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

- Relocation
  - No Protection

- Relocation
  - Protection
Sharing

Logical Name: Space 1

Logical Name: Space 2
Segmented Addressing

Segment Table

S  n-s
Segment #  Offset
P  Base  Length

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

S  n-s
Segment #  Offset

TLB

Addr. Calc + Check


Segmentation Example

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

Virtual Address Space 0
(LNS 0)

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
\vdots
\end{array}
\]

Virtual Page Numbers

Virtual Address Space 1
(LNS 1)

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
\vdots
\end{array}
\]

Page Table 0

\[
\begin{array}{c}
PFN \\
DPN \\
PFN \\
\vdots
\end{array}
\]

Physical Memory

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
\vdots
\end{array}
\]

Physical Frame Numbers

Page Table 1

\[
\begin{array}{c}
PFN \\
PFN \\
DPN \\
\vdots
\end{array}
\]

Secondary Storage (Disk)
Paged Addressing

Translation Lookaside Buffer:
Small, fully associative cache of PTEs (~64 entries)
Caching vs. Paging

**Caching**
- Block Size: Cache block
  - ~16-128 bytes
- Miss Rate: Cache Miss Rate
  - 1 - 20%
- Hit Time: Cache Hit Time
  - 1 cycle
- Miss Penalty: Cache Miss Penalty
  - ~100 cycles
- Organization: Direct-Mapped, Set-Associative, Fully Associative
- Write Hit Policy: Write-through or Write Back
- Write Miss Policy: Write allocate or Write no-allocate

**Paging**
- Page
  - ~4K - 64K bytes
- Page Miss Rate
  - 0.00001 - 0.001%
- Page Hit Time
  - ~100 cycles
- Page Fault
  - ~10^6 cycles
- Fully Associative
- Write back
- Write allocate
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.