Design Document for Assignment 2

OS 451
Sandra Anderson, Riley Wilk, John Buscher

Overview/Project Breakdown:

Overall Goals of the Assignment: The overarching goal of this assignment is to make our version of OS161 able to support running programs. To accomplish this, a program identifier and process identifier systems will need to be implemented. The file descriptor system will be able to support opening and closing files, as well writing and reading to and from files. The process identifier system will be able to allocate pid's to threads, reclaim pid's after a process exits, locate control blocks based on the pid, and identify the pid of the current thread.

Identifying processes:

Process management requires unique allocation of process ids and a table mapping process ids to process control information. We can use a B-Tree structure to keep track of pids currently in use, which makes it easy to efficiently assign a new pid to a newly forked process, and allows us to easily find that process using its pid and reclaim the memory once a process exits.

File descriptors:

File descriptor management is the central component of the open(), read(), write(), and close() system calls. File descriptors and file control information can be stored using a system of two hashtables (each with a corresponding reader/writer lock set); a system-wide hashtable mapping from absolute filepath to uio (along with some metadata) and a process-specific hashtable mapping from file descriptor to a file control block including pointers to the associated vnode and uio. This allows the current status of the file to be accessible simply with the filepath, as in the initial open() call, and to be accessible with the file descriptor for subsequent calls.

fork (John/Sandra):

The idea of fork is to create a new thread, running in a new process, which returns with the same process control metadata as the parent thread (except for the pid). We will need to allocate a new pid, copy in the ids of open files, create a new address space and copy over the memory values associated with the old address space, set up the new thread to correctly return to the same instruction as the old thread will, and then return the new pid in the old thread, and 0 in the new thread.

execv :
execv is the function responsible to actually run the user programs with arguments. The key here is to safetly handle arguments being passed in by the user, and properly forward them onto the program that the user wants to run.

waitpid/exit (Sandra):

Waitpid/Exit are the two syscalls responsible for synchronization between processes, similar to fork/join in kernel threads. They need to implement the same conditional control flow, such that wait suspends execution if and only if the child is still running, exit leaves the process in existence if and only if the process can still be waited for, and all memory is eventually freed.

Other system calls:

I don't think we need to implement any other system calls than what is listed here.

Synchronization issues:

In-Depth Analysis and Implementation:

execv:

The implementation of execv will be very similar to the implementation of runprogram. The key difference is that execv will be able to handle arguments passed into the program. Block(s) of memory will have to be allocated in the kernel to act as a buffer for the arguments passed into execv, and be copied into the kernel using the given copyin function.

In order to handle the memory demands of execv, we will pre-allocate a block of memory and hold it in kernel space indefinitely, using locks to make sure that only one thread calling execv used that block at a time. If multiple threads were to use that block, the kernel could run out memory.

Similarly to runprogram, and address space will need to be initialized for the new program, and the process. After switching the current thread to this new address space, all of the arguments will need to be copied out of the kernel, using the function copyout.

Note, Copyin/copyout implements a memory check that verifies that the originating process specified a valid pointer, and will copy null-terminated strings correctly.

Finally, after all arguments are hopefully successfully copied into the new address space, execv will call enter_new_process() to switch to usermode

Another note, if at any point during execv something fails (such as a malloc), all open files, and allocated memory will need to be closed and free before, and the lock will need to be released returning from the function with an error.

Some of the error cases are listed below:
- The arguments passed into the program exceeds the size of the memory block in the kernel, E2BIG is returned
- If there are failures loading the ELF, the error from loadELF function is returned, (ENOENT, ENOTDIR, ENOEXEC)
- If any mallocs fail in execv, ENOMEM is returned
- If copyin or copyout fails, the error from that function will be returned.

File descriptor management

A file descriptor of -1 indicates a NULL file, consistent with the return value of open in the error case.

