CS 3423
Fall 2019
Final Exam Review

True/false & short answer

1. Indicate true or false for the following statements:

__T__ When a process terminates, orphaned children will be adopted and waited on by the `init` (PID = 1) process.

__F__ The set of *basic* regular expressions includes the "?" quantifier.

__F__ If a process attempts to write to a full pipe (assuming at least one process currently has the pipe open for read), that process will be terminated.

__T__ If a process attempts a `read()` on an empty pipe, and any process currently has the pipe open for writing, the `read()` operation will block and wait for input to appear on the pipe.

__F__ A `write()` error will occur if a process seeks past the end of a file, then attempts to write at this location.

__T__ Objects in Python do not require declaration nor specification of their type

__T__ Many basic types in Python are immutable (i.e. an assignment to an object creates a reference to, rather than a copy of, the object)

__F__ Children that are not waited on by their parent are known as "orphans."

__F__ `lseek` will fail if we attempt to seek beyond the end of the file.

2. **Short answer:**
   [20pts] Short answer (3 sentences maximum):
   Explain why the following code fragment will not work as expected:

```
pipe(fd);
switch(cpid = fork()) {
   case -1:   perror("fork failed"); break;
   case  0:
      while(read(fd[0], buffer, sizeof(buffer)) > 0) {
         printf("recv msg: %s\n", buffer);
      }
      break;
   default:
      for(i = 0; i < num_msgs; i++) {
         write(fd[1], msg, sizeof(msg));
      }
      waitpid(cpid, NULL, 0);
      break;
}
```

Neither process closes the write end of the pipe. Thus, after finishing reading all of the messages, the child's final `read()` will block and wait indefinitely. (The `waitpid` by the parent is irrelevant, though it does ensure the parent is now blocked as well. However, even if the parent terminated, the child would still be blocked, because it holds a file descriptor to the write end of the pipe itself.)

Bash, sed, and awk

3. Write a Bash filename pattern (not a regex) that will only match the two indicated files:

   cs310v3.docx **cs310v3.pdf** cs310v3.txt cs310v3.xlsx
   cs513v3.pdf cs899d2.docx cs899d2.pdf

   cs[35]1[03]v3.{docx,pdf}

4. Write a bash script that takes a filename as a parameter. After verifying that the file exists and is readable, pass each line of the file into "`wc -w`" in order to obtain a word count for each line. Find the longest line (in terms of number of words), and print this line, as well as its length.

```
#!/bin/bash

if [ $# -lt 1 ]; then
   echo "Usage: $0 <textfile>"
   echo
   exit 1
fi

file=$1
if [ ! -r "$file" ]; then
   echo "ERROR: could not open input file for reading"
   echo
   exit 1
fi

let max=0
longline=""
while read line; do

   wcout=$(echo $line | wc -w)
   if ((wcout > max)); then
      let max=wcout
      longline=$line
   fi

done < "$file"

echo "long line = $max words, contents:"
echo $longline
```

5. Write a bash script that will take one or more directory names as arguments, and count the directories *under* each (that is, the number of subdirectories, subdirectories of subdirectories, and so on; do not count the arguments themselves).
   Hint: you may use utilities we've discussed in class to help accomplish this.

```
#!/bin/bash

if [[ $# -eq 0 ]]; then
    echo "Usage: $0 <dir> [dir2 ... dirN]" 1>&2
    exit 1
fi

output=`find "$@" -type d | wc -l`
let count=$((output - $#))

echo "count = $count"
```

6. Write a sed command that will remove all lines longer than 80 characters.

```
/.\{81,\}/d
```

7. Write a sed command that will delete all leading blank lines at the top of a file.

```
/./,$!d

- alternative solution: -

0,/./{h;d};H;z;x;s/^\n//
```

8. Write a sed script that will:

   a. In all lines starting with "*PTY", replace the string "*PTY" with "/*" characters and end the line with "*/"
   b. In the lines starting with "*PTY", replace "X11R6" with "XFree86"
   c. Do not replace "X11R6" on any other lines.

```
/^\*PTY/ {
   s__/\*_
   s_$_\*/_
   s_X11R6_XFree86_g
}
```

