How Do Programs Use Names?

- Registers: i, t, *t

  Register addressing mode

- Code: spacesInString

  Absolute or PC-relative addressing mode

- Static data: s

  Absolute addressing mode

- Stack: i, t, return address

  Relative addressing mode

- Heap:

  Indirect addressing mode

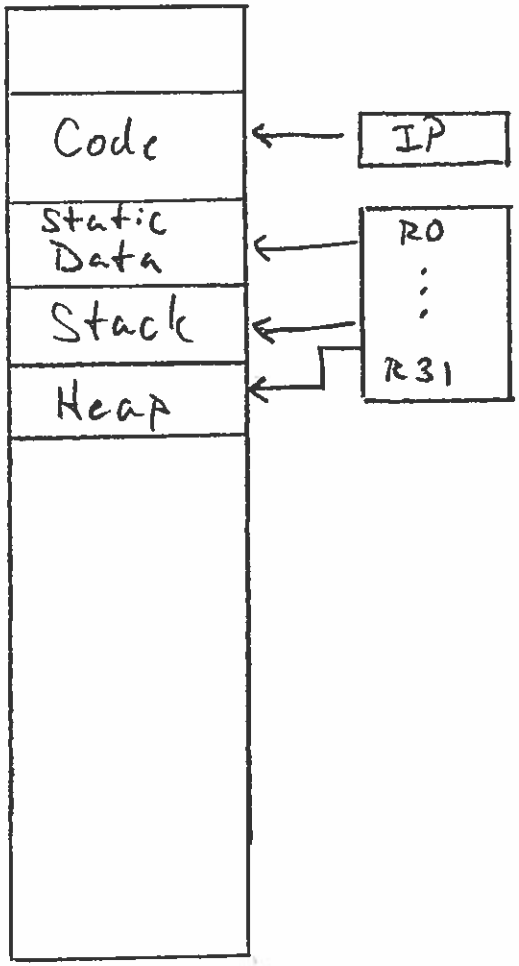
- Dynamically-linked libraries:

  Indirect addressing mode

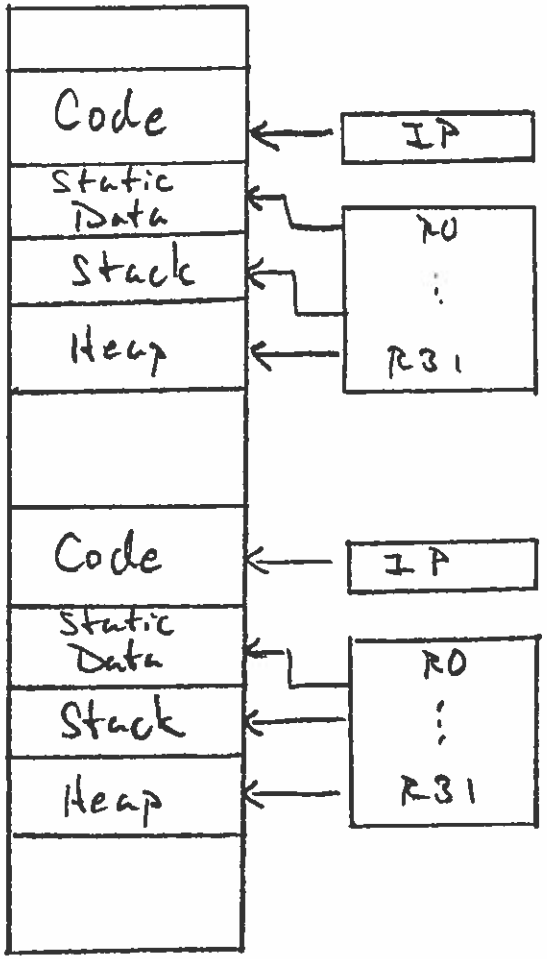
```
static char s[LEN];

int spacesInString(char *t) {
    int i=0;
    while (*t) {
        if (*t++ == ' ')
            i++;
    }
    return i;
}
```

