COMP 3500 Introduction to Operating Systems

Project 4 – Processes and System Calls

Points Possible: 100
Submission via Canvas
No collaboration among groups. Students in one group should NOT share any project code with any other group. Collaborations among groups in any form will be treated as a serious violation of the University's academic integrity code.

Objectives:

- To collaborate with your group members using CVS
- To start building a multi-tasking operating system
- To develop system calls to manage processes
- To develop system calls to manage file system states
- Use GDB to debug OS/161

1. Working in Teams

This project requires you to work in teams of two, and you are expected to gain real-world experience of working in a team effectively. The collaboration will be made possible with a shared CVS repository. It is desirable to resolve conflicts among you and your group members about things such as naming conventions and programming style at an early stage to avoid confusion later. The following issues need to be considered while you and partners are working together.

1.1 Naming Conventions

You are suggested to come up with a protocol for naming global variables and variables passed as arguments to functions. To achieve this goal, you may ask your group members to write some functions, and you can make calls to those functions in your own code while she or he is implementing it. Please make sure that you have a common naming convention and a consistent way of writing function names. You may choose a model and stick to it. For example, given a function called "my function", you might write the name as my_function, myFunction, MyFunction, etc.

1.2 CVS Use

You and your group members have to decide when and how often to commit changes. Please commit changes of your work as early and often as possible. You also need to agree upon how much detail to log while committing files. In addition, you need to figure out a way of maintaining the system integrity. For example, what procedures to follow to ensure that you are able to extract a working version of some sort from CVS,

what tests to run on a new version to make sure you haven't inadvertently broken something, etc.

It is essential to make use of explicit CVS logs. If for some reason you become incommunicado, your partner should be able to reconstruct your design, implementation and debugging process from your CVS logs. It is worth noting that leaving something uncommitted for a long period of time is dangerous, which means that you are undertaking a large piece of work without effectively breaking it down. In general, when some new code compiles and doesn't break the world, commit it. When you have got a small part working, commit it. When you have implemented something and written a lot of comments, commit it. Hours spent hand-merging hundreds of lines of code wastes time you may never get back. The combination of frequent commits and good, detailed comments will facilitate more efficient program development.

Important! Whenever you cvs add a file, do `chmod 640` or `chmod 644` on the file before committing it.

Please leverage the CVS features to conduct your collaborative project. Specifically, you may use tags to identify when major features are added and when they are stable. You also may investigate CVS branches, which provide completely separate threads of development.

1.3 Communication

Important! You need to identify in your design documents who was responsible for various parts of your solutions.

It is a good idea to open communication among team members. Your group will become productive if you and your partners can direct the communication to issues of content, e.g., "How shall we design open()?" rather than "What do you mean, you never checked in your version of `foo.c`?".

It is vital to make a partnership work, and your are responsible for managing the partnership. If you and your partners discover that you are having difficulty working together, please come speak with the TA or the instructor, who will be working with you to help your partnership work more effectively.

2. Overview

2.1 Towards a Real Multi-tasking Operating System

In the previous project (i.e., project 3), you may have considered your solutions to the cats and mice problem as separate programs, but they were really functions that were

linked into the kernel itself and thus ran inside the kernel, in kernel mode. We took this approach because the kernel was unable to run user-space code. In this project, however, you will be making OS/161 deal with user-space programs. At present, OS/161's only working system call available to user-space code is reboot. The kernel itself doesn't currently understand what a process is or maintain any per-process state. Your current OS/161 system has minimal support for running executables - nothing that could be considered a true process. In this assignment you will be making OS/161 a real multi-tasking operating system. After the next assignment, your OS/161 will be able to run multiple processes simultaneously from actual compiled programs stored in your account. Specifically, the programs will be loaded into your OS/161 and executed in user mode by the System/161. This will occur under the control of your kernel and the command shell in `bin/sh`.

2.2 System Calls

First, you have to implement the interface between user-mode programs ("userland") and the kernel. You will be provided part of the code needed for this assignment, and you are required to design and implement the missing pieces. Your job also includes implementation of the subsystem that keeps track of the multiple tasks you will have in the future. Importantly, you need to design data structures for "process" (**Hint**: look at kernel include files of your favorite operating system for suggestions, specifically the `proc` structure.)

