Lecture Overview

- Memory management (Part 2)
  - Paging implementation
    - Page table
    - Translation look-aside buffers
    - Multilevel page tables
    - Inverted pages tables
  - Segmentation

Memory Management using Pages

- Pages solve external fragmentation problems associated with contiguous memory allocation schemes
- Pages allow the operating system to load programs into non-contiguous memory blocks
  - The physical and logical address spaces are divided into small blocks, called frames and pages respectively
  - Processes work with logical addresses which are automatically mapped into physical addresses by the MMU

Paging Implementation

- Since address translation must occur for every address reference, translation must be efficient
- A process' logical address is divided into pages and each page maps to a physical page frame, it is necessary to keep a page table that maps pages to frames
  - Generally, the page table is per process and kept in the process descriptor
  - When a process is scheduled, just like its registers and program counter, the OS must correctly set up the process' page table (which may include setting special register values)

Hardware Support for Paging

- The simplest case is to implement the page table as a dedicated set of registers
  - Very quick
  - These register need to be saved and loaded each time we switch between executing processes
  - Only good for small page tables (< 256 entries), too expensive for large page tables (typical page tables might have up to 1 million entries)

Hardware Support for Paging

- Large page tables must be kept in main memory
- A *page table base register (PTBR)* points to page table
  - Changing page tables only requires changing this register
- This approach increases translation time though, for example, to access location *i*
  - Must first index into page table using PTBR + page number from *i* (this is one memory access)
  - The resulting frame number is combined with the page offset from *i* to produce the physical address, then we can access the memory (this is another memory access)
  - This means that it takes two memory accesses to access all memory locations, slowing memory access by a factor of 2

Translation Look-Aside Buffers

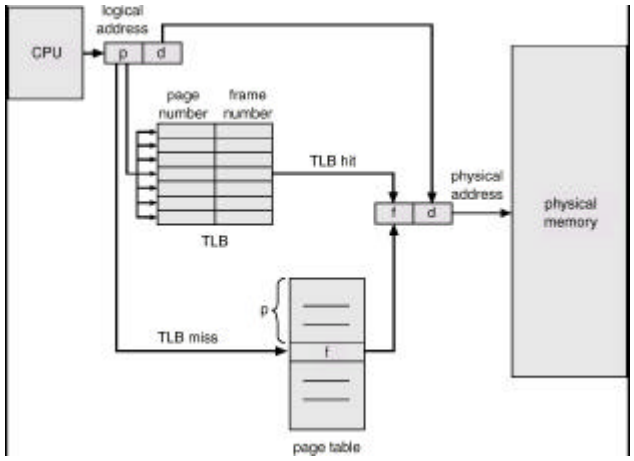
- To decrease memory access times, a standard solution is to use a set of small, fast-lookup, hardware registers called a *translation look-aside buffer (TLB)*
  - Also called *associative registers*
  - Each register in the set contains a key and a value; the key is the page number and the value is the frame number
  - When a TLB is given a key, it searches all registers (typically between 8 and 2048) in parallel and outputs the value if the key is found
  - When a frame number is found, the memory reference to the page table is eliminated and a small percentage of TLB overhead is added to the memory access (e.g., 10 percent)

Translation Look-Aside Buffers

- When a frame number is not found, it is then added to the TLB register set so that it will be found the next time
  - If the TLB is full, the OS must remove an entry
- Every time a new page table is loaded (i.e., a process switch), the TLB must be flushed
- The abstract view of a TLB is presented on the next slide

Translation Look-Aside Buffers

Abstract view of a TLB



Translation Look-Aside Buffers

- The performance of a TLB depends on its *hit ratio*
  - Hit ratio is the percentage of time that a frame number is found in the TLB
  - Associative lookup time is *a* time units
  - Memory cycle time is *m* time units
  - Hit ratio is *r*
  - *Effective Access Time (EAT)* = $(m + a)\,r + (2m + a)(1 - r)$

