CMSC 216
Introduction to Computer Systems
Lecture 19
Process Control and System-Level I/O

Jan Plane & Alan Sussman
{jplane,als}@cs.umd.edu

Administrivia

- Project 4 due Tuesday, Nov. 9
- Project 5 out on Wednesday
- Exam #2 on Thursday, Nov. 18
  - practice exam available by 1 week before
- Read Chapter 10
- Don't forget course projects policy
  - have to make a good faith effort on **all** projects before end of semester
  - that means submit a version that works on at least 75% of public tests

Sections 8.2-8.5, Bryant and O'Hallaron

Process Control

`execve()` example

```
/* #include statements omitted */
extern char **environ;
int main() {
  char *args[] = { "ls", "-l", NULL };
  pid_t child_pid;
  if ((child_pid = fork()) < 0)
    err(EX_OSERR, "fork error");
  if (child_pid) {  /* parent code */
    wait(NULL);
    printf("Parent pid = %d; my child had pid = %d\n",
           getpid(), child_pid);
  }
  else {  /* child code */
    printf("PID %d replacing myself\n", getpid());
    execve("/bin/ls", args, environ);
    err(EX_OSERR, "exec error");  /* why no if statement? */
  }
  return 0;
}
```

Other exec functions

- There are 5 other functions that perform exec operations
  - **l**: use an argument list, terminated by **NULL**
  - **v**: use an argument vector
  - **p**: search the **PATH** list of directories
  - **e**: can specify environment
- `execle(const char *filename, ..., NULL, char * const envp[]);`
- `execv(const char *filename, char * const argv[]);`
- `execl(const char *filename, ..., NULL);`
- `execvp(const char *progname, char * const argv[]);`
- `execlp(const char *progname, ..., NULL);`

How to use the other functions

```
char *filename = "/bin/ls";
char *args[] = { "ls", "-l", NULL };

execle("/bin/ls", "ls", "-l", NULL,
       environ);
execv(filename, args);
execl("/bin/ls", "ls", "-l", NULL);
execvp(args[0], args);
execlp("ls", "ls", "-l", NULL);
```


Why you specify the program twice

- The first time, you tell the system which program to run
- The second time, it's **argv[0]** for that program
  - used to by the program to know what its name is
  - sometimes the same program runs differently under different names
- Can be used for bizarre means...

An admittedly contrived example

```
$ cat sleeper.c
#include <stdio.h>
#include <unistd.h>
int main() {
  printf("PID = %d\n", getpid());
  execlp("sleep", "blargh", "60", NULL);
  return 1;
}
$ ./sleeper &
PID = 674
$ ps -fp 674
UID        PID  PPID  C STIME TTY          TIME CMD
bofh       674 26789  0 22:12 pts/6    00:00:00 blargh 60
```

A simple shell

- With fork, exec, and waiting, we have all the tools we need to build our own shell
- Basic idea: infinite loop with prompt
- Inside loop:
  - read in a command line
  - parse the command line
  - fork; parent waits, child execs command

A simple shell program

```
while (1) {
  /* prompt and read into buffer */
  /* parse buffer into string vector argv */
  switch (fork()) {
    case -1:
      perror("fork error");
      break;
    case 0:
      execvp(argv[0], argv);
      err(EX_OSERR, "exec error");
      break;
    default:
      wait(NULL);
      break;
  }
}
```


Tools for working with processes

- **`ps`**: UNIX command to see the current list of processes
  - several different options (**`-e`**, **`-f`**, **`-p`**, **`-u`**, etc.)
- **`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
- **`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

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+\)
  - **`SIGTSTP`**: terminal stop, (Ctrl+Z)

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`**!

Chapter 10, Bryant and O'Hallaron

SYSTEM-LEVEL I/O

UNIX file management

- 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()` system call

- 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()` system call

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

```
#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;
}
```