# 3 Lecture: History

**Outline:**

Announcements
Defining of an Operating System: The System Calls
Two stories
OS History
- Ancient times
- The middle ages
- The renaissance: families
- Modern Times

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?
- Tool of the week: Make(1)

## 3.1 Announcements

- Asgn 1 due Wedesday
- 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>`
  - About that Makefile...
  - `free(NULL)` called by `snprintf(3)`. Watch for overflows
  - (also, `-m32` and `intel-all`)
  - Compiling on 32- vs. 64-bit architectures.
  - (and `printf(3)`, etc, might use `malloc(2)`. Consider `puts(3)` and/or `write(2)`.)
- tryAsgn1
  - How to do alignment? Arithmetic: `stdint.h` and, esp., `uintptr_t`
  - `free(NULL)` called by `snprintf(3)`. Watch for overflows
  - (also, `-m32` and `intel-all`)
  - How to use late days
  - Don't leave things until the last minute
  - Path to tests can be discovered $(n/2 + 1)$

- lab02 out.
- Coming attractions:

| Event | Subject | Due Date | | | Notes |
|---|---|---|---|---|---|
| asgn2 | LWP | Mon | Jan 26 | 23:59 | |
| asgn3 | dine | Wed | Feb 4 | 23:59 | |
| lab03 | problem set | Mon | Feb 9 | 23:59 | |
| midterm | stuff | Wed | Feb 11 | | |
| lab04 | scavenger hunt II | Wed | Feb 18 | 23:59 | |
| asgn4 | /dev/secret | Wed | Feb 25 | 23:59 | |
| lab05 | problem set | Mon | Mar 9 | 23:59 | |
| asgn5 | minget and minls | Wed | Mar 11 | 23:59 | |
| asgn6 | Yes, really | Fri | Mar 13 | 23:59 | |
| final (sec01) | | Fri | Mar 20 | 10:10 | |
| final (sec03) | | Fri | Mar 20 | 13:10 | |

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

## 3.2 OS History

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

### 3.2.1 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)

### 3.2.2 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

### 3.2.3 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[2] vs. $2.4M[3] 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

### 3.2.4 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
- Where does minix fit in this?

Minix vs. Linux? A matter of Philosophy

- Minix
  - Designed for readability

[2] According to the CPI, this is $1 267 235 in 2024
[3] $25 352 698 in 2024

  - Modularity
  - Message passing
  - extensibility

- linux
  - Efficiency
  - "Production" system

## 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.

| Linux: (RH7.0) 222 |
| --- |
| Solaris: 253 |
| **Minix: 53** |

## 3.4 System Calls Again

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

| **System Call** |
| --- |
| 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[4], 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 interupt 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

[4] 1440k floppy has 2 sides, 80 tracks, 18 sectors. This makes 512 bytes per sector. (1474560/80)/18 → 18432/18 → 512

```
#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 %%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()`

```
/*********************./lib/i386/rts/_sendrec.s***************/
! See ../h/com.h for C definitions
SEND = 1
RECEIVE = 2
BOTH = 3
SYSVEC = 33

SRCDEST = 8
MESSAGE = 12
!*=============================================*
!                              _send and _receive                              *
!*=============================================*
! _sendrec() save ebp, but destroy eax and ecx.
.define _sendrec
.sect .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.

### 3.6.4 Tool of the week: Make(1)

Make is a program to control program builds automagically, but it can be much, much more.

- Based on dependencies—there's no need to regenerate a file if its source hasn't changed.

  A dependency looks like: `target: source`.

  A particular target can have multiple dependency lines

- Implicit Rules—Make knows how to do certain things (compile C source, for example: if a .o file depends on a .c file if there is one of the same name in the directory).

  You can define your own if you want.

- Explicit rules: After a tab, an series of instructions for making the thing:

```
foo.o: foo.c
        gcc -c -Wall -ansi -pedantic foo.c
```

- It supports variables. In particular CC, SHELL, CFLAGS.

  $(VARNAME) to evaluate. $$ for a dollar sign

- Remember TABS

  (Quite possibly the dumbest decision since `creat()`).

- In rules:

| | |
|---|---|
| @ | do a line silently |
| # | comment |
| - | proceed even in the face of errors |

- Interfaces nicely with RCS (will check out files for you.)

- To use: `make thing`

  if "thing" isn't specified, it makes the first target.

- Emacs: M-x compile

For more info, read the man page, or the info page.