Relocation and Protection

Single Program



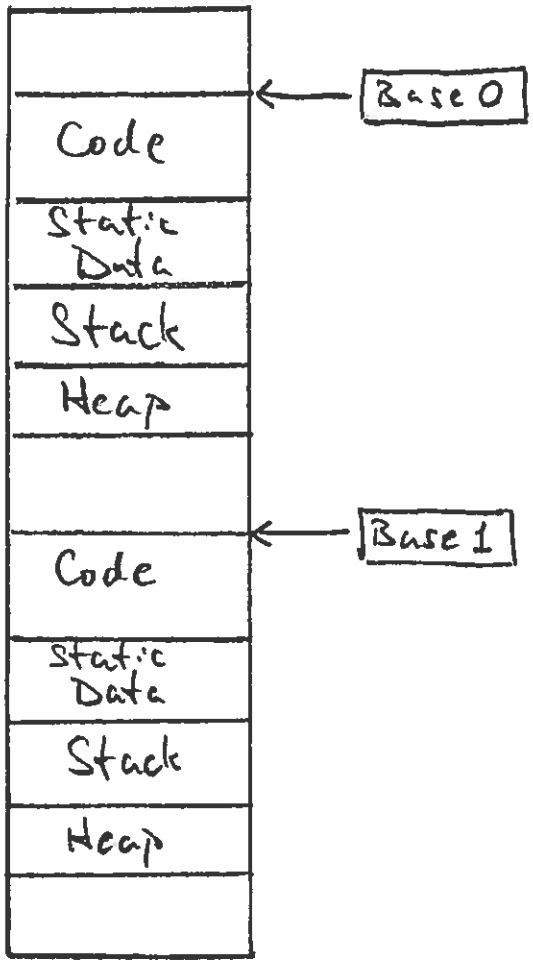
Multiple Programs



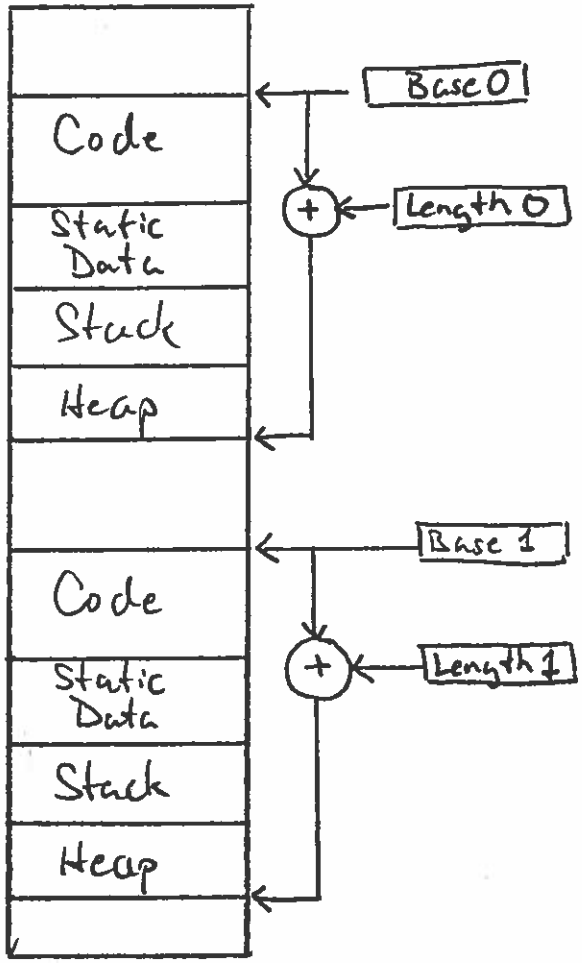
- Relocation
- No Protection

Relocation and Protection

Base Register Addressing

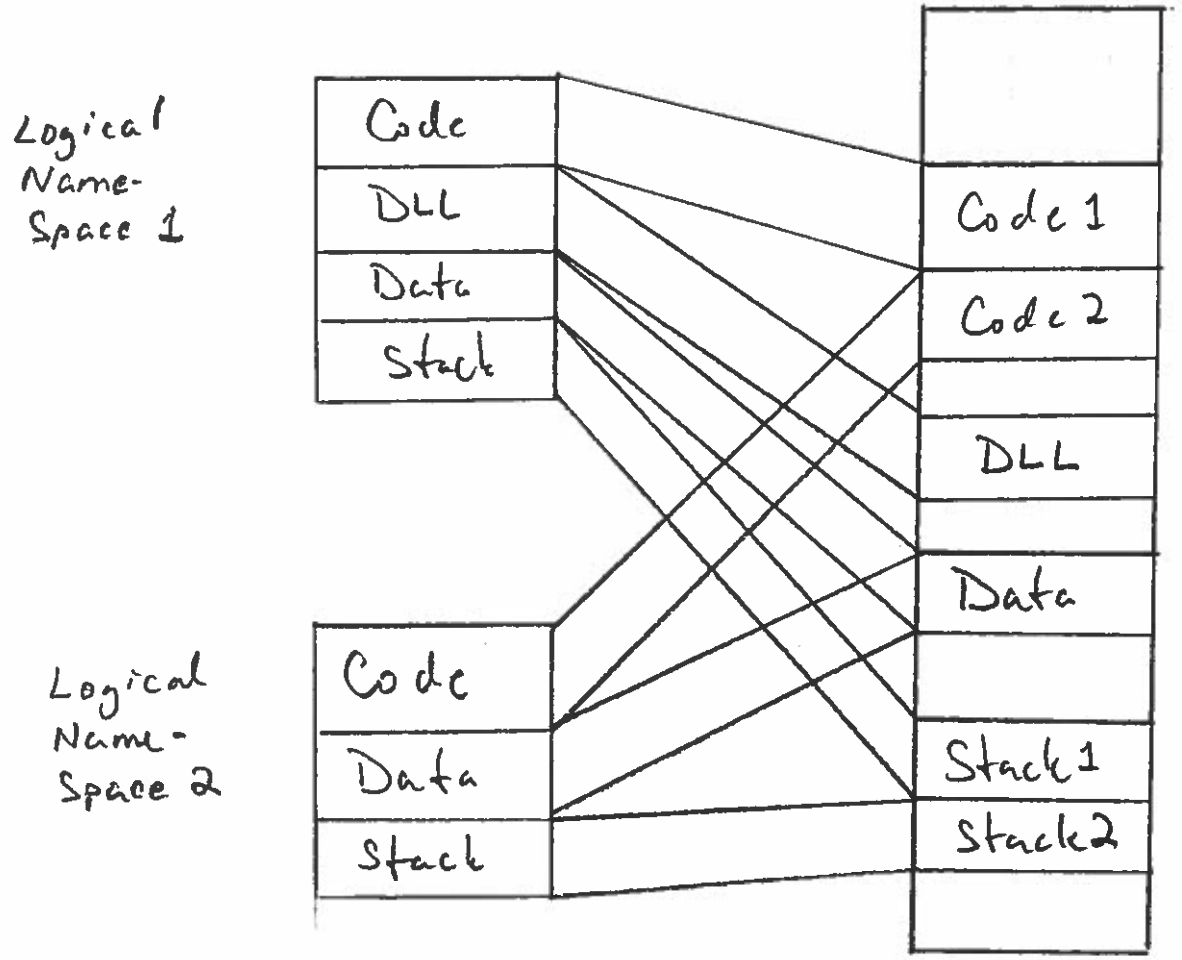


