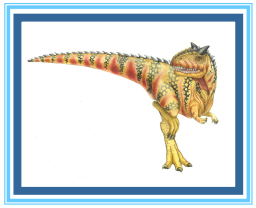
Chapter 8: Main Memory



Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table – covered lightly in class; will not be on the exam
- Example: The Intel 32 and 64-bit Architectures – not covered in class; will not be on the exam
- Example: ARM Architecture – not covered in class; will not be on the exam

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

Background

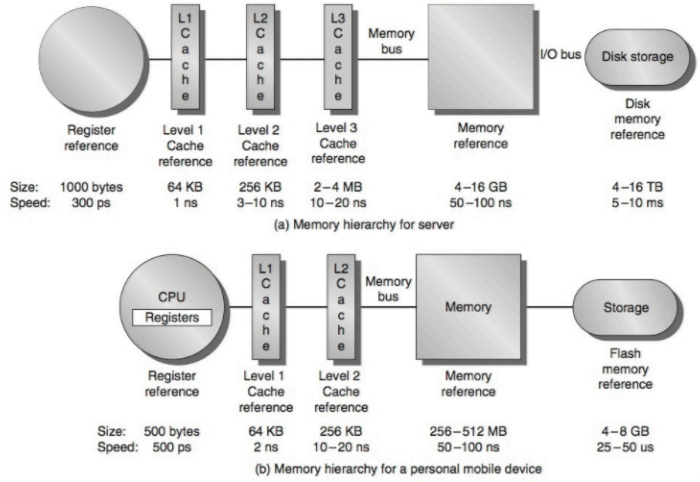
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Background (Walker 1 of 1)

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

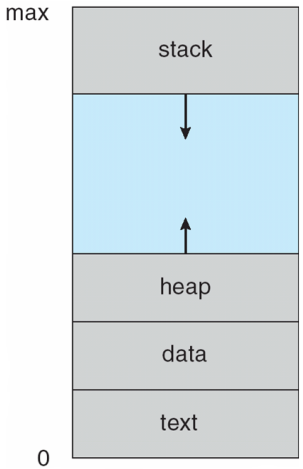
Memory Hierarchy (Walker 1 of 1)

Figure from 2011 Hennessey / Patterson architecture book



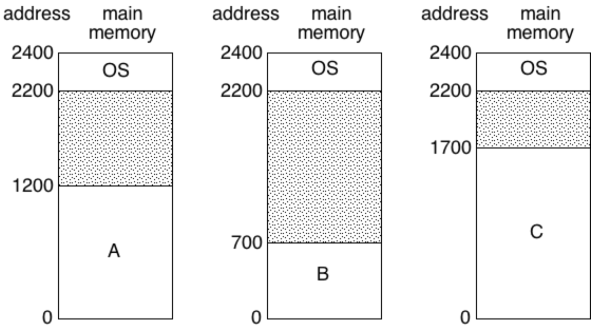
Process in Memory (Walker 1 of 1) (Review)

- Process includes (earlier text, reordered) the following sections (or segments):
  - *Stack* containing temporary data
    - Function parameters, return addresses, local variables
  - *Heap* containing memory dynamically allocated during run time
  - *Data* section containing (static) global variables
  - The program *code*, also called *text* section
- Each process also has a current state, including the program counter and general-purpose registers
  - But what if the process isn't running at the moment? We'll see…



Memory in a Uniprogrammed System (Walker 1 of 1)

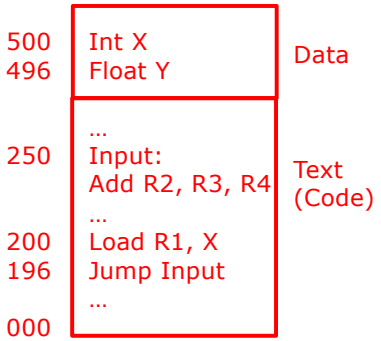
- OS gets a fixed segment of memory (usually highest memory)
- One process executes at a time in a single memory segment
  - Process is always loaded at address 0
  - Compiler and linker generate *physical* addresses
  - Maximum address = memory size – OS size