9. An ISP has a data file which contains information about user's connection times and bandwidth usage as well as storage usage. There are two types of lines in the file, a connect line which has 5 fields containing the login, start, and stop times, as well as the bandwidth usage broken into the number of bytes in versus the number of bytes out.

```
<username> <start_time> <stop_time> <bytes_in> <bytes_out>
ssilvestro    1378          2463     12379894     34563
```

and a storage usage line which contains the number of bytes of disk space currently being used by the user which has the following format:

```
*storage*  <username>  <bytes_of_storage>
*storage*  ssilvestro  2784643
```

Write an awk script to find the total amount of connect time being used by the user `ssilvestro`, the total amount of bandwidth consumed (both in and out) as well as the *maximum* amount of storage used during this period. The output should look like this:

```
ssilvestro
Connect time = 1085
   Bandwidth = 123144457
   Storage   = 2784643
```

```
$1 == "ssilvestro" {
   connect_time += ($3 - $2)
   total_bandwidth += ($4 + $5)
}

$1 == "*storage*" && $2 == "ssilvestro" {
   if($3 > max_storage)
      max_storage = $3
}

END {
   print "ssilvestro"
   print "Connect Time =", connect_time
   print "   Bandwidth =", total_bandwidth
   print "   Storage   =", max_storage
}
```

Python

10. Write a Python script that will accept a regular expression as well as one or more filenames as command-line arguments. Search each file, counting the number of occurrences of the regular expression. Your script should print the total number of matches found in each file (if this number is greater than zero), as well as a total for all files. (Do not use ***any*** external commands to do this (for example, `grep`)).

```
#!/usr/bin/env python3

import sys
import re

if len(sys.argv) < 3:
  print("Usage: " + sys.argv[0] +
    " <regex> <file1> [file2 ... fileN]", file=sys.stderr)
  sys.exit(1)

sys.argv.pop(0)
regex = sys.argv.pop(0)

count = 0
for filename in sys.argv:

  with open(filename, "r") as infile:
    linenum = 0
    filecount = 0
    for line in infile:
      linenum += 1
      result = re.findall(regex, line)
      if result:
        count += len(result)
        filecount += len(result)
    if filecount > 0:
        print(filename + " = " + str(filecount))

print()
print("total count = " + str(count))
```

11. Write a Python script that will read data of the following format from standard input:

```
14 78.3 99 0 44.6 -52 9.12 -3 42 32 -81.9 0 0 0 4.78 21.19 5.5
3 21.09 83.69 2.515 56.8 22.1
```

The first number on each line is an integer that refers to the position of a value in the same line.

After reading all lines of input, your script should print the average of these values. For example, given the two lines above, the output should be `3.6475` (because $(4.78 + 2.515) / 2 = 3.6475$).

```
#!/usr/bin/env python3

import sys
import re

sum = 0
numlines = 0
for line in sys.stdin:
        tokens = line.split(" ")
        target = tokens[int(tokens[0])]
        sum += float(target)
        numlines += 1

print("average = " + str(sum / numlines))
```

12. Write a Python script that reads the contents of all files listed as arguments. The script should print a redacted version of the input files to standard out, such that all phone numbers are replaced with XXX-XXX-XXXX. After the contents are printed, print a sorted list of the actual phone numbers that were removed, with the numbers each formatted as such: ###-###-####.
Assume that more than one phone number can exist per line, and these numbers may take any of the following forms: XXX-XXX-XXXX, (XXX)XXX-XXXX, (XXX) XXX-XXXX.

