Virtual Memory

Slide 1

COMP3231 Operating Systems
2005/S2

Basic Features of Virtual Memory

- Piecewise-linear mapping of virtual to physical addresses
  - Paging: fixed and (usually) equal-sized blocks
    - only small internal fragmentation
    - programmer not aware of technique used
  - Segmentation: variable-sized blocks
    - external fragmentation
    - programmer aware of technique used
    - can be combined with fixed-sized

Slide 2

- Mapping is
  - defined at run-time, and
  - can change
- Process does not have to be contiguous in physical memory
- Address space can have holes (unmapped regions)
- Process image may be only partially resident
  - Allows OS to swap individual pages/segments to disk
  - Saves memory for infrequently used code or data
  - What happens if program accesses non-resident memory?

Slide 3

Typical Virtual Address Space Layout (UNIX):

<table>
<thead>
<tr>
<th>code (text)</th>
<th>data</th>
<th>bss</th>
<th>DLLs</th>
<th>stack</th>
<th>kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 0-th page typically not used
- text segment is read-only
- data segment is initialised data (partially R/O)
- bss segment is uninitialised data (heap), can grow
- shared libraries (DLLs) allocated in free middle region
- stack at top of user space, grows downward
- kernel space is in reserved (shared) region

Slide 4
MIPS R3000 Address Space Layout

**Slide 5**

- **kseg3**: 0xFFFFFFFF
- **kseg2**: 0xC0000000 - 0xA0000000
- **kseg1**: 0x80000000
- **kuseg**: 0x00000000

**Slide 6**

- **kseg3**: 0xFFFFFFFF
- **kseg2**: 0xC0000000 - 0xA0000000
- **kseg1**: 0x80000000
- **kuseg**: 0x00000000

**Slide 7**

- **kseg3**: 0xFFFFFFFF
- **kseg2**: 0xC0000000
- **kseg1**: 0x80000000
- **kuseg**: 0x00000000

**Slide 8**

- **kseg3**: 0xFFFFFFFF
- **kseg2**: 0xC0000000
- **kseg1**: 0x80000000
- **kuseg**: 0x00000000

### kseg2
- 512 MBytes
- Fixed mapping to physical memory:
  - Physical to virtual address: adding/subtracting fixed offset
  - Not cacheable
  - Only kernel-mode accessible
  - Boot ROM and device access

### kuseg
- 2 Gigabytes
- Dynamically mapped
- User-mode and kernel accessible
- Page-size: 4k

### kseg3
- Page-size: 4k

### kseg1
- 512 MBytes
- Fixed mapping to physical memory:
  - Physical to virtual address: adding/subtracting fixed offset
  - Cacheable
  - Only kernel-mode accessible
  - Usually where the kernel code is placed

### kseg2
- 1024 MBytes
- VM mapped
- Cacheable
- Only kernel mode accessible
Slide 9

Page table for process address space:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>VM</th>
<th>PT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>

Slide 10

**Shared Pages**

- Private code and data
  - Each process has its own copy of code and data
  - Code and data pages can appear anywhere in the VAS
- Shared code
  - Single copy of code shared between all processes executing it
  - Code must be “pure” (re-entrant), i.e., not self-modifying
  - Code must appear at the same address in all processes
    - Alternative: Position independent code (generally used for shared libraries)

Slide 11

Example: Shared pages:

Slide 12

**Page Table Structure**

- Page table is (logically) a mapping from page numbers to frame numbers
- Each page-table entry (PTE) also has
  - A valid bit (or present bit), indicating that there is a valid mapping for the page
  - A modified bit (also called dirty bit), indicating that the page may be modified in memory
Referencing an invalid page triggers a page fault:
- Illegal address (protection error)
- Page not resident, needs to be swapped in:
  1. Get empty frame
  2. Load page
  3. Update page table (enter frame #, set V, reset M)
  4. Restart faulting instruction

ADDRESS TRANSLATION (PAGING)
- Every (virtual) memory address issued by CPU must be translated to physical
  - Every load or store instruction
  - Every instruction fetch
- Need translation hardware
- In paging system, translation involves replacing page # by frame #

Address translation in a paging system:

PAGE TABLES
Assume we have
- 32 bit virtual addresses (4 GByte address space)
- 4 KByte page size ($2^{12}$)
- How many entries can the page table have for one process?

