PSET 4

Jonathan Lam
Prof. Hakner
ECE 357
Operating Systems
10/30/19

Problem 1 – Signal Numbers & Behavior

The answers below apply to Linux 5.3.7 x86_64.

a) Which signal is generated for all foreground processes attached to a terminal session when that terminal receives a Ctrl+Z character?

SIGTSTP (i.e., signal number 20)

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

Ctrl+\

(This is SIGQUIT, found using stty -a.)

c) What UNIX command would I use to make Ctrl+I the character which causes SIGINT to be sent?

```
stty intr ^I
```

d) I send a certain process signal #60 on a Linux system, using the kill syscall, 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.

di) How? Include a code snippet to unblock signal #60.

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, 60);
sigprocmask(SIG_UNBLOCK, &set, NULL);
```

dii) 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?

Signal 60 is one of the real-time signals, so it would queue. Therefore, we can expect the handler function to be called 32 times.

However, the a standard signal like signal 2 (SIGINT) doesn’t queue, so it would only be called once (on top of the fact that it would exit the program if its handler was not changed from the default terminate action). This is because the pending signals mask is only a bitmap, and doesn’t hold any additional information about a signal (e.g., quantity) except that that one or more is pending.

Problem 2 - Interrupted and Restarted System Calls

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

This message is printed only if there is the SIGUSR1 signal sent to the process after write has written some bytes to the pipe but has not finished writing (returned). If the write syscall is interrupted before it has written any bytes, or if the write syscall is not interrupted by a SIGUSR1 signal at all, then the message is not written. It’s erratic because of the asynchronity of the message sending and the read/write/sleep cycles of the child processes.

b) Do you notice any pattern to the integer reported with the “pipe short write” messages? Explain?

All of the integers are integer multiples of 4096, greater than 0 and less than 65536. This makes sense, since write is buffered to 4K for files (including pipes, which act like files).

c) What happens if you change sa_flags to 0? Why?

This would remove the SA_RESTART flag, which causes an automatic restart if a signal was received before any data was written to the pipe (whereas SA_RESTART causes write to silently fail and “retry” if a signal is received before writing any data, since there shouldn’t be any negative effects).

This change causes write to return -1 and set errno to EINTR if 0 bytes have been written, which causes “EINTR” to be written to stderr.

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

This causes the program to never terminate. This is because by default, a child creates a zombie after termination. However, the signal(SIGCHILD,SIG_IGN) causes no zombie to be created in Linux. The only way for the parent to exit is when the kill syscall fails. If the zombie doesn’t exist (as in the original program code), then the kill syscall fails; however, if the zombie does exist, then the kill continues to send the signal to the zombie without error (as the manpage says that a zombie that has not been recalled with the wait syscall is a valid existing process), so the parent never terminates.

Jonathan Lam
Prof. Hakner
ECE 357
Operating Systems
11/6/19

```
// catgrepless.c -- uses less b/c less is more
// (and b/c less always goes into int. mode and doesn't "skip" short files)

#include <errno.h>
#include <fcntl.h>
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define BUF_SIZE 4096

#define ERR_FAT(prog, op, ctx, msg) {\
    dprintf(2, "%s: ERROR: %s \"%s\": %s\n", prog, op, ctx, msg);\
    exit(EXIT_FAILURE);\
  }

// handle broken pipe (SIGPIPE) in grep process (when more exits first);
// when encountered, write debug data and end this process
jmp_buf jmp_env;
void sigpipe_hand(int sig) { siglongjmp(jmp_env, 1); }
void sigint_hand(int sig) { siglongjmp(jmp_env, 2); }

int main(int argc, char **argv) {

  char buf[BUF_SIZE], *pattern, **infile, **gargv, **margv;
  int gpid, mpid, wstatus, ftg[2], gtm[2], // file-to-grep, grep-to-more pipes
      rlen, wlen, wtot, ifd, ofd, bp, jstatus; // jump status

