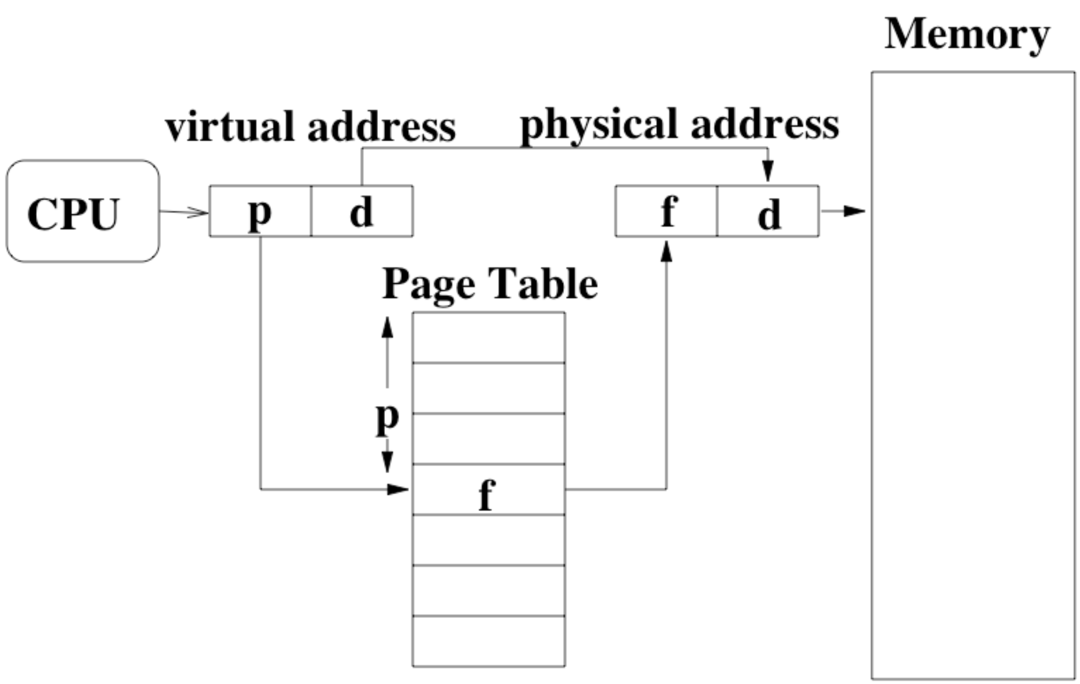
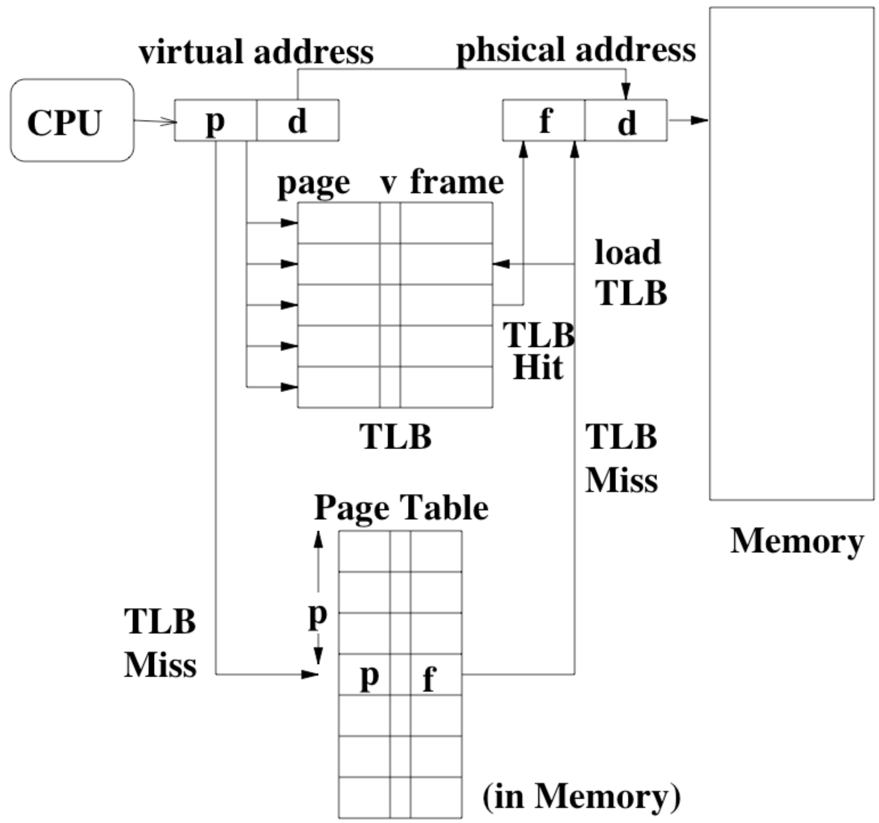
Last Class: Paging

