CS170 Lab 2 - Multiprogramming with KOS

sites.cs.ucsb.edu/~rich/class/cs170/labs/kos_mp/index.html

CS170 KOS

- CS170 Lab Assignment 2 -- Processes and Multiprogramming
- Rich Wolski and James Plank
- URL: http://www.cs.ucsb.edu/~rich/class/cs170/labs/kos_mp
- Directory: **/cs/faculty/rich/public_html/class/cs170/labs/kos_mp**
- Culinary advice from a previous chef
- An example Makefile that will work for submitting your solution
- You must include simulator_lab2.h to pick up the right set of simulator defines for this lab.
- A C function for printing the intial stack: printstack.c
- **Due: Wednesday Nov 16th, 2022 at 11:59 PM**

Lab 2 -- Simple t

In this lab, you will ad Notice the use of the word "add" in the p orking well, you may find yourself tuni nt works in this way. Each piece you c lf spending time on *Lab 1* parts to mak eality.
Specifically, you will d d you will allow up to eight user processe a timer that you can use to perform tim

Also, you will be able hat make use of **malloc()** and the stan

When you are done, y lls:

- **ioctl()** (in limited form)
- **fstat()** (in limited form)
- **getpagesize()**
- **sbrk()**
- **execve()**
- **getpid()**
- **fork()**
- **getppid()**
- **wait()**
- **getdtablesize()**
- **close()** (in limited form)

Moreover, you'll be able to execute a shell and use it to run user programs (although file redirection and pipes won't work).

Changes to the simulator

- For this lab, you need to include the version of **simulator_lab2.h** from this directory. Note that this is different from the **simulator.h** in lab 2. You will need to link your code with **/cs/faculty/rich/cs170/lib/main_lab2.o** and **/cs/faculty/rich/cs170/lib/libsim.a**. There is a Makefile in this directory.

- For this lab, there are two special simulator variables:

  ```
  extern int User_base;
  extern int User_limit;
  ```

  These define what **KOS** addresses the user sees as its memory. When a user job is executing, its ad and it can use up to **User_Limit** b mory that it has allocated for use r process cannot access any mem nd **(User_Base + User_Limit)**. Ot

- **User_Base** and user_program()**. It now loads the sp ory + User_Base)**, and will fail if the s. It returns -1 on a failure. On suc u can use this as the end of the he

- There is now a t start_timer(int ticks)**, which tel every **ticks** ticks of the clock. You default is no timer.

Your Job in this

Basically, we want to do four things in this lab:

1. Allow the user to compile and run programs that use the standard I/O library and **malloc()**.
2. Implement the parts of KOS that enable a simple shell to work: **fork()**, **execve()** and **wait()**.
3. Implement process id's.
4. Make time-slicing work.

I will describe each of these in turn.

Allow the user to compile and run programs that use the standard I/O library and malloc()

As before, set up **KOS** so that it loads the program **a.out** as its user process when it starts up. What we first want to do is allow these **a.out** programs to use the standard I/O library and **malloc()**. To do this, we need to do some busy work and implement some system calls that normally we would not care about. The first few are: **getpagesize()**, **getdtablesize()**, and a simple **close()**. **Getpagesize()** should return the value of the **PageSize** variable in **simulator.h**. **Getdtablesize()** should return 64. **Close()** should be implemented so that it returns an error (that's not really how to implement **close()**, but it will suffice for this lab).
We also have to implement one case of **ioctl()** and one case of **fstat()**. Read the man page on **ioctl()**. I don't expect you to understand much about **ioctl()** except for its syntax. You are going to need to implement **ioctl()** when the first argument is 1 and the second argument is **JOS_TCGETP**. Your job is to fill in the third argument which is a pointer to a **(struct JOStermios)**. You do this by calling **ioctl_console_fill(char *addr)**, where **addr** is the KOS address of the third argument. Then return zero. This is probably confusing, but it must be done.

Fstat() is called by the **uld buffer I/O on** file descriptors 0, 1 an **stat_buf_fill(char *ad** **s of the stat struct** that the user passed t **that the file** descriptor should allo **file descriptors** one and two, try 256.

Last, you must implem **detail. As before,** the stack and the hea **k()** should not be allowed to increase pa

When you have imple **ms should be** allowed to use the sta **c()**). Look at the programs in

/cs/faculty/rich/cs17

, and try ones like **hw2.c** that use **printf()**. You can write, compile and run your own programs too -- just follow the compilation steps in Makefile.xcomp and the instructions from lab 1.

