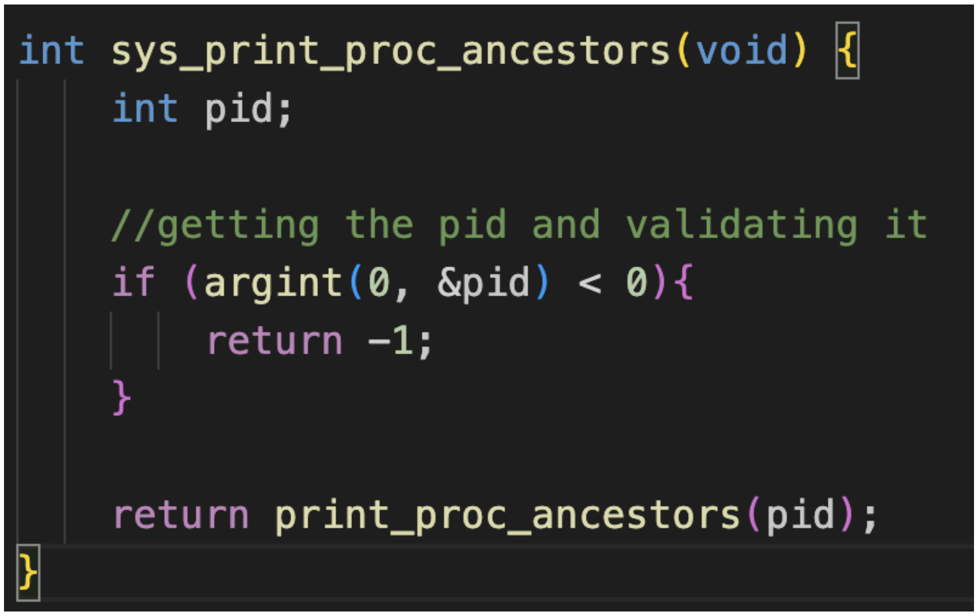
xv6 System Call Implementation Q&A (40 points)

Answer the following questions about system call implementation.

(1) (10 points) Where is the wrapper function of this system call that prints the ancestor processes of the specified process (i.e., the int print proc ancestors(int pid) function) defined?

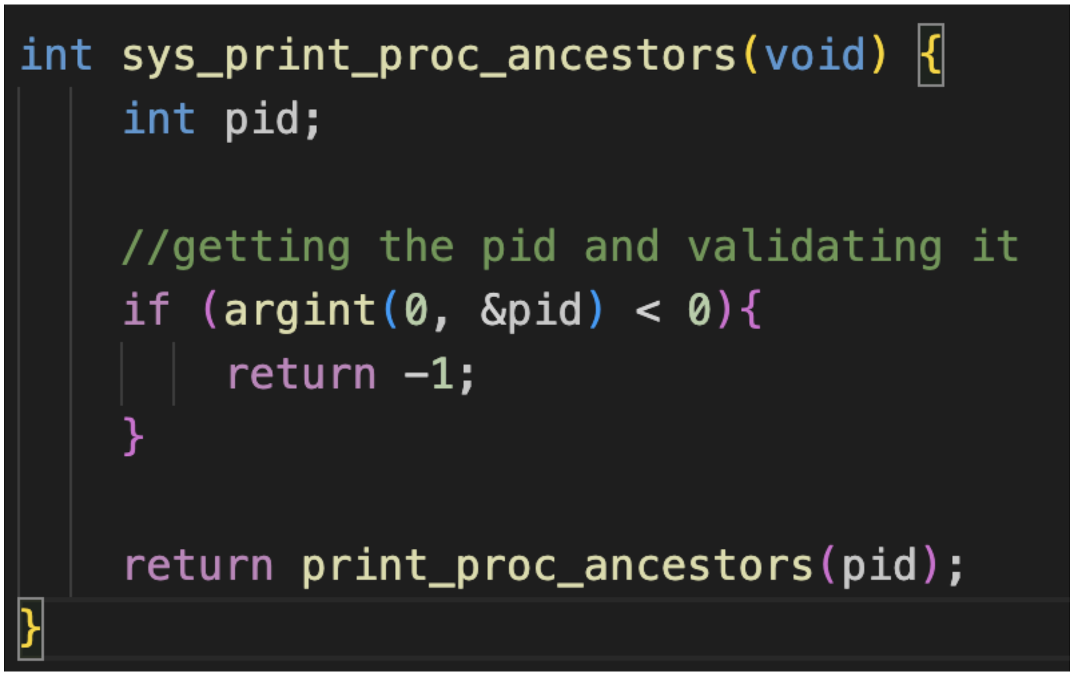
Answer: The wrapper function of **int print proc ancestors(int pid)** is **sys_print_proc_ancestors(void)** and it is defined in the **sysproc.c** file. The declaration of this wrapper function is done in the syscall.h and syscall.c file. This wrapper function calls our kernel-level implemented function **int print proc ancestors(int pid)** which has the function declaration done in the user.h and defs.h file.



```
int sys_print_proc_ancestors(void) {
    int pid;

    //getting the pid and validating it
    if (argint(0, &pid) < 0){
        return -1;
    }

    return print_proc_ancestors(pid);
}
```

(2) (10 points) Explain (with the actual code) how the above wrapper function triggers the system call.

