3 Lecture: The Process Model

Outline:
Announcements
Two stories
Defining of an Operating System: The System Calls
System Calls Again
System Call Mechanisms
  How to do it
OS Pre-history: The boot process
  How it all begins (on an Intel PC with a floppy)
  How it continues
  And onwards: How does the OS get control back?
OS History
  Ancient times
  The middle ages
  The renaissance: families
  Modern Times

3.1 Announcements

• Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10 (in 03-201)</td>
<td></td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• Remember:
  – Lab 1 due Today
  – Asgn 1 due Friday

• For Lab01
  – 26 done (out of 35(ish))
  – tryLab01
  – Remember how late days work: latedays

• For Asgn1:
  – don’t call sbrk(2) for every malloc(3) call.
    (quilting?)
  – remember how pointer arithmetic works (in the size of the pointee)
  – uintptr_t from <stdint.h>
  – Turn in one copy if working w/a partner.
  – About that Makefile...
3.2 Two stories
Let’s look at it again:

- Startup (Booting)
- Establishing the OS
- Switching to a user-level program
- Recovering supervisor privilege

3.3 Defining of an Operating System: The System Calls

Tanenbaum set out to write a Unix... what does that mean?

What defines an operating system from the users’ point of view?

The system calls.

Just as the instruction set architecture defines an architecture, the system calls define an Operating System.

Before we talk about IO services, implementation, etc....

Look like C functions, but are direct requests for OS services.

A system call is an entry into the kernel.

<table>
<thead>
<tr>
<th>System Call</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux: (RH7.0)</td>
<td>222</td>
</tr>
<tr>
<td>Solaris:</td>
<td>253</td>
</tr>
<tr>
<td>Minix:</td>
<td>53</td>
</tr>
</tbody>
</table>

3.4 System Calls Again

We said “An OS is defined by its system calls”. What does that mean?

A system call is the means by which the kernel provides access to a particular operating system service, and the services available through system calls (e.g., reading and writing the disk, allocating memory, starting new processes, etc.) are reserved to the kernel; there is no other way of doing these things.

3.5 System Call Mechanisms

Last time we said

- It’s all about privilege.

3.5.1 How to do it

How done? Machine dependent, usually in assembly, but hidden from the users’ view.

In any case, some sort of trap is involved to get supervisor privileges.

A trap is a software generated interrupt. Exactly what happens varies from architecture to architecture, but the result usually involves:

- saving the state of the currently executing program
• elevating of processor privilege level to “supervisor”
• transfer of control to a pre-registered routine
• *The OS does something*
• Then, usually:
  – privilege level is reduced back to user
  – original state of executing program is restored

3.6 OS Pre-history: The boot process

3.6.1 How it all begins (on an Intel PC with a floppy)
• Power on
• Boot Sector of fd0: 512b, loaded into 0x00007c00.
• jump to 0x00007c00

3.6.2 How it continues
• OS Sets up whatever it needs to set up, and, depending on what sort of system it is...
• (maybe) picks a program to run
• (maybe) shifts out of supervisor mode
• (maybe) runs the program

3.6.3 And onwards: How does the OS get control back?
How does the OS preempt a user process? It’s not running.
It takes a series of carefully-planned steps
• The O.S. runs in supervisor mode which allows it to manipulate the interrupt vector (definition) and register an ISR.
• Before changing privilege levels, the O.S.
  – installs ISRs for a timer and also for the system call interrupt, and
  – requests a timer interrupt for some future time.
• The O.S. then changes privilege levels and yields to the user process.
• Eventually, one of two things happens:
  1. the timer interrupts, or
  2. the process makes a system call

\[1440\text{kb floppy has 2 sides, 80 tracks, 18 sectors. This makes 512 bytes per sector}. \ (1474560/80)/18 \rightarrow 18432/18 \rightarrow 512\]
```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdarg.h>
#include <errno.h>

#define SYSCALL_OPEN 0x5

/* make sure this is legitimate */
 ifndef_i386
 #error "This can only be compiled for an x86"
 endif

int open(const char *pathname, int flags, ...)
{
  mode_t mode;
  va_list ap;
  int res;

  /* use the variable arguments support */
  va_start(ap,flags);
  mode = va_arg(ap,mode_t); /* extract the mode from the top of the stack */
  va_end(ap);

  /* load the given values into the registers and execute
   * the given syscall. The syscall code goes into eax
   * and return value comes from there.
   */

  asm ("movl %0,%%eax" : : "g" (SYSCALL_OPEN) : "eax");
  asm ("movl %0,%%ebx" : : "g" (pathname) : "ebx");
  asm ("movl %0,%%ecx" : : "g" (flags) : "ecx");
  asm ("movl %0,%%edx" : : "g" (mode) : "edx");
  asm ("int $0x80" : : : "eax");
  asm ("movl %0,%%eax,%0" : : "g" (res) :);

  if ( res < 0 ) {
    errno = -res;
    res = -1;
  }

  return res;
}
```

