Virtual Memory

Finally! A lecture on something I understand – PAGE FAULTS!

I wish we were still doing NAND gates…

Study Chapter 5.4-5.5
You can never be too rich, too good looking, or have too much memory!

Now that we know how to FAKE a FAST memory, we’ll turn our attention to FAKING a LARGE memory.
Top 10 Reasons for a BIG Address Space

1. Programming CONVENIENCE
   • create regions of memory with different semantics: read-only, shared, etc.
   • avoid annoying bookkeeping

2. Usage UNCERTAINTY
   • provides for run-time expansion of stack and heap

3. Isolating ISA from IMPLEMENTATION
   • details of HW configuration shouldn’t enter into SW design
   • HOW MUCH and WHERE memory is

4. To support lazy programmers.

5. To emulate a Turning Machine’s tape.

6. So we never need to run garbage collection

7. Performing multiplies via table lookup

8. Generates good Comp 411 final problems.

9. To provide unique addresses for every internet host.

10. Keeping TI’s memory division in business.
Lessons from History...

There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management.

Gordon Bell and Bill Strecker speaking about the PDP-11 in 1976

A partial list of successful machines that eventually starved to death for lack of address bits includes the PDP 8, PDP 10, PDP 11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, Motorola 6800, AMI 6502, Zilog Z80, Cray-1, and Cray X-MP.

Hennessy & Patterson

Why? Address size determines minimum width of anything that can hold an address: PC, registers, memory words, HW for address arithmetic (branches/jumps, loads/stores). When you run out of address space it’s time for a new ISA!
Squandering Address Space

STACK: How much to reserve? (consider RECURSION!)

HEAP: \( N \) variable-size data records...
Bound \( N \)? Bound Size?

OBSERVATIONS:
• Can’t BOUND each usage... without compromising use.
• Actual use is SPARSE
• Working set even MORE sparse

CODE, large monolithic programs (eg, Office, Firefox)....
• only small portions might be used
• add-ins and plug-ins
• shared libraries/DLLs
•••
Extending the Memory Hierarchy

So far, we’ve used SMALL fast memory + BIG slow memory to fake a BIG FAST memory (caching).

Can we combine RAM and DISK to fake DISK sized at near RAM speeds?

VIRTUAL MEMORY

- use of RAM as cache to much larger storage pool, on slower devices
- TRANSPARENCY - VM locations "look" the same to program whether on DISK or in RAM.
- ISOLATION of actual RAM size from software.
Virtual Memory

ILLUSION: Huge memory
\(2^{32}\) (4G) bytes? \(2^{64}\) (18E) bytes?)

ACTIVE USAGE: small fraction
\(2^{24}\) bytes?)

Actual HARDWARE:
- \(2^{31}\) (2G) bytes of RAM
- \(2^{39}\) (500G) bytes of DISK...
  ... maybe more, maybe less!

ELEMENTS OF DECEIT:
- Partition memory into manageable chunks-- "Pages"
  (4K-8K-16K-64K)
- MAP a few to RAM, assign others to DISK
- Keep "HOT" pages in RAM.
Simple Page Map Design

FUNCTION: Given Virtual Address,

- Map to PHYSICAL address
  OR
- Cause PAGE FAULT allowing page replacement

Why use HIGH address bits to index pages?
... LOCALITY.
Keeps related data on same page.

Why use LOW address bits to index cache lines?
... LOCALITY.
Keeps related data from competing for same cache lines.

"Page Map"
A special memory that holds Virtual-to-Physical Mappings
Virtual Memory vs. Cache

**CACHE:**
- Relatively short blocks (16-64 bytes)
- Few lines: scarce resource
- Miss time: 3x-20x hit time

**VIRTUAL MEMORY:**
- Disk: long latency, fast xfer
  → Miss time: $\sim10^5$ x hit time
  → Write-back essential!
  → Large pages in RAM
- Lots of lines: one for each page
- Tags in page map,
  data in physical memory
Virtual Memory: A H/W view

Pagemap Characteristics:

• One entry per **virtual** page!

• RESIDENT bit = 1 for pages stored in RAM, or 0 for non-resident (disk or unallocated). Page fault when R = 0.

