Question 1: How are file descriptors associated with vnodes? What data structures are used to maintain this association?

Vnodes are associated to file descriptors by file descriptors (using the filedescriptor struct). The file descriptor structure contains other useful information, such as the offset and mode with respect to the file. The file descriptor structure is stored in an array called the file table (using the filetable struct) inside each thread. The reason that the file table structure uses an array to store the file descriptors is because there is the index to value associations, such that the file descriptor id is the index of the array.

The file descriptor ids are assigned to the file descriptor structure by the value of the first element of the array with NULL as it's results, this could be a recycled file descriptor id, or a newly inserted file descriptor in the end of the file table.

Question 2: Briefly explain how you implemented PIDs. Please indicate how your kernel generates a PID for each new process. Please also indicate how your kernel determines that a PID is no longer needed by the process to which it was assigned (and is therefore available for re-use).

The kernel generates a PID for a process by first checking a linked list called *recycled_pids*. If the list is empty, it then returns the lowest unused pid (a global integer), then increments the value of lowest unused pid. Since a process in OS161 can only have 1 thread, instead of making a process structure, we just store the PID and other process information in the thread structure. When a PID is no longer needed, it is added to the *recycled_pids* list so it can be reused. A PID is considered to be "no longer needed" when the following 2 conditions are met:
A) The parent process no longer needs the pid since either the parent process has exited or the parent process has called wait_pid() and returned.
B) The process that owns the pid has exited
This is sufficient, since the parent of the process is the only thread other than the process itself which needs the PID, so once they are done with it, it is no longer needed.

Question 3: Briefly explain how your kernel implements the waiting required by waitpid. Did you use synchronization primitives? If so, which ones and how are they used? What restrictions, if any, have you imposed on which processes a process is permitted to wait for?

Instead of using CV's for waitpid, I just disabled interrupts before entering the loop, then called *thread_sleep* on the PID it's waiting for in that loop. *thread_wakeup* is called on that PID when a thread with that PID exits. If the thread has already exited, the parent will never enter the loop and sleep since it first checks to see if the child has already exited. To check if a PID belongs to one of it's child processes and to check if the child has exited, each thread keeps a child table which is a linked list containing the child's PID, weather the child has exited or not, and the exit code of the child. I have only allowed a process to wait on it's child processes.

Extra Information

[runprogram & execv]:

Runprogram and execv are fundamentally the same, except execv requires a bit of extra work. Since execv takes the program arguments from the userspace, and not the kernel space, we must first copy the data to kernel space, so that we can destroy, and recreate the processes address space without losing this data. Then both runprogram and execv copy this data to the new userspace stack.

It was important to align the stack memory properly, making sure our pointer was divisible by eight, and our variables (other than characters) be stored at addresses divisible by 4. Also, since the stack grows down, but the arguments must be in order going up in the stack, we had to position the arguments at the bottom of our stack frame, and store the argument strings at the top, this required pre-calculating the size of the stack frame.

Another extra consideration for execv is error handling. Since the arguments come from user space and not the kernel, they cannot be trusted and must be error checked more rigourously. This involved making sure all pointers were valid, and argument sizes were acceptable. We used a max number of arguments of 16, since this was the limit imposed by the kernel menu.

[Open]

The open system call essentially opens the given filename (device or file) and creates a new file descriptor structure and assigns a new file descriptor id, as well as sets the offsets and flags for the file descriptor structure, and finally adds to the given thread's file table.

The open system call checks for potential errors, as well as follows the OPEN_MAX limit as defined in kern/limits.h.

[Close]

The close system call essentially closes the vnode in the file descriptor by the given file descriptor id. The system call will return error value if the system call is not successful.

[Read & Write]

The read and write system call essentially gets the file descriptor from the file table, and depends on the operation, it will either call VOP_WRITE or VOP_READ to read or write to the given file descriptor id. These system calls will also update the offsets of the file when ever a read or write operation has been performed.

The system call checks for potential errors (as seen from testbin/badcall tests) and will return the proper error codes.