POSIX I/O, System Calls
CSE 333 Winter 2020

Instructor: Justin Hsia

Teaching Assistants:
Andrew Hu               Austin Chan               Brennan Stein
Cheng Ni                Cosmo Wang               Diya Joy
Guramrit Singh          Mengqi Chen               Pat Kosakanchit
Rehaan Bhimar          Renshu Gu               Travis McGaha
Zachary Keyes
Administrivia

- Exercise 7 posted *tomorrow*, due Monday (1/27)
  - Comment your code, check linter and valgrind

- Homework 1 due tomorrow night (1/23)
  - Watch that `HashTable` doesn’t violate the modularity of `LinkedList`
  - Watch for pointer to local (stack) variables
  - Use a debugger (*e.g.* `gdb`) if you’re getting segfaults
  - Clean up “to do” comments, but leave “STEP #” markers
  - Late days: don’t tag `hw1-final` until you are really ready

- Homework 2 will be released on Friday (1/24)
Lecture Outline

- POSIX Lower-Level I/O
- System Calls
Remember This Picture?

A brief diversion...

OS / app interface (system calls)

HW/SW interface (x86 + devices)

C application
C standard library (glibc)

C++ application
C++ STL/boost/standard library

Java application
JRE

operating system

hardware

CPU memory storage network
GPU clock audio radio peripherals
We Need To Go Deeper...

- So far we’ve seen the C standard library to access files
  - Use a provided `FILE*` stream abstraction
  - `fopen()`, `fread()`, `fwrite()`, `fclose()`, `fseek()`

- These are convenient and portable
  - They are buffered*
  - They are implemented using lower-level OS calls
From C to POSIX

- Most UNIX-en support a common set of lower-level file access APIs: **POSIX** – Portable Operating System Interface
  - `open()`, `read()`, `write()`, `close()`, `lseek()`
    - Similar in spirit to their `f*()` counterparts from the C std lib
    - Lower-level and unbuffered compared to their counterparts
    - Also less convenient

🌟 You will have to use these to read file system directories and for network I/O, so we might as well learn them now
open() / close()

⚠️ To open a file:

- Pass in the filename and access mode
  - Similar to fopen()
- Get back a “file descriptor”
  - Similar to FILE* from fopen(), but is just an int
  - Defaults: 0 is stdin, 1 is stdout, 2 is stderr

```c
#include <fcntl.h>     // for open()
#include <unistd.h>    // for close()
...
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 number of bytes read
  - Might be fewer bytes than you requested (!!!)
  - Returns 0 if you’re already at the end-of-file
  - Returns `-1` on error (and sets `errno`)

- There are some surprising error modes (check `errno`)
  - `EBADF`: bad file descriptor
  - `EFAULT`: output buffer is not a valid address
  - `EINTR`: read was interrupted, please try again (ARGH!!!! 😤😠)
  - And many others...
One way to read() n bytes

Which is the correct completion of the blank below?

Vote at http://PollEv.com/justinh

char* buf = ...; // buffer of size n
int bytes_left = n;
int result; // result of read()

while (bytes_left > 0) {
    result = read(fd, ____, bytes_left);
    if (result == -1) {
        if (errno != EINTR) {
            // a real error happened,
            // so return an error result
        }
        // EINTR happened,
        // so do nothing and try again
        continue;
    }
    bytes_left -= result;
}
One method to \texttt{read()} $n$ bytes

```c
int fd = \texttt{open}(filename, O_RDONLY);
char* buf = ...; // buffer of appropriate size
int bytes_left = n;
int result;

