CPS 104
Computer Organization and Programming
Lecture 20: Virtual Memory

Nov. 10, 1999
Dietolf (Dee) Ramm
http://www.cs.duke.edu/~dr/cps104.html
Outline of Today’s Lecture

- Virtual Memory.
  - Paged virtual memory.
  - Virtual to Physical translation: The page table.
  - Fragmentation.
  - Page replacement policies.
  - Reducing the Virtual to Physical address translation time.
    - The TLB
    - Parallel access to the TLB and Cache
  - Memory Protection
  - Putting it all together: The SPARC-20 memory system
Virtual Memory
Memory System Management
Problems

- Different Programs have different memory requirements.
  - How to manage program placement?

- Different machines have different amount of memory.
  - How to run the same program on many different machines?

- At any given time each machine runs a different set of programs.
  - How to fit the program mix into memory? Reclaiming unused memory? Moving code around?

- The amount of memory consumed by each program is dynamic (changes over time)
  - How to effect changes in memory location: add or subtract space?

- Program bugs can cause a program to generate reads and writes outside the program address space.
  - How to protect one program from another?
Virtual Memory

Provides *illusion* of very large memory
– Sum of the memory of many jobs greater than physical memory
– Address space of each job larger than physical memory

Allows available (fast and expensive) physical memory to be well utilized.

Simplifies memory management: code and data movement, *protection*, ... (*main reason today*)

Exploits memory hierarchy to keep average access time low.

Involves at least two storage levels: *main* and *secondary*

*Virtual Address* -- address used by the programmer

*Virtual Address Space* -- collection of such addresses

*Memory Address* -- address in physical memory
also known as “*physical address*” or “*real address*”
Review: A Simple Program’s Memory Layout

- What are the possible addresses generated by the program?
- How big is our DRAM?
- Is there more than one program running?
- If so, how do we allocate memory to each?
Paged Virtual Memory: Main Idea

- Divide memory (virtual and physical) into fixed size blocks (Pages, Frames).
  - Pages in Virtual space.
  - Frames in Physical space.

- Make page size a power of 2: \( \text{page size} = 2^k \)

- All pages in the virtual address space are contiguous.

- Pages can be mapped into physical Frames in any order.

- Some of the pages are in main memory (DRAM), some of the pages are on secondary memory (disk).

- All programs are written using Virtual Memory Address Space.

- The hardware does on-the-fly translation between virtual and physical address spaces.

- Use a Page Table to translate between Virtual and Physical addresses.
Basic Issues in Virtual Memory System Design

- size of information blocks (pages) that are transferred from secondary to main storage (M)

- block of information brought into M, and M is full, then some region of M must be released to make room for the new block --> *replacement policy*

- which region of M is to hold the new block --> *placement policy*

- missing item fetched from secondary memory only on the occurrence of a fault --> *demand load policy*

Paging Organization

**virtual and physical** address space partitioned into blocks of equal size
Virtual and Physical Memories

Virtual Memory

- Page-0
- Page-1
- Page-2
- Page-3
- ...
- Page N-2
- Page N-1
- Page N

Physical Memory

- Frame-0
- Frame-1
- Frame-2
- Frame-3
- Frame-4
- Frame-5

Disk

Page -2
Virtual to Physical Address translation

Page size: 4K

Virtual Address

Virtual Page Number  Page offset

Page Table

Physical Frame Number  Page offset

Physical Address
Address Map

\[ V = \{0, 1, \ldots, n - 1\} \text{ virtual address space} \]
\[ M = \{0, 1, \ldots, m - 1\} \text{ physical address space} \]

\[ n > m \]

MAP: \( V \rightarrow M \cup \{\emptyset\} \) address mapping function

\[ MAP(a) = a' \text{ if data at virtual address } a \text{ is present in physical address } a' \text{ and } a' \text{ in } M \]

\[ = \emptyset \text{ if data at virtual address } a \text{ is not present in } M \]
The page table

