Memory Management

- Address binding
  - Linking, loading
  - Logical vs. physical address space
- Fragmentation
- Paging
- Segmentation

- Reading: Silberschatz (8th ed.), Chapter 8

---

Memory Management

- Observations:
  - Process needs at least CPU and memory to run.
  - CPU context switching is relatively cheap.
  - Swapping memory in/out from/to disk is expensive.

- Need to subdivide memory to accommodate multiple processes!

- How do we manage this memory?
Requirements for Memory Management

- **Relocation**
  - We do not know a priori where memory of process will reside.

- **Protection**
  - No uncontrolled references to memory locations of other processes.
  - Memory references must be checked at run-time.

- **Sharing**
  - Data portions and program text portions.

- **Logical organization**
  - Take advantage of semantics of use.
  - Data portions (read/write) vs. program text portions (read only).

- **Memory hierarchy**
  - RAM vs. secondary storage
  - Swapping

Preparing a Program for Execution

- **compiler**: translates symbolic instructions, operands, and addresses into numerical values.

- **linker**: resolves external references; i.e. operands or branch addresses referring to data or instructions within some other module

- **loader**: brings program into main memory.
Address Binding

- Compile-time binding:
  - `branch A` assembler `branch 150`

- Load-time binding (static relocation):
  - Branch instructions:
    - `branch A` assembler `branch 50`
    - Linker `linker 150`
    - Loader `loader 1150`
  - Other modules:
    - Other program `other module`

- Execution-time binding (dynamic relocation):
  - Branch instructions:
    - `branch A` assembler `branch 50`
    - Linker `linker 150`
    - Loader `loader 1150`
  - Other modules:
    - Other program `other module`
    - MMU `MMU`

Dynamic Loading, Dynamic Linking

- Dynamic Loading:
  - Load routine into memory only when it is needed.
  - Routines kept on disk in relocatable format.
- Load-time Linking:
  - Postpone linking until load time.
- Dynamic Linking:
  - Postpone linking until execution time.
  - Problem: Need help from OS!

- Call `x`
- Stub
- Load routine `x` to location of routine.
Logical vs. Physical Address Space

- **Logical address**: address as seen by the process (i.e. as seen by the CPU).
- **Physical address**: address as seen by the memory.

Example:

Load time:

```
branch A
A: ...
```

```
00  branch 50  100  branch 150  1100  branch 1150
assembler  linker  loader
```

Execution time:

```
branch A
A: ...
```

```
00  branch 50  100  branch 150  1100  branch 150
assembler  linker  loader  MMU
```

Logical vs. Physical Memory Space

- **Logical address**: address as seen by the process (i.e. as seen by the CPU).
- **Physical address**: address as seen by the memory.

```
<table>
<thead>
<tr>
<th>process</th>
<th>base</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>28</td>
<td>1000</td>
</tr>
<tr>
<td>P_2</td>
<td>1028</td>
<td>3000</td>
</tr>
<tr>
<td>P_3</td>
<td>5034</td>
<td>250</td>
</tr>
</tbody>
</table>
```
Swapping

Fragmentation

Memory Management: Paging / Segmentation
Paging

- Contiguous allocation causes (external) fragmentation.
- Solution: Partition memory blocks into smaller subblocks (pages) and allow them to be allocated non-contiguously.

Simple relocation:

```
logical memory
```

Paging:

```
logical memory
```

Basic Operations in Paging Hardware

```
CPU
```

Example: PDP-11 (16-bit address, 8kB pages)
Internal Fragmentation in Paging

- Example:

![Diagram showing internal fragmentation in paging]

- Last frame allocated may not be completely full.
- **Average internal fragmentation** per block is typically **half frame size**.
- Large frames vs. small frames:
  - Large frames cause more fragmentation.
  - Small frames cause more overhead (page table size, disk I/O)

Implementation of Page Table

- Page table involved in every access to memory. Speed very important.
- Page table in registers?
  - Example: 1MB logical address space, 2kB page size; needs a page table with 512 entries!
- Page table in memory?
  - Only keep a page table base register that points to location of page table.
  - Each access to memory would require two accesses to memory!
- Cache portions of page table in registers?
  - Use translation lookaside buffers (TLBs): typically a few dozens entries.
  - **Hit ratio**: Percentage of time an entry is found. Hit ratio must be high in order to minimize overhead.
Hierarchical (Multilevel) Paging

