OS161 Design Documentation

Team members: John Daly, Diep (Emma) Vu, Eric Zhao

I. LOGISTICS

- A brief discussion of who does what:

  John:
  - `[FILESYS] open`
  - `[FILESYS] read`
  - `[FILESYS] write`
  - `[FILESYS] close`
  - `[PROCSYS] fork`
  - `Scheduling: Implementing the algorithm in schedule()`

  Eric:
  - `[PROCSYS] execv`
  - `[PROCSYS] _exit`
  - `[FILESYS] __getcwd`
  - `[FILESYS] chdir`
  - `Scheduling: Look into how we usurp the existing round robin scheduling`

  Emma:
  - `[PROCSYS] waitpid`
  - `[PROCSYS] getpid`
  - `[FILESYS] dup2`
  - `[FILESYS] lseek`
  - `Scheduling: implementation and maintenance of process related data structures (process ids)`

  Scheduling
  - Equitable Shortest Job First (preemptive) link

- Trello board to keep track of progress: https://trello.com/b/l4AIJtb7/project-3-4-dvz-team
- Communication channel: Slack
- Base code from John's Project 2 solution

- Git branch convention: `<username>-<feature>` i.e, `vud-execv` so that it is easy to keep track and convenient when merging
- Plan to work as a group: use Trello board to see the progress, communicate via Slack, work together in Pasta Lab (especially when merging and resolving conflicts). We plan to have weekly check-ins to see the progress as a team and help each other on blocks.
- How to manage access to the repo: John will have access to the main branch and for accepting a merge request. Emma and Eric will contribute by working on the features in different branches. and all 3 of us will sit together to do a merge to the main branch.

II. QUESTIONS ANSWERING

[TABLES]

Our project will use a global process table and a global file table, along with local file tables to manage files and processes.

The process table is a globally accessible table of processes that have methods for searching and obtaining a reference by pid, putting a process on the table, and getting one off the table. The table is created by a bootstrap process, and has a counter to track the number of processes available.

The file structure is a small structure that contains a vnode, a reference count, a lock, a seek position, and flags determining whether it can be written to or read from or not. It also has a local file descriptor and a global file descriptor which will be expanded on shortly. This will be the structure which we treat as a “file” for convenience and for not having to deal with the vfs directly. It can be created, destroyed, duplicated, or copied (these are different).

The global and per proc file tables are similar in how they work but different in actual function. The global file table is a table that is updated every time a new open is issued, opening the file with a new global file descriptor (global and local start out as the same). The local table is a table specific to a process and is always a subset of the global table, as some processes could have open files that others don’t have access to. When an opened process is put on the local file table, it will also be put on the global file table. The global file table can be searched by string name, but other operations are done by file descriptor. The local file descriptor is used for searching in local file tables, but the global file descriptor is used for the global table, allowing files to be universally identified in the global table, but uniquely identified in the local table.

[SCHEDULING]

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

- Will there be one or many kernel threads? Preemptive or non-preemptive.
    - There will be many preemptive kernel threads.
- 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 make additions to the PCB structure to include the necessary fields to make scheduling work. The additional fields for PCB are:
        - level of priority
        - execution time
        - Remaining time
        - A cv for waitpid
        - A pid
        - A parent pid
        - A list of pointers to child processes
        - A pointer to this process's file table
- How will you handle process termination? How will this effect parent / child processes?
    - When processes exit, if the parent is still running (and is not the kernel process) an exit code will be posted to the global process table with the parent's pid, the child's pid, and the exit code. Parents will then be alerted via their condition variable and will scan through the exit codes to find the right one (if it's waiting) or will continue to wait if it's not the right one. Essentially, an exit code in the table will signify if a process has exited or not. Children will not be stopped when a parent exits, but will not post an exit code on exit if this is the case.
- How does your implementation protect the kernel data structures?
    - Our implementation will be synchronized to avoid bad interleavings that could corrupt the kernel data structures. For the file structure structs, we will include locks on file operations to keep operations synchronized.
    - Operations involving PCBs and thread structures will be synchronized to avoid corruption.
    - All syscalls will now be synchronized by some mechanism to avoid any synchronization issues.
- How does your implementation protect the kernel and user processes from other processes?
    - Our implementation will have synchronization on its various parts to not allow processes to interact with each other in bad ways. When

