CS 350 Assignment 2 Design Questions Group 021

Question 1 (file descriptors):

File descriptor is a structure containing a vnode pointer "v", an integer "flg", and an off_t (64 bit integer). "os". v is the abstract representation of the file using by the file descriptor; flg indicates how to open the file, such as read-only (represented by 0) and write-only (represented by 1); os is the offset within the file that file descriptor associated with.

A new field fdt (a fd_table structure) is added to the thread structure. Each fd_table contains a pointer to an array (array structure defined by kernel) of file descriptors called "fds" and an integer "num_fd". fds contains all file descriptors of the thread, where their ID is their index in the array. Standard Input, Standard Output, and Standard Error (with fd ID 0, 1, 2) are created when a fdt is initialized. The I/O are operated through console device (con:). The fd_num keeps track of total number of fds opened to prevent from opening too many files.

Open includes a char pointer "filename", an integer "flags", an integer mode, and an integer pointer "return_val". After check for valid filename and flags, vfs_open is called to open the file/device/other kernel object named by the pathname filename in mode specified by flags, and store contents into a vnode "v". If success, vfs_open will return 0. fd_create will then be called with v, flags, and an offset of 0, which will create a file descriptor "fd". fd will then be appending to current thread's fdt's fds array through fd_table_add_fd, which returns the added fd's ID upon success. If added successfully, the fd ID will be stored at the address pointed to by return_val, and 0 will be returned to indicate a successful Open of a file. Current thread's fd_num field will increase by 1 since one more file opens. Whenever an error happens, the open function will return -1, and the error number indicating the type of error is stored in the address pointed by return_val.

Close includes an integer "fd" and an integer pointer "return_val". It first checks for valid input of fd, such as deadbeef or a fd ID bigger than the length of array fds or smaller than 0. Then fd_table_rm_fd removes the fd from current thread's fdt's fds by calling vfs_close and setting fd to null. It returns 0 on success and 1 if fail. The fdt's num_fd is decreased by 1 every time a fd is removed. Any failure will cause the close function to return -1 and store the error number in the address pointed by return_val. Otherwise, it returns 0.

Read takes an integer "fd", a void pointer "buf", a size_t buflen, and an integer pointer "return_val". Write has the same parameters except buf is a const vode pointer. Both functions check for valid address, valid fd, and makes sure the file is not read-only/write-only accordingly. An uio structure is initialized to store data read by VOP_READ from (or write by VOP_WRITE to) the given fd's vnode. If any of the above condition fail, function returns -1 and stores the error number in the address pointed to by return_val. fd's offset will be set to u's offset to indicate the number of byte read (wrote) already. If u's uio_resid equals 0, no more bytes are required to be read (write), then the number of bytes read (wrote) already (which equals to buflen) will be stored in the address return_val points to. If not read (write) completely, the amount of bytes left, calculated by (buflen-u.uio_resid), will be stored. Upon success, both function returns 0.

Question 2 (process identifiers):

For PID management, a structure called processInfo is defined.

```
        struct processInfo{
                pid_t self;             //the PID corresponding to this processInfo
                int exited;             //if the process have exited: active = 0, exited = 1, error = -1
                pid_t parent;           //the PID of its parent
                pid_t *children;                //the array that contains its children's PID
                int num_children;       //the number of children it has
                int exitcode;           //the exitcode of this process
                struct lock *plock;     //used with pcv
                struct cv *pcv;         //the parent will be sleeping on this cv if the process have not
                                        //finished yet. This process wills wakeup its parent when it
                                        //exits.
        };
```

An array of pointer to processInfo called proc_info_array is created to store the processInfo for all processes. This array is initialized to support 128 processes to save memory, but its size can be increased if there are more processes created. Each process has a PID, which can be used to access its corresponding processInfo from proc_info_array. Since we assumed there is only one thread in each process, the PID can be stored in the thread structure.

The kernel generates a PID in the "thread_create" function by calling "add_proc_info". The "add_proc_info" function first creates an empty processInfo pointer (npinfo). It then finds the next available PID (npid); this is basically finding the index of the first NULL element in proc_info_array (get_next_pid actually returns index+2 since index starts from 0, PID starts from 2). Next, it sets the "self" field (self = npid) and the "parent" field (parent = curthread->pid). Finally, it stores npinfo into array_proc_info at index = npid – 2, and returns npid.

The kernel determines that a PID (along with the corresponding processInfo) is no longer needed after the parent of the process with that PID exits.

"Fork" first creates a new thread by calling "thread_create"; the PID for this thread is assigned in "thread_create" as discussed above. Then it creates a new trapframe, and copies the current thread's trapframe into the new trapframe using "memcpy". After that the current thread's file table is duplicated. Next, a stack for the new thread is created and the current thread's stack is memory copied. Now a clone of the current thread is made, a new process is created to append the new thread to. The new process also copies the current process' address space. Finally, the new thread is switched to user mode and activated. Fork returns the PID generated in "thread_create".

Since we have stored our PID in the thread structure, "getpid" is simply returning the PID of the current thread.

In "_exit", the current thread/process grabs its processInfo, and if its processInfo is not NULL, sets the field "exited" to equal 1, and "exitcode" to equal the passed-in exitcode. It then grab the lock (plock), wakeups its parent from the conditional variable (pcv), and release the lock. It also destroys the processInfo for each of its children. Finally, it calls "thread_exit".

Question 3 (waiting fro processes):
A process is only allowed to wait for its children.

As described in question 2, a lock and a conditional variable are created for each processInfo. They are used to achieve the waiting.

In "waitpid", first the inputs are validated. Next, the current thread will find the processInfo (pinfo) corresponding to the given PID. Then it will acquire the lock (pinfo->plock) and waits on the conditional variable (pinfo->pcv) while the process with the given PID is still running (pinfo->exited == 0). When the process with the PID exits (runs _exit), it will acquire the lock, wakeup the process waiting on the cv, and then release the lock. Then the waiting thread will proceed to check if the process with PID exited successfully and stores the exitcode in the status parameter. Finally it releases the lock, stores the PID in the return value parameter and returns 0.

Question 4 (argument passing):
For "runprogram", argc and argv are placed on the kernel stack at first, in order for it to be used by user programs, they must be copied out to the user level and they must be properly aligned. First an array of address called addr is created; addr should have size argc + 1. "runprogram" then gets the length of each argv element and store a properly aligned stack address into addr base on the length. Next it will use the values in addr to copy out the arguments. Finally, it will run the user program with the pointer that point to the copied out arguments.

For "execv", the argc and argv need to be copied in from the user level to the kernel stack first, and then copied out to the user level. The steps are the similar to the steps in "runprogram".