Problem:
- Page table is very large
- Access has to be fast, lookup for every memory reference
- Where to store the page table?
  - Register?
  - Main memory?
Most processes do not use a full 4GB address space:
- e.g., 0.1–1MB text, 0.1–10MB data, 0.1MB stack
- Need compact representation that doesn’t waste space

Three basic schemes:
- use data structures which adapt to sparsity
- use data structures which only represent resident pages
- use VM techniques for page tables

Two-level page table (32-bit addresses):
- Note: Unused PT pages not allocated (NULL pointer in root page table)

Address translation with 2-level PT (2LPT):
- Root page table
- Second-level page tables
- Match
- Entry
- Chain
- Page #
- Frame #
- Hash Anchor Table
- Inverted Page Table [Frame Table]

Alternative: Inverted page table
Inverted page table (IPT) operation:

- “Inverted page table” is an array of page numbers sorted by frame number (it’s a frame table)
- Algorithm:
  1. Compute hash of page number (usually least significant bits)
  2. Use this to index into the hash anchor table (HAT)
  3. HAT contains candidate frame number
  4. Use this to index into frame table
  5. Match the page number in FT entry to original
  6. If match, use frame # for translation
  7. If no match, get next candidate frame # from chain field
  8. If NULL chain entry => page fault

Properties of Inverted Page Tables

- IPT grows with size of RAM, not virtual address space!
- Frame table is needed anyway (for page replacement)
- Need separate data structure for non-resident pages
- Saves vast amount of space (esp. in 64bit Address systems)
- Searching does not come for free—efficiency?
- Currently used in some IBM and HP workstations

Alternative: Virtual linear array page table:

- Assume a 2-level PT
- Assume 2nd level PT nodes are in virtual memory
- Assume all 2nd level nodes are allocated contiguously
  ⇒ 2nd level nodes then form a contiguous array

Virtual linear array page table operation:

- Index into 2nd level page table without referring to root PT!
- Simply use the full page number as the PT index!
- Leave unused parts of PT unmapped!
- If access is attempted to unmapped part of PT, a secondary page fault is triggered
  - This will load the mapping for the PT from the root PT
  - Root PT is kept in physical memory (cannot trigger page faults)
Problem:
- Each virtual memory reference can cause two physical memory accesses
  - one to fetch the page table entry
  - one to fetch/store the data
- Intolerable performance impact!

Solution:
- High-speed cache for page table entries (PTEs)
  - Called the translation lookaside buffer (TLB)
  - Contains recently used page table entries
  - Associative high-speed memory, similar to a memory cache
  - May be under OS control (unlike memory cache)

**Translation Lookaside Buffer**
- Given a virtual address, processor examines the TLB
- If matching PTE is found (TLB hit), the address is translated
- Otherwise (TLB miss), the page number is used to index the process page table
  - If PT contains a valid entry, reload TLB and restart
  - Otherwise (page fault) check if page is on disk
    - If on disk, swap in
    - Otherwise allocate a new page or raise an invalid address exception

**TLB operation::**
- TLB operation process:
  - **TLB hit**: The address is translated directly.
  - **TLB miss**: The page number is used to index the process page table.
    - If the page is valid in memory, it is loaded.
    - If not, a page fault occurs.

**TLB properties:**
- TLB entries contain (per-page) write-protect bits
- TLB may or may not be under OS control:
  - **Hardware-loaded TLB**:
    - On miss, hardware performs PT lookup and reloads TLB
    - Example: Pentium, most 32-bit architectures, PowerPC
  - **Software-loaded TLB**:
    - On miss, hardware generates a TLB miss exception, and exception handler reloads TLB
    - Example: MIPS, and most modern architectures
- TLB size: typically 64-128 entries
- Modern architectures generally have separate TLBs for instruction fetch and data access
- TLB can also be used for inverted page tables

*Page Tables and the TLB* 14
Page Tables and the TLB

- Page table is (logically) an array of frame #’s indexed by page #
- The TLB holds a (recently used) subset of PT entries
  - each TLB entry must be identified (tagged) with the page # it translates
  - access is by associative lookup:
    - all TLB entries’ tags are concurrently compared to the page #
  - TLB is associative (or context-addressable) memory

