Name: Joseph Hany Boulis
ID: 900182870

Design Document LAB 10

1) PFork System call:
    a) Purpose of this syscall is to create two children from a parent process: one is an active child and the other is a standby one in case the parent exitted.
    b) Since this syscall has no input we used the SYSCALL_DEFINE0 macro which means that this syscall takes no input.
    c) It outputs the PID of the active child.
    d) Its implementation:
        i) I replicated the creation of the child which takes place in the _do_fork in order to have two similar processes with two different PIDs.
        ii) I had to capture the task struct of each of these processes by reversing the assigned PID (which was produced by _do_fork) and getting the task structs.
        iii) I needed these structs for three main reasons:
            (1) Each task struct (either the one related to the active or the standby processes) should be aware of the PID of its sibling task. I,e, the active process should save the standby process PID in its task struct. Similarly, the standby process should save the active process PID in its task struct. Why do we need this? To be able to resume execution of the standby process right after the termination of the active process.
            (2) If we are in the active process, we need to fill the pfork_active_pid field with a non-zero (mentioned in the assignment document) value to indicate that this process was created by calling the pfork syscall. Likewise, standby process should save a non-zero value in its pfork_standby_pid. The chosen non-zero value is 1.
            (3) We should initialize the status of both the active and standby process with zeros.

        iv) After creating the standby child, we should be able to put it on the wait queue by sending a stop signal using the following function kill_pid(struct* pid,SIGSTOP,1)
2) get_pfork_status system call:
    a) Purpose of this syscall is to get the status of the current running process of the pfork.
    b) Since this syscall has no input we used the SYSCALL_DEFINE0 macro which means that this syscall takes no input.
    c) It outputs the pfork_status.
    d) Its implementation:

        i) In order to get the current running thread we used the global variable current which returns a pointer to task_struct for the current running thread.
        ii) Next, we point to the attribute pfork_status and return it.

3) Get_pfork_sibling_pid system call
    a) Purpose of this syscall is to get the status of the pid of the sibling of the current running process.
    b) Since this syscall has no input we used the SYSCALL_DEFINE0 macro which means that this syscall takes no input.
    c) It outputs the sibling PID.
    d) Its implementation:
        i) If the current process is the active we return the PID of the standby
        ii) If the current process is the standby we return the PID of the active

4) Set_pfork_status system call
    a) Purpose of this syscall is to get the status of the pid of the sibling of the current running process.
    b) Since this syscall has 1 input we used the SYSCALL_DEFINE1 macro which means that this syscall takes 1 input.
    c) It has no output.
    d) Its implementation:
        i) If the current process is the active set both the status of the active and the standby
        ii) Else set the status of the standby only
5) Pfork_who system call:
    a) Purpose of this syscall is to know which process is running now.
    b) Since this syscall has no input we used the SYSCALL_DEFINE0 macro which means that this syscall takes no input.
    c) It outputs a number indicating whether the current process is the active, standby, or even not a Pfork process.
    d) Its implementation:
        i) If the current process is the active one return 2
        ii) If the current process is the standby one return 1
        iii) Else return 0
6) Sched.h file:
    a) In order to be able to do all the above mentioned work, we should add those three attributes:
        i) long pfork_standby_pid;
        ii) long pfork_active_pid;
        iii) int pfork_status;

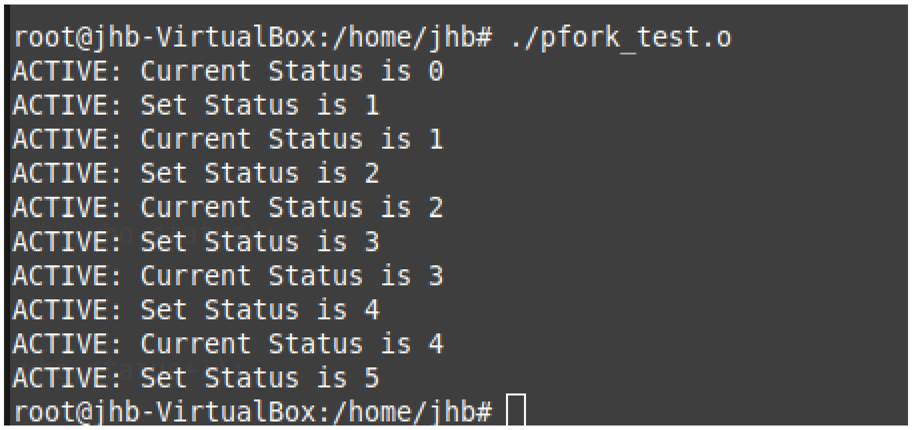
       in the sched.h file. Specifically, in the task_struct after the endif of CONFIG_THREAD_INFO_IN_TASK.
7) Exit.c:

a) In order to make the standby process execute after the active process exits, we should modify exit from group function and add the following logic:

