Memory Management Part 2
Nested Virtualization

- Real Machine (L0)
  - VMM$_0$
  - Virtual Machine (L1)
    - VMM$_1$
    - Virtual Machine (L2)
      - VMM$_2$
VMX

• New processor mode: root
  – ring -1: root mode
  – rings 0-3: non-root mode
• Certain actions cause processor in non-root mode to switch to root mode
  – VMexit
• When in root mode, processor can switch back to non-root mode
  – VMenter
VMCS

• Virtual machine control structures
  – guest state
    - virtualized CPU registers (non-root mode)
  – host state
    - registers to be restored when switching to root mode (VMexit)
  – control data
    - which events in non-root mode cause VMexits
Nested Virtualization on VMX

• The VMM is designed to use VMX extensions (including EPT)
• It supports VMs that appear to be real x86’s (but without VMX extensions)
• Can the VMM run in a VM of the level-0 VMM?
Nested Virtualization with VMX

<table>
<thead>
<tr>
<th>Real Machine (L0)</th>
<th>Guest OS</th>
<th>Guest OS Map</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Virtual Machine (L1)</td>
<td></td>
</tr>
<tr>
<td>VMM&lt;sub&gt;0&lt;/sub&gt;</td>
<td>VMCS</td>
<td>VMem Map</td>
</tr>
<tr>
<td></td>
<td>Virtual Machine (L2)</td>
<td></td>
</tr>
<tr>
<td>VMM&lt;sub&gt;1&lt;/sub&gt;</td>
<td>VMCS</td>
<td>VMem Map</td>
</tr>
<tr>
<td></td>
<td>Guest OS</td>
<td>Guest OS Map</td>
</tr>
</tbody>
</table>
Composed Virtualization

Real Machine (L0)

Virtual Machine (L1)

VMM₀

VMCS 0-1

VMem Map 0-1

Virtual Machine (L2)

VMM₁

VMCS 1-2

VMem Map 1-2

Guest OS

Guest OS Map

VMCS 0-2

VMem Map 0-2

Guest OS
Traditional OS Paging Issues

- Fetch policy
- Placement policy
- Replacement policy
A Simple Paging Scheme

• Fetch policy
  – start process off with no pages in primary storage
  – bring in pages on demand (and only on demand) (this is known as demand paging)

• Placement policy
  – it doesn’t matter — put the incoming page in the first available page frame

• Replacement policy
  – replace the page that has been in primary storage the longest (FIFO policy)
Performance

1) Trap occurs (page fault)
2) Find free page frame
3) Write page out if no free page frame
4) Fetch page
5) Return from trap
Improving the Fetch Policy

Fault here
Bring these in as well
Improving the Replacement Policy

• When is replacement done?
  – doing it “on demand” causes excessive delays
  – should be performed as a separate, concurrent activity

• Which pages are replaced?
  – FIFO policy is not good
  – want to replace those pages least likely to be referenced soon
The “Pageout Daemon”

Diagram:
- **In-Use Page Frames**
- **Pageout Daemon**
- **Disk**
- **Free Page Frames**
Choosing the Page to Remove

• Idealized policies:
  – FIFO (First-In-First-Out)
  – LRU (Least-Recently-Used)
  – LFU (Least-Frequently-Used)
Implementing LRU
Clock Algorithm

**Back hand:**
if (reference bit == 0)
    remove page

**Front hand:**
reference bit = 0
Global vs. Local Allocation

- Global allocation
  - all processes compete for page frames from a single pool
- Local allocation
  - each process has its own private pool of page frames
Thrashing

- Consider a system that has exactly two page frames:
  - process A has a page in frame 1
  - process B has a page in frame 2
- Process A causes a page fault
- The page in frame 2 is removed
- Process B faults; the page in frame 1 is removed
- Process A resumes execution and faults again; the page in frame 2 is removed
- ...

The Working-Set Principle

• The set of pages being used by a program (the working set) is relatively small and changes slowly with time
  – \( WS(P,T) \) is the set of pages used by process \( P \) over time period \( T \)

• Over time period \( T \), \( P \) should be given \( |WS(P,T)| \) page frames
  – if space isn’t available, then \( P \) should not run and should be swapped out
Linux Intel x86 VM Layout

- Kernel: 4GB
- User: 3GB
- Total: 7GB
Memory Allocation

- **User**
  - virtual allocation
    - fork
    - pthread_create
    - exec
    - brk
    - mmap
  - real allocation
    - (not done)

- **OS kernel**
  - virtual allocation
    - fork, etc.
    - kernel data structures
  - real allocation
    - page faults
    - kernel data structures
Linux and Real Memory

kernel

user

Virtual Memory

3GB

1GB

Real Memory
Lots of Real Memory

Virtual Memory

kernel

user

Real Memory

3GB

1GB
Address Space

OS kernel

Illegal

User

0xffffffffffffffff

0xfffff80000000000

0xfffff7fffffffffff

0x0000800000000000

0x00007fffffffffff

2^{47} bytes

2^{64} – 2^{48} bytes

2^{47} bytes
Mem_map and Zones

- Zone HighMem
- Zone Normal
- Zone DMA

mem_map

page frames
Page Lists

Zone DMA

Zone Normal

Zone HighMem

Free Pages

Inactive Pages

Active Pages
Buddy Lists

32K

16K

16K

8K

8K

4K

4K
Slab Allocation
Page Management

• Replacement
  – two-handed clock algorithm
  – applied to zones in sequence
  – essentially global in scope
Page Scanning

Zone DMA

Zone Normal

Zone HighMem

Free Pages

Inactive Pages

Active Pages

DMA Zone

Normal Zone

HighMem
Windows x86 Layout

- **Kernel**: 0 - 4GB
- **User**: 4GB - 7GB

- **Kernel**: 0 - 2GB
- **User**: 2GB - 4GB
Windows Paging Strategy

- All processes guaranteed a “working set”
  - lower bound on page frames
- Competition for additional page frames
- “Balance-set” manager thread maintains working sets
  - one-handed clock algorithm
- Swapper thread swaps out idle processes
  - first kernel stacks
  - then working set
- Some of kernel memory is paged
  - page faults are possible
Windows Page-Frame States

- Active
- Modified
- Standby
- Free
- Zeroed
- Transition

Transition arrows indicate the state transitions between different page-frame states in the Windows operating system.