16 Lecture: Signals, POSIX Signals, Timers and Alarms

Outline:

16.1 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.

- Today: From last time: the stats, chdir, chmod, chown, holes
- Are you reading the feedback and/or running the test cases?
- Back to umask
- lab05 How to do mypwd()?
- Midterm archive out.
- You will always lose points for unchecked malloc()s
- Bring questions Wedensday and Friday.
- Midterm is scheduled for next Monday.
  - All (well, most) prototypes and constants you'll need will be on the back of the exam.

16.2 stuff to talk about

- Where does file ownership come from?
- Number of inodes is limited

16.3 Resource Limitations

Consider what goes into creating a file:

- Inode
- Dirent
- Space to store data (exept holes)

16.4 A note on the lab

Implementing:

```
char *getcwd(char *buf, size_t size);
        NULL and ERANGE on error
```

Inodes are unique to a filesystem, but inode and device are sufficient.

- current working directory and chdir()
- opendir, readdir, closedir
- Inodes on mounted filesystems.

16.5 getpwd()

```
The _only_ part of the struct dirent that is reliable is the d_name
field.  Everything else requires one of the stats.

Why?  Well, what's happened is that you've encountered a mountpoint.
Readdir() reads the directory file itself, so it's reading the name and
inode number that are stored in the file.  It has no knowledge of
mounted filesystems, though, so it doesn't know that there's a
filesystem mounted on that directory.  In the struct dirent you're
seeing the inode number of the directory that the other fs was mounted
on top of.  When you call stat(), though, it consults the kernel's mount
tables and gives you the inode/device/rest of the info pertaining to the
root of the mounted filesystem which is what you actually want.

I hope this helps.
```

16.6 The rest of the quarter

16.6.1 Subjects

- Signals
- tty line disciplines (IO device management) (maybe)
- Process Management (fork(), the exec()s, process groups, etc.)
- Process Environments (env, resource limits)

- Other useful stuff (strtok(), setjmp(), longjmp())

The next two subjects (Terminal IO and Signals) show the most differentiation between the UNIX family members.

16.7 Asynchronous Events and signals

In the systems programming realm we have to give up the precious illusion that programs exist in isolation. Sometimes things come up. (e.g. a child process exits or a seg fault.) Sometimes it's events we asked for (SIGALRM or SIGVTALRM).

A Signal is a software interrupt indicating some "important" condition. There are three possible reactions to a signal:

`SIG_IGN`	Ignore it
`SIG_DFL`	default (ignore or die)
catch	whatever you want

- From Hardware: SIGILL, SIGPWR, SIGFPE
- From Software: via `kill(1)`, `kill(2)`, `killpg(2)`, `raise(3)`

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
int raise (int sig);
```

- From Terminal Line discipline (non-canonical (cooked) mode processing)

For all default actions, see page 266 in Stevens (table 10.1)
Behavior is undefined after ignoring certain signals (E.g., SIGSEGV or SIGFPE)

16.8 Unreliable Signal Handling: `signal(2)`

- one shot vs. repetition

```
SIGNAL(2)           Linux Programmer's Manual           SIGNAL(2)

NAME

       signal - ANSI C signal handling

SYNOPSIS
       #include <signal.h>

       void (*signal(int signum, void (*sighandler)(int)))(int);
```

Possible values:

- `SIG_IGN`
- `SIG_DFL`

- pointer to handler
- `SIG_ERR` on error

SIGKILL and SIGSTOP cannot be caught or ignored.

16.8.1 Aside: Function pointers

That horrible prototype could better be re-written:

