Design Documentations

Question 1:

File descriptor is implemented as a new data structure, which has public members: file_ID, flag, offset and vnode. And also there is another data structure called DescriptorTable, which has an array of all file descriptors that a thread can have. By default, every thread has one descriptorTable initially has 3 file descriptors: stdin, stdout and stderr. Moreover, each descriptorTable also contain an array storing available file IDs that can be signed to every new created file descriptor. Totalnum represents the total number of IDs available in the available IDs array and next is just an integer, which has value of next available ID. There are also some methods to modify descriptorTable: constructor, destructor, add_one, delete_one, search, get_new_ID. As described by name, constructor just creates a descriptorTable for a thread, and destructor just destroy the descriptorTable in case of memory leak. Add_one, delete_one and search just trace thought the descriptor array to add, remove or search for a descriptor. Get_new_ID just obtain an available file id. Open() just create a new file descriptor and open or create a file by using vfs_open to obtain a vnode. Get a available file ID from current thread's descriptorTable and assign it to new file descriptor. And assign vnode obtained by vfs_open to the file descriptor and other correspond fields. At last, add this new file descriptor into current thread's descriptorTable. Close() just remove the descriptor from curthread descriptorTable's file descriptor array. Read() and Write() just go into current thread's descriptorTable and find out the given descriptor and initialize the UIO with the filedescriptor and flag UIO_READ/UIO_WRITE and update the filedescriptor's offset

Question 2:

We create a kernel array call processTable and a structure process. processTable is global and unique to store every process. The index of the process in the array is equal to the process id. We also add pid to the thread structure to indicate which process it belongs to. And the process structure has member:

1. pid, represents the process id of a process
2. parent_pid, represents the process's parent's process id
3. exitCode, represents the exitcode for the process
4. exitStatus, 0 for not exit, 1for exit
5. p_lock, the lock used for each process's operation
6. p_cv, the cv used for waitpid
7. myThread, the pointer indicates the thread belongs to the process

We used the functions new_pid() and create_process() to generate a new pid. createprocess() is used to allocate the space for new processes and initialize the processes,

and new_pid() is to go through the processTable to check if there are free pid not assigned, if there is none then call create_process().
The member exitStatus of the process structure indicates whether the process has exited or not. 0 for not exited, 1for exited. Then if it has exited its pid can be reused.
fork:
We used sys_fork() to copy the trapframe argument that is passed in and store the states of current thread and call the thread_fork to fork a new process and new thread. Then md_forkentry() will be used by the child process to copy the trapframe of the parent present. Then it will active the t_vmspace. and set the registers v0, a3, epc. Then dispatch to usermode. If the thread_fork() is called by the sys_fork(), then the thread_fork() has to set the process's parent to its parent process and copy the file table and the address space
If the thread_fork() is not called by sys_fork(), it will call constructor to create new file descriptor table.
_exit:
It sets the exitCode according to the argument that is passed in. Find the process according to the current thread's pid in the processTable. Set the exitStatus to 1. Then wake up all the process that are waiting for the current process to finish and call thread_exit()
getpid:
Just return the pid value of the current thread

Question 3:
waitpid:
First check if the arguments are valid. Then check if the process exists. Then get the process from the process table and check if current thread is its parent. Then acquire the lock. Then if the exitStatus of the child process is 0, which is not exit, then let the parent process cv_wait. After the parent process get the lock of the process, set the status according to the exitCode and set return value to pid.
We used a lock called p_lock and a cv called p_cv for every process. The restriction is that the process that call sys_waitpid() has to be the parent of the process the pid indicates.

Question 4:
argc is calculated using a function called getargc(char **args) in execv, while it is a given parameter in runprogram. In both runprogram and execv, argv can be thought as an array of pointers that points to a string. The arguments are first stored in the kernel buffer.

After the old address space is destroyed, the arguments are copied from the kernel buffer to the user address space of the new process.
Implementation of execv:

1. Allocate space on kernel address space and store the arguments in the kernel address space.
2. Open the program file into v.
3. Destroyed the old address space.
4. Create new address space and activate it.
5. Load the elf file into v.
6. Close the virtual file system.
7. Free the old user stack.
8. Define the new user stack from the address space.
9. Restore the arguments from the kernel buffer to the user address space
10. Warp to user mode.

.