Memory Management Part 2
Nested Virtualization

- \( VMM_2 \)
- Virtual Machine (L2)
- \( VMM_1 \)
- Virtual Machine (L1)
- \( VMM_0 \)
- Real Machine (L0)
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

- **Real Machine (L0)**
  - VMM\(_0\)
  - VMCS
  - VMem Map

- **Virtual Machine (L1)**
  - VMM\(_1\)
  - VMCS
  - VMem Map

- **Virtual Machine (L2)**
  - Guest OS
  - Guest OS Map
Composed Virtualization

<table>
<thead>
<tr>
<th>Guest OS</th>
<th>Guest OS Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Machine (L2)</td>
<td></td>
</tr>
<tr>
<td>$VMM_1$</td>
<td>VMCS 1-2</td>
</tr>
<tr>
<td>Virtual Machine (L1)</td>
<td></td>
</tr>
<tr>
<td>$VMM_0$</td>
<td>VMCS 0-1</td>
</tr>
<tr>
<td>Real Machine (L0)</td>
<td></td>
</tr>
</tbody>
</table>
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”

In-Use Page Frames → Pageout Daemon → Free Page Frames

Disk
Choosing the Page to Remove

• Idealized policies:
  – FIFO (First-In-First-Out)
  – LRU (Least-Recently-Used)
  – LFU (Least-Frequently-Used)
Implementing LRU
Quiz 1

Your computer is running one process. Pretty much all available real memory is being actively used and processor utilization is around 90%. You now add another process that’s similar to the first in terms of both memory and processor utilization. Assume the LRU page replacement policy is used.

a) Processor utilization will rise to nearly 100%.

b) Processor utilization will stay at around 90%.

c) Processor utilization will drop precipitously.
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
Two Issues

• If a process is active, which of its pages should be in real memory?
• If there is too much of a demand for memory, which processes should run (and which should not run)?
Clock Algorithm

**Front hand:**
reference bit = 0

**Back hand:**
if (reference bit == 0)
    remove page
Linux Intel x86 VM Layout

- Kernel: 4GB
- User: 3GB

Operating Systems in Depth
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

Virtual Memory

kernel

user

3GB

0

Real Memory

1GB

0
Lots of Real Memory

- Kernel: 3GB
- User: 1GB

Virtual Memory

Real Memory
Address Space

- **OS kernel**: 0xffffffffffffffff
- **Illegal**: 0xffffffff80000000000000
- **User**: 0x00007fffffffffff

- 2^47 bytes
- 2^64 – 2^48 bytes
- 2^47 bytes

Addresses:
- 0xffffffffffffffff
- 0xffffffff80000000000000
- 0xffffffff7fffffffffff
- 0x00007fffffffffff
- 0x0000000000000000
- 0x0000000000000000
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
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
Buddy Lists

- 32K
- 16K
- 8K
- 4K
Quiz 2

You’re designing the algorithm for allocating an often-used kernel data structure that fits within a cache line. Which of the following is the most important thing to consider, with respect to the time required to access the data structure, for each allocation?

a) whether the size is rounded up to a power of 2
b) whether the address of the data structure is aligned to a particular power of 2
c) whether the address of the data structure is offset from a particular power of 2, and by how much
d) none of these really matter
E-Way Set-Associative Cache

\[ E = 2^e \text{ lines per set} \]

\[ S = 2^s \text{ sets} \]

Address of word:
- t bits
- s bits
- b bits

- tag
- set index
- block offset

data begins at this offset

valid bit

\[ B = 2^b \text{ bytes per cache block (the data)} \]
Intel Core i5 and i7 Cache Hierarchy

Processor package

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

L3 unified cache (shared by all cores)

Main memory

L1 i-cache and d-cache:
- 32 KB, 8-way,
- Access: 4 cycles

L2 unified cache:
- 256 KB, 8-way,
- Access: 11 cycles

L3 unified cache:
- 8 MB, 16-way,
- Access: 30-40 cycles

Block size: 64 bytes for all caches
Slab Allocation

![Slab Allocation Diagram](image-url)
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