19 Lecture: Terminal IO Management

19.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn4	mytar	Mon	Nov 27	23:59	
asgn5	mytalk	Fri	Dec 1	23:59	
lab07	forkit	Mon	Dec 4	23:59	
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Be working on Asgn4 (now's the chance to make it good.)
- `/bin/pwd` under linux calls `getcwd(3)` which reads the `/proc` filesystem

  56 submitted.17 failed to build

19.1.1 Example: A simple clock

See the code in figure 79.

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

void ticker(int signum) {
  static int num=0;
  if( ++num%2 )
    printf("Tick\n");
  else
    printf("Tock\n");
}

int main(int argc, char *argv[]) {
  struct sigaction sa;
  struct itimerval itv;

  /* first set up the handler */
  sa.sa_handler = ticker;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  if ( -1 == sigaction(SIGALRM, &sa, NULL)) {
    perror("sigaction");
    exit(1);
  }

  /* then set up the timer: one signal/second */
  itv.it_interval.tv_sec  = 1;
  itv.it_interval.tv_usec = 0;
  itv.it_value.tv_sec  = 1;
  itv.it_value.tv_usec = 0;
  if ( -1 == setitimer(ITIMER_REAL, &itv, NULL)) {
    perror("setitimer");
    exit(1);
  }

  /* now await developments */
  for(;;)
    pause();


  return 0;
}
```

Figure 79: A simple clock

19.2 Terminal IO Managament

We talk about "The Terminal" a lot, but what is one?

> For the purposes here, a terminal is any interactive IO device from an xterm to a real tty

19.2.1 Why do we care?

Stevens Chapter 18.

Terminal processing is messy because of the amount of variability in terminals:

- does it have lowercase?
- can it back up (backspace)?
- can it scroll up?
- does it support tabs?
- etc.

Most IO is processed in CANONICAL mode: Input is assembled into lines and returned at that point.

There is a limit to the line size (`MAX_CANON`) that can be read in Canonical mode. (usu. beeps when filled)

In addition:

- (usually) characters are echoed
- (usually) backspaces are honored
- (usually) signals are generated from special characters

This corresponds to *cooked* mode.

19.2.2 ioctl() — the kitchen sink

ioctl() is the traditional catchall for device manipulation. Its semantics are defined by each particular device driver.

```
#include <sys/ioctl.h>

int ioctl(int d, int request, ...)

[The  "third"  argument  is  traditionally char *argp, and
will be so named for this discussion.]
```

19.2.3 termios — ioctl() rationalized

```
#include <termios.h>
#include <unistd.h>

int tcgetattr ( int fd, struct termios *termios_p );
int tcsetattr ( int fd, int optional_actions,
                        struct termios *termios_p );
int tcdrain ( int fd );
int tcflush ( int fd, int queue_selector );
```

struct termios has:

```
tcflag_t c_iflag;      /* input modes */
tcflag_t c_oflag;      /* output modes */
tcflag_t c_cflag;      /* control modes */
tcflag_t c_lflag;      /* local modes */
cc_t c_cc[NCCS];       /* control chars */
```

`tcdrain()` waits until all output written to the object referred to by fd has been transmitted.

`tcflush()` discards data written to the object referred to by fd but not transmitted, or data received but not read, depending on the value of queue_selector:

`TCIFLUSH`	flushes data received but not read.
`TCOFLUSH`	flushes data written but not transmitted.
`TCIOFLUSH`	flushes both data received but not read, and data written but not transmitted.

19.2.4 The `tcsetattr()` action flags

tcsetattr() action flags:

TCSANOW	the change occurs immediately.
TCSADRAIN	the change occurs after all output written to fd has been transmitted. This function should be used when changing parameters that affect output.
TCSAFLUSH	the change occurs after all output written to the object referred by fd has been transmitted, and all input that has been received but not read will be discarded before the change is made.

19.2.5 The termios flag sets

There are an improbable number of flags, (see Chapter 18 or the man page) but manipulation of them is the same. `tcflag_t` is a bitfield and can be manipulated as usual.

See also table 11.3 in Stevens.

c_iflag input flags — things on the input line (e.g. `IGNPAR`, ignore parity).

c_oflag output flags — things on the output line (e.g. `OLCUC`, map lowercase to uppercase)

c_cflag control flags — control issues (modem, e.g.) (e.g. `PARODD`, expect odd parity (else even))

c_lflag local flags — local processing. (e.g.

`ECHO`	—	enable echoing or not
`ICANON`	—	Canonical Mode or not
`ISIG`	—	Generate Signals, or not

19.2.6 The termios character array

The `c_cc` array allows the program to access (and set) many of the special characters used by the tty line driver. This includes such characters as `ERASE`, the delete character, or `INTR`, the character that generates a `SIGINT`.

`NL` and `CR` cannot be changed.

19.3 Non-canonical IO

What you really need to know:

First turn off the `ICANON` bit. Now what?

Now the system won't buffer lines, but we don't really want `read()` to return once for each byte. (remember?)

The solution: Two more entries in `c_cc`:

`c_cc[VMIN]`	—	Minimum amount of data to return
`c_cc[VTIME]`	—	How long to wait ($\times 0.1s$)

`TIME` of zero means wait forever.

	MIN > 0	MIN $= 0$
TIME > 0	TIME is an interbyte timer; Returns when either MIN characters have been read, or TIME has elapsed between characters. Caller can block indefinitely.	TIME is a read timer Returns when TIME has elapsed.
TIME $= 0$	returns when MIN characters have been read Caller can block indefinitely	returns immediately.

19.4 Termios Example: turning off echoing

Figures 80, 81, and 82, and 83 show a program that turns off echoing and copies its input to its output. This program is still in canonical mode.

19.5 AnOTHer fUn eXAMPLe using non-canonical IO

Figure 84 shows another example where the case of echoed letters is perturbed randomly. This uses non-canonical IO so the weirdness is visible in real time. (There's really no end to the fun this kind of thing can produce.)

19.6 Example: expanding tabs

For this class, most of what you're going to be interested in is in the local flags, but the input and output flags can be very interesting. Consider the following:

One of the fields in the output flags (`c_oflag`) is `TABDLY` to control the handling of tabs. Possible values are `TAB0`, `TAB1`, `TAB2`, and `XTABS`.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <termios.h>

void echo_off(int fd);
void echo_on(int fd);
void cat (FILE *in, FILE *out);


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

  echo_off(fileno(stdin));
  cat(stdin,stdout);
  echo_on(fileno(stdin));

  return 0;
}
```

