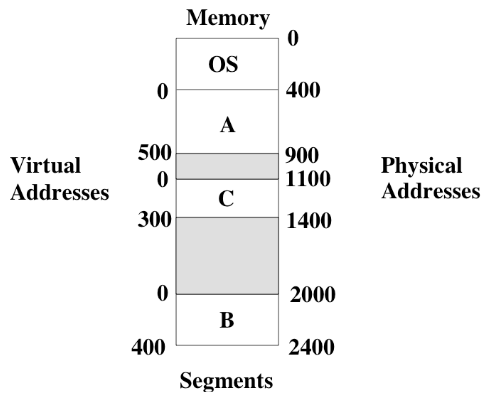
Last Class: Memory Management

- Static & Dynamic Relocation
- Fragmentation
- Paging



Recap: Paging

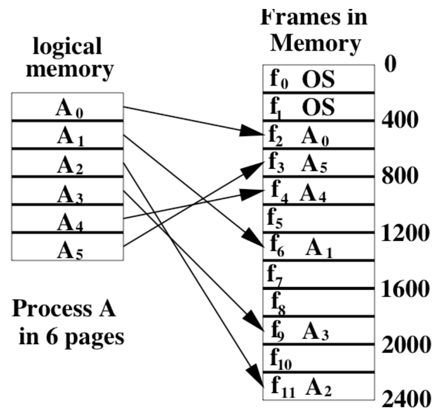
Processes typically do not use their entire space in memory all the time.

Paging

1. divides and assigns processes to fixed sized *pages*,
2. then selectively allocates pages to *frames* in memory, and
3. manages (moves, removes, reallocates) pages in memory.

Paging: Example

Mapping pages in logical memory to frames in physical memory

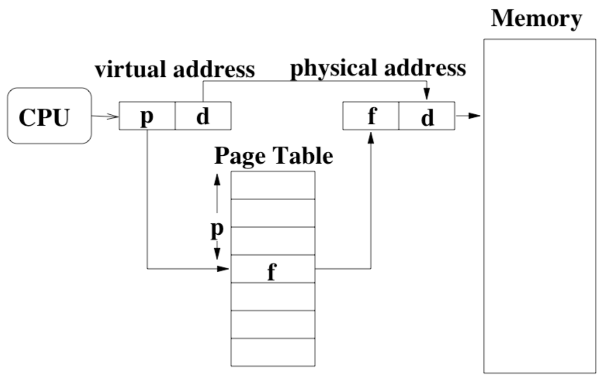


Paging Hardware

- **Problem:** How do we find addresses when pages are not allocated contiguously in memory?
- **Virtual Address:**
  - Processes use a virtual (logical) address to name memory locations.
  - Process generates contiguous, virtual addresses from 0 to size of the process.
  - The OS lays the process down on pages and the paging hardware translates virtual addresses to actual physical addresses in memory.
  - In paging, the virtual address identifies the page and the page offset.
  - *page table* keeps track of the page frame in memory in which the page is located.

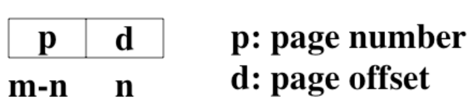
Paging Hardware

Translating a virtual address to physical address

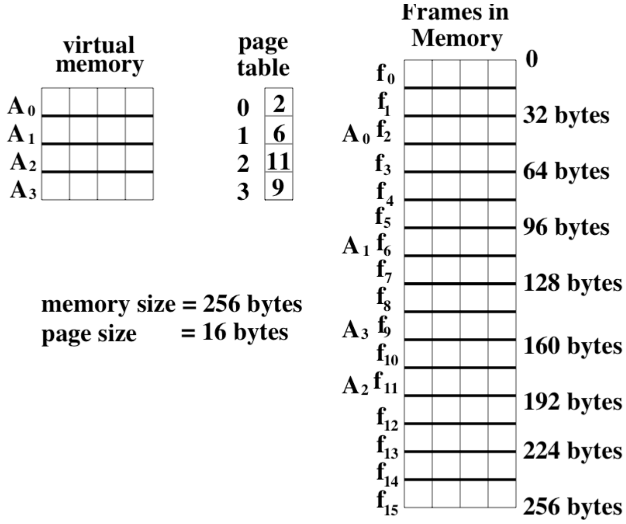


Paging Hardware: Practical Details

- Page size (frame sizes) are typically a power of 2 between 512 bytes and 8192 bytes per page.
- Powers of 2 make the translation of virtual addresses into physical addresses easier. For example, given
- virtual address space of size 2^m bytes and a page of size 2^n, then
- the high order *m-n* bits of a virtual address select the page,
- the low order *n* bits select the offset in the page