Example input file:

```
Lorem ipsum (555)232-8891 elit, sed do eiusmod 123-323-1112
tempor ut labore et dolore (123)323-1113 magna aliqua. Ut
enim ad (123) 323-1111 minim veniam, quis nostrud.
```

Expected output:

```
Lorem ipsum XXX-XXX-XXXX elit, sed do eiusmod XXX-XXX-XXXX
tempor ut labore et dolore XXX-XXX-XXXX magna aliqua. Ut
enim ad XXX-XXX-XXXX minim veniam, quis nostrud.
555-232-8891
123-323-1112
123-323-1113
123-323-1111
```

```python
#!/usr/bin/env python3

import sys
import re

def fixnum(num):
    return re.sub(r"\(?(\d{3})\)?[- ]?(\d{3})-(\d{4})",
       r"\1-\2-\3", num)

if len(sys.argv) < 2:
    print("Usage: " + sys.argv[0] + " <infile>",
file=sys.stderr)
    sys.exit(1)

numbers = [ ]
for filename in sys.argv[1:]:
   with open(filename, "r") as infile:
      for line in infile:
         line = line.rstrip("\n")
         results = map(fixnum,
            re.findall(r"\(?\d{3}\)?[- ]?\d{3}-\d{4}", line))
         numbers.extend(results)
         line = re.sub(r"\(?\d{3}\)?[- ]?\d{3}-\d{4}",
            "XXX-XXX-XXXX", line)
         print(line)

for number in numbers:
   print(number)
```

Python regular expressions:

13. Fill in the value of the Group 1 column for each row of the table.

String	Pattern	Group 1
aooobc	/(a?o*)bc	aooo
aooobc	/([a-z]+?)/	a
aooobc	/([a-z]+?)+/	c
xyz+543.c	/(\+\d+)/	+543
xyz+543.c	/(.*)/	xyz+543.c
xyz+543.c	/(.*?)/	*Nothing*
xyz+543.c	/(.*)?/	xyz+543.c
xyz+543.c	([a-z0-9+]+)+([0-9]+)	xyz+54
xyz+543.c	([a-z0-9+]+)+	xyz+543
xyz+543.c	/([a-z]+?)/	x
-^oo^--^..^-	/(^..^)/	*Nothing*
-^oo^--^..^-	/(\^..\^)/	^oo^

Low-level I/O

14. Write a C **function** whose prototype is provided below:

```
int replaceRec(int fd, int iRecNum, struct custData * record);
```

This function accepts a file descriptor, a record number (indexed at zero), and a pointer to a structure. Your function should move the structure at the specified position to the end of the file, then replace the original record (at `iRecNum`) with the record passed by pointer to your function. Your function should exclusively use low-level I/O functions only.

```
int replaceRec(int fd, int iRecNum, struct custData * record) {
        int i;
        off_t seekPos;
        size_t recsize = sizeof(struct custData);
        struct custData buffer;

        seekPos = recsize * iRecNum;

        // Seek to the position of the indicated
        // record (i.e., the record at offset iRecNum).
        if(lseek(fd, seekPos, SEEK_SET) == -1) {
            perror("seek failed");
            return EXIT_FAILURE;
        }

        // Read the record into our local buffer in memory.
        if(read(fd, buffer, recsize) != recsize) {
            perror("read failed");
            return EXIT_FAILURE;
        }

        // Seek to the end of the file (this is the position
        // immediately after the last byte. i.e., if we write
        // anything at this position, it will be appended to
        // the end of the file.
        if(lseek(fd, 0, SEEK_END) == -1) {
            perror("seek failed");
            return EXIT_FAILURE;
        }

        // Write the record from our buffer to the file
        // (i.e., append the record to the end of the file)
        if(write(fd, buffer, recsize) != recsize) {
            perror("write failed");
            return EXIT_FAILURE;
        }

        // Seek again to the position of indicated record
        // (the record whose offset is iRecNum).
        if(lseek(fd, seekPos, SEEK_SET) == -1) {
            perror("seek failed");
```

```
            return EXIT_FAILURE;
        }

        // Finally, overwrite the old record in the file with
        // the record that was passed by pointer into this
        // function.
        if(write(fd, record, recsize) != recsize) {
            perror("write failed");
            return EXIT_FAILURE;
        }

        return 0;
}
```

15. Write a C program that will accept two filenames as command-line arguments. Using only low-level I/O functions, print the contents of the first file to standard out, followed by the contents of the second file. (You are essentially duplicating the functionality of "`cat file1 file2`"). You must use a buffer size greater than 1 byte.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>

