Multi-level Page Tables & Paging+ segmentation combined

- Basic idea: use two levels of mapping to make tables manageable.
    - Each segment contains one or more pages.
- Segments correspond to logical units: code, data, stack.
- Segments vary in size and are often large, while pages are fixed size.
- Pages are for the use of the OS; they are fixed-sized to make it easy to manage memory.

Going from paging to P+S is like going from single segment to multiple segments, except at a higher level. Instead of having a single page table, have many page tables with a base and bound for each (base and bounds now refer to page table's base and bound). Call the stuff associated with each page table a segment.

Show System 370 example: 24-bit virtual address space as follows:

segment # (4 bits)	page # (8 bits)	offset (12 bits)

Segment table contains real address of page table along with the length of the page table (a sort of bounds register for the segment). Page table entries are only 12 bits, real addresses are 24 bits.

Example of S/370 paging (all numbers in **hexadecimal**):

Segment table (base|bounds|protection): 2000|14|R, 0|0|0, 1000|D|RW.

Segement #	Base	Bound	RW bits
0	2000	14	10
1	0	0	00
2	1000	D	01

Assume page table entries are each 2 bytes long.

- What is the physical address of the page table entry for translating virtual address 2070?
- Suppose the physical memory where the page tables for the above segments reside contain the following:

Physical address	Contents
1000	06
1002	0B
1004	04
.	.
.	.
1020	07
1022	0C
.	.
.	.
2000	13
2002	2A
2004	03
.	.
.	.
2020	11
2022	1F

Map the following addresses from virtual to physical:

- 2070 read:
- 202016 read:
- 104C84 read: 11424 read:
- 210014 write:

Pros:

- If a segment isn't used, then there's no need to even have a page table for it.
- Can share at two levels: single page, or single segment (whole page table)
- Pages eliminate external fragmentation, and make it possible for segments to grow without any reshuffling.
- If page size is small compared to most segments, then internal fragmentation is not too bad.

Cons:

- If translation tables are kept in main memory, overheads could be very high: 1 or 2 overhead references for every real reference.
- Page tables can still be large.

Dealing with very large page tables

Another problem with large address spaces is very large page tables.
Solution:

- Don’t keep the entire page table in memory. Use virtual memory for the page tables as well.

Multi-level paging

Basically, like segmentation+paging, except it could be 2 or more levels. There is no notion such as segments mapping to logical entities. Show picture:

Only the top level page table needs to be resident in memory. If the present bit for the next-level page table is false, then it can be paged in. At a given time, only a small part of entire set of page tables needs to be resident in memory.

Alternative way of dealing with large page tables: Map user page tables via an in-memory system page table

VAX 11/780 used this.

- Address is 32 bits, top two select segment. Four base-bound pairs define page tables (system, P0, P1, unused).

- Pages are 512 bytes long.

First segment contains operating system stuff, two contain stuff of current user process:

- Segment 0 page table is always kept in memory (system stuff, shared between processes).
- P0 and P1 point to user addresses. The page tables for P0 and P1 are kept in system's *virtual memory. Show a picture.*

Effectively, we are using the system page table to map the user page tables so that the user page tables can be scattered or even moved to disk. Translating a user address requires:

- Compute the address of the user page table entry. This is a *virtual address* that is guaranteed to be in segment 0 (system's space).
- Translate the PTE address to a physical address. This is normal translation, via the system's page table. A page fault could occur here, in which it needs to be serviced to bring the PTE into memory.
- Look up the PTE and complete the translation. A page fault could also occur here, in which case it needs to be serviced to bring the page being accessed by the user into physical memory.

System base-bounds pairs are physical addresses, system tables must be contiguous.

The result is a two-level scheme.

Problem with multi-level paging:

Extra memory references to access translation tables can slow programs down by a factor
of two or more. Solution: Use translation look-aside buffers (TLB) to speed up address translation. (EECS 370 material)

Kernel vs. user mode

Who is allowed to modify the base+bound registers? Or more generally, the translation data used by the translation box?

- Can user processes be allowed to modify the translation data?

- If only kernel is allowed to modify the translation data, where is the translation data stored (whose address space?) and how does kernel modify the data?

- How does machine know that the kernel is running?

- How do we change from user mode to kernel mode? Can the user process change it?

- How do we switch from user process to kernel?

- How does the OS regain control once it has given control to the user process?
- How do we create and start a process? (As opposed to just starting a thread)
- Is hardware support needed for any of the above steps?