In the first phase of your project, you are advised to read and understand the parts of the system available for you. Our code can run one user-level C program at a time, provided that it does nothing but shut the system down. To do this, we have implemented sample user programs like reboot, halt, poweroff, as well as others that make use of features you will be adding in this and future assignments.

Important! The code you have written so far for OS/161 has only been run within, and only been used by the operating system kernel. In a real world operating system, the kernel's main function is to support user-level programs. Such support is made possible through system calls. One system call, `reboot()`, was implemented in the function `sys_reboot()` in main.c. With GDB in hand, you are capable of putting a breakpoint on `sys_reboot` and running the "reboot" program. In doing so, you can use "backtrace" to see how it got there.

2.3 Run User Level Programs

You can run normal programs compiled from C on the System/161 simulator. The programs are compiled with a cross-compiler, namely `cs161-gcc`. This compiler runs on

the host machine and produces MIPS executables; it is the same compiler used to compile the OS/161 kernel.

Important! In order to create new user programs, you need to edit the Makefile in `bin`, `sbin`, or `testbin` (depending on where you put your programs) and then create a directory similar to those that already exist. Use an existing program and its Makefile as a template.

2.3 Design

Your design documents become an important part of the work you submit. The design documents must reflect the development of your solutions. Simply explaining what you programmed in the design documents is not sufficient. Please do **NOT** attempt to code first and design later, because you are likely to end up in a software "tar pit". You are suggested to plan everything you will do with your partners. Do **NOT** rush into coding until you can clearly describe to one another what problems you need to solve and how the pieces relate to each other. To enforce you to follow this design process, we set a separate due date for the design document.

In most cases, it is hard to write or talk about new software design. You will be facing problems that you have not seen before (e.g., finding terminology to describe your ideas can be difficult). However, the problem becomes easier to be solved if you can gain hands-on experience by going ahead and trying. Please make an attempt to describe your ideas and designs to your group members. To reach an understanding, you may have to invent terminology and notation-this is fine, and please explain it in your design document. If you do this, once you have completed your design, you will find that you have the ability to efficiently discuss problems that you have never seen before.
We provide the following questions as a guide for reading through the code. You are recommended to divide up the code and have each partner answer questions for different modules. After reading the code and answering questions, you can get together and exchange summations of the reviewed code. By the time you have done this, you are in a position to discuss strategy for designing your code for this assignment.

3. Code Reading

You do not need to submit your answers to these questions, but you must make an effort to answer them. The questions are designed to direct your attention to those parts of the OS/161 that are particularly pertinent to this project to improve your understanding of process-related code in the OS/161.

3.1 `kern/userprog`: the user program

`kern/userprog`: This directory contains the files that have responsibility to load and run of user-level programs. At present, the only files in the directory include `loadelf.c`, `runprogram.c`, `anduio.c`. However, you may add more of your own during this assignment. Understanding these files plays an important role in getting started with the implementation of multiprogramming. Please note that you will have to look in files outside this directory in order to answer some of the questions.

`loadelf.c`: This file contains the functions needed for loading an ELF executable from the filesystem and into virtual memory space. ELF is the name of the executable format produced by `cs161-gcc`. Currently, the virtual memory space does not provide what is normally meant by virtual memory. Although there exists translation between the addresses that executables believe they are using and physical addresses, there is no mechanism for providing more memory than exists physically.

`runprogram.c`: This file contains one function, `runprogram()`, which is designed to run a program from the kernel menu. It is a good base for writing the `fork()` system call, but only a base -- when writing your design doc, you should determine what more is required for `fork()` that `runprogram()` does not concern itself with. Additionally, once you have designed your process system,`runprogram()` should be altered to start processes properly within this framework, e.g., a program started by `runprogram()` should have the standard file descriptors available while it is running.

`uio.c`: This file contains functions used to move data between kernel and user space. Knowing when and how to cross this boundary is critical to properly implementing userlevel programs. As a result, this is a file that must be read very carefully. You also need to study the code in `lib/copyinout.c`.

Questions

1. What are the ELF magic numbers?
2. What is the difference between UIO_USERISPACE and UIO_USERSPACE? When should one use UIO_SYSSPACE instead?
3. Why can the `struct uio` that is used to read in a segment be allocated on the stack in `load_segment()` (i.e., where does the memory read actually go)?
4. In `runprogram()`, why is it important to call `vfs_close()` before going to usermode?
5. What function forces the processor to switch into usermode? Is this function machine dependent?

