Problem 1 -- Signal Numbers

In the questions below, answer both in terms of symbolic signal names (such as SIGSTOP) and the specific signal number on your system. Be sure to mention what type of UNIX (e.g. BSD, Linux) you are running, what architecture, and what kernel version.

a) Which signal is delivered to all foreground processes attached to a terminal session when that terminal receives a Control-C character?

b) If we wanted to stop a program running in the foreground AND cause it to dump core (if possible), what key sequence do we use?

c) What signal is delivered to the following program:

```
main()
{
 *(int *)0 = 1;
}
```

d) Which signals are un-ignorable and un-blockable?

e) Go back over items a-d and mark which ones are "Synchronous" vs "Asynchronous"

Problem 2 -- Do Signals Count?

Below we have a program which asks the question "do signals count?" Type the program in and run it.

a) There are 3 places in the code where I've asked a question in the comment. Answer each question:

b) What are your observations on the number of signals received? Explain what is happening.

c) Now replace SIGUSR1 with SIGRTMIN and repeat the experiment. What do you see now and why?

```
#include <unistd.h>
#include <stdio.h>
#include <sys/signal.h>
#include <sys/wait.h>
#include <errno.h>

int signo,nsig,counter;
int rcvr_pid;

void handler(s)
{
         if (s==signo)
                  counter++;
}

int main(ac,av)
char **av;
{
 struct sigaction sa;
 int pid,status,nchild,i;
         signo=SIGUSR1;
         nsig=10000;
         rcvr_pid=getpid();
         sa.sa_handler=handler;
         /* WHAT would happen if I forgot this line below? */
         sa.sa_flags=0;
         /* or this line below? */
         sigemptyset(&sa.sa_mask);
         if (sigaction(signo,&sa,0)== -1)
         {
                  perror("sigaction");
                  return -1;
         }
         switch(pid=fork())
         {
```

```
            case -1:
                    perror("fork");
                    return -1;
            case 0:
                    sender();
                    return 0;
            }
            fprintf(stderr,"Waiting for sender\n");
            /* WHY do I have this thing with EINTR below?? */
            while (wait(&status)>0 || errno==EINTR)
                    ;
            printf("stderr,Received a total of %d of signal #%d\n",counter,signo);
}


sender()
{
int i;
            for(i=0;i<nsig;i++)
            {
                    kill(rcvr_pid,signo);
            }
}
```

Problem 3 -- Fixing your cat

Revisit your concatenation program from Unit 1. Make sure to correct any mistakes with error handling/reporting and partial write handling. Then modify it to accept only the following syntax:

```
catgrepmore pattern infile1 [...infile2...]
```

For **each** `infile` specified, open that file, but instead of copying to a specific file or to standard output, write the contents of the file into a pipeline of `grep pattern` piped into `more` (you could also use `pg` or `less`). This will require forking and execing two child processes. It is recommended that your original program be the parent of both the grep and the more children (as opposed to using grandchildren).

The intention is to display only those lines of the file that match the pattern, and to page through these results one screen at a time. When the user exits the pager program (by pressing the q key), move on to the next `infile`, if any. **Do not quit the entire program**. When the user sends a keyboard interrupt SIGINT (typically by pressing Control-C), do not abruptly terminate the program. Instead, first write to stderr the total number of files and bytes processed, and then exit.

Is this a fairly silly exercise? Perhaps. One could accomplish the same thing with existing UNIX utilities in a number of ways.

Be on the lookout: Setting up the pipes, and in particular avoiding dangling file descriptors, is important to proper program operation. If your program terminates unexpectedly or gets stuck in an endless loop, this may be a symptom of sloppy file descriptor work.

In order to properly test all aspects of your program, you must use sample files that are at least 16K long, so that the pipe buffer can actually fill up. You must also test for proper behavior when the user quits the pager program, or sends SIGINT. Using the ps command, make sure that when your program exits, under any circumstances, that the more and grep commands have also exited. If they get stuck, it is probably because of dangling file descriptors.

The same restrictions apply as in PS#1: Do not use the stdio library for opening, closing, reading, writing, forking, execing or setting up the pipe. Use the system calls directly. You may use library functions for argument parsing, error checking and reporting, printing out the exiting message, etc.

Helpful note: In order for `more` to receive keystrokes without interfering with reading from stdin, it opens a special device `/dev/tty` which is your session terminal. We haven't covered this yet. Furthermore, to receive the keystrokes without waiting for a newline character, more will put the `tty` device into a special mode. It may happen during development and testing of this program that more exits without restoring the `tty` to the normal mode, and your terminal session will appear to be "crashed". Characters will not echo and nothing will seem to work. To get out of this mode, press Control-J several times, then type the command `stty sane` and then press Control-J (NOT the Enter key!) again.