[Review] Memory Mapping: Paging

- Motivation: Partitioning suffers from large external fragmentation

Paging
- view physical memory as composed of several fixed-size frames
  - a "frame" is a physical memory allocation unit
- view logical memory as consisting of blocks of the same size: pages
- allocation problem
  - put "pages" into "frames"
    - a page table maintains the mapping
  - allocation need not preserve the contiguity of logical memory
    - e.g., pages 1, 2, 3, 4 can be allocated to frames 3, 7, 9, 14
  - how does this avoid external fragmentation?
- paging played a major role in virtual memory design
  - separation between the meaning of a location in the user's virtual space and its actual physical storage

Outline

- Announcements
  - Graded exams returned today
  - Graded write-up 2 will be returned on March 22nd
  - Lab 4 due March 27th, TA will post suggested solutions for parts of Lab 3
  - Questions?
- Exam solutions
- Memory management
  - Review from Lecture 12
  - Paging
  - Segmentation
  - Segmentation with paging
  - Example memory organizations

[Silberschatz/Galvin: Sections 8.5 – 8.8]
Memory Mapping: Paging (contd.)

Mapping of pages to frames
- the mapping is hidden from the user and is controlled via the OS

- Allocation of frames to processes (Nachos Lab 4)
  - the OS maintains a map of the available and allotted frames via a structure called a frame table
    - whether a frame is allocated or not
    - if allocated, to which page of which process

- Address translation
  - performed on every memory access
  - must be performed extremely efficiently so as to not degrade performance
  - typical scheme
    - frames (and pages) are of size $2^k$
    - for each logical address of $a = m + n$ bits
      - the higher order $m$ bits indicate the page number $p_i$ and
      - the remaining $n$ bits indicate the offset $w_i$ into the page

Page Table Structure

- How can we support efficient page table lookups?

- Page table is stored in fast on-chip registers
  - loaded and saved with each process at context switch time
  - constructed from fast and expensive logic to enable rapid translation
    - reasonable if the size of the page table is small (e.g., ~256 entries)
      - modern systems require millions of page table entries

- More reasonable: Store page table in memory
  - a single page table base register that
    - is loaded in with the process
    - points to the beginning of the page table
    - $p_i$ is now the offset into this table
  - problem
    - requires two accesses to memory for each value
    - even with caches, can become very slow

Translation Lookaside Buffers

- Improve lookup costs for memory-allocated page tables
  - a special hardware memory unit called the translation lookaside buffer (TLB) is constructed using associative registers
    - sizes of the order of 2K are reasonable
  - a portion of the page table is cached in the TLB
    - little performance degradation if a value is a hit in the TLB
    - if not, a memory access is needed to load the value into the TLB
      - an existing value must be flushed if the TLB is full
**Multi-level Page Tables**

- **Rationale:** Modern systems support a very large logical address space
  - page tables themselves become very large
    - e.g., for a system with 32-bit logical addresses and 4K pages
      - we need $2^{20}$ page table entries (4 bytes per PTE implies 4 MB of space)
    - solution: page the page table itself
    - cost: additional memory accesses (but caching helps)

**Logical address**

<table>
<thead>
<tr>
<th>PID</th>
<th>p1#</th>
<th>disp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>base-addr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p2#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Physical address**

<table>
<thead>
<tr>
<th>size</th>
<th>base-addr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Page Tables and Sharing**

- **Page tables permit different virtual addresses (frames of different processes) to map to the same physical address**
  - convenient sharing of common code (dynamically-linked system libraries)
  - shared data segments for IPC

**Inverted Page Tables**

- **Observation**
  - usually, only a portion of all the pages from the system’s memory can be stored in the physical memory
  - so while the required page table for all of logical memory might be massive, only have a small subset of it contains useful mappings

- **We can take advantage of this fact in both TLB and page table design**

**Inverted Page Tables (contd.)**

- **Efficiency considerations**
  - the inverted page table is organized based on physical addresses via frame numbers
    - searching for the frame number can be very slow
  - use a hash table based on
    - the PID and logical page number as keys
  - recently located entries of the inverted page table can be stored in a TLB-like structure based on associative registers

- **Main disadvantage of inverted page tables: sharing**
  - each process that shares an object will have its own (disjoint) space where the shared object is mapped
  - not possible to maintain with standard inverted page tables
    - since space for only one <PID, page number> tuple
Memory Mapping: Protection Issues with Paging

- Partition protection scheme
  - Check that address lies between base and base+limit
  - Cannot be used on page-based systems: WHY?

