Memory (Part III):
Virtual Memory

Professor Travis Peters
CSCI 460 Operating Systems
Fall 2019

Some slides & figures adapted from Stallings instructor resources.
Some slides adapted from Adam Bates’s F’18 CS423 course @ UIUC
https://courses.engr.illinois.edu/cs423/sp2018/schedule.html
Today

Announcements
• Project Proposal Due 11/01!
• HW5 posted today (Due 11/01)
• PA2 posted today (Due 11/08) — two weeks… but start early ;)

Goals & Learning Objectives
• Understand Virtual Memory (and its connection to paging & segmentation)
Last Time...

Paging & Segmentation

- Programs broken into non-contiguous pieces (pages or segments)
- Logical addresses dynamically translated into physical addresses at run-time

→ Not all memory needs to be in MM during execution!
Virtual Memory — The Basics

Paging & Segmentation

• Programs broken into non-contiguous pieces (pages or segments)
• Logical addresses dynamically translated into physical addresses at run-time

→ Not all memory needs to be in MM during execution!

So what is Virtual Memory?

• All memory can be addressed as if it were part of main memory
• References to memory are dynamically translated
• Limited only by the addressing scheme of the system & amount of secondary memory

Image Credit: https://www.enterprisestorageforum.com/storage-hardware/virtual-memory.html
Virtual Memory — How?

How it Works
• To start, bring bare minimum of process into main memory (MM)
  • **Resident Set** == the part of a process in MM
• Access process memory in the normal way!

What if a logical address references a part of memory that is not in MM?
→ page/segment table(s) make it easy to know if something is in MM
→ generate interrupt: “address not in memory!” & move process to “blocked” state
→ bring missing part of process into MM (i.e., OS reads from disk)
→ …schedule other processes to run…
→ I/O interrupt indicates that missing process “piece” has been read/loaded in MM
→ blocked process moves back to “ready” state
Virtual Memory — Why?

1. More processes can be maintained in main memory
   - We need not load all parts of every process, so MM can accommodate many more processes
   - More processes → more likelihood of processes ready to run → better processor utilization

2. A process may be larger than all of main memory!
   - No need to worry about actual limitations of MM (1MB, 1GB, 4GB, 16GB — whatever!)
   - No need for programer to worry about structure/size of program
     - OS breaks up program into arbitrary pieces; most are stored outside of MM
     - OS brings in pieces when needed

Any Limitations?
- Virtual Memory is only constrained by the availability of secondary memory
  - **Real Memory** — where processes execute; main memory
  - **Virtual Memory** — where processes can be stored as needed; the perceived (available) memory space
Virtual Memory — Locality

- **Principle of Locality**
  - Work (program/data references) within a process tend to cluster
  - Over a short period of time, execution is likely confined to small section of a program
  - **Example:** a particular (sub)routine
    - wasteful to spend time loading in all parts of a program in the short time before process is swapped/suspended
    - trigger a “fault” if something is needed that isn’t in memory (i.e., tell OS to load something into MM)

- **Thrashing**
  - Don’t want to spend excessive time loading/unloading stuff into MM (i.e., handling swapping, rather than executing meaningful computations)
  - OS’s job: try to “guess” based on recent history what is likely to be needed (load/keep that stuff…)
## Towards Virtual Memory — Characteristics of Paging & Segmentation

<table>
<thead>
<tr>
<th>Simple Paging</th>
<th>Virtual Memory Paging</th>
<th>Simple Segmentation</th>
<th>Virtual Memory Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory partitioned into small fixed-size chunks called frames</td>
<td>Main memory not partitioned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program broken into pages by the compiler or memory management system</td>
<td>Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internal fragmentation within frames</td>
<td>No internal fragmentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No external fragmentation</td>
<td>External fragmentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating system must maintain a page table for each process showing which frame each page occupies</td>
<td>Operating system must maintain a segment table for each process showing the load address and length of each segment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating system must maintain a free frame list</td>
<td>Operating system must maintain a list of free holes in main memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor uses page number, offset to calculate absolute address</td>
<td>Processor uses segment number, offset to calculate absolute address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All the pages of a process must be in main memory for process to run, unless overlays are used</td>
<td>Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed</td>
<td>All the segments of a process must be in main memory for process to run, unless overlays are used</td>
<td>Not all segments of a process need be in main memory for the process to run. Segments may be read in as needed</td>
</tr>
<tr>
<td></td>
<td>Reading a page into main memory may require writing a page out to disk</td>
<td></td>
<td>Reading a segment into main memory may require writing one or more segments out to disk</td>
</tr>
</tbody>
</table>
Memory Management Formats

Virtual Address
- Page Number
- Offset

Page Table Entry
- Page Number
- Other Control Bits
- Frame Number

Virtual Address
- Segment Number
- Offset

Segment Table Entry
- Segment Number
- Other Control Bits
- Length
- Segment Base

Paging
- If Page is Present (P)
  → use frame number

Segmentation
- If Page Modified (M)
  → write out
Any (potential) problems?

What if amount of VM is huuuuge?!

Example:
$2^{31} = 2$ GB Virtual Memory
$2^9 = 512$ B pages

Q: How many pages are possible?
(How many page table entries could there be?)

$$2^{31} \div 2^9 = 2^{31-9} = 2^{22}$$
page table entries per process!

We could store page tables in VM as well!
Any problems with that?
Address Translation in 2-Level Paging System

Page tables themselves could be paged in/out… but at least part of a page table must be in MM for a process to run… → 2-Level Paging System

- 10 bits
- 10 bits
- 12 bits

Virtual Address

Program

Paging Mechanism

Main Memory

$X$ page tables
$Y$ entries per page table
$= X \times Y$ total pages

Typically, size of a page table $\leq$ size of a page i.e., a 2nd-level page table can fit within a single page