6.In what file are `copyin` and `copyout` defined? `memmove`? Why can't `copyin` and `copyout` be implemented as simply as `memmove`?
7.What (briefly) is the purpose of `userptr_t`?

3.2 `kern/arch/mips/mips`: traps and syscalls

Exceptions, which are fundamental for an operating system, are the mechanism that enables the OS to control execution and therefore do its job. Exceptions can be envisioned as the interface between the processor and the operating system. When the OS boots, it installs an "exception handler" (carefully crafted assembly code) at a specific address in memory. Once the processor raises an exception, the exception handler is invokes. The exception handler sets up a "trap frame" and calls into the operating system. Since "exception" is such an overloaded term in computer science, operating system lingo for an exception is a "trap" - when the OS traps execution. Interrupts are exceptions, and more significantly for this assignment, so are system calls. Specifically, `syscall.c` handles traps that happen to be syscalls. Understanding at least the C code in this directory paves a path towards being a real operating systems junkie. Therefore, you are highly recommended to read through it carefully.

`trap.c: mips_trap()` is the key function that is responsible for returning control to the operating system. This is the C function called by the assembly exception handler. `md_usermode()` is the function for returning control to user programs.

`kill_curthread()` is the key function for handling broken user programs; when the processor is in user mode and hits something it can't handle (i.e., a bad instruction), it raises an exception. There is no way to recover from this, so the OS needs to kill the process. Part of this assignment will be to write a useful version of this function.

`syscall.c`: `mips_syscall()` is the function that delegates the actual work of a system call to the kernel function that implements it. Notice that `reboot()` is the only case currently handled. You will also find a function, `md_forkentry()`, which is a stub where you will place your code to implement the `fork()` system call. It should get called from `mips_syscall()`.

Questions

1.What is the numerical value of the exception code for a MIPS system call?
2.Why does `mips_trap()` set `curspl` to `SPL_HIGH` "manually", instead of using `splhigh()`?
3.How many bytes is an instruction in MIPS? (Answer this by reading `mips_syscall()` carefully, not by looking somewhere else.)

4.Why do you "probably want to change" the implementation of `kill_curthread()`?

5.What would be required to implement a system call that took more than 4 arguments?

3.3 `~/cs161/src/lib/crt0`: user program startup

`lib/crt0`: This is the user program startup code. There is only one file in here, `mips-crt0.S`, which contains the MIPS assembly code that receives control first when a user-level program is started. It invokes the user program's `main()`. This is the code that your `execv()` implementation will be interfacing to, so be sure to check what values it expects to appear in what registers and so forth.

`lib/libc`: This is the user-level C library. You are not expected to read a lot of code here, even though it may be instructive in the long run to do so. Job interviewers have a habit of asking people to implement standard C library functions on the whiteboard. For present purposes you need only look at the code that implements the user-level side of system calls, which we detail below.

`errno.c`: This is where the global variable `errno` is defined.

`syscalls-mips.S`: This file contains the machine-dependent code needed for implementing the user-level side of MIPS system calls.

`syscalls.S`: This file is created from syscalls-mips.S at compile time and is the actual file assembled into the C library. The actual names of the system calls are placed in this file using a script `calledcallno-parse.sh` that reads them from the kernel's header files. This avoids having to make a second list of the system calls. In a real system, typically each system call stub is placed in its own source file, to allow selectively linking them in. OS/161 puts them all together to simplify the makefiles.

Questions

1.What is the purpose of the `SYSCALL` macro?

2.What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)

4. Coding Exercises

Tag your repository as `asst2-begin` before you begin this assignment.

4.1 System Calls and Exceptions

Implement system calls and exception handling. A range of system calls that you may want over the course of this semester is listed in `kern/include/kern/callno.h`. For this assignment you must implement:

- `open, read, write, lseek, close, dup2`
- `getpid`
- `fork, execv, waitpid, _exit`

Your syscalls should handle a variety of error conditions without crashing the OS/161. You must consult the OS/161 man pages (see html files in `~/cs161/src/man/syscall`) included in the distribution and understand fully the system calls that you must implement. You should return the error codes as described in the man pages. Your syscalls must return the correct value in case of success or error code in case of failure as specified in the man pages. Some of the grading scripts rely on the return of appropriate error codes. Note that following the guidelines is as important as the correctness of the implementation.