Virtual Address

Virtual Page Number

Page offset

Virtual Address

Page offset

Physical Address

Physical Frame Number

Page offset

Page table reg

V D AC Frame number

Access Control

Dirty bit

Valid bit

Physical Address

Valid bit

Dirty bit
Address Mapping Algorithm

If $V = 1$
then page is in main memory at frame address stored in table
else address located page in secondary memory

Access Control
$R = \text{Read-only, } R/W = \text{read/write, } X = \text{execute only}$

If kind of access not compatible with specified access rights,
then $\textit{protection\_violation\_fault}$

If valid bit not set then $\textit{page fault}$

$\textit{Protection Fault:}$ access control violation; causes trap to hardware,
or software fault handler

$\textit{Page Fault:}$ page not resident in physical memory, also causes a trap;
usually accompanied by a $\textit{context switch:}$ current process
suspended while page is fetched from secondary storage

e.g., VAX 11/780
each process sees a 4 gigabyte ($2^{32}$ bytes) virtual address space
1/2 for user regions, 1/2 for a system wide name space shared
by all processes.

page size is 512 bytes (too small)
**Optimal Page Size**

Choose page size that minimizes fragmentation (partial use of pages)

- **large page size** => internal fragmentation more severe
  - BUT decreases the # of pages / name space => **smaller page tables**

In general, the trend is towards **larger page sizes** because

- Memories get larger as the price of RAM drops.
- The gap between processor speed and disk speed grow wider.
- Programmers demand larger virtual address spaces (larger program)
- Smaller page table.

Most machines at 4K-8K byte pages today, with page sizes likely to increase
**Fragmentation**

*Page Table Fragmentation* occurs when page tables become very large because of large virtual address spaces; linearly mapped page tables could take up sizable chunk of memory.

EX: VAX Architecture (late 1970s)

<table>
<thead>
<tr>
<th>XX</th>
<th>Page Number</th>
<th>Disp</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>P0 region of user process</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>P1 region of user process</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>system name space</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: this implies that page table could require up to $2^{21}$ entries, each on the order of 4 bytes long (8 M Bytes)

Alternatives to linear page table:

1. Hardware associative mapping: requires one entry per page frame ($O(|M|)$) rather than per virtual page ($O(|N|)$)
2. "Software" approach based on a hash table (inverted page table)

```
page# disp
```

```
Present Access Page# Phy Addr
```

Page Table

associative or hashed lookup on the page number field
Page Replacement Algorithms

Just like cache block replacement!

*Least Recently Used:*
-- selects the least recently used page for replacement

-- requires knowledge about past references, more difficult to implement (thread through page table entries from most recently referenced to least recently referenced; when a page is referenced it is placed at the head of the list; the end of the list is the page to replace)

-- good performance, recognizes principle of locality
Page Replacement (Continued)

Not Recently Used:
Associated with each page is a reference flag such that
ref flag = 1 if the page has been referenced in recent past
= 0 otherwise

-- if replacement is necessary, choose any page frame such that its
reference bit is 0. This is a page that has not been referenced in the
recent past

-- clock implementation of NRU:

<table>
<thead>
<tr>
<th>ref bit</th>
<th>page table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

last replaced pointer (lrp)
if replacement is to take place, advance lrp to next entry (mod
table size) until one with a 0 bit
is found; this is the target for
replacement; As a side effect,
all examined PTE's have their
reference bits set to zero.

An optimization is to search for a page that is both
not recently referenced AND not dirty. Why?
Virtual Address and a Cache

It takes an extra memory access to translate VA to PA

This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible
Translation Lookaside Buffer (TLB)

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer* or *TLB*.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access</th>
</tr>
</thead>
</table>

TLB access time comparable to cache access time (much less than main memory access time).

Typical TLB is 64-256 entries fully associative cache with random replacement.
Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped.

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Many mid-range machines use small n-way set associative organizations.