it comes to the file system, the monitor will halt user processes attempting to access the kernel's file structure if it is currently locked from another user process trying to access it.
        - All syscalls will be locked to not interfere with each other, all kernel data structures that could be corrupted by interleavings will be made into monitors.
    - What kind of scheduling policy will you implement? Why?
        - We will implement a preemptive Equitable shortest job first algorithm (EQ-SJF) with a starvation prevention measure. There will be two parameters, `e` the percentage of a process's burst time to completion and the time spent by a process in the waiting queue, and `q`, the priority to a process when its waiting time exceeds `q` times the burst time. Depending on how to assign different values of `e`, the algorithm will be preemptive or non-preemptive. At `e=0`, EQ-SJF behaves as the normal shortest job first with preemption and at `e = 1` it behaves as the normal shortest job first without preemption.
        - This EQ-SJF algorithm will be able to prevent starvation by giving the long processes higher priority (`q`) when they reach a certain amount of waiting time.
    - How will you manage concurrent file accesses?
        - Opening and closing files should be done atomically with synch primitives.
        - Writing to files and reading from files will be mediated by synchronization primitives that will lock file operations.
        - Getting current working directory and changing directory will need to use synchronization primitives
        - Seeking a position in a file and writing to a file will need to use synchronization primitives
    - 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?
        - Data can be transferred between user and kernel space using the `copyin/copyout/copyinstr/copyoutstr`, which is how we'll handle all copying of data between kernel and user space.

[FILESYS] + [PROCSYS]

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

- What is this syscall supposed to accomplish?
  - `_exit:` Cause the current process to exit. The exit code `exitcode` is reported back to other process(es) via the `waitpid()` call.
  - `waitpid:` Waits for the process specified by pid to exit and return an encoded exit status in the integer pointed to by status
  - `getpid:` returns the process id of the current process.
  - `execv:` replaces the currently executing program with a newly loaded program image. This occurs within one process; the process id is unchanged.
  - `open:` opens a file or a device with a name and takes mode arguments to specify how the device or file should be opened. Returns nonnegative on success, -1 on failure, setting errno.
  - `close:` closes the file handle passed into it, returns 0 on success and -1 on failure, setting errno.
  - `read:` if file is open for reading, reads an amount of bytes from the file and stores them in a buffer passed in. Updates the seek position. Returns the amount of bytes read. Returns -1 otherwise and stores error code in errno.
  - `write:` If file is open to writing, write an amount of bytes into the file specified from a buffer passed in. Returns the number of bytes written on success, on failure returns -1 and sets errno appropriately.
  - `fork:` duplicates the process currently running. The state of the current process is copied entirely into the new process and no data is shared between them except for open file handles.
  - `lseek:` alters the current seek position of the file handle filehandle, seeking to a new position based on pos and whence.
  - `dup2:` clones the file handle `oldfd` onto the file handle `newfd`. If `newfd` names an already-open file, that file is closed.
  - `__getcwd:` This gets the name of the current working directory. This gets the name of the current working directory stored inside a pointer to a sequence of chars up to buflen.
  - `chdir`: The current directory of the current process is set to the directory named by `pathname`.
- Which of your components and OS/161's with it interface with?

  `_exit:`

- `src/kern/arch/mips/syscall/syscall.c:` add new `[PROCSYS]` syscall functions to the `switch` case
- `src/kern/include/kern/syscall.h:` header file for defining SYS_exit
- `src/kern/include/kern/wait.h`: header file for defining __WEXITED
- `src/build/userland/lib/libc/syscalls.S`:for loading the syscall number into v0, the register the kernel expects to find it in, and jump to the shared syscall code: SYSCALL(_exit, 3)
- `src/userland/include/unistd.h:` contains the user-level system call interfaces.
- `src/userland/bin/sh/sh.c: exitinfo_exit, cmd_exit, docommand` method
- `src/userland/lib/libc/stdlib/exit.c:` C standard function for exit process
- `src/kern/include/syscall.h:` add more syscall after implementing kernel syscall under prototypes for IN-KERNEL entry points

`waitpid:`

- `src/userland/bin/sh/sh.c:` dowait method
- `src/userland/lib/libc/stdlib/system.c:` system method

`execv:`

- `src/kern/syscall.h:` header file for defining SYS_execv
- `src/userland/include/unistd.h:` contains the user-level system call interfaces.
- `src/kern/syscall/runprogram.c:` reference when implementing the execv() system call. This will allow run programs to take arguments from the menu after implementing `execv`.
- `src/userland/lib/libc/stdlib/system.c:` use in system method to execute child process
- `userland/lib/crt0/mips/crt0.s:` code for starting up a user program. When the kernel starts running a process it passes control here.