Important! The file `include/unistd.h` contains the user-level interface definition of the system calls that you will be writing for OS/161. The interface is different from that of the kernel functions that you will define to implement these calls. You need to design this interface and put it in `kern/include/syscall.h`. As you discovered in Assignment 0, the integer codes for the calls are defined in `kern/include/kern/callno.h`. You should consider a number of issues regarding implementing system calls. Perhaps the most obvious one is: can two different user-level processes (or user-level threads, if you choose to implement them) find themselves running a system call at the same time? Be sure to argue for or against this, and explain your final decision in the design document.

4.2 Managing Per-process File System State

`open(), read(), write(), lseek(), close(), dup2()`

Given a process, the first file descriptors (0, 1, and 2) are considered to be standard input (stdin), standard output (stdout), and standard error (stderr). These file descriptors must start out attached to the console device ("con:"), but your implementation should allow programs to use `dup2()` to change them to point elsewhere.

These system calls may seem to be tied to the file system, but these system calls are necessary for manipulation of file descriptors, or process-specific filesystem state. A large part of this assignment is to design and implement a system to keep track of this state. Some of this information like the current working directory is specific only to the process, but others such as file offset is specific to the process and file descriptor. Don NOT rush this design. Please give a deep thought about the state you need to maintain,

how to organize it, and when and how it has to change. To keep consistency, please place most of your implementation in the following files:

`userprog/file.c` Function implementations and variable instantiations
`include/file.h` Function prototypes and data types

- **`open(const char *path, int oflag, mode_t mode)`**: It should be as simple as it seems.
- **`read(int fd, void *buf, size_t nbytes)`**: You can use `struct uios`, and make use you understand it. You also can check out `VOP_READ`.
- **`write(int fd, const void *buf, size_t nbytes)`**: Involves I/O to userland. You can use `struct uios` again. Please check out `VOP_WRITE`.
- **`lseek(int fd, off_t offset, int whence)`**: Can we always perform an lseek? For example, can we perform `lseek()` beyond the end of a file? Please use `VOP_TRYSEEK`.
- **`close(int fd)`**: It may be interesting because of the refcounting issue (think about garbage collection)!
- **`dup2(int oldfd, int newfd)`**: If `newfd` is already opened, close it. Upon successful completion, both file descriptors refer to the same file table object and share all properties of the object.

There are a number of things to note here:

- The first thing that needs to be designed is the per-process file table. The file descriptor that you hand back to an application is just the index into the process's file table. You need to decide what goes into this file table. A familiarity with the VFS layer is a must before designing this.
- These file system calls are not implementation dependent. Thus, they must work with any file system by the virtue of the higher-level VFS layer.
- How are open files represented in user space? File descriptors.
- There are three standard file descriptors, STDIN, STDOUT, and STDERR. To what device should these be initially attached?
- File descriptors are indexes into the file handles. How do you want to assign new file descriptors, and should there be a limit to the number of file descriptors? How do you assign file descriptors?
- Each file descriptor has an associated file offset. What happens with the offset if you open the same file twice?
- How should you convert between file descriptors and more meaningful information? How are open files represented in the kernel? vnode. Keep track of this correspondence using the file table.
- The file table helps associate open file descriptors with the corresponding vnodes, keeps track of the current file offset.

- What happens to a file table when you fork a process, or when the process exits? Each process has one file table, thus it is copied the a process forks. What happens to the seek position when you:(1) dup2()? (2) fork()? And (3) open the same file twice?
- When a process exits, its a good idea to close all its open files. Please be careful here. Think about a case where the process has forked.

4.3 Managing Process State

- **`getpid()`**: This will be the simplest function you write in this semester. A pid, or process ID, is a unique number identifying a process. The implementation of `getpid()` is not challenging, but pid allocation and reclamation are the important concepts that must be implemented. Your system should not crash because over the lifetime of its execution the system has used up all the pids. Design your pid system; implement all the tasks associated with pid maintenance, and only then implement `getpid()`.