TLB is a shared piece of hardware
Page tables are per-process (address space)
TLB entries are process-specific

Slide 29

Page tables and the TLB:
- TLB is a shared piece of hardware
- Page tables are per-process (address space)
- TLB entries are process-specific

Solutions:
1. On context switch need to flush the TLB (invalidate all entries)
   - high context-switching overhead (x86)
2. or tag entries with address-space ID (ASID)
   - called a tagged TLB
   - used (in some form) on all modern architectures
   - TLB entry: ASID, page #, frame #, valid and write-protect bits
   - On TLB load:
     - construct entry on-the-fly from data in PTE, or
     - load a complete TLB entry from PT

Example: MIPS R4000 TLB entry (32-bit mode): (R3000 similar, see Hardware Guide on course web page)

<table>
<thead>
<tr>
<th>MASK</th>
<th>127</th>
<th>121</th>
<th>109</th>
<th>96</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0_PAGEMASK</td>
<td>7</td>
<td>12</td>
<td>109</td>
<td>0</td>
</tr>
<tr>
<td>C0_ENTRYHI</td>
<td>95</td>
<td>77</td>
<td>72</td>
<td>64</td>
</tr>
<tr>
<td>VPN2</td>
<td>19</td>
<td>1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>ASID</td>
<td>63</td>
<td>38</td>
<td>35</td>
<td>32</td>
</tr>
<tr>
<td>PFN</td>
<td>2</td>
<td>24</td>
<td>3</td>
<td>1 1 1</td>
</tr>
<tr>
<td>C0_ENTRYLO1</td>
<td>31</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C0_ENTRYLO0</td>
<td>31</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Explanation:
- Current TLBE is mirrored in CP0 registers C0_PAGEMASK, C0_ENTRYHI, C0_ENTRYLO1, C0_ENTRYLO0
- Each entry (of 48) maps a pair of pages (buddies) with common value of VPN2 (= virtual page #/2)
  - ENTRYLO0 maps page \((2 \times \text{VPN2})\)
  - ENTRYLO1 maps page \((2 \times \text{VPN2} + 1)\)

- MASK defines size of page (0: page size is 4kB)
- ASID address-space ID tag
- PFN physical frame number
- C page cacheability/coherency
- D dirty: page is writable
- V mapping is valid
- G global: ignore ASID (in tag word in TLB, in ENTRYLO in CP0)
- 0 must be zero
MIPS addressing (highly simplified):

- Virtual addressing:
  - Addresses in the bottom half of the VAS (<0x8000 0000) are translated by the TLB.

- Quasi-physical addressing:
  - Addresses in the upper half of the VAS (≥0x8000 0000) are translated by masking out the top bits:
    \[ pa = va & 0xffffff \]
  - These addresses are called kseg0 addresses.
  - Kseg0 addresses are only available in kernel mode.

- Typical use: \( (pa + K0SEG0 \text{BASE}) \) to refer to \( pa \)
  - Used e.g. by exception handlers.

---

Exception handling on MIPS:

- MIPS has 4 different exception handlers:
  - 32-bit mode fast TLB miss handler TLB
    - At PA 0x000 (VA 0x8000 0000)
    - Unless already in exception mode.
  - 64-bit mode fast TLB miss handler XTLB
    - At PA 0x080 (VA 0x8000 0080)
    - Unless already in exception mode.
  - Cache error handler CACHE
    - At PA 0x100 (VA 0x8000 0100)
  - General exception handler GENERAL
    - At PA 0x180 (VA 0x8000 0180)
    - All other exceptions
    - Including TLB exceptions in exception mode.

