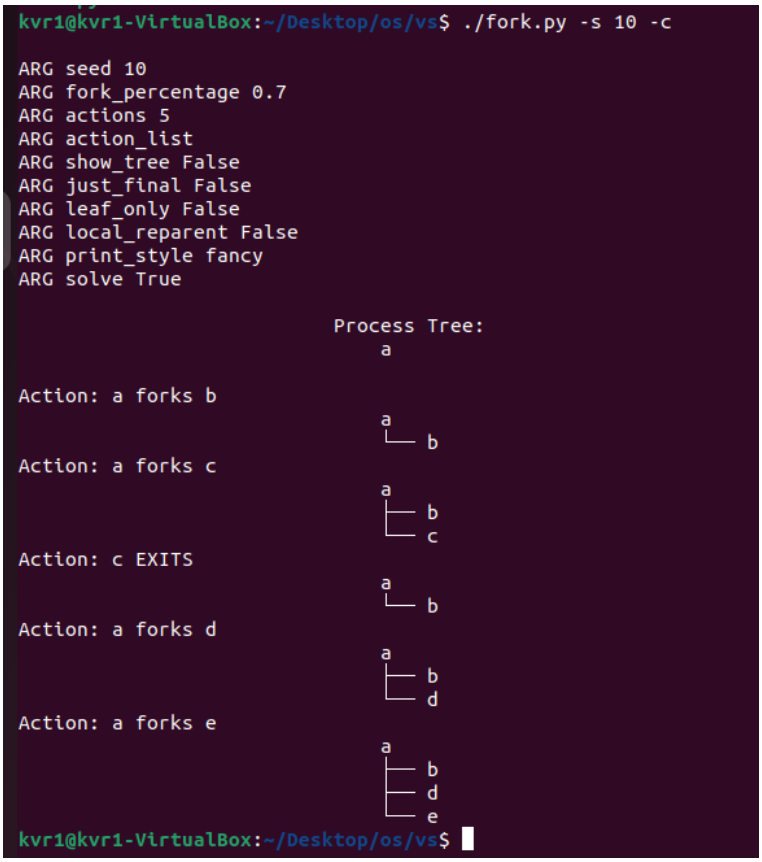

Figure 1: The output obtained after giving the command './fork.py -s 10 -c'

'-s' is used to sed a random seed. '-c' is used to check the process tree at each step. '-s 10' is used to indicate a flag or option for specifying a size or something related to 's'. easch step in the tree indicates forking process. Using different seeds tells us how the process tree varies based on different parameters.