- Relocation
- No Protection

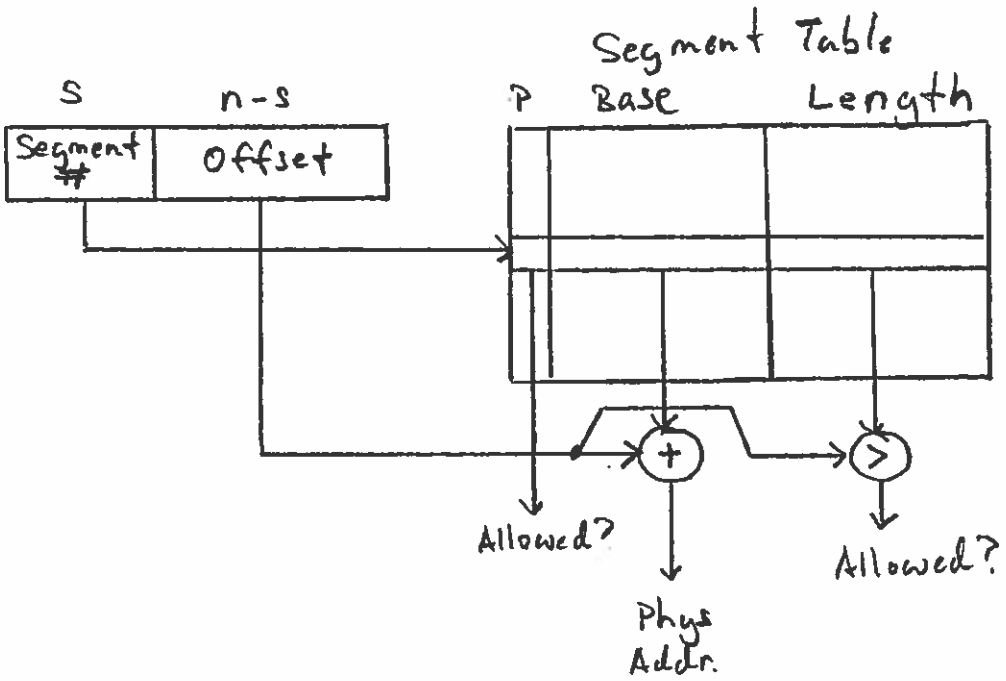


- Relocation
- Protection

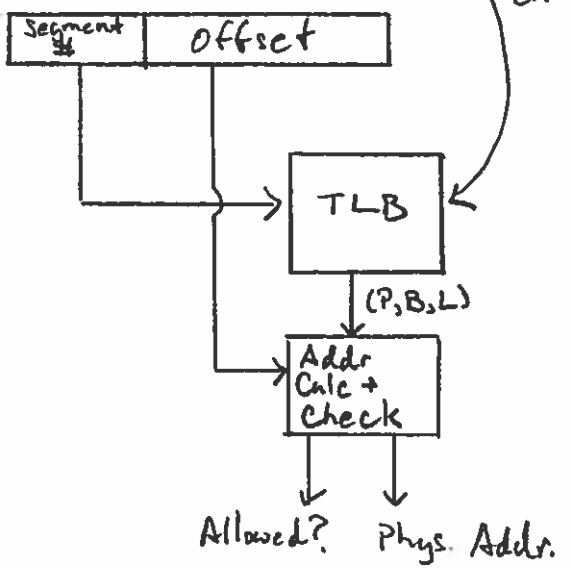
Sharing


Logical Name-Space 1
Code
DLL
Data
Stack
Logical Name-Space 2
Code
Data
Stack
Code 1
Code 2
DLL
Data
Stack 1
Stack 2

