Homework 1
COM S 352
Fall 2021

1. (15 points) For each state transition below, state whether there is a context switch involved. If there is a context switch, describe the processes involved. How are they chosen?

a. ready -> running
b. running -> blocked
c. blocked -> ready

2. (15 points) Three programs are serviced in a multiprogramming system. Program A contains 50ms of computation followed by 100ms of I/O on hardware device 1. Program B contains 20ms of computation followed by 50ms of I/O on hardware device 2. Program C contains 50ms of computation followed by 100ms I/O on hardware device 2. Each device can service only one I/O request at a time. What is the minimum time it will take to complete all three programs? Create a table like Figure 4.4 in the reading to show the operation of the three programs.

3. (15 points) Describe the purpose of each of the following terms individually:

- system call
- interrupt
- trap

Now, describe how the three are related to each other.

4. (15 points) The RISC-V architecture that we will use in examples throughout this class allow processors to be implemented with three privilege levels defined as follows (http://docs.keystone-enclave.org/en/latest/Getting-Started/How-Keystone-Works/RISC-V-Background.html):

> *Privilege level defines what the running software can do during its execution. Common usage of each privilege level is as follows:*
>
> - *U-mode: user processes*
> - *S-mode: kernel (including kernel modules and device drivers), hypervisor*
> - *M-mode: bootloader, firmware*

However, the specification allows simple embedded processors to implement just M-mode (i.e., all code on these simple processor implementations must run at the same privilege level).

What is the purpose of multiple privilege levels? Compare and contrast multiple privilege levels vs a single privilege level, name at least one advantage for each.

5. (40 points) Programming Challenge - Extending XV6 grep

Read the following description on pipes and complete the short programming exercise.

Pipes On the Command Line

One facet of an operating system is the utilities it provides. The UNIX design approach is to separate utilities into small independent programs which can be combined to form more complex behavior.

An example of a utility is `grep`, it searches a plain text file for matches to a regular expression. For example, we can search for the word "xv6" in a file named README with the following command:

```
$ grep xv6 README
```

As output, `grep` prints every line that matches the expression. For example:

```
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
The code in the files that constitute xv6 is
(kaashoek,rtm@mit.edu). The main purpose of xv6 is as a teaching
```

More complex behavior can be created by connecting the output of one program into the input of another, this is known as piping. Suppose we wanted to print the line number of each match found. The command `cat -n` will print a file with line numbers. If we take the output of the cat command and pipe it into the input of `grep` we get our desired behavior.

```
$ cat -n README | grep xv6

     1 xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
     2 Version 6 (v6).  xv6 loosely follows the structure and style of v6,
     7 xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
    30 The code in the files that constitute xv6 is
    36 (kaashoek,rtm@mit.edu). The main purpose of xv6 is as a teaching
```

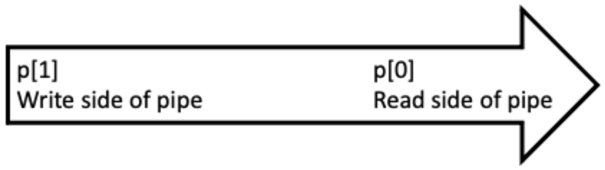
Here the line numbers are being added to every line in the file by the `cat` command, then the lines are being piped (the `|` symbol) from the standard out of `cat` into the standard in of `grep`.

Pipes In Code

Let's see how the XV6 shell (sh.c) implements the pipe symbol described above. When the shell finds a pipe symbol | it calls `fork` twice to create two children, one to execute the program on the left (e.g., `cat`) and one to execute the program on the right (e.g., `grep`). Before that, `pipe(p)` is called to create a pipe that will connect the two processes. Here is the code:

```
  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
      panic("pipe");
    if(fork1() == 0){
      close(1);
      dup(p[1]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->left);
    }
    if(fork1() == 0){
      close(0);
      dup(p[0]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->right);
    }
    close(p[0]);
    close(p[1]);
    wait(0);
    wait(0);
    break;
```

The pipe struct is an array of two file descriptors, `p[0]` represents the read end of the pipe and `p[1]` represents the write end of the pipe. A file descriptor is an id of something that can be read or written, it may be a file, but it can also be the ends of a pipe.



Another thing that can be read or written is standard out (e.g., printf) and in (e.g., getchar), they have specially assigned file descriptors. Standard out is 1 and standard in is 0. For example:

```
printf("Enter a letter:"); // the string is written to file descriptor 1
c = getchar(); // the input is read from file descriptor 0
```

The goal is to connect the standard out of the first process to the write side of the pipe and the standard in of the second process to the read side of the pipe. Because `pipe()` is called before the forks, both processes have a copy of the same pipe in their memories.

