CMSC 216
Introduction to Computer Systems
Lecture 17
Process Control and System-Level I/O

Sections 8.2-8.5, Bryant and O'Hallaron

PROCESS CONTROL (CONT.)

Signals

- A message to a process to notify it of an event
- Kernel notifies processes of many events:
  - **`SIGSEGV`**: segmentation violation (aka segfault)
  - **`SIGFPE`**: floating point exception
  - **`SIGCHLD`**: child process stopped/terminated
- Users can trigger sending of signals:
  - **`SIGINT`**: interrupt (Ctrl C)
  - **`SIGQUIT`**: quit and dump core (Ctrl \)

Signals, cont.

- Processes can also send signals to each other (via the kernel)
- Processes have default actions when receiving signals (sometimes ignore, sometimes quit)
- There are mechanisms for processes to block signals, but two (**`SIGKILL`** and **`SIGSTOP`**) can't be blocked
- Signals are sent with the **`kill`** UNIX command, or in a program using the **`kill()`** function
  - they do not always send the signal **`SIGKILL`**!

Tools for working with processes

- **`ps`**: UNIX command to see the current list of processes
  - several different options (**`-e`**, **`-f`**, **`-u`**, -p, etc.)
  - -e to display all the processes
  - -f to display full format listing
  - -u displays processes associated with user name
- **`strace`**: Linux tool to print trace of all system calls a program and its children perform (precedes rest of command line)
- **`pgrep`** *name*: prints pids of processes with *name* in their command lines
- **`pkill`** *name*: sends a signal to all processes with *name* in the command lines
- **`kill`** *pid-list*: sends a signal to all specified processes
  - kill -9 <process_id>
- **`top`**: display current info about system and processes
- Ctrl Z, **`bg`**, **`fg`**, and **`&`**: process backgrounding
- **`/proc`**: a (virtual) part of the filesystem on Linux that has all sorts of info on kernel data structures related to processes, and other OS related system info

Chapter 11, Bryant and O'Hallaron

SYSTEM-LEVEL I/O

UNIX I/O

- Important:
  - Standard I/O vs. Unix I/O
    - Standard I/O (**`FILE *`**, **`fgets()`**, **`printf()`**, etc.) generally features buffers; Unix I/O does not
    - Mixing the two can cause weird things to happen
- A file in UNIX is just a sequence of bytes
- All I/O devices are modeled as files in UNIX
  - disks, network sockets, etc.
- The kernel maintains a file descriptor table, holding information about all open files, for each process
- File descriptors are nonnegative integers used by processes to access files in a process' table

File operations

- A process requests access to a file via an **`open()`** system call; the kernel returns a file descriptor
- The process reads, writes, and moves around the file using the file descriptor and other system calls (**`read()`**, **`write()`**, **`lseek()`**)
- When finished with a file, the process closes the file via the **`close()`** system call
  - All open files are closed by the kernel when a process terminates

File descriptors

- In each process, there is a file descriptor associated with each open file
- Multiple processes can have the same file open; closing a file in one process will not close the file in other processes
- Standard input, output and error have file descriptors 0, 1, and 2, respectively
- Don't use those numbers in your programs: **`STDIN_FILENO`**, **`STDOUT_FILENO`**, and **`STDERR_FILENO`** are provided for you

The `open()` function

- Two forms:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *filename, int flags);
int open(const char *filename, int flags, mode_t mode);
```

  - **`flags`** is the result of a bitwise OR of any of several options:
    - **`O_RDONLY`**: read only
    - **`O_WRONLY`**: write only
    - **`O_RDWR`**: read and write
    - **`O_CREAT`**: create file if it doesn't exist
    - **`O_EXCL`**: cause an error if file already exists
    - **`O_TRUNC`**: if file exists, truncate it to an empty file
    - **`O_APPEND`**: cause each write operation to write to end of file
  - **`mode`** specifies the permissions to be used if/when creating a new file
    - only needed/checked if **`O_CREAT`** is specified
    - we'll use **`0666`**; this allows all users to read and write the file
    - mode is made less permissive by the process' **`umask`**
  - returns -1 on error, or a file descriptor for the opened file

The `close()` function

- When finished with a file, your programs should then release all memory associated with the use of that file using the **`close()`** function