```
typedef void (*sigfun)(int);
sigfun signal(int signum, sigfun handler);
```

An example of signal handling code can be found in Figure 69.

16.9 What is unreliable about it?

To understand this, we have to look at how signals are delivered.

16.10 Signal Delivery

Because signals are asynchronous events, it is hard to say too much about when they will be delivered. Generally, however, processes that have asked to be notified of events want to know about them, so the system tries to deliver them quickly.

Certain "slow" system calls (those that may block forever: e.g. `wait()`, `read()`, `write()`, `sleep()`, `pause()`) are interrupted when a signal is delivered.

Some implementations (4.3BSD, e.g.) automatically restart some of these calls. POSIX (below) allows for such behavior, but doesn't enforce it. (What if you don't want it to restart?)

What happens when a signal is delivered:

- If the handler is set to `SIG_IGN`, the signal is ignored.
- If the handler is set to `SIG_DFL`, the default action is performed.
- If the handler is set to a function:
  1. The handler is (usually) set to `SIG_DFL` (or implementation-dependent blocking)
     (Compare Linux to Solaris — Linux implements the BSD4.3 blocking semantics, Solaris restores the default handler.)
  2. The signal handler is called with argument `signum`

The resetting of the signal action upon delivery of a signal causes the unreliability: Even if the signal handler is immediately re-registered, there is a window of time in which another instance of the signal will have the default action, not the intended one.

Also, it is not possible to find out what is currently being done with a signal without changing it (however briefly). You can do something like the code in Figure 70, but you can't always get it.

These *race conditions* make signal handling in this manner unreliable.

```
/*
 * sigtst:
 *
 * Author: Dr. Phillip Nico
 *         Department of Computer Science
 *         California Polytechnic State University
 *         One Grand Avenue.
 *         San Luis Obispo, CA  93407  USA
 *
 * Email:  pnico@calpoly.edu
 *
 * Revision History:
 *         $Log:$
 */

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

#include <signal.h>

typedef void (*sigfun)(int);

void handler(int sig) {
  /* Works once */
  printf("Hello, I caught a SIGINT.  Neener, neener!");
  fflush(stdout);
}

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

  if ( ( old = signal(SIGINT,handler) ) == SIG_ERR ) {
    perror("signal");
    exit(-1);
  }

  for (;;)
    /* nothing */;

  return 0;
}
```

Figure 69: A program that catches SIGINT

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

#include <signal.h>
#include <sys/types.h>

typedef void (*sigfun)(int);

static int ctr = 0;

void temp(int sig) {
  signal(SIGINT,temp);
  ctr++;
}


int main(int argc, char *argv[]){
  int i;
  sigfun old;

  ctr = 0;                      /* reset the counter */
  old = signal(SIGINT,temp);    /* get the old value */
  signal(SIGINT,old);           /* put it back */


  #ifdef ONEWAY
  switch ( (int)old ) {         /* cheat a little and act like it's an int */
  case SIG_IGN:
    /* ignore it */
    break;
  case SIG_DFL:
    /* resend the signal */
    kill ( getpid(), SIGINT);
    break;
  default:
    while(ctr--) {              /* run old() for all the missed signals */
      (*old)(SIGINT);
    }
    break;
  }
  #else
  while(ctr--) {                /* re-send all the missed signals */
    kill(getpid(),SIGINT);
  }
  #endif

  return 0;
}
```

Figure 70: Checking the current action with unreliable signals

16.10.1 Afterwards

The program continues merrily on its way. If a system call was interrupted, it returns with an error and sets `errno` to `EINTR`.

Example: if a signal is caught during:

```
c = getchar()
```

`c` will be `EOF`, whether or not the real end of file has been reached, and `errno` will be `EINTR`. (Use `feof(FILE *)` to find out.)

16.11 Tool of the Week: rcs(1)

RCS (Revision Control System) is a version-control system. (not nearly as fancy as SVN, CVS, git, . . . , but requires almost no setup.)

mkdir RCS Create repository

ci -u foo.c Checks in "foo.c" and removes it

ci -u foo.c Checks in "foo.c" and keeps a read-only copy (unlocked)

ci -i foo.c Checks in "foo.c" and keeps a read-write copy (unlocked)

co -u foo.c Checks out "foo.c" read-only (unlocked)

co -l foo.c Checks out "foo.c" read-write (locked)

rcsdiff foo.c shows the difference between this version and the last

rlog foo.c show the log messages.

Emacs interface:

C-x v i Install buffer into RCS

C-x v v Toggle-read-only checks the file in or out

C-c C-c Complete checkin