Signals and Pipes

Learning Objectives

- Understand synchronization issues with respect to signals
- Learn about the client-server model using pipes and named pipes
- Understand basic concepts of servers and daemons

More on Signals

- Two main system calls that deal with signals: `signal()` and `sigaction()`. The `sigaction()` call is consistent across various systems whereas `signal()` is not necessarily consistent across systems but is simpler to use. See man page for details on both.
- Examples:
  - signal-ex1.c
  - signal-ex2.c

Signal Handling after Exec

When a program is exec'd the status of all signals is either default or ignore. The exec functions changes the disposition of any signals that are being caught to their default action (why?) and leaves the status of all other signals alone.
For example:

```
if (fork()==0) { // the child process
    if (signal(SIGINT, SIG_IGN) == SIG_ERR)
        err_ret("failed to set SIGINT behavior");
    if (signal(SIGTSTP, SIG_IGN) == SIG_ERR)
        err_ret("failed to set SIGTSTP behavior");
    execvp(program, argv);
    // the exec'd program will ignore the
    // signals SIGINT and SIGTSTP
```

Signal Handling for an Application

How an application ought to set its signal handling:
An application process should catch the signal only if the signal is not currently being ignored.

```
int sig_int(), sig_quit();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
        signal(SIGQUIT, sig_quit);
```

See code example: signal-ex3.c

Other Signal Issues

- Signals are set to their default after being caught (under Linux and System V UNIX semantics). This can be changed by using `sigaction()` instead of the `signal()` system call.
- What if the program was updating some complicated data structures... then `exit` or `siglongjmp()` is not a good solution. We can set a flag to indicate the interrupt and continue processing. The program can deal with the interrupt later.
- Under the POSIX semantics a process can also block signals so that they are delivered later when it is ready to deal with the signals. (See man pages for `sigaction()` and `sigprocmask()` under Linux).
- What if the parent and the child are both trying to catch an interrupt? They both might then try to read from standard input. That would be confusing, to say the least. The solution is to have the parent program ignore interrupts until the child is done.

Random Rogue and Blocking Signals

- Random rogue: shows how to catch and avoid segmentation faults: signal-ex4.c
- Shows how to block a signal and receive later at a time of our choosing: signal-ex5.c

Signal Handling in the Shell

- How to handle `Ctrl-c` or `SIGINT`?
  - The shell ignores it on startup.
  - The shell catches it on startup and sends appropriate signal to foreground jobs when it catches the signal.
  - The shell catches it on startup, then ignores it when starting a foreground job and catches it again afterwards.
- How to handle `Ctrl-z` or `SIGTSTP`?
  - The shell ignores it on startup.
  - The shell catches it on startup and send appropriate signal to foreground jobs when it catches the signal.
  - The shell catches it on startup, then ignores it when starting a foreground job and catches it again afterwards.
- How do we prevent background jobs from getting affected by `Ctrl-c` and `Ctrl-z`?
- How to implement `fg` and `bg` built-in commands?

Pipes in Unix

- A pipe allows communication between two processes that have a common ancestor.
- A pipe is a half-duplex (data flows in only one direction) FIFO buffer with an API similar to file I/O.

```
#include <unistd.h>

int pipe(int filedes[2]);
//returns filedes[0] for reading, filedes[1] for writing
```

- Reading from a pipe whose write end has been closed causes an End Of File to be returned. Writing to a pipe whose read end has been closed causes the signal `SIGPIPE` to be generated. The write returns with `errno` set to `EPIPE`.
- The size of pipe is limited to `PIPE_BUF`. A write of `PIPE_BUF` or less will not interleave with the writes from other processes. The constant `PIPE_BUF` is defined in the file /usr/include/linux/limits.h
- Examples: hello-pipe1.c, hello-pipe2.c

The Power Of Pipelines

Find the 10 most frequent words in a given text file (and their respective counts).

```
cat Shakespeare.txt | tr -cs "[A-Z][a-z][']" "[\012*]" | tr A-Z a-z |
sort | uniq -c | sort -rn | sed 10q
```

Generate all anagrams from a given dictionary.

