README

Implementation of all the system calls is in sysproc.c :-

A. **getppid** (Function sys_getppid):

We call the myproc() function, which returns the current calling process. Now, myproc() returns also a proc struct, which has a field parent, containing the pointer to the parent of myproc(). So, we call myproc()->parent, which is the proces corresponding to the parent of the calling process, and then we call the pid field by **myproc()->parent->pid**, which is present in the proc struct of the myproc()->parent, which is the ppid of the calling process.

B. **yield** (Function sys_yield):

We call the yield() function call from proc.c file, which de-schedules the calling process by changing the filed myproc()->state to RUNNABLE from RUNNING. Appropriate locks are used for this purpose.

C. **getpa** (Function sys_getpa):

We take as an input the virtual address A of the variable declared by the user using argaddr() function. Then, we pass the given virtual address A to the walkaddr function present in vm.c file and get the physical address using the formula:

walkaddr(myproc()->pagetable, A) + (A & (PGSIZE - 1))

D. **forkf** (Function sys_forkf):

Implementation:

We call the forkf function which is created by us in the proc.c file from the sysproc.c file where we have declared the sys_forkf system call. It takes as input the function pointer which is passed in the forkf system call.

In the forkf function in the proc.c file, we use the same idea as used by us in the fork function declared in the same file proc.c. We just need to change the point of execution of the child which is created by the parent when forkf is called. So, after copying the trapframe of the parent to the child, we just change the trapframe program counter of the child, so that the parent continues its execution as it was doing earlier, and the child first executes the function called in the forkf system call and then continues execution from the point forkf() is called. We do this by just changing the epc field of the child's trapframe after copying the parents trapframe values :

np->trapframe->epc = *((int*)(f))

Questions:

Q) Explain the outputs of the program when the return value of f is 0, 1, and -1.

When return value is 1:

A) The a0 register of child process (which calls f function) gets value 1. This is the value which is returned to x.

Hence the child runs the condition of (x>0).

This means the output is of type {pid: Parent} twice (parent and child) with some of the characters mixed… This mixing is due to scheduler-ordering of parent and child as the wait(0) appears after fprintf in (x>0) case.The sleep in first statement of executing conditional does not disambiguate since both parent and child sleep for 1 second.

When return value is 0:

In this case, we have the child process having x=0; which executes the last else condition.

Since parent sleeps for 1 second upon entering conditional, the child prints first. Hence there is no mixing

When return value is -1:

In this case, the x<0 block runs for child while x>0 block runs for parent. Since only parent sleeps for 1 second, child outputs first; resulting in no mixing.

Q) How does the program behave if the return value of f is changed to some integer value other than 0, 1, and -1?

A)Since we check the cases on x wrt 0, any positive return value is analogous to return value of 1. Similarly, any negative return value is analogous to return value of -1 from f.

Q) How does the program behave if the return type of f is changed to void and the return statement in f is commented?

A)We get the case of x>0 for child; which means mixed output. This is due to the fact that the function calls (passing and returning use a0 register); here they store value of 1 in register in trapframe of child process which does not get reset to 0.

E. **waitpid** (Function sys_waitpid):

We take 2 arguments from the user wrapper, an integer (p) and the pid. If the address of the arguments p or pid is negative (error), we return -1. Else we return waitpid(pid, p) which we have declared in the proc.c file. Waitpid waits for process with the given pid to complete and returns the pid of the process it waits for.

In the waitpid() function in the proc.c file, we use the idea as used in the implementation of the wait() function, just we change the if condition that the parent process, instead of waiting for any of its child, waits for the child with the given pid. The modified if condition is:

if(np->pid == pid && np->parent == p)

The edge cases are well-handled by returning -1 in caes of any errors.

F. **ps** (Function sys_ps):

This is the wrapper function around the function ps(). We simply call the ps() function defined in proc.c through this system call function.

We have added 3 new variables of type uint64 to the proc structure :-

- ctime - Stores the creation time of the process

  We update this value by using the ticks which are obtained (as used in the uptime() system call) while allocating the process for the first time in the fork() function in the proc.c file from the init process.

- stime - Stores the start time of the process

  We update this value by using the ticks which are obtained (as used in the uptime() system call) while scheduling the process for the first time in the forkret() function in the proc.c file.

- etime - Stores the execution time of the process

  We update this value by using the ticks which are obtained (as used in the uptime() system call) while exiting the process in the exit() function in the proc.c file from the init process. If a process is running at the time of ps() call, we just update its value to the current value of ticks that we obtain and subtract stime (start time) from it to get the execution time.

  We take the idea from the procdump() function to get the current state and the name of each process.

  We run a for loop over all the processes and print all the required values from the proc structure of the process which already stores all the values corresponding to the values of the items asked.

G. **pinfo** (Function sys_pinfo):

In this system call, we just have to add the values that we obtained in the ps() system call to the structure which is called by the user wrapper which provides 2 arguments, the pid of the process whose information is required and the pointer to the procstat structure.

We first define a new file procstat.h in the kernel containing the structure procstat. Then, we take the inputs in the file sysproc.c of the pid and the address of the structure as passed by the user and pass them to the pinfo() function implemented in proc.c file.

In the pinfo() function of proc.c, we first create a structure procstat and store the values we had printed in the ps() call int the structure. Now, the work left is to transfer the values of this structure to the address of the struct pointer in the user wrapper that we are having. We use the copyout() function for this purpose:

copyout(myproc()->pagetable, pst, (char*)(pstat), sizeof(*pstat)) < 0

where myproc() is the current calling process, pst is the passed address of the struct pointer from the user wrapper and pstat is the procstat structure that we have obtained above.