CSE 333
Lecture 6 - system calls, intro to file I/O

Hal Perkins
Department of Computer Science & Engineering
University of Washington
New exercise posted this morning, due before class Fri.

Lectures and sections this week: I/O and system calls

- Key material for next part of the project (& interesting by itself!)

HW1 due Tuesday(!!)

- How doth it go?

- Yes, you can use late days (up to 2)
  - But you really, really don’t want to...

- Suggestion from the graders: clean up the “to do” comments, but leave the “step 1”, “step 2” markers so they can find things quickly
Administrivia - Code Quality

Code quality ("style") **really** matters - and not just in classes

Rule #0: reader’s time is **much** more important than writer’s
  - Good comments are essential, clarity/understandability is critical

Rule #1: match existing code

Rule #2: use tools. examples:
  - Compiler warnings: just fix them!
  - Clint style warnings: fix most of them; be sure you understand anything you don’t fix and can justify it (ok to have a type as malloc parameter, not ok to have spaces instead of tabs or magic numbers instead of #define, etc. ...)
  - Valgrind warnings: fix all of them unless you know why it’s not an error (example: printing uninitialized bytes in a debugging tool)
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
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 (with system calls)
- provides a powerful API (the OS API - POSIX, Windows, ...)

<table>
<thead>
<tr>
<th>OS API</th>
<th>file system</th>
<th>network stack</th>
<th>virtual memory</th>
<th>process mgmt</th>
<th>etc. . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>file system</td>
<td>network stack</td>
<td>virtual memory</td>
<td>process mgmt</td>
<td>etc.</td>
</tr>
</tbody>
</table>

- `open()`, `read()`, `write()`, `close()`, ...
- `connect()`, `listen()`, `read()`, `write()`, ...
- `brk()`, `shm_open()`, ...
- `fork()`, `wait()`, `nice()`, ...
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
OS as a protection system

a CPU (thread of execution) is running user-level code in process A; that CPU is set to unprivileged mode
OS as a protection system

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.
OS as a protection system

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
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
OS as a protection system

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
Details on x86 / Linux

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.
Details on x86 / Linux

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

process is executing your program code

IP →

SP ←

0xFFFFF000

linux-gate.so

Linux kernel

kernel stack

stack

shared libraries

heap (malloc/free)

read/write segment

.data, .bss

read-only segment

.text, .rodata

0x00000000

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

unpriv

CPU
Details on x86 / Linux

- process calls into a glibc function (e.g., fopen)
  - we'll ignore the messy details of loading / linking shared libraries

0x00000000

0xFFFFFFFF

linux-gate.so

Linux kernel

kernel stack

stack

shared libraries

heap (malloc/free)

read/write segment

.data, .bss

read-only segment

.text, .rodata

0x00000000

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

Linux kernel

unpriv

CPU

IP

SP

CPU

unpriv

CSE333 lec 6 syscall fio // 01-21-14 // Perkins
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

- 0xFFFFFFFF

- your program
  - C standard library
  - POSIX

- glibc

- unpriv
- CPU

- Linux kernel

- architecture-independent code
- architecture-dependent code

- 0x00000000

- IP
- SP
### 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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0x00000000</td>
<td>Architecture-dependent code</td>
</tr>
<tr>
<td>$0xFFFFFFF</td>
<td>Linux kernel</td>
</tr>
<tr>
<td>$0xFFFFFFFF</td>
<td>Linux-gate.so</td>
</tr>
<tr>
<td>$0xFFF</td>
<td>Stack</td>
</tr>
<tr>
<td>$0x0</td>
<td>Data, BSS</td>
</tr>
<tr>
<td></td>
<td>Read-only segment</td>
</tr>
</tbody>
</table>

#### Your Program
- C standard library
- POSIX
- glibc

- CPU
- Priv

- Architecture-independent code
- Architecture-dependent code

CSE333 lec 6 syscall fio // 01-21-14 // Perkins
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

0x00000000

0xFFFFFFFF

linux-gate.so

SP
IP

Linux
kernel

kernel stack

stack

shared libraries

heap (malloc/free)

read/write segment

.data, .bss

read-only segment

.text, .rodata

your
program

C standard
library

POSIX

glibc

architecture-independent code

architecture-dependent code

Linux kernel

priv

CPU
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 the processor back to glibc

