CSI 3131
Assignment 1
Deadline: 23 May 2021

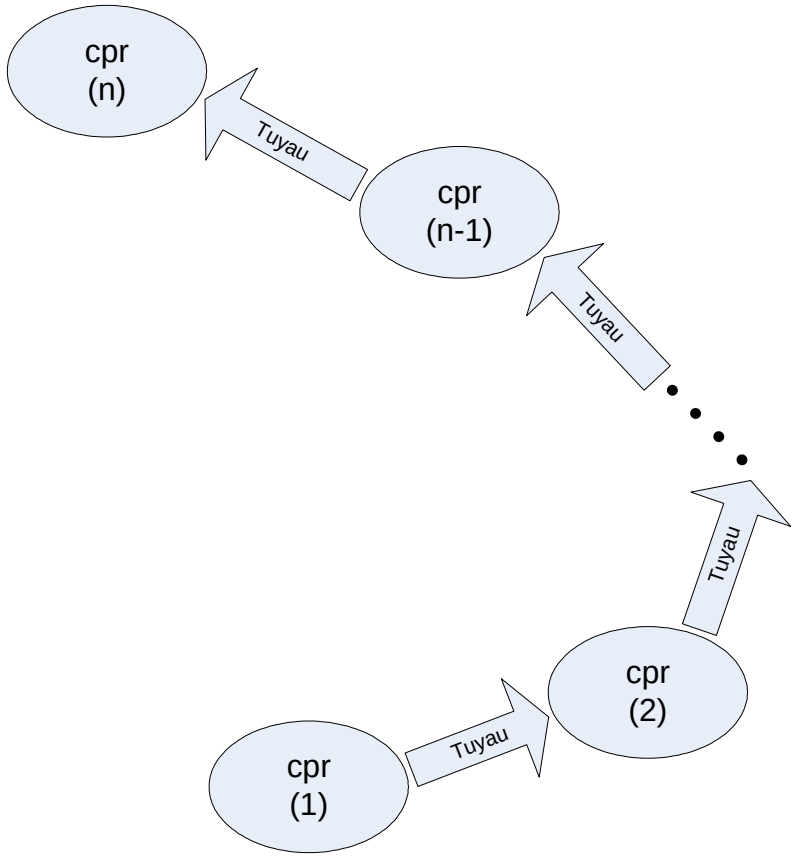
Description

You will have to complete a *cpr* program (create process) which will have to create a child process. The child process in turn will create a child and so on to create n processes. The *cpr.c* file is provided to you and you must complete it.

The command to create the processes demonstrated below is "**cpr num**" where num is the number *n* of total processes to be created (ie *n-1* children). The first created process (labeled *n*) is created by running the command. This first process creates a child by executing the command "**cpr num-1**", that is, decrements its argument by 1 and creates its child with the new value. So the child will create another child by decrementing its argument. When a process receives an argument with the value 1, it will not create a child.

When a process creates a child, it must first create a pipe and attach the writing end of the pipe to the standard output of the child before executing the "**cpr num-1**" command. So all child processes (ie process 1 through *n-1*) write to pipes by writing to their standard output.

All processes that create children (ie process 2 through n) read from the read end of the pipe and write any data read to their standard output. So any output written to the pipes eventually appears on the screen (via process n) after passing through one or more pipes. Note that you do not attach the reading ends of the pipes to the standard inlets (although this is possible).



Actions taken by the processes include the following:

- Process 1: simply writes to its standard output "Process 1 begins", waits 5 seconds (using the *sleep (5)* call), and then writes to its standard output "Process 1 terminates".
- Process 2 to n: Creates a child as described above, also writes to its standard output the same messages as Process 1 (by substituting the number 1 with the value of the argument received by the process, 2..n) as well as read from the read end of the pipe to write the read data to its standard output. You must write the read messages and data in the order necessary to make the messages of all the processes appear in the following order (in this example 5 processes have been created, ie *cpr 5* is executed)

  Process 5 begins
  Process 4 begins
  Process 3 begins
  Process 2 begins
  Process 1 begins
  Process 1 ends
  Process 2 ends
  Process 3 ends
  Process 4 ends
  Process 5 ends

There should be a 5 second delay between the "Process 1 Begins" and "Process 1 Ends" messages. Note that a parent does not execute the *wait* call because it knows that its child is done when the pipe write end is closed (it can no longer read the pipe read end).