- Process generates virtual addresses from 0 to Max.
- OS divides the process onto pages; manages a page table for every process; and manages the pages in memory
- Hardware maps from virtual addresses to physical addresses.



The Translation Look-aside Buffer (TLB)



v: valid bit that says the entry is up-to-date

Costs of Using The TLB

- What is the effective memory access cost if the page table is in memory?

- What is the effective memory access cost with a TLB?

A large TLB improves hit ratio, decreases average memory cost.

Costs of Using The TLB

- What is the effective memory access cost if the page table is in memory?
  - ema = 2 * ma
- What is the effective memory access cost with a TLB?
  - ema = (ma + TLB) * p + (2ma + TLB) * (1-p)

A large TLB improves hit ratio, decreases average memory cost.

Initializing Memory when Starting a Process

1. Process needing *k* pages arrives.
2. If *k* page frames are free, then allocate these frames to pages. Else free frames that are no longer needed.
3. The OS puts each page in a frame and then puts the frame number in the corresponding entry in the page table.
4. OS marks all TLB entries as invalid (flushes the TLB).
5. OS starts process.
6. As process executes, OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full.

Saving/Restoring Memory on a Context Switch

- The Process Control Block (PCB) must be extended to contain:
  - The page table
  - Possibly a copy of the TLB
- On a context switch:
  1. Copy the page table base register value to the PCB.
  2. Copy the TLB to the PCB (optionally).
  3. Flush the TLB.
  4. Restore the page table base register.
  5. Restore the TLB if it was saved.
- **Multilevel Paging:** If the virtual address space is huge, page tables get too big, and many systems use a multilevel paging scheme (refer OSC for details)

Sharing

Paging allows sharing of memory across processes, since memory used by a process no longer needs to be contiguous.

- Shared code must be reentrant, that means the processes that are using it cannot change it (e.g., no data in reentrant code).
- Sharing of pages is similar to the way threads share text and memory with each other.
- A shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address.
- The user program (e.g., emacs) marks text segment of a program as reentrant with a system call.
- The OS keeps track of available reentrant code in memory and reuses them if a new process requests the same program.
- Can greatly reduce overall memory requirements for commonly used applications.

Today: Segmentation

Segments take the user's view of the program and gives it to the OS.

