15-213 Recitation 9: 11/04/02

Outline

- Error Handling
- I/O
- Linux man pages

Reminder

- Exam 2: Tuesday, Nov. 12

  Review session next Monday evening
- L6 Due: Friday, Nov. 19
  - This week's lectures
  - Text book: 10.9

Annie Luo

e-mail:

luluo@cs.cmu.edu

Office Hours:

Thursday 6:00 – 7:00

Wean 8402

Coverage of Exam 2

- Similar to last year's exam 2
  - Performance optimization
  - Cache simulation
  - Cache miss rate
  - Cache miss analysis
  - Processes
  - Signals
  - Virtual memory

- Come to the special recitation
  - Mon. 11/11 evening
  - Location TBD

Error Handling

- Should *always* check return code of system calls
  - Not only for 5 points in your lab!
  - There are subtle ways that things can go wrong
  - Use the status info kernel provides us

- Approach in this class: Wrappers

- Different error handling styles
  - Unix-Style
  - Posix-Style
  - DNS-Style

Unix-Style Error Handling

- Special return value when encounter error (always –1)
- Set global variable **errno** to an error code
  - Indicates the cause of the error
- Use **strerror** function for text description of **errno**

```
void unix_error(char *msg)
{
  fprintf(stderr, “%s: %s\n“,
             msg, strerror(errno));
  exit(0);
}
… …
if ((pid = wait(NULL)) < 0)
  unix_error(“Error in wait”);
```

POSIX-Style Error Handling

- Return value only indicate success (0) or failure (nonzero)
- Useful results returned in function arguments

```
void posix_error(int code, char *msg)
{
  fprintf(stderr, “%s: %s\n“,
            msg, strerror(code));
  exit(0);
}
… …
if ((retcode = pthread_create(…)) != 0)
  posix_error(retcode, “Error in pthread”);
```

DNS-Style Error Handling

- Return a NULL pointer on failure
- Set the global **h_errno** variable

```
void dns_error(char *msg)
{
  fprintf(stderr, “%s: DNS error %d\n“,
            msg, h_errno);
  exit(0);
}
… …
if ((p = gethostbyname(name)) == NULL)
  dns_error(“Error in gethostbyname”);
```

Example: Wrappers

```
void Kill (pid_t pid, int signum)
{
  int rc;
  if((rc = kill(pid, signum)) <0)
      unix_error("Kill error");
}
```

- Appendix B: csapp.c and csapp.h
- Unix-Style, for kill function
- Behaves exactly like the base function if no error
- Prints informative message and *terminates* the process

Not All Errors are Fatal

- Wrappers are not always the correct path
  - Treat all information as fatal errors
  - Terminate the program with **`exit()`**
  - Sometimes an error is not fatal

```
void sigchld_handler(int signum)
{
  pid_t pid;

  while((pid = waitpid(-1, NULL, 0)) > 0)
    printf("Reaped %d\n", (int)pid);

  if(errno != ECHILD)
    unix_error("waitpid error");
}
```

I/O

- Full coverage in Lecture 24, and Chapter 11 in textbook
- But folks were having some issues with the Shell Lab
  - fflush? printf? fprintf? scanf? fscanf?
- And these issues will pop up with the Malloc Lab

Unix I/O

- Why need I/O?
  - copy data between main memory and external devices

- All devices modeled as files in Unix
  - Disk drives, terminals, networks
  - A Unix file is a sequence of bytes
  - Input/Output performed by reading and writing files

Kernel I/O

- Why Kernel I/O
  - System level I/O functions provided by the kernel
  - Understand system concepts (process, VM, etc.)
  - When impossible/inappropriate to use standard library I/O
    - Info such as file size, creation time
    - Networking programming
- How kernel I/O works
  - Kernel maintains all file information
  - Apps only keep track of file descriptor returned from kernel

Functions - Open

- `int open(const char *pathname, int flags);`
  - Return value is a small integer – file descriptor

- Special file descriptors
  - Defined in `<unistd.h>`
  - Default open with each shell-created process
  - Standard Input (descriptor = 0)
  - Standard Output (descriptor = 1)
  - Standard Error (descriptor = 2)

Functions – Read and Write

- `ssize_t read(int fd, void *buf, size_t count);`
  - Copy count >0 bytes from a file (fd) to memory (buf)
  - From current position *k* (maintained by kernel)
  - Trigger EOF when *k* is greater than file size
  - No “EOF character”

- `ssize_t write(int fd, void *buf, size_t count);`
  - Copy count >0 bytes from memory (buf) to a file (fd)

Functions – Close

- `int close(int fd);`
  - Kernel frees data structures created by file open (if any)
  - Restores file descriptor to a pool of available descriptors
  - What if a process terminates?