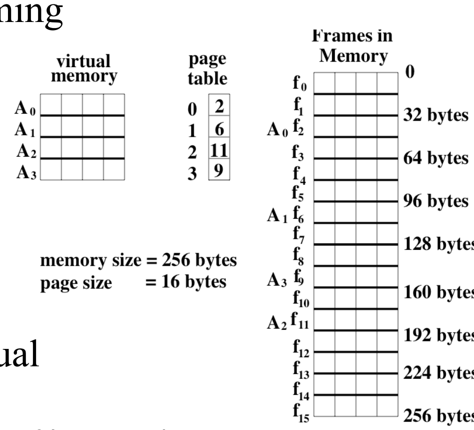


Address Translation Example



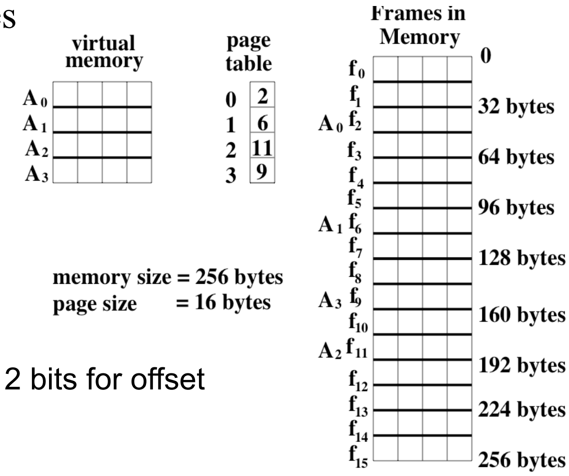
Address Translation Example

- How big is the page table?
  - 16 entries (256 memory bytes / 16 byte pages)
- How many bits for an address, assuming we can address 1 byte increments?
  - 8 bits (to address 256 bytes)
- What part is p, and d?
  - 4 bits for page and 4 for offset
- Given virtual address 24, do the virtual to physical translation.
  - page p=1 (24 / 16 = 1), offset d=8 (24 % 16 = 8)
  - frame f=6 (from page table), offset d=8



Address Translation Example 2

- Typical 32-bit architecture: address **words** (4 bytes) rather than bytes



- How many bits for an address?
  – 6 bits (64 addresses of 4-byte words in 256 byte memory space)
- What part is p, and d?
  – 4 bits for for page (still 16 pages), 2 bits for offset
- Given virtual address 13, do the virtual to physical translation.
  – p=3 (13 words / 4-word pages), d=1 (13 % 4)
  – Frame=9, d=1

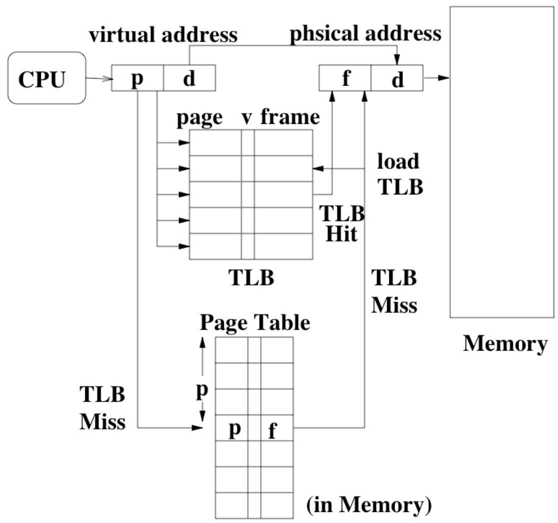
Making Paging Efficient

How should we store the page table?

- **Registers:** Advantages? Disadvantages?
- **Memory:** Advantages? Disadvantages?

- **TLB (translation look-aside buffer):** a fast fully associative memory that stores page numbers (key) and the frame (value) in which they are stored.
  – if memory accesses have locality, address translation has locality too.
  – typical TLB sizes range from 8 to 2048 entries.

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?
  – ema = 2 * ma
- What is the effective memory access cost with a TLB?
  – TLB hit ratio p, TLB access cost
  – ema = (ma + TLB) * p + (2 * ma + 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.

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.