4.4 Managing Processes

`fork(), execv(), waitpid(), _exit()`

These system calls play a big role in this assignment, because they enable multiprogramming and make the OS/161 a useful entity.

- **`fork()`**: `fork()` is the mechanism necessary for creating new processes. It should make a copy of the invoking process and make sure that the parent and child processes each observe the correct return value (that is, 0 for the child and the newly created pid for the parent). You need to carefully design `fork()` and consider it together with what `execv()` does to make sure that each system call is performing the correct functionality.
    - This system call creates a copy of a calling process.
    - How are you going to copy the file table?
    - What happens to open descriptors?
    - You will have to duplicate the address space. Let's see the `dumbvm` code for a function that may be of help.
    - Is there anything else needed to be replicated/copied? e.g., Current directory
    - A new process must get a PID. `fork()` must abide by the semantics of your PID management mechansism.
    - Please make a child process return 0 and behave like its parent. It is useful to understand the machine dependent system call mechanism for return values and errors in `kern/arch/mips/mips/syscall.c` You may

do something similar in `md_forkentry()`. This is subtle, and please address this issue in your design document.

- Trapframe handling: Please discuss this issue in your design. When a process makes a system call, where and how does it know where to return? It saves return address on the trapframe, and, therefore, the trapframe must be copied. Otherwise, child processes do not know where to return.
- Return 0 to the child process, the process id of the child to the parent. how to return a different value to the child process? Look at how other system calls return and deal with trapframe appropriately. Note that this is machine-dependent code. You have to place it into an appropriate directory. See also `md_forkentry`.

- **`execv(const char *path, const *char argv[])`:** `execv()`, although "only" a system call, is really the heart of this assignment. It is responsible for taking newly created processes and make theme execute something useful (i.e., something different than what the parent is executing). Essentially, it must replace the existing address space with a brand new one for the new executable (created by calling `as_create` in the current `dumbvm` system) and then run it. While this is similar to starting a process straight out of the kernel (as `runprogram()` does), it's not quite that simple. Remember that this call is coming out of userspace, into the kernel, and then returning back to userspace. You must manage the memory that travels across these boundaries very carefully. (Also, notice that `runprogram()` doesn't take an argument vector -- but this must of course be handled correctly in `execv()`).
    - **`execc()`** loads a new executable in the address space of a process
    - **`execc()`** is similar to `runprogram()` in `kern/userprog/runprogram.c`
    - Load the new executable
        - Opens the file
        - Creates an address space into which the image would be loaded and activates it. Please don't worry about what this does until assignment 3, but remember to do it now.
        - Loads the executable into the address space
        - `load_elf` returns `entrypoint`.
        - Defines user stack
        - Returns to user mode
        - Please follow the above 6 steps, and you also need to `copyout` argument to the user stack. What about the old address space?
    - Coping with the argument vector is the hard part. This is hard and subtle.

- Where do we get the arguments from the old program? They are user-level pointer that are passed as arguments to the system call. How can you get hold of them? `copyin` for pointers,`copyinstr` for strings. You need to `copyin` both the pointers and the strings. Where in the process's address space should we put the argument vector? On the stack. Where do we get the arguments from the old program? They are user-level pointer that are passed as arguments to the system call.
- What is a stack? It is just a region of memory in the address space. Stack is used for temporary data during function calls. Why do we need it? To make efficient use of memory. Now getting the arguments into place is basically the opposite of getting them from user space. You will again need `copyin/copyout`. Place things wherever you want, but above the stackpointer. Stackpointer is where the process will start scratching on the stack.
- Note that the argument vector comes from user space. The functions in `kern/lib/copyinout.c` will be very useful in copying everything to/from the kernel address space. Why is this always important? Consider this example. Tom has a secret file. He's been using it, so its buffered in memory. Malicious dude Mal wants to get to it, and he has a pretty good guess where its buffered. Malicious Mal can pass a kernel address to open with `O_CREAT`. Viola, the filename appears with the contents of the file. This is bad, which is why we can't trust any thing coming from userland. Thus, `copyin/copyout`.
- Each of the elements of the argument vector (argv[i]) is a string residing in userland and needs to be copied with care.
- What happens to `argv[i]` in the new address space?
- Make sure you null-terminate argv (i.e. `argv[argc] = NULL`).
- Make sure that all of your pointers are word-aligned.
- Don't forget to set up `stdin`, `stdout` and `stderr` (see also in `runprogram`).
- Since handling arguments in `execv` is so hard, lets do an example. We need to place ls foo on teh user stack. This is what it should look like.

