Review: The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
  - Example: 90% of time in 10% of the code
Review: Levels of the Memory Hierarchy

- **Registers**
  - Capacity: 100s Bytes
  - Access Time: <10s ns
  - Cost: $.01-.001/bit

- **Cache**
  - Capacity: K Bytes
  - Access Time: 10-100 ns
  - Cost: $.01-.001/bit

- **Main Memory**
  - Capacity: M Bytes
  - Access Time: 100ns-1us
  - Cost: $.01-.001/bit

- **Disk**
  - Capacity: G Bytes
  - Access Time: ms
  - Cost: 10^3-10 cents

- **Tape**
  - Capacity: infinite
  - Access Time: sec-min
  - Cost: 10^10

Outline of Today’s Lecture

- Recap of Memory Hierarchy
- Virtual Memory
- Page Tables and TLB
- Protection
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 very well utilized

Simplifies memory management

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 of word in physical memory also known as “physical address” or “real address”

---

Basic Issues in VM System Design

- size of information blocks that are transferred from secondary to main storage

- 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 --> *fetch/load policy*

**Paging Organization**

Virtual and physical address space partitioned into blocks of equal size
Address Map

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

MAP: \( V \rightarrow M \cup \{0\} \) address mapping function

\[ \text{MAP}(a) = a' \quad \text{if data at virtual address } a \text{ is present in physical address } a' \text{ and } a' \text{ in } M \]
\[ = 0 \quad \text{if data at virtual address } a \text{ is not present in } M \]

Processor

Name Space \( V \)

Addr Trans Mechanism

fault handler

Main Memory

Secondary Memory

physical address

OS performs this transfer

missing item fault

Paging Organization

<table>
<thead>
<tr>
<th>P.A.</th>
<th>frame 0</th>
<th>1K</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1024</td>
<td>1K</td>
</tr>
<tr>
<td>7168</td>
<td>7</td>
<td>1K</td>
</tr>
</tbody>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>Addr Trans MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>31744</td>
</tr>
</tbody>
</table>

Virtual Memory

Address Mapping

<table>
<thead>
<tr>
<th>VA</th>
<th>page no.</th>
<th>disp</th>
</tr>
</thead>
</table>

Page Table

Base Reg

index into page table

Page Table

V Access Rights PA

table located in physical memory

physical memory address

actually, concatenation is more likely

unit of mapping

also unit of transfer from virtual to physical memory
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 Rights
\[ R = \text{Read-only}, \; R/W = \text{read/write}, \; X = \text{execute only} \]

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 rights violation; causes trap to hardware,
microcode, or software fault handler

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

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

\textit{ASIDE}: Why access cache with PA at all? VA caches have a problem!
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

ASIDE: Why access cache with PA at all? VA caches have a problem!

synonym problem:

two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address!

for update: must update all cache entries with same physical address or memory becomes inconsistent

determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits

TLBs

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, though shorter than, cache access time (still much less than main memory access time)
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. Most mid-range machines use small n-way set associative organizations.

Translation with a TLB

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

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:

Solutions:
go to 8K byte page sizes
go to 2 way set associative cache (would allow you to continue to
use a 10 bit index)
Fragmentation & Relocation

*Fragmentation* is when areas of memory space become unavailable for some reason

*Relocation*: move program or data to a new region of the address space (possibly fixing all the pointers)

---

**External Fragmentation**: Space left between blocks.

---

**Internal Fragmentation**: program is not an integral # of pages, part of the last page frame is "wasted" (obviously less of an issue as physical memories get larger)

---

Optimal Page Size

Choose page that minimizes fragmentation

large page size => internal fragmentation more severe
BUT increase in the # of pages / name space => larger 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 desire larger virtual address spaces

Most machines at 4K-64K byte pages today, with page sizes likely to increase
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 thru 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
\[
\text{ref flag} = 1 \quad \text{if the page has been referenced in recent past} \\
= 0 \quad \text{otherwise}
\]

-- 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:

```
g 0  
g 0  
g 0  
0 0  
```

An optimization is to search for a page that is both not recently referenced and not dirty.

Demand Paging and Prefetching Pages

*Fetch Policy*
when is the page brought into memory?
if pages are loaded solely in response to page faults, then the policy is *demand paging*

An alternative is *prefetching:*
- anticipate future references and load such pages before their actual use
  - reduces page transfer overhead
  - removes pages already in page frames, which could adversely affect the page fault rate
  - predicting future references usually difficult

Most systems implement demand paging without prepaging
(One way to obtain effect of prefetching behavior is increasing the page size
Summary

- Virtual memory: a mechanism to provide much larger memory than physically available memory in the system.
- Placement, replacement, and other policies can have significant impact on performance.
- Interaction of Virtual memory with physical memory hierarchy is complex and addresses translation mechanisms must be designed carefully for good performance.