OS APIs

CS 450 : Operating Systems
Michael Saelee <saelee@iit.edu>
1950s: Punchcards & Batch processing

- a program is completely defined by a “batch” of punchcards
- batches are manually fed into mainframes, which execute a single batch at a time (a “job”)
- programmer defines any and all routines needed for the job
1950s-1960s: Support libraries

- useful, reusable routines (e.g., for math, I/O) distributed as collections of punchcards
- these routines can be “linked” (statically) into programs without much modification
- first support libraries — the original OSes
- no standardization!
1960s: Automatic batch processing

- to keep up with faster processors, reading and starting/transitioning between jobs require automation

- “monitor” programs also keep track of usage, resources expended, etc.

- grow to become *runtime libraries* that automatically manage the execution of multiple batches of jobs (in sequence)
Pros/Cons of Batch processing?
- Pros
  - Full use of hardware
  - No worrying about other users/jobs

- Cons
  - Long wait to submit jobs / get results
  - No interactivity!
    - no live debugging
    - no feedback loop
  - Poor hardware utilization
  - Lack of reliable system libs
1970s: Rise of Timesharing

- to let many users share a computer *concurrently*, software is needed to *automatically save/restore context* between jobs
- resources (e.g., CPU & memory) are *virtualized*
- jobs are *isolated* and *protected* from each other
- overhead is offset by *increased utilization*
Mainframes and the Unix Revolution - Computerphile
https://youtu.be/-rPPqm44xLs
Pros/Cons of Timesharing?
- Pros
  - Interactivity
  - live debugging
  - fast feedback
  - Better utilization (multiple users & jobs)
  - Reliable set of system libs

- Cons
  - Overhead due to context switching / sharing
  - One crashed job (may) crash the whole system
1980s: Era of (some) bad ideas

- Consumer OSes (e.g., MS-DOS, Mac OS) of this era greatly simplify earlier offerings
  - lack of memory protection
  - cooperative multitasking vs. preemptive multitasking
- Step back in many ways for system developers
1990s-Present: Modern OSes

- Preemptively multitasked OSes are the norm
- High degrees of *virtualization*, *isolation*, and *concurrency*
- Robust, efficient, largely abstracted I/O layer
- Large, sophisticated system call interfaces
  - Standardization attempts for portability
Portable Operating System Interface (POSIX)
IEEE Std 1003.1™-2017
The Open Group Technical Standard Base Specifications, Issue 7
http://pubs.opengroup.org/onlinepubs/9699919799/
What does POSIX specify?

- Note: “system interface” ≠ system call interface

- Categories of APIs:
  - Process management, memory management, file operations, I/O operations, IPC (local and networking), concurrency control (threads, locks, etc.), …

- And much more!
Pros/Cons of modern OS design?
- Pros
  - Transparent multi-user, multi-job multiplexing
  - Large amount of hardware abstraction to simplify access/development
  - Sophisticated system APIs

- Cons
  - Tons of overhead!
  - Rigid system APIs
    - not necessarily designed for programmer convenience
  - not needed/desired by all applications!
Looking Ahead

- OS-as-library type architectures are making a comeback!
- Focus on security and robust access to hardware
- Very little system-level abstraction/virtualization
- Significantly reduced overhead — any desired abstractions are provided with user-space libraries
Traditional operating systems limit the performance, flexibility, and functionality of applications by fixing the interface and implementation of operating system abstractions such as interprocess communication and virtual memory. The exokernel operating system architecture addresses this problem by providing application-level management of physical resources. In the exokernel architecture, a small kernel securely exports all hardware resources through a low-level interface to untrusted library operating systems. Library operating systems use this interface to implement system objects and policies. This separation of resource protection from management allows application-specific customization of traditional operating system abstractions by extending, specializing, or even replacing libraries.

So what sort of OS are we going to inspect/build?
Those who do not understand Unix are condemned to reinvent it, poorly

- Henry Spencer
Previously: Unix v6, ca. 1975
< 10,000 lines of code; multi-user, preemptively multitasked OS
- thanks to the power of modern emulators, can boot up, tweak, re-build, and debug it!

- but!

- non-ANSI C

- archaic architecture (PDP-11)

- seeming irrelevance (not true)
- instead, use x86-based v6 clone developed at MIT

- monolithic, preemptively multitasked, multiprocessor-capable, 32-bit, UNIX-like OS
### limited syscall API:

<table>
<thead>
<tr>
<th><strong>System call</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fork()</code></td>
<td>Create process</td>
</tr>
<tr>
<td><code>exit()</code></td>
<td>Terminate current process</td>
</tr>
<tr>
<td><code>wait()</code></td>
<td>Wait for a child process to exit</td>
</tr>
<tr>
<td><code>kill(pid)</code></td>
<td>Terminate process pid</td>
</tr>
<tr>
<td><code>getpid()</code></td>
<td>Return current process's id</td>
</tr>
<tr>
<td><code>sleep(n)</code></td>
<td>Sleep for n seconds</td>
</tr>
<tr>
<td><code>exec(filename, *argv)</code></td>
<td>Load a file and execute it</td>
</tr>
<tr>
<td><code>sbrk(n)</code></td>
<td>Grow process’s memory by n bytes</td>
</tr>
<tr>
<td><code>open(filename, flags)</code></td>
<td>Open a file; flags indicate read/write</td>
</tr>
<tr>
<td><code>read(fd, buf, n)</code></td>
<td>Read n bytes from an open file into buf</td>
</tr>
<tr>
<td><code>write(fd, buf, n)</code></td>
<td>Write n bytes to an open file</td>
</tr>
<tr>
<td><code>close(fd)</code></td>
<td>Release open file fd</td>
</tr>
<tr>
<td><code>dup(fd)</code></td>
<td>Duplicate fd</td>
</tr>
<tr>
<td><code>pipe(p)</code></td>
<td>Create a pipe and return fd's in p</td>
</tr>
<tr>
<td><code>chdir(dirname)</code></td>
<td>Change the current directory</td>
</tr>
<tr>
<td><code>mkdir(dirname)</code></td>
<td>Create a new directory</td>
</tr>
<tr>
<td><code>mknod(name, major, minor)</code></td>
<td>Create a device file</td>
</tr>
<tr>
<td><code>fstat(fd)</code></td>
<td>Return info about an open file</td>
</tr>
<tr>
<td><code>link(f1, f2)</code></td>
<td>Create another name (f2) for the file f1</td>
</tr>
<tr>
<td><code>unlink(filename)</code></td>
<td>Remove a file</td>
</tr>
</tbody>
</table>
very limited set of user-level programs:

- shell, cat, echo, grep, kill,
  ln, ls, mkdir, rm, wc

- no compiler/debugger/editor

  - development (kernel/user) takes place on another platform!
Next time: Introduction to xv6 and x86/PC development
But we still have some time before we’re able to work on xv6!
Also nice to know how a “real” modern OS is put together …

**Assignment 2: Linux From Scratch**

- compile and set up Linux kernel & utils direct from source
- create a customized disk image, bootable from emulator