- **`waitpid(pid_t wpid, int *status, int options)`**: Although it may seem simple at first, `waitpid()` requires a fair bit of design. Read the specification carefully to understand the semantics, and consider these semantics from the ground up in your design. You may also wish to consult the UNIX man page, though bear in mind that you are not required to implement all the things UNIX `waitpid()` supports -- nor is the UNIX parent/child model of waiting the only valid or viable possibility.

- These system calls are very closely tied to your PID management system. It's a synchronization problem.
- How are you going to assign PIDs? You are not allowed to just run out. PID recycling?
- Be very specific when you define the semantics of your waitpid() and exit() interactions. The man pages give the minimum requirements. Note that when a process exits() its PID may not be given to a new process right away since the parent might be interested in finding the exit code. What if the parent has already exited itself?
- What does interested mean? Unix defines it strictly in terms of parent/child. The parent can get the child's exit status.
- You may wish to implement `WNOHANG` for `waitpid()`, which allows `waitpid()` to be non-blocking. This is not required but may simplify your shell.
- What PID related data structures are you going to keep in the parent and in the child? Do you need to synchronize the access to those structures and if yes then how?
- How can you make a parent wait for a child? What happens if a child tries to wait for its parent?
- How can you deadlock? (You shouldn't, of course.) Two processes waiting for each other

- **`_exit()`**: The implementation `of _exit()` is closely connected to the implementation of waitpid(). They are essentially two components of the same mechanism. The code for `_exit()` will be relatively simpler and the code for `waitpid()` relatively more complicated. The basic rule is to understand what waitpid() is doing, so your implementation of `_exit()` can work with it.
  - `_exit()` releases all resources used by the process. What are these? Do we always free all resources? What about the exit code. What happens if a child exits before its parent or before any other process that has "expressed interest" in the exit status?
  - Don't forget kill curthread()

Remember to handle all the corner cases. Can you think of some good ones for these system calls?

4.5 Dealing with fatal exceptions in user processes

`kill_curthread()`:Please implement `kill_curthread()` in a simple manner. It is worth noting essentially nothing about the current thread's user space state can be trusted if it has suffered a fatal exception -- it must be taken off the processor in a judicious manner, but without returning execution to the user level.

5. Error Handling of System Calls

The man pages in the OS/161 distribution contain a description of the error return values that you must return. If there are conditions that can happen that are not listed in the man page, return the most appropriate error code from `kern/errno.h`. If none seem particularly appropriate, consider adding a new one. If you intend to add an error code for a condition for which Unix has a standard error code symbol, please use the same symbol if possible. If not, you can make up your own, but note that error codes should always begin with E, should not be EOF, etc. Consult Unix man pages to learn about Unix error codes; on Linux systems "man errno" will do the trick.

If you add an error code to `src/kern/include/kern/errno.h`, you need to add a corresponding error message to the file `src/kern/include/kern/errmsg.h`.

6. Testing

6.1 System Calls

In `bin/sh/sh.c` you will find skeleton code that enables you to test your new system call interfaces. When executed, the shell prints a prompt, and allows you to type simple commands to run other programs. Each command and its argument list are passed to the `execv()` system call, after calling `fork()` to get a new thread for execution. The shell also make it possible to run a job in the background using &. You can exit the shell by typing "exit".

Once you have implemented the system calls like fork(), you are expected to use the shell to execute the following user programs from the bin directory: **`cat`**, **`cp`**, **`false`**, **`pwd`**, and **`true`**. You may find some of the programs in the testbin directory helpful. You need to "bullet-proof" the OS/161 kernel from user program errors. There should be nothing a user program can do to crash the operating system.

6.2 Scheduling

Currently, the OS/161 scheduler implements a round-robin queue. As we mentioned in class, this is not always the best method for achieving optimal processor throughput. For this reason, you are required to implement a second scheduling algorithm. Explain in your design document, why you selected the algorithm you did, and under what conditions you expect it to perform well. Figure out what information you will need to maintain in order to completely implement the algorithm.