Answer:

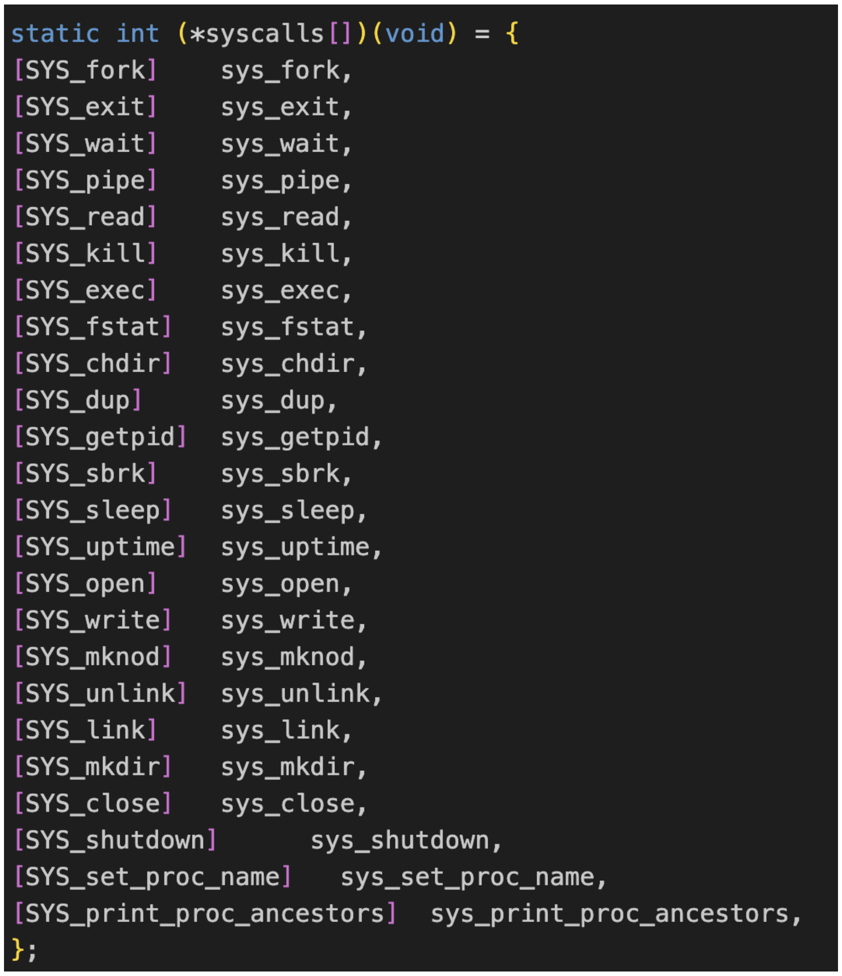


```
int sys_print_proc_ancestors(void) {
    int pid;

    //getting the pid and validating it
    if (argint(0, &pid) < 0){
        return -1;
    }

    return print_proc_ancestors(pid);
}
```

This is the wrapper function that will trigger the system call. Here we will first declare a variable pid. Using the argint function of xv6 we extract the integer argument present at index 0 and store it in the variable pid using a pointer. This operation's value is then checked if it is less than 0 or not. This is done because, in xv6, negative values are returned when any error occurs. We are then returning the value returned by the print_proc_ancestors function which will be either 0 or -1 depending on the failure or success. If the call is successful then we will return 0 else we will be returning -1 from the print_proc_ancestors function. This is how the system call will be triggered and the value of the parameter will be passed which is extracted using argint function.

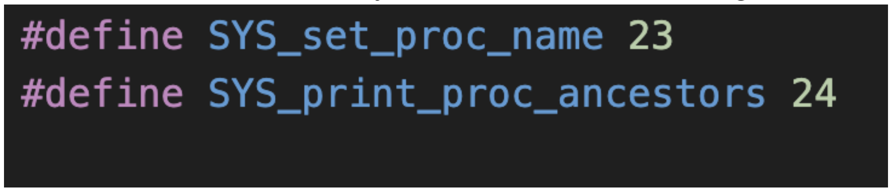
(3) (10 points) Explain (with the actual code) how the OS kernel locates a system call and calls it (i.e., the kernel-level operations of calling a system call).

