CSE333 SECTION 5

Important Dates

- October 27th – Homework 2 Due
- October 29th – Midterm

String API vs. Byte API

- Recall: Strings are character arrays terminated by ‘\0’
- The String API (functions that start with str<…>) rely on the null terminating character
- The Byte API (functions that start with mem<…>) ask for a number of bytes to process

Examples

strcpy(src, dst)	memcpy(dst, src, bytes)
strcmp(str1, str2)	memcmp(str1, str2, bytes)
strchr(str, char)	memchr(data, char, bytes)

File I/O in C - Streams

- Reading and Writing using the notion of a stream
- Input can either be text or binary data
- Streams are either buffered (default) or unbuffered
- Standard Streams: stdin(fd 0), stdout(fd 1), stderr(fd 2)

Lib C File I/O

Utilizes FILE * for I/O.

#include <stdio.h>

File *f;

FILE *fopen(… char *filename, char *mode)

Modes:

- r - read only
- r+ - Read and Write
- And more! man fopen

Lib C File I/O

int fclose(FILE)

Returns 0 on success, otherwise EOF and set errno

size_t fread(data, size of chunks, number of chunks, FILE)

Returns the number of chunks read

size_t fwrite(data, size of chunks, number of chunks, FILE)

Returns the number of chunks written

fread_example.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define READBUFSIZE 128
int main(int argc, char **argv) {
  FILE *f;
  char readbuf[READBUFSIZE];
  size_t readlen;

  if (argc != 2) {
    fprintf(stderr, "usage: ./fread_example filename\n");
    return EXIT_FAILURE;  // defined in stdlib.h
  }

  // Open, read, and print the file
  f = fopen(argv[1], "rb");  // "rb" --> read, binary mode
  if (f == NULL) {
    fprintf(stderr, "%s -- ", argv[1]);
    perror("fopen failed -- ");
    return EXIT_FAILURE;
  }

  // Read from the file, write to stdout.
  while ((readlen = fread(readbuf, 1, READBUFSIZE, f)) > 0)
    fwrite(readbuf, 1, readlen, stdout);
  fclose(f);
  return EXIT_SUCCESS;  // defined in stdlib.h
}
```

printf(...) is equivalent to fprintf(stdout, ...)

stderr is a stream for printing error output to a console

fopen opens a stream to read or write a file

perror writes a string describing the last error to stderr

stdout is for printing non-error output to the console

Buffered I/O – Potential Problems?

- Data written using fwrite(…) is copied into a buffered allocated by stdio and written into memory when,
  - When fflush(…) is called
  - Buffer size is exceeded
  - For stdout, when a new line is reached (“line buffered”)
  - When fclose(…) is called
  - When your process exits gracefully

- Are there any potential problems?

Why is this a gotcha?

- What happens if...
  - your computer loses power before the buffer is flushed?
  - your program assumes data is written to a file, and it signals another program to read it?

- What are the performance implications?
  - data is copied into the stdio buffer
    - consumes CPU cycles and memory bandwidth
    - can potentially slow down high performance applications, like a web server or database ("zero copy")

What to do about it

- Turn off buffering with setbuf( )
  - this, too, may cause performance problems
  - e.g., if your program does many small fwrite( )'s, each of which will now trigger a system call into the Linux kernel
- Use a different set of system calls
  - POSIX provides open( ), read( ), write( ), close( ), and others
  - no buffering is done at the user level
- but...what about the layers below?
  - the OS caches disk reads and writes in the FS buffer cache
  - disk controllers have caches too!

stat

Returns the information about a specific file

- int stat(const char *path, struct stat *buf);
- int fstat(int fd, struct stat *buf);

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

POSIX I/O

- What's the difference?
  - Unbuffered at the user level
  - Less convenient
- When would I use it? Networking
- How do I use it?
  - #include <fcntl.h>
  - #include <unistd.h>
  - #include <sys/types.h>
  - #include <sys/uio.h>
  - man 2 <open, close, read, write>
- POSIX I/O uses file descriptors instead of FILE
  - Essentially an int representing the file

open / close

To open a file...

- pass in the filename and access mode, similar to fopen
- get back a “file descriptor”
  - similar to a (FILE *) from fopen, but is just an int

```
#include <fcntl.h>

...

 int fd = open("foo.txt",
               O_RDONLY);
 if (fd == -1) {
   perror("open failed");
   exit(EXIT_FAILURE);
 }

...