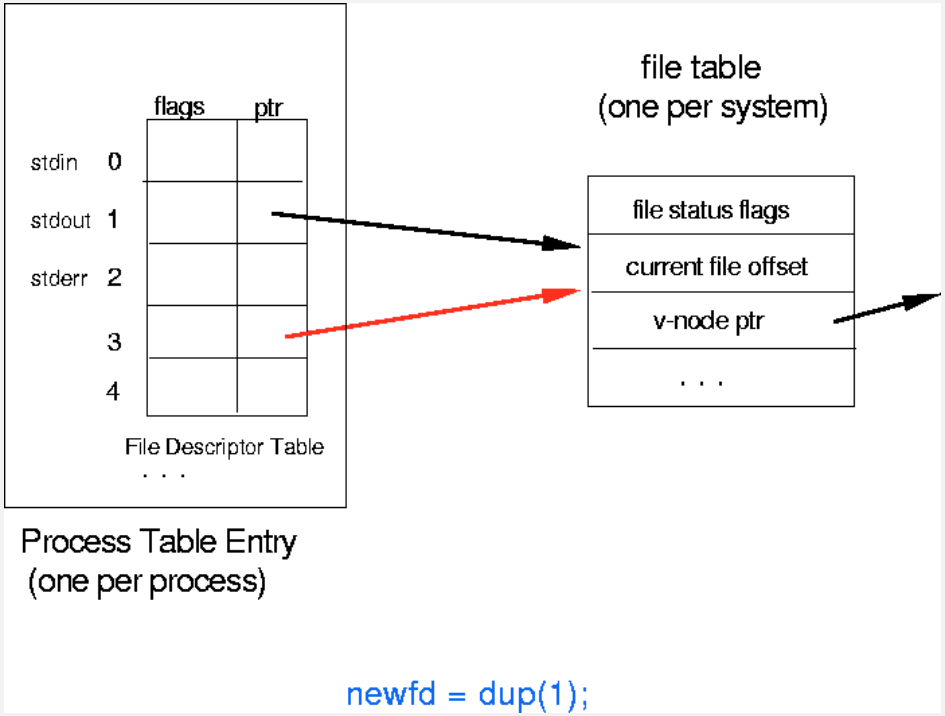
```
sign < $2 | sort | squash | awk '{if (NF > 1) print $0}'
```

```
---squash---
#!/bin/sh
/usr/bin/awk '
$2 != prev  { prev = $2; if (NR > 1) printf "\n"}
            { printf " %s ", $2 }
END         { printf "\n" }'

---sign.c---
#include <stdio.h>
#include <string.h>
#define WORDMAX 101
int compchar(char *x, char *y) { return ((*x) - (*y));}
void main(void)
{
        char thisword[WORDMAX], sign[WORDMAX];

        while (scanf("%s", thisword) != EOF) {
                strcpy(sign, thisword);
                qsort(sign, strlen(sign), 1, compchar);
                printf("%s %s\n", sign, thisword);
        }
}
```

Pipes and I/O redirection



- See I/O redirection example 1: dup1.c
- See I/O redirection example 2: dup2.c
- We can combine I/O redirection with pipes to create pipelines of commands. See code examples
  - pipe2.c
  - pipe3.c
  - pipe4.c

Process Groups

- A process group is a collection of one or more processes. Each process group has a unique process group ID.
- Each process under Unix belongs to a process group. Each process group *may* have a process group leader. The leader is identified by having its process group ID equal to its process ID.
- The two related systems calls allow a process to find out its process group's ID and to change its process group.

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

pid_t getpgrp(void);
int set setpgid(pid_t pid, pid_t pgid);
```

- A process can set the process group ID of itself or its children. Furthermore it cannot change the process group ID of its child after the child calls the `exec()` system call.
- Only processes in the foreground group can perform I/O from the terminal. A process that is not in the foreground group will receive a `SIGTTIN` or `SIGTTOU` signal when it tries to perform I/O on the terminal, which will stop the process.

See several examples in the folder process-groups under `synchronization-part2` lab folder.

Named Pipes (FIFOs)

- Named Pipes (FIFOs) allow arbitrary processes to communicate.

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

int mkfifo ( const char *pathname, mode_t mode );

mkfifo pathname
```

- If we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.
- The reads/writes can be made blocking or non-blocking.
- If we have multiple writers for a FIFO, atomicity is guaranteed only for writes of size no more than PIPE_BUF.

Uses of FIFOs

- Can be used by shell commands to pass data from one shell pipeline to another, without creating intermediate temporary files.

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

A real example of a nonlinear pipeline:

```
wc < fifo1 &
cat /usr/share/dict/words | tee fifo1 | wc -l
```

- Look at the following simple example using one fifo: hello-fifo.c
- Look at the following example for a two-way communication using two fifos: fifo-talk.c

Client Server Communication Using FIFOs

- The server creates a FIFO using a pathname known to the clients. Clients write requests into this FIFO.
- The requests must be *atomic* and of size less than `PIPE_BUF`, which is defined in `limits.h` standard header file.
- The server replies by writing to a client-specific FIFO. For example, the client specific FIFO could be `/tmp/serv1.xxxxx` where `xxxxx` is the process id of the client.

See wikipedia: client server model for more information on the client-server model.

Fifo Server Client Example

- Shows a simple server that listens on a fifo to any client that connects to it and chats with the client.
- Server code: fifo-server.c
- Client code: fifo-client.c

Daemons and Servers

- A Daemon is a process that lives for a long time. Daemons are often started when the system is bootstrapped and terminate only when the system is shutdown. They run in the background because they don't have a controlling terminal.
- A Server is a process that waits for a client to contact it, requesting some type of service. Typically, the server then sends some reply back to the client.
- A common use for a daemon process is as a server process.

Writing a Daemon

Coding rules for a daemon.

1. Call `fork()` and have the parent exit.
2. Call `setsid` to create a new session. Then the process becomes a session leader of a new session, becomes the process group leader of a new process group and has no controlling terminal.
3. Change the current working directory to the root directory. Alternately, some daemons might change the working directory to some specific location.
4. Set the file creation mask to 0, so it does not use anything inherited from its parent process.
5. Unneeded file descriptors should be closed.
6. Use the syslog system call to log messages to the syslogd daemon, which is a central facility for all daemons to log messages. See man page for syslog (`man 3 syslog`) for more details.

```c
#include    <stdlib.h>
#include    <unistd.h>
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>

int daemon_init(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        return(-1);
    else if (pid != 0)
        exit(0);    /* parent goes bye-bye */

    /* child continues */
    setsid();       /* become session leader */

    chdir("/");     /* change working directory */

    umask(0);       /* clear our file mode creation mask */

    return(0);
}
```