Guessing Game

- C provides some basic facilities
- C libraries help make those primitive facilities useful

For each library routine prototype that follows, guess how to use it:
  - What are the arguments? What is the return value?
  - How would an error be indicated?
  - Where was memory created? How will it be freed?

Copy Memory

```c
void *memcpy(void *dest, const void *src, size_t n);
```
SYNOPSIS
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
The memcpy() function copies n bytes from memory area src to memory area dest. The memory areas must not overlap. Use memmove(3) if the memory areas do overlap.

RETURN VALUE
The memcpy() function returns a pointer to dest.

Copy A String

char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t n);
SYNOPSIS
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
The strcpy() function copies the string pointed to by src, including the terminating null byte ('\0'), to the buffer pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy.

The strncpy() function is similar, except that at most n bytes of src are copied.
Warning: If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

If the length of src is less than n, strncpy() pads the remainder of dest with null bytes.

Concatenate Two Strings
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
These are slightly modified versions of slides prepared by Steve Gribble

SYNOPSIS
#include <string.h>
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);

DESCRIPTION
The strcat() function appends the src string to the dest string, overwriting the terminating null byte ('\0') at the end of dest, and then adds a terminating null byte. The strings may not overlap, and the dest string must have enough space for the result.

The strncat() function is similar, except that
  * it will use at most n characters from src; and
  * src does not need to be null-terminated if it contains n or more characters.
As with strcat(), the resulting string in dest is always null-terminated.

These are slightly modified versions of slides prepared by Steve Gribble

Duplicate a String

char *strdup(const char *s);
char *strndup(const char *s, size_t n);
SYNOPSIS
#include <string.h>
char *strdup(const char *s);
char *strndup(const char *s, size_t n);

DESCRIPTION
The strdup() function returns a pointer to a new string which is a duplicate of the string s. Memory for the new string is obtained with malloc(3), and can be freed with free(3).

The strndup() function is similar, but only copies at most n characters. If s is longer than n, only n characters are copied, and a terminating null byte (\0) is added.

RETURN VALUE
The strdup() function returns a pointer to the duplicated string, or NULL if insufficient memory was available.

ASCII String to Double

double strtod(const char *nptr, char **endptr);
RETURN VALUE

These functions return the converted value, if any. If endptr is not NULL, a pointer to the character after the last character used in the conversion is stored in the location referenced by endptr.

If no conversion is performed, zero is returned and the value of nptr is stored in the location referenced by endptr.

If the correct value would cause overflow, plus or minus HUGE_VAL (HUGE_VALF, HUGE_VALL) is returned (according to the sign of the value), and ERANGE is stored in errno. If the correct value would cause underflow, zero is returned and ERANGE is stored in errno.

ssize_t getline(char **lineptr, size_t *n, FILE *stream);

Get a Line of Text
DESCRIPTION
getline() reads an entire line from stream, storing the address of the buffer containing the text into *lineptr. The buffer is null-terminated and includes the newline character, if one was found.

If *lineptr is NULL, then getline() will allocate a buffer for storing the line, which should be freed by the user program. (In this case, the value in *n is ignored.)

Alternatively, before calling getline(), *lineptr can contain a pointer to a malloc(3)-allocated buffer *n bytes in size. If the buffer is not large enough to hold the line, getline() resizes it with realloc(3), updating *lineptr and *n as necessary.

In either case, on a successful call, *lineptr and *n will be updated to reflect the buffer address and allocated size respectively.

RETURN VALUE
On success, getline() and getdelim() return the number of characters read, including the delimiter character, but not including the terminating null byte. This value can be used to handle embedded null bytes in the line read.

Both functions return -1 on failure to read a line (including end-of-file condition).
Read a Line of Text
(from the terminal)
(with history)

char * readline (const char *prompt);

DESCRIPTION
readline will read a line from the terminal and return it, using prompt as a prompt. If prompt is NULL or the empty string, no prompt is issued. The line returned is allocated with malloc(3); the caller must free it when finished. The line returned has the final newline removed, so only the text of the line remains.

readline offers editing capabilities while the user is entering the line. By default, the line editing commands are similar to those of emacs. A vi-style line editing interface is also available.

This manual page describes only the most basic use of readline. Much more functionality is available; see The GNU Readline Library and The GNU History Library for additional information.

RETURN VALUE
readline returns the text of the line read. A blank line returns the empty string. If EOF is encountered while reading a line, and the line is empty, NULL is returned. If an EOF is read with a non-empty line, it is treated as a newline.
Read/Write a File

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

DESCRIPTION

The function fread() reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.