#define BUF_SIZE 64

int main(int argc, char ** argv) {
    int fdin1, fdin2;
    char * inFilename1, * inFilename2;
    char * buffer[BUF_SIZE];
    ssize_t bytesRead;

    // The user must supply exactly two command line arguments.
    if(argc != 3) {
        fprintf(stderr, "Usage: %s <file1> <file2>\n", argv[0]);
        return EXIT_FAILURE;
    }
    inFilename1 = argv[1];
    inFilename2 = argv[2];

    // Open both files for read-only access.
    if((fdin1 = open(inFilename1, O_RDONLY)) == -1) {
        perror("open file failed");
        return EXIT_FAILURE;
    }
    if((fdin2 = open(inFilename2, O_RDONLY)) == -1) {
        perror("open file failed");
        return EXIT_FAILURE;
    }

    // Read the first file in a loop, fetching BUF_SIZE number of
    // bytes each time. Unless the file's size is a multiple of
    // BUF_SIZE, the last read operation will read < BUF_SIZE bytes.
    // This is why the call to write() specifies bytesRead number of
    // bytes, rather than BUF_SIZE.
    while((bytesRead = read(fdin1, buffer, BUF_SIZE)) > 0) {
        if(write(STDOUT_FILENO, buffer, bytesRead) != bytesRead) {
            perror("write failed");
            return EXIT_FAILURE;
        }
    }
    close(fdin1);

    // Repeat for the second file.
    while((bytesRead = read(fdin2, buffer, BUF_SIZE)) > 0) {
```

```
        if(write(STDOUT_FILENO, buffer, bytesRead) != bytesRead) {
            perror("write failed");
            return EXIT_FAILURE;
        }
    }
    close(fdin2);

    return EXIT_SUCCESS;
}
```

Process Control

16. Indicate true or false for the following statement:

__F__ It is possible for the following program to produce the output string "`acabac`" (Hint: drawing a process diagram may help).

```
int main() {
   printf("a");
   if(fork() == 0) {
      if(fork()) {
         printf("b");
      } else {
         printf("c");
         exit(0);
      }
      printf("a");
   }
   printf("c");
   return 0;
}
```

Further explanation: the second 'a' cannot come before 'b' (*one* 'a' can come before 'b', but not *both* 'a's).

17. List all possible outputs for the following program (hint: drawing a process diagram may help):

```
int main() {
   if(fork() == 0) {
      printf("a");
      if(fork()) {
         printf("b");
      } else {
         exit(0);
      }
   }
   printf("c");
   return 0;
}
```

{ abcc, acbc, cabc }

18. Write a C program that can be used as a convenience, and whose functionality is equivalent to the following shell command:

```
awk -f getuid.awk < /etc/passwd | sort -r
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

// example: awk -f getuid.awk < /etc/passwd | sort -r

int main(int argc, char ** argv) {
   int fd[2];
   int infilefd;

   // Create the pipe we will use to pass data from awk to sort.
   if(pipe(fd) == -1) {
      perror("pipe creation failed");
      return EXIT_FAILURE;
   }

   // Create a child process, which we will use to run "awk"
   // (the parent will be used to run "sort").
   switch(fork()) {
      case -1:
         perror("fork failed");
         return EXIT_FAILURE;

      case 0: // child (awk)
         // Open the input file for read-only access.
         if((infilefd = open("/etc/passwd", O_RDONLY)) == -1) {
            perror("open failed");
            return EXIT_FAILURE;
         }

         // Dup the input file's fd to standard input.
         if(dup2(infilefd, STDIN_FILENO) == -1) {
            perror("dup2 failed");
            return EXIT_FAILURE;
         }

         // Dup the pipe's write end to standard output.
         if(dup2(fd[1], STDOUT_FILENO) == -1) {
            perror("dup2 failed");
            return EXIT_FAILURE;
         }

         // Cleanup unneeded/duplicate file descriptors.
         close(infilefd);
         close(fd[0]);
         close(fd[1]);
```

