Design and Implementation Issues

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

Next
- I/O
Considerations with page tables

Two key issues with page tables

- **Mapping must be fast**
  - Done on every memory reference, at least 1 per instruction

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

- **Simplest solutions**
  - Page table in registers
    - Fast during execution, potentially $$$ & slow to context switch
  - Page table in memory & one register pointing to start
    - Fast to context switch & cheap, but slow during execution
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
    - Cache the virtual-to-physical translations in the hardware
    - Called a translation lookaside buffer (TLB)
    - TLB is 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
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
  - Choosing a victim is called “TLB replacement policy”
  - Implemented in hardware, usually simple (e.g., LRU)
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)

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

$\alpha = 80\%$ \quad $\varepsilon = 20 \text{ nsec}$ \quad memory-access = 100 nsec

$$EAT = 0.8 * (20 + 100) + 0.2 * (20 + 2 * 100) = 140 \text{ nsec}$$
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)

\[ \text{Virtual address: } 0x00403004 \]

\[
\begin{array}{c|c|c}
\text{p1} & \text{p2} & \text{offset} \\
 1 & 3 & 4 \\
\end{array}
\]

Example

Virtual address: 0x00403004

Page 1 | Page 2 | Offset
--------|--------|-------
 1     | 3     | 4     

P1 = 1  P2 = 3  Offset = 4
Three-level page table in Linux

- Designed to accommodate the 64-bit Alpha
  - Adjust for a 32-bit proc. – middle directory of size 1

<table>
<thead>
<tr>
<th>Global directory</th>
<th>Middle directory</th>
<th>Page table</th>
<th>Offset</th>
</tr>
</thead>
</table>

Virtual address

- Page size
- cr3 register
- Page directory
- Page middle directory
- Page table
- Page frame in physical memory
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) …
Page size

- 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

Average process size \(s\)
Page size \(p\)
Page entry size \(e\)

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 \(p = 4\text{KB}\)

Page table space
Internal fragmentation
Design issues – global vs. local policy

- When you need a page frame, pick a victim from
  - Among your own resident pages – Local
  - Among all pages – Global

- Local algorithms
  - Basically every process gets a fixed % of memory

- Global algorithms
  - Dynamically allocate frames among processes
  - Better, especially if working set size changes at runtime
  - How many page frames per process?
    - Start with basic set & react to Page Fault Frequency (PFF)

- Most replacement algorithms can work both ways except for those based on working set

Why not working set based algorithms?
Load control

- Despite good designs, system may still thrash
  - Sum of working sets > physical memory
- Page Fault Frequency (PFF) indicates that
  - Some processes need more memory
  - but no process needs less
- Way out: Swapping
  - So yes, even with paging you still need swapping
  - Reduce number of processes competing for memory
  - ~ two-level scheduling – careful with which process to swap out (there’s more than just paging to worry about!)

What would you like of the remaining processes?
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
Cleaning policy

To avoid having to write pages out when needed – paging daemon
- Periodically inspects state of memory
- Keep enough pages free
- If we need the page before it’s overwritten – reclaim it!

Two hands for better performance (BSD)
- First one clears R, second checks it
- If hands are kept close, only heavily used pages have a chance
- If back is just ahead of front hand (359 degrees), original clock
- Two key parameters, adjusted at runtime
  • Scanrate – rate at which hands move through the list
  • Handspread – gap between them
Virtual memory interface

- So far, transparent virtual memory
- Some control 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)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>MOVE</td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>2</td>
<td></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?
- 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 process starts up
  - Second process has a page fault
  - Buffer for the first process may be chosen to be paged out!

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?

- Pros and cons

![Memory Management System Diagram]
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

![Diagram of segmentation with symbol table, source text, and segments]
# Paging vs. segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td># Linear address spaces</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can procedure &amp; data be distinguished &amp; separately protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing procedures bet/processes facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was the technique invented?</td>
<td>Get a large virtual space w/o more physical memory?</td>
<td>Allow programs &amp; data to be broken into logically independent address spaces Aid sharing &amp; protection</td>
</tr>
</tbody>
</table>
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)
- 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>Page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment # (18b)</td>
<td>Page entry</td>
</tr>
<tr>
<td>Page # (6b)</td>
<td>Page entry</td>
</tr>
<tr>
<td>Offset (10b)</td>
<td>Page entry</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
Next time

- Principles of I/O hardware and software
- Disks and disk arrays
- … file systems