Question 1 (file descriptors)

We have the following structure to implement a file descriptor:

```
struct files {
        char *filename;                 // file name
        struct vnode* vn;               // pointer to the file's vnode
        int flags;                      // record the permission
        int offset;                     // the file pointer offset
        int fd;                         // record the index in the file table
};
```

We also created an array within the thread structure, as a file table, to record the file descriptors of each thread:

struct files* files[MAX_OPENED_FILES];

MAX_OPENED_FILES is pre-set to 20, which means an error will be encountered when the number of opened files exceeds 20 under a thread.

The first three positions of the array are reserved for standard input, standard output, and standard error.

Each system call returns 0 on success, and the actual return value must also be recorded. Respectively, it returns -1 on error, and errno is set according to the error encountered. Therefore, it must contains an extra parameter which is a pointer to the actual return value (index of the file table when succeed, error when error). We call this parameter **retval**.

open:

- basic error checking (if filename is valid, if address is valid, if flag is valid, etc.)
- allocate a vnode for vfs_open, error return if vfs_open fails
- allocate a file descriptor to record the file information, error return if allocation fails
- find the next available place in the thread's file table starting index 3 (since index 0, 1, and 2 are reserved for stdin, stdout, and stderr), error return if file table is full
- write the file descriptor into the file table, if success, return 0

close:

- error checking for a valid fd
- call vfs_close to close the file
- call files_destroy to destroy the file descriptor

read:

- error checking for a valid fd similar to close
- create an uio named copyUIO and a char* console to acquire the console device
- allocate nbytes on kernel heap to use as a buffer for reading from console into kernel, initialize all the standard device if fd is less than 3 and files[fd] is NULL
- call mk_kuio to pass parameters in order to set up UIO
- call VOP_READ to read from vnode to UIO which holds the kernel buffer
- call copyout to copy the buffer into the the user space buffer

write - similar to read except that:

- call copyin to copy data from user space buffer to kernel space buffer
- call VOP_WRITE to write the vnode according to the UIO

Question 2 (process identifiers)

A new field **pid_t pid** is added in the thread structure to record the current thread's PID.
A process structure is created to record processes' information to keep everything on track:

```
struct process {
        int exitcode;                   // saved exit code when sys_exit
        pid_t pid;                      // the process id for each process
        int active;                     // 1 if currently executing; 0 if called by sys_exit
        struct array *children;         // used to store its children processes
        struct semaphore *processSem;   // used to wait child thread to complete
        int parentWaiting;              // used to determine if its parent is waiting for it
};
```

A process table is also created, as a global array of processes, to maintain the processes in the system (e.g. store all the children processes in the process structure):

```
extern struct process *process_table;
```

when a thread is created, we loop through the process table to find the next available PID such that its active field is 0. This determines that this PID is no longer needed by the process, and therefore it is available for reuse.
A new field, **forkcalled**, is also added under thread structure to determine whether the thread is called by sys_fork.
Fork is involved in the following functions:
- sys_fork:

- allocate a new tf and call memcpy to copy the memory to the new tf
- pass the new tf and the function md_forkentry as parameters to call thread_fork
- call P() to wait md_forkentry function finish executing, once P gets the access, return 0

- md_forkentry:

- copy the tf to its own kernel stack and make it local since it cannot be used by other threads, then free the original trapframe to avoid memory leak
- change the return value to 0 for a child thread and increment epc by 4 to avoid repeat
- call V() to release token for sys_fork and call mips_usermode to return the user mode

- thread_fork:

- if the current thread is called by sys_fork then:
    - copy the virtual address space to the newguy
    - copy the file table with file descriptors to the newguy and increment the open file numbers
- if not, do the normal thread fork

getpid:

- simply return curthread->pid to get the PID of the current thread.

exit:

- find the current PID in the process table and update its exitcode
- close all the files of the current thread by looping through the file table
- set all children's parentWaiting field to zero (interrupts must be disabled)
- set its active field to zero and call V() to release token for waitpid

Question 3 (waiting for processes)

We used a semaphore between waitpid (discussed in waitpid part below) and exit functions (discussed in exit part above). A process can only wait for its child.

waitpid:

- basic error checking for valid options and status
- loop through the children array to find whether the child's PID exists in the array
- update the status to the child's exitcode and remove the child from the array
- call P() to wait the child finish executing; once it exits and the access is obtained

Question 4 (argument passing)

argc - calculate the size of passed argument (argv) using a loop; adding 1 space at the end

argv - call as_define_stack to get the initial stack pointer address (assume the stackptr is 0x80000000 when initialized); copy the arguments directly into the user space stack where stack pointer pointes to using copyoutstr.

Since pointers must be divisible by 4 in MIPS, we used casting (to char*) to pad the address.

Execv:

- open the specified program file, load it into ELF file, and then close it
- create a new address space for this process and destroy the original one if necessary, and then activate the new address space
- store each argument to user mode address (using copyoutstr) where the stack pointer points to as the pointer decrements, and then align the pointer
- align the pointer after storing all arguments to user space

The figure beside illustrates an example of "foo", "hello", where padding is necessarily used