getpid:

- `src/userland/bin/tac/tac.c` openfiles method
- `src/userland/lib/hostcompat/hostcompat.c` hostcompat_stop method

lseek:

- `src/kern/include/kern/seek.h:` shared in `libc` between `<fcntl.h>` and `<unistd.h>` and thus get their own file.
- `src/kern/include/kern/syscall.h:` header file for defining SYS_lseek
- `src/userland/include/unistd.h:` contains the user-level system call interfaces.
- `src/userland/sbin/mksfs/disk.c:` writing and reading a block disk on user space

open:

- This syscall primarily interfaces with the VFS subsystem to open files.

close:

- This syscall also interfaces with the VFS subsystem to close file handles.

read:

- This syscall will primarily interface with the VFS subsystem for the actual writes, but will also interface with the uio subsystem located in lib/.

write:

- This syscall will primarily interface with the VFS subsystem but will also interface with the uio subsystem in lib/.

fork:

- This syscall will interact with the thread subsystem for forking a thread, and the proc subsystem for working with processes.

`__getcwd:`

- This syscall will interact with os161's file directory structure.
- Only the C library can call this

`chdir:`

- This syscall will interact with os161's file directory structure.

`dup2:`

- `src/kern/include/kern/syscall.h:` header file for defining SYS_dup2
- `src/userland/include/unistd.h:` contains the user-level system call interfaces.

- 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 with performing this syscall?
  - `_exit: _exit` and `waitpid` are connected to the implementation and might have to use synchronization when designing `_exit` and `waitpid` (Synchronization is needed in case something needs to update during the exit step of a thread. Interleaving during this step could cause issues when it updates.).
  - `waitpid:` it will need to access threads by their ID. If the child process is blocked waiting for a resource held by the parent process, and the parent process is blocked waiting for the child process to exit, a deadlock can occur so synchronization is needed.
  - `execv`: this process will need to access the kernel's function pertaining to program management in order to replace the current program with a newly loaded program. Synchronization will be needed to prevent two execv from running concurrently causing interleaving during the loading of a new process.
  - `getpid:` This process will need to access threads by their ID. Synchronization is needed to protect the data structure because a different thread might be replaced than the intended thread when this method gets called.
  - `open:` This syscall will have to work with the mode_t data type and the vnode struct to accomplish opening files. This will need to

be protected by synchronization because there is a parameter that causes this function to add a file if it doesn't exist, and this could cause a race condition if not synchronized. This will also have to work with the proc structure in adding an open handle to the process's open files.

- `close:` This syscall will have to work with a vnode data structure in removing a reference, and will have to work with the proc data structure to remove an open handle to the file. It will require synchronization to make sure the open handle is added to the process's list of open files atomically.
- `read:` This syscall will have to work with the vnode and uio data structures to achieve reading. It will need to be synchronized to make sure things aren't written to the file while it's being read from.
- `write:` This syscall will need to interface with the vnode and uio data structures to achieve writing. It needs to be synchronized to make sure no two files are written to simultaneously or read from while it's writing.
- `fork:` This syscall will need to interface with the thread and process structures to achieve process duplication and thread handoff. It will also need to interface with address spaces and trapframes directly to transfer the contents of registers and address spaces. The child thread shouldn't run until the parent thread is done forking so this will probably require synchronization.
- `__getcwd:` This syscall will need to access the file directory structure in order to get the current directory. Along with that, there needs to be a synchronization mechanism in place to prevent any changes to file directory structure before it returns. However these synchronization methods could be complicated due to the need to lock a tree structure both up and down.
- `chdir:` This syscall will need to access the file directory structure in order change to the desired directory. There needs to be a synchronization primitives between changing directory and getting the current directory
- `dup2:` This syscall needs to access file directory structure in order to create a duplicated reference to the file. There needs to be synchronization primitives between duplicating the reference file and writing to modify the file
- `lseek:`This syscall needs to access file directory structure in order to look at a specific section of the file. There needs to be

synchronization primitives for seeking at a section of the file and writing in the file.