- Special bits in the page table entry enforce per-frame protection
  - an accessibility bit
    - whether a page is invalid, readable, writable, executable
  - a valid/invalid bit to indicate whether a page is in the user's (logical) space
  - system support
    - all addresses are interpreted by the MMU
    - OS intervention required to manipulate page tables and TLBs

- Sometimes, the hardware may support a page-table length register
  - specifies size of the process page table
    - trailing invalid pages can be eliminated
    - useful when processes are using a small fraction of available address space

Memory Mapping: Segmentation

- A segment is a logical piece of the program
  - e.g., the code for the program functions, its data structures, symbol tables

- Segmentation views logical memory is broken into such segments
  - segments are of variable size (unlike pages)

- Accessing a segment
  - the logical address is regarded as two-dimensional
    - a segment pointer to an entry in the segment table
    - a displacement into the segment itself

- Allocating a segment
  - a segment is a partition with a single base-limit pair
    - the limit attribute stores the segment length
      - prevents programs from accessing locations outside the segment space

Memory Mapping: Segment Table Lookup

- Mapping logical addresses to physical addresses
  - the mapping is maintained by the segment table
  - the segment number s# is used to index into the (process’) segment table
    - where the corresponding segment size and base address are stored

Memory Mapping: Segmentation Hardware

- Segment registers
  - some designs (e.g. Intel x86) provide registers to identify segments
    - loading a segment register loads a (hidden) segment specification register from
      - the segment table
  - construction of the logical address is done explicitly

- TLBs
  - some designs, such as the MIPS 2000, only provide a TLB
    - the OS is responsible for loading this, and doing appropriate translation

- Traditional approach: Store the segment table in memory
  - segment table base register (STBR), segment table length register (STLR)
    - saved and restored on each context switch
  - translation of address (s,d)
    - check that s is valid: s < STLR
    - check that offset d is valid: d < length
    - compute physical address
Segmentation: Pros and Cons

- **Pros**
  - protection in terms of ensuring that illegal address accesses are avoided, comes for free
    - the segment length check plays an important role here
  - sharing segments across programs is straightforward by loading identical segment table base register values

- **Cons**
  - external fragmentation is potentially a big problem
  - contrast this with paging where only internal fragmentation is possible

Memory Mapping: Segmentation and Paging

- Overlay a segmentation scheme on a paging environment
  - several examples
    - originally proposed for GE 645 / Multics
    - Intel x86 uses segment registers to generate 32-bit logical addresses, which are translated to physical addresses by an optional multi-level paging scheme
  - alleviates the problem of external fragmentation

Memory Mapping: Examples

- **Multics (c. 1965)**
  - 34-bit logical address
    - 18-bit segment number, 16-bit offset
    - [8-bit major segment, 10-bit minor segment], [6-bit page, 10-bit offset]
    - Both the segment table and segment itself are paged!
  - segmentation structure
    - Segment table is paged
    - major segment number indexes page table for segment table
    - minor segment number is offset within the page of the segment table
    - this gives the page table of the desired segment and the segment length
  - paging structure
    - one-level page table, 1KB pages
  - TLB
    - 16 entries; key = 24-bit (seg# & page#); value = frame#

Memory Mapping: Examples (contd.)

- **OS/2 (on Intel 386+)**
  - segmentation with paging: up to 16K segments (max 4 GB), 4KB pages
  - logical address space of process is divided into two partitions
    - first partition (up to 8K segments) are private (local descriptor table: LDT)
    - second partition (up to 8K segments) are shared (global descriptor table: GDT)
    - Each GDT/LDT entry is 8 bytes (24-bit segment base, 20-bit limit)
  - logical address: 16-bit segment selector, 32-bit offset
    - 16-bit selector: 13 (segment number) + 1 (GDT/LDT) + 2 (protection: rings)
  - machine has six 64-bit microprogram segment description registers
    - automatically loaded when a selector is loaded into a segment register
  - Translation of logical address into 32-bit physical address
    - load selector into segment register: microprogram register gets entry
    - check that offset is within limit: 20 bits interpreted as bytes or pages
    - form 32-bit linear address by adding 24-bit base to 32-bit offset
    - resolve 32-bit linear address using a 2-level page table scheme
      - 10-bit page directory, 10-bit page#, 12-bit offset
Next Lecture

- Virtual Memory
  - background: process working sets
  - demand paging
  - page replacement
  - frame allocation
  - thrashing

Reading
- Silberschatz/Galvin: Chapter 9