CS3510: Operating Systems I

Quiz 1: Autumn 2021

Srivatsan – T: CS20BTECH11062

1.The OS uses a 'mode bit' provided by the hardware to distinguish between the system calls which are run in kernel mode and the function calls typically run in user modes. The transition between the modes is implicit as in the hardware bit changes to kernel mode (i.e it changes to 0) whenever a system call must be performed and resets back to user mode after executing the call to prevent user from explicitly tampering with the 'mode bit'.

By default, the machine starts in kernel modes and loads up the OS. It then switches to user mode when the execution of user program starts. When the user wants to perform a system call that is a privileged instruction (i.e can be run only in kernel mode) the mode bit is changes to 0 and the OS takes control of the computer and executes the system call and changes to mode bit to 1 before returning to the next instruction of the user program. Examples of such privileged instructions include I/O control, time management and interrupt management.

Thus the OS takes care of security even when its executing a system call on behalf of a user program.

2. The function fork() creates a child process for the given process and begins parallel execution of the lines of code that follow the fork() function call with one difference between the parent and child process. Pid of the child version is 0 while the parent version has the pid>0 that specifies the pid of the child process that was created.

Fork() returns the pid which can be 0 or >0 as mentioned above. It returns a negative value whenever it couldn't create a child process and the function fails. There can be various reasons for the fork() function failing to create a child process. For example, there can be shortage in the system memory thus making the fork() unable to reserve memory space and copies of the data from the parent process thus failing to create a child process. There can be a maximum limit of child process that an OS permits a parent process to have, and if the limit is exceeded the fork() might fail to create another child for the same parent.

The function exec() accepts the directory of a new process and replaces the current process with the process found in the file directory. It completely erases the current memory space and its values and loads up the registers with the values of the new processes found in the directory and the program terminates when the newly loaded process completes its execution. The function fails when the file directory specified as argument to the function doesn't contain a valid process that can replace the current process execution.

The function exec() also fails and returns control when an error occurs in the new process that replaced the old process.

3. Virtual machines are typically slower when compared to native OS's especially when the underlying architecture of the host OS is different from the virtual OS we try to run. This is because for every instruction we want to execute, we must translate it to the instruction set of the host OS for it to be run on hardware. Sometimes there is no one to one correspondence between the instruction so there arises a need to run multiple target instructions to simulate the instruction on the host OS thus resulting in slower speed of execution. So, it doesn't truly reflect the time taken by the process on a native OS that we are trying to simulate using virtualization which might be a data that can be crucial to the process's feasibility.

Virtual machines have higher hardware needs than regular machines especially when the process that is to be executed is hardware intensive. This is because the system resources such as memory and CPU are shared between the various virtual machines so the computing power and other resources that a single virtual machine ends up getting decreases which can cause bottlenecks during execution. So, we need to equip a virtual machine with more powerful hardware for smooth functioning of a process than to a regular machine.

4. Just before the execution of fork() the value of NUMS array is as follows {1,1,1,1,1}. This is caused by the for loop just before it that replaces all values in the array with 1.

Now the fork() gets executed and splits the execution of consecutive lines into the parent version with pid>0 containing the pid of the child process and the child version with pid = 0.

Since the nums array is declared globally, it simulates a shared memory space for the parent and child processes, and both try to write data into the same array nums.

The parent version of execution executes the block under the condition pid>0 and encounters a wait() statement right in the beginning of the block which makes it wait until the child execution stops to continue further the block.

The child version of execution executes the block under the condition of pid = 0 and encounters a for loop changing contents of the nums array as nums[i] += -i and reaches its termination after the loop and the print statement.

So the nums array now contains 1-0, 1-1, 1-2, 1-3, 1-4 i.e {1,0,-1,-2,-3} as a result of the for loop in the child execution.

The child process prints the corresponding i and nums[i] for each iteration of the loop and output at line X comes out as:

```
CHILD: 0 1CHILD: 1 0CHILD: 2 -1CHILD: 3 -2CHILD: 4 -3
```

Now the wait statement invoked in the parent execution exits since all the children have completed their execution and proceeds into a for loop changing the contents of the nums array yet again as nums[i] *= -i and reaches its termination after the loop execution and the print statement.

So, the nums array now contains 1*0, 0*-1, -1*-2, -2*-3, -3*-4 i.e {0,0,2,6,12} as a result of the for loop in the parent execution.

The parent prints the corresponding i and nums[i] values for each iteration of the loop and output at line Y comes out as:

```
PARENT: 0 0PARENT: 1 0PARENT: 2 2PARENT: 3 6PARENT: 4 12
```

And the nums array finally contains the values {0,0,2,6,12}.

5. Yes. Sometimes when there are multiple hindrances for a process to be run, i.e not be able to be in the ready queue, we would want all the hindrances to be solved before we let the process run again. So naturally we would place the process in all the waiting queues that are hindering its execution and bring It back to ready queue only when it has come out of all the various waiting queues.

Example: Consider a process that forks at some point and the parent process requiring a I/O after the forking. Now the process can't run further because of 2 reasons. **One**, because there is an active child process which must finish executing before we run and end the parent process. **Two,** because there is an I/O need for the parent process that needs to be satisfied before it can be resumed.

Now if the parent process is placed only in the child termination queue, it would resume once the child process is completed and not care about I/O resulting in non-desirable outputs for the parent process. On the other hand, if the process is placed only in the I/O queue, it would resume once its I/O needs are satisfied which might result in the child process becoming an orphan which is an error in some OS's.

So, the ideal way to include the parent process in both child termination queue and I/O wait queue and add it to ready queue only when there is no copy of the process in any waiting queues.

Similarly, the process can also take a long time to complete and thus ended up ready queue because of time slice expiry. If the process also has a child, it makes sense to add it in child termination queue before it ends up in ready queue. A process being in ready queue signifies that all its dependencies have been satisfied and its ready to run any time. So we must add the processes to ready queues only when it's one or more dependencies have been met and settled and that can be done only if its placed in all the waiting queues that it has to be placed in and added to the ready queue only when it's been dequeued from all the waiting queues.