---

32-bit TLB miss handling on MIPS (simplified):

- TLB refill exception: TLB miss in non-exception mode
- MIPS processor does the following:

  \[
  \begin{align*}
  & CP0.EPC \rightarrow PC \\
  & CP0.CAUSE.ExcCode \rightarrow TLB ; \text{if read fault} \\
  & CP0.CAUSE.ExcCode \rightarrow TLBS ; \text{if write fault} \\
  & CP0.BadVaddr \rightarrow faulting address \\
  & CP0.EntryHi.VPN2 \rightarrow faulting address/(2^\text{pagesize}) \\
  & CP0.STATUS.EXL \rightarrow 1 ; \text{enter exception mode} \\
  & CP0.PC \rightarrow 0x8000 0000 ; \text{fast TLB miss handler} \\
  \end{align*}
  \]

  \[ \text{Note: ASID is already contained in CP0.EntryHi.ASID} \]

---

TLB miss handling...:

- Software does:
  1. Look up PTE corresponding to fault address.
  2. If found:
     1. Load \( C0 ENTRYLO0 \) and \( C0 ENTRYLO1 \) with translations.
     2. Load TLB using \( tlbwr \) instruction.
     3. Return from exception.
     4. Else handle page fault.

- TLB entry (i.e., \( C0 ENTRYLO0, C0 ENTRYLO1 \)) can be:
  - Created on the fly, or
  - Stored completely and in the right format in the PT.
Other TLB exceptions:

- vectored to GENERAL handler:
  - TLB invalid: entry exists but V flag off
  - TLB modified: write to page with D flag off
- Processor handles as for TLB miss, except:
  - CP0.CAUSE.ExcCode ← TLBL; if read invalid fault
  - CP0.CAUSE.ExcCode ← TLBS; if write invalid fault
  - CP0.CAUSE.ExcCode ← Mod; if write protect fault
  - CP0.PC ← 0x80000180; gen except handler

Software must:
- interpret and fix the cause of the invalid access
- fix PT entry
- reload TLB

OS/161 Refill Handler:
- Switches to kernel stack
- Calls common exception handler
- Unoptimised
- Written for ease of kernel programming, not efficiency
- Does not have a page table
- If 64 TLB entries are full, kernel panics
- supports only 256K user-level address space

Segmentation

- Variable-sized segments instead of fixed-size pages
- Size may be dynamic (changeable at run time)
- Natural support for modularisation (dynamic libraries)
- Lends itself to sharing along logical module boundaries
- Lends itself to module-based protection
- External fragmentation but no internal fragmentation
- Introduce a non-linear address space (segment #, offset)
- Per-process segment table instead of page table

Protection using segments:

<table>
<thead>
<tr>
<th>Address</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-20K</td>
<td>Dispatcher</td>
</tr>
<tr>
<td>30K-50K</td>
<td>Process A</td>
</tr>
<tr>
<td>80K-90K</td>
<td>Process B</td>
</tr>
<tr>
<td>140K-190K</td>
<td>Process C</td>
</tr>
</tbody>
</table>

- No access allowed
- Branch instruction (not allowed)
- Reference to data (not allowed)
Flat vs. segmented address space:

- **Flat address space**: no protection...
- **Segmented address space**: protection...
  ...against segment overrun

### Segment Tables:
- Translates segment number to main memory address
- Each entry contains:
  - physical start address of segment
  - segment length
  - valid bit
  - dirty bit

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Number</td>
<td></td>
</tr>
<tr>
<td>Offset</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment Table Entry</th>
<th>Length</th>
<th>Segment Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other Control Bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Address translation in a segmentation system:

**Combined Paging and Segmentation**
- Paging is transparent to the programmer
- Paging eliminates external fragmentation
- Paging supports (relatively) fine-grain memory management
- Segmentation is visible to the programmer
- Segmentation allows for growing data structures, modularity, and module-based support for sharing and protection
- Combination: Segments broken into fixed-size pages
- Examples: Pentium, PowerPC, HP PA-Risc, IA-64
### Slide 45

**Combined segmentation and paging:**
- **Per-process** segment table
- **Per-segment** page table

**Virtual Address**

<table>
<thead>
<tr>
<th>Segment Number</th>
<th>Page Number</th>
<th>Offset</th>
</tr>
</thead>
</table>

**Segment Table Entry**

<table>
<thead>
<tr>
<th>Other Control Bits</th>
<th>Length</th>
<th>Segment Base</th>
</tr>
</thead>
</table>

**Page Table Entry**

<table>
<thead>
<tr>
<th>Other Control Bits</th>
<th>Frame Number</th>
</tr>
</thead>
</table>

