CS330 Assignment 1 Readme

Ujwal Jyot Panda(201060), Harsh Trivedi(200422), L Gokulnath(200542), Akhil Jain(200077)

Part A

It seems fork call won't run more than approximately 64 (may vary from system to system) when called from a single process as it fails every time after 64 fork calls.

Brief description of the system calls

1.) **getppid** :
 It takes no arguments. If a parent process exists for the calling process, it returns the pid of the parent process, else returns -1. First we acquire the wait_lock and the p->lock for the calling process to access the parent. If there is no parent, then we release these locks and return -1. But, if there is a parent, then we access that parent, acquire the p->lock for the parent and get its pid, release the locks and return the same.
 Syntax - `getppid(void);`

2.) **yield** :
 It takes no arguments. If called, the calling process voluntarily calls the scheduling algorithm for 1 CPU cycle. Here we use the yield() function defined in kernel/proc.c
 Syntax - `yield(void);`

3.) **getpa** :
 It takes 1 argument: a virtual address. We use the formula walkaddr(p->pagetable, A) + (A & (PGSIZE - 1)), using walkaddr from kernel/vm.c to compute the physical address corresponding to virtual address A and return the same. If the virtual address is invalid, then 0 is returned.
 Syntax - `getpa(int* A);`

4.) **forkf** :
 Very similar to fork function call, to execute the custom function, we change the user process's program counter to point to the memory

address of the function f. The PC returns to the main after executing the function f, since when fork was called, the ra register would have been set to the point in the main function where fork was called, and it isn't changed thereafter. If the return address of the function f is 0, then 0 is returned to main, similarly if return address is any other number, the same number is returned to main. So if the function f's return value is 1, the child executes the code which the parent would have executed in main in case of a normal fork. If f's return value is -1, the child would have shown an error according to the code in the main.

Output when return value is 0:

```
Hello world! 100
4: Child.
3: Parent.
```

Output when return value is 1:

```
Hello world! 100
4: Parent.
3: Parent.
```

Output when return value is -1:

```
Hello world! 100
Error: cannot fork
Aborting…
3: Parent.
```

Syntax - `forkf(<function pointer>);`

When the return type is changed to void and the return line is commented, the function returns 1 in the child process. However, on removing the fprintf command in the function f (and the variable initialisation), the function returned 0. So it seems the behaviour is undefined, and basically depends on what all system calls are called in f. The fprintf command possibly makes some changes in the p->trapframe->a0 register, since if that was not the case, the return value should have been 0, as it had been set in the code for the fork function.

5.) **waitpid** :

Takes two arguments: an integer (child_pid) and a pointer (which will store the return value of waitpid). If child_pid is passed as -1 then we

call the wait function. Otherwise, we loop through the process table searching for the child-processes of the given process. If we find a child-process with the given pid, then we return the pid of the child-process. If the parent gets killed or there are no children of the parent then we return -1.
Syntax - `waitpid(int child_pid,int* ret_pointer);`

6.) **ps :**
We loop through all the processes in the process table. After acquiring the appropriate locks, if the state of a process is “UNUSED” then, we leave it and continue the loop. Else, we check the validity of the process state. Then, we get the parent_pid of the process using appropriate locks.
We now need to define three new values in the proc structure : ctime (creation time), stime (start time) and etime (etime). Process gets allocated during the fork call, so we initialise ctime in allocproc. Process is first scheduled in the forkret call, which gives us the stime(set as -1 if the process is not yet scheduled). End time is updated from the exit system call. Now, we need to check whether the process is in the zombie state or not. If it is there in the zombie state, then we simply output the (etime-stime). Else we get the current time using the ticks variable and output (current time - stime).
Syntax - `ps(void)`

7.) **pinfo**:
Similar to ps, we use the **copyout** function defined in kernel/vm.c for copying the struct from kernel’s virtual memory to user process’s virtual memory. We also acquire and release the necessary locks as done in ps.
Syntax - `pinfo(int pid, struct procstat* pstat);`