Problem 1 -- A three-command pipeline

First, write three fairly simple programs:

`wordgen`: Generate a series of **random** potential words, one per line, each with between 3 and *nc* characters (choose *nc* according to taste). 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. 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, 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 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. 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.

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)

```
$ 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)

```
$ 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 above is an inefficient linear-search version of the program and it was only able to process about 300 candidates per second. The word list was pretty long also)

```
$ ./pager <words.txt
2
1080
&c
10-point
10th
11-point
12-point
 ..... more lines here, who cares? ....
---Press RETURN for more---q
 ..... 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. If the program is taking too long, trim down the word list or make wordsearch more efficient (e.g. binary search, hashing).

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.

Problem 2 -- 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 this:

```
$ ./launcher
Accepted 24991 words, rejected 126
IRS
MAW
HOC
CPU
PATE
TUN
 .... more words
---Press RETURN for more---
*** 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. Exit status 13 means the child died because of signal #13, which on Linux is SIGPIPE.

Problem 3 -- Add signal handling

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.

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