The function fwrite() writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

RETURN VALUE

fread() and fwrite() return the number of items successfully read or written (i.e., not the number of characters). If an error occurs, or the end-of-file is reached, the return value is a short item count (or zero).

fread() does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.
Read/Write a File

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

RETURN VALUE (read)
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

RETURN VALUE (write)
On success, the number of bytes written is returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately.
Using C streams

```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, "fopen failed -- ");
        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
}
```

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

Fork exec

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid;
    int status;
    fprintf("parent : \dn", getpid());
    pid = fork();
    if (pid == 0) {
        char* argv0 = "/bin/bash", 0;
        print("child : \dn", getpid());
        execv(argv0, argv);
        perror("exec failed"); // should never get here
    }
    /* in parent */
    waitpid(pid, &status, 0);
    printf("Child exited with status \dn", status);
    printf("Parent exiting\n");
}
```

two identical processes at this point, except that child has pid==0
overwrite my virtual address space with /bin/bash and start it up
CSE 333
Lecture 6 - system calls, intro to file I/O

Remember this picture?

brief diversion

OS / app interface (system calls)

HW/SW interface (x86 + devices)

C application
C++ application
Java application
C standard library (glibc)
C++ STL / boost / standard library
JRE

-----------------------------------------------

operating system

-----------------------------------------------

hardware

CPU memory storage network
GPU clock audio radio peripherals

These are slightly modified versions of slides prepared by Steve Grable
What’s an OS?

Software that:

1. directly interacts with the hardware
   • OS is trusted to do so; user-level programs are not
   • OS must be ported to new HW; user-level programs “are portable”

2. manages (allocates, schedules, protects) hardware resources
   • decides which programs can access which files, memory locations, pixels on the screen, etc., and when

3. abstracts away messy hardware devices
   • provides high-level, convenient, portable abstractions
     • e.g., files vs. disk blocks

OS as an abstraction provider

The OS is the “layer below”

- a module that your program can call (via system calls)
- provides a powerful API (the OS API)
**OS as a protection system**

OS isolates processes from each other
- but permits controlled sharing between them
  - through shared name spaces (e.g., FS names)

OS isolates itself from processes
- and therefore must prevent processes from accessing the hardware directly

OS is allowed to access the hardware
- user-level processes run with the CPU in unprivileged mode
- when the OS is running, the CPU is set to privileged mode
- user-level processes invoke a system call to safely enter the OS

---

A CPU (thread of execution) is running user-level code in process A; that CPU is set to unprivileged mode.

---

These are slightly modified versions of slides prepared by Steve Gribble.
code in process A invokes a system call; the hardware then sets the CPU to privileged mode and traps into the OS, which invokes the appropriate system call handler.

because the CPU executing the thread that’s in the OS is in privileged mode, it is able to use privileged instructions that interact directly with hardware devices like disks.
These are slightly modified versions of slides prepared by Steve Gribble

OS as a protection system

Once the OS has finished servicing the system call (which might involve long waits as it interacts with HW) it:

(a) sets the CPU back to unprivileged mode, and
(b) returns out of the system call back to the user-level code in process A

The process continues executing whatever code that is next after the system call invocation.
Details on x86 / Linux

A more accurate picture:
- consider a typical Linux process
- its thread of execution can be several places
  - in your program’s code
  - in glibc, a shared library containing the C standard library, POSIX support, and more
  - in the Linux architecture-independent code
  - in Linux x86-32/x86-64 code

Some routines your program invokes may be entirely handled by glibc
- without involving the kernel
  - e.g., strcmp( ) from stdio.h
- ∃ some initial overhead when invoking functions in dynamically linked libraries
- but, after symbols are resolved, invoking glibc routines is nearly as fast as a function call within your program itself
Details on x86 / Linux

Some routines may be handled by glibc, but they in turn invoke Linux system calls

- e.g., POSIX wrappers around Linux syscalls
  - POSIX readdir() invokes the underlying Linux readdir()
- e.g., C stdio functions that read and write from files
  - fopen(), fclose(), fprintf() invoke underlying Linux open(), read(), write(), close(), etc.

Your program can choose to directly invoke Linux system calls as well

- nothing forces you to link with glibc and use it
- but, relying on directly invoked Linux system calls may make your program less portable across UNIX varieties
Details on x86 / Linux

Let’s walk through how a Linux system call actually works

- we’ll assume 32-bit x86 using the modern SYSENTER / SYSEXIT x86 instructions

Details on x86 / Linux

Remember our process address space picture

- let’s add some details
Details on x86 / Linux

0xFFFFFFFF

linux-gate.so

Linux kernel

kernel stack

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

SP → stack

shared libraries

heap (malloc/free)

read/write segment

.data, .bss

read-only segment

.text, .rodata

IP →

0x00000000

0xFFFFFFFF

linux-gate.so

Linux kernel

kernel stack

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

SP → stack

shared libraries

heap (malloc/free)

read/write segment

.data, .bss

read-only segment

.text, .rodata

IP →

0x00000000

Details on x86 / Linux

0xFFFFFFFF

linux-gate.so

Linux kernel

kernel stack

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

SP → stack

shared libraries

heap (malloc/free)

read/write segment

.data, .bss

read-only segment

.text, .rodata

IP →

0x00000000

Details on x86 / Linux

0xFFFFFFFF

linux-gate.so

Linux kernel

kernel stack

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

SP → stack

shared libraries

heap (malloc/free)

read/write segment

.data, .bss

read-only segment

.text, .rodata

IP →

0x00000000

process is executing your program code

process calls into a glibc function (e.g., fopen)

‣ we’ll ignore the messy details of loading / linking shared libraries

These are slightly modified versions of slides prepared by Steve Gribble
Details on x86 / Linux

glibc begins the process of invoking a Linux system call
• glibc’s fopen( ) likely invokes Linux’s open( ) system call
• puts the system call # and arguments into registers
• uses the call x86 instruction to call into the routine __kernel_vsyscall located in linux-gate.so

Details on x86 / Linux

linux-gate.so is a vdso
• a virtual dynamically linked shared object
• is a kernel-provided shared library, i.e., is not associated with a .so file, but rather is conjured up by the kernel and plunked into a process’ s address space
• provides the intricate machine code needed to trigger a system call
Details on x86 / Linux

linux-gate.so eventually invokes the SYSENTER x86 instruction

- SYSENTER is x86’s “fast system call” instruction
- it has several side-effects
  - causes the CPU to raise its privilege level
  - traps into the Linux kernel by changing the SP, IP to a previously determined location
  - changes some segmentation related registers (see cse451)

The kernel begins executing code at the SYSENTER entry point

- is in the architecture-dependent part of Linux
- it’s job is to:
  - look up the system call number in a system call dispatch table
  - call into the address stored in that table entry; this is Linux’s system call handler
  - for open, the handler is named sys_open, and is system call #5

These are slightly modified versions of slides prepared by Steve Gribble
Details on x86 / Linux

The system call handler executes

- what it does is system-call specific, of course
- it may take a long time to execute, especially if it has to interact with hardware
- Linux may choose to context switch the CPU to a different runnable process

Eventually, the system call handler finishes

- returns back to the system call entry point
- places the system call’s return value in the appropriate register
- calls SYSEXIT to return to the user-level code
Details on x86 / Linux

SYSEXIT transitions the processor back to user-mode code
  • has several side-effects
  • restores the IP, SP to user-land values
  • sets the CPU back to unprivileged mode
  • changes some segmentation related registers (see cse451)
  • returns execution back to glibc

Details on x86 / Linux

glibc continues to execute
  • might execute more system calls
  • eventually returns back to your program code
If you’re curious

Download the Linux kernel source code
- get version 2.6.34.8
- available from http://www.kernel.org/

Take a look at:
- arch/x86/kernel/syscall_table_32.S [system call table]
- arch/x86/kernel/entry_32.S [SYSENTER entry point and more]
- arch/x86/vdso/vdso32/sysenter.S [user-land vdso]
And: http://articles.manugarg.com/systemcallinlinux2_6.html

Also...

man, section 2: Linux system calls
- man 2 intro
- man 2 syscalls (or look online here)

man, section 3: glibc / libc library functions
- man 3 intro (or look online here)
**strace**

A useful Linux utility that shows the sequence of system calls that a process makes:

```bash
bash$ strace ls 2>&1 | less
```

... etc.

These are slightly modified versions of slides prepared by Steve Gribble
Let’s do some file I/O...

We’ll start by using C’s standard library
- these functions are implemented in glibc on Linux
- they are implemented using Linux system calls

C’s stdio defines the notion of a stream
- a stream is a way of reading or writing a sequence of characters from/to a device
  - a stream can be either text or binary; Linux does not distinguish
  - a stream is buffered by default; libc reads ahead of you, writes behind
  - three streams are provided by default: stdin, stdout, stderr
- you can open additional streams to read/write to files

```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, "fopen failed -- ");
        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
}
```

Using C streams

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

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

versions of slides prepared by Steve Gribble
Writing is easy too

see cp_example.c

A gotcha

By default, stdio turns on buffering for streams

- data written by fwrite( ) is copied into a buffer allocated by stdio inside your process’s address space

- at some point, the buffer will be drained into the destination
  - when you call fflush( ) on the stream
  - when the buffer size is exceeded (often 1024 or 4096 bytes)
  - for stdout to a console, when a newline is written (“line buffered”)
  - when you call fclose( ) on the stream
  - when your process exits gracefully (exit( ) or return from main( ))
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!
Exercise 1

Write a program that:
- uses argc/argv to receive the name of a text file
- reads the contents of the file a line at a time
- parses each line, converting text into a uint32_t
- builds an array of the parsed uint32_t’s
- sorts the array
- prints the sorted array to stdout

- hints: use “man” to read about getline, sscanf, realloc, and qsort

```
bash$ cat in.txt
1213
3231
000005
52
bash$ ex1 in.txt
5
52
1213
3231
bash$
```

Exercise 2

Write a program that:
- loops forever; in each loop, it:
  - prompts the user to input a filename
  - reads from stdin to receive a
    filename
  - opens and reads the file, and prints
    its contents to stdout, in the format
    shown on the right
- hints:
  - use “man” to read about fgets
  - or if you’re more courageous, try
    “man 3 readline” to learn about
    libreadline.a, and google to learn
    how to link to it
See you on Wednesday!