• Contains PHYSICAL page number (PPN) of each resident page

• DIRTY bit says we’ve changed this page since loading it from disk (and therefore need to write it back to disk when it’s replaced)
Virtual Memory: A S/W view

Problem: Translate VIRTUAL ADDRESS to PHYSICAL ADDRESS

```c
int VtoP(int VPageNo, int PageOffset) {
    if (R[VPageNo] == 0)
        PageFault(VPageNo);
    return (PPN[VPageNo] << p) | PageOffset;
}

/* Handle a missing page... */
void PageFault(int VPageNo) {
    int i;

    i = SelectLRUPage();
    if (D[i] == 1)
        WritePage(DiskAdr[i],PPN[i]);
    R[i] = 0;

    PPN[VPageNo] = PPN[i];
    ReadPage(DiskAdr[VPageNo],PPN[i]);
    R[VPageNo] = 1;
    D[VPageNo] = 0;
}
```
The HW/SW Balance

IDEA:

- devote HARDWARE to high-traffic, performance-critical path
- use (slow, cheap) SOFTWARE to handle exceptional cases

```
int VtoP(int VPageNo, int PO) {
  if (R[VPageNo] == 0) PageFault(VPageNo);
  return (PPN[VPageNo] << p) | PO;
}

/* Handle a missing page... */
void PageFault(int VPageNo) {
  int i = SelectLRUPage();
  if (D[i] == 1) WritePage(DiskAdr[i], PPN[i]);
  R[i] = 0;
  PA[VPageNo] = PPN[i];
  ReadPage(DiskAdr(VPageNo), PPN[i]);
  R[VPageNo] = 1;
  D[VPageNo] = 0;
}
```

HARDWARE performs address translation, detects page faults:

- running program is interrupted ("suspended");
- PageFault(...) is called;
- On return from PageFault; running program can continue
Page Map Arithmetic

(v + p) bits in virtual address
(m + p) bits in physical address
$2^v$ number of VIRTUAL pages
$2^m$ number of PHYSICAL pages
$2^p$ bytes per physical page
$2^{v+p}$ bytes in virtual memory
$2^{m+p}$ bytes in physical memory

Typical page size: 4K – 128K bytes
Typical (v+p): 32 or 64 bits
Typical (m+p): 27 – 33 bits
(128 MB – 8 GB)

Comp 411 – Spring 2012
4/11/2012
Example: Page Map Arithmetic

Suppose...

32-bit Virtual address

\(2^{14}\) page size (16 KB)

\(2^{28}\) RAM (256 MB)

Then:

\[
\text{# Physical Pages} = \frac{2^{28}}{2^{14}} = 16384
\]

\[
\text{# Virtual Pages} = \frac{2^{32}}{2^{14}} = 2^{18}
\]

\[
\text{# Page Map Entries} = 262,144
\]

Use SRAM for page map?? OUCH!
RAM-Resident Page Maps

SMALL page maps can use dedicated RAM...
but, gets this approach gets expensive for big ones!

SOLUTION: Move page map into MAIN MEMORY:

Virtual Address

Physical Memory

PROBLEM:
Each memory reference now takes 2 accesses to physical memory!

1) Load VPN $\rightarrow$ PPN
2) Load Mem[PPN | PO]

The overhead for the pagemap is smaller than you might think. From the previous example: $4 \times 2^{18}/2^{28} = 0.4\%$
Translation Look-aside Buffer (TLB)

PROBLEM: 2x performance hit... each memory reference now takes 2 accesses!

SOLUTION: a special CACHE of recently used page map entries

IDEA:
LOCALITY in memory reference patterns → SUPER locality in references to page map

VARIATIONS:
• sparse page map storage
• paging the page map

TLB: small, usually fully-associative cache for mapping VPN→PPN
Optimizing Sparse Page Maps

For large Virtual Address spaces only a small percentage of page table entries contain Mappings. This is because some range of addresses are never used by the application. How can we save space in the pagemap?

On TLB miss:

- look up VPN in “sparse” data structure (e.g., a list of VPN-PPN pairs)
- only have entries for ALLOCATED pages
- use hashing to speed up the search
- allocate new entries “on demand”
- time penalty? LOW if TLB hit rate is high…
Multilevel Page Maps

Given a HUGE virtual memory, the cost of storing all of the page map entries in RAM may STILL be too expensive...