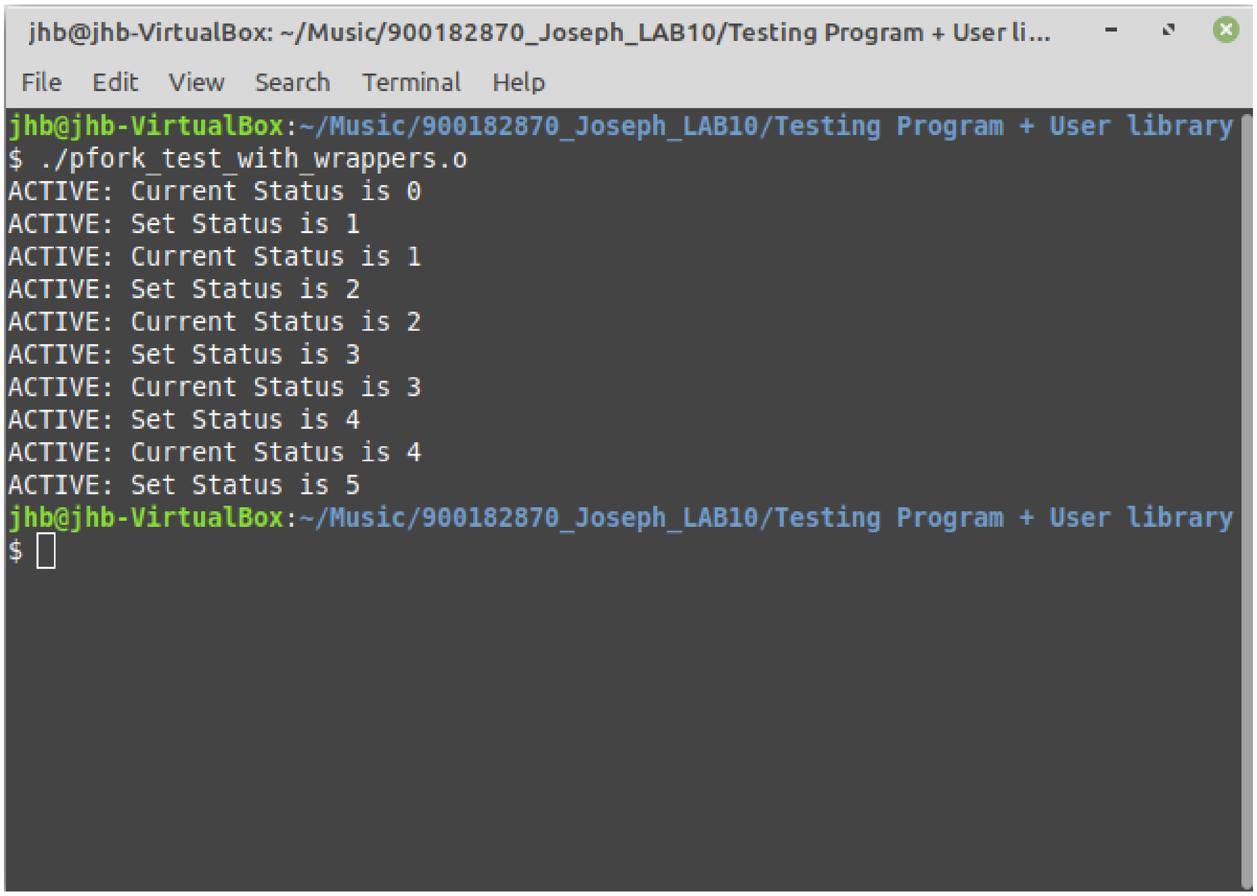
   i) If the current process is the active one and is exiting, we should send continue signal to the standby process to be put on the running queue by using kill_pid(pid_struct,SIGCONT,1).

8) Finally, we should edit syscall_64.tbl file and add the above syscalls in it to be seen by the kernel while compiling.

Screenshots:



```
root@jhb-VirtualBox:/home/jhb# ./pfork_test.o
ACTIVE: Current Status is 0
ACTIVE: Set Status is 1
ACTIVE: Current Status is 1
ACTIVE: Set Status is 2
ACTIVE: Current Status is 2
ACTIVE: Set Status is 3
ACTIVE: Current Status is 3
ACTIVE: Set Status is 4
ACTIVE: Current Status is 4
ACTIVE: Set Status is 5
root@jhb-VirtualBox:/home/jhb# 
```



jhb@jhb-VirtualBox: ~/Music/900182870_Joseph_LAB10/Testing Program + User li...
File Edit View Search Terminal Help

```
jhb@jhb-VirtualBox:~/Music/900182870_Joseph_LAB10/Testing Program + User library
$ ./pfork_test_with_wrappers.o
ACTIVE: Current Status is 0
ACTIVE: Set Status is 1
ACTIVE: Current Status is 1
ACTIVE: Set Status is 2
ACTIVE: Current Status is 2
ACTIVE: Set Status is 3
ACTIVE: Current Status is 3
ACTIVE: Set Status is 4
ACTIVE: Current Status is 4
ACTIVE: Set Status is 5
jhb@jhb-VirtualBox:~/Music/900182870_Joseph_LAB10/Testing Program + User library
$ 
```