Segmented Addressing

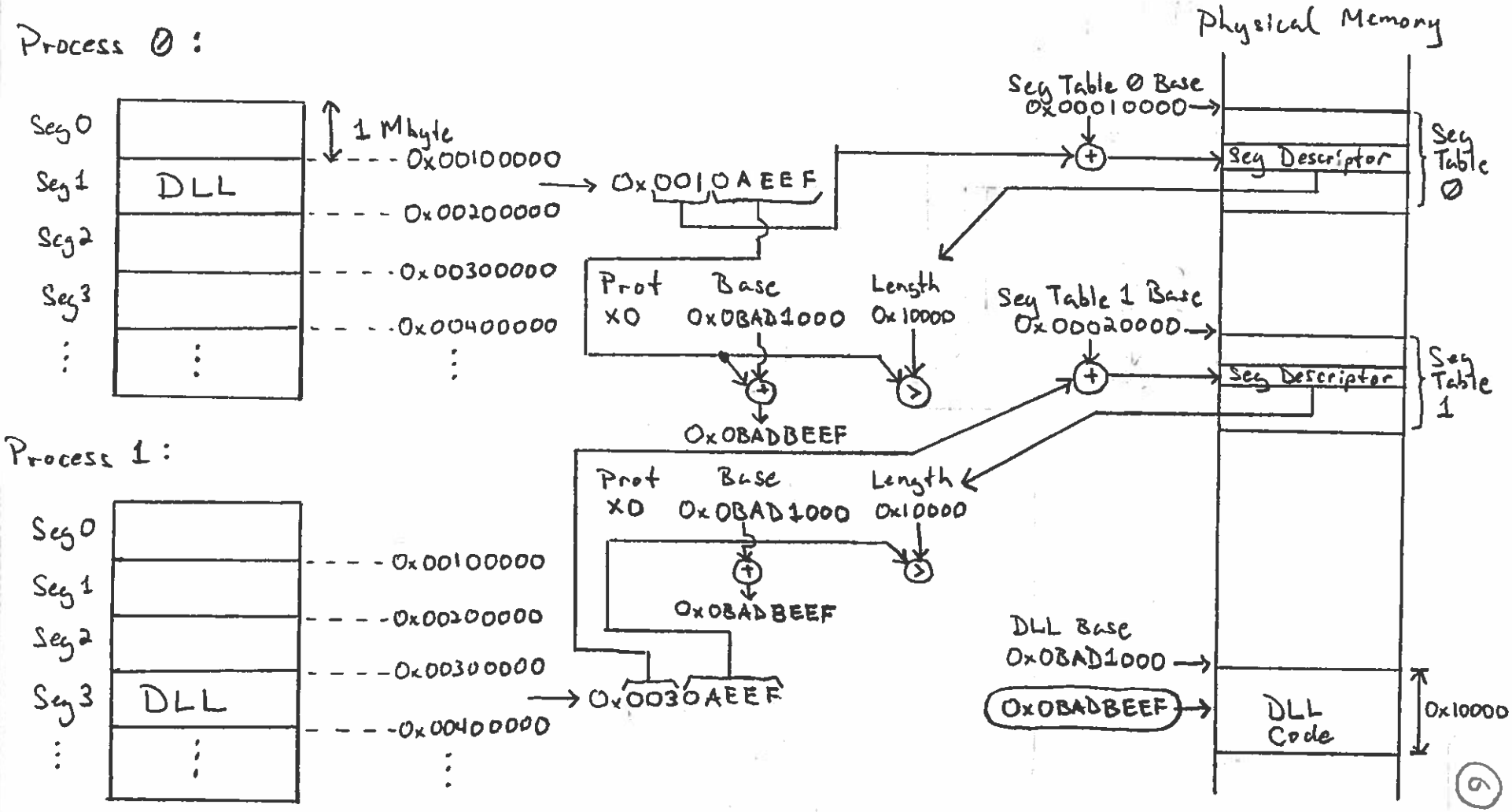


Translation Lookaside Buffer:
Small, Fully associative cache of segment descriptors (~64 entries)



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, <u>where</u> should the name reside in memory?