SOLUTION: A hierarchical page map... take advantage of the observation that while the virtual memory address space is large, it is generally sparsely populated with clusters of pages.

Consider a machine with a 32-bit virtual address space and 64 MB (26-bit) of physical memory that uses 4 KB pages.

Assuming 4 byte page-table entries, a single-level page map requires 4MB (>6% of the available memory). Of these, more than 98% will reference non-resident pages (Why?).

A 2-level look-up increases the size of the worse-case page table slightly. However, if a first level entry has its non-resident bit set it saves large amounts of memory.

A clever designer will notice that if the 2nd level tables are “page-sized” they too can be “paged out” (stored on disk).

Doesn’t that mean we now have to do 3 accesses to get what we want?
Example: Mapping VAs to PAs

Suppose
- virtual memory of $2^{32}$ (4G) bytes
- physical memory of $2^{30}$ (1G) bytes
- page size is $2^{14}$ (16 K) bytes

<table>
<thead>
<tr>
<th>VPN</th>
<th>R</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

1. How many pages can be stored in physical memory at once?
   $$2^{30-14} = 2^{16} = 64K$$

2. How many entries are there in the page table?
   $$2^{32-14} = 2^{18} = 256K$$

3. How many bits are necessary per entry in the page table?
   (Assume each entry has PPN, resident bit, dirty bit)
   $$16 \ (PPN) + 2 = 18$$

4. How many pages does the page table require?
   $$\frac{4 \cdot 2^{18}}{2^{14}} = 2^6 = 64$$

5. A portion of the page table is given to the left. What is the physical address for virtual address $0x00004110$?
**Contexts**

A *context* is a complete set of mappings from VIRTUAL to PHYSICAL locations, as dictated by the full contents of the page map:

We might like to support multiple VIRTUAL to PHYSICAL Mappings and, thus, multiple Contexts.

Several programs may be simultaneously loaded into main memory, each in its separate context:

“Context Switch”: Reload the page map!

You end up with pages from different applications simultaneously in memory.
Contexts: A Sneak Preview

1. TIMESHARING among several programs --
   - Separate context for each program
   - OS loads appropriate context into pagemap when switching among pgms

2. Separate context for OS “Kernel” (eg, interrupt handlers)...
   - “Kernel” vs “User” contexts
   - Switch to Kernel context on interrupt;
   - Switch back on interrupt return.

HARDWARE SUPPORT: 2 HW pagemaps

Every application can be written as if it has access to all of memory, without considering where other applications reside.

First Glimpse at a VIRTUAL MACHINE
Example: UltraSPARC II MMU

Huge 64-bit address space (only 44-bits implemented)

Virtual Address

\[ \begin{array}{ll}
C\# & 51 \quad 13 \\
C\# & 48 \quad 16 \\
C\# & 45 \quad 19 \\
C\# & 42 \quad 22 \\
\end{array} \]

C\# = context

Physical Address

41 (2200 GB)

4 page sizes: 8KB, 64KB, 512KB, 4MB

TLB miss: SW refills TLB from TSB cache (HW helps compute address)

TSB – Translation Storage Buffer

TSB (direct-mapped)
Using Caches with Virtual Memory

Virtual Cache
Tags match virtual addresses

These TAGs are virtual, they represent addresses before translation.

• Problem: cache becomes invalid after context switch
• FAST: No MMU time on HIT

Physical Cache
Tags match physical addresses

These TAGs are physical, they hold addresses after translation.

• Avoids stale cache data after context switch
• SLOW: MMU time on HIT

Counter intuitively perhaps, physically addressed Caches are the trend, because they better support parallel processing.
OBSERVATION: If cache line selection is based on *unmapped* page offset bits, RAM access in a physical cache can *overlap* page map access. Tag from cache is compared with physical page number from MMU.

Want “small” cache index → go with more associativity
Summary

Virtual Memory

Makes a small PHYSICAL memory appear to be a large VIRTUAL one

Break memory into manageable chunks PAGES

Pagemap

A table for mapping Virtual-to-Physical pages
Each entry has Resident, Dirty, and Physical Page Number
Can get large if virtual address space is large
Store in main memory

TLB – Translation Look-aside Buffer

A pagemap cache

Contexts –

Sets of virtual-to-physical mapping that allow pages from multiple applications to be in physical memory simultaneously (even if they have the same virtual addresses)