Problem 1 -- Signals and Pipes Q&A

1A) I have a program that someone else wrote, which I launch from a shell prompt. Every once in a while, the program hangs. The developer of that program wants me to send a `core` (core dump) file when this happens, for the purpose of debugging what might be causing this hang. Suggest two ways I could bring this to fruition.

1B) Let’s say we want to do a little hazing of those students who are just learning the command line. Whenever they are running a program that requests a line of input from the keyboard, and they press BACKSPACE to correct a mistake, we want it to terminate the program instead. How can we set up this prank?

1C) In problem set 1, you wrote a simple I/O buffering library. I gave you a "free pass" on partial writes (we just treated these as errors) with the caveat that later in the class, we’ll encounter problems with partial writes. Explain a situation, where pipes and signal handlers are in use, where you could be bitten by this partial write "bug." Be specific!

1D) You’re a senior system administrator and a junior employee places a panic-stricken call to you (yeah, I know, who uses phones anymore?) It seems there is a process 1234 that can’t be killed, even with `kill -9`. Mustering all of your years of experience and wisdom, what do you think is causing this?

1E) Most interrupts originate from hardware devices external to the CPU, such as disk, network, or the timer. What is an Interprocessor Interrupt? How does it relate to signals?

Problem 2 -- Signal Handler Experiments

Below is a program. You should compile and run it (the header files need to be added) on a Linux system.

2A) Run it with the the signal number set to 15 and various counts like 1000, 2000, etc. What do you observe about the reported count? Why is that the case?

2B) Increase the count to fairly high numbers, such as 50000. You’ll find that something tragic happens to the child. Analyze and explain. If necessary, turn on core dumps or run the program inside a debugger to do a post-mortem analysis.

2C) Recompile the program so the sigaction flags are the default 0. Now repeat experiment 2B with various counts. What do you observe now? Explain.

2D) Finally repeat experiment 2C but use a signal number between 34 and 63. What are you seeing now with respect to the reported counts? Discuss.

```
int niter;
volatile int count;

void hh(int sig)
{
          count++;
}


void xx(int sig)
{
          fprintf(stderr,"Child got signal %d, count is %d \n",sig,count);
          exit(128);
}

main(int argc,char **argv)
{
 struct sigaction sa;
 int i,cpid,ws,signum;
 struct timespec ts;
          niter=atoi(argv[1]);
          signum=atoi(argv[2]);
          sa.sa_handler=hh;
          sa.sa_flags=SA_NOMASK;
          sigemptyset(&sa.sa_mask);
          (void)sigaction(signum,&sa,NULL);
          signal(SIGINT,xx);
          switch (cpid=fork())
          {
           case 0:
                    for(;;)
                      ;
                    fprintf(stderr,"Child broke loop\n");exit(1);
         default:
                    for(i=0;i<niter;i++)
                              kill(cpid,signum);
                    ts.tv_sec=1;ts.tv_nsec=0;
                    nanosleep(&ts,0);
                    kill(cpid,SIGINT);
          }
          wait(&ws);
          fprintf(stderr,"Parent got wait status %x\n",ws);
          return 0;
}
```

NOTE: This experiment has been tested on Linux systems. Your results on MacOS may be quite different.

Submission: You must submit screenshots of running these experiments, in addition to your explanatory writeup.

Problem 3 -- A three-command pipeline

In this problem, you will set up a pipeline connecting three simple programs, each of which you will write. These programs are:

1) `wordgen`: Generate a series of random potential words, one per line. Each of these "words" consists of between 3 and 10 characters. The number of characters and the value of each character are all chosen randomly. Be sure to seed the pseudorandom number generator, e.g. from the pid or time of day, so that the output is indeed random from run to run. For simplicity, pick characters from among the UPPERCASE letters only. If this command has an argument, that is the limit of how many potential words to generate. Otherwise, if the argument is 0 or missing, generate words in an endless loop.

2) `wordsearch`: First read in a list of dictionary words, one per line, from a file that is supplied as the command-line argument. Then read a line at a time from standard input and determine if that potential word matches your word list. If so, echo the input word (including the newline) to standard output, otherwise ignore it. Id est, `wordsearch` is a *filter*. The program continues until end-of-file on standard input. To simplify, you can convert all of the words in your word list to UPPERCASE. I don't care how efficiently you search for each candidate word. In fact, a linear search will be best because it wastes the most time, and we want this filter to be slow and CPU-intensive.

3) `pager`: This is a primitive version of the `more` command, and resembles an earlier popular UNIX command `pg`. Read a line at a time from standard input and echo each line to standard output, until 23 lines have been output. Display a message `---Press RETURN for more---` to standard output and then wait for a line of input to be received from the terminal. The terminal is not standard input in this case. You can open the special file `/dev/tty` to get the terminal (open it once at the start of the program). Since the terminal waits until a line of input is received, your `pager` program will thus pause until a newline is hit. Continue doing this, in chunks of 23 lines, until either end-of-file is seen on standard input, or the command q (or Q) is received while waiting for the next-page newline.

Note: You are permitted to use regular library functions for these three programs. There is no requirement, for example, to use read and write system calls directly.

