How Do Programs Use Names?

- Registers: i, t, *t
  Register addressing mode
- Code: spaces In String
  Absolute or PC-relative addressing mode
- Static data: s
  Absolute addressing mode
- Stack: i, t, return address
  Relative addressing mode
- Heap:
  Indirect addressing mode
- Dynamically-linked libraries:
  Indirect addressing mode

```
static char s[LEN];
int spacesInString (char *t) {
  int i = 0;
  while (*t) {
    if (*t++ == ' ')
      (i++)
  }
  return i;
}
```
Relocation and Protection

Single Program

Multiple Programs

- Relocation
- No Protection
Relocation and Protection

Base Register Addressing

Base 0

Code Static Data Stack Heap

Base 1

Code Static Data Stack Heap

Relocation Protection

No Protection
Sharing

Logical Name: Space 1

Logical Name: Space 2
Segmented Addressing

Translation Lookaside Buffer:
Small, Fully associative cache
6 segment descriptors
(≈ 64 entries)

Segment Table

<table>
<thead>
<tr>
<th>Segment #</th>
<th>Offset</th>
<th>Base</th>
<th>Length</th>
</tr>
</thead>
</table>

Allowed?

Allowed?

TLB

(S, B, L)

Addr. Calc + Check

Allowed?
Segmentation Example

Ex: 32-bit logical address space
    12-bit segment number ⇒ 4096 segments
    20-bit offset ⇒ max segment size = 1 Mbyte

Process 0:

Seg 0
    ├── DLL
    │    └── 0x00100000
    ├── 0x00200000
    └── 0x00300000
    └── 0x00400000

Process 1:

Seg 0
    ├── Seg 1
    │    └── 0x00100000
    ├── Seg 2
    │    └── 0x00200000
    └── Seg 3
        └── DLL
Resource Management

Given a name, where should the name reside in memory? or, "What parts of a name space should I keep in memory?"

Problems:
① Physical memory is finite (i.e., small)
② Multiple name spaces may not fit in memory
③ Each name space may not fit in memory

Solution: Virtual Memory

Keep most of name spaces in secondary storage (disk) and move "important" portions into physical memory automatically.

Solve 2 problems:
- Mapping (relocation)
- Management

(very similar to hardware caches)
Memory Fragmentation
using variable-sized (segments) units

As segments come and go, the storage is "fragmented"; therefore, at some point segments must be moved around to compact storage ⇒ "burping the memory"
Page Addressing
Paged Addressing

Virtual Address

Virtual Page # Offset

Dirty PFN or DPN

PFN Offset

Physical Address

Translation Lookaside Buffer:
Small, fully associative cache of PTEs (~64 entries)
<table>
<thead>
<tr>
<th></th>
<th>Caching</th>
<th>Paging</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block Size</strong></td>
<td>Cache block ~16-128 bytes</td>
<td>Page ~4K-64K bytes</td>
</tr>
<tr>
<td><strong>Miss Rate</strong></td>
<td>Cache Miss Rate 1-20%</td>
<td>Page Miss Rate 0.00001-0.001%</td>
</tr>
<tr>
<td><strong>Hit Time</strong></td>
<td>Cache Hit Time 1 cycle</td>
<td>Page Hit Time ~100 cycles</td>
</tr>
<tr>
<td><strong>Miss Penalty</strong></td>
<td>Cache Miss Penalty ~100 cycles</td>
<td>Page Fault ~10^6 cycles</td>
</tr>
<tr>
<td><strong>Organization</strong></td>
<td>Direct-Mapped, Set-Associative, Fully Associative</td>
<td>Fully Associative</td>
</tr>
<tr>
<td><strong>Write Hit Policy</strong></td>
<td>Write-through or Write back</td>
<td>Write back</td>
</tr>
<tr>
<td><strong>Write Miss Policy</strong></td>
<td>Write allocate or Write no-allocate</td>
<td>Write allocate</td>
</tr>
</tbody>
</table>
Page Frame Management

- Allocate some number of page frames to each process.
- Maintain free page list. Evict pages when free pages falls below "low watermark". Evict pages from a process using LRU replacement policy.
- If dirty bit is clear, don’t copy page back to disk.

How many page frames should each process get?
Monitor page fault frequency for each process

- Try to keep working set of each process in memory
  - Page fault frequency above some upper limit
    ⇒ Increase page frame allocation.
  - Page fault frequency below some lower limit
    ⇒ Decrease page frame allocation.
- "Swap out" entire process if there are insufficient page frames for working set.