Implement the parts of KOS that enable a simple shell to work: fork(), execve() and wait()

This means you must do multiprogramming. In the last lab, you allowed one user program to execute and gave it all of memory. This time, you should divide memory into 8 equal parts and when you create a new process, it will use one of those parts. Context switching will now involve saving/restoring the registers, **User_base** and **User_limit**.

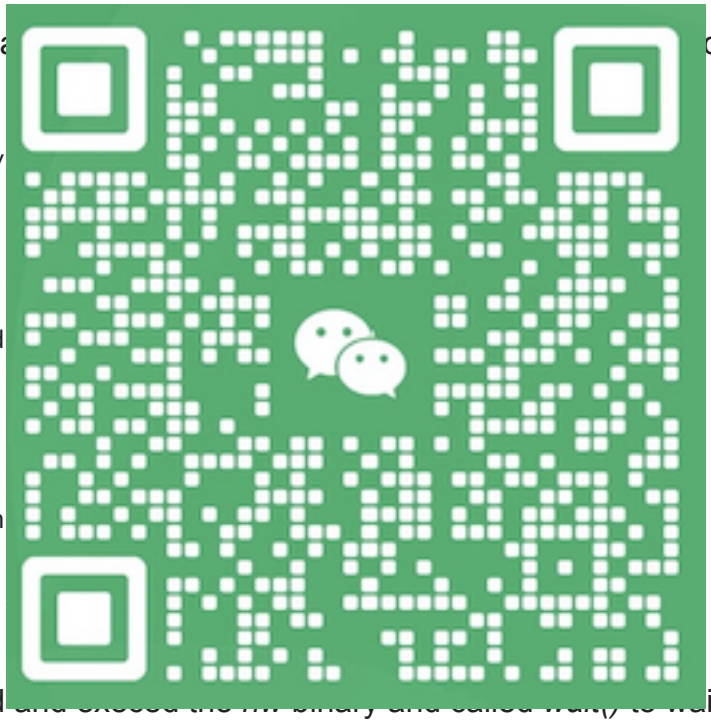
You'll need to modify the **exit()** system call so that instead of halting **KOS**, it simply terminates a process. **Fork()** and **execve()** are rather straightforward. You should ignore the third argument to **execve()** -- we won't have any environment variables. You also must implement **wait()** and process id's, which should work as in Unix. In particular, processes have parents; otherwise they are orphans. If a process exits and its parent hasn't called **wait()** it becomes a zombie -- it releases its memory but maintains a PCB until its parent either dies or calls **wait()**. Orphans must do the correct thing when they die.

To test this, run

/cs/faculty/rich/cs170/test_execs/ksh

in your version of kos.

WARNING: This is a simple shell with few features. You do not need to use it and can instead, write your own test codes.

The *ksh* binary is capa  can run hw from ksh thus:

```
./kos -a /cs/faculty
```

and you should see

```
KOS booting... done.
Probing console... d
ksh:
```

now run *hw*

```
ksh:/cs/faculty/rich
Hello world
the write statement
ksh:
```

The *ksh* binary forked and exec'ed the *hw* binary and called *wait()* to wait for it to complete before printing the *ksh:* prompt again.
This binary has few features and many bugs. You can use it to test fork, exec, and wait or you can write your own test codes. If it doesn't work for you, then you must write your own test codes (which is a good idea anyway). The TAs will not be able to answer questions about *ksh* -- either it works for you and you trust it or it doesn't and/or you don't in which case write your own testers.

Implement process id's

Finally, you need to implement **getpid()** and **getppid()** to work with your process id's. This is straightforward. I have no process with process id 0. Orphans return pid 0 as their parent.

