Design and Implementation Issues

Today
- Design issues for paging systems
- Implementation issues
- Segmentation

Next
- File systems
Considerations with page tables

Two key issues with page tables

- **Mapping must be fast**
  - Done on every memory reference, at least 1 per instruction
  - Simplest solutions
    - Page table in registers
      - Fast during execution, potentially $$$ & slow to context switch
    - Page table in memory & one register pointing to start (Page Table Base Register, PTBR)
      - Fast to context switch & cheap, but slow during execution

- **With large address spaces, page tables will be large**
  - w/ 32 bit & 4KB page → 12 bit offset, 20 bit page # ~ 1million
  - w/ 64 bit & 4KB page → $2^{12}$ (offset) + $2^{52}$ pages ~ $4.5 \times 10^{15}$!!!
Speeding things up a bit

- Simple page table 2x cost of memory lookups
  - First into page table, a second to fetch the data
  - Two-level page tables triple the cost!

- How can we make this more efficient?
  - Goal – make fetching from a virtual address about as efficient as fetching from a physical address
  - Observation – large number of references to small number of pages
  - Solution – use a hardware cache inside the CPU – Translation Lookaside Buffer (TLB)
    - Cache the virtual-to-physical translations in the hardware
    - Traditionally managed by the memory management unit (MMU)
TLBs

- TLB – Translates virtual page #s into page frame #s
  - Can be done in single machine cycle
- TLB is implemented in hardware
  - It’s a fully associative cache (parallel search)
  - Cache tags are virtual page numbers
  - Cache values are page frame numbers
    - With this + offset, MMU can calculate physical address
Managing TLBs

- Address translations mostly handled by TLB
  - >99% of translations, but there are TLB misses
  - If a miss, translation is placed into the TLB

- Hardware (memory management unit (MMU))
  - Knows where page tables are in memory
    - OS maintains them, HW access them directly

- Software loaded TLB (OS)
  - TLB miss faults to OS, OS finds page table entry & loads TLB
  - Must be fast
    - CPU ISA has instructions for TLB manipulation
    - OS gets to pick the page table format
Effective access time

- Associative Lookup = $\varepsilon$ time units
- Hit ratio - $\alpha$ - percentage of times that a page number is found in the associative registers (ratio related to TLB size)

Effective Memory Access Time (EAT)

$$EAT = \alpha \times (\varepsilon + \text{memory-access}) + (1 - \alpha) \times (\varepsilon + 2\times \text{memory-access})$$

$\alpha = 80\%$ \hspace{1cm} $\varepsilon = 20$ nsec \hspace{1cm} memory-access = 100 nsec

$$EAT = 0.8 \times (20 + 100) + 0.2 \times (20 + 2 \times 100) = 140$$ nsec
Managing TLBs

- OS must ensure TLB and page tables are consistent
  - When OS changes protection bits in an entry, it needs to invalidate the line if it is in the TLB

- What happens on a process context switch?
  - Remember, each process typically has its own page tables
  - Need to invalidate all the entries in TLB! (flush TLB)
    - A big part of why process context switches are costly
    - *Can you think of a hardware fix to this?*

- When the TLB misses, and a new process table entry is loaded, a cached entry must be evicted
  - How to choose a victim is called “TLB replacement policy”
  - Implemented in hardware, usually simple (e.g., LRU)
Hierarchical page table

- Handling large address spaces - page the page table!
- Same argument – you don’t need the full page table
- Virtual address (32-bit machine, 4KB page):
  Page # (20 bits) + Offset (12 bits)
- Since page table is paged, page number is divided:
  Page number (10 bits) + Page offset in 2\textsuperscript{nd} level (10 bits)

\[ p1 | p2 | offset \]

- \( p1 \) - index into the outer page table
- \( p2 \) - displacement within outer page

Example
Virtual address: 0x00403004

\[
\begin{array}{cccc}
 0000 & 0000 & 0100 & 0000 \ 0011 & 0000 & 0100 \\
P1 = 1 & P2 = 3 & Offset = 4
\end{array}
\]
Three-level page table in Linux

- Designed to accommodate the 64-bit Alpha
  - To adjust for a 32-bit proc. – middle directory of size 1
Inverted and hashed page tables

- Another way to save space – inverted page tables
  - Page tables are index by virtual page #, thus their size
  - Inverted page tables – one entry per page frame
    - Problem – too slow mapping!
  - Hash tables may help
  - Also, Translation Lookaside Buffer (TLB) …

Traditional page table with an entry per each $2^{52}$ pages

1GB physical memory has $2^{18}$ 4KB page frames

Indexed by virtual page

Hash table

Indexed by hash on virtual page

Virtual page | page frame
OS can pick a page size \((\text{how?})\) - small or large?