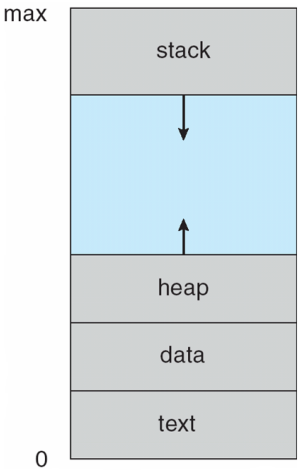


Addresses in Memory (Walker 1 of 1)

In the program code, instructions may refer to addresses within the process

- Locations of variables
- Locations to branch or jump to

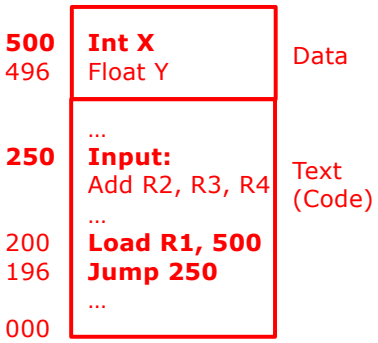


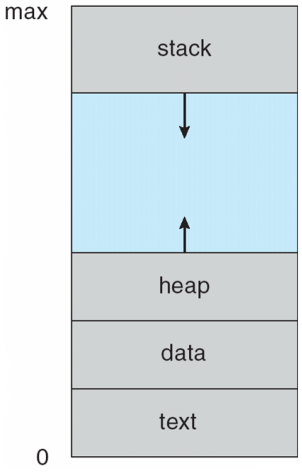


Addresses in Memory (Walker 1 of 1)

In the program code, instructions may refer to addresses within the process

- Locations of variables
- Locations to branch or jump to



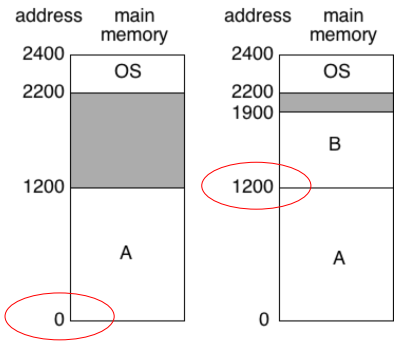


Linking & Loading (Walker 1 of 1)

- *Compiler* (system program) generates an object file for each source file
- *Linker* (system program; often thought of as part of the compiler) combines all the object files into a single executable object file (program)
- *Loader* (part of OS) loads an executable object file into memory at location(s) determined by the operating system
  - May be address 0, but probably is somewhere else
- Realistically, the OS will allow multiple processes to be in memory at the same time, but the compiler doesn't know where the program will be loaded into memory
  - Compiler just assumes program starts at 0
  - Compiler records *relocation information* (list of addresses to be modified later and instructions that refer to those addresses), and stores it in the object file
- A *Relocatable loader* then loads the program at a starting memory location specified by OS, modifying all addresses by adding the real starting location to those addresses

Static Relocation (Walker 1 of 2)

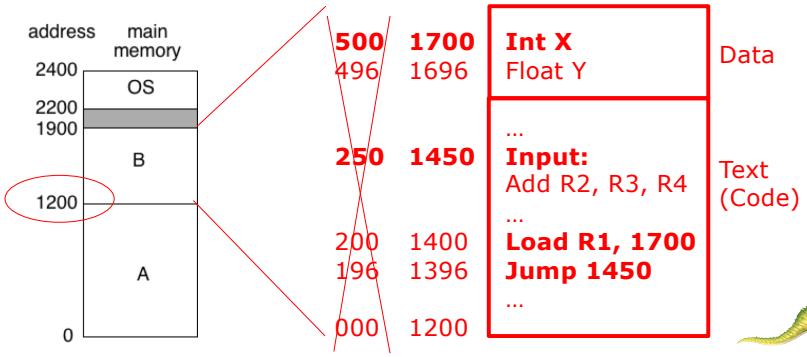
- Compiler and linker assume each process starts at address 0
- With static relocation, the OS loads a process into memory by:
  - Finding and allocating an area of memory where it fits completely
  - Adjusting the addresses in the processes to reflect its assigned location in memory