- **Problem:** Page tables can become very large! (e.g. 32-bit address space?)
- **Solution:** Page the page table itself! (e.g. page directory vs. page table)
- **Two-level paging:**
  - Example: 32 bit logical address, page size 4kB

- Three-level paging (SPARC), four-level paging (68030), ...
- AMD64 (48-bit virtual addresses) has 4 levels.
- Even deeper for 64 bit address spaces (5 to 6 levels)

Variations: Inverted Page Table

- Array of page numbers indexed by frame number.
  - page lookup: search for matching frame entry
- Size scales with physical memory.
- Single table for system (not per process)
- Used in early virt. memory systems, such as the Atlas computer.
- Not practical today. (Why?)
Variations: Hashed Page Table

- Used by many 64bit architectures:
  - IBM POWER
  - HP PA-RISC
  - Itanium
- Scales with physical memory
- One table for whole system
- Difficult to share memory between processes

Software-loaded TLBs: Paging – MIPS Style

Memory Management: Paging / Segmentation
Recap: Memory Translation -- “VAX style”

1. Split virtual address
2. Concatenate more-significant bits with Process ASID to form page address.
3. Look in the TLB to see if we find translation entry for page.
4. If YES, take high-order physical address bits.
   – (Extra bits stored with PFN control the access to frame.)
5. If NO, system must locate page entry in main-memory-resident page table, load it into TLB, and start again.

Memory Translation -- MIPS Style

- In principle: Do the same as VAX, but with as little hardware as possible.
- Apart from register with ASID, the MMU is just a TLB.
- The rest is all implemented in software!
- When TLB cannot translate an address, a special exception (TLB refill) is raised.
- Note: This is easy in principle, but tricky to do efficiently.
MIPS TLB Entry Fields

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>PFN</td>
</tr>
<tr>
<td>ASID</td>
<td>Flags</td>
</tr>
<tr>
<td>G</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>V</td>
</tr>
</tbody>
</table>

- **VPN**: higher order bits of virtual address
- **ASID**: identifies the address space
- **G**: if set, disables the matching with the ASID
- **PFN**: Physical frame number
- **N**: 0 - cacheable, 1 - noncacheable
- **D**: write-control bit (set to 1 if writeable)
- **V**: valid bit

MIPS Translation Process

1. CPU generates a program (virtual) address on a instruction fetch, a load, or a store.
2. The 12 low-end bits are separated off.
3. Case 1: TLB matches key:
   1. Matching entry is selected, and PFN is glued to low-order bits of the program address.
   2. **Valid?**: The V and D bits are checked. If problem, raise exception, and set BadVAddr register with offending program address.
   3. **Cached?**: IF C bit is set, the CPU looks in the cache for a copy of the physical location’s data. If C bit is cleared, it neither looks in nor refills the cache.
4. Case 2: TLB does not match: **TLB Refill Exception** (see next page)
TLB Refill Exception

- Figure out if this was a correct translation. If not, trap to handling of address errors.
- If translation correct, construct TLB entry.
- If TLB already full, select an entry to discard.
- Write the new entry into the TLB.

Segmentation

- Users think of memory in terms of segments (data, code, stack, objects, ...).
- Data within a segment typically has uniform access restrictions.
Memory Management: Paging / Segmentation

Segmentation Hardware

- CPU
- Segment table
- Physical memory

Advantages of Segmentation

- Data in a segment typically semantically related
- Protection can be associated with segments
  - Read/write protection
  - Range checks for arrays
- Data/code sharing
- Disadvantages?
Solution: Paged Segmentation

- Example: MULTICS

<table>
<thead>
<tr>
<th>segment number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>18bit</td>
<td>16bit</td>
</tr>
</tbody>
</table>

**Problem:** 64kW segments → external fragmentation!
**Solution:** Page the segments.

<table>
<thead>
<tr>
<th>segment number</th>
<th>page #</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>18bit</td>
<td>6bit</td>
<td>10bit</td>
</tr>
</tbody>
</table>

**Problem:** need $2^{18}$ segment entries in segment table
**Solution:** Page the segment table.

<table>
<thead>
<tr>
<th>page #</th>
<th>page offset</th>
<th>page #</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>8bit</td>
<td>10bit</td>
<td>6bit</td>
<td>10bit</td>
</tr>
</tbody>
</table>