CIS 415: Operating Systems

Paging

Spring 2013
Prof. Kevin Butler
Memory Mgmt. Basics

• Allocation
  ‣ Previously
    • Allocate arbitrary-sized chunks (e.g., in old days, a process; now arbitrary allocation done on the heap)
  ‣ Challenges
    • Fragmentation and performance

• Swapping
  ‣ Need to use the disk as a backing store for limited physical memory
  ‣ Problems
    • Complex to manage backing of arbitrary-sized objects
    • May want to work with subset of process (later)
• Programs are provided with a virtual address space (say 1 MB).
• Role of the OS to fetch data from either physical memory or disk.
  ‣ Done by a mechanism called (demand) paging.
• Divide the virtual address space into units called “virtual pages” each of which is of a fixed size (usually 4K or 8K).
  ‣ For example, 1M virtual address space has 256 4K pages.
• Divide the physical address space into “physical pages” or “frames”.
  ‣ For example, we could have only 32 4K-sized pages.
• Role of the OS to keep track of which virtual page is in physical memory and if so where?
  ‣ Maintained in a data structure called “page-table” that the OS builds.
  ‣ “Page-tables” map Virtual-to-Physical addresses.
Page Tables

Virtual Address

Virtual Page # | Offset in Page
---|---

<table>
<thead>
<tr>
<th>VP #</th>
<th>PP #</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_{P1})</td>
<td>(p_{P1})</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(v_{Pn})</td>
<td>(p_{Pn})</td>
<td></td>
</tr>
</tbody>
</table>

Physical Address

Physical Page # | Offset in Page
---|---

Oregon Systems Infrastructure Research and Information Security (OSIRIS) Lab
Logical to Physical Memory

```
logical memory

page 0
page 1
page 2
page 3

0 1
1 4
2 3
3 7

page table

frame number
0 0
1 1
2 page 0
3 page 2
4 page 1
5
6
7 page 3

physical memory
```
Paging Example

32-byte memory and 4-byte pages
Free Frames

Before allocation

After allocation
Page Table Entry Format

• Physical page Number

• Valid/Invalid bit.

• Protection bits (Read / Write / Execute)

• Modified bit (set on a write/store to a page)
  ‣ Useful for page write-backs on a page-replacement.

• Referenced bit (set on each read/write to a page).
  ‣ Will look at how this is used a little later.

• Disable caching.
  ‣ Useful for I/O devices that are memory-mapped.
Valid (v)/Invalid (i) Bit in Page Table
Issues to Address

• Size of page-tables would be very large!
• For example, 32-bit virtual address spaces (4 GB) and a 4 KB page size would have ~1 M pages/entries in page-tables.
• What about 64-bit virtual address spaces?!
• A process does not access all of its address space at once! Exploit this locality factor.
• Use multi-level page-tables. Equivalent to paging the page-tables.
• Inverted page-tables.
Example: 2-level Page Table

32-bit virtual address

Page dir | Page table | Page offset
---|---|---
10 | 10 | 12

Page Tables

32-bit physical address

Page

Directory

Code

Data

Stack
• Linux: 4-level page table (necessary for 64-bit arch)
• Virtual address space is sparse and widely scattered
  ‣ stack at top, heap at bottom, dynamic libraries in the middle
• Page-table lookup needs to be done on every memory-reference for both code & data!
  ‣ Can be very expensive if this is done by software.

• Usually done by a hardware unit called the MMU (Memory-Management Unit).
  ‣ Located between CPUs and caches.
Role of the MMU

• Given a Virtual Address, index in the page-table to get the mapping.

• Check if the valid bit in the mapping is set, and if so put out the physical address on the bus and let hardware do the rest.

• If it is not set, you need to fetch the data from the disk (swap-space).
  ‣ We do not wish to do this in hardware!
Requirements

• Address translation/mapping must be very fast! Why?
  ‣ Because it is done on every instruction fetch, memory reference instruction (loads/stores). Hence, it is in the critical path.

• Previous mechanisms access memory to lookup the page-tables. Hence it is very slow!
  ‣ CPU-Memory gap is ever widening!

• Solution: Exploit the locality of accesses.
• Translation Look-Aside Buffer
• Typically programs access a small number of pages very frequently.
• Temporal and spatial locality are indicators of future program accesses.
• Temporal locality
  ‣ Likelihood of same data being re-accessed in the near future.
• Spatial locality
  ‣ Likelihood of neighboring locations being accessed in the near future.
• TLBs act like a cache for page-table.
• Magic behind the TLB: associative memory
  ‣ also called content-addressable memory (CAM)
• Implements lookup table functionality in a single clock cycle using dedicated comparison circuitry
  ‣ How does this differ from regular memory?
Address Translation w. TLB
TLB Cache

• Typically, TLB is a cache for a few (8/16/32) Page-table entries.

• Given a virtual address, check this cache to see if the mapping is present, and if so we return the physical address.

• If not present, the MMU attempts the usual address translation.

• TLB is often designed as a fully-associative cache.

• TLB entry has
  ‣ Used/unused bits, virtual page number, Modified bit, Protection bits, physical page number.
Address Translation Steps

• Virtual address is passed from the CPU to the MMU (on instruction fetch or load/store instruction).

• Parallel search of the TLB in hardware to determine if mapping is available.

• If present, return the physical address.

• Else MMU detects miss, and looks up the page table as usual (*page walk:* this is not a page fault)

• If page table lookup succeeds, return physical address and insert mapping into TLB evicting another entry.

• Else it is a *page fault.*
Hit Ratio

• Fraction of references that can be satisfied by TLB is called *hit ratio* \((h)\).

• For example, if it takes 100 ns to access page-table entry and 20 ns to access TLB,
  ‣ average lookup time = \(20 \times h + 100 \times (1 - h)\).
• Page-tables could become quite large!

• Above mechanisms pages the page-tables and uses TLBs to take advantage of locality.

• Inverted page-tables organize the translation mechanism around physical memory.

• Each entry associates a physical page with the virtual page stored there!
  ‣ Size of Inverted Page-table = Physical Memory size / Page size.
Virtual Address

Virtual Page # | Offset in Page

Inverted Page-table

If VP# is present, then PP# is available.

Page-table (can be on disk)

No entry for VP# in the table

IVT implemented in
a) Software using hashing.
b) Hardware using associative memory

Usual paging mechanism
Segmentation

- Can be used as a programming convenience
- Several times you have different segments (code, data, stack, heap), or even within data/heap you may want to define different regions.
- You can then address these segments/regions using base + offset.
- You can also define different protection permissions for each segment.
- However, segmentation by itself has all those original problems (contiguous allocation, fitting in memory, etc.)
• Define segments in the virtual address space.

• In programs, you refer to an address using [Segment Ptr + Offset in Segment].
  ‣ E.g Intel family

• Segment Ptr leads you to a page table, which you then index using the offset in segment.

• This gives you physical frame #. You then use page offset to index this page.

• Virtual address = (Segment #, Page #, Page Offset)
Summary

• Memory Management
  ‣ Limited physical memory resource
• Keep key process pages in memory
  ‣ Swapping (and paging, later)
• Memory allocation
  ‣ High performance
  ‣ Minimize fragmentation
Summary

• Paging
  ‣ Non-contiguous allocation
  ‣ Pages and frames
  ‣ Fragmentation
  ‣ Page tables
  ‣ Hardware support
  ‣ Plus, Segmentation