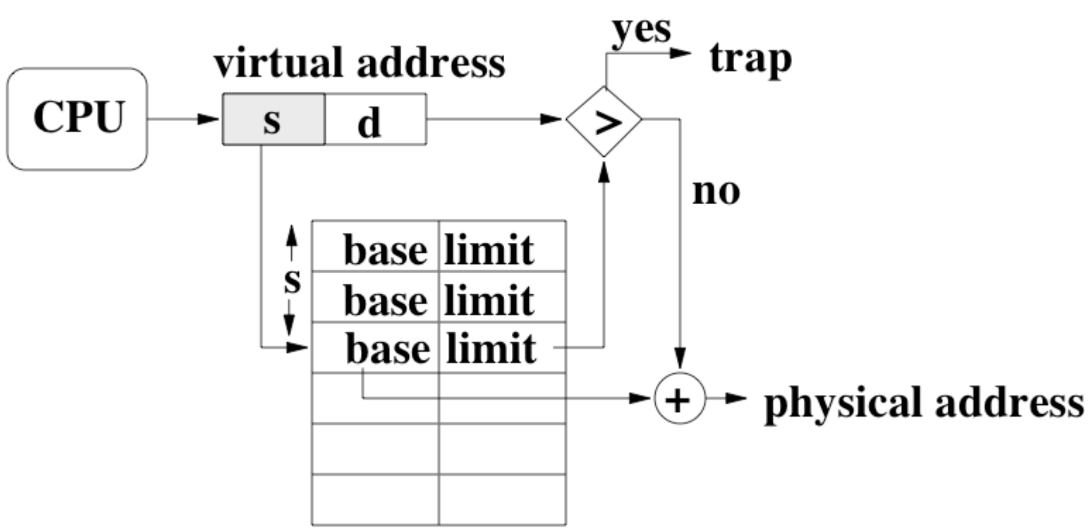
- User views the program in logical *segments*, e.g., code, global variables, stack, heap (dynamic data structures), not a single linear array of bytes.
- The compiler generates references that identify the segment and the offset in the segment, e.g., a code segment with offset = 399
- Thus processes thus use virtual addresses that are segments and segment offsets.

⇒ Segments make it easier for the call stack and heap to grow dynamically. Why?

⇒ Segments make both sharing and protection easier. Why?

Implementing Segmentation

- Segment table: each entry contains a base address in memory, length of segment, and protection information (can this segment be shared, read, modified, etc.).