- What error states can occur and how will you handle them?
  - `_exit`: No error states will be returned since `_exit` is a void function but could raise panic if the `exitcode` is invalid
  - `waitpid`:
    - EINVAL: If the option isn't in the wait pid struct, I will print an EINVAL error
    - ECHILD: If the pid is not in the child of the current process, then I will print ECHILD and return
    - ESRCH: If the pid argument is not in the process I will print out ESEARCH and return
    - EFAULT: If the status argument isn't valid, I will print out EFAULT and return
  - `execv`: On success, `execv` does not return; instead, the new program begins executing. On failure, `execv` returns -1, and sets errno to a suitable error code for the error condition encountered.
    - ENODEV: The device prefix of *program* did not exist.
    - ENOTDIR: A non-final component of *program* was not a directory.
    - ENOENT: *program* did not exist.
    - EISDIR: *program* is a directory.
    - ENOEXEC: *program* is not in a recognizable executable file format, was for the wrong platform, or contained invalid fields.
    - ENOMEM: Insufficient virtual memory is available.
    - E2BIG: The total size of the argument strings exceeeds `ARG_MAX`.
    - EIOA: hard I/O error occurred.
    - EFAULT: One of the arguments is an invalid pointer.
  - `getpid`: This process does not fail
  - `open`: This syscall has the following error states:
    - ENODEV: device or file did not exist. Stop and set errno, return -1.
    - ENOTDIR: file/device name was not a directory. Stop and set errno, return -1.
    - ENOENT: file/device name not found, O_CREAT not set. Stop and set errno, return -1.
    - EEXIST: file already exists, user set O_EXCL. Stop and set errno, return -1.

        - EISDIR: file is a directory but was opened for writing. Stop and set errno, return -1.
        - EMFILE: process has too many open files. Stop and set errno, return -1.
        - ENFILE: system limit on open files was reached. Stop and set errno, return -1.
        - ENXIO: object was a device with no fs. Stop and set errno, return -1.
        - ENOSPC: fs was full and tried to create a file in it. Stop and set errno, return -1.
        - EINVAL: flags not set correctly. Stop and set errno, return -1.
        - EIO: I/O error occurred. Stop and set errno, return -1.
        - EFAULT: filename was invalid. Stop and set errno, return -1.
    - `close`: This syscall has the following error states:
        - EBADF: file to close not valid. Stop and set errno, return -1.
        - EIO: I/O error occurred. Stop and set errno, return -1.
    - `read`: This syscall has the following error states:
        - EBADF: file to read from was not valid, or was not open to reading. Stop and set errno, return -1.
        - EFAULT: section of address space to read from was invalid. Stop and set errno, return -1.
        - EIO: I/O error occurred. Stop and set errno, return -1.
    - `write`: This syscall has the following error states:
        - EBADF: file to write to was not valid, or was not open to writing. Stop and set errno, return -1.
        - EFAULT: the section of the address space to write to was invalid. Stop and set errno, return -1.
        - ENOSPC: no free space remaining to write to in the file. Stop and set errno, return -1.
        - EIO: I/O error occurred. Stop and set errno, return -1.
    - `fork`: This syscall has the following error states:
        - EMPROC: current user reached the limit for number of processes. Set errno and return once with the value -1.
        - ENPROC: system wide limit on number of processes was reached. Set errno and return once with the value -1.
        - ENOMEM: not enough memory for new process. Set errno and return once with the value -1.
    - `lseek`: On success, lseek returns the new position. On error, -1 is returned, and errno is set according to the error encountered.
        - EBADF: fd is not a valid file handle.

        - ESPIPE: fd refers to an object which does not support seeking.
        - EINVAL: whence is invalid.
        - EINVAL: The resulting seek position would be negative.
    - `dup2:` dup2 returns newfd. On error, -1 is returned, and errno is set according to the error encountered.
        - EBADF: oldfd is not a valid file handle, or newfd is a value that cannot be a valid file handle.
        - EMFILE: The process's file table was full, or a process-specific limit on open files was reached.
        - ENFILE: The system's file table was full, if such a thing is possible, or a global limit on open files was reached.
    - `__getcwd:` This syscall has the following error states:
        - ENOENT: occurs if the component of the pathname no longer exists
        - EIO: occurs when an I/O error happens
        - EFAULT: occurs if the buf argument points to an invalid address
    - `__chdir:` This syscall has the following error states:
        - ENODEV: occurs if the device prefix of pathname argument did not exist
        - ENOTDIR: occurs when the a non-final component of the pathname argument is not a directory OR if the pathname does not refer to a directory
        - ENOENT: occurs if the component of the pathname no longer exists
        - EIO: occurs when an I/O error happens
        - EFAULT: occurs if the buf argument points to an invalid address
- Will data need to be moved between user space and kernel space?
    - `_exit:` This syscall doesn't need data between kernel and user space since it is not taking any arguments from user space. The exitcode argument is a parameter passed from the kernel, not from user space.
    - `waitpid:` This syscall needs to move between kernel space and user space since the function needs to copy the exit status of the child process from kernel space to user space
    - `getpid:` This syscall needs to move between kernel and user space since he function needs to return the PID of the current

