1. File descriptors are stored in the kernel as an *fd* structure containing the following:

```
struct fd {
    char *name; // The file's name (for debugging)
    int flags; // The read-write, read-only, or write-only flag that the file was opened with
    off_t position; // The position in the file
    struct vnode *node; // The actual vnode used for operations on the file
};
```

   Each process has a file table, which is just an array of pointers to *fd* structures where NULL entries closed/unused entries and non-NULL entries are open files.

   Open searches through the file table to find the first NULL entry and when it finds it, it created an *fd* in that entry and opens the *vnode*. If it can't find a space, then it returns an error.

   Close checks that the given file descriptor is valid and open. Then it closes the *vnode*, frees the memory used by the *fd*, and then sets the corresponding file table entry to NULL.

   Read and write both first check that the given *fd* is valid and properly readable/writeable, depending on the operation. Then, it checks that the input/output buffer is valid and prepares a *UIO* to contain the operation on the *vnode*. Then it fires off the actual VOP_READ or VOP_WRITE call, cleans up resources, and returns.

2. Processes are implemented as a structure, containing the PID of the current process and the parent process. It also contains a pointer to the process's thread, an indicator for whether the process is finished or not, the exit code of the process, a lock and cv for waiting on the process and file table stuff.

   PIDs are generated by scanning for NULL elements in an array of process structures (the array is called runningprocesses). The index of the first NULL is used as the PID so it is easy to find its corresponding process structure in the array.

   A process is officially finished when it has exited and waitpid has been called on its PID. How it should be done is to keep track of whether its parent process has finished. If the parent is already finished, it means that no other process can wait on it and therefore we don't need to remember its exit code any more. When a process finishes it should alert all its child processes (that it hasn't already waited on) that it is finished so they know to clean themselves up when they finish.

   All access to the runningprocesses array is synchronized with a global lock (called process_lock). This ensures that we can`t remove a process while data from it is being used by the kernel. When creating a new process it also prevents two threads from finding and using the same PID.

   fork - Fork copies the address space and trapframe for the current process, then finds a PID for the new process and allocates a new process structure for it. It then creates a new

thread, passing pointers to the new address space and trapframe to the new thread. The new thread starts on the child_fork function, which increments the epc of the passed in trapframe, activates the address space and calls mips_usermode to continue execution.

getpid - This was very simple, because the thread data structure was modified to keep track of the PID it belongs to. All that had to be done was to return this value.

_exit - Exit sets the value of p_finished to true indicating that the process is complete, sets p_exitcode (the exit code for the process), and broadcasts on the condition variable that is used to wake up threads waiting on the process. After all this is done it calls thread_exit to end the thread of execution for the process.

3. When waitpid is called the kernel first checks to make sure the given PID is valid (within bounds of runningprocesses array and corresponding to a non-NULL process) and then whether the corresponding process is a child of the current process. It then uses a lock and condition variable belonging to the child process in order to wait until the process's p_finished flag has been switched to true.

   The lock covers the exit code for the child process and the waiting part of current process. It does this to make sure that the exit code and finish flag have been appropriately set before the parent process can check the exit code.

   The condition variable is used so the parent process can be put to sleep until the child process is finished. Exit broadcasts on the condition variable, thereby waking up all processes waiting on the child process.

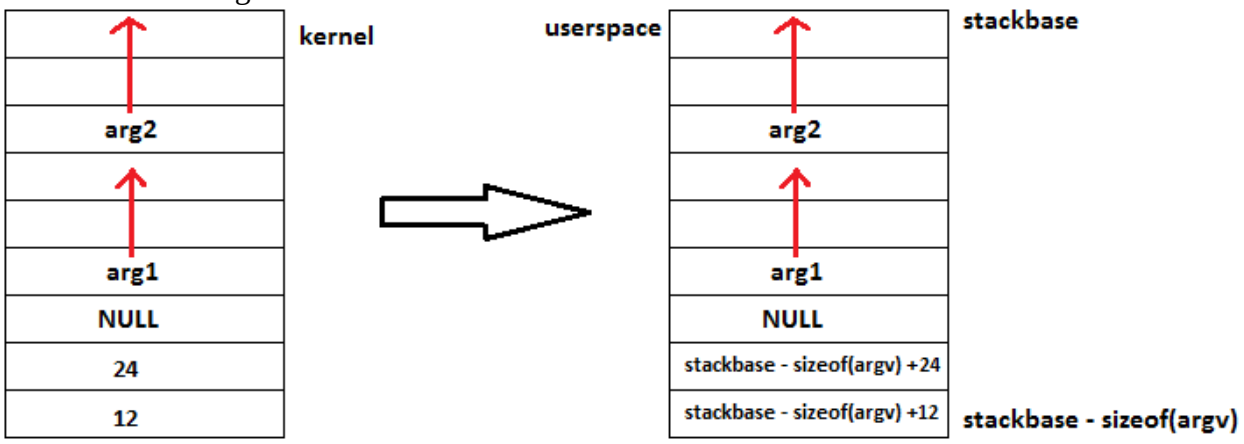
4. Execv and runprogram differed, in that when runprogram is called there is no address space. This makes their implementation of argument passing slightly different, because the location of the argument pointer is in userspace in execv and kernelspace in runprogram. However, the following idea for passing the arguments its used for both.