A file control block needs to contain the following information at minimum:
1. The associated vnode
2. The current offset of the file for this file handle
3. The access permissions for this file handle

open()

The open system call opens the object specified by the *filename* argument in a manner consistent with the *flags* argument. More precisely, the call should follow this general process:

1. Call vfs_open with the given arguments, resulting in a file descriptor and the location of a suitable vnode.
2. Initialize a file control block for this descriptor, using the filepath, flags, and vnode.
3. Return the file descriptor.

read()

The read system call attempts to read a number of bytes specified in the *buflength* argument from the current seek position of the file specified by the *fd* argument into the buffer specified by the *buf* argument. The specific procedure taken can be outlined as follows:

1. Look up the file descriptor in the descriptor-to-control table.

    a. If the result is NULL, return -1 with the EBADF error code.
2. Initialize a local uio to read from the file to the specified buffer.
3. Call vfs_read with the uio and vnode from the file control block.
4. Return the number of bytes read.

write()

The write system call attempts to write a number of bytes specified in the *buflength* argument from the buffer specified by the *buf* argument to the current seek position of the file specified by the *fd* argument (or the end of the file if it was opened in APPEND mode). The specific procedure taken can be outlined as follows:

1. Look up the file descriptor in the descriptor-to-control table.
    a. If the result is NULL, return -1 with the EBADF error code.
2. Initialize a local uio to write to the file from the specified buffer.
3. Call vfs_write with the uio and vnode from the file control block.
4. Return the number of bytes written.

close()

The close system call closes a particular process's access to the file corresponding to the *fd* argument. Its process is as follows:

1. Look up the given descriptor in the descriptor-to-control table.
2. Call vfs_close with the vnode from the file control block.

Process Management:

We need to be able to do the following things:
1. Identify the pid of a currently running thread
2. Given a pid, locate the control block of the process it represents
3. Allocate a pid that is not currently used
4. Reclaim a pid after the associated process exits

I suggest a B-Tree like directory structure for maintaining the (pid, &PCB) pairs, as it is easier to flexibly allocate than an array and has less memory overhead than a binary tree or linked list. We can balance the tree when assigning pids, and since all PCB pointers are stored at the leaves, garbage collection can proceed without disturbing the existing tree structure. If all processes on a given tree-leaf exit, then we can collect the leaf and re-allocate it whenever we have more processes.

We can synchronize the whole directory with a mutex held at the root node, or implement finer grained locks if time permits. Fork, getpid, waitpid, and exit are the syscalls which would need to acquire and release this lock, because they are the ones which access the tree. Execv should only need to access the PCB of the currently running file, and the same with other syscalls.

I suggest we reserve pid = 0 or pid = -1 as a magic NULL value.

A PCB has to contain the following information, at minimum:

1. The pid of this process
2. The pid of the parent process, if any
3. The pids of all child processes, if any
4. Identifiers (pointers?) for all threads that belong to that process
5. The pid of the process that this process is waiting on, if any
6. Information about files that this process has open, if any (?)
7. The exit code for this process, if any
8. Locks/cv to synchronize wait/exit cleanly

Waitpid/Exit:

Waitpid, like thread join, has two cases: if the child process has exited or not. If the child has already exited, then waitpid should release the remaining memory holding the child's exit information, and collect the child's pid for re-use. If the child has not exited, then waitpid should suspend all threads in the process calling waitpid, and set process control information to indicate that it is waiting. We need to validate that the *status* parameter is valid before waiting, because an invalid pointer causes the waitpid call to fail. The other errors are: options not valid, process does not exist, process is not a child; all of them are readily detectable and will be implemented to fail without waiting.

Exit, likewise, has three cases; if it is currently being waited for, if it could be waited for in the future, or if the parent has already exited, so it can't be waited for. If it is currently being waited for, then it needs to wake up all of the threads in the parent process, and pass along the exit value to be returned by waitpid, and then clean up all of its control memory. If it can't be waited for, then there is no need to save the exit value, and all of its control memory can be freed immediately. If it could be waited for in the future, then enough control memory has to be saved in order to maintain that the process and pid exist, as well as the identity of its parent and its exit value.

Waitpid():

1. Lock the process directory as writer
2. Check that the parameters are valid (status is a valid pointer, process exists and is a child)
3. If the child process has not exited:
    a. Suspend all other threads associated with this process
    b. Indicate which process we are waiting on in the process control block
    c. Atomically release process directory lock and relinquish processor, using CV

4. Return the value that the child process placed in our PCB

Exit(void):

Exit has no parameters, cannot return, and cannot return an error value.