---

### Slide 46

**Address translation with paged segments:**

1. Virtual Address
2. Segment Table
3. Page Table
4. Frame Number

---

### Slide 47

**Address-space switch**

- The address space is defined by page or segment table(s)
- Part of that information is cached in the TLB
- Data and instructions are cached in the CPU caches

**Switching AS (during a context switch):**

- The page/segment table pointer needs to be changed
  - if the TLB is tagged:
    - The ASID register needs to be reloaded (CTR_I on MIPS)
  - otherwise:
    - The TLB must be flushed (all entries invalidated)
- if the CPU caches are not tagged:
  - The caches must be flushed

---

### Slide 48

**Demand Paging/Segmentation**

- With VM, only parts of the program image need to be resident for execution
- Can swap presently unused pages/segments to disk
- Reload non-resident pages/segments on demand
  - Reload is triggered by a page or segment fault
  - Faulting process is blocked and another scheduled
  - When page/segment is resident, faulting process is restarted
  - May require freeing up memory first
    - Replace currently resident page/segment
    - How determine replacement "victim"?
  - If victim is unmodified ("clean") can simply discard
    - This is the reason for maintaining the "dirty" bit in the PT
Why does demand paging/segmentation work?

➜ Program executes at full speed while only accessing resident set of pages or segments
➜ TLB miss introduces delay of several micro-seconds
➜ Page/segment fault introduces delay of several milli-seconds

Why do it?

Answers:
1. Less physical memory required per process
   ➜ can fit more processes into memory
   ➜ improved chance of finding a runnable process
2. Principle of locality

Principle of Locality:

➜ Process’ program and data references tend to cluster
➜ Only a small number of pages/segments will be needed during a (short) time window
   ● Called the memory working set of a process
   ● System keeps at least working set of process resident
   ● process can execute while it stays within working set
   ● Working set tends to change gradually
     - Get only a few page/segment faults during time window
     - Possible to make intelligent guesses about which pieces will be needed in the future
     - May be able to pre-fetch pages/segments

Thrashing:

➜ CPU utilisation tends to increase with the degree of multiprogramming
   ● number of processes in the system
   ● Higher degree of multiprogramming—less memory available per process
   ➜ Some process’s working sets may no longer fit in RAM
     ● increasing page fault rate
   ➜ Eventually many processes have insufficient memory
     - can’t always find runnable process
     - decreasing CPU utilisation
     - system is I/O limited
   ➜ This is called thrashing

Thrashing...:
Remember this?:

```c
void ResetArray (int array[10000][10000]) {
  int i, j;
  for (i=0; i<10000; i++) {
    for (j=0; j<10000; j++) {
      array[i][j] = 0;
      /* OR array[j][i] = 0 */
    }
  }
}
```

What's the difference?

---

**VM Management Policies**

Operation and performance of VM system is dependent on a number of policies:
- Page table format (may be dictated by hardware)
  - multi-level
  - hashed
- Page size (may be dictated by hardware)
- Fetch policy
- Replacement policy
- Resident set size
  - minimum allocation
  - local vs global allocation
- Page cleaning policy
- Degree of multiprogramming ...

---

**Example Page Sizes:**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlas</td>
<td>512 words (48-bit)</td>
</tr>
<tr>
<td>Honeywell/Multics</td>
<td>1k words (36-bit)</td>
</tr>
<tr>
<td>IBM 370/XA, 370/ESA</td>
<td>4k bytes</td>
</tr>
<tr>
<td>DEC VAX</td>
<td>512 bytes</td>
</tr>
<tr>
<td>IBM AS/400</td>
<td>512 bytes</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>4k and 4M bytes</td>
</tr>
<tr>
<td>ARM</td>
<td>4k and 64k bytes</td>
</tr>
<tr>
<td>MIPS R4000</td>
<td>4k–16M bytes in powers of 4</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>8k–4M bytes in powers of 8</td>
</tr>
<tr>
<td>UltraSPARC</td>
<td>8k–4M bytes in powers of 8</td>
</tr>
<tr>
<td>PowerPC</td>
<td>4k bytes plus “blocks”</td>
</tr>
<tr>
<td>Intel IA-64</td>
<td>4k–256M bytes in powers of 4, etc</td>
</tr>
</tbody>
</table>
Multiple page sizes provide the flexibility to optimise the use of the TLB.