   Getting argc is trivial, because the passed in pointer for argv is a NULL terminated array. All that has to be done is count the number of elements in argv before NULL is reached, which can be done with a simple for or while loop. The interesting part is copying argv.

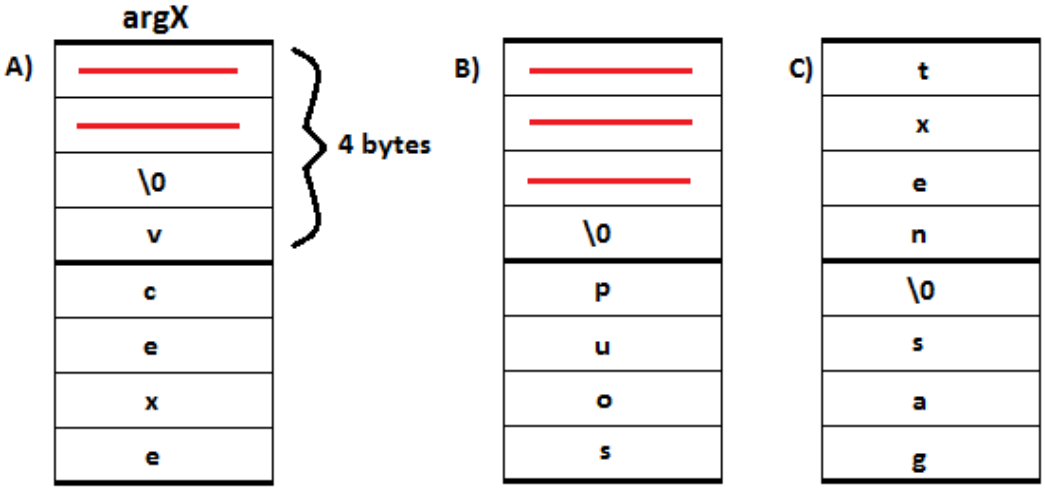
   Working backwards, argv has to be in the address space of the process when it switches back to user mode, otherwise there will be memory permission issues. The code/data parts of the address space are off limits because argv could get overwritten, or overwrite memory there that is being used. This leaves the stack, which is totally empty when the program starts. Therefore, argv can be placed at the bottom of the stack (at the base address) and then the stack pointer can just be incremented by the size of argv so we don't overwrite it.

   The more difficult part is copying argv. The following diagram shows how it is structured. The bottom of each diagram corresponds to the address of the pointer to this

data structure. 3, 6 and NULL corresponds to the pointer array. Here the values of the pointers are given as relative offsets instead of actual addresses, because we don't know what their actual address will be until we copy it into the stack. Arg1 and arg2 are the actual values of arguments.



Each argument must be position at an address divisible by 4 (since pointers have this restriction), but each character only takes 1 byte. A shows how the argument "execv" would actually take up 8 bytes in argv. The 2 crossed out bytes are unused. The next argument would be placed directly above the two red lines. B shows how we have to ignore the remaining 3 bytes because the terminating character spills over into the next 4-byte segment. C shows how a 3 character argument will only take up 4 bytes and the next argument can be placed directly after it.



execv - First execv checks to make sure it has been given valid arguments. Then it counts the arguments by the process described earlier (also checking them to make sure they are valid pointers). Next it copies the arguments from user memory to kernel memory, organizing the arguments as described above. It then opens the file containing the ELF for the new program and deletes the old address space. A new address space is created and the program is loaded into it. The arguments are copied onto the stack in the new address space, and the pointers are updated. Finally, it switches back to user mode , passing the pointer for the arguments, stack and entry point for the program.