Question 2

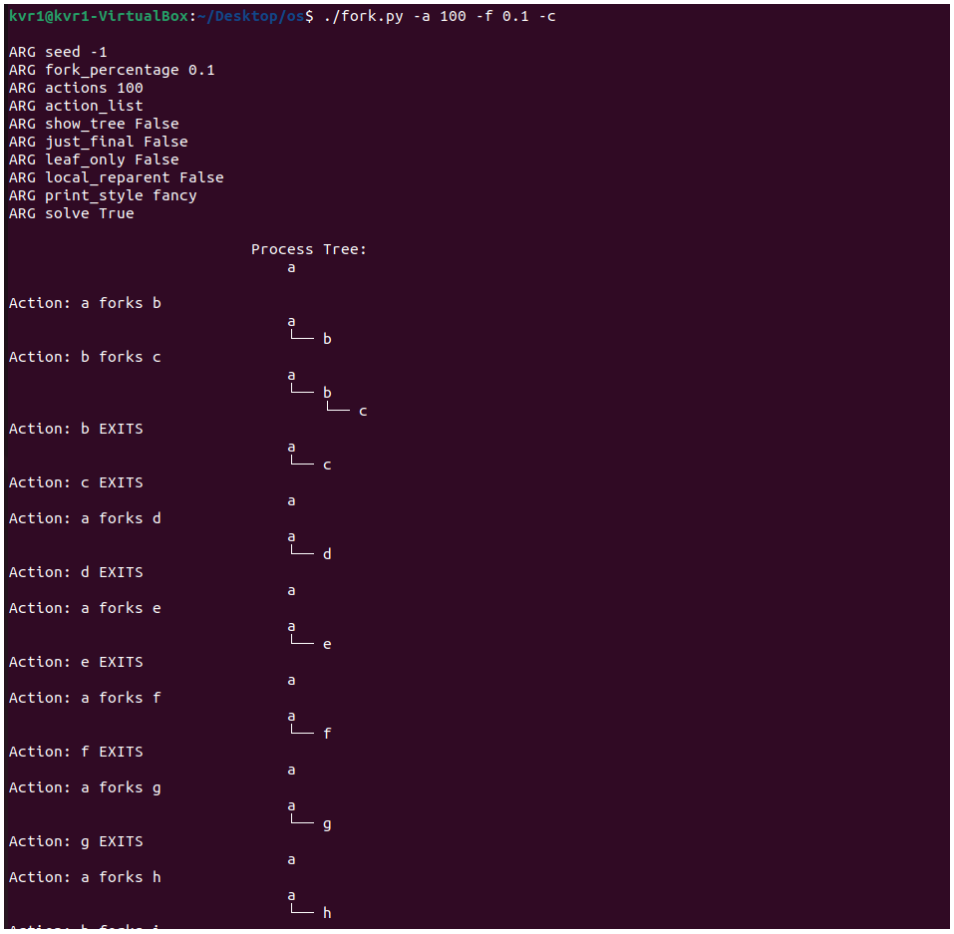

Figure 2: The output obtained after giving the command './fork.py -a 100 -f 0.1 -c'

'-a 100' specifies that 100 actions to be taken. '-f 0.1' sets the fork percentage as 0.1 means a 10% chance of forking and 90% chance of exiting. '-c' displays final process tree. Higher fork percentage generally lead to wide and deeper trees with more processes. Lower fork percentage lead to narrower and shallower trees fewer processes.

Question 3

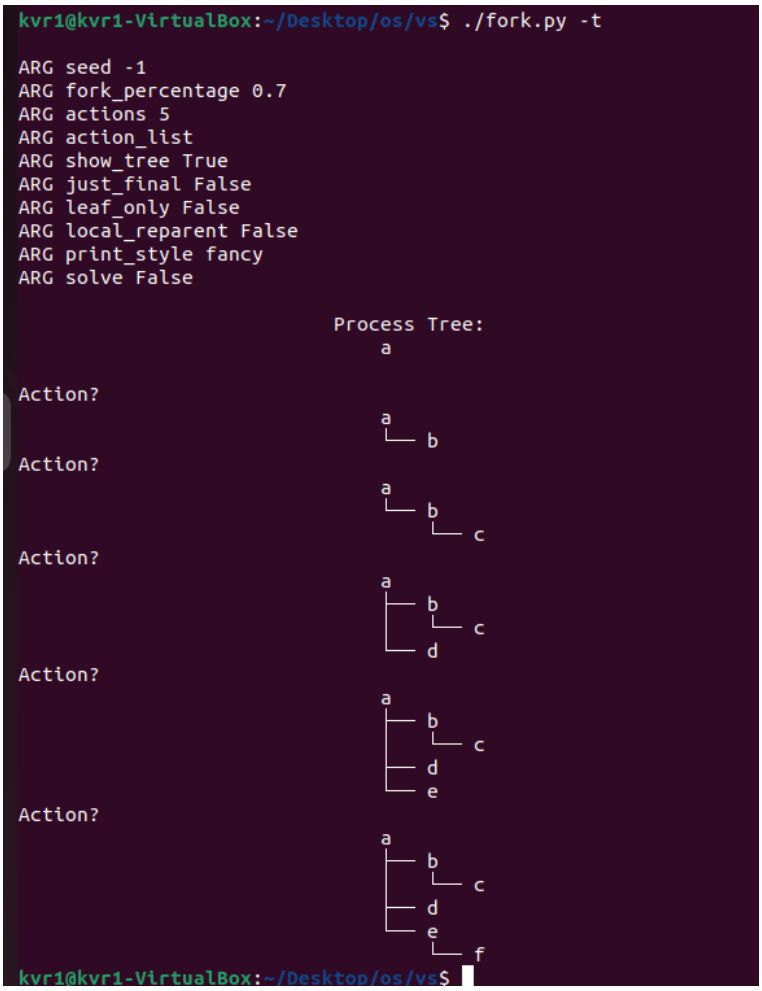

Figure 3: The output obtained after giving the command './fork.py -t'

Using '-t' switches the output to display the process directly. A node in a tree represents a process and parent-child relationships indicating forking actions. By examining this, such as no.of children per parent and their positions, we can check the sequences and can identify patterns which helps to discern the specific actions such as forking, termination and other manipulations.