while (bytes_left > 0) {
    result = \texttt{read}(fd, buf + (n - bytes_left), bytes_left);
    if (result == -1) {
        if (errno != EINTR) {
            // a real error happened, so return an error result
        }
        // EINTR happened, so do nothing and try again
        continue;
    } else if (result == 0) {
        // EOF reached, so stop reading
        break;
    }
    bytes_left -= result;
}
\texttt{close}(fd);
```
Other Low-Level Functions

- Read man pages to learn about:
  - `write()` – write data
    - `#include <unistd.h>`
  - `fsync()` – flush data to the underlying device
    - `#include <unistd.h>`
  - `opendir()`, `readdir()`, `closedir()` – deal with directory listings
    - Make sure you read the section 3 version (`e.g. man 3 opendir`)
    - `#include <dirent.h>`

- A useful shortcut sheet (from CMU):
  [Link](http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf)
C Standard Library vs. POSIX

- C std lib implements a subset of POSIX
  - e.g. POSIX provides directory manipulation that C std lib doesn’t
- C std lib implements automatic buffering
- C std lib has a nicer API

- The two are similar but C std lib builds on top of POSIX
  - Choice between high-level and low-level
  - Will depend on the requirements of your application
Lecture Outline

- POSIX Lower-Level I/O
- System Calls
What’s an OS?

Software that:

- Directly interacts with the hardware
  - OS is trusted to do so; user-level programs are not
  - OS must be ported to new hardware; user-level programs are portable

- Manages (allocates, schedules, protects) hardware resources
  - Decides which programs can access which files, memory locations, pixels on the screen, etc. and when

- Abstracts away messy hardware devices
  - Provides high-level, convenient, portable abstractions (e.g. files, disk blocks)
OS: Abstraction Provider

- The OS is the “layer below”
  - A module that your program can call (with system calls)
  - Provides a powerful OS API – POSIX, Windows, etc.

File System
  - open(), read(), write(), close(), ...

Network Stack
  - connect(), listen(), read(), write(), ...

Virtual Memory
  - brk(), shm_open(), ...

Process Management
  - fork(), wait(), nice(), ...
OS: Protection System

- OS isolates process from each other
  - But permits controlled sharing between them
    - Through shared name spaces (e.g. file names)

- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly

- OS is allowed to access the hardware
  - User-level processes run with the CPU (processor) in unprivileged mode
  - The OS runs with the CPU in privileged mode
  - User-level processes invoke system calls to safely enter the OS
A CPU (thread of execution) is running user-level code in Process A; the CPU is set to unprivileged mode.
System Call Trace (high-level view)

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.
System Call Trace (high-level view)

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.
System Call Trace (high-level view)

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

1. Sets the CPU back to unprivileged mode and
2. Returns out of the system call back to the user-level code in Process A.
System Call Trace (high-level view)

The process continues executing whatever code is next after the system call invocation.

Useful reference: CSPP '8.1–8.3 (the 351 book)
“Library calls” on x86/Linux

- A more accurate picture:
  - Consider a typical Linux process
  - Its thread of execution can be in one of 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 x86-64 code
“Library calls” on x86/Linux: Option 1

- Some routines your program invokes may be entirely handled by glibc without involving the kernel
  - *e.g. `strcmp()` from `stdio.h`*

- There is some initial overhead when invoking functions in dynamically linked libraries (during loading)
  - But after symbols are resolved, invoking glibc routines is basically as fast as a function call within your program itself!
“Library calls” on x86/Linux: Option 2

- 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()`, `close()`, `write()`, etc.
“Library calls” on x86/Linux: Option 3

- Your program can choose to directly invoke Linux system calls as well
  - Nothing is forcing 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
    - x86-64 code is similar, though details always change over time, so take this as an example – not a debugging guide
System Calls on x86/Linux

Remember our process address space picture?

- Let’s add some details:
System Calls on x86/Linux

Process is executing your program code

- **0xFFFFFFF**: Linux-gate.so
- **kernel stack**
- **Stack**
- **Shared Libraries**
- **Heap (malloc/free)**
- **Read/Write Segment**
  - `.data, .bss`
- **Read-Only Segment**
  - `.text, .rodata`

