15213 Recitation Section C

Shimin Chen

Nov. 4, 2002

Outline

- Error handling
- I/O
- Man pages

Important Dates

- Lab 6 (Malloc)
  - Due Tuesday, November 19
  - Next Monday's recitation

- Exam 2
  - Tuesday, November 12
  - Review session next Monday evening

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

- Appendix B

Different Error Handling Styles

- Unix-Style
  - e.g. kill, signal, fork, etc.
- Posix-Style
  - e.g. pthread_create
- DNS-Style
  - e.g. gethostbyname

Unix-Style Error Handling

- Special return value when encounter error (always –1)
- Global variable **errno** set to an error code
- Use **strerror** function for text description of **errno**
  - Or use **perror**

```
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");
```

Unix-Style Error Handling Cont'd

```
if ((pid = wait(NULL)) < 0) {
  perror("Error in wait");
  exit (0);
}
```

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

Handle Errors Gracefully

- The wrappers shown above calls **`exit()`**

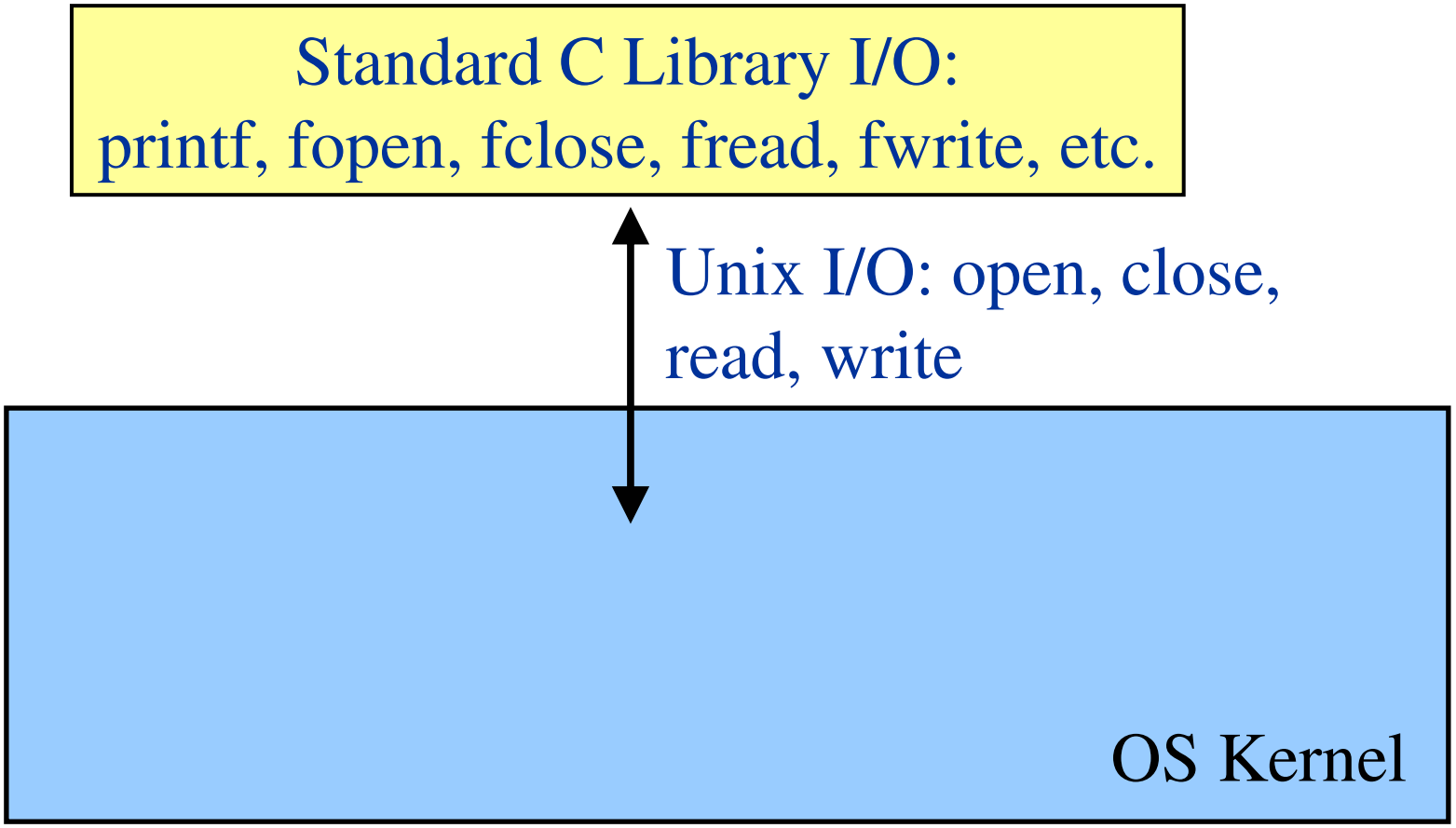
```
void sigchld_handler(int signum)
{
  pid_t pid;
  while((pid = waitpid(…)) > 0)
    printf("Reaped %d\n", (int)pid);
  if(errno != ECHILD)
    unix_error("waitpid error");
}
```

- In many situations, we want to handle errors more gracefully.
- For example: web server, etc.

I/O

- Full coverage in Lecture 24 on Next Thu. Chapter 11 in textbook.
- But people were having some issues with the Shell lab
- And these issues will pop up with the Malloc lab …

Unix I/O & Standard I/O



Unix I/O

- System calls
  - Reading and writing raw byte arrays
  - File descriptors (small integers)

- Functions
  - `int open(const char *pathname, int flags);`
  - `ssize_t read(int fd, void *buf, size_t count);`
  - `ssize_t write(int fd, const void *buf, size_t count);`
  - `int close(int fd);`

Special File Descriptors

- 0: standard in
- 1: standard out
- 2: standard error

Standard I/O

- The C library I/O routines
  - **`printf, scanf`**
  - Formatting
    - **`printf("%x\n", 0x15213);`**
  - Buffering
    - *Eventually* will call the Unix I/O routines (syscalls)
    - Buffers input to minimize the number of syscalls

Example: buffered_io.c

```
#include <stdio.h>

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

strace

- **`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
  - **`FILE*`**
  - **`stdin`**, **`stdout`**, **`stderr`**
  - **`FILE *fopen (const char *path, const char *mode);`**
- fprintf, fscanf, …
  - Take an extra first argument: **`FILE*`**
    - `int printf(const char *format, ...);`
    - `int fprintf(FILE *stream, const char *format, ...);`
  - **`printf(arg1,arg2,…) = fprintf(stdout,arg1,arg2,…)`**
  - **`scanf(arg1,arg2,…) = fscanf(stdin,arg1,arg2,…)`**

Flushing a File Stream

- Force the C library to write any buffered data using Unix I/O
- `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, revisited

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

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

Man Pages

- Contains detailed description of
  - Commands
  - System calls
  - C library functions
  - etc.

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.
```

Man page Sections

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

Sections

- To specify a man section
  - **`man` *`n`* …**
  - **`man 3 kill`**

- Man page for the command "**`man`**"
  - **`man man`**

Summary

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