1. Lock the file table as writer
2. Lock the process directory as writer
3. Halt all other threads of the process, forcing them to exit and freeing their control blocks and stacks
4. Detach from all open files
5. Free all child processes from the obligation of being waited for
    a. If the child is already exited, free its pid and control memory
6. If there is a process waiting for us, hand-off exit value and broadcast to wake all of its threads
7. If a process could wait for us in the future, save exit value in our own PCB
8. Otherwise, free our pid and control memory
9. Release the file table and process directory locks
10. Kill the currently running thread (the last thread that is associated with this process)

For synchronization, plan A is to lock the entire process directory tree with a single reader/writer lock, for which exit would always need a writer lock and waitpid sometimes would, and mandate that every system call which needs both a file table lock and a process directory lock grabs the process directory lock first.

Plan B, slightly more nuanced, would allow per-process locks, but require that deadlock never happen by locking in dictionary order. This would have to be carefully managed in concert with the file descriptor management, because exit has to detach any open files, and file close has to detach all processes with a claim to the file. In addition, a lock on the directory would still be necessary, because fork and exit can alter the structure of the directory, but many other system calls need to look up a process by its pid, which requires searching, therefore reading, the directory.

Fork:

The key points of fork are the baseline thread-fork, the pid allocation, and the creation of a new address space with the old values copied over.

Since we are not using thread_join directly, thread_fork is provided by OS161.

Pid allocation is managed by locking the process directory and selecting a new pid in a portion of the pid space which keeps the directory relatively balanced. Then we need to allocate a new PCB and copy over the file handles of the old PCB, as well as initializing the other values correctly (including the pid of the parent). We also need to add the new pid to the parent's list of children.

We also need to add the new process to the reference counter or pid manager of the file handle for all open files.

We can create a new address space using as_create. We specifically *don't* want to sanitize the memory contents, but copy them directly from the parent using memcpy. We already know that we are copying the entire virtual memory from the parent to the child.

At the end, we use thread_fork to place the new process in a newly running thread, with a correctly set entrypoint < this is likely to be tricky > and use the pids kept in each respective TCB to figure out which is the parent and which the child, in order to return different values.

Risk Analysis:

We are dividing risk into a best case, average case, and worst case scenario. Best case assumes minimal to no bugs or design flaws, given that the surrounding code is well-understood from the design and code reading process. Average case assumes several, but not overwhelming, bugs, taking about 2-3x of the best-case estimate, or up to 4x for code with intricate synchronization. Worst case indicates finding a fatal design flaw after several rounds of debugging, which pushes up the time to more like 3x the average case, or 8-10x the best case. However, in reality time spent debugging follows something like the tail of an exponential distribution, so no absolute bound can be placed on the worst case. We are aiming for a 98% or 99% confidence interval.

Time spent writing and running tests is included.

Process management, including fork, waitpid, and exit (Sandra):

Process directory tree, including bootstrap initialization:
Best case: 3 hours
Average case: 8 hours
Worst case: 20 hours

Fork:
Best case: 6 hours
Average case: 20 hours
Worst case: 60 hours

Waitpid:

Best case: 4 hours
Average case: 12 hours
Worst case: 40 hours

Exit:
Best case: 6 hours
Average case: 20 hours
Worst case: 60 hours

Total (for Sandra):
Best case: 19 hours
Average case: 60 hours
Worst case: 180 hours

I will be implementing the pid directory data structure first, followed in order by fork and exit. Waitpid requires correct behaviour by fork and exit, so even though it is simpler it needs to be delayed. The parts of fork which deal with virtual memory and instruction loading might make more sense as John's responsibility (who is implementing execv), and we will see at the time who is busier.

If time becomes an issue, I intend to postpone correct management of open files on fork and exit for last. Those are likely to be the hardest points to test and the most likely to have deadlock or race conditions.

Execv, potentially fork (John):

Execv:
Best case: 12 hours
Average case: 40 hours
Worst case: 120 hours

Fork (see above):

Fork will be implemented after execv if the amount of time taken for execv is closer to the best case scenario than it is to the worst case. A lot of time will also be spent writing thorough tests for execv, as there are a lot of edge cases that need to be tested for and handled correctly.

