1: every thread maintains a file table. Actually the file descriptor is the index of file table. File table is made from an array of file objects, for each file object, it contains the following fields:(1) vnode (2) mode (3) dup (used when fork() get called) (4) offset(used when reading or writing a file)

Open: first do error checking. Then scan the entire filetable. When we find a empty slot, then the index of that slot will be the file descriptor. Then create a new file object, set the fields for appropriate value, return the file descriptor.

Close: first fo error checking, then check if the file we gonna close is forked(check value of dup). If yes, just decrease the dup by 1 then we are done. Else we need to do the vfs_close() operation to close the file.

Read: first do error checking, then create a kernel buffer and a uio object, then we setup a kernel I/O ,the VOP_READ will read the content and store them in the kernel buffer . We still need to copy this chuck of memory to the user-space. By using copyout(), not only the memory gets copied, but also we can ensure the the user-space pointer void* ubuf is valid(error checking). Return total bytes that are read(len - resid value in uio object)

Write: first do error checking, then set up a USER-LEVEL uio, it is different from read() and the reason we don't setup a kernel-level I/O is that the user-space buffer void* ubuf already holds the data. what we need to do is just use VOP_WRITE to write the content pointed by ubuf to the file, then we are done. Return the total bytes written (len - resid value in uio object)

2: I am using a global variable "p_table" to manage process. The "p_table" is an array of process and the index of "p_table" will be the PID. the function get_pid() do the job to find a PID: It searches the entire "p_table", try to find a empty slot, then return this position. When a process is about to exit, it will check whether it has a parent. If not, we are pretty sure no one will wait this process, and thus this PID will be no longer needed. if it has a parent, then after this process grabs the exitcode, it must notify its parent to get this exitcode, after that, this PID will be no longer needed.

Fork: it takes a parameter which is the trapframe of the current thread(parent). First we make a copy of parent's trapframe and address space. We must do it as early as possible because the trapframe of the parent process could be changed and will not be what we want anymore. Before calling thread_fork(), set call_from_fork as true because we are going to set relationship in thread_fork(). Then call thread_fork() to fork a new process. Note that i am using semaphore to strictly control the running order of process. Specifically, we don't want to thread_fork returns before the child process has been set up everything. In md_forkentry(), we first need to copy the parent's trapframe to

the stack, then free the memory we just copied from parent process, setup the return value of the child process as 0, advance the epc by 4 because we don't want to forever calling this function. Then copy the parent's addrspace, activate it and now the child is safe to enter user mode. After that, fork returns child's PID

Getpid: this is trivial. Because every thread keeps its PID info, just return curthread->pid that is it.

_exit: we first get the exact process that is about to exit. Then we test whether this process has parent. If the answer is no, we actually even don't need to fill the exitcode(it is still filled under my implementation); because we are pretty sure no other process will wait for it. If the answer is yes, we understand the parent could be waiting or may be waiting at the later time. By getting the parent process and checking waiting field: if parent process is waiting, then the parent process is sleeping, the child process must call cv_broadcast() to notify the parent to wake up; if the parent process is not currently waiting, we must keep the child process info because we are unsure whether parent process will call waitpid() at a later time. Lastly, call thread_exit() to actually exit.

3: waitpid: first do error checking. To prevent deadlock, ONLY parent can wait its child. If parent process wait others other than child or child is trying to wait its parent is not permitted. If parent process finds that the child process has been exited, then just fill the exitcode and return child PID. I am using semaphore, lock and condition variable to solve synchronization problem. Semaphore ensures that only one thread can execute sys_waitpid() at a time. When the parent process is trying to wait its child and finds that the child has not exited, it will first set the waiting field as yes(1), then trying to sleep until the the child exits. (cv and lock used here)Once the child calls broadcast to notify leaving, parent process sets the waiting field as no(0), then destroy child process and free the slot, since the parent process knows that no one can wait this child again! (this answers the question 2 for PID re-use issue)

4:For exception handling, specifically, if user-level code hits a fatal fault, we must let this thread exit immediately(by calling sys__exit()). If it is the fatal kernel-mode fault, we first set register a3 as error state(1), then set register v0 for appropriate return value(code), after that, exit the current running thread