To complete the assignment:

1. Start with the provided *cpr.c* file. Complete the documentation to indicate your name and student number. Take the time to properly document your code. Also take the time to properly design the solution to the problem before coding.
2. Complete the *createChildAndRead (int prcNum)* function.
3. Submit your assignment by downloading the *cpr.c* file. Remember to click on the submit button after (and only after) uploading the file.
4. A word of warning, if you want to write messages to the terminal during process creation, write to the standard error output with the following call: *fprintf (stderr, "message \ n")*.
5. Insert a delay of 10 seconds (with *sleep (10))* before the termination of the processes to introduce a delay after having written the message "Process terminated". During the timeout for each process, observe the status of the processes with the command "ps –u test1" (replace test1 with your username if you are using one of SITE's Linux machines to do your homework). You will notice that a process is a zombie. Explain this observation. Add your explanation as a comment at the beginning of your source code where indicated.

Useful information:

1. A file descriptor is an integer, which serves as a handle to identify an open file. This descriptor is used with other functions or calling systems such as *read ()* and *write ()* to do I/O operations with the corresponding file.

2. In UNIX / Linux, each process when it is created includes 3 open file descriptors:
   a. Standard input, identified with descriptor 0, so read (0, *buf*, 4) will read 4 bytes of standard input and copy them to the *buf* buffer. Normally, the standard input for a program is the terminal keyboard.
   b. The standard output is identified with descriptor 1, and normally corresponds to the terminal screen.
   c. Standard error output is identified with descriptor 2, and normally corresponds to the terminal screen.
   d. In fact, when a process is started from a command typed in the shell, standard input, standard output, and standard error output are three descriptors that refer to the terminal (tty file) of the shell. So reading from the tty file corresponds to reading the keyboard while writing to the tty file corresponds to writing to the terminal screen.
   e. Note that several functions of the standard C library use these descriptors by default. For example, *printf ("string")* writes "string" to standard output (normally on the screen).
   f. It is possible to ask the UNIX shell to pipe the standard output of one process to the standard input of the other process. This involves inserting the character "| "Between two commands: for example" who | wc ". In this example, the shell will create a UNIX pipe that will connect the standard output of the who process to the standard input of the wc process. So the data written by the who process using

descriptor 1 (standard output) is sent to the pipe. This data will be read by the wc process which binds its data using descriptor 0 (standard input), that is, from pipe.

3. You will use the following C functions:
   a. fork () - study in class (for more detailed information, use the man fork command)
   b. smoking pipe()
      - Also introduced in class.
      - Note that it is possible to attach multiple processes to each end of a pipe. The SE holds the pipe, as long as processes are attached to the ends of the pipe.
   c. execvp (const char * program, const char * args []) (other forms of this call exist such as execlp (const char * program, const char * program,….))
      - Replaces the process image with the program specified in the first argument of the function.
      - The second argument is an array of strings which will be given to the new program as arguments (the array is terminated with NULL).
      - A convention dictates that args [0] is the file name of the program to be executed.

   d. dup2 (int oldfd, int newfd) - clones (duplicate) the oldfd descriptor on the newfd file descriptor. If newfd matches an open file, that file will be closed first. So the same file can be accessed either by oldfd or by newfd (basically two connections to the file). Seehttp://mkssoftware.com/docs/man3/dup2.3.asp or "man dup2" for more information. For example, the following program:

