CS330A: Operating Systems

Assignment 1

Akhil Agrawal
200076

Akshat Garg
200084

Suket Raj
201013

Uttam Kumar
201071

September 29, 2022

1 Implementation of System Calls

Note: The following entries were made in different files for implementing the system calls.

- system call name entry in "usys.pl"
- system call number in "syscall.h"
- extern variable for the system call in "syscall.c"
- kernel funcion entry in "defs.h"
- user function entry in "user.h"

The descriptions provided below are mainly for the wrapper functions in "proc.c" which are called from the sys_*systemcallname* function in "sysproc.c". The wrapper function does the required task, and returns the value to corresponding syscall function which in turn stores that value in the a0 register of the calling process' trapframe. The user program picks up the returned value(if required) from the a0 register.

1.1 getppid(void)

The implementation works similar to the getpid() system call. The process control block of a process is a structure that contains various fields describing the process, including a pointer to the parents' process control block. To access the parents' PCB, we first acquire the calling process' wait_lock. Now to access the PID of the parent process from the parents' PCB, we first acquire the parents' spin_lock, and simply fetch the required PID (*myproc()->parent->pid*). Once we fetch parents PID, we release all the locks that were acquired.

1.2 yield(void)

Made the necassary entries for the sys_yield function in "sysproc.c". The sys_yield function simply called the pre-defined yield() function in "proc.c" which does the required task.

1.3 getpa(&x)

Used the walkaddr function defined in "vm.c" which takes a virtual address as an argument and returns the corresponding physical address. We added the sys_getpa in "sysproc.c" which uses the argaddr function in "syscall.c" to fetch the virtual address passed through the user program. The virtual address passed by the user is contained in the a0 register of the calling process' trapframe; the argaddr function copies that virtual address from a0 to the argument pointer. This pointer acts as the argument to the walkaddr function which returns the corresponding physical address.

1.4 forkf(f)

We implement this system call similar to the fork() call. We first allocate a new process (using *allocproc()* function), copy the virtual memory and the saved user registers from the parent to the child. We increment the reference counts of the open file descriptors, as now both child and parent will be accessing them. We assign the current process as the parent of the new process that is created to establish the parent-child relationship.
Now we have passed a **function pointer** as an argument which is to be executed in the child process. A

function pointer is an address(or pointer) pointing to the first instruction in the function to be executed. Therefore we change the *user program counter* of the child process to point to this address. This will ensure that whenever the child process is scheduled, it will start the execution from the function itself. We do not change the register storing the return address ($\$ra$) because the register already stores the correct location (the location from where the forkf(f) was called) where the child process should return to once the function execution is completed. We observe that we did not set the return value in the child process, as it will be determined by the user defined function passed as the argument.

Now, if the return value of the function is changed to some value other than 0, when we call forkf(f), it creates a new process. It returns the pid of the newly created process in the parent, but returns the *return value* of the function f passed as an arguement to the forkf() call.

Also, in this implementation, the forkf() function accepts a function pointer of type *int (*) (void)* only as an argument. Therefore if we change the functions' return type to some other type say *char*, the function pointer will be of the form *char (*) (void)*, and passing this in forkf() will cause a compilation error.

1.5 waitpid(id, addr)

The implementation is similar to wait() system call. We first fetch the integer argument(id), and the pointer passed by the user using the argaddr function. These arguments are passed to the waitpid() defined in "proc.c" and called from sys_waitpid function. The waitpid function gets the calling process PCB pointer p. It then loops over the process table. If the parent of a process is p and its PID is id, then we check. We return only if the child is in the ZOMBIE state. Before returning we have to copy the *exit status* of the child into the argument pointer passed by the user. If addr is NULL, we simply dont care about the exit status of the child, and return its PID after freeing it from the process table; otherwise, we copy the status using the *copyou*t function. If due to some reason, the copyout returns error, we return -1 indicating an error in calling waitpid(). Otherwise, after successful copyout, we return the PID of the child process after freeing it from the process table. If no matching process is found in the process table, it means the calling process has no child and hence return -1. For all other cases, we return -1.

1.6 ps(*void*)

First, we had to introduce new member variables for storing *creation time, start time,* and *execution time*. A new variable *started* is introduced to check if the process has started as the scheduler can schedule the process to *RUNNING* multiple times.
The implementation includes iterating through the process table and check if a process is *UNUSED*. For each such process it prints out the *pid, parent pid, state, creation time, start time, execution time,* and *size*. parent pid is 0 if a process doesn't have a parent.
Creation time is set in the *allocprocess()* function in *"kernel/proc.h"*. Start time is set in the *scheduler* function in the same file if the process is not yet started i.e. going to *RUNNING* state for the first time. Execution time is the difference in time between the time a process goes to *ZOMBIE* state or otherwise the current time and the start time. The *size* is printed from the *sz* member variable of *proc* structure. The return value of this system call is always set to 0.

1.7 pinfo(int, void*)

This wrapper function call reaches the sys_pinfo which classifies the first argument as -1 or a process id. If the first argument is -1, it calls the *pinfo* in the file *"kernel/proc.c"* with the first argument as the process id of current process otherwise the argument already recieved.
The $pinfo(int\ id,\ uint64\ addr)$ function is executed similar to the ps function, but instead of all the processes, it finds the process with *id* recieved as the first argument, creates a new *procstat* structure variable and fills all the details that we print in *ps* system call for every process. Then we copy this structure into the *addr* recieved as second argument byte by byte using the *copyout* function. Return -1 in case of any error or id not found otherwise return 0.