```
#include <unistd.h>
int close(int fd);
```

  - returns 0 if successful, -1 on error
  - errors can occur if you try to free an already freed descriptor, or if something else interrupts the closing operation

A simple example using `open()`

open.c:

```
/* #include statements omitted */

/* same as 0666, but a bit more symbolic */
#define DEF_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)

int main() {
  int fd;

  if ((fd = open("missing-file.txt", O_RDONLY)) < 0)
    perror("can't open missing-file.txt for read");
  else
    close(fd);

  fd = open("file.txt", O_WRONLY | O_TRUNC | O_CREAT, DEF_MODE);
  if (fd < 0)
    perror("can't open file.txt");
  else
    close(fd);

  fd = open("new-file.txt", O_WRONLY | O_APPEND | O_CREAT, DEF_MODE);
  if (fd < 0)
    perror("can't open new-file.txt for write");
  else
    close(fd);

  return 0;
}
```

Running the simple example

```
$ ls -l
total 8
-rw-rw----+ 1 bofh bofh   33 Apr 29 20:53 file.txt
-rwxrwx---  1 bofh bofh 5195 Apr 29 20:52 open
-rw-rw----+ 1 bofh bofh  696 Apr 29 20:52 open.c
$ cat file.txt
My, this is an interesting file.
$ ./open
can't open missing-file.txt for read: No such file or directory
$ ls -l
total 9
-rw-rw----+ 1 bofh bofh    0 Apr 29 20:55 file.txt
-rw-rw----+ 1 bofh bofh    0 Apr 29 20:55 new-file.txt
-rwxrwx---  1 bofh bofh 5195 Apr 29 20:52 open
-rw-rw----+ 1 bofh bofh  696 Apr 29 20:52 open.c
```

Performing I/O

- Typically, a program will use the **`read()`** and **`write()`** functions to perform I/O operations on a file descriptor

```
#include <unistd.h>
ssize_t read(int fd,
                void *buffer, size_t n);
```

  - reads up to **`n`** bytes from **`fd`** into **`buffer`**
  - returns -1 on error, 0 on EOF, or number of bytes read (may be < **`n`**)

```
ssize_t write(int fd,
                const void *buffer, size_t n);
```

  - writes up to **`n`** bytes from **`buffer`** to **`fd`**
  - returns -1 on error, or number of bytes written (may be < **`n`**)

Standard I/O vs. Unix I/O

- Standard I/O (**`FILE *`**, **`fgets()`**, **`printf()`**, etc.) generally features buffers; Unix I/O does not
- Mixing the two can cause weird things to happen; what does this output?

```
printf("u");
write(STDOUT_FILENO, "m", 1);
printf("d\n");
```

- **`mud`**
- Even stranger things happen with reading...

Mixing buffered/unbuffered reads

buf.c:

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

#define BUFFER_SZ 100
static char buffer[BUFFER_SZ];

int main() {
  char line_s[11];
  char line_u[11] = {0};
  setbuffer(stdin, buffer, BUFFER_SZ);
  fgets(line_s, 11, stdin);
  read(STDIN_FILENO, line_u, 10);
  printf("L1: %s\n", line_s);
  printf("L2: %s\n", line_u);
  fgets(line_s, 11, stdin);
  read(STDIN_FILENO, line_u, 10);
  printf("L3: %s\n", line_s);
  printf("L4: %s\n", line_u);
  return 0;
}
```

data.txt:

```
aaaaaaaaaabbbbbbbbbbcccccccccc
ddddddddddeeeeeeeeeeffffffffff
gggggggggghhhhhhhhhhiiiiiiiiii
jjjjjjjjjjkkkkkkkkkkllllllllll
mmmmmmmmmmnnnnnnnnnnoooooooooo
```

Execution:

```
$ ./buf < data.txt
L1: aaaaaaaaaa
L2: kkkkkkkkkk
L3: bbbbbbbbbb
L4: llllllllll
```