L5: Operating Systems

Frans Kaashoek
6.033 Spring 2011
Operating systems

• Today: organization
• Next week: techniques for managing multiple activities
• Week after: virtual machines and performance
OS challenge and goals

• **Challenge:**
  • one computer, many programs

• **Goals:**
  • Multiplexing
  • Protection
  • Cooperation
  • Portability
  • Performance
Approach

• Virtualization
• Abstractions
One program per computer

- Memory holds *instructions* and *data*
- CPU *interprets* instructions
Strawman solution

- OS switches processor(s) between programs

```c
Program 1:
for (;;) {
    next instruction
}
```

Main memory

- Program 1
- Program 2
- Program 3
- Data for P1
- Data for P2
- Data for P3
Problem: no boundaries

- A program can modify other programs' data
- A program jumps into other program's code
- A program may get into an infinite loop
Goal: enforcing modularity

- Give each program its private memory for code, stack, and data
- Prevent one program from getting out of its memory
- Allowing sharing between programs when needed
- Force programs to share processor
Approach: memory virtualization

- Modify processor to support virtual addresses
Table records mapping

- Each program has its own translation map
  - Physical memory doesn’t have to be contiguous
- When switching program, switch map
- Maps stored in main memory
### Space-efficient map

<table>
<thead>
<tr>
<th>Address</th>
<th>Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 bits</td>
<td>12 bits</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vp</th>
<th>pp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- \(0x02020 \rightarrow 4 \times 4096 + 0x20 = 0x4020\)
Intel x86-32 two-level page table

- Page size is 4,096 bytes
  - 1,048,576 pages in $2^{32}$
  - Two-level structure to translate
x86 page table entry

- **W**: writable?
  - Page fault when $W = 0$ and writing
- **U**: user mode references allowed?
  - Page fault when $U = 0$ and user references address
- **P**: present?
  - Page fault when $P = 0$
Page fault

- Switches processor to a predefined handler in software
  - Handler can stop program and raise error
  - Handler can update the page map and resume at faulted instruction
**Naming view**

- **Apply naming model:**
  - Name = virtual address
  - Value = physical address
  - Context = Page map
  - Lookup algorithm: index into page map

- **Naming benefits**
  - Sharing
  - Hiding
  - Indirection (demand paging, zero-fill, copy-on-write, ...)

Threat: page map address is unprotected

• If program can modify page map, then it can access any physical address
• Must protect page table!
Protecting page maps: kernel and user mode

- Kernel mode: can change page-map register, U/K
- In user mode: cannot
- Processor starts in kernel mode
- On interrupts, processor switches to kernel mode
What is a kernel?

- The code running in kernel mode
  - Trusted program: e.g., sets page-map, U/K register
  - All interrupt handlers (e.g. page fault) run in kernel mode
How transfer from U to K, and back?

• Special instruction: e.g., int #

• Processor actions on int #:
  • Set U/K bit to K
  • Lookup # in table of handlers
  • Run handler

• Another instruction for return (e.g., reti)
  • Kernel sets U/K bit to U
  • Calls reti
Process: a virtual processor

- Kernel sets up a hard timer to deliver interrupt every, say, 100 msec
- Interrupts transfers control to kernel
  - Kernel saves current program state
    - Program counter, stack pointer, pmap reg, etc.
  - Kernel chooses a runnable program
  - Kernel loads saved program state
  - Kernel returns from interrupt
  - Processors resumes execution w. new state
- No process can hog processor
Abstractions

• Pure virtualizing is often not enough
  • E.g., Portability
  • E.g., Cooperation

• Example OS abstractions:
  • Disk -> FS
  • Display -> Windows
  • DRAM -> heap w. allocate/deallocate

• Design of abstraction important
main() {
    int fd, n;
    char buf[512];

    chdir("/usr/rtm");

    fd = open("quiz.txt", 0);
    n = read(fd, buf, 512);
    write(1, buf, n);
    close(fd);
}

Where do abstractions live?

- Library in user space?
  - No! Program must use disk only through abstraction
- Use kernel to enforce abstractions
- Methods of abstraction are system calls
  - E.g., int 33
How does kernel read user memory?

- `read(fd, buf, 512)`
- Kernel and user share page map
  - E.g., user program in low addresses
  - E.g., kernel in high addresses
  - Page map entry has U/K bit
    - Set U bit only for low addresses
Kernel complexity

- 1975 Unix kernel: 10,500 lines of code
- 2008 Linux 2.6.24 line counts:
  - 85,000 processes
  - 430,000 sound drivers
  - 490,000 network protocols
  - 710,000 file systems
  - 1,000,000 different CPU architectures
  - 4,000,000 drivers
  - 7,800,000 Total
Monolithic kernel

- Avoiding chaos:
  - Internal interfaces simplify
  - Loadable kernel modules
Microkernel

- Apply client/server to OS
- Kernel maps devices into driver’s address space
Micro versus monolithic

- Many kernels are monolithic
- Why change a working kernel?
- Microkernel benefits not that easy to get
  - Message more costly than function calls
  - Sharing between servers not that easy
  - What do you do if file server is down?
- Easy to have servers on monolithic too
Summary

- OS virtualizes for sharing/multiplexing
- Abstract for portability and cooperation
- Provides enforced modularity:
  - Program versus programs
  - Program versus kernel