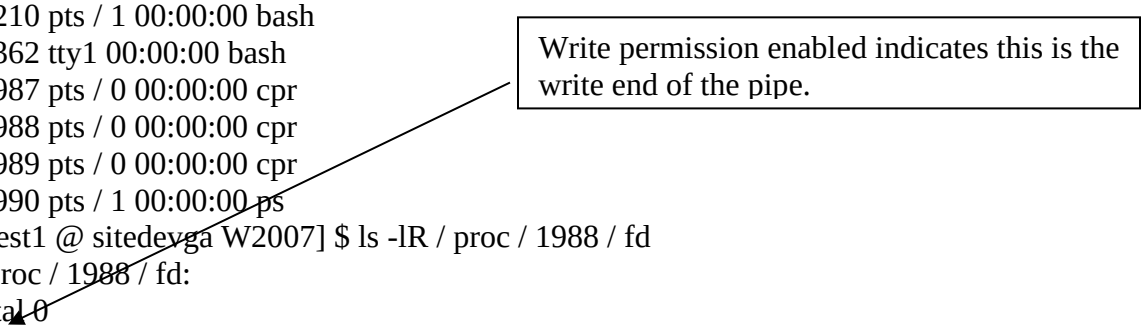
```
int main (int argc, char * argv []) {
 int fd;
 printf ("Hello world!")
 fd = open ("outFile.dat", "w");
 if (fd! = -1) dup2 (fd, 1);
        printf ("Hello world!");
close (fd);
}
```

   will write the standard output of the program to the file "outFile.dat". The first "Hello world! Will be printed on the console, while the second will be stored in the file "outFile.dat".
   e. *read (int fd, char * buff, int bufSize)*- read from the file (or pipe) identified by the descriptor fd a number of bufSize characters and copy the characters read into the buffer buff. The function returns the number of characters read (can be less than bufSize), or -1 during an error, or 0 when the end of the file is reached (in the case of the pipe, no process is attached to the writing end and no data exists in the pipe).
   f. *write (int fd, char * buff, int buffSize)* - Writes in the file (or pipe) fd the number of buffSize characters to find in the buff buffer.
   g. close (int fd) - close an open file. The descriptor fd can be reused.
   h. You will probably want to use the printf () function to format your output. This function writes to standard output (df 1). But be careful because this function

buffers the output data and does not immediately write to standard output (does a write at df 1). To force an immediate write, use fflush (stdout). Another alternative would be to use sprintf () to format the output to a memory buffer and then use write () to write the data in the buffer to standard output.

4. Here is a clue how to observe process references to pipes:
   a. Add a long delay (say 300 seconds) at different points in your code. When you run your program, various processes will be created which you can examine through the / proc directory. Obtain the PIDs (process identifiers) of the processes created with the command "ps –u test1" (replace test1 with your username if you are using one of SITE's Linux machines to do your homework). You can find what the different file descriptors refer to by looking at the process's fd directory as follows:

```
[ test1 @ sitedevga W2007] $ ps -u test1
 PID TTY TIME CMD
1114? 00:00:08 sshd
 1115 pts / 0 00:00:00 bash
 1210 pts / 1 00:00:00 bash
 1362 tty1 00:00:00 bash
 1987 pts / 0 00:00:00 cpr
 1988 pts / 0 00:00:00 cpr
 1989 pts / 0 00:00:00 cpr
 1990 pts / 1 00:00:00 ps
[ test1 @ sitedevga W2007] $ ls -lR / proc / 1988 / fd
/ proc / 1988 / fd:
total 0
lrwx ------ 1 test1 test1 64 Jan 18 11:50 0 -> / dev / pts / 0
l-wx ------ 1 test1 test1 64 Jan 18 11:50 1 -> pipe: [11950]
lrwx ------ 1 test1 test1 64 Jan 18 11:50 2 -> / dev / pts / 0
lr-x ------ 1 test1 test1 64 Jan 18 11:50 3 -> pipe: [11951]
```



Note that for the process with PID 1988, the standard output (fd 1) refers to a pipe. In addition it is attached to the write end of the pipe because write permission is enabled from the pipe. Also note that df 3 is attached to the read end of a different pipe (because its identifier 11951 is different).

Linux / UNIX offers documentation pages (man pages). The man command "function name" will print detailed information about the given function to the screen.