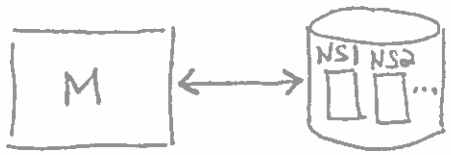
or, "What parts of a name space should I keep in memory?"

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: <u>Virtual Memory</u>

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



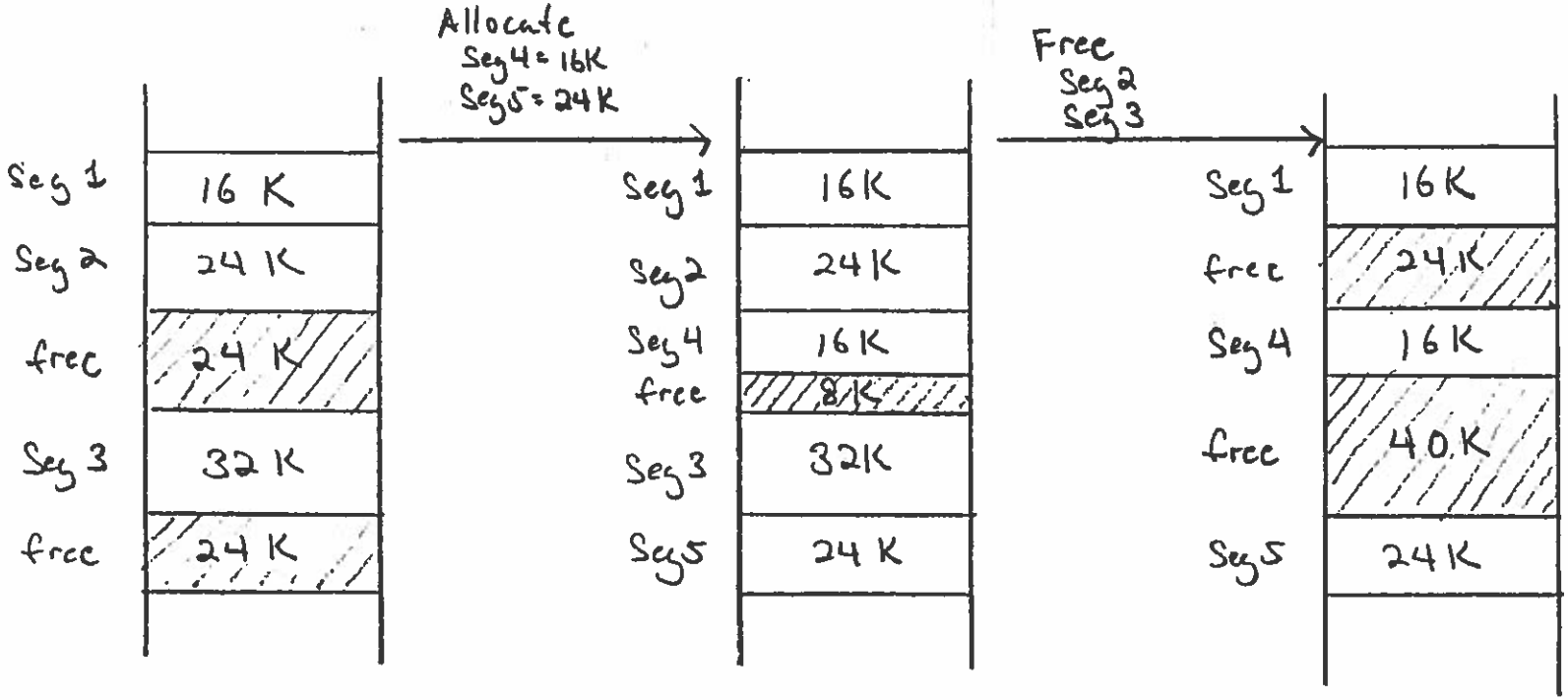
Solve 2 problems:

