CSC-335: Project 3 Design Report

Hope Crisafi, Tyler Nass, Zach Dubinsky

Professor Anderson

Wednesday, May 9, 2023

What are the major components that you will have to add to OS/161 and how will they interact with OS/161?

1. **Will there be one or many kernel threads? Preemptive or non-preemptive.**

There will be one kernel thread, and it will be preemptive, because in round robin scheduling, the thread will need to remove the process that is running when the burst time is up

2. **What data structures will you need to add? Be specific as possible here, perhaps even writing out the structs with fields you will be adding to or defining.**
    - We will need to add a PCB to the kernel, which contains all the information for a single process in one place, including some or all of the following:
        - PID (process identifier)
        - Open files (probably make a file struct of some kind - should be a monitor?)
        - Who the parent process is (PID? Pointer to its PCB?)
        - Children processes (PIDS? Pointers?)
        - Return value/exit code
        - Cpu burst statistics (for scheduling)
        - Synchronization mechanisms - lock on editing PCB, etc. (make PCB a monitor?)
        - Current working directory
        - Arguments to the process when it was started
    - We will also need a Process State table containing information about processes and their states.
    - For file system management, many operating systems have been known to use a B+Tree which is good for inserting, deleting, and searching for nodes. This would be the ideal data structure for a file system in OS/161.

3. **How will you handle process termination? How will this affect parent / child processes?**

When a process terminates, we will use added functions in proc.c to ensure that all aspects of process termination are handled. If we need to terminate child processes they will be marked as zombies and terminated by the scheduler. If they do not need to be terminated we will find a new parent process. The process state table of the parent and child processes will be updated accordingly.

4. **How does your implementation protect the kernel data structures?**

Just as in previous projects, we will implement synchronization mechanisms (as a form of security), such as locks and semaphores, to ensure that only the kernel thread running can modify kernel data structures. Also, we will implement defensive programming techniques, such as KASSERT statements to ensure that user processes cannot directly modify kernel data structures. If not already implemented, we can ensure that user memory or processes will not have access to kernel memory.

5. **How does your implementation protect the kernel and user processes from other processes?**

Again, by using KASSERTS and synchronization to ensure that processes have limited/restricted access to shared resources. Synchronization mechanisms will ensure that the current running process can run without fear of interruption or interleaving, and KASSERTS will ensure that no other processes can modify their data structures or access their memory.

6. **What kind of scheduling policy will you implement? Why?**

Because OS/161 has a built-in round robin scheduler, we will modify that to make it a priority-based round-robin scheduler. This is for simplicity's sake; much of the algorithm is already implemented and we would only need to add a few priority conditions. This includes modifying the thread structure to have a "priority" instance variable, and modifying all functions that handle threads (ex. fork) to account for the priority number.

Our best priority condition thus far is contingent on process aging. Place every thread on a queue and let k be a time quantum. In the beginning, every thread executes in FCFS fashion on the queue for k time, and is placed at the end of the queue for round robin functionality. When a thread executes, its priority is increased. Threads that require longer CPU bursts will eventually have high priority as they remain on the queue longer. Each thread will be allowed to run for a quantum nk, increasing linearly, where n is the priority of a thread above a certain threshold. So long as we cap n there are no starvation issues, and turnaround time is increased for threads requiring long CPU bursts.

7. **How will you manage concurrent file accesses?**

Each file will have its own set of locks. The first lock will be a reader lock, which a process can acquire if it wishes to read from a file. There can be as many reader locks given to a process at any given time, but they only give access to the file when it is not being written into. When the file needs to be written into, these locks will block other processes from reading. The second type of lock will be a writer lock. Only one process can acquire this lock at a time to prevent race conditions. When a process with the writer lock wants to write, all reading processes will be blocked out from reading until the writing is done. The writing lock prevents the holder from writing while readers are reading.

8. **How will you deal with transferring data from user space to kernel space to user space (e.g., when you are writing exec)? What built-in features of OS/161 will you use to do this?**

In src/kern/vm/copyinout.c, there are several built-in functions that handle the copying from user/kernel space. Specifically, the *copyin* and *copyout* functions can do this, and they have built-in protections. Also, in src/kern/lib/uio.c, there are data structures provided to aid data transfer to different spaces (buffers) on the byte-level.

Examine the man page for each syscall you must implement. Reflect on and discuss the following:

1. **What is this syscall supposed to accomplish?**
2. **Which of your components and OS/161's does it interface with?**
3. **What kernel data structures will it have to access and modify? Will synchronization be required to protect data structures or communicate between processes / kernel threads while performing this syscall?**
4. **What error states can occur and how will you handle them?**
5. **Will data need to be moved between user space and kernel space?**

- **getpid**
    - Returns the process id. Simple getter for a PCB struct data field.
    - Interfaces with process (proc) struct and PCB struct. May be called by the scheduler. Cannot be implemented until the PCB is implemented.
    - Need to modify proc and PCB structs. There are no synchronization issues, and the syscall does not fail because a process must be running for it to be called.
    - getpid does not fail. However, all syscalls must be atomic.
    - The user space calls to the kernel to access the PCB information. The PCB is kept hidden from the user space.