 close(fd);
```

Reading from a file

```
ssize_t read(int fd, void *buf, size_t count);
```

- returns the # of bytes read
  - might be fewer bytes than you requested (!!!)
  - returns 0 if you're at end-of-file
  - return -1 on error
- warning: read has some very surprising error modes!

read( ) error modes

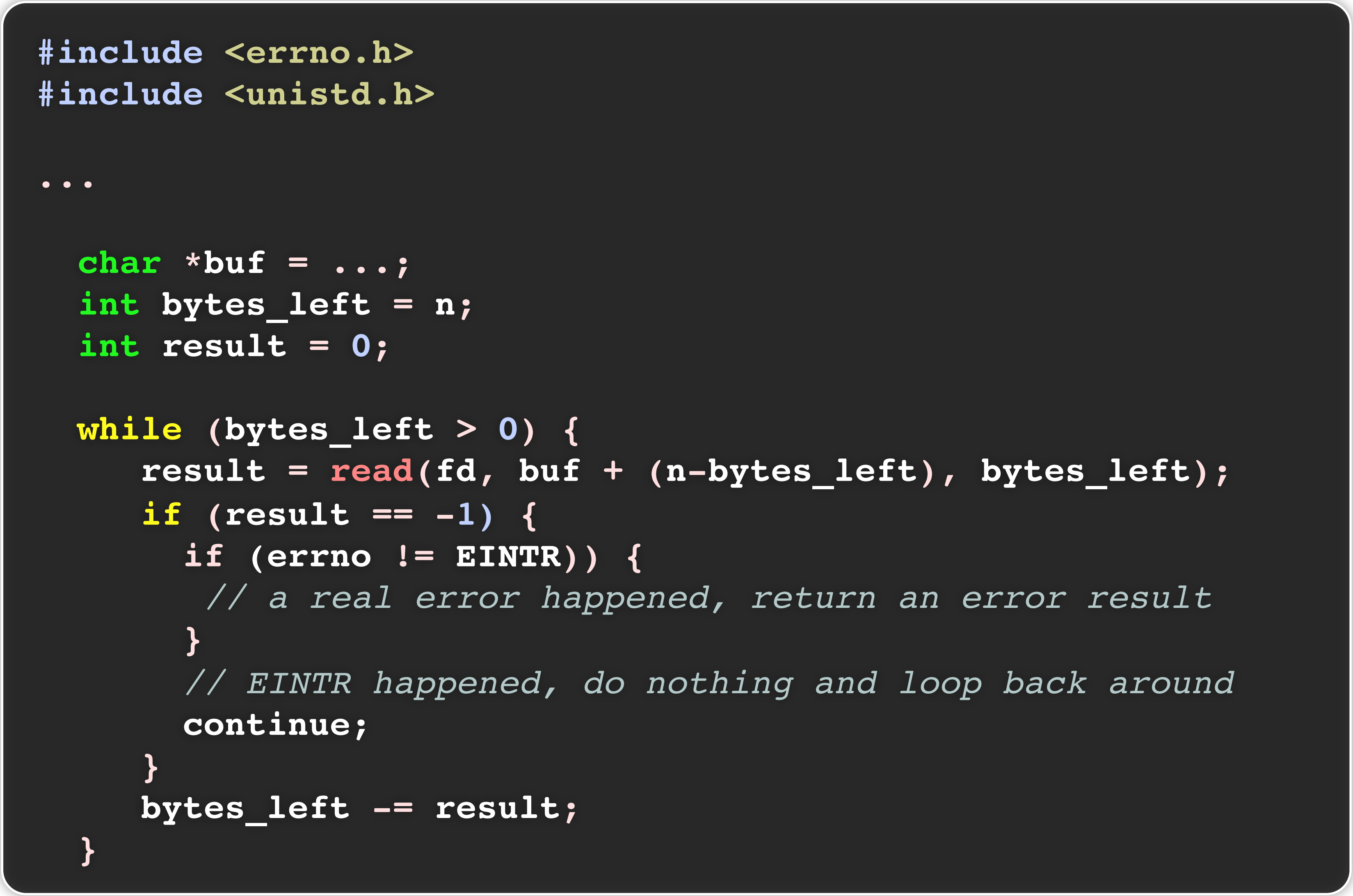
On error, the “errno” global variable is set

- you need to check it to see what kind of error happened

What errors might read( ) encounter?

- EBADF -- bad file descriptor
- EFAULT -- output buffer is not a valid address
- EINTR -- read was interrupted, please try again
- and many others

How to read( ) n bytes



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

...

  char *buf = ...;
  int bytes_left = n;
  int result = 0;

  while (bytes_left > 0) {
     result = read(fd, buf + (n-bytes_left), bytes_left);
     if (result == -1) {
       if (errno != EINTR)) {
        // a real error happened, return an error result
       }
       // EINTR happened, do nothing and loop back around
       continue;
     }
     bytes_left -= result;
  }
```

Other low-level functions

Read the man pages to learn about

- write( ) -- write data
- fsync( ) -- flush data to the underlying device
- opendir( ), readdir( ), closedir( ) -- get a directory listing