Lecture #6: Virtual Machines and Performance
wrapping up operating systems
operating systems enforce modularity on a single machine using virtualization in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

2. programs should be able to communicate with each other

3. programs should be able to share a CPU without one program halting the progress of the others

**today’s goal:** run multiple operating systems at once
**common computing environment:** running multiple OSes on a single physical machine

- **host OS**
  - virtual machine
    - running guest OS

- **physical hardware**
  - U/K, PTR, page table, ...
common computing environment: running multiple OSes on a single physical machine

<table>
<thead>
<tr>
<th>virtual machine running guest OS</th>
<th>virtual machine running guest OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical hardware</td>
<td></td>
</tr>
<tr>
<td>U/K, PTR, page table, …</td>
<td></td>
</tr>
</tbody>
</table>

problem: how to (safely) share physical hardware?
**virtual machine monitor:** virtualizes the physical hardware for the guest OSes

- virtual machine running guest OS
- virtual machine running guest OS

**virtual machine monitor (VMM)**

**physical hardware**

- U/K, PTR, page table, ...

Guest OSes run in user mode.

Privileged instructions in guest OS will cause an exception, which the VMM will intercept (“trap”) and **emulate**. If the VMM can’t emulate an instruction, it will send the exception back to the guest OS for handling.

**First question:** what does it mean to emulate?
virtual machine monitor: virtualizes the physical hardware for the guest OSes

first example: virtualizing memory (again)

first question: what does it mean to emulate?

in this example, it means that the VMM needs to step in and translate guest physical addresses to host physical addresses
virtual machine monitor: virtualizes the physical hardware for the guest OSes

1. guest OS loads its PTR, which triggers an exception; the VMM intercepts
2. VMM combines the guest page table with its own page table to create a host page table
3. physical hardware uses the host page table
**virtual machine monitor:** virtualizes the physical hardware for the guest OSes

**first example: virtualizing memory (again)**

1. guest OS loads its PTR, which triggers an exception; the VMM intercepts

   in modern hardware, the physical hardware is aware of both page tables, and performs the translation from guest virtual to host physical itself
**virtual machine monitor:** virtualizes the physical hardware for the guest OSes

<table>
<thead>
<tr>
<th>guest OS</th>
<th>guest OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual hardware</td>
<td>virtual hardware</td>
</tr>
<tr>
<td>U/K</td>
<td>U/K</td>
</tr>
<tr>
<td>PTR</td>
<td>PTR</td>
</tr>
<tr>
<td>page table</td>
<td>page table</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>virtual machine monitor (VMM)</th>
<th>physical hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>U/K, PTR, page table, ...</td>
<td>...</td>
</tr>
</tbody>
</table>

guest OSes run in user mode

privileged instructions in guest OS will cause an exception, which the VMM will intercept ("trap") and **emulate**

if the VMM can’t emulate an instruction, it will send the exception back to the guest OS for handling

**second question:** what about when the VMM needs to emulate an instruction that doesn’t trigger an exception?
**virtual machine monitor**: virtualizes the physical hardware for the guest OSes

**second example**: virtualizing the U/K bit

<table>
<thead>
<tr>
<th>guest OS</th>
<th>guest OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual hardware</td>
<td>virtual hardware</td>
</tr>
<tr>
<td>U/K</td>
<td>U/K</td>
</tr>
<tr>
<td>PTR</td>
<td>PTR</td>
</tr>
<tr>
<td>page table</td>
<td>page table</td>
</tr>
<tr>
<td>virtual machine monitor (VMM)</td>
<td>virtual machine monitor (VMM)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>physical hardware</th>
<th>physical hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>U/K, PTR, page table, ...</td>
<td>U/K, PTR, page table, ...</td>
</tr>
</tbody>
</table>

**para-virtualization**: modify guest OS slightly

**binary translation**: VMM replaces problematic instructions with ones that it can trap and emulate

**hardware support**: architecture provides a special operating mode for VMMs in addition to user mode, kernel mode

**second question**: what about when the VMM needs to emulate an instruction that doesn’t trigger an exception?
**monolithic kernel:** no enforced modularity within the kernel itself

**Hardware**

Basic interprocess communication, virtual memory, scheduling, file server, device drivers, network, ...

**Application**

**microkernels:** enforce modularity by putting subsystems in user programs

**Hardware**

Basic interprocess communication, virtual memory, scheduling

**Application**

application

IPC

device driver

network

...
**performance:** performance issues have influenced a lot of the system designs you’ve seen so far

**latency:** how long does it take to complete a single request?

example: how long does it take to retrieve a particular piece of memory in an OS?

**throughput:** how many requests per unit of time?

example: how many reads or writes can a system do to a disk at once?

**utilization:** what fraction of resources are being utilized? this puts our performance measurements in context

often we see latency remain low until the system is heavily utilized, and then it rises.

throughput, on the other hand, increases as requests increase, and maxes out when the system is fully utilized
virtual machines allow us to run multiple isolated OSes on a single physical machine, similar to how we used an OS to run multiple programs on a single CPU.

monolithic kernels provide no enforced modularity within the kernel. microkernels do, but redesigning monolithic kernels as microkernels is challenging

performance affects all aspects of system design. throughput, latency, and utilization are important metrics (but not the only ones), and we have general, systems-level techniques for attempting to improve them (e.g., caching, batching). knowing when to apply them requires a solid understanding of the system itself and how it’s used