```
            // Execute: awk -f getuid.awk
            execlp("awk", "awk", "-f", "getuid.awk", NULL);
            perror("execlp failed");
            return EXIT_FAILURE;


        default: // parent (sort; the shell will wait on this process)
            // Dup the pipe's read end to standard input.
            dup2(fd[0], STDIN_FILENO);

            // Cleanup unneeded/duplicate file descriptors.
            close(fd[0]);
            close(fd[1]);

            // Execute: sort -r
            execlp("sort", "sort", "-r", NULL);
            perror("execlp failed");
            return EXIT_FAILURE;

    }

    // It would be impossible for us to read this line of code:
    // if the fork failed, we would have exited immediately. If either
    // execlp failed, we likewise would have returned immediately. Thus,
    // there is no actual path to this return statement.
    return EXIT_FAILURE;
}
```

19. Write a C function named `handleNewReq()`, whose prototype is given below, that will accept a pointer to a buffer, the pathname of an executable, and an `argv` array. The function should execute the given program (preserving the parent process), using the supplied argument vector, and should write the data structure to standard input of the new process.

    Bonus points: ensure that we do not create any zombies, and also avoid waiting for a potentially unbounded length of time.

    ```
    int handleNewReq(char * exename, char * argv[], struct init *
    data);
    ```

```c
int handleNewReq(char * exename, char * argv[], struct init * data) {
   int fd[2];
   pid_t cpid;
   size_t structSize = sizeof(struct init);

   // Create the pipe that the parent will use to send the data
   // structure to the child process.
   if(pipe(fd) == -1) {
      perror("pipe failed");
      return EXIT_FAILURE;
   }

   // Fork to create the child process.
   switch(cpid = fork()) {
      case -1:
         perror("fork failed");
         return EXIT_FAILURE;

      case 0:                          // child
         // Dup the read end of the pipe into the child's stdin.
         if(dup2(fd[0], STDIN_FILENO) == -1) {
            perror("dup2 failed");
            return EXIT_FAILURE;
         }

         // Cleanup unneeded/duplicate file descriptors.
         close(fd[0]);
         close(fd[1]);

         // Execute the command passed into this function.
         execvp(exename, argv);
         perror("execvp failed");
         return EXIT_FAILURE;

      default:                  // parent
         // Write the data structure to the child via the write
         // end of the pipe.
         if(write(fd[1], data, structSize) != structSize) {
            perror("write failed");
            return EXIT_FAILURE;
         }
```

```
            // Cleanup unneeded/duplicate file descriptors.
            close(fd[0]);
            close(fd[1]);

            // Wait for the child to complete before returning to the
            // caller.
            waitpid(cpid, NULL, 0);
    }

    return 0;
}
```

20. Write a C program that will fork `num` (where `num` is a command-line argument) children. Every child whose PID is odd will write their PID to every other child via a pipe (but not parent). Each child will then read every message written to them, one-at-a-time, then write each received PID along with its own PID to the original parent using the following message structure.

```
struct message {
   pid_t my_pid;
   pid_t recv_pid;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

#define INIT_PID 9999999

struct message {
   pid_t my_pid;
   pid_t recv_pid;
};

int main(int argc, char ** argv) {
   // Initialize cpid to be non-zero in order to allow the parent to
   // enter the for-loop below.
   pid_t cpid = INIT_PID;
   pid_t my_pid;
   pid_t their_pid;
   int my_index = 0;
   int num_fork;
   int i;
   struct message msg;
   size_t msg_size = sizeof(struct message);

   // The user should supply exact one command line argument.
   if(argc != 2) {
      fprintf(stderr, "Usage: %s <num>\n", argv[0]);
      return EXIT_FAILURE;
   }
   num_fork = atoi(argv[1]);


   // We need a pipe for every child, as well as the parent.
   int fd[num_fork + 1][2];

   // Create all of the necessary pipes.
   for(i = 0; i <= num_fork; i++) {
      if(pipe(fd[i]) == -1) {
         perror("pipefailed");
```