  // assume input of "catgrepmore pattern infile1 [...infile2...]"
  if(argc < 3)
    ERR_FAT(argv[0], "parsing arguments", "[too few arguments]",
            "Usage:\n\tcatgrepmore pattern infile1 [...infile2...]");
  pattern = argv[1];
  infile = argv+2;

  // sets up args for execing children
  // actually uses less instead of more
  gargv = (char *[3]) {"grep", pattern, NULL};
  margv = (char *[2]) {"less", NULL};

  // initialize bytes/files processed, ifd/ofd to -1 to indicate not open
```

```
bp = 0, ifd = ofd = -1;

// handle SIGPIPE, SIGINT: should only be called after loop below begins
// SIGPIPE may occur if more exists before grep has finished writing; for
// this, simply continue to the next file. For SIGINT, do the same but also
// print out number of bytes/files processed. Also make sure all ifd/ofd
// are closed (if opened).
if((jstatus = sigsetjmp(jmp_env, 1)) != 0) {
  // make sure children are dead; this is not necessary for more, but less
  // persists for SIGINT
  if(kill(mpid, SIGTERM) < 0)
    ERR_FAT(argv[0], "kill: more after signal", *infile, strerror(errno));
  waitpid(mpid, &wstatus, 0);
  waitpid(gpid, &wstatus, 0);

  // make sure fds closed (if opened)
  if(ifd != -1 && close(ifd) < 0)
    ERR_FAT(argv[0], "close: input file", *infile, strerror(errno));
  if(ofd != -1 && close(ofd) < 0)
    ERR_FAT(argv[0], "close: output pipe from parent for input file",
            *infile, strerror(errno));

  // print message if Ctrl+C
  if(jstatus == 2)
    dprintf(2, "Warning: SIGINT encountered. %d bytes/%d files processed.\n",
            bp, infile-argv-1);
  infile++;
}
signal(SIGINT, sigint_hand);
signal(SIGPIPE, sigpipe_hand);

for(; *infile; infile++) {

  // pipe 1 is from this program to grep; pipe 2 is from grep to more
  // pipe 2 is created first because to involve fewer file closings
  if(pipe(gtm) < 0)
    ERR_FAT(argv[0], "pipe: creating pipe 2", *infile, strerror(errno));
  switch(mpid = fork()) {
    case -1:
      ERR_FAT(argv[0], "fork: to more child", *infile, strerror(errno));
      break;
    case 0:
      signal(SIGINT, SIG_DFL);
      signal(SIGPIPE, SIG_DFL);

      if(dup2(gtm[0], 0) < 0)
        ERR_FAT(argv[0], "dup2: pipe 2 read to more child in", *infile,
                strerror(errno));
      if(close(gtm[0]) < 0)
        ERR_FAT(argv[0], "close: pipe 2 read in more child", *infile,
                strerror(errno));
```

```
    // this is not used, close
    if(close(gtm[1]) < 0)
      ERR_FAT(argv[0], "close: pipe 2 write in more child", *infile,
              strerror(errno));

    if(execvp(*margv, margv) < 0)
      ERR_FAT(argv[0], "execvp: more", *infile, strerror(errno));
  default:
    if(close(gtm[0]) < 0)
      ERR_FAT(argv[0], "close: pipe 2 read in parent", *infile,
              strerror(errno));
}

if(pipe(ftg) < 0)
  ERR_FAT(argv[0], "pipe: creating pipe 1", *infile, strerror(errno));
ofd = ftg[1];
switch(gpid = fork()) {
  case -1:
    ERR_FAT(argv[0], "fork: to grep child", *infile, strerror(errno));
    break;
  case 0:
    signal(SIGINT, SIG_DFL);
    signal(SIGPIPE, SIG_DFL);