Test each program individually:

```
$ ./wordgen 1000 | wc
Finished generating 1000 candidate words
   1000    1000    7391
```

(this verifies that wordgen generates the correct number of target words. Feel free to examine the output more thoroughly to see that it is indeed spitting out different random words.)

```
$ time ./wordgen 1000000  >/dev/null
Finished generating 1000000 candidate words

real    0m0.136s
user    0m0.135s
```

```
sys     0m0.000s
```
(this shows that wordgen is a fairly quick producer, generating over a million candidates per second)

```
$ echo "COMPUTER" | ./wordsearch words.txt
Accepted 416290 words, rejected 50254
COMPUTER
Matched 1 words
```
words.txt is a list of 466544 dictionary words that I got from GitHub. I chose to reject words that have characters other than letters in the English alphabet. Thus 50254 of the words were reported as rejected during the initial read-in. Obviously the word COMPUTER is a match. Note that only `COMPUTER` is going to stdout, while the "Accepted" and "Matched" outputs are to stderr.

```
$ time ./wordgen 10000 | ./wordsearch  words.txt >/dev/null
Accepted 416290 words, rejected 50254
Finished generating 10000 candidate words
Matched 327 words

real    0m28.929s
user    0m28.755s
sys     0m0.016s
```
(This demonstrates how slow wordsearch is! The performance amounts to about 300 candidates per second while searching the 400,000+ word list linearly each time)

```
$ ./pager <words.txt
2
1080
&c
10-point
10th
11-point
12-point
 ..... more lines here ....
---Press RETURN for more---

 ..... another page of output, etc......
---Press RETURN for more---

q
*** Pager terminated by Q command ***

$echo $?
0
```
(This demonstrates the pager command. I pressed RETURN twice and q RETURN once.)

So far, this assignment has nothing to do with operating systems other than maybe `/dev/tty`.

Let's test the operation of all three programs:

```
 ./wordgen 5000 | ./wordsearch words.txt | ./pager
```

This should generate a few dozen matching words (it is random, after all) and then exit. Now:

```
 ./wordgen | ./wordsearch words.txt | ./pager
```

In this pipeline, we expect that wordgen creates output a lot faster than wordsearch can consume it. Therefore flow control will engage in the first pipe. The pager program doesn't incur much CPU time, but if the user isn't quick at pressing RETURN, the output from wordsearch will back up and flow control will engage at the second pipe. On the other hand, if the user reads the screens quickly, pager may be waiting for the pipe to fill up.

This may take a while before the first screen of output appears, while in the first case it was almost instant. Why? It's because the wordsearch program is using the stdio library, and when stdout goes to a pipe, it is in file-buffered mode. wordsearch will build up 4K worth of output before stdout flushes into the pipe. Therefore pager is waiting for 4K worth of matched words, which takes a little time. If the program is taking too long, trim down the word list or make wordsearch more efficient (e.g. binary search, hashing).

Note that since wordgen is not given an argument in the second example above, it generates an infinite stream of words and thus wordsearch will endlessly be finding matches and outputting them. What happens when you hit Q in the pager program? Think about why the other two processes exit eventually, but not immediately.

You can always substitute the system pager program `pg` or `more` for comparison and testing. `pg` may not be installed by default on many modern Linux distros. It is installed on the ECE357 VM.

Problem 3B -- Launcher

Now write a program which launches these three programs with the pipeline connecting them as demonstrated. The `launcher` program takes one (optional) argument which is passed directly to `wordgen`. After launching the three child processes, the launcher program sits and waits for all children to exit, and reports the exit status of each.

```
$ ./launcher
Accepted 24991 words, rejected 126
IRS
MAW
HOC
CPU
PATE
TUN
 ... more matching words ...
---Press RETURN for more---
 ... more matching words ...
*** Pager terminated by Q command ***
Child 21290 exited with 0
Child 21289 exited with 13
Child 21288 exited with 13
$ ./launcher 100
Finished generating 100 candidate words
Child 21294 exited with 0
```

```
Accepted 24991 words, rejected 126
Matched 1 words
ROW
Child 21295 exited with 0
Child 21296 exited with 0
```

Note in the second example, there were only 100 candidates and 1 match, so the `Press RETURN` was never displayed by pager. Also note the exit status (I didn't bother to decode it like we did in PS3): In the first example, both the wordgen and the wordsearch died from a signal #13, which on Linux is the number of SIGPIPE. The reason for this should be obvious, if not, think about it some more! In the second example, all children exited normally: wordgen exited after generating 100 words, and wordsearch and pager both exited after seeing EOF on their pipe inputs.

Problem 3C -- Add signal handling

Now add the following simple feature: when `wordsearch` terminates, it should report (on stderr) the number of words matched (this feature is already illustrated in the examples above. You can just build it in to begin with). Here's the problem: when the user terminates pager with the Q command, this will result in a broken pipe the next time wordsearch tries to write to its output. By default, this causes SIGPIPE delivery and wordsearch wouldn't get a chance to output its helpful message.

Solve this problem by adding a signal handler for SIGPIPE so this message is always displayed. Your solution could involve `setjmp` too, or you could choose to block SIGPIPE and deal with the EPIPE error.

Submission: Submit all 4 program listings (wordgen, wordsearch, pager, launcher) and a screenshot/output paste. Demonstrate both scenarios (wordgen has a finite list and all children exit normally vs wordgen is infinite and the pipeline is terminated by pager). Be reasonable about including pages and pages of repetitive meaningless word lists! You do not need to included intermediate versions, such as the version of wordsearch without the signal handling.