```
      return EXIT_FAILURE;
   }
}

// The parent will now create num_fork number of child processes.
// The "cpid > 0" within the loop condition ensures that child
// processes will exit the loop, as we do NOT want them to begin
// participating (without this, we would end up with 2^num_fork
// child processes).
for(i = 1; i <= num_fork && cpid > 0; i++) {
   switch(cpid = fork()) {
      case -1:
         perror("fork failed");
         return EXIT_FAILURE;

      case 0:  // child process
               // The child will simply update their index number to
               // equal the current value of the loop counter.
         my_index = i;
   }
}

// All processes will obtain their PID value.
my_pid = getpid();

// No process will ever need to write to its own pipe, thus
// we should close our own write end.
close(fd[my_index][1]);

if(my_index > 0) {
   // Children close all other child pipes' read ends,
   // as a child will only ever read from its own pipe.
   for(i = 0; i <= num_fork; i++) {
      if(my_index != i) {
         close(fd[i][0]);
      }
   }
} else {
   // Parent closes both ends of ALL child pipes, as it
   // will never have to read OR write on any of their
   // pipes (it will only read from its own pipe).
   for(i = 1; i <= num_fork; i++) {
      close(fd[i][0]);
      close(fd[i][1]);
   }
}


// Only child processes will enter this block...
if(my_index > 0) {
// Only odd-PID children will write to other children
   if(my_pid % 2 == 1) {
      // Loop through all child process' indexes
      // (i.e., from 1 <= i <= num_fork).
      for(i = 1; i <= num_fork; i++) {
         // Make sure I do not write a message to myself...
         if(i != my_index) {
```

```
                // Now write my PID to this sibling processs...
                if(write(fd[i][1], &my_pid, sizeof(pid_t)) !=
                        sizeof(pid_t)) {
                   perror("write failed");
                   return EXIT_FAILURE;
                }
             }
          }
       }

       // ALL children should now clean up their pipe ends (close write
       // ends of all other children)
       for(i = 1; i <= num_fork; i++) {
          if(i != my_index) {
             close(fd[i][1]);
          }
       }
    }

    // All children will now read their pipes for the PIDs they
    // received from their siblings
    if(my_index > 0) {
       while(read(fd[my_index][0], &their_pid, sizeof(pid_t)) ==
          sizeof(pid_t)) {
          // Build the message I will send to my parent by populating
          // this data structure with my PID as well as the sibling who
          // wrote to my pipe.
          msg.my_pid = my_pid;
          msg.recv_pid = their_pid;

          // Send the data structure to my parent via their pipe.
          if(write(fd[0][1], &msg, msg_size) != msg_size) {
             perror("write failed");
             return EXIT_FAILURE;
          }
       }

       // Each child will now close their own pipe's read end (as they
       // are done reading), as well as the write end of the parent's
       // pipe (as they are also done writing).
       close(fd[my_index][0]);
       close(fd[0][1]);
    } else {            // parent will now read its pipe for messages
       while(read(fd[0][0], &msg, msg_size) == msg_size) {
          printf("pid %d received data from pid %d\n", msg.my_pid,
                msg.recv_pid);
       }

       // Finally, the parent will close it's own pipe's read end, as
       // the pipe is now empty (we previously closed the write end, so
       // we don't need to worry about doing that).
       close(fd[0][0]);
    }

    return EXIT_SUCCESS;
}
```

Inodes

21.

a. How many *data blocks* are utilized for a file with 8GB of data? Assume 4K blocks.
`2,097,152 data blocks / direct pointers needed`

b. How many *blocks of direct pointers* (blocks pointed to by indirect pointers) are necessary to reference the data blocks in question a? Assume 4 byte addresses.
`2,048 blocks of direct pointers needed`

c. How many *blocks of indirect pointers* (blocks pointed to by double indirect pointers) are necessary to reference the direct pointer blocks in question b?
`2 blocks of indirect pointers needed`

d. How many *blocks of double indirect pointers* (blocks pointed to a triple indirect pointer) are necessary to reference the indirect pointer blocks in question c?
`1 block of indirect pointers needed`

e. How many total blocks are needed (not including the inode)?
`2,099,203 total blocks needed`