CS330A: Design Assignment 1

Group 15

Aarchie [200004] Kembasaram Nitin [200505]
Harsh Jain [200412] Udit Prasad [201055]

1-A

(a) uptime

We have implemented this user call using the pre-defined system call **uptime**
command - *uptime*

(b) forksleep

After checking for given constraints on the arguments, we forked a child and if the argument is
1: parent process sleeps for 5 ticks and prints its pid whereas child prints without sleeping.
0: parent process prints its pid without sleeping whereas child prints it after sleeping for 5 ticks.
command - *forksleep m n*

(c) pipeline

After checking for given constraints on the arguments, we created a pipe which will be used by all the processes to pass its value to the next process.
We are using recursion to create children according to the given argument and each child is adding its pid to x and passing the value in pipe to its child.
command - *pipeline n x*

(d) primefactors

After checking the constraint on the argument, we created a pipe and using the technique from previous question, we created a pipeline and used recursion to create process for different primes and parent process will pass the remaining value to the child process (next prime) **command** - *primefactors n*

1-B

(a) getppid

i First, we used the process table entry of the calling process to retrieve the parent of that process

ii Then, we retrieved the pid of parent from its process table

(b) yield

We have implemented this system call directly using the one given in *proc.c*

(c) getpa

i First, pointer is taken as an argument from the user program

ii Then, using this expression
PA = $walkaddr(p\rightarrow pagetable, A) + (A \ \& \ (PGSIZE - 1))$
we calculated the physical address.

(d) forkf

- First, function address is taken as argument from the user program.
- Now, we used the basic fork() implementation to implement forkf, In fork, the child's program counter is set same as the parent's one. But we changed it to the passed address i.e. function's address by using *np→trapframe→epc = f;*. Hence the child will first execute the function after returning to user mode.

1. When the return value of f is 0 :
   Parent → forkf(f) will return the pid of child, which will be a positive value and thus *if(x > 0)* block executes
   Child → First, f is executed as explained above. We have used *np→trapframe→a0 = 0;* which ensures that child returns 0 to forkf as a0 is EAX register but it gets overwritten by return value of f and thats why x = 0 in this case which leads to execution of *if(x==0)* block.
2. When the return value of f is 1 :
   Parent will run similar as in **(1)**
   In case of child, when the return value is 1, then x =1 after f is completed which again leads to execution of *if* $(x > 0)$ block
3. When the return value is -1
   Parent will run similar as in **(1)**
   For child, x = -1 after f is completed which leads to printing of error message that fork was unsuccessful.
   (if $(x < 0)$ *failure)*
4. When the return value is other than 1,0 and -1 , then if the return value is positive, result will be same as **(2)** whereas if the return value is negative, result will be same as **(3)**
5. When f is made a void function, we tried these steps:
   - When we commented the whole function f(), then x was 0 in case of child process which means the EAX register was not changed (np→trapframe→a0) which we cross-checked by changing the values of this register.
   - When whole function was not commented out, then again the value of this register was overwritten

(e) waitpid

- waitpid takes 2 argument i.e. pid and status.
- If the first argument is -1, we return *wait(status)*
- We initialise *childFound = 0*
- Now, we scan through all the processes to match its pid with the passed pid and if it matches, we check if the parent is the calling process or not. If we are successful we change *childFound* = 1.
- If *childFound == 1*and that process's state is *zombie*, we return the pid of that process i.e. passed pid
- If we cannot find such a process, we return -1

(f) ps

- We created 3 more fields in the proc structure i.e. *start_time*, *creation_time* and *end_time*
- creation time was set during allocation of the process i.e. in the function *allocproc(void)*
- start time was set when the process was scheduled for the first time in the function *scheduler(void)*
- end time was set when the process executes the *exit* system call
- etime was set as *current time - start time* for non-zombie process and *end time - start time* for zombie ones
- ppid was calculated as we did in **(a)**, size and cmd(name) was extracted from proc structure.

(g) pinfo

- We created a struct with the fields mentioned in the question in **procstat.h**
- Then we called pinfo from user program with 2 arguments i.e. pid and pointer to procstat structure
- If pid is -1 , then we keep p = *myproc()* otherwise p will be the process with that pid
- Now in system call implementation, we made a procstat and filled all the required fields using the proc structure of **p**
- After that , we copy the content of this newly created procstat to passed struct pointer using *copyout*. We took help of *filestat* system call to implement this.