  - Kernel closes all open files and free memory (for this proc)

Standard I/O

- Higher level I/O functions
  - **`fopen`**, **`fclose`**, **`fread`**, **`fwrite`**, **`fgets`**, **`fputs`**,
  - **`scanf`** and **`printf`**: formated I/O
  - *Eventually* calls the kernel I/O routines

- Models an open file as a stream:
  - A pointer to FILE
  - Abstraction for file descriptor and for a stream buffer

- Why use stream buffer?

  - Reduce expensive Unix system calls!

Example: buffered_io.c

```
#include <stdio.h>

int main(void)
{
  printf("1"); printf("5");
  printf("2"); printf("1");
  printf("3");
  return 0;
}
```

- How many system calls (kernel I/O routine) used?

Use `strace` to Check

- **`strace` *<program>***
  - Runs ***<program>*** and prints out info about all the system calls
- Let's run **`strace`** on **`buffered_io`**

```
unix> strace ./buffered_io
…
write(1, "15213", 515213) = 5
…
```

File Streams

- C Library equivalent of file descriptors

- More formatted I/O routines: **`fprintf`** and **`fscanf`**
  - Take an extra first argument **`FILE*`**
    - **`int printf(const char *format, ...);`**
    - **`int fprintf(FILE *stream, const char *format, ...);`**

  - **`printf(args) == fprintf(stdout, args)`**
  - **`scanf(args) == fscanf(stdin, args)`**

So Which Should I Use?

- Use standard I/O routines for disks and terminals
  - Full duplex: can input and output on the same stream
- Nasty restrictions when use stdio on sockets
  - Input cannot follow output (and vice versa) without **`fflush,`** **`fseek,`** **`fsetpos,`** or **`rewind`**
  - Flush the buffer for the first restriction
  - Use two streams for the second restriction
- So do not use standard I/O functions for network sockets

Flushing a File System

- `stderr` is not buffered

- `stdout` is line-buffered when it points to a terminal
  - Partial lines not appear until `fflush` or `exit`
  - Can produce unexpected results, esp. with debugging output

- There may also be input buffering when `stdin` points to a terminal

Flushing a File System (cont.)

- Forces a write of all buffered data
  - `int fflush(FILE *stream);`
  - `fflush(stdout);`

- Example: **`buffered_io_flush.c`**

```
#include <stdio.h>

int main(void)
{
  printf("1"); printf("5");
  fflush(stdout);
  printf("2"); printf("1");
  printf("3");
  return 0;
}
```

strace Revised

- Let's run **strace** on **buffered_io_flush**

```
unix> strace ./buffered_io_flush
…
write(1, "15", 215) = 2
write(1, “213", 3213) = 3

…
```

Unix Man pages

```
unix> man kill

KILL(1)              Linux Programmer's Manual              KILL(1)

NAME
       kill - terminate a process

SYNOPSIS
       kill [ -s signal | -p ]  [ -a ] pid ...
       kill -l [ signal ]

DESCRIPTION
       kill  sends the specified signal to the specified process.
       If no signal is specified, the TERM signal is  sent.   The
       TERM  signal  will  kill processes which do not catch this
       signal.  For other processes, if may be necessary  to  use
       the KILL (9) signal, since this signal cannot be caught.

       Most modern shells have a builtin kill function.
OPTIONS
       pid ...
              Specify the list of processes that kill should sig-
              nal.  Each pid can be one of four things.   A  pro-
              cess  name in which case processes called that will
```

Man Page Sections (cont.)

- Section 1: Commands
  - "Stuff that you could run from a Unix prompt"
  - **`cp(1)`, `bash(1)`, `kill(1)`**
- Section 2: System Calls
  - "Talking with the Kernel"
  - **`kill(2)`, `open(2)`**
- Section 3: Library Calls
  - "The C Library"
  - **`printf(3)`, `scanf(3)`, `fflush(3)`**

Man Page Sections

- To specify a man section:
  - **`man n …`**
  - **`man 2 kill`**

- http://www.linuxcentral.com/linux/man-pages/

Summary

- Error handling
  - You should always check error codes
  - Wrappers can help in most cases
- I/O:
  - Be sure to call **`fflush`** with debugging code
- Man pages:
  - Information grouped into sections