**CPU**
- **unpriv**

**Linux kernel**
- **architecture-independent code**
- **architecture-dependent code**

**glibc**
- **C standard library**
- **POSIX**

**Your program**
- **process**: Stack
- **program code**: IP
- **kernel stack**: SP
System Calls on x86/Linux

Process calls into a \texttt{glibc} function

- \texttt{e.g. fopen()} 
- We’ll ignore the messy details of loading/linking shared libraries

\begin{itemize}
  \item \texttt{glibc}
  \item Linux kernel
  \item Stack
  \item Shared Libraries
  \item Heap (malloc/free)
  \item Read/Write Segment \texttt{.data, .bss}
  \item Read-Only Segment \texttt{.text, .rodata}
  \item \texttt{0x00000000}
  \item Linux kernel
  \item \texttt{CPU}
  \item \texttt{unpriv}
  \item \texttt{IP}
  \item \texttt{SP}
\end{itemize}
**System Calls 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`

![Diagram of system calls and stack memory]

- **Your program**
  - C standard library
  - POSIX
- **glibc**
  - architecture-independent code
- **Linux kernel**
  - architecture-dependent code
- **Unpriv**
  - CPU
System Calls on x86/Linux

`linux-gate.so` is a vdso

- A virtual dynamically-linked shared object
- Is a kernel-provided shared library that is plunked into a process’ address space
- Provides the intricate machine code needed to trigger a system call (details not important)
System Calls on x86/Linux

`linux-gate.so` eventually invokes the `SYSENTER` x86 instruction

- `SYSENTER` is x86’s “fast system call” instruction
  - 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)

![Diagram showing system calls and kernel stack]
System Calls on x86/Linux

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
System Calls on x86/Linux

The system call handler executes:

- What it does is system-call specific
- 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
System Calls on x86/Linux

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

![Diagram showing system call handling process]

- Stack
- Shared Libraries
- Heap (malloc/free)
- Read/Write Segment `.data, .bss`
- Read-Only Segment `.text, .rodata`
- `0x00000000`
- `0xFFFFFFFF`

- Linux kernel
  - `linux-gate.so`
  - `kernel stack`

- `SP`, `IP`

- `C standard library`
- `POSIX`
- `glibc`

- `architecture-independent code`
- `architecture-dependent code`

- `CPU`
- `priv`
System Calls on x86/Linux

**SYSEXIT** transitions the processor back to user-mode code

- Restores the IP, SP to user-land values
- Sets the CPU back to unprivileged mode
- Changes some segmentation-related registers (see CSE451)
- Returns the processor back to glibc

Architecture:
- **Stack**
- **Shared Libraries**
- **Heap (malloc/free)**
- **Read/Write Segment .data, .bss**
- **Read-Only Segment .text, .rodata**

Kernel:
- `0x00000000`
- `0xFFFFFFFF`

C standard library:
- glibc

POSIX:
- `Linux kernel`

Your program:
- `C standard library`

Unprivileged code:
- `unpriv`

CPU:
- `unpriv`
System Calls on x86/Linux

glibc continues to execute

- Might execute more system calls
- Eventually returns back to your program code

Your program

architecture-independent code

architecture-dependent code

C standard library

POSIX

glibc

Linux kernel

unpriv

CPU

Stack

Shared Libraries

Heap (malloc/free)

Read/Write Segment .data, .bss

Read-Only Segment .text, .rodata

Linux kernel

kernel stack

0xFFFFF000

0x00000000

0xFFFFFFFF

linux-gate.so
strace

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

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

```
evre("/usr/bin/ls", ["ls"], /* 41 vars */) = 0
brk(NULL) = 0x15aa000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f03bb741000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=126570, ...}) = 0
mmap(NULL, 126570, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f03bb722000
close(3) = 0
open("/lib64/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0\0\0\0\0\0\0\0\0\0\0\0\0\0\1\0\0\0\0\0\...", 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=155744, ...}) = 0
mmap(NULL, 2255216, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0x23000) = 0x7f03bb51d000
mprotect(0x7f03bb31e000, 2093056, PROT_NONE) = 0
mprotect(0x7f03bb31e000, 2093056, PROT_NONE) = 0
mmap(0x7f03bb31e000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x23000) = 0x7f03bb51d000
mprotect(0x7f03bb31e000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x23000) = 0x7f03bb51d000
... etc ...
```
If You’re Curious

- Download the Linux kernel source code
  - `man`, section 1: `user commands`

- `man`, section 2: Linux system calls
  - `man 2 intro`
  - `man 2 syscalls`

- `man`, section 3: glibc/libc library functions
  - `man 3 intro`

- *The book:* *The Linux Programming Interface* by Michael Kerrisk (keeper of the Linux man pages)