<table>
<thead>
<tr>
<th>0xFFFFF000</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux-gate.so</td>
</tr>
<tr>
<td>Linux kernel</td>
</tr>
<tr>
<td>stack</td>
</tr>
<tr>
<td>shared libraries</td>
</tr>
<tr>
<td>heap (malloc/free)</td>
</tr>
<tr>
<td>read/write segment</td>
</tr>
<tr>
<td>.text, .rodata</td>
</tr>
<tr>
<td>read-only segment</td>
</tr>
</tbody>
</table>

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

Linux kernel

unpriv

CPU

CSE333 lec 6 syscall fio // 01-21-14 // Perkins
Details on x86 / Linux

glibc continues to execute

- might execute more system calls
- eventually returns back to your program code

0xFFFFFFFF

<table>
<thead>
<tr>
<th>Linux kernel</th>
<th>kernel stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>shared libraries</td>
<td></td>
</tr>
<tr>
<td>heap (malloc/free)</td>
<td></td>
</tr>
<tr>
<td>read/write segment</td>
<td>.data, .bss</td>
</tr>
<tr>
<td>read-only segment</td>
<td>.text, .rodata</td>
</tr>
</tbody>
</table>

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

Linux kernel

unpriv

CPU

CSE333 lec 6 syscall fio // 01-21-14 // Perkins
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)

The book: *The Linux Programming Interface* by Michael Kerrisk (keeper of the Linux man pages)
- If you want a copy: go to the book web site (man7.org/tlpi), get discount code there, then order from the publisher
  ‣ Book + ebook for cost of printed copy from Amazon
strace

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

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

```bash
echo[005c7424] execve("/bin/ls", ["ls"], [/* 47 vars */]) = 0
[003caffd] brk(0) = 0x9376000
[003cc3c3] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7800000
[003cc2c1] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
[003cc184] open("/etc/ld.so.cache", O_RDONLY) = 3
[003cc1e] fstat64(3, {st_mode=S_IFREG|0644, st_size=92504, ...}) = 0
[003cc3c3] mmap2(NULL, 92504, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb77e9000
[003cc1bd] close(3) = 0
[003cc184] open("/lib/libselinux.so.1", O_RDONLY) = 3
[003cc204] read(3, "/etc/selinux/ld.so.preload", 512) = 512
[003cc1e] fstat64(3, {st_mode=S_IFREG|0755, st_size=122420, ...}) = 0
[003cc3c3] mmap2(0x6d6000, 125948, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x6d6000
[003cc3c3] mmap2(0x6f3000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0x6f3000
[003cc3c3] mmap2(0x6f3000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c) = 0x6f3000
[003cc1bd] close(3) = 0
[003cc184] open("/lib/librt.so.1", O_RDONLY) = 3
[003cc204] read(3, "/etc/selinux/ld.so.preload", 512) = 512
[003cc1bd] close(3) = 0
... etc.
```
strace

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

```bash
bash$ strace ls 2>&1 | less
...
[00110424] open(".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY|O_CLOEXEC) = 3
[00110424] fcntl64(3, F_GETFD) = 0x1 (flags FD_CLOEXEC)
[00110424] getdents64(3, /* 6 entries */, 32768) = 184
[00110424] getdents64(3, /* 0 entries */, 32768) = 0
[00110424] close(3) = 0
[00110424] fstat64(1, {st_mode=S_IFIFO|0600, st_size=0, ...}) = 0
[00110424] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb77ff000
[00110424] write(1, "bomstrip.py\nmountlaptop.sh\ntester", ... 43
bomstrip.py
mountlaptop.sh
tester
tester.c
) = 43
[00110424] close(1) = 0
[00110424] munmap(0xb77ff000, 4096) = 0
[00110424] close(2) = 0
[00110424] exit_group(0) = ?
```

...
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
  - three streams are provided by default: **stdin**, **stdout**, **stderr**
  - you can open additional streams to read/write to files
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, "%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
}
```

`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, ...)`
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
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 Friday!