    Example

    If $a = 20$ ns, $m = 100$ ns, and $r = 80\%$, then

    EAT = $(100 + 20)\,0.8 + (2(100) + 20)(1 - 0.8) = 140$ ns

    For $r = 98\%$, then EAT = 122 ns

Multilevel Paging

- Modern computers support large logical address spaces (2^{32} to 2^{64})
  - With an address space of 2^{32} and a page size of 4k (2^{12}), a page table has one million entries
  - Assuming that each entry is 4 bytes, the page table is 4MB
- We need an approach to reduce the memory requirements of the page table
- One approach is to page the page table

(In truth, since we are currently assuming an entire process must be in memory to execute, this doesn't actually reduce memory usage, but when we talk about virtual memory in the next lecture we will see how this helps.)

Multilevel Paging

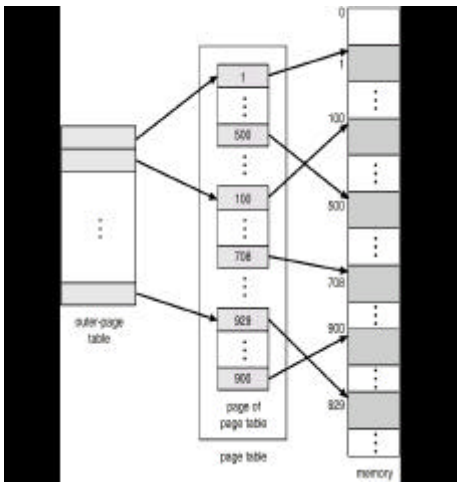
- A logical address (on 32-bit machine with 4K page size) is divided into two parts
  - 20-bit page number
  - 12-bit frame offset
- Paging the page table further divides the *page number*
  - 10-bit page number
  - 10-bit page offset
- Thus, a logical address is

page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

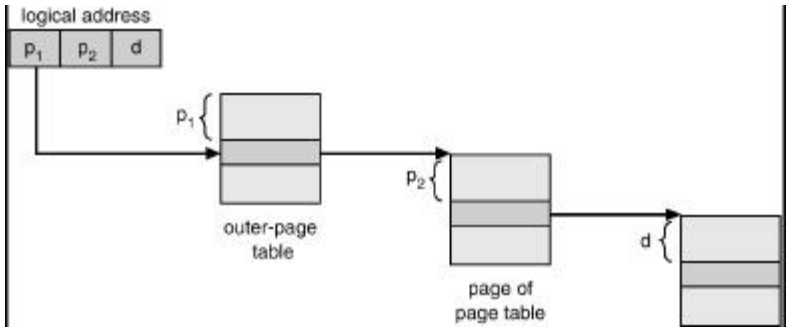
Multilevel Paging

Abstract view of a two-level page table



Multilevel Paging

Address-translation scheme for a two-level 32-bit paging architecture



Multilevel Paging

- Consider a computer with an address space of 2^{64}
  - With a 4k page and for convenience we make the inner page table fit in one page (i.e., 2^{10} * 4 bytes), we have

page number		page offset
p_1	p_2	d
42	10	12

  - Even if we page again, we have

page number			page offset
p_1	p_2	p_3	d
32	10	10	12

  - We would still need at least four-level paging if we wanted to make the table manageable

Multilevel Paging

- Since each level is stored as a separate table in memory, converting a logical address to a physical one may take four memory accesses for a four-level table
- This would effectively quintupled the memory access time if it were not for caching
- Four a 4-level page table with a cache hit rate of 98 percent