- *Relocatable loader* with *static relocation*
  - We have now modified the addresses in the program, so it runs at this location, but it will not run if moved
  - *Static* meaning it was relocated to this place in memory, but it will only run at that location

Static Relocation (Walker 1 of 2)

- Compiler and linker assume each process starts at address 0
- With static relocation, the OS loads a process into memory by:
  - Finding and allocating an area of memory where it fits completely
  - Adjusting the addresses in the processes to reflect its assigned location in memory

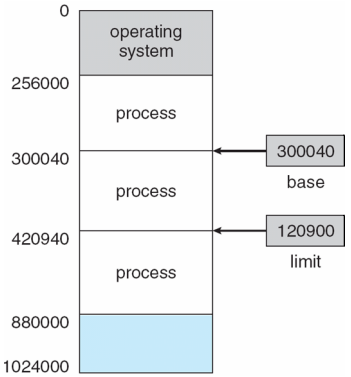


Static & Dynamic Relocation (Walker 1 of 1)

- Problems with *static relocation*:
  - Processes can not move — if swapped out, it must return to the same location in memory
  - No protection – a process can access another's memory, can even access the OS's memory
- An alternative: *dynamic relocation*
  - Basic idea is to dynamically change each memory address as the process runs
  - Address translation done by hardware — between the CPU and the memory is a *memory management unit* (MMU) that converts *logical addresses* to *physical addresses*
  - This address translation happens for every memory reference the process makes, and the OS and hardware must now manage two different addresses:
    - Logical address — address seen by the process
    - Physical address — address in physical memory (seen by OS)

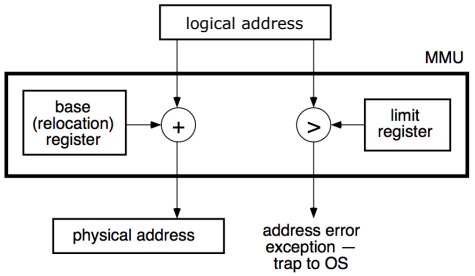
Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

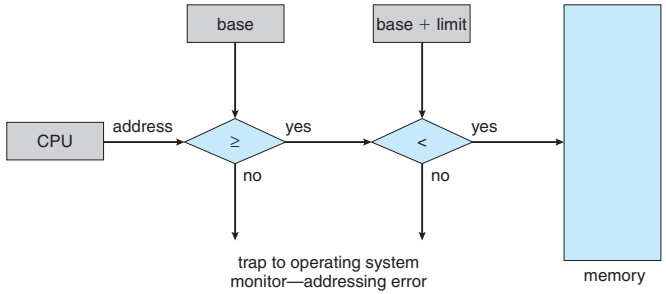


Implementing Dynamic Relocation
(Walker 1 of 1)

- The OS and hardware must now manage two different addresses:
  - Logical address — seen by the process
  - Physical address — address in physical memory (seen by OS)
- *Memory Management Unit* (MMU) converts *logical addresses* to *physical addresses*
  - Address translation: physical address = logical address + base
- Memory management unit also provides protection
  - If logical address > limit, then an address exception will occur
  - An exception is similar to interrupts or traps, but is caused by an error when an instruction executes



Hardware Address Protection



Evaluation of Dynamic Relocation
(Walker 1 of 1)

- Advantages:
  - OS can easily move a process (swap it out, swap it in elsewhere)
  - A process cannot access another's memory or the OS's memory
  - Hardware changes are minimal, but fairly fast and efficient
- Disadvantages:
  - Entire process must be contiguous in memory
  - Compared to static relocation, memory addressing is slower due to translation
  - Memory allocation is complex (partitions, holes, fragmentation, etc. as we will see later)
  - Not possible to share code or data between processes
  - Process are limited to physical memory size

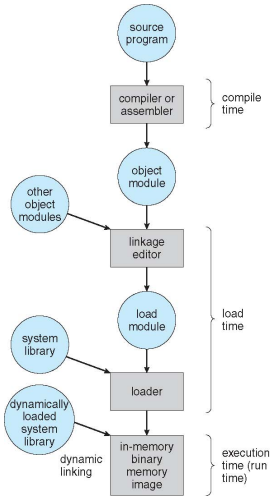
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