Paging Summary

- Paging is a big improvement over relocation:
  - They eliminate the problem of external fragmentation and therefore the need for compaction.
  - They allow sharing of code pages among processes, reducing overall memory requirements.
  - They enable processes to run when they are only partially loaded in main memory.
- However, paging has its costs:
  - Translating from a virtual address to a physical address is more time-consuming.
  - Paging requires hardware support in the form of a TLB to be efficient enough.
  - Paging requires more complex OS to maintain the page table.

Segmentation

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

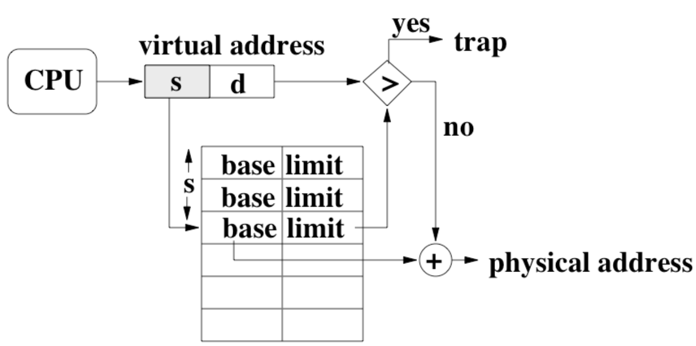
- User views the program in logical *segments*, e.g., code, global variables, stack, heap (dynamic data structures), not a single linear array of bytes.
- The compiler generates references that identify the segment and the offset in the segment, e.g., a code segment with offset = 399
- Thus processes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

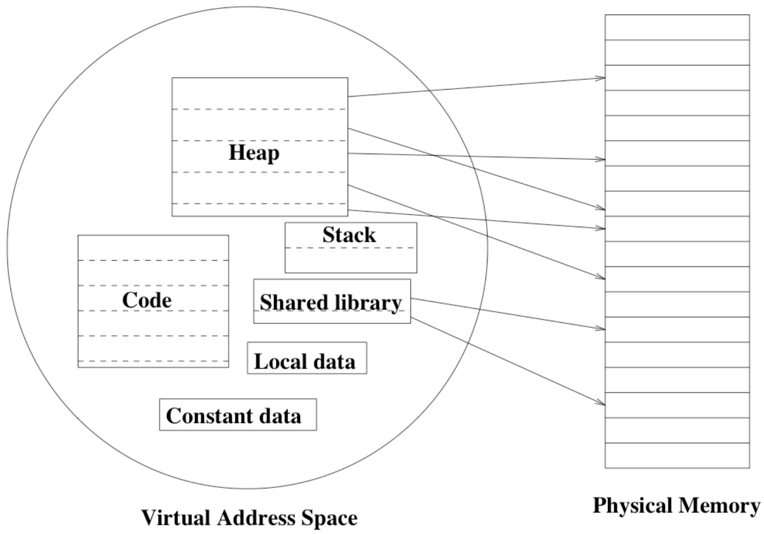
Let's combine the ease of sharing we get from segments with efficient memory utilization we get from pages.