process to the calling user program, which requires copying data from kernel space to user space.
    - `execv`: This process will need to move the specified program from the user to the kernel in order for it to replace a different process and run.
    - `open`: This syscall does not need to move data between user and address space, can create blank files in user space.
    - `close`: This syscall does not need to move data between user and kernel space.
    - `read`: This syscall needs to move data from user space to kernel space so it can read the contents.
    - `write`: This syscall must move data from kernel space to user space so it can write to files in user space.
    - `fork`: This syscall does not need to move data between kernel and user space
    - `dup2`: This doesn't require to move data between kernel space and userspace since it only takes integer arguments for file descriptors and returns an integer output. All the necessary information is already available in kernel space
    - `__getcwd`: This syscall needs to move the name of the directory from the kernel to the user. This is related to the getcurrdir function inside VFS.h
    - `chdir`: This syscall needs to move the newly specified directory from kernel space to the user. This is related to the vfs_chdir function inside VFS.h
    - `lseek`: This syscall doesn't need to move data between the kernel and userspace since it needs to seek for a specific section of code in the file so it is solely within the kernel space and it doesn't require any data to be copied out to the user space

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

  The first things that need to be done are `_exit` and a preliminary version of `write` so that we can use the palindrome test to make sure we understand the process. Beyond this point, the first stage of development is that all syscalls dubbed FILESYS will be handled because the VFS system is a simpler starting point to get things moving. Additionally, the

process structure will be edited to try and support the next stage. Stage 2 will be implementing PROCSYS syscall, and this will be done so we can more easily have actual tests for stage 3, which will be implementing the rest of the scheduling and completing the project.

We also plan to have a global process id table that stores the current state of the process, number of process, the process id in order to manage processes concurrently. We also plan to have a file open table that keeps track of current files are opened/modified by the process and it would need lock to avoid synchronization problems.

Once we have the global process id table, we have everything we need to work on the Equitable Shortest Job First scheduling algorithm. Using the table, we can keep track of data that helps determine what processes have yet to start/finish.

III. BUGS / UNIMPLEMENTED FEATURES / TESTING / CHANGES

[KNOWN BUGS AS OF PROJECT TURNED IN FRIDAY MAY 19]

- Fork displays strange results and crashes
- Lseek hangs deep into testing in badcall and in bigseek
- Open, read, write, close do not handle error conditions correctly
- Files do not decrement vfs ref counts on destroy
- Getcwd displays strange results
- Shell runs for a while but can crash randomly

[UNIMPLEMENTED FEATURES AS OF FRIDAY MAY 19]

- remove() syscall - not strictly necessary but shows up in many tests and is annoying
- waitpid()
- execv()
- Duplication function for files, only has copy

[TESTING]

- Testing for read/write/open/close was done using opentest, closetest, readwritetest, filetest, etc in the testbin folder
- Testing for fork is done with forktest and forkbomb
- Testing for lseek and dup2 is done with bigseek and redirect
- Testing for getcwd and chdir is done by starting the shell

[REASONS FOR CHANGES]

Our file and proc tables didn't change that much except for bug fixes, but there were significant changes to the file struct. The global and local fd was added to make dup2 possible and the seek position was changed to a pointer so most of the copy code could be reused with the difference of sharing pointers or copying the value, the option was necessary. Also, the flags for readable/writable was added to files to make this status easily checkable.

Another thing that changed was our policy on parent processes terminating. We decided that it would be best to let children continue to run, but would not post an exit code. This makes sense to us because child processes could be important to run on their own, but when the parent is terminated there's nobody to read the exit codes so it should just not post an exit code. This was changed because it just made more sense to us to do it like this.

One more thing we changed is our policy that all syscalls should be locked regardless of the call. This arose out of just wanting to be absolutely sure everything was synchronized and we wouldn't get strange issues.