- Hardware support: multiple base/limit registers.



Implementing Segmentation

- Compiler needs to generate virtual addresses whose upper order bits are a segment number.
- Segmentation can be combined with a dynamic or static relocation system,
  - Each segment is allocated a contiguous piece of physical memory.
  - External fragmentation can be a problem again
- Similar memory mapping algorithm as paging. We need something like the TLB if programs can have lots of segments

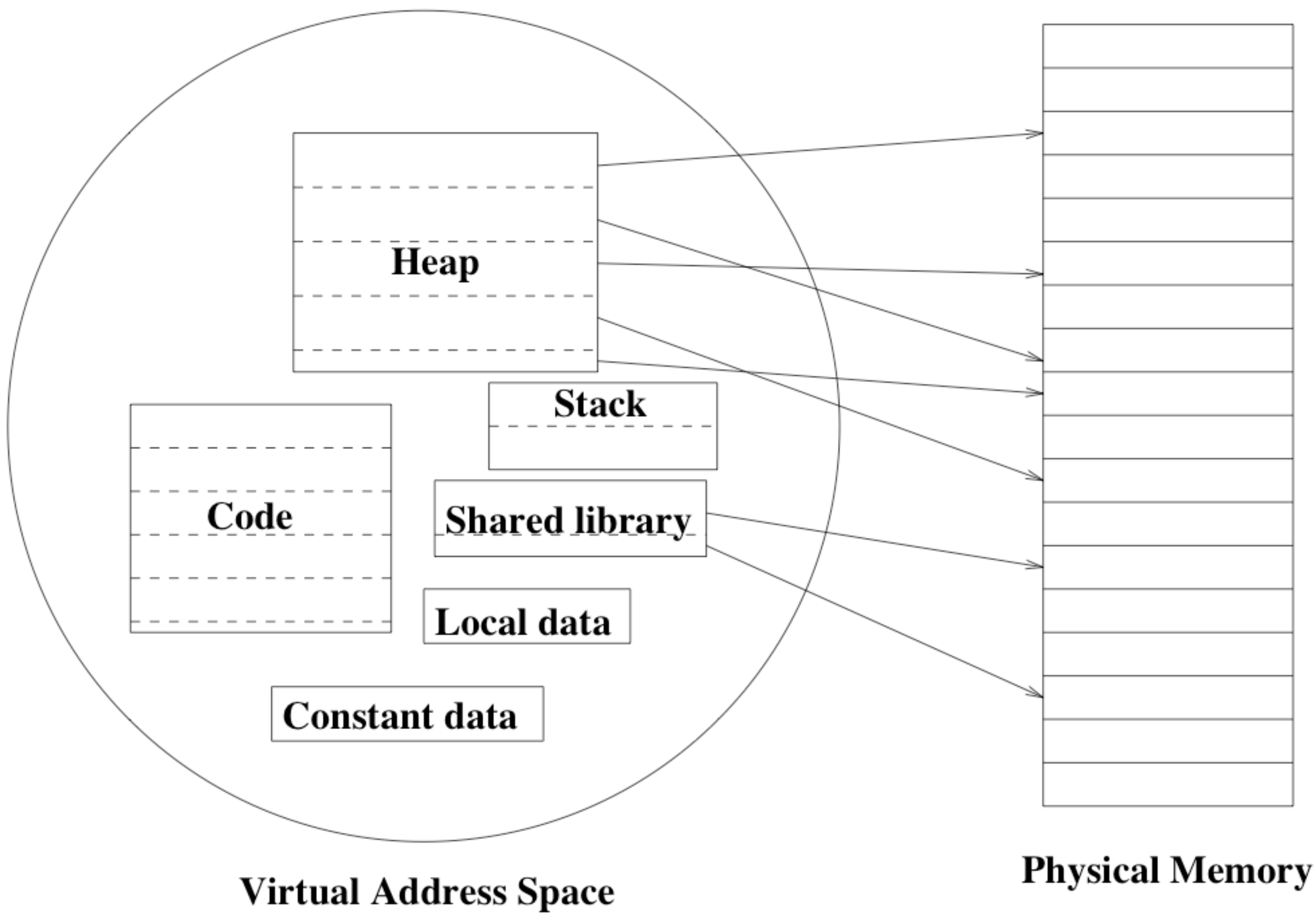
- **Let's combine the ease of sharing we get from segments with efficient memory utilization we get from pages.**