Ideally, your scheduler should be configurable, e.g., it should be possible to specify the time slice for the round robin scheduler, and for a multi-level feedback queuing system,

you might want to specify the number of queues and/or the time slice for the highest priority queue. OS/161 should display at bootup time the scheduler currently in use. It is acceptable to recompile your code in order to change these settings, as with the HZ parameter of the default scheduler. It is also fine to require a recompile to switch schedulers. However, this shouldn't require editing more than a couple `#defines` or the kernel config file to make these changes.

Test your scheduler by running several of the test programs from `testbin` (e.g., `add.c`, `hog.c`, `farm.c`, `sink.c`, `kitchen.c`, `ps.c`) using the default time slice and scheduling algorithm (i.e., round robin). Experiment with the scheduling algorithm that you implemented. Write down what you expect to happen with the algorithm. Then compare what actually happened to what you predicted and explain the difference.

7. Desgin Questions

What new data structures will you need to manage multiple processes?
What relationships do these data structures have with the rest of the system?
How will you manage file accesses? When the shell invokes the `cat` command, and the `cat` command starts to read `file1`, what will happen if another program also tries to read `file1`? What would you like to happen?

8. How to Proceed

8.1 Before the design document is due:

1. Meet with your partners. Review the files described above. Separately answer the questions provided. Compare your solutions.
2. Discuss high level implementations of your solutions. Do detailed design in parallel with your partners.
3. Determine which functions you should change and which structures you may create to implement the system calls. Discuss how you will pass arguments from user space to kernel space and vice versa. Discuss how you will keep track of open files. For which system call is this useful? Discuss how you will keep track of running processes. For which system call is this useful?

8.2 Before the assignment is due:

1. Carefully divide up the work. It is suggested that one of you should first build the basic support for per-process file system state management (which you should design together), one should focus on support for processes, and the scheduler, and one should deal with remaining per-process file system state calls, and testing.

2. For the parts you are assigned, verify that the collaborated design will really work. If something needs to be redesigned, do it now, and run it by your partners.
3. Implement.
4. Test, test, and test. Test your partner's code especially.
5. Fix.

8.3 On the assignment due date:

1. Commit all your final changes to the system. Make sure your partners have committed everything as well. Make sure you have the most up-to-date version of everything. Re-run make on both the kernel and userland to make sure all the last-minute changes get incorporated.
2. Run the tests and generate scripts. If anything breaks, curse (politely of course) and repeat step 1.
3. Tag your CVS repository
4. Your assignment has been successfully submitted if all of the appropriate files have been committed to your repository.

9. Deliverables

Turn in your design document by creating an asst2 directory at the top-level of your repository and committing your document. Your design document should be in plain text, with lines no more than 80 characters wide.

9.1 Final Submission

Your final submission should include in your asst2 directory:

- Your final design document.
- A script of the OS/161 shell running various basic commands (`cat, rm, cp, and pwd`) under the OS/161.
- A script of the OS/161 running the various tt* tests successfully.

9.2 Design Document

Your design document must contain:

- A high level description of how you are approaching the problem.
- A detailed description of the implementation, e.g., data structures, why they were created, what they are encapsulating, what problems they solve.
- A discussion of the pros and cons of your approach.
- Alternatives you considered and why you discarded them.

If you are unable to complete this project for some reasons, please describe in your final design document the work that remains to be finished. It is important to present an honest assessment of any incomplete components.

Now, submit your tarred and compressed file named <group_ID>_asst2.tgz through Canvas. You must submit your single compressed file through Canvas (no e-mail submission is accepted. For example, suppose that I am a member of group6, my submitted file would read “group6_asst2.tgz”.

10. Grading Criteria

The approximate marks allocation will be: 1/3 design, 1/3 for functionality, 1/3 for clarity and attention to detail.

1) A high level design description: 5%
2) A detailed description of the implementation: 15%
3) A discussion of the pros and cons of your approach: 15%
4) Alternatives you considered and why you discarded them: 5%
5) System call – managing file system state: 25%
6) System call – managing processes: 25%
7) clarity and attention to detail: 10%

11. Late Submission Penalty

- Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

12. Rebuttal period

- You will be given a period a week (7 days) to read and respond to the comments and grades of your homework or project assignment. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.