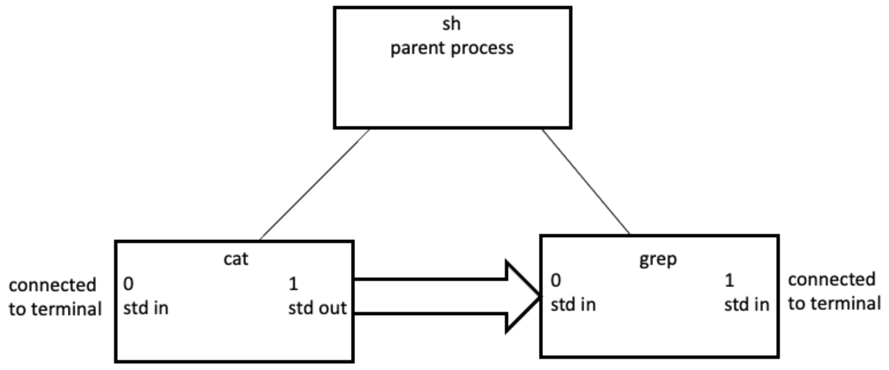
Here is an explanation for the first process.

```
// close standard out
close(1);
// replace the file descriptor on the write side of the pipe with the
// recently closed file descriptor 1
dup(p[1]);
// close the no longer used file descriptors of the pipe
close(p[0]);
close(p[1]);
// runcmd calls exec to execute the program on the left side (e.g., cat)
runcmd(pcmd->left);
```

Here is an explanation for the second process.

```
// close standard in
close(0);
// replace the file descriptor on the read side of the pipe with the
// recently closed file descriptor 0
dup(p[0]);
// close the no longer used file descriptors of the pipe
close(p[0]);
close(p[1]);
// runcmd calls exec to execute the program on the right side (e.g., grep)
runcmd(pcmd->left);
```

Finally, we have achieved something that looks like this.



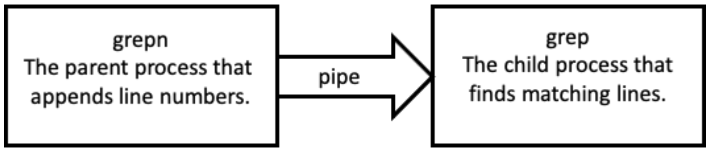
Problem

As it turns out, `grep` also has a `-n` flag in GNU Linux, so in this case we could have achieved similar output without a pipe:

```
$ grep -n xv6 README
```

However, the version of `grep` implemented for xv6 is very limited, it has is no `-n` flag.

Create a new version of `grep` called `grepn` that always prints line numbers. Instead of copying and modifying the existing `grep`, you must use pipes in a way that is similar to the command `cat -n README | grep xv6` shown above. However, you will not write a command in a shell, you will fork a new process to execute `grep` and connect a pipe to it in C code. Here is a graphical representation.


grepn
The parent process that appends line numbers.
pipe
grep
The child process that finds matching lines.

In the figure above you are writing the code only for `grepn`. It needs to create a child process that runs `grep` and communicates with the child using a pipe.

Complete the following. **Important: you are allowed to copy any code from the `grep()` and `main()` methods in `grep.c`. You are not allowed to use the `match()` code in `grep.c`, your implementation must demonstrate creating and piping to a separate `grep` process that performs the text search.**

1. First, setup the new utility in the user folder. Create the file `grepn.c` in `user`. In the file `Makefile` search for `UPROGS` and add the line: `$U/_grepn\`

2. The `main` method of `grepn` can be copied from `grep`.
```
int
main(int argc, char *argv[])
{
  int fd, i;
  char *pattern;

  if(argc <= 1){
    fprintf(2, "usage: grepn pattern [file ...]\n");
    exit(1);
  }
  pattern = argv[1];

  if(argc <= 2){
    grep(pattern, 0);
    exit(0);
  }

  for(i = 2; i < argc; i++){
    if((fd = open(argv[i], 0)) < 0){
      printf("grepn: cannot open %s\n", argv[i]);
      exit(1);
    }
    grep(pattern, fd);
    close(fd);
  }
  exit(0);
}
```

3. Create a pipe that will be used to communicate with the standard in of the `grep` process.

4. Use `fork` to create a child process.

5. Use `dup` to direct the standard out of the parent into the standard in of the child.

6. Close all unused file descriptors.

7. Use `exec` to run `grep` in the child process. Pass the required arguments (i.e., the regular expression pattern).

8. Implement the `grep()` method. Again, you can copy relevant parts of the `grep()` method from `grep`.
   a. Fill a buffer by reading from the file descriptor (`fd`).
   b. Search for occurrences of new line `"\n"`.
   c. Every time a new line is found, `printf()` the line number followed by the line itself. If the pipe is configured as described above, the `printf()` should be directed into the `grep` process.
   d. When the end of the buffer is reached, shift everything past the last new line over to make room and loop back to step *a*.

Test your program.

```
$ grepn xv6 README

 1 xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
 2 Version 6 (v6).  xv6 loosely follows the structure and style of v6,
 7 xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
30 The code in the files that constitute xv6 is
36 (kaashoek,rtm@mit.edu). The main purpose of xv6 is as a teaching
```

What to turn in?

For questions 1-4 submit a typewritten document (pdf preferred). Handwritten answers will be rejected. For problem 5, submit only `grepn.c`. Document your code. Attach both to the Canvas submission.