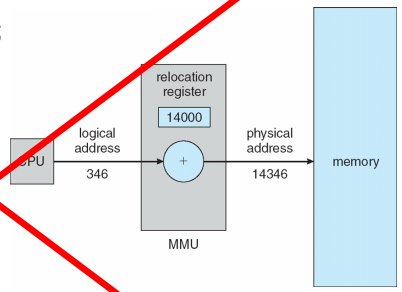
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

Swapping (Walker 1 of 1)

- Swapping = medium-term scheduling
- If the CPU scheduler chooses a process to run, and that process is not in memory (e.g., a new process or one previously swapped out), it must be brought into memory to run
- If there isn't room enough in memory for it plus the other processes, one or more processes can be *swapped out* to make room
  - OS swaps a process out by storing its complete state to disk
  - Need a policy to pick which process to swap out, such as: swap out blocked processes before ready processes
- When swapped out process becomes active again, OS must swap that entire process back in (into memory)
  - With static relocation, the process must be replaced in the same location
  - With dynamic relocation, OS can place the process in any free partition (must update the base / relocation and limit registers)
- Note that the entire process must be swapped in and out

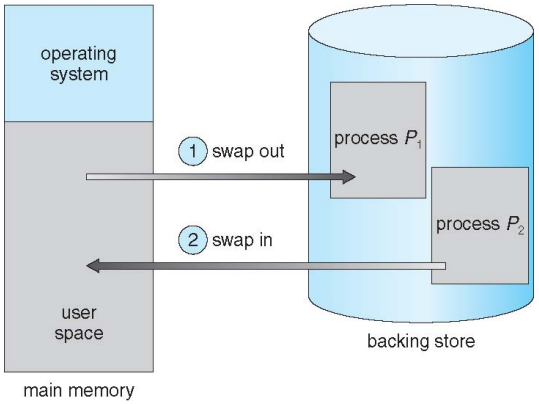
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

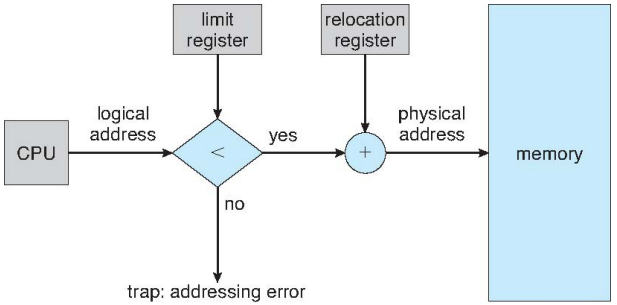
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers

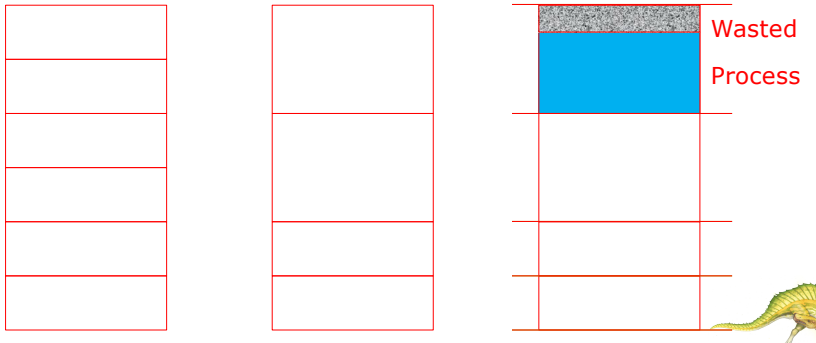


Dynamic Relocation + Contiguous Allocation
(Walker 1 of 3)

- OS can place entire process contiguously in memory at a fixed location, and then the MMU will handle the address translation
  - But.. *Where* do we put the program in memory?
- Partitioning – the physical memory is divided into *partitions* and a process is loaded into a free partition (a "hole" in the memory space)
- Fixed-size partitions:
  - Memory is divided into a predetermined set of fixed-size partitions
  - Partitions may all be the same size, or they may be different sizes (e.g., some big, some small), but the sizes do not change
  - Possible *internal fragmentation* (wasted space inside partition)
- Dynamic (variable-size) partitions:
  - Process is allocated a partition of exactly the right size
  - Possible *external fragmentation* (wasted space outside partition)
- Algorithms such as first-fit, best-fit, etc. are used to choose a free space for a particular process (see upcoming slide)