- Mapping (relocation)
- Management

(very similar to hardware caches)

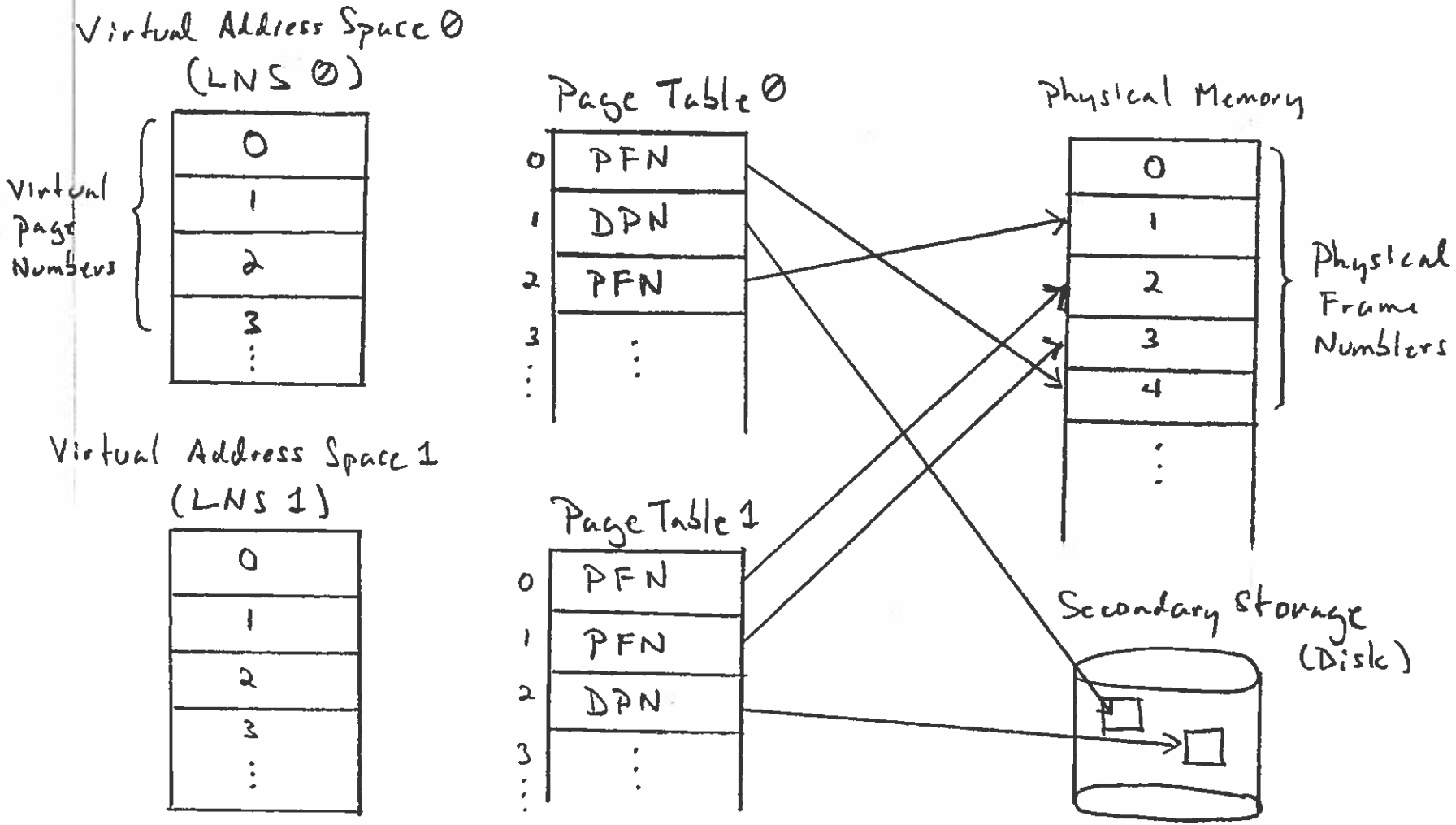
Memory Fragmentation

using variable-sized (segments) units

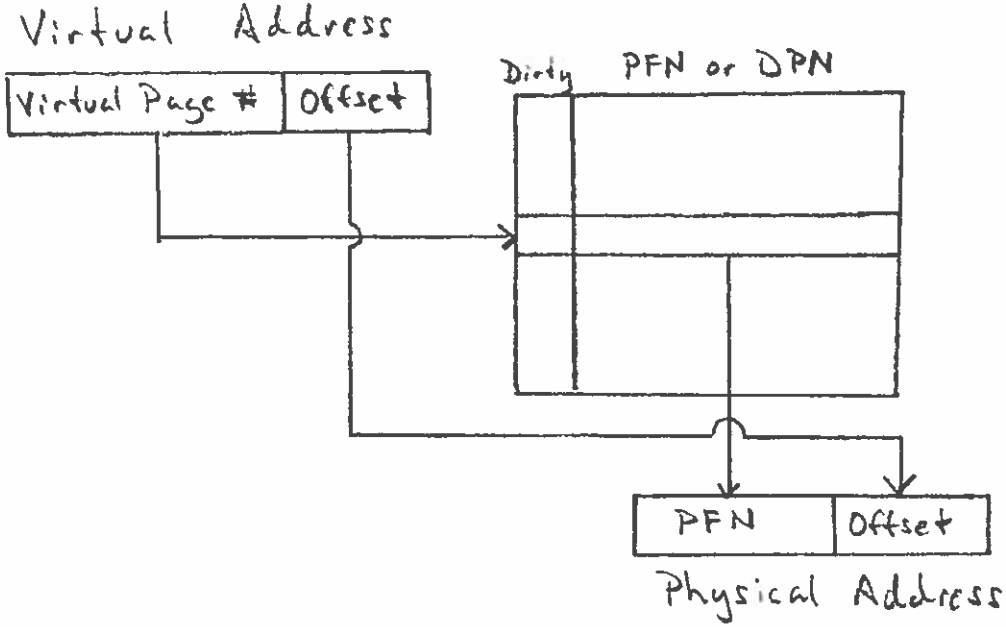


As segments come and go, the storage is "fragmented"; therefore, at some point segments must be moved around to compact storage ⇒ "burping the memory"

Paged Addressing

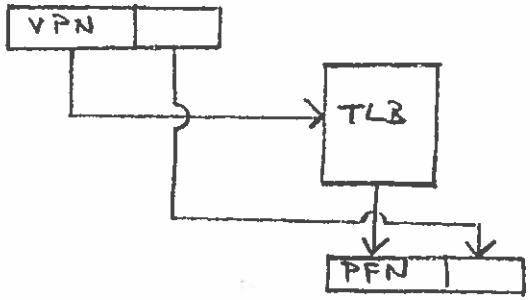


Paged Addressing



Translation Lookaside Buffer:

Small, fully associative cache of PTEs (~ 64 entries)



Caching vs. Paging

	Caching	Paging
Block Size:	Cache block ~ 16 - 128 bytes	Page ~4K - 64K bytes
Miss Rate:	Cache Miss Rate 1 - 20%	Page Miss Rate 0.00001 - 0.001%
Hit Time:	Cache Hit Time 1 cycle	Page Hit Time ~100 cycles
Miss Penalty:	Cache Miss Penalty ~100 cycles	Page Fault ~ 10^6 cycles
Organization:	Direct-Mapped, Set-Associative, Fully Associative	Fully Associative
Write Hit Policy:	Write-through or Write back	Write back
Write Miss Policy:	Write allocate or Write no-allocate	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 clean, 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.