Problem 1 -- Q&A

1A) What key sequence would you use to cause a foreground process to terminate and dump core (if possible)?

1B) When would a signal `SIGTTOU` be sent to a process? What happens to the process when it receives it (assuming that this signal's disposition is default)?

1C) Process "A" sends (using the `kill` system call) signal #40 to process "B". It does so 3 times in a row. At the time, process "B" has signal #40 in the blocked signals mask and has set up a handler for this signal. At some later time, process "B" unblocks signal #40. What happens and why?

1D) Take a look at this program:

```
#include <sys/signal.h>
#include <stdio.h>
#include <setjmp.h>

jmp_buf wormhole;

void handler(int sig)
{
 static int i;
         ++i;
         fprintf(stderr,"In handler instance %d\n",i);
         longjmp(wormhole,i);
}

main()
{
/* The keyword volatile prevents the compiler from optimizing away
 * the accesses to *p
 */
volatile char *p=0;
         signal(SIGSEGV,handler);
         if (setjmp(wormhole))
         {
                  fprintf(stderr,"The other side\n");
                  *p++;
         }
         *p--;
         return 1;
}
```

What does it do? What is its wait exit status? Don't just tell me, explain what is happening and why.

Problem 2 -- Analyze This

Observe the following program:

```
int pp[2];

main()
{
 char buf[4096];
 int n;
         pipe(pp);
         if (!fork())
         {
                  child('A');
                  return 0;
         }
         if (!fork())
         {
                  child('B');
                  return 0;
         }
         close(pp[1]);      /* THIS LINE */
         while ((n=read(pp[0],buf,sizeof buf))>0)
                  write(1,buf,n);
         return 0;
}

child(c)
{
 char buf[1024];
 int i;
         memset(buf,c,1024);
         for(i=0;i<4096;i++)
                  if (write(pp[1],buf,1024) != 1024)
                           perror("omg");
         return 0;
}
```

2A) Can this program ever produce as part of its output the sequence ABA? Explain why or why not.

2B) What happens if I remove the line of code marked `THIS LINE`? Explain why

Problem 3 -- A three-command pipeline

In this programming assignment, you will launch a pipeline of three separate commands and collect the exit status of

each of the children.

First, write three fairly simple programs:

`wordgen`: Generate a series of **random** potential words, one per line, each with between 3 and 15 characters. The length of each random word is also to be random. 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. Note: for simplicity, make the words UPPERCASE.

`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 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. At the end of the program, `wordsearch` outputs to standard error the total number of words matched. To simplify coding, 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, part of the point is that this program is going to be slow and CPU-intensive.

`pager`: This is a primitive version of the `more` command, and resembles an earlier popular UNIX command `pg`. Read from standard input and echo each line to standard output, until 23 lines have been output. Display a message `---Press RETURN for more---` to the terminal and then wait for a line of input to be received from the terminal. The terminal is not standard input/output in this case. You will open the special file `/dev/tty` to get the terminal. 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 that all 3 programs should exit with code 0.

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)

```
$ 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. It should be obvious that all of the messages in the example output above were sent to standard error.

```
$ echo "COMPUTER"| ./wordsearch words.txt
Accepted 465120 words
COMPUTER
Matched 1 words
```

`words.txt` is a list of 465120 dictionary words that I originally took from GitHub. I removed abbreviations, numbers, and non-ASCII characters. Note that the messages `Accepted 465120 words` and `Matched 1 words` in the output above went to standard error, while the matched word `COMPUTER` went to standard output.

```
$ time ./wordgen 10000 | ./wordsearch  words.txt >/dev/null
Accepted 465120 words
Finished generating 10000 candidate words
Matched 327 words

real    0m28.929s
user    0m28.755s
sys     0m0.016s
```

This demonstrates how slow wordsearch is! The above is an inefficient linear-search version of the program and it was only able to process about 300 candidates per second.

```
$ ./pager pager.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

 ..... more lines here, but I won't show them to you!
---Press RETURN for more---
 ..... another page of output, etc......
---Press RETURN for more--- q
*** Pager terminated by Q command ***

$echo $?
0
```

So far, this assignment has nothing to do with operating systems other than maybe `/dev/tty`. OK, test the operation of the following pipeline:

```
 ./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 prompt 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? Because pager is reading BUFSIZ (4K) bytes at a time. The wordsearch program needs to generate 4K worth of output before pager sees the first read system call return. In the first case, the list of matching words is small. If the

program is taking too long, trim down the word list.

You can always substitute the system pager program `pg` or `more` for comparison.

Note that since wordgen is not given an argument, it generates endlessly 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 don't exit immediately.

The Launcher

Now write a program which launches these three programs with the pipeline connecting them as demonstrated. The `launcher` program takes one 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 child.

```
$ ./launcher
Accepted 465120 words
IRS
MAW
HOC
CPU
PATE
TUN
 .... more words
---Press RETURN for more--- q
*** 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 465120 words
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` message was never displayed by pager. Also note the exit status, which I did not bother to decode. In the first example, the first two children (we can infer the order since pids are given out sequentially) died with signal 13, which on Linux is SIGPIPE. Note that the wordsearch program didn't get a chance to output the number of words matched. We're going to fix that in a second. The pager child exited voluntarily (exit code 0). In the second example, there were no SIGPIPEs and all children exited voluntarily and normally.

Add signal handling for wordsearch:

Here's a 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 message at the end reporting the total number of words matched.

Solve this problem by adding a signal handler for SIGPIPE so this message is always displayed. Your solution could involve `setjmp` too. Now the output looks like this:

```
$ ./launcher
SAY
PREE
CCR
ISY
SOME
EVA
MEN
 ... (some more words here) ...
---Press RETURN for more---q
*** Pager terminated by Q command ***
Child 21465 exited with 0
pid 21464 received signal 13
Matched 2001 words
Child 21464 exited with 0
Child 21463 exited with 13
```

You see now that the wordsearch program, which was pid 21464, caught the signal 13, output its message, and exited normally.

Submission

Submit all 4 program listings and a screenshot/output paste. Be reasonable about including pages and pages of repetitive meaningless word lists!