Problem 1 -- Signal Numbers & Behavior

When answering the questions below, 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 generated for all foreground processes attached to a terminal session when that terminal receives a Control-Z character?

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

c) What UNIX command would I use to make Control-I the character which causes SIGINT to be sent? This is admittedly an odd, but somewhat amusing thing to do.

d) I send a certain process signal #60 on a Linux system, using the `kill` system call, 32 times in a row. At the time, that process has signal #60 in its blocked signals mask, and has a handler function established. After some period of time has elapsed, the target process unblocks signal #60

i) How? Include a code snippet to unblock signal #60:

ii) How many times does the handler function run? If instead of signal #60, we had been talking about signal #2, how many times would the handler function run? Justify your answers.

Problem 2 -- Interrupted and Restarted System Calls

Below I have a little program which demonstrates interrupted and restarted system calls, and also uses pipes. It spawns two child processes, one of which makes 1024 write system calls to the pipe with a buffer size of 64K, the other continuously reads with a buffer of 555 bytes. After spawning these processes, the main process repeatedly sends SIGUSR1 to the writer child. Run this program as directed below, redirecting standard error to a file (rather than having it scroll past you), and answer some questions about the program and the results:

a) You should see that the "pipe short write" condition comes up, not necessarily with each and every write. Why is this happening?

b) Along the same lines of thought, do you notice any pattern to the integer reported with the "pipe short write" messages? Explain?

c) Change the `sa_flags` to 0 at the point indicated, recompile and re-run the program. What do you observe? What is the explanation?

d) Why do I do `signal(SIGCHILD,SIG_IGN)` ? What happens if I do not?

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

```
int p[2];

f()
{
          return;
}

main()
{
 int r,w;
 struct sigaction sa;
          signal(SIGCHLD,SIG_IGN);     // Question (d)
          sa.sa_handler=f;
          sigemptyset(&sa.sa_mask);
          sa.sa_flags=SA_RESTART;               // Change this to 0 for part (c)
          sigaction(SIGUSR1,&sa,0);
          pipe(p);
          switch(w=fork())
          {
           case 0:
                    writer();
                    exit(2);
                    break;
          }
          switch(r=fork())
          {
           case 0:
                    reader();
                    exit(3);
                    break;
          }
          close(p[0]);
          close(p[1]);
          for(;;)
          {
            int i;
                    if (kill(w,SIGUSR1)<0)
                    {
                              perror("kill");
                              exit(1);
                    }
                    for(i=0;i<1000;i++)
                              /*This is just to delay a little*/;
          }
}
```

```
reader()
{
  char buf[65536];
  int n;
          close(p[1]);
          for(;;)
          {
                    if ((n=read(p[0],buf,555))<0)
                    {
                              perror("pipe read");
                              exit(-1);
                    }
                    if (n==0)
                    {
                              fprintf(stderr,"Read returned EOF\n");
                              exit(0);
                    }
          }
}

writer()
{
 char buf[65536];
 int  n,x;
          close(p[0]);
          for(x=0;x<1<<10;x++)
          {
                    n=write(p[1],buf,65536);
                    if (n<0)
                    {
                              if (errno==EINTR)
                                        fprintf(stderr,"EINTR\n");
                              else
                              {
                                        perror("pipe write");
                                        exit(-1);
                              }
                              continue;
                    }

                    if (n<65536)
                              fprintf(stderr,"pipe short write %d\n ",n);
          }
}
```

Submission: Problems #1 and #2: only clean, computer-printed sheets. Do not fill out this sheet by hand. No hand-

written corrections!

Problem 3 -- Fixing your cat

Revisit your concatenation program from Unit 1. Make sure to correct any mistakes with error handling/reporting and/or 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, along with setting up two pipes, and the appropriate I/O redirection. It is recommended that your original program be the parent of both the grep and the more children (as opposed to using grandchildren). Each input file will cause a new pipeline to be created by the parent.

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, write to stderr the total number of files and bytes processed thus far, and then continue to the next input file.

Is this a fairly silly exercise? Perhaps. One could accomplish the same thing with existing UNIX utilities in a number of ways. But, this will give you practice with signal handlers, pipes, and potentially setjmp depending on how you choose to code it.

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 64K long, so that the pipe buffers 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 messages, 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. 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.