  EAT = (0.98) 120 + (0.02) 520

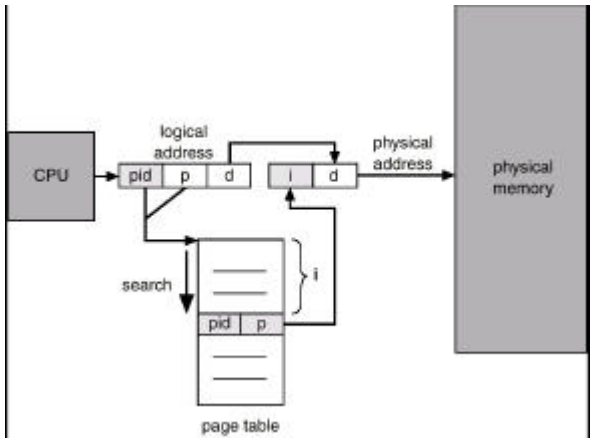
  = 128 nanoseconds.

  which is only a 28 percent slowdown in memory access time

Inverted Page Table

- As we can see, dealing with growing addresses spaces is problematic when using standard page tables
- Another approach to dealing with the resource requirements of page tables, is *inverted page tables*
  - One entry for each real page of memory
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
  - Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
  - Use hash table to limit the search to one or at most a few page table entries

Inverted Page Tables

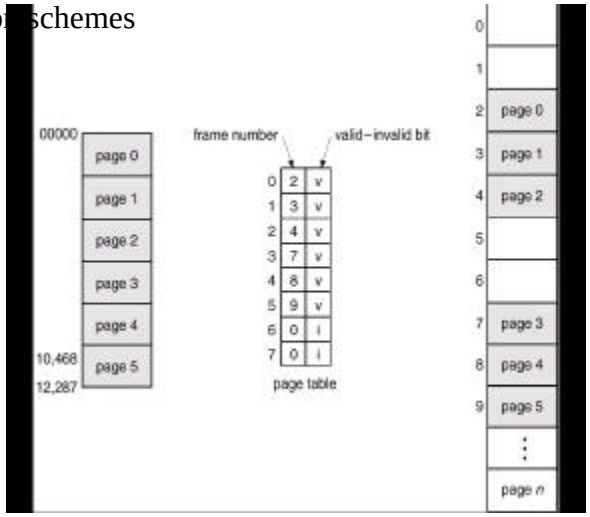


Paging and Memory Protection

- Memory protection is implemented by associating protection bit(s) in page table with each frame
  - Can also use bits to specify read/write/execute access
  - Illegal accesses are trapped by the OS
- *Valid/invalid* bit also attached to page table entries
  - *Valid* indicates that the associated page is in the process' logical address space, and is thus a legal page
  - *Invalid* indicates that the page is not in the process' logical address space
- Since a process does not usually use its entire page table, it is not necessary to have entries for all possible pages in the page table
  - Can have a *page table length register* to indicate size of table

Paging and Memory Protection

Bit flags are added to page table entries to signify various protection schemes

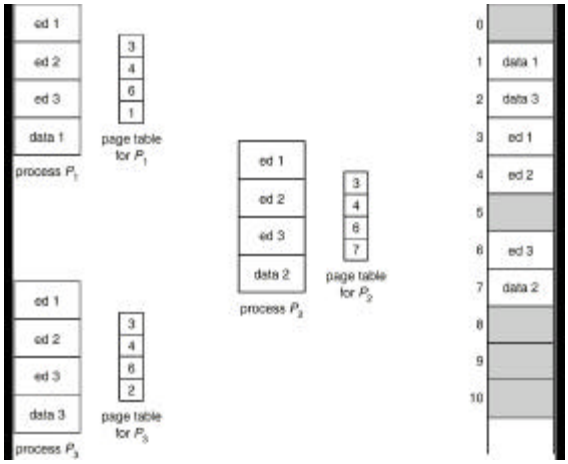


Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Shared code must appear in same location in the logical address space of all processes
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example

