Virtual Memory Basics
Memory Management

• The Fifties:
  - Absolute Addresses
  - Dynamic address translation

• The Sixties:
  - Paged memory systems and TLBs
  - Atlas’ Demand paging

• Modern Virtual Memory Systems
**Types of Names for Memory Locations**

- **Machine language address**
  - as specified in machine code

- **Virtual address**
  - ISA specifies translation of machine code address into virtual address of program variable

- **Physical address**
  - operating system specifies mapping of virtual address into name for a physical memory location

Translation of machine code address into virtual address may involve a segment register.

Physical memory location => actual address signals going to DRAM chips.
Absolute Addresses

EDSAC, early 50’s

effective address = physical memory address

Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)

Addresses in a program depended upon where the program was to be loaded in memory

But it was more convenient for programmers to write location-independent subroutines

⇒ How could location independence be achieved?

Led to the development of loaders and linkers to statically relocate and link Programs.
**Dynamic Address Translation**

**Motivation:**
In the early machines, I/O operations were slow and each word transferred involved the CPU. Higher throughput if CPU and I/O of 2 or more programs were overlapped. *How?*

⇒ **multiprogramming**

**Location independent programs:**
Programming and storage management ease

⇒ need for a **base register**

**Protection:**
Independent programs should not affect each other inadvertently

⇒ need for a **bound register**
Simple Base and Bound Translation

Base and bounds registers only visible/accessible when processor running in *kernel* (a.k.a *supervisor*) *mode*
Separate Areas for Program and Data

What is an advantage of this separation?
Used today on Cray vector supercomputers

Permits sharing of program segments.
As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.

Called Burping the memory.
Processor generated address can be interpreted as a pair \(<\text{page number}, \text{offset}>\)

A page table contains the physical address of the base of each page

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Address Space of User-1

Page Table of User-1

What requirement does fixed-length pages plus indirection through page table relax?

Relaxes the contiguous allocation requirement.
Private Address Space per User

- Each user has a page table
- Page table contains an entry for each user page

OS ensures that the page tables are disjoint.
Where Should Page Tables Reside?

Space required by the page tables is proportional to the address space, number of users, ...

⇒ Space requirement is large
too expensive to keep in registers

Special registers just for the current user:
- What disadvantages does this have?
  may not be feasible for large page tables

Main memory:
- needs one reference to retrieve the page base address and another to access the data word
⇒ doubles number of memory references!

Affects context-switching overhead, and needs new management instructions.
Page Tables in Physical Memory

- Page Table, User 1
- Page Table, User 2

User 1

User 2
A Problem in Early Sixties

There were many applications whose data could not fit in the main memory, e.g., Payroll

*Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory*

Programmers moved the data back and forth from the secondary store by *overlapping* it repeatedly on the primary store

*tricky programming!*
Manual Overlays

Assuming an instruction can address all the storage on the drum

Method 1 - programmer keeps track of addresses in the main memory and initiates an I/O transfer when required

Method 2 - automatic initiation of I/O transfers by software address translation

Brooker’s interpretive coding, 1960

Method 1 problem?
Method 2 problem?

British Firm Ferranti, did Mercury and then Atlas
Method 1 too difficult for users
Method 2 too slow.
**Demand Paging in Atlas (1962)**

“A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor.”

*Tom Kilburn*

Primary memory as a **cache** for secondary memory

<table>
<thead>
<tr>
<th>Central Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primary</strong></td>
</tr>
<tr>
<td>32 Pages</td>
</tr>
<tr>
<td>512 words/page</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Secondary (Drum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32x6 pages</td>
</tr>
</tbody>
</table>

User sees 32 x 6 x 512 words of storage

---

Single-level Store
Hardware Organization of Atlas

Effective Address

- Initial Address Decode
- Fixed (ROM)
  - 16 pages
  - 0.4 ~ 1 µsec
- Subsidiary
  - 2 pages
  - 1.4 µsec
- Main
  - 32 pages
  - 1.4 µsec
- Drum (4)
  - 192 pages
- Tape
  - 8 decks
  - 88 µsec/word

Compare the effective page address against all 32 PARs

- match ⇒ normal access
- no match ⇒ page fault

the state of the partially executed instruction was saved

Atlas Autocode example here.
Atlas Demand Paging Scheme

On a page fault:

- input transfer into a free page is initiated
- the Page Address Register (PAR) is updated
- if no free page is left, a page is selected to be replaced (based on usage)
- the replaced page is written on the drum - to minimize drum latency effect, the first empty page on the drum was selected
- the page table is updated to point to the new location of the page on the drum

This was called the Supervisor program, which clearly foreshadowed the operating system.
Caching vs. Demand Paging

**Caching**
- cache entry
- cache block (~32 bytes)
- cache miss (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~10 cycles)
- a miss is handled in **hardware**

**Demand paging**
- page-frame
- page (~4K bytes)
- page miss (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled mostly in **software**
Modern Virtual Memory Systems
Illusion of a large, private, uniform store

Protection & Privacy
- several users, each with their private address space and one or more shared address spaces
  page table = name space

Demand Paging
- ability to run a program larger than the primary memory

⇒ What is another big benefit?

The price is address translation on each memory reference

Portability on machines with different memory configurations.
Every instruction and data access needs address translation and protection checks.

A good VM design needs to be fast (~ one cycle) and space efficient.
Linear Page Table

Page Table Entry (PTE) contains:
- PPN (physical page number) of memory-resident page,
- DPN (disk page number) of page swapped to disk, or
- non-existent page
- Status bits for protection and usage

OS changes page table base register to point to base of page table for active user process

Page Table

VPN
Offset

Data Pages

Data word

Offset

PT Base Register
Size of Linear Page Table

With 32-bit addresses, 4-KB pages, and 4-byte PTEs:
⇒ $2^{20}$ PTEs, i.e., 4 MB page table per user
⇒ 4 GB of swap needed to back up full virtual address space

Larger pages?
• more internal fragmentation (don't use all memory in page)
• larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???
• Even 1MB pages would require $2^{44}$ 8-byte PTEs (35 TB!)

What is the “saving grace”?

Virtual address space is large but only a small fraction of the pages are populated. So we can use a sparse representation of the table.
Hierarchical Page Table

Virtual Address

31 22 21 12 11 0
p1  p2  offset

10-bit L1 index 10-bit L2 index

Root of the Current Page Table
(Processor Register)

Level-1 Page Table

p1

Level-2 Page Tables

p2

offset

page in primary memory
page in secondary memory
PTE of a nonexistent page

Data Pages

Virtual Address

31 22 21 12 11 0
p1  p2  offset

10-bit L1 index 10-bit L2 index

Root of the Current Page Table
(Processor Register)

Level-1 Page Table

p1

Level-2 Page Tables

p2

offset

page in primary memory
page in secondary memory
PTE of a nonexistent page

Data Pages

Page 23
Address translation is very expensive!
In a two-level page table, each reference becomes
the best case is _____?
the worst case is _______?

Solution: *Cache translations in TLB*
TLB hit ⇒ *Single Cycle Translation*
TLB miss ⇒ *Page Table Walk to refill*

3 memory references
2 page faults (disk accesses) + ..
**TLB Designs**

Typically 32-128 entries
Usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages \(\Rightarrow\) more likely that two entries conflict
  - Sometimes larger TLBs are 4-8 way set-associative

Random or FIFO replacement policy
Typically only one page mapping per entry
No process information in TLB \(\Rightarrow\) ?

*TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB*

Example: 64 TLB entries, 4KB pages, one page per entry
TLB Reach = ______________________________?
Handling A TLB Miss

Software (MIPS, Alpha)
TLB miss causes an exception and the operating system walks the page tables and reloads TLB privileged “untranslated” addressing mode used for walk

Hardware (SPARC v8, x86, PowerPC)
A memory management unit (MMU) walks the page tables and reloads the TLB

If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction
**Hierarchical Page Table Walk: SPARC v8**

**Virtual Address**

<table>
<thead>
<tr>
<th>Index 1</th>
<th>Index 2</th>
<th>Index 3</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>23</td>
<td>17</td>
<td>11</td>
</tr>
</tbody>
</table>

- Context Table Register
- Context Table
- root ptr
- L1 Table
- PTP
- L2 Table
- PTP
- PTE
- L3 Table
- PTP

**Physical Address**

<table>
<thead>
<tr>
<th>PPN</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>11</td>
</tr>
</tbody>
</table>

*MMU does this table walk in hardware on a TLB miss*
Need to restart instruction.
Soft and hard page faults.