Question 4

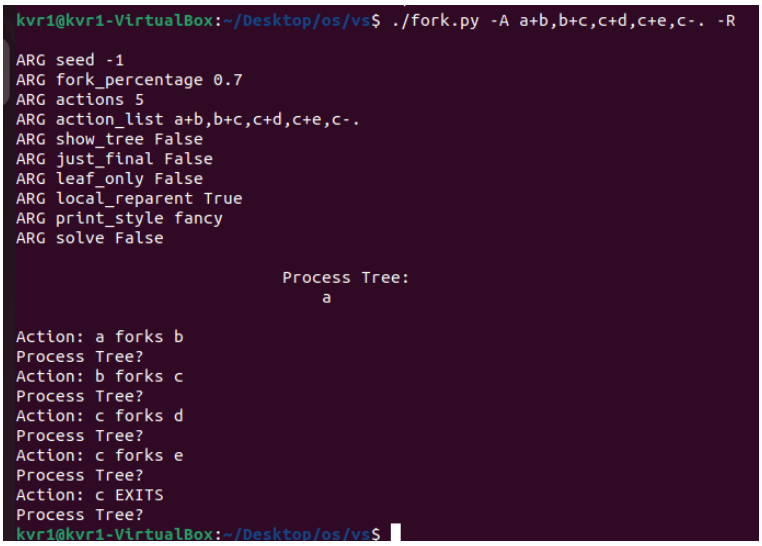

Figure 4: The output obtained after giving the command './fork.py -A a+b,b+c,c+d,c+e,c-. -R'

Here, the given command indicates a hierarchical structure with 'a' as parent of 'b', 'b' as parent of 'c', 'c' as parent of 'd' and 'e'. When 'c' exits, the tree undergoes changes and 'd' and 'e' are orphaned. These orphaned 'd' and 'e' are adopted by the init process. Using '-R' it reveals hoe the orphans are handled by showing their attachment to a new parent or as children of init process.

Question 5

Figure 5: The output obtained after giving the command './fork.py -F -s 123'

'-F' flag is used to eliminate intermediate steps and directly fills the final process tree. Using this approach will be beneficial because it focuses only on the end result of the process tree. We can observe how variations in initial condition affecting the resulting process by using different seeds.

Question 6

Figure 6: The output obtained after giving the command './fork.py -F -t -s 123'

Using '-t' and '-F' flags together in ./fork.py it displays the final process tree and examines the tree structure which allow for interfering the actions taken during the script's execution. If the parent-child relationships are clear, it will be easier to conclude the actions such as forking or exiting processes. If it is more complex, concluding the exact actions will be challenging. So experimenting with different seeds to know them is required.

PART 2

Question 1

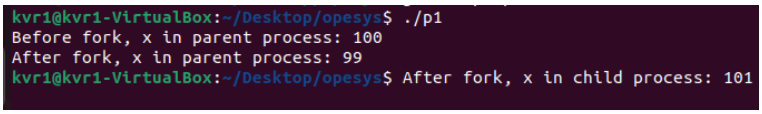

Figure 7: The output obtained after running the code

fork() creates a new process which is exact copy of the original process which is the parent process. After fork() both processes are independent. So changes made in one process does not affect others. Here 'x' is incremented to 100 in child's process where as 'x' remains 99 parent because it is unaffected.

Question 2

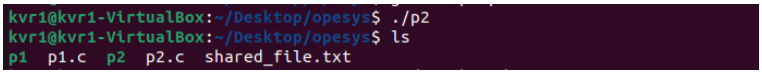

Figure 8: The output obtained after running the code

After fork() both child and parent can access the file descriptor because child's process is a copy of parent's process. While writing concurrently, the order in which they write will be unpredictable which leads to incomplete data. Overwriting may happen which lead to data corruption or loss of integrity. Using synchronization, separate file descriptors and non-blocking of I/O can prevent the issue that arise while writing concurrently.