    if(dup2(ftg[0], 0) < 0)
      ERR_FAT(argv[0], "dup2: pipe 1 read to grep child in", *infile,
              strerror(errno));
    if(dup2(gtm[1], 1) < 0)
      ERR_FAT(argv[0], "dup2: pipe 2 write to grep child out", *infile,
              strerror(errno));

    if(close(ftg[0]) < 0)
      ERR_FAT(argv[0], "close: pipe 1 read in grep child", *infile,
              strerror(errno));
    if(close(ftg[1]) < 0)
      ERR_FAT(argv[0], "close: pipe 1 write in grep child", *infile,
              strerror(errno));
    if(close(gtm[1]) < 0)
      ERR_FAT(argv[0], "close: pipe 2 write in grep child", *infile,
              strerror(errno));

    if(execvp(*gargv, gargv) < 0)
      ERR_FAT(argv[0], "execvp: grep", pattern, strerror(errno));
  default:
    if(close(ftg[0]) < 0)
      ERR_FAT(argv[0], "close: pipe 1 read in parent", *infile,
              strerror(errno));
    if(close(gtm[1]) < 0)
      ERR_FAT(argv[0], "close: pipe 2 write in parent", *infile,
              strerror(errno));
```

```
    }

    if((ifd = open(*infile, O_RDONLY)) < 0)
      ERR_FAT(argv[0], "open: file for reading", *infile, strerror(errno));

    while(rlen = read(ifd, buf, BUF_SIZE)) {
      if(rlen < 0)
        ERR_FAT(argv[0], "reading of input file", *infile, strerror(errno));

      wtot = wlen = 0;
      while((wtot += wlen) < rlen) {
        if((wlen = write(ofd, buf+wtot, rlen-wtot)) < 0)
          ERR_FAT(argv[0], "writing of input file to pipe to grep", *infile,
                  strerror(errno));
        bp += wlen;
      }
    }

    if(close(ofd) < 0)
      ERR_FAT(argv[0], "closing output pipe", *infile, strerror(errno));
    if(close(ifd) < 0)
      ERR_FAT(argv[0], "closing input file", *infile, strerror(errno));
    ifd = ofd = -1;

    // no need to handle wait status
    waitpid(gpid, &wstatus, 0);
    waitpid(mpid, &wstatus, 0);
  }