- **fork**
    - Duplicates the current process and creates a “copy” of the PCB. Returns 0 to the child process, and returns the process ID of the child to the parent process.
    - Interfaces with the process struct and the PCB. Presumably, the forked process is placed on the scheduler and will interface with this component. Interfaces with address space at the kernel level to give virtual memory to the new process.
    - This call creates concurrently executing processes, so the user should be careful of synchronization issues when using concurrency. Synchronization issues with file access should be handled in file accessors. A thread should not continue executing while forking so that its internal state is not changed, we do not allow interrupts while forking at thus a fork must be atomic.
    - We can produce an error if the current user already has too many processes, if there are too many processes on the system, or if there is not enough memory on the system to create another process. These are handled by asserting all of these conditions are false, and if they are not, the process is not created. Errors are reported via the errno syscall which is already implemented as global variables.
    - The kernel space should create a new process by copying the PCB of the caller process. The data that moves between the kernel and user space is only the return: for the caller process we return the ID of the new process; for the new process we return 0; if we cannot create a new process we return -1 and call the errno syscall.
- **execv**
    - Replaces the current executing user program with a user program, specified by a string path name. The new program begins executing on the same process, and thus the same process control block.
    - Uses the loadelf syscall to load an executable file to run as the program, and does run the program on a process. Only the address space of the process is modified, but this happens indirectly. Should be executed atomically.
    - The process data structure will be modified, but indirectly. There are no synchronization issues, but the program that is running will not continue if execv is called.
    - Errors are reported via errno if: the program specifier was formatted incorrectly; the program did not exist; the program is a directory; the program is not an executable; the program’s executable was written incorrectly; there is not enough virtual memory; the arguments are too large in size; an I/O error occurred; one of the arguments is invalid.
    - Data is moved between the user and kernel space when the address space of the process is changed to the executable file. No other data transfers occur.

- **waitpid**
    - Given an input PID, wait for the process specified by the PID to exit and return an exit status value in int* status. It also returns the PID of the process once it exits.
    - Interfaces with proc structs & PCB structs
    - Will access and modify the PST to update the fact that when this is called, the process that called it is now waiting, and that it is no longer waiting when the targeted process exits.
    - Error states:
        - EINVAL - The options argument requested invalid or unsupported options.
        - ECHILD - The pid argument named a process that was not a child of the current process.
        - ESRCH - The pid argument named a nonexistent process.
        - EFAULT - The status argument was an invalid pointer.
        - In each of these error states, the function will set the global variable errno to the correct label and return without waiting for any other processes to terminate.
    - Data will need to be moved between user space and kernel space because it needs to deal with procs.
    - We will implement it such that the only thread that may be tracking a given thread would be the thread's parent thread.
- **_exit**
    - Causes the current process to terminate. Takes an argument, int exitcode, which represents the exit code that should be sent when the process terminates. This function does not return.
    - Interfaces with the PCB to get the process to terminate and the PST to alert it that the process has been terminated.
    - It will access the PCB and PST.
    - There are no errors for _exit
    - No data is moved.
- **open**
    - This syscall "opens" a specified file by providing a file descriptor integer reference to the file. Note that the file may be anything whose I/O is supported by read, write, and close syscalls.
    - Interfaces with the filesystem, and thus the kernel since the file system is accessed by the kernel. Alters the process's file table, and the kernel's file table.
    - Calls to open must be atomic. The filesystem may need to be modified if the file does not exist: we can choose to create a new file if one with the specified file name does not exist. The process file table will be altered which must happen atomically, similarly for the kernel's file table.

    - If we encounter any errors we return -1 and set errno accordingly. The following are errors we may encounter: the device prefix of the filename does not exist; the file path contains something that is not a directory that should be a directory or does not exist; the file does not exist and we do not wish to create a new file; the file is a directory, opened for writing; the process's file table was full or too many files have been opened; the system's file table is full or too many files have been opened on the system; the process does not have permission to open files; there is not enough room to create a new file, and we want to; improper flags; an I/O error occurs; the pointer to filename is invalid.
    - The kernel needs to move the file to the user process's file table. The kernel needs to give an integer file descriptor for that file in the table.
- **read**
    - Used to read a section of length size_t buflen from file int fd. It does this by storing the next buflen bytes from the integer file descriptor, fd, based on the current seek position in the file. The seek position is advanced by the number of bytes read - i.e. calling read again immediately will pick up where the last read left off.
    - Interfaces with the file system (file control - fcntl).
    - Must be atomic. In order to not lose the correct seek position, read can only be accessed by one process at a time.
    - Errors:
        - EBADF - fd is not a valid file descriptor, or the file at fd has not been opened for reading
        - EFAULT - the address space pointed to by void* buf is partially or completely invalid
        - EIO - I/O error occurred.
    - Needs to access the data of the file, so it does move data between user and kernel space.
- **write**
    - Writes to the file fd at most buflen bytes. It writes starting at the current seek position of the file, and the data to write is passed in through parameter const void* buf.
    - Like read, write interfaces with the file system.
    - Must be atomic so the correct seek position is not lost.
    - Errors:
        - EBADF - fd is not a valid file descriptor, or the file at fd has not been opened for writing
        - EFAULT - the address space pointed to by void* buf is partially or completely invalid
        - EIO - I/O error occurred.