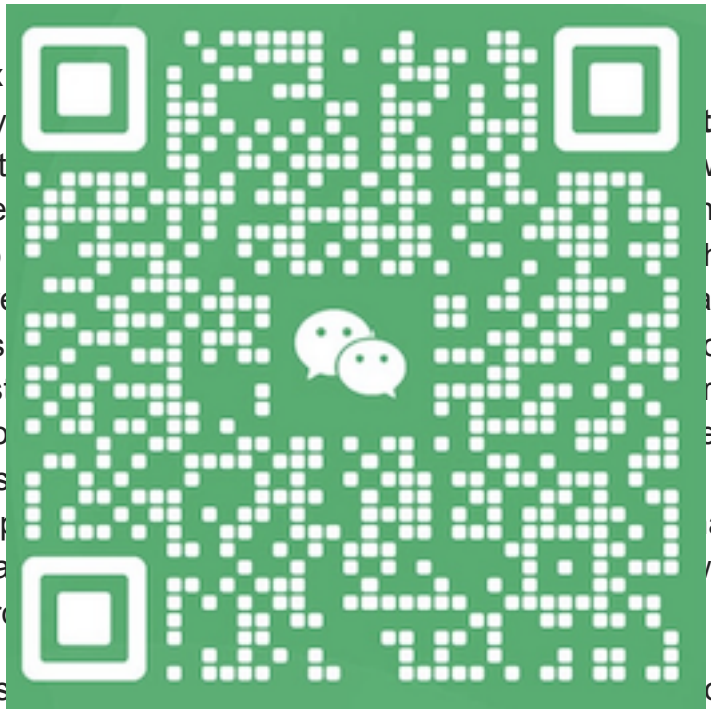
Make time-slicing work

This is straightforward -- set the timer for some number of ticks, and then reschedule the CPU when you get the interrupt.
As in the last lab, there is a cook book of things that you might do to get this lab working. As before, it is not mandatory that you do things the same way that I did. However, it may make your life easier.

A Word About System Calls

In this lab, you need to implement working or almost working versions of several system calls. If you do not know what a system call is, don't panic. I'll tell you. When you want the operating system to do something for you, you make a system call. The arguments for the system call are documented. For example, try the command

```
man getpid
```



on your favorite Linux files (for the types) and the prototy **tpid()** on a Linux system, the program t v to service the call, and return the ne nenting the KOS kernel so you have to he ones that are specified in the lab are ab gives you some details on what they s ok at the man page on the CSIL sys re, and what their return values are. In o e same thing that the "real" system calls
There are some excep ave to do with the way things like the sta ill tell you what values to return regar

Thus, if you find yours o the TAs, but also consult your local man page for details.

File Modularity for Submission

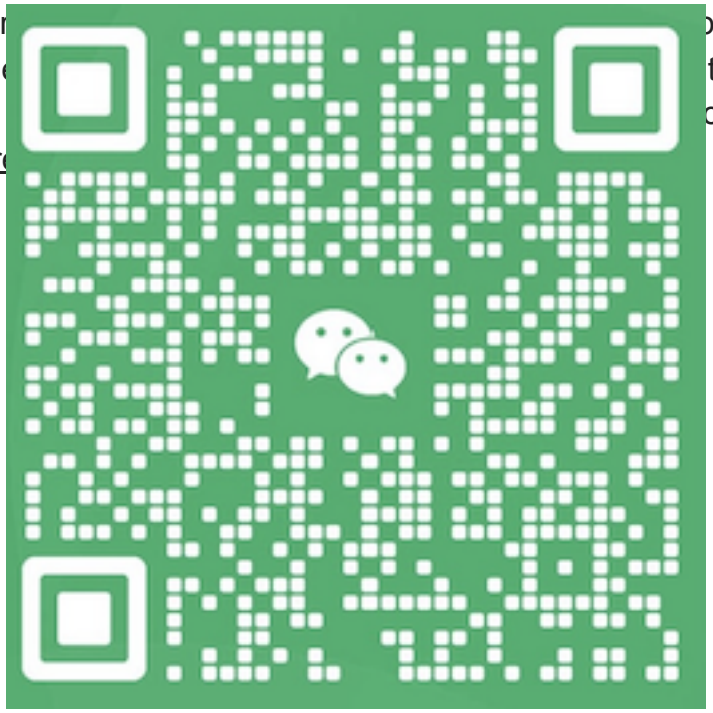
For this lab, your solution needs to conform to the following file modularity:

- console_buf.c
- console_buf.h
- exception.c
- kos.c
- kos.h
- memory.c
- memory.h
- scheduler.c

- scheduler.h
- syscall.c
- syscall.h

The submission acceptance program will be looking for these files. You should design your solution so that it is contained in files with these names.
This requirement has a few ramifications. First, this list is the file list -- not a subset of the file list. Thus, you can't include extra files and then use "#include" to include those files in these file names. The submission program will use these files and a prepared Makefile to build your program. It will not be prepared for hidden dependencies introduced by extra included files.

This requirement also means that you simply can't concatenate a bunch of files that contain your solution to create one of more of these files. The concatenation is likely to cause double definitions as header files are included multiple times which will cause your compile to fail.

Header files and libra  been using in class (like libfdr, kt.h, the class makefile use. Thus you should or use the example included here