CMSC 216
Introduction to Computer Systems
Lecture 22
System-Level I/O and Time Measurement

Larry Herman & Alan Sussman

Notes

- Project 4 secret tests on Grace
- Project 6 posted, public tests on Grace
- Exam #2 returned tomorrow in discussion section
- Read Chapter 10 on System-Level I/O, Reek Section 16.3 on Time Measurement

Chapter 10, Bryant and O'Hallaron

SYSTEM-LEVEL I/O

Interprocess Communication

- Processes can communicate with each other via various means, including signals and pipes
- Signals have a limited "vocabulary" - only 64 signals on some machines; other limitations as well
- Pipes allow arbitrary strings to be written from one process to another
- Two kinds of pipes: FIFOs (or "named pipes", created by **`mkfifo()`**), and anonymous pipes (often just called "pipes", created by **`pipe()`**)

Pipes

- Note: much of this information is from the pipe man page, in section 7 ("**`man 7 pipe`**")
  - the **`pipe()`** system call discussed below is in section 2
- Provide a unidirectional IPC channel, with a read end and a write end
- Created with the **`pipe()`** system call:

```
#include <unistd.h>
int pipe(int fd[2]);
```

  - **`fd`** needs to be an array of 2 elements
  - return value is 0 on success, -1 on error
  - On success, a new pipe is created, **`fd[0]`** will be the file descriptor for the read end, and **`fd[1]`** will be the file descriptor for the write end

Pipes, cont.

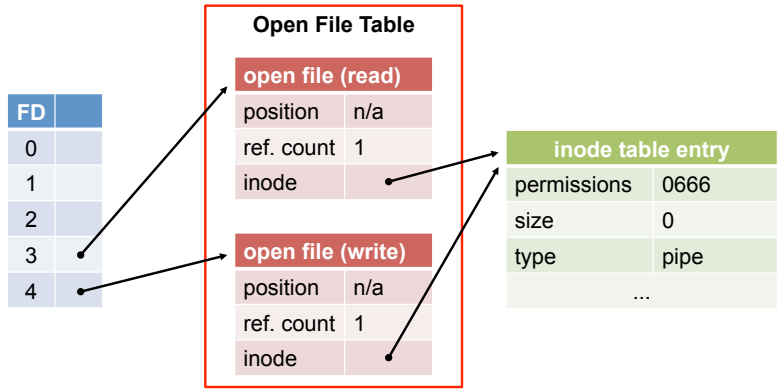
- An anonymous pipe's file descriptors are only visible in the process that created the pipe (and its children)
- I/O is blocking – read() from an empty pipe does not return until data is in the pipe, and write() to a full pipe does not return until sufficient space exists to complete the write
- Reads on the pipe will return EOF if no process has the write end open; writes to a pipe that has no readers cause an error
- Can't move the file position around (no seeking)

Pipes, cont.

- Generally, we create a pipe, then fork()
  - parent and child can then communicate via the pipe
- Because of the blocking I/O, we need to close ends of the pipe that we're not using, both immediately after the fork and once we're finished reading/writing
  - otherwise, read won't signal EOF appropriately or cause errors when we want to

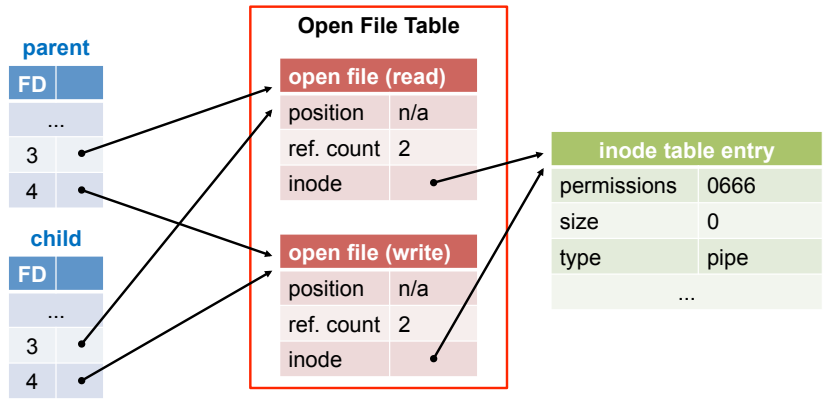
Opening a pipe

```
int fd[2];
pipe(fd);
```



After a fork

```
int fd[2];
pipe(fd);
fork();
```



An example with `pipe()`

```
/* #include statements and most error checking omitted */

int main() {
  int fd[2];
  char buf[BUFSIZ];  /* BUFSIZ is defined in stdio.h */

  if (pipe(fd) < 0)
    err(EX_OSERR, "pipe error");
  if (fork()) {  /* parent code */
    close(fd[1]);
    read(fd[0], buf, BUFSIZ);
    printf("Message from child: %s\n", buf);
  }
  else {  /* child code */
    char msg[] = "Hi there!";
    close(fd[0]);
    write(fd[1], msg, sizeof(msg));
  }
  return 0;
}
```


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`:

```
aaaaaaaaaabbbbbbbbbbccccccccccddddddddddeeeeeeeeeeffffffffffgggggggggghhhhhhhhhhiiiiiiiiiijjjjjjjjjjkkkkkkkkkklllllllllmmmmmmmmmmnnnnnnnnnnoooooooooo
```

Execution:

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

Sections 8.2-8.5, Bryant and O'Hallaron

More On Process Control

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`**!

Section 16.3, Reek

TIME MEASUREMENT

Time

- On a computer there are many ways to measure time
- Conceptual difference:
  - *wall time* is always running
  - *process time* is the time your program was running
    - process time doesn't count:
      - the time when your program was stopped for others
      - the time when your program stopped to wait for I/O
    - is composed of:
      - user time: when the OS is running your code
      - system time: when the OS is running system code (handling system calls)
- Difference in how to measure
  - interval time
  - clock cycles
- What time to use depends on what you are measuring

Timing a program

- To time an entire program in the shell:
  - **`time`** *program-name program-arguments*
  - runs the program and prints its execution time in the form (tcsh)

```
2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
```

    - the first two numbers are user and system time
    - the third number is wall time
    - the fourth is percentage of wall time: (user+system)/wall
    - the remainder are paging and I/O statistics
- This provides some idea about timing
  - but it's hard to know what to do about it - what functions are taking most of the time?

Representing time-of-day

- How to deal with
  - time zones
  - daylight saving time rules
  - computers moving between time zones
  - leap seconds?
- Answer:
  - UNIX keeps time internally as seconds and fractions of a second
    - 2^{32} bit values work nicely for 1ns accuracy for > 100 years
  - use a reference starting point
    - called the epoch - midnight Jan. 1, 1970
  - keep all time in UTC form until printed
    - no time zones or daylight saving time to deal with