- Large pages can be used for code.
- Small pages for thread stacks.
- Most operating systems support only one page size.
- Dealing with multiple page sizes is hard!

Fetch Policy

- Determines when a page should be brought into memory.
  - Demand paging only loads pages in response to page faults.
  - Many page faults when process first started.
  - Pre-paging brings in more pages than needed at the moment.
    - Improve I/O performance by reading larger chunks.
    - Pre-fetch when disk is idle.
    - Wastes I/O bandwidth if pre-fetched pages aren’t used.

Replacement Policy

- Which page is chosen to be tossed out?
  - Page removed should be the page least likely to be referenced in the near future.
  - Most policies attempt to predict the future behavior on the basis of past behavior.
- Constraint: locked frames:
  - Kernel code.
  - Main kernel data structures.
  - I/O buffers.
  - Performance-critical user pages (e.g., for DBMS).
- Frame table has lock bit.

Basic Replacement Policies

Optimal:

- Toss the page that won’t be used for the longest time.
- Impossible to implement.
- Only good as a theoretical reference point:
  - The closer a practical algorithm gets to optimal, the better.

Example:

- Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
- Four frames.
- How many page faults?
Basic Replacement Algorithms: FIFO:
- First-in, first-out: Toss oldest page
  - Easy to implement
  - Age of a page isn’t necessarily related to its usage
Example:
- reference string: 1 2 3 4 1 2 5 1 2 3 4 5
- four frames
- how many page faults?
- three frames
Belady’s anomaly: more frames ≠ fewer page faults

Basic Replacement Algorithms: LRU
- Toss least recently used page
  - Assumes that a page that hasn’t been referenced for a long time is unlikely to be referenced in the near future
  - Will work if locality holds
  - Implementation requires time stamp to be kept for each page, updated on every reference
  - Impossible to implement efficiently
  - Most practical algorithms are approximations of LRU
How many page faults for example sequence?
- reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Basic Replacement Algorithms: Clock
- Clock policy, also called second chance
  - Employs a usage or reference bit in frame table
  - Set to one when page is used
  - When scanning for a victim, reset all reference bits
  - Toss first page with zero reference bit.
- How do we know when a page has been referenced?
  - Use the valid bit in the PTE:
    - when page is mapped (valid bit set), set reference bit
    - on TLB fault:
      - turn on valid bit in PTE
      - turn on reference bit in frame table

Example of operation of clock policy:
**Performance**

**Slide 65**

![Graph Showing Performance](image)

**Slide 66**

Note: there are other algorithms (working set, ageing, NFU, ...) — we don’t expect you to know them in this course.

**Slide 67**

**Basic Replacement Algorithms: Page buffering**

- Replace pages before running out of memory
- “Replaced” frame is added to one of two lists
  - free-frame list if page has not been modified
  - modified-frame list if dirty
- clean (write back) modified frames asynchronously
  - migrates frames from modified to free list
- when in need of a frame, get it from free-frame list
- on page fault check both lists first

---

**Resident Set Size**

**Slide 68**

- Fixed allocation
  - gives a process a fixed number of pages within which to execute
  - when a page fault occurs, one of the pages of that process must be replaced
- Variable allocation
  - number of pages allocated to a process varies over the lifetime of the process
Variable Allocation, Global Scope:
- Easiest to implement
- Adopted by many operating systems
- Operating system keeps list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no free frame, replaces one from any process

Cleaning Policy
- Demand cleaning
  - a page is written out only when it has been selected for replacement
- Precleaning
  - pages are written out in batches
  - generally used with frame buffering

Variable Allocation, Local Scope:
- Allocate number of page frames to new processes based on
  - application type
  - program request,
  - other criteria...
- When page fault occurs, select page from among the resident set of the process that suffers the fault
- Reevaluate allocation from time to time

Load Control (Degree of Multiprogramming)
- Determines the number of runnable processes
- Controlled by:
  - Admission control:
    - only let new process’s thread proceed from new to ready state if enough memory available
  - Suspension:
    - move all threads of some processes into special suspended state
    - swap complete process image of suspended processes to disk
- Tradeoff:
  - Too many processes will lead to thrashing
  - Too few will lead to idle CPU/excessive swapping