  exit(EXIT_SUCCESS);
}
```

Example test case: ((arch)linux 5.3.8)

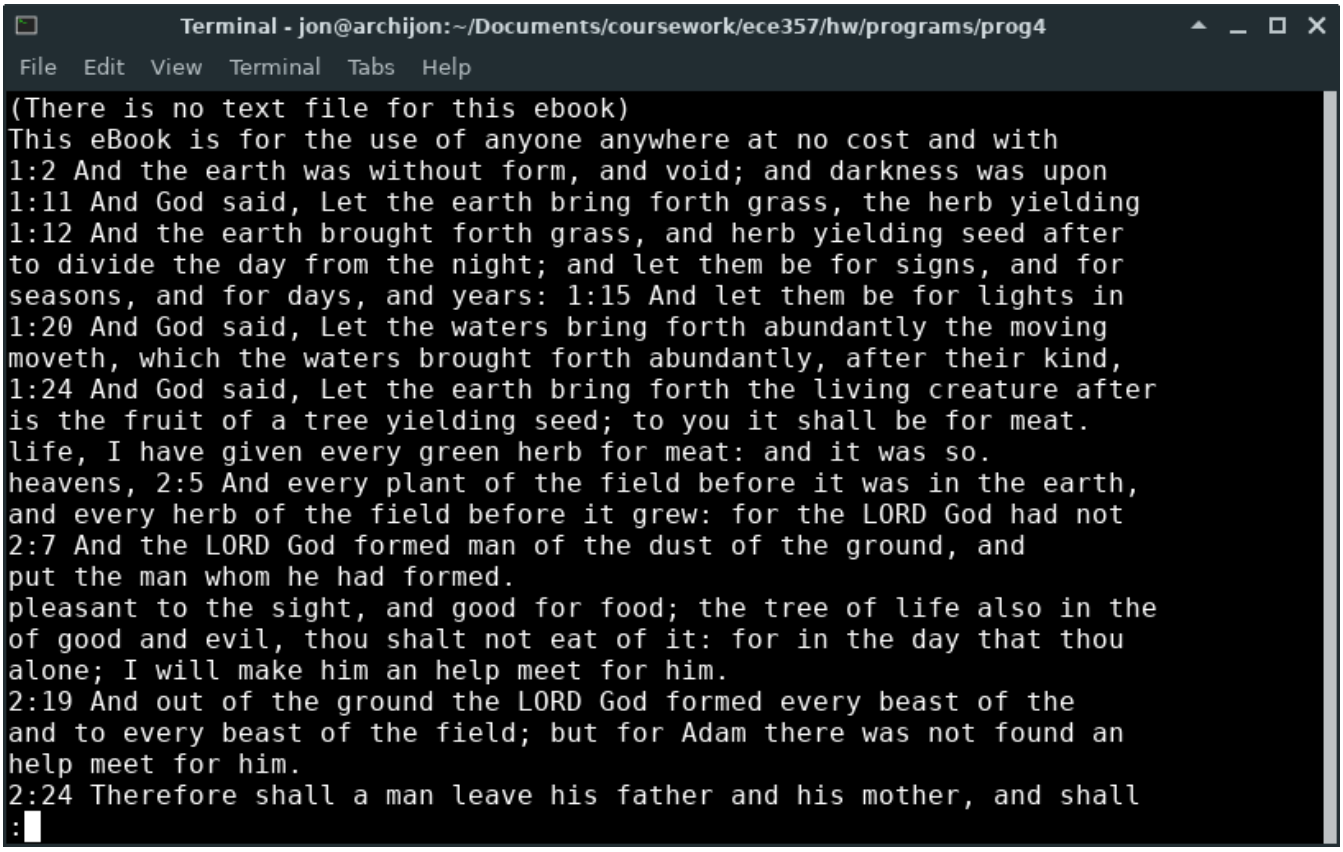
shell command:

```
[jon@archijon prog4]$ ./catgrepless for bible.txt kitty.c catgrepless.c
```

press:

```
q, Ctrl+C, q
```

screenshots: (i.e., this is just less working normally, quitting after each q or Ctrl+C)



Terminal - jon@archijon:~/Documents/coursework/ece357/hw/programs/prog4

File Edit View Terminal Tabs Help

```
  for(fnamep = fnames, ++argv; --argc; ++argv)
      errop = "creating (for writing)", errctx = out_file;
  for(fnamep = fnames; *fnamep; fnamep++) {
        errop = "opening (for reading)", errctx = *fnamep;
      // account for partial write scenario; most likely due to a pipe/socket
        for(bufp = buf; bufp-buf < rlen; bufp++)
    // report bytes transferred for file to stderr
(END)
```

Terminal - jon@archijon:~/Documents/coursework/ece357/hw/programs/prog4

File Edit View Terminal Tabs Help

```
  // sets up args for execing children
  // SIGPIPE may occur if more exists before grep has finished writing; for
    // make sure children are dead; this is not necessary for more, but less
    // persists for SIGINT
      ERR_FAT(argv[0], "close: output pipe from parent for input file",
  for(; *infile; infile++) {
    switch(mpid = fork()) {
        ERR_FAT(argv[0], "fork: to more child", *infile, strerror(errno));
    switch(gpid = fork()) {
        ERR_FAT(argv[0], "fork: to grep child", *infile, strerror(errno));
      ERR_FAT(argv[0], "open: file for reading", *infile, strerror(errno));
(END)
```

stderr:

```
        Warning: SIGINT encountered. 531895 bytes/2 files processed.
```