Figure 1: A Linux `open()` implementation
/* .lib/posix/_open.c */
#include <lib.h>
#include <fcntl.h>
#include <stdarg.h>
#include <string.h>

PUBLIC int open(const char *name, int flags, ...)
{
    va list argp;
    message m;

    va_start(argp, flags);
    if (flags & O_CREAT) {
        m.m1.i1 = strlen(name) + 1;
        m.m1.i2 = flags;
        m.m1.i3 = va_arg(argp, Mode_t);
        m.m1.p1 = (char *) name;
    } else {
        _loadname(name, &m);
        m.m3.i2 = flags;
    }
    va_end(argp);
    return (syscall(FS, OPEN, &m));
}

Figure 2: Minix open() implementation

/******* lib/other/syscall.c ***********/
PUBLIC int _syscall(who, syscallnr, msgptr)
    int who;
    int syscallnr;
    register message *msgptr;
{
    int status;

    msgptr->m_type = syscallnr;
    status = _sendrec(who, msgptr);
    if (status != 0) { /* _sendrec itself failed. */
        msgptr->m_type = status;
    }
    if (msgptr->m_type < 0) {
        errno = -msgptr->m_type;
        return(-1);
    }
    return(msgptr->m_type);
}

Figure 3: Minix _syscall()
/* send and receive */

sendrec() save ebp, but destroy eax and ecx.

.define _sendrec

.text

sendrec:
.push ebp
mov ebp, esp
.push ebx
mov eax, SRCDEST(ebp) ! eax = dest-src
mov ebx, MESSAGE(ebp) ! ebx = message pointer
mov ecx, BOTH ! _sendrec(srcdest, ptr)
.int SYSVEC ! trap to the kernel
.pop ebx
.pop ebp
ret

Figure 4: Minix sendrec() abstracted
Either one causes an interrupt.

- The ISR (installed by the OS and in write-protected memory) runs with supervisor privilege. Now O.S. code is running as superman again and can decide what to do.

Note: This is all about privilege. The OS gets supervisor privilege back when the interrupt handler executes. The user process cannot stop this because the user process does not have the authority to block interrupts or to change the handler.

3.7 OS History

The history of the development of operating systems parallels the history of computer architecture. (duh?)

3.7.1 Ancient times

Vacuum tubes/relays. No programming languages. Bug meant an actual insect. (aside about Dijkstra: (paraphrased) we should eliminate the use of the word bug and replace it with error.).

“Human operating systems” rewire the machines

- programmed by the builders.
- time blocked out for exclusive access
- machines rewired with plugboards
- Eventually evolved into card-fed computers.
  (*library actually meant a filing cabinet*)

3.7.2 The middle ages

Transistors made machines small and reliable. OS loaded a program, then replaced itself. Batch processing invented (IBM 1401→7094→1401)

- jobs loaded onto a tape offline
- run in the main computer
- output printed offline
- operations controlled by JCL

3.7.3 The renaissance: families

Computer families developed. Different machines of different sizes. (e.g., IBM System/360 note Brooks’ experience)

- operating systems very complex, written in assembly, tied to a particular processor (family).

Innovations:

- multiprogramming (but still essentially batch systems)
- timesharing (MULTICS: Multiplexed Information and Computing Service)
• the rise of the minicomputer! (PDP-1 1961, $120,000$ vs. $2.4M$ for a 7094).

• a neglected PDP-7 and Unix: (1969)
  – History
    * Space Travel (Thompson and Ritchie)
    * Small system
  – features
    * Small system
    * modular
    * portable (Written in C!)
      · BCPL (untyped (all datatypes are bitfields))
      · B (untyped)
      · C (weakly typed)
      · Ansi C (more strongly typed) – 1989

3.7.4 Modern Times
In modern days, the rise of the PC parallels the rise of the minicomputer:

• Windows/unix (Tanenbaum vs. Torvalds?)
  The family tree:
    – AT&T
    – BSD (Berkeley Software Distributions)
    – Linux?
    – POSIX 1988

• Network operating systems. Distributed operating systems.
  one image comprises many machines. (high availability?)

Minix vs. Linux? A matter of Philosophy

• Minix
  – Designed for readability
  – Modularity
  – Message passing
  – extensibility

• linux
  – Efficiency
  – “Production” system

\(^3\)According to the CPI, this is $968,640.80 in 2016
\(^4\)$19,372,816.05 in 2016