5: suppose out test input: argc: 3; argv = {"test","1234","456",NULL}
execv: first do error checking, we must check every memory segment to make sure they are healthy. After we finish error-checking, we have an another copy of argc in kernel heap space. as drawn and explained the layout of argv in Fig 1(we don't know exact address), Because we are going to run another program, we must clear the current thread's address space, runprogram() will help us

create a brand new one. Set call_from_execv as true(1), because we just make a copy of same argv object in the kernel buffer, we will free it later on.

Runprogram: after stack has been defined. We first decrease the stackptr by (argc+1) * 4 . (NULL also needs 4 bytes)Then set the argv as the same as stackptr since we are going to fill the memory address.By doing this, we use an iteration to first decrease the stackstr to the bottom of element then let the argv points to the current stackptr, then use copyout to actually copy the content from kernel to user space.(order is fixed ,cannot be changed) after we done iteration, set the last index of argv as NULL.
last,we ensure that stackptr is 8 bytes aligned. If runprogram is calling from execv(), we free the memory allocated from execv() and finally go to user mode. See Fig 2 to the final layout in memory

| | |
|---|---|
| 32 | |
| 31 | |
| 30 | |
| 29 | "\0" |
| 28 | "6" |
| 27 | "5" |
| 26 | "4" |
| 25 | "\0" |
| 24 | "4" |
| 23 | "3" |
| 22 | "2" |
| 21 | "1" |
| 20 | "\0" |
| 19 | "t" |
| 18 | "s" |
| 17 | "e" |
| 16 | "t" |
| 15 | argv[3] |
| 14 | argv[3] |
| 13 | argv[3] |
| 12 | argv[3] = NULL |
| 11 | argv[2] |
| 10 | argv[2] |
| 9 | argv[2] |
| 8 | argv[2] = 26 |
| 7 | argv[1] |
| 6 | argv[1] |
| 5 | argv[1] |
| 4 | argv[1] = 21 |
| 3 | argv[0] |
| 2 | argv[0] |
| 1 | argv[0] |
| 0 | argv[0] = 16 |

FIG 1

| | | |
|---|---|---|
| stackptr(initial position) | 0x80000000 | argv[3] |
| | 0x7fffffff | argv[3] |
| | 0x7ffffffe | argv[3] |
| u_argv[3] | 0x7ffffffd | argv[3]=0x0(NULL) |
| | 0x7ffffffc | argv[2] |
| | 0x7ffffffb | argv[2] |
| | 0x7ffffffa | argv[2] |
| u_argv[2] | 0x7ffffff9 | argv[2]=0x7fffffe3 |
| | 0x7ffffff8 | argv[1] |
| | 0x7ffffff7 | argv[1] |
| | 0x7ffffff6 | argv[1] |
| u_argv[1] | 0x7ffffff5 | argv[1]=0x7fffffe7 |
| | 0x7ffffff4 | argv[0] |
| | 0x7ffffff3 | argv[0] |
| | 0x7ffffff2 | argv[0] |
| u_argv, stackptr(begin loop) | 0x7ffffff1 | argv[0]=0x7fffffec |
| | 0x7ffffff0 | "\0" |
| | 0x7fffffef | "t" |
| | 0x7fffffee | "s" |
| | 0x7fffffed | "e" |
| stackptr(1st loop) | 0x7fffffec | "t" |
| | 0x7fffffeb | "\0" |
| | 0x7fffffea | "4" |
| | 0x7fffffe9 | "3" |
| | 0x7fffffe8 | "2" |
| stackptr(2nd loop) | 0x7fffffe7 | "1" |
| | 0x7fffffe6 | "\0" |
| | 0x7fffffe5 | "6" |
| | 0x7fffffe4 | "5" |
| stackptr(end loop) | 0x7fffffe3 | "4" |
| | 0x7fffffe2 | |
| | 0x7fffffe1 | |
| stackptr(final position) | 0x7fffffe0 | |

FIG 2