Motivation

- We talked about memory management but several problems remain
- External Fragmentation
- Processes (or sum of processes) larger than physical memory
- Sharing

Sharing Between Processes

- Schemes so far have considered only a single address space per process
  - A single name space per process
  - No sharing

Multiple Name Spaces

How can one share code and data between programs?

Program P's Virtual Address Space

- Heap
- Run-Time Stack
- Program Data
- Program Text

Mapped identically for all processes

User Code

Program Data

Run-Time Stack

Heap

$2^n-1$
Segmentation

- New concept: A segment — a memory “object”
  - A virtual address space
- A process now addresses objects — a pair $\langle s, addr \rangle$
  - $s$ — segment number
  - $addr$ — an offset within an object
- Two separate registers. Segment register + offset register
- Single register. Logically divided

Memory Management Basics

Segmentation allows sharing
- ... but leads to poor memory utilization
  - We might not use much of a large segment, but we must keep the whole thing in memory (bad memory utilization).
  - Suffers from external fragmentation
  - Allocation/deallocation of arbitrary size segments is complex
- Writing problems to adapt to the amount of System Memory is very difficult.
  - Write it to a minimum, but when more is available, the application can’t use it.
  - Write it to the maximum, but then it won’t run on machines with less.
- What if my program must have more memory than the minimum?
  - Use a technique called overlays.

Paging

- Physical memory partitioned into equal sized frames
- A memory address is a pair $\langle f, o \rangle$
  - $f$ — frame number ($f_{\text{max}}$ frames)
  - $o$ — frame offset ($o_{\text{max}}$ bytes/frames)
  - Physical address = $o_{\text{max}} \times f + o$

Physical Address Specification

- Example: A 16-bit physical address space with 512 byte frames
  - $\log_2 512 = 9 \Rightarrow 9$ bits for offset
  - 7 bits for frame number
  - $\langle f, o \rangle = (3,6)$
  - $f = 0 \times 3 + 3 = 3$
  - $o = 6$
  - Physical address = $0 \times 3 + 6 = 6$
  - PA: 0000001100001110 = 1542 = 0x606
Paging

- A process' virtual address space is partitioned into equal sized pages

A virtual address is a pair \((p, o)\)
- \(p\) — page number \((p_{\text{max}}\) pages\)
- \(o\) — page offset \((o_{\text{max}}\) bytes/pages\)

Virtual address \(= o_{\text{max}}\times p + o\)

Virtual Address Translation

- A page table maps (virtual) pages to (physical) frames

Page Table Implementation

- 1 table per process
- Part of process' state

Contents:
- Flags — dirty bit, resident bit, clock/reference bit
- Frame number
Virtual Address Translation

- Problem — VM reference requires 2 memory references!
  - One access to get the page table entry
  - One access to get the data

- Page table can be very large; a part of the page table can be on disk.
  - For a machine with 64-bit addresses and 4 k pages, what is the size of a page table?

- What to do?
  - Most computing problems are solved by some form of...
    - Indirection
    - here: Caching

Translation Lookaside Buffer

- Cache recently accessed page-to-frame translations in a TLB
  - For TLB hit, physical page number obtained in 1 cycle
  - For TLB miss, translation is updated in TLB
  - Has typically better than 99% hit ratio!!

Multi-Level Paging

- Add additional levels of indirection to the page table by sub-dividing page number into k parts
  - Create a “tree” of page tables
  - TLB still used, just not shown

Virtual Memory

- We have achieved:
  - Eliminated external fragmentation
  - Enabled sharing

- TBD:
  - Process with more virtual than physical memory

- Problem: Size of the page table
  - 32 bit, assuming 4k pages
    - $2^{32}$ addresses
    - $2^{20} \approx 1$ million pages
Multi-Level Paging

- Example: Two-level paging

<table>
<thead>
<tr>
<th>CPU</th>
<th>p1</th>
<th>p2</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>16</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

- Virtual Addresses
- Physical Addresses

The Problem of Large Address Spaces

- So far, we have looked at Hierarchical Paging (aka. Forward mapped page table) implementations.

- With large address spaces (64-bits) forward mapped page tables become cumbersome.
  - E.g. 7 levels of tables, requires 7 memory references to resolve an address.
  - With several address spaces active simultaneously, they consume large amounts of system memory (must be resident).

- Total (virtual) address space size is growing faster than system (real) memory size.

Mappings are limited by total system memory

- Other schemes make use of the fact that the number of pages mapped to physical frames is proportional to the number of frames, not the sum of all the address spaces:

- HTabs – Hashed Page Tables

- Inverted Page Tables

HTabs – Hashed Page Tables

- Hashing function (used by both HW and OS) maps a Virtual Address (page number, pid, p) to an index.

- A fixed-location table is indexed by index, which points to the first of a linked list of mapping VA to frame mappings that looks like:
  <pid, page#, frame#, next entry>
Inverted Page Tables

- Hash page numbers to find corresponding frame number
  - Page frame number is not explicitly stored (1 frame per entry)
  - Protection, dirty, used, resident bits also in entry

Inverted Page Table – For N frames of memory