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

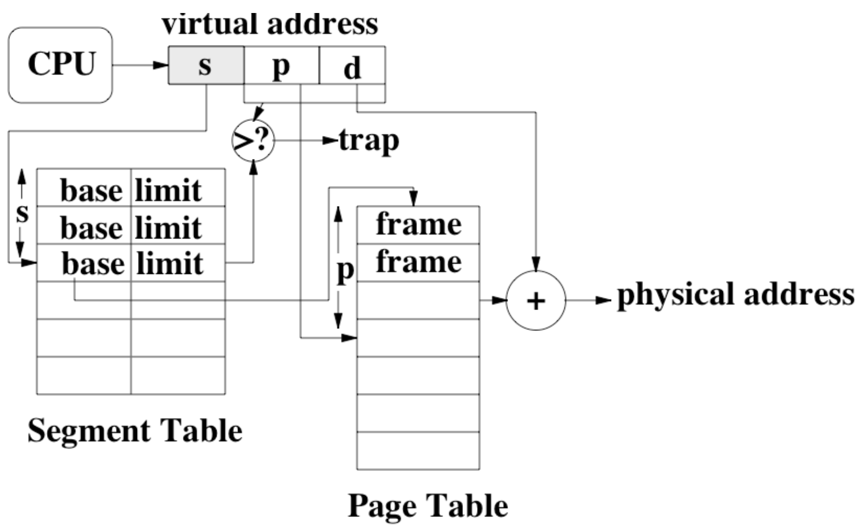
Combining Segments and Paging



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

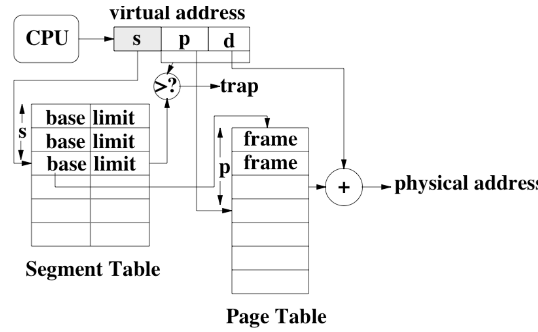


Sharing Pages and Segments: Implementation Issues

- 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:** Can do multiple levels of paging and segmentation when the tables get too big.

Addresses in Segmented Paging: Example

- Memory size of 256 addressable words (assume word=byte),
- a page table indexing 8 pages,
- a page size of 32 words, and
- 8 logical segments



- How many bits is a physical address?
  8 bits (256 addresses)
- How many bits is a virtual address?
  11 bits (3+3+5)
- How many bits for the seg #, page #, offset?
  3 seg, 3 page, 5 offset

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?

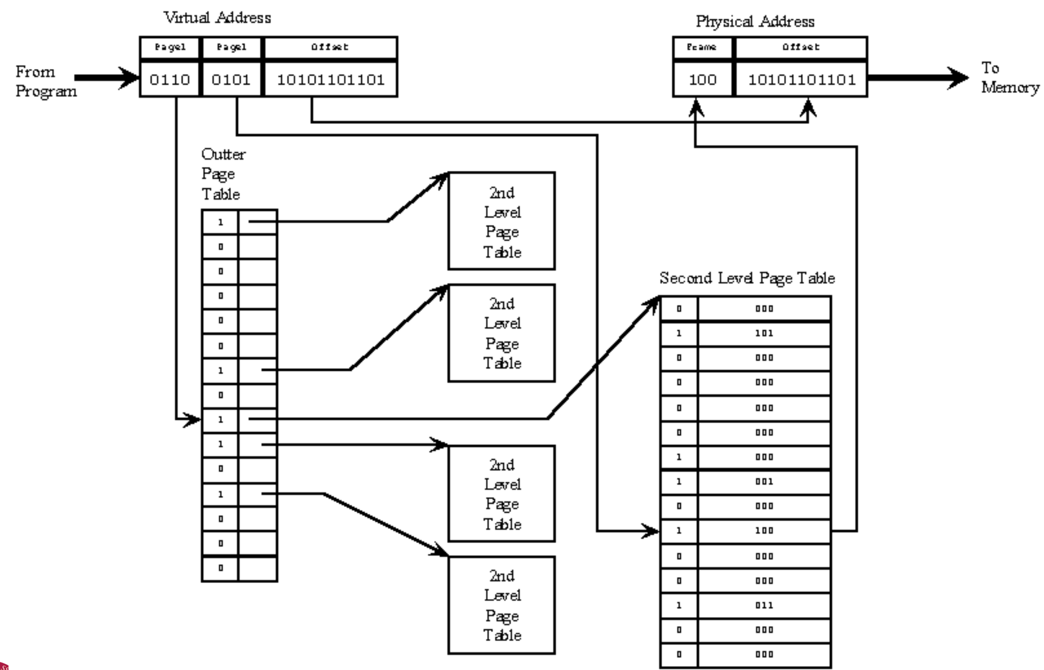
Segmented Paging: Costs and Benefits

- **Benefits:** merge compiler + OS view of memory, flexible process growth, no external fragmentation, 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?

Large Address Spaces

- Techniques to scale to very large address spaces
  - 64 bit machines
- Sparse page tables
- Inverted page table
  - Hash table with key-value lookups (instead of array)
  - Key = page number, value = frame number
  - Page table lookups are slow
  - Use TLBs for efficiency

Multilevel Page Tables



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.

Putting it all together

- **Paging:**
  - simplifies memory allocation since any page can be allocated to any frame
  - page tables can be very large (especially when virtual address space is large and pages are small)

- **Segmentation & Paging**
  - only need to allocate as many page table entries as we need (large virtual address spaces are not a problem).
  - easy memory allocation, any frame can be used
  - sharing at either the page or segment level
  - increased internal fragmentation over paging
  - two lookups per memory reference