Question 3

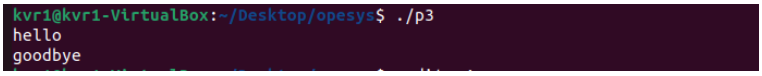

Figure 9: The output obtained after running the code

The childs process starts immediately after calling the fork() however we can use usleep(10000) to make the parent process get delayed slightly. We can also use synchronization or process priorities also to make this happen.

Question 4

```
kvr1@kvr1-VirtualBox:~/Desktop/MemOS_grp_assgn$ ./p4
hello (pid:19337)
parent of 19338 (pid:19337)
kvr1@kvr1-VirtualBox:~/Desktop/MemOS_grp_assgn$ child (pid:19338)
total 100
-rw-rw-r-- 1 kvr1 kvr1    69 Jan 31 22:13 output.txt
-rwxrwxr-x 1 kvr1 kvr1 16000 Jan 31 22:13 p1
-rw-rw-r-- 1 kvr1 kvr1   417 Jan 30 17:19 p1.c
-rwxrwxr-x 1 kvr1 kvr1 16232 Jan 31 22:13 p2
-rw-rw-r-- 1 kvr1 kvr1   969 Jan 31 20:41 p2.c
-rwxrwxr-x 1 kvr1 kvr1 16080 Jan 31 22:13 p3
-rw-rw-r-- 1 kvr1 kvr1   443 Jan 30 17:29 p3.c
-rwxrwxr-x 1 kvr1 kvr1 16216 Jan 31 22:14 p4
-rw-rw-r-- 1 kvr1 kvr1  1146 Jan 30 07:05 p4.c
-rw-rw-r-- 1 kvr1 kvr1   635 Jan 30 17:37 p5.c
-rw-rw-r-- 1 kvr1 kvr1   625 Jan 30 17:39 p6.c
-rw-rw-r-- 1 kvr1 kvr1   493 Jan 30 17:41 p7.c
-rw-rw-r-- 1 kvr1 kvr1  1535 Jan 31 20:08 p8.c
```

Figure 10: The output obtained after running the code

variants like execl() and execv() pre-date others and catered to specific argument passing styles or operating system capabilities. Variants like execle() and execvpe() allow modifying the environment for the new process. Overall, the multiple variants offer a trade-off between simplicity, control, and potential security risks. Choosing the appropriate exec() form depends on your specific program requirements and the desired interaction with the executed program.

Question 5

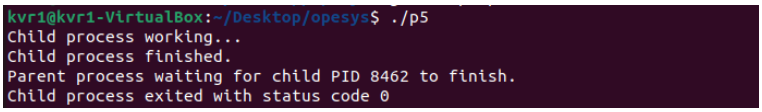

Figure 11: The output obtained after running the code

What wait() returns: returns an integer which represents status of child process which is usually represented as '0' which indicates no errors. Process ID of child's process is terminated which acknowledges the parent for which child process it was waiting.

what happens if we use wait() in child's process: Calling wait() in child's process is pointless. If it was run it makes child process to wait for other which makes no sense. It might end up in a deadlock where it waits for itself to finish which is impossible scenario.

Question 6

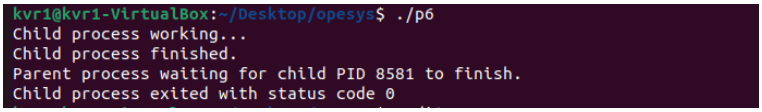

Figure 12: The output obtained after running the code

wait(): Used to wait for a child process usually when only one child exists.

waitpid(): When there are multiple processes, we use waitpid() with ID to wait for a particular process without affecting other processes. It also gives more information like signal termination deatils or can access core dump info by using different options. waitpid combination with other wait options helps non-blocking wait behaviour.

Question 7

Figure 13: The output obtained after running the code

Here in Child process, file descriptor associated with standard output is closed. When child process calls print(). This attempts to write to standard output but as STDOUT is closed, the output after calling print() is not displayed. So using STDOUT does not print any further output.

Question 8

Figure 14: The output obtained after running the code

We create an unconditional pipe, represented by two file descriptors one for reading, one for writing. Child1 closes the unused reader end of the pipe, writes a message to the pipe and then child2 reads the message and prints the received message. Meanwhile parent process waits for both child's processes to finish using and then prints a message indicating completion.