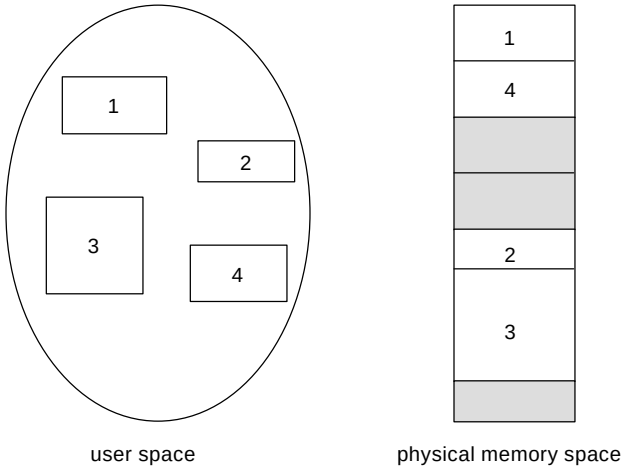
Processes share pages for editor code, but have their own pages for data



Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments; a segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays

Logical View of Segmentation

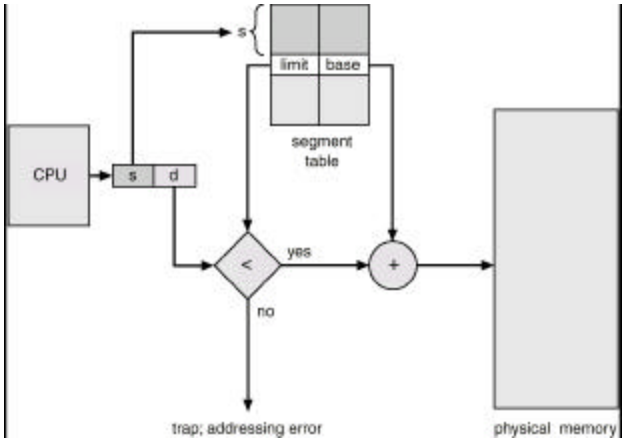


Segmentation Architecture

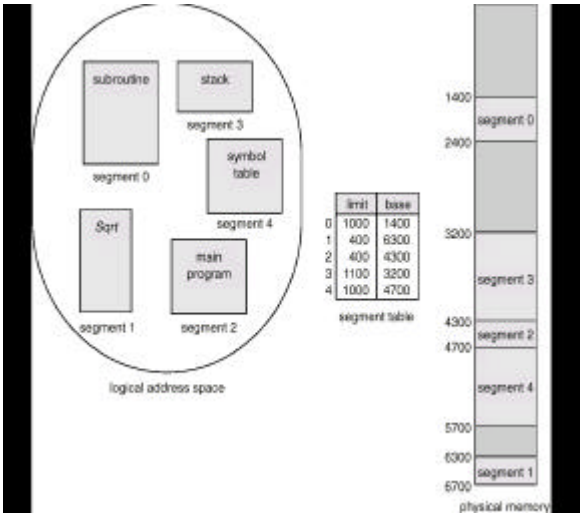
- Logical address consists of <segment-number, offset>
- *Segment table* maps two-dimensional physical addresses; each table entry has
  - *base* – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory
- *Segment-table length register (STLR)* indicates number of segments used by a program
  - Segment number *s* is legal if *s* < STLR.

Segmentation Architecture

Logical address translation



Segmentation Architecture



Segmentation Architecture

- Sharing
  - Allows sharing of segments
  - Shared segments must have same segment number
- Allocation
  - First fit/best fit
  - External fragmentation problems
- Protection
  - With each entry in segment table associate
    - Validation bit = 0 ⇒ illegal segment
    - Read/write/execute privileges

Segmentation with Paging

- It is possible to combine segmentation with paging
  - Just like before, paging allows segments to be non-contiguous and alleviates external fragmentation
- In general, paging is sufficient and segmentation is not necessary or relevant on newer computing systems other than the Intel x86 architecture
  - Everything that segmentation offers, paging offers too

Comparing Memory Management Strategies

- Hardware support
- Performance
- Fragmentation
- Relocation
- Swapping
- Sharing
- Protection