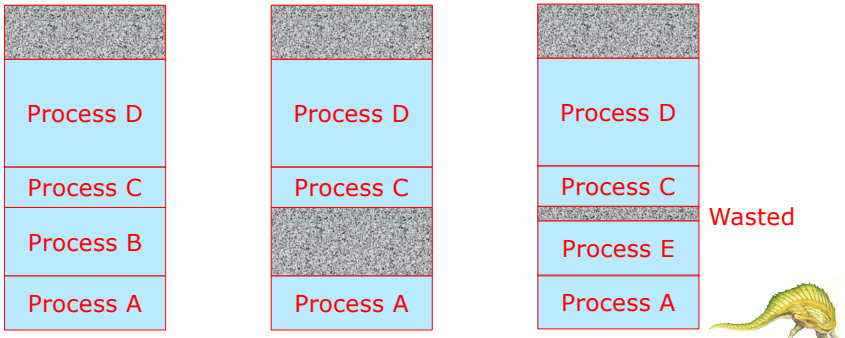
Dynamic Relocation + Contiguous Allocation
(Walker 2 of 3)

- Fixed-size partitions:
  - Memory is divided into a predetermined set of fixed-size partitions
  - Partitions may all be the same size, or they may be different sizes (e.g., some big, some small), but the sizes do not change
  - Possible *internal fragmentation* (wasted space inside partition)



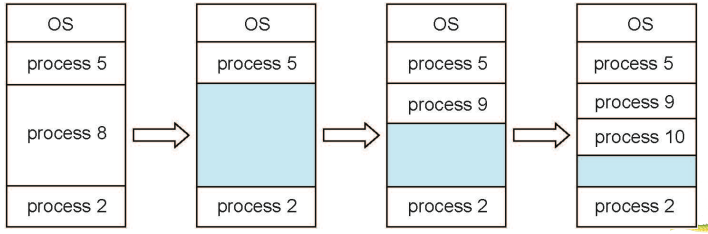
Dynamic Relocation + Contiguous Allocation
(Walker 3 of 3)

- Dynamic (variable-size) partitions:
  - Process is allocated a partition of exactly the right size
  - Possible *external fragmentation* (wasted space outside partition)



Multiple-partition allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

Chapter 8: Memory Management

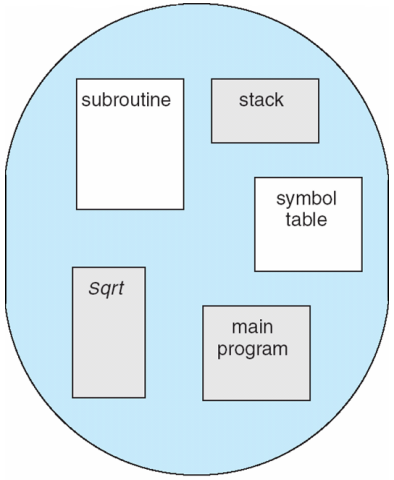
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

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
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays

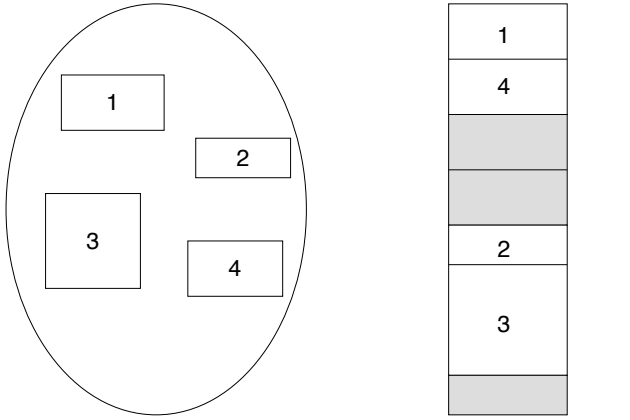
User's View of a Program



