Assignment 1 : CS330
Team : 31

Kumar Arpit	Kajal Deep	Pratham Jain	Jaya Meena
200532	200483	200712	200472

September 29, 2022

Implementation of new system calls:

A. getppid
B. yield
C. getpa
D. forkf
E. waitpid
F. ps
G. pinfo

To add a new system call, we need to first assign a number to the system call, add a pointer to the function of the system call which contains the actual implementation of the system call, then define the prototype of the function and finally define the function for the system call. To do these, the following files have been modified :

- **syscall.h :** every new system call is assigned a number here as **"#define SYS_* n"**, where n is the number assigned to system call.
- **syscall.c : " [SYS_*] sys_* "** defines a pointer "sys_*" to the function of the system call while **"extern uint64 sys_*(void)"** defines the prototype of the function.
- **sysproc.c :** defines a simpler function "sys_*" which pass the arguments to the actual function "*()" and calls it from **proc.c**.
- **proc.c :** has the actual definition of the function.
- **defs.h :** the system call functions defined in proc.c are declared here also.
- **user.h :** defines the function which eventually calls the system call.
- **usys.pl :** an entry is made to define the wrapper function of the system call exposed to the users in usys.S.

- **A. getppid:** Returns the pid of the parent of the calling process. Takes no argument. If the calling process has no parent (for whatever reason), it returns -1.
  - We get the structure of the calling process using myproc().
  - After acquiring appropriate locks, we check if parent exists. If it does we assign its pid to return variable else, we assign -1.
  - We release the locks. We return.
- **B. yield:** The calling process is de-scheduled i.e., it voluntarily yields the CPU to other processes.Returns zero always. Takes no argument.
  - The sys_yield() function in sysproc.c calls yield() function from proc.c.
  - The predefined yield() function deschedules the process and we tail its return value from sys_yield().
- **C. getpa:** Takes a virtual address (i.e., a pointer) as the only arguments and returns the corresponding physical address.
  - We get the current process structure using myproc().
  - We first acces the pointer in pagetable of of the process, if error occurs we return -1.
  - Else we sum up the value and finally returns according to the mentioned formula walkaddr(p→pagetable, A) + (A & (PGSIZE - 1))
- **D. forkf:** In forkf() call, the parent behaves just like the usual fork() call, but the child first executes a function (it must not have any argument) right after returning to user mode and then returns to the code after the forkf() call. The forkf system call takes the function address as an argument.
  - forkf works almost exactly like the usual fork call. We infact copy the fork code as is, mostly.
  - The only point where it differs is that we set the epc or the program counter of the child to the passed function.
  - This works because the passed function takes no arguments.
  - After executing f, it returns to the place where a regular fork call would have returned in the child.
  - Do note that we cast the function pointer to void* so a user may try to jump to any arbitrary function though this will likely lead to an error.
  - **Asked questions on given code:** (Do note that the sleep(1) is a rather short interval which causes some of the processes to overlap while printing even if one might expect the non sleeping process to finish earlier, probably intensified because it calls another function g).
    - **return 0 -** In this case, after executing f and printing "Hello World 100" in it, the function in child program returns 0. This behaviour is similar to that of fork() function which also returns 0 to the child process. Thus, the else block is executed while the parent is sleeping and the "Child" is printed first and then "Parent" (assuming parent sleeps enough).
    - **return 1 -** In this case, after executing f and printing "Hello World 100" in it, the function in child program returns 1. This behaviour is different from that of fork() function which returns 0 to the child process. So now in both processes, the else if (x¿0) block is executed and they print almost at same time after sleeping.

    - **return -1 -** In this case, after executing f and printing "Hello World 100" in it, the function in child program returns -1. This behaviour is different from that of fork() function which returns 0 to the child process. So now in the child, the if (x¡0) block is executed (which is equivalent to a fail block for a normal fork call). Thus the child prints the error message and then aborts and then the parent wakes and prints its pid. (Again assuming parent sleeps enough, else it may print before or simultaneously).
    - **no return (void) -** Undefined behavior. Since the function does not return any value explicitly, nothing can be said about the value recieved at forkf()'s position. For our compiler, if the body of the called function was not empty it seemed to return 1 (causing else if block to be executed) and 0 otherwise (causing else block to be executed in child) but again this is an undefined behaviour by C and should not be predicted.

- **E. waitpid:** This system call takes two arguments: an integer and a pointer. The integer argument can be a pid or -1. The system call waits for the process with the passed pid to complete provided the pid is of a child of the calling process. If the first argument is -1, the system call behaves similarly to the wait system call. The normal return value of this system call is the pid of the process that it has waited on. The system call returns -1 in the case of any error.

  - If pid is -1, **wait()** is called else two proc struct, p and np are declared. p contains the current running process. A loop is run to scan through the process table looking for a process with pid as the given pid. If there is no process with the given pid or even if there is a process with the given pid but it is not the child of the current process p, then -1 is returned. Else the state of the process is checked. If it is terminated, i.e, is in the ZOMBIE state, then the exit status of the process is copied to the address provided in the argument and pid is returned. Else the parent p goes to the sleeping state waiting for the child to terminate and the loop continues until the required child terminates.

- **F. ps:** This system call walks over the process table and prints the following pieces of information for each process that has a state other than UNUSED: pid, parent pid, state, command, creation time, start time, execution time, size.

  - **proc.h** is modified to add three new fields, creattime (the time when the process is allocated), starttime (the time when the process is scheduled for the first time) and endtime (the time when the process becomes zombie) in the proc structure.
  - The for loop scans each process of the process table and prints the required information if the process is not in the UNUSED state. The ppid value is -1 if the parent of the process is not defined, else it is the pid of the parent. If the process is SLEEPING, RUNNING or is in the RUNNABLE state then the execution time is defined as the (current time - start time) else (endtime - start time).

- **G. pinfo:** It returns the information about a specific process to the calling program in a structure procstat.

  - A new structure is defined as :
    struct procstat {
    int pid; // Process ID
    int ppid; // Parent ID
    char state[8]; // Process state

char command[16]; // Process name

int ctime; // Creation time

int stime; // Start time

int etime; // Execution time

uint64 size; // Process size

};

in a new file procstat.h .This file is included in proc.c .

- The required information are accessed as in the last case of ps for the process with the given pid and stored in a procstat structure "ker". In case, there does not exist any such process, -1 is returned. Then, those informations are copied from ker to the user passed argument "pstat" using **copyout()** function and finally 0 is returned.