Translation with a TLB
TLB Design

- Must be fast, not increase critical path
- Must achieve high hit ratio
- Generally small highly associative (64-128 entries FA cache)
- Mapping change
  - page added/removed from physical memory
  - processor must invalidate the TLB entry (special instructions)
- PTE is per process entity
  - Multiple processes with same virtual addresses
  - Context Switches?
- Flush TLB
- Add ASID (PID)
  - part of processor state, must be set on context switch
Hardware Managed TLBs

- Hardware Handles TLB miss
- Dictates page table organization
- Complicated state machine to “walk page table”
- Exception only if access violation
Software Managed TLBs

- Software Handles TLB miss
  - OS reads translations from Page Table and puts them in TLB
  - special instructions

- Flexible page table organization

- Simple Hardware to detect Hit or Miss

- Exception if TLB miss or access violation
Choosing a Page Size

What if page is too small?

- Too many misses
- BIG page tables

What if page is too big?

- Fragmentation
  - don’t use all of the page, but can’t use that DRAM for other pages
  - want to minimize fragmentation (get good utilization of physical memory)
- Smaller page tables

- Trend it toward larger pages
  - increasing gap between CPU/DRAM/DISK
Reducing Translation Time

Machines with TLBs go one step further to reduce # cycles/cache access

They overlap the cache access with the TLB access

Works because high order bits of the VA are used to look in the TLB while low order bits are used as index into cache
Overlapped Cache & TLB Access

IF cache hit AND (cache tag = PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag = PA)] and TLB hit THEN
   access memory with the PA from the TLB
ELSE do standard VA translation

TLB

Cache

assoc lookup

index

PA Hit/
Miss

20

12

page #
disp

PA Data Hit/
Miss
Problems With Overlapped TLB Access

Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation.

This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache.

Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:

![Diagram of cache and memory]

This bit is changed by VA translation, but is needed for cache lookup.

Solutions:
- go to 8K byte page sizes;
- go to 2 way set associative cache; or
More on Selecting a Page Size

- **Reasons for larger page size**
  - Page table size is inversely proportional to the page size.
  - Faster cache hit time when cache "page size; bigger page => bigger cache" (no aliasing problem).
  - Transferring larger pages to or from secondary storage, is more efficient (Higher bandwidth)
  - The number of TLB entries is restricted by clock cycle time, so a larger page size reduces TLB misses.

- **Reasons for a smaller page size**
  - don’t waste storage; data must be contiguous within page.
  - quicker process start for small processes(?)

- **Hybrid solution**: multiple page sizes:

  Alpha, UltraSPARC: 8KB, 64KB, 512 KB, 4 MB pages
Memory Protection

- Paging Virtual memory provides protection by:
  - Each process (user or OS) has different virtual memory space.
  - The OS maintain the page tables for all processes.
  - A reference outside the process allocated space cause an exception that lets the OS decide what to do.
  - Memory sharing between processes is done via different Virtual spaces but common physical frames.
Putting it together: The SparcStation 20:

- The SparcStation 20 has the following memory system.
- Caches: Two level-1 caches: I-cache and D-cache

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Instruction cache</th>
<th>Data cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization</td>
<td>20Kbyte 5-way SA</td>
<td>16KB 4-way SA</td>
</tr>
<tr>
<td>Page size</td>
<td>4K bytes</td>
<td>4K bytes</td>
</tr>
<tr>
<td>Line size</td>
<td>8 bytes</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Replacement</td>
<td>Pseudo LRU</td>
<td>Pseudo LRU</td>
</tr>
</tbody>
</table>

- TLB: 64 entry Fully Associative TLB, Random replacement
- External Level-2 Cache: 1M-byte, Direct Map, 128 byte blocks, 32-byte sub-blocks.
SparcStation 20 Data Access

Virtual Address

Physical Address

To Memory

Data Cache

Data Select

To CPU

TLB

To Memory
SparcStation 20: Instruction Memory

Instruction Address 2

Physical Address 36

To Memory

Instruction Cache 4 bytes

1K

Instruction Select

Tag data tag tag tag tag tag tag

Tag 0 tag 1 tag 2 tag 3 tag 4 tag 5 tag 6

To CPU (instruction register)

TLB

Tag 0 tag 1 tag 2 tag 3 tag 4 tag 5 tag 6

To Memory