Overview
You are to write and test a C/C++ program that simulates a simple operating system. This simulated OS kernel (s_osk) should be able to handle a small set of simulated system calls related to process management. Some mechanisms (e.g., FIFOs) of the native OS (Linux) will be used to implement some of the basic OS features (e.g., system calls) in this simulated OS. Key Unix mechanisms used in this assignment include:

- Process creation, termination and synchronization.
- Inter-process communication using FIFOs.
- Signals and signal handlers.

These mechanisms will be used to implement a simulated OS kernel (s_osk) and run-time library (s_runlib). Features to be simulated in the implementation include:

- s_begin: Process startup.
- s_spawn: Process creation.
- s_end: Process termination.
- s_yield: Voluntary context switching.
- s_getuptime: Get system clock value.
- s_gettid: Get my task id.
- s_getptid: Get my parent’s task id.

The syntax and semantics of these simulated system calls are described below.

Conceptual Model
In this simulated system, a task is a simulated process. There is one system task (s_osk) and zero or more user tasks. The system task controls the creation of all user tasks. After s_osk initializes itself, it creates one task using the standard Unix fork-exec*.\(^1\) This one user task may directly or indirectly create other user tasks via the s_spawn simulated system call.

Note that since this is a simulated OS running on top of Linux, it is the passage of virtual time that is important to this system. Except for one special situation (the timeout alarm described below), any real time that passes because of any other users outside of the simulated environment is ignored by this system.

For the most part, the system is non-preemptive. That is, a task retains control of the CPU (in virtual time) unless it makes a simulated system call and voluntarily gives up control. The only exception to this rule is the timeout alarm. A timeout alarm is set to insure that your program

\(^{1}\)exec* is an abbreviation for all of the different forms of the exec system call; e.g., exec, execv.
does not run on forever. An alarm should be sounded \textit{alarmTime} seconds of real time after the start of your s.osk. When this alarm occurs, all simulated tasks must be terminated by s.osk. This alarm is a safety feature to prevent buggy software from continuing to run.

The OS kernel s.osk has a number of configuration options described below. s.osk terminates when either the timeout alarm has expired or when all user tasks have terminated. Optionally, s.osk should have the capability for displaying the CPU usage of its tasks, and the termination time of the last \textit{termHistory} user tasks to terminate.

Since the simulation is for a single CPU system, there should be only one active thread of control among all of the simulated tasks. Suppose a user task is the following:

\begin{verbatim}
int main (int argc, char *argv[]) {
  s_begin(...);  // local initialization
  tid = s_spawn(...);  // create a child whose task id is tid
  ... compute A ...
  s_yield();       // yield the cpu to another task
  ... compute B ...
  s_end;           // quit
}
\end{verbatim}

The two lines marked "compute A" and "compute B", will execute without interruption. s.yield gives up control of the CPU to another process. This task should be placed on the back of the RUN queue but will not get control back until it reaches the front of the RUN queue.

\textbf{System Calls}

The syntax and semantics of the simulated system calls are shown below. You may deviate from the interface below, but you should give a good argument for any deviations.

\begin{itemize}
  \item void s\_begin(void); // Process startup
    
    This performs initialization and is required because of the cooperation between the user task and the system task described below in the Implementation Notes.
  \item tid\_d tid = s\_spawn(const char *path, const char *arg, ...); // Process creation
    
    This is like a combined fork-exec. The return value is the task ID. Note that the task ID is a non-negative integer and is not necessarily the Linux process ID since the task ID is used to quickly locate the attributes of the task. Also note that all user tasks are initiated by s.osk; i.e., s.osk calls fork-exec*, NOT a user task.
  \item void s\_end(int status): Process termination.
    
    Terminate the task and return \textit{status} as the exit status.
  \item void s\_yield(void): Voluntary context switching.
    
    Give up control of the CPU. The task remains runnable.
  \item void s\_getuptime(struct timeval *clock): Get the value of the system clock.
    
    The system clock is of type 'struct timeval'. \textit{clock} is an OUT variable.
  \item tid\_d tid = s\_gettid(void): Get my task id.
\end{itemize}
• tid \_ ptid = s\_getptid(void): Get my parent’s task id.

**Implementation Notes**

In this implementation, you can assume that the system will be small and therefore simple data structures are appropriate (i.e., there is no need at this time for sophisticated data structures). However, s\_osk will need one or more data structures to keep track of user tasks. It is up to you to determine what you will need, but remember that simplicity will be a virtue in this assignment. Below is a summary of implementation features which are then discussed more fully.

- Task spawning (s\_spawn) should be implemented as a combined fork-excl call. Note that the first user task should be passed into s\_osk through s\_osk’s command line arguments.

- A system call should be implemented as a request message sent to the well-known FIFO sys\_callFifo which is read by s\_osk. The corresponding return from the system call should be a reply message sent from s\_osk to the user task. We will adopt the convention that this reply will be sent to the reply FIFO with the name replyFifoNNNNN where NNNNN is the Linux process ID of the user task. The reply FIFOs should be opened read-only by the user task and write-only by the system task. The request FIFO sys\_callFifo should be opened write-only by the user tasks and read-only by the system task. To insure low overhead, messages to and from the FIFOs should be sent and received using the unbuffered I/O system calls write and read, not their buffered counterparts. **In all cases, the open() call should be a blocking call.**

- The system clock should record the passage of virtual time. Since this is a non-preemptive system, s\_osk can only determine the passage of virtual time by getting that information from the user tasks. One approach is to include the amount of virtual time consumed by a user task in the s\_exit system call. s\_exit can get this information by calling getrusage(2). Note that if the timeout alarm expires, we still want to be able to record the passage of virtual time and cleanly shut the system down.