Answer:



```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_shutdown]      sys_shutdown,
[SYS_set_proc_name]   sys_set_proc_name,
[SYS_print_proc_ancestors]   sys_print_proc_ancestors,
};
```

Using the **system call dispatcher** which is present in syscall.c, the OS kernel locates a system call and calls it. This dispatcher is an array of function pointers. The index values are defined in the syscall.h file in the following manner.



```
#define SYS_set_proc_name 23
#define SYS_print_proc_ancestors 24
```

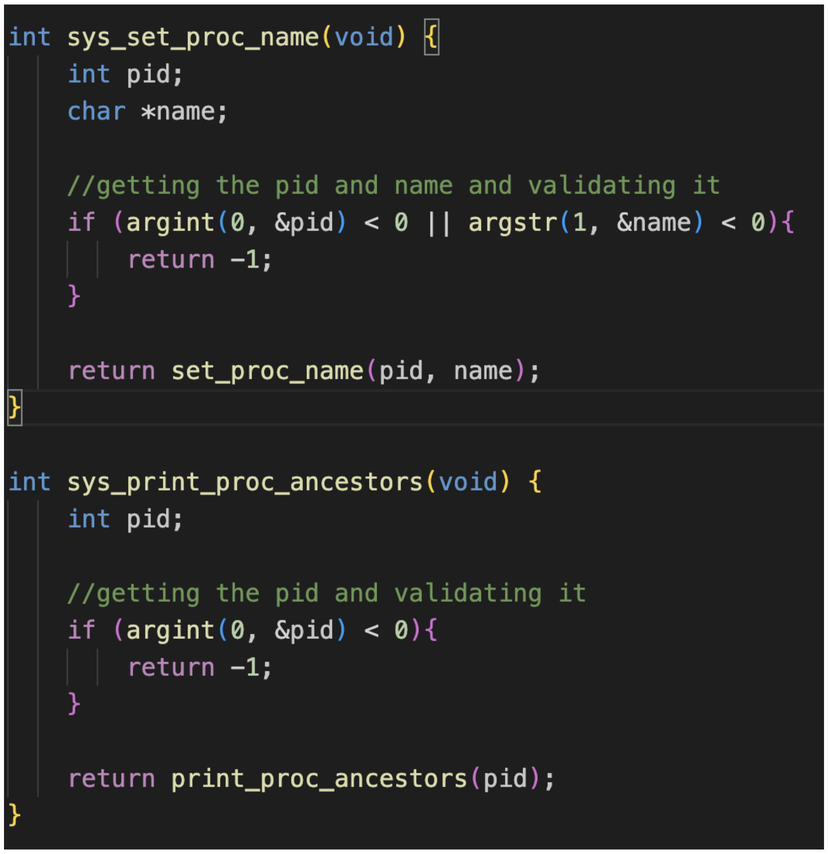
The dispatcher calls the function using a function pointer retrieved by the system call number.

(4) (10 points) How are arguments of a system call passed from user space to the OS kernel?

Answer: Using the system call Interface mechanism we pass the arguments of a system call from user space to the OS kernel. In the following code snippet below you can see we are making use of this mechanism with the help of 2 functions, i.e. **argint** and **argstr.**

- **argint** function is defined in xv6 which is used to fetch the integer argument and store it in pid. argint takes 2 arguments, the first argument is the index value from where the integer argument needs to be extracted and the second argument is a pointer to a variable where the value needs to be stored. Over here we have used argint(0, &pid) to fetch the integer argument from index 0 and store that value in the pid variable.
- **argstr** function is defined in xv6 which is used to extract string value and store it in the variable name. argstr also takes 2 arguments, the first argument is the index value from where the string needs to be extracted and the second argument is where it needs to be stored. Over here we have used argstr(1, &name) to extract the string value which is then stored in the name variable.

These functions help the kernel extract and use the arguments that are stored in a specific format in the memory.



```
int sys_set_proc_name(void) {
    int pid;
    char *name;

    //getting the pid and name and validating it
    if (argint(0, &pid) < 0 || argstr(1, &name) < 0){
        return -1;
    }

    return set_proc_name(pid, name);
}

int sys_print_proc_ancestors(void) {
    int pid;

    //getting the pid and validating it
    if (argint(0, &pid) < 0){
        return -1;
    }

    return print_proc_ancestors(pid);
}
```