**Small**
- Less internal fragmentation
- Better fit for various data structures, code sections
- Less unused program in memory, but ...
- More I/O time, getting page from disk ... most of the time goes into seek and rotational delay!
- Larger page tables

\[
\text{Average process size } s \\
\text{Page size } p \\
\text{Page entry size } e \\
\text{overhead } = \frac{se}{p} + \frac{p}{2}
\]

Taking first derivative respect to \(p\) and equating it to zero
\[
\frac{-se}{p^2} + \frac{1}{2} = 0
\]
\[
p = \sqrt{2se}
\]

\(s = 1\text{MB}\)
\(e = 8\text{ bytes}\)
Optimal \(p = 4\text{KB}\)
Separate instruction & data spaces

- One address space – size limit
- Pioneered by PDP-11: 2 address spaces, Instruction and Data spaces
  - Double the space
  - Each with its own page table & paging algorithm
Shared pages

- In large multiprogramming systems – multiple users running same program - share pages?
- Some details
  - Not all is shareable
  - With I-space and D-space, sharing would be easier
  - What do you do if you swap one of the sharing process out?
    - Scan all page tables may not be a good idea
- Sharing data is slightly trickier than sharing code
  - Fork in Unix
  - Sharing both data and program bet/ parent and child; each with its own page table but pages marked as READ ONLY
  - Copy On Write
Virtual memory interface

- So far, transparent virtual memory
- Some control over the memory map for expert use
  - For shared memory – fast IPC
    - For distributed shared memory
      Going to disk may be slower than going to somebody else’s memory!
Implementation issues

Operating System involvement w/ paging:

- **Process creation**
  - Determine program size, allocate space for page table, for swap, bring stuff into swap, record info into PCB

- **Process execution**
  - Reset MMU for new process, flush TLB, make new page table current, pre-page?

- **Page fault time**
  - Find out which virtual address cause the fault, find page in disk, get page frame, load page, reset PC, ...

- **Process termination time**
  - Release page table, pages, swap space, careful with shared pages
Page fault handling

- Hardware traps to kernel
- General registers saved by assembler routine, OS called
- OS find which virtual page cause the fault
- OS checks address is valid, seeks page frame
- If selected frame is dirty, write it to disk (CS)
- Get new page (CS), update page table
- Back up instruction where interrupted
- Schedule faulting process
- Routine load registers & other state and return to user space
Instruction backup

- As we’ve seen, when a program causes a page fault, the current instruction is stopped part way through …

- Harder than you think!
  - Consider instruction: MOV.L #6(A1), 2(A0)

One instruction, three memory references (instruction word itself, two offsets for operands)

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>MOVE</td>
<td>6</td>
</tr>
<tr>
<td>1002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

- Which one caused the page fault? What’s the PC then?
- It can even get worse – auto-decrement and auto-increment as a side-effect of instruction execution?

- Some CPU designers have included hidden registers to store
  - Beginning of instruction
  - Indicate autodecr./autoincr. and amount
Locking pages in memory

- Virtual memory and I/O occasionally interact
- Process issues call for read from device into a buffer within its address space
  - While waiting for I/O, another processes starts up
  - Second process has a page fault
  - Buffer for the first process may be chosen to be paged out!
  - If I/O device is doing a DMA transfer to that page, …

- Solutions:
  - Pinning down pages in memory
  - Do all I/O to kernel buffers and copy later
Backing store

- How do we manage swap area?
  - Allocate space to process when started
  - Keep offset to process swap area in PCB
  - Process can be brought entirely when started or as needed

- Some problems
  - Size – process can grow … split text/data/stack segments in swap area
  - Do not allocate anything … you may need extra memory to keep track of pages in swap!
Separation of policy & mechanism

- How to structure the memory management system for easy separation? Mach:
  1. Low-level MMU handler – machine dependent
  2. Page-fault handler in kernel – machine independent, most of paging mechanism
  3. External pager in user space – user-level process

- Where do you put the page replacement algorithm?
  - In external pager? No access to R and M bits
    - Either pass it to the pager or
    - fault handler informs external pager which page is the victim

- Pros and cons
Segmentation

- So far - one-dimensional address spaces
- For many problems, having multiple AS is better
  e.g. compiler with various tables that grow dynamically
- **Multiple AS → segments**
  - A logical entity – programmer knows
  - Different segments of different sizes
  - Each one growing independently
  - Address now includes segment # + offset
  - Protection per segment can be different

```
Symbol table
  free

Source text
  free

Segments
```
Segmentation and paging

- Paging pros and cons
  - Pros
    • Easy to allocate physical memory
    • Naturally leads to virtual memory
  - Cons
    • Address translation time
    • Page tables can be large

- Segmentation pros and cons
  - Pros
    • It’s more logical
    • Facilitates sharing and reuse
  - Cons
    • All the problems of variable partitions
Segmentation w/ paging - MULTICS

- Large segment? Page them e.g. **MULTICS** & Pentium
- Process: $2^{18}$ segments of ~64K words (36-bit)
- Most segments are paged
- Process has a segment table (itself a paged segment)
  - One entry per segment
- Segment descriptor indicates if in memory
- Segment descriptor points to page table
- Address of segment in secondary memory in another table

```
<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Descriptor segment</th>
<th>Page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment # (18b)</td>
<td>Page # (6b)</td>
<td>Offset (10b)</td>
</tr>
<tr>
<td></td>
<td>Page entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page entry</td>
<td></td>
</tr>
</tbody>
</table>
```
Segmentation w/ paging - MULTICS

With memory references
- Segment # to get segment descriptor
- If segment in memory, segment’s page table is in memory
- Protection violation?
- Look at the page table’s entry - is page in memory?
- Add offset to page origin to get word location
- … to speed things up - TLB

Segment Descriptor

<table>
<thead>
<tr>
<th>Segment length (in pages)</th>
<th>Main memory address of the page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>9</td>
</tr>
</tbody>
</table>

Page size
- 0 – 1024 words
- 1 = 64 words

Segment paged?
- Misc bits

Protection bits
Next time

- Principles of I/O, disks and disk arrays
- File and file systems
- ...