The creation of and accessing of the FIFOs can be tricky. One common problem is deadlock. Since all open calls are blocking, it is possible that each process will end up waiting for each other in a circular wait. So, it is important in the the startup sequence to avoid this situation. A second problem is that the user task’s startup routine s\_begin() creates the reply FIFO and uses the process id in the name. This means that s\_osk can not open the reply FIFO until the user task startup routine tells s\_osk the name of the reply FIFO.

You will need to define the format of the simulated system call request messages and system call reply messages. Note that since these messages can be variable length, you will need a protocol that can handle this situation. One approach is to use a message format such as (n, pid, opcode, message body) where n is the message length in bytes, pid is the process id of the sender, opcode is the request/reply, and message body is the remainder of the message.

**Additional Guidelines**

- Code readability is of the utmost importance. The Web page will contain a summary of coding guidelines that you should follow in spirit. I am not rigid about these guidelines, but unreadable code will be penalized.

- All actual system calls should be wrapped so that any fatal errors will cause an error message to be displayed followed by an exit. I will not examine any solutions that do not follow this
rule. By convention, the wrapped system call name will be the same as the actual system call name except the first character should be capitalized (e.g., Fork is the wrapped version of fork).

- **s_osk** should have the following command-line options:
  - `-a alarmTime`: Set the maximum number of seconds of real time s_osk should be allowed to run before terminating. The default value is 10 seconds.
  - `-v verboseLevel`: Set s_osk into verbose mode at level `verboseLevel`. Normally, s_osk runs in silent mode where it outputs nothing until it terminates when it outputs to stdout the value of the system clock, and the number of user tasks that have been spawned and the number that have terminated. This corresponds to verbose level 0. In verbose level 1, s_osk displays a message on stdout whenever a user task is created and when it terminates indicating the real time, task id, the system call type. In the case of spawn, the s_spawn arguments are displayed. In the case of termination, the s_exit arguments are displayed. In verbose level 5, every simulated system call message received by s_osk is displayed on stdout. The default value is 1.
  - `-m maxTasks`: Set the maximum number of concurrent tasks. The default value is 10.

The command-line is:

```
s_osk [-a alarmTime] [-v verboseLevel] [-m maxTasks] arg0 arg1 arg2 ...
```

The sequence of arguments arg0, arg1, arg2, ... are the arguments of the first user task. Note that arg0 should be the simulated user task executable itself and can be assumed to be in the current directory.

- **s_osk** should create the well-known FIFO `syscallFifo` and read requests from it.

- All processes should quit cleanly. For example, remove all FIFOs when they are no longer needed. No task should be hung after the server quits.

- The FIFOs should only be opened once by each process.

- The Web page will contain a basic Makefile and the source listing of a sample user task.

**What to Submit**

There are two submission dates:

1) On Friday at 1 PM, October 17, submit four items: 1) A description of the message formats; 2) a description of the message protocol indicating the sequence of operations for creating and initializing the FIFOs; 3) a description of the s_osk data structures; and 4) a source listing and output indicating whether one simple user task can successfully start; i.e., the user task can create its reply FIFO and send a message to s_osk and s_osk can open the reply FIFO and send a message back to the user task. If the code can not accomplish this, describe your progress or remaining hurdles towards meeting this goal. The message formats and data structures can be COMMENTED structures or class definitions.

2) On Friday at 1 PM, October 24, submit the hardcopy and electronic items described below.
The CS422S Web page contains a link to the documentation template. You should complete the template and submit it in both hardcopy AND electronic form. Submit the completed documentation template AND a listing of the source code. The electronic submission (described below) should include the completed documentation template, the source code, the Makefile, test scripts, and test output.

Words of Caution
Here are some observations from my years of experience with projects like this:

- **The documentation template is non-trivial.** Do not expect to complete it in less than an hour. Furthermore, working code but no documentation is almost useless. So, don't forget to fill out the documentation template.

- Trivial bugs can consume tens of hours of time. Yes, I said tens of hours, not just hours. You need to start small, test the control structure, and incrementally add features on top of what seems like rock solid code.

- Keep different versions of your "rock solid code versions" so that you can rollback to and recover from a stable version. This also helps if you mistakenly delete your latest source code!!!

- Don’t ignore error messages and think they will disappear on their own. They don’t. They just come back and bite you when you least want to be bitten.

- Try to understand the origins of your bugs rather than always doing trial and error changes. (Some trial and error may be appropriate in small test cases.)

- If the approach you are taking seems like it will be a nightmare to implement, then don’t implement it. Find a better way or better understand the system calls you are trying to use.

- The more lines of code you have, the more chances for bugs to appear.

- Have a plan. Don’t try to do everything at once.

Demonstration
You must demonstrate your program to a grader/consultant. Details of this demonstration will be provided in class and on the course Web page. Also, see the section Late Policy.

Electronic Submission
The end result should be that you mail to kenw@arl.wustl.edu a single shar (shell archive) file containing your files. Do NOT submit object code or executables. The following commands will create a shar file named A.shar containing the files Makefile, demo1, demo2, ... , s_osk.c, s_user1.c, s_user2.c and other files and then send mail to me:

```
shar Makefile demo1 demo2 ... s\_osk.c s\_user1.c s\_user2.c ... > A.shar
mail kenw@arl.wustl.edu < A.shar    # mail is usually in /bin
```

The files demo* are shells scripts that demonstrate that the workings of your program. I will supply some of these scripts, but you should include your own as described in the documentation template. I should be able to enter the following to unpack your shar file, compile your code (using gcc, or g++), run s_osk for the default values, and run the demonstration scripts.
sh A.shar
make compile     # compile the client and server code
make run1       # run demo1
...            # run other demonstrations

Late Policy
You can submit this programming assignment one week late for a 20% penalty. Note that you should submit something even if the final version still has bugs.