Figure 80: A `cat` without echoing: `main()`

```
void echo_off(int fd) {
  struct termios tio;

  tcgetattr(fd, &tio);           /* get the current values */

  tio.c_lflag &= ~ECHO;          /* unset the ECHO bit */

  tcsetattr(fd, TCSADRAIN, &tio); /* make new values current */
}
```

Figure 81: A `cat` without echoing: `echo_off()`

```
void echo_on(int fd) {
  struct termios tio;

  tcgetattr(fd, &tio);           /* get the current values */

  tio.c_lflag |= ECHO;           /* set the ECHO bit */

  tcsetattr(fd, TCSADRAIN, &tio); /* make new values current */

}
```

Figure 82: A `cat` without echoing: `echo_on()`

```
void cat (FILE *in, FILE *out) {
  /* copy the infile to the outfile */
  int c;

  while ( (c=getc(in)) != EOF )
    putc(c,out);

}
```

Figure 83: A `cat` without echoing: `cat()`

Why delay? Consider a real teletype. The head has to move.

`XTABS` causes it to expand tabs to spaces. Using this, we can write `expand` in such a way that we don't have to do the work. The program is in Figures 85, and 86 (the `cat()` function Figure 87 is the same as above).

19.7 But it only works on ttys

Look at Figure 88. Isn't that disappointing?

You can try some tty-specific thing and look for `ENOTTY` in `errno`, or

You can check with `isatty()`: `int isatty ( int desc );`

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <termios.h>

#define CTRL_D 4

int upcase_some(int c) {
    c = tolower(c);
    if (random() > RAND_MAX/2)
        c = toupper(c);
    return c;
}

int main() {
    struct termios old, new;
    int c;

    /* get the attributes */
    if ( -1 == tcgetattr(STDIN_FILENO, &old) ) {
        perror("stdin");
        exit ( -1 );
    }

    new = old;
    new.c_lflag &= ~ECHO;
    new.c_lflag &= ~ICANON;
    new.c_cc[VMIN] = 1;
    new.c_cc[VTIME] = 0;
    if ( -1 == tcsetattr(STDIN_FILENO, TCSADRAIN, &new) ) {
        perror("stdin");
        exit ( -1 );
    }

    /* because we're not in canonical mode we won't be able to
     * detect EOF in the normal way. We'll just check for ^D.
     */
    while (CTRL_D != (c=getchar()))
        putchar(upcase_some(c));

    if ( -1 == tcsetattr(STDIN_FILENO, TCSADRAIN, &old) ) {
        perror("stdin");
        exit ( -1 );
    }
    return 0;
}
```

Figure 84: Echo with random case

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <termios.h>

static tcflag_t set_tabs(int fd, tcflag_t tabstyle);
void cat (FILE *in, FILE *out);

int main(int argc, char *argv[]){
  tcflag_t tabs;

  tabs = set_tabs(fileno(stdout), XTABS);

  cat(stdin,stdout);

  set_tabs(fileno(stdout), tabs);

  return 0;
}
```

Figure 85: A termio-based `expand`: `main()`

```
static tcflag_t set_tabs(int fd, tcflag_t tabstyle) {
  /* Set the tab handling to the given style and return
   * the old tab handling mode.
   *   Assumes the given file descriptor is a tty, and
   * aborts the program if not.
   */
  struct termios tio;
  tcflag_t oldtabs;

  if ( tcgetattr(fd, &tio) ) {  /* get the current termio info */
    perror("tcgetattr");
    exit(-1);
  }

  oldtabs = tio.c_oflag & TABDLY; /* select current tab processing */

  tio.c_oflag &= ~TABDLY;       /* mask out current tab handling */
  tio.c_oflag |= tabstyle;      /* set to the given style         */

  /* turn on the new attributes */
  if ( tcsetattr(fd, TCSADRAIN, &tio) ) {
    perror("tcsetattr");
    exit(-1);
  }

  return oldtabs;               /* return the old version */
}
```

Figure 86: A termio-based `expand`: `SetTabs()`

```
void cat (FILE *in, FILE *out) {
  /* copy the infile to the outfile */
  int c;

  while ( (c=getc(in)) != EOF )
    putc(c,out);

}
```

Figure 87: A termio-based `expand`: `cat()`

```
% myexpand
Hello   tabbing world
Hello   tabbing world

% myexpand | tr ' ' '+'
tcgetattr: Invalid argument

% myexpand > outfile
tcgetattr: Inappropriate ioctl for device
%
```

Figure 88: termio expand only works on ttys