Combining Segments and Paging

- Treat virtual address space as a collection of segments (logical units) of arbitrary sizes.
- Treat physical memory as a sequence of fixed size page frames.
- Segments are typically larger than page frames,

⇒ Map a logical segment onto multiple page frames by paging the segments

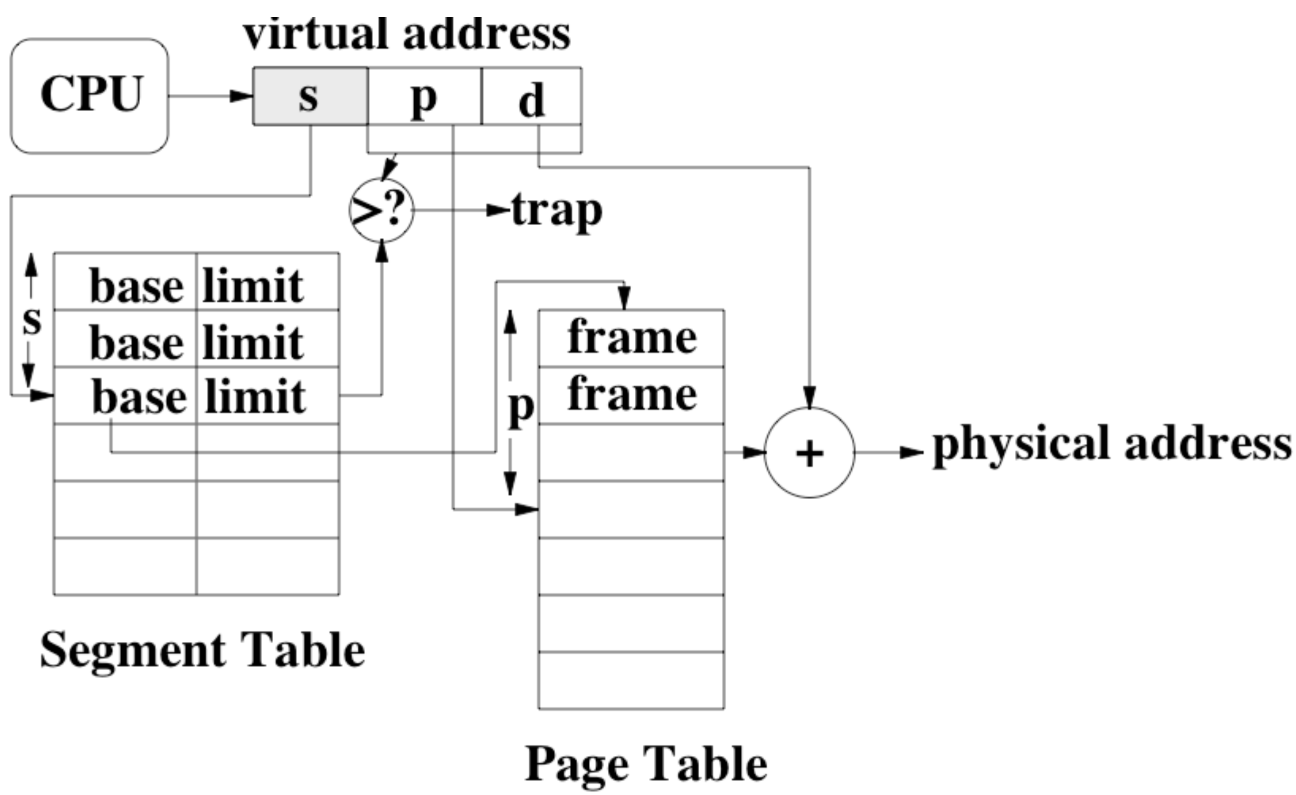
Combining Segments and Paging



Addresses in Segmented Paging

- A virtual address becomes a segment number, a page within that segment, and an offset within the page.
- The segment number indexes into the segment table which yields the base address of the page table for that segment.
- Check the remainder of the address (page number and offset) against the limit of the segment.
- Use the page number to index the page table. The entry is the frame. (The rest of this is just like paging.)
- Add the frame and the offset to get the physical address.

Addresses in Segmented Paging



Addresses in Segmented Paging: Example

- Given a memory size of 256 addressable words,
- a page table indexing 8 pages,
- a page size of 32 words, and
- 8 logical segments

- How many bits is a physical address?
- How many bits is a virtual address?
- How many bits for the seg #, page #, offset?
- How many segment table entries do we need?
- How many page table entries do we need?

Sharing Pages and Segments

- Share individual pages by copying page table entries.
- Share whole segments by sharing segment table entries, which is the same as sharing the page table for that segment.
- Need protection bits to specify and enforce read/write permission.
  - When would segments containing code be shared?
  - When would segments containing data be shared?

Sharing Pages and Segments: Implementation Issues

- Where are the segment table and page tables stored?
  - Store segment tables in a small number of associative registers; page tables are in main memory with a TLB (faster but limits the number of segments a program can have)
  - Both the segment tables and page tables can be in main memory with the segment index and page index combined used in the TLB lookup (slower but no restrictions on the number of segments per program)
- Protection and valid bits can go either on the segment or the page table entries

- **Note:** Just like recursion, we can do multiple levels of paging and segmentation when the tables get too big.

Segmented Paging: Costs and Benefits

- **Benefits:** faster process start times, faster process growth, memory sharing between processes.
- **Costs:** somewhat slower context switches, slower address translation.
- Pure paging system => (virtual address space)/(page size) entries in page table. How many entries in a segmented paging system?
- What is the performance of address translation of segmented paging compared to contiguous allocation with relocation? Compared to pure paging?
- How does fragmentation of segmented paging compare with contiguous allocation? With pure paging?

Inverted Page Tables

- Techniques to scale to very large address spaces
- Multi-level page tables
- Inverted index
  - Page table is a hash table with key-value lookups
    - Key=page number, value = frame number
    - Page table lookups are slow
    - Use TLBs for efficiency

Putting it all together

- **Relocation** using Base and Limit registers
  - simple, but inflexible
- **Segmentation:**
  - compiler's view presented to OS
  - segment tables tend to be small
  - memory allocation is expensive and complicated (first fit, worst fit, best fit).
  - compaction is needed to resolve external fragmentation.