        - ENOSPC - there is no free space remaining in the filesystem containing the file fd.
    - Needs to access the data of the file, so it does move data between user and kernel space.
- **lseek**
    - lseek is used to manipulate the seek position of a file without reading or writing. It takes a signed value, pos, and an integer argument, whence, that tells it what to do with the signed value. It can use the signed value as an offset from the beginning of the file, as an offset from the current seek position, or as an offset from the end of the file. The function returns the new seek position.
    - It interfaces with the filesystem, like read and write.
    - Must be atomic.
    - Errors:
        - EBADF - fd is not a valid file handle.
        - ESPIPE - fd refers to an object that does not support seeking.
        - EINVAL - the whence value is invalid or the resulting seek position from the function call would be negative.
    - Needs to access the data of the file, so it does move data between user and kernel space.
- **close**
    - Closes a file, specified by an integer file descriptor. Returns 0 on success, and -1 on error.
    - Like open, it interfaces with the file system and modifies the file table(s). The file should be removed from a process's file table. The file should not be removed from the kernel's file table.
    - Errors: Note that, despite any errors in the underlying implementation, the file should still be closed if it is found.
        - EBADF - fd is not a valid file handle
        - EIO - I/O error occurred.
    - Gets rid of the information about this file from the process's file table.
- **dup2**
    - Clones one file handle (oldfd) onto another file handle (newfd) — i.e. it makes newfd refer to the same file that oldfd refers to. If the file referred to by the passed in value of newfd is already open, it is closed.
    - Interface with
    - Should be atomic, therefore synchronization is not an issue
    - Potential errors:
        - EBADF - either oldfd is not a valid file handle or newfd is a value that cannot be a valid file handle.

        - EMFILE - the proc's file table was full or a process specific limit on open files was reached, meaning the file cannot be opened.
        - ENFILE - system's file table was full or a global limit on open files was reached.
        - In the case of an error, newfd is not changed. The global variable errno is set to the appropriate error code. Then the function returns -1.
    - Data does not need to be moved between user space and kernel space. The logical address of oldfd is all that is necessary to get newfd to refer to the same file, so no interaction with the physical address is necessary. All of the necessary data is already in user space in the process table.
- **chdir**
    - Changes the current directory to a directory specified with const char* pathname for the process.
    - This will interface with the process's file table, and the kernel's file table. The addrspace of the process should be updated. The call should be atomic.
    - Errors:
        - ENOTDIR can mean a non-final component of the pathname was not a directory or it can mean more generally that the pathname does not refer to a directory
        - ENODIV means the device prefix of pathname does not exist.
        - ENOENT means the pathname itself does not exist.
        - EFAULT means the pathname is an invalid pointer,
        - EIO an I/O error occurred.
        - In the case of an error, the current directory is not changed. Global errno is set according to the appropriate error. The function returns -1.
    - Data transfers between kernel and user space is relegated to the return of the system call: 0 on success and -1 on failure. The kernel structures track the current directory of the process.
- **__getcwd**
    - Gets the name of the current working directory. Saves the current directory into char* buf, a buffer of length size_t buflen. The return value is the length of the data returned.
    - Interfaces with the addrspace.
    - Does not modify anything. Accesses the address of the current working directory so that it can produce its name.
    - Errors:
        - ENOENT - a component of the path name no longer exists
        - EFAULT - buf points to an invalid address
        - EIO - an I/O error occurred

- **Give a timeline of implementation focusing on dependencies, i.e., what components need to be implemented before other components.**

Before we do anything, we should fix the loose ends from Project 2, namely, we must change the shared buffer to an array instead of a linked list. We will also implement the data structures that are needed to successfully implement our respective syscalls. These structures are the PCB and the B+ tree for our file system.

Once the preliminary features are in place, we will implement the "simpler" syscalls and ensure their functionality with built-in tests. All goals presented up to this point are planned to be completed by the Partial Implementation due date (5/19).

By the Final Due date (6/2), we will implement the remaining syscalls and test their functionality. This extended deadline for the complete implementation will afford us the time to build confidence in our implementation efforts. We can hone our skills with smaller components of the project with an early deadline. We can move to the more complex aspects of the project when ready, but do not set unrealistic deadlines for ourselves

- **Briefly discuss how you are going to work together as a group, how you are going to manage access to the repository, and what base code you will be starting from or porting.**

We all had almost the same implementation for project 2, but we have decided to use Tyler's repository for Project 3 and 4. To avoid merge conflicts, there will have to be communication about who edits what, and when. We have mapped who is going to do what function, so no one is editing or overwriting someone else's work. Only one person will be working on a file at a time. Communication will be essential, and we have made a group chat for just that.