File descriptor management (Riley):

Open:
Best case: 4 hours
Average case: 12 hours
Worst case: 40 hours

Read:
Best case: 3 hours
Average case: 9 hours
Worst case: 30 hours

Write:
Best case: 3 hours
Average case: 9 hours
Worst case: 30 hours

Close:
Best case: 2 hours
Average case: 8 hours
Worst case: 20 hours

Open will be written first, as it includes the setup of the necessary structures and is likely the most complex, so the time taken should be indicative of how long writing the other calls will take. Close will be written next, for testing convenience, followed by read and write. In each case, the implementation of the actual call should be fairly straightforward; the bulk of the time will come from testing. If time is running short, fewer tests will likely be written for read and write such that they function appropriately in most scenarios but possibly cannot handle some complex scenarios.

Code Reading Questions:

Question 1: What are the ELF magic numbers?

An ELF magic number is the 16 byte header for an ELF file. It describes the kind of file it is, as well as details about the executable, such as what CPU architecture the file is for.

Question 2: What is the difference between UIO_USERISPACE and UIO_USERSPACE? When should one use UIO_SYSSPACE instead?

UIO_USERISPACE refers to the user's process code, and is executable
UIO_USERSPACE refers to the user's process data
UIO_SYSSPACE should be used when writing/reading to/from kernel space

Question 3: Why can the struct uio that is used to read in a segment be allocated on the stack in load_segment() (i.e., where does the memory read actually go)?

The uio struct contains a pointer to the actual address for the segment to read. There’s no problem with having the pointer to the actual memory segment on the stack.

Question 4: In runprogram(), why is it important to call vfs_close() before going to usermode?

If a different process needs to access that file, vfs_close must be called on it in order for that to happen.

Question 5: What function forces the processor to switch into usermode? Is this function machine dependent?

The function ‘enter_new_process’ switches the processor to switch usermode. That calls mips_usermode which calls asm_usermode, which is not machine dependent.

Question 6: In what file are copyin and copyout defined? memmove? Why can't copyin and copyout be implemented as simply as memmove?

copyin and copyout are defined in kern/vm/copyinout.c
memmove is implemented in common/string/memmove.c

copyin and copyout are copying data from userland to the kernel or vice versa, as a result, they have to be more careful in what is exposed to the between the two modes. memmove just copies from the same mode.

Question 7: What (briefly) is the purpose of userptr_t?

userptr_t is used in copyin to represent the source address that the user wants to copy into the kernel, and in copy out as the destination address to copy to from the kernel.

Question 8: What is the numerical value of the exception code for a MIPS system call?

The numerical value is found in trapframe.h, EX_SYS = 8

Question 9: Why do you "probably want to change" the implementation of kill_curthread() for this assignment?

A user should not be able to cause the kernel to panic by forcing kill_curthread() to call the panic code. We’ll likely want to just use our __exit syscall to kill the current thread once __exit is implemented

Question 10: What would be required to implement a system call that took more than four arguments?

Arguments past the first 4 would need to be retrieved from the user stack at sp+16

Question 11: What is the purpose of the SYSCALL macro?

The SYSCALL macro is a system call dispatcher that takes the number of the syscall, stores it in the v0 register, then jumps to the shared system call code.

Question 12: What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)

Under the __syscall label in syscalls-mips.S at line 84 there is the call to the function syscall

Question 13: After reading syscalls-mips.S and syscall.c, you should be prepared to answer the following question: OS/161 supports 64-bit values; lseek() takes and returns a 64-bit offset value. Thus, lseek() takes a 32-bit file handle (arg0), a 64-bit offset (arg1), a 32-bit whence (arg3), and needs to return a 64-bit offset value. In void syscall(struct trapframe *tf) where will you find each of the three arguments (in which registers) and how will you return the 64-bit offset?

The file handle will be stored in register a0, the offset will be stored in the registers a2/a3, and the whence arg will be stored on the user level stack at sp+16.
The 64 bit return value will be stored in v0 and v1.

NAMING CONVENTION:

```
function_names()
variable_names
```

```
This is to be the most consistent with existing OS161 code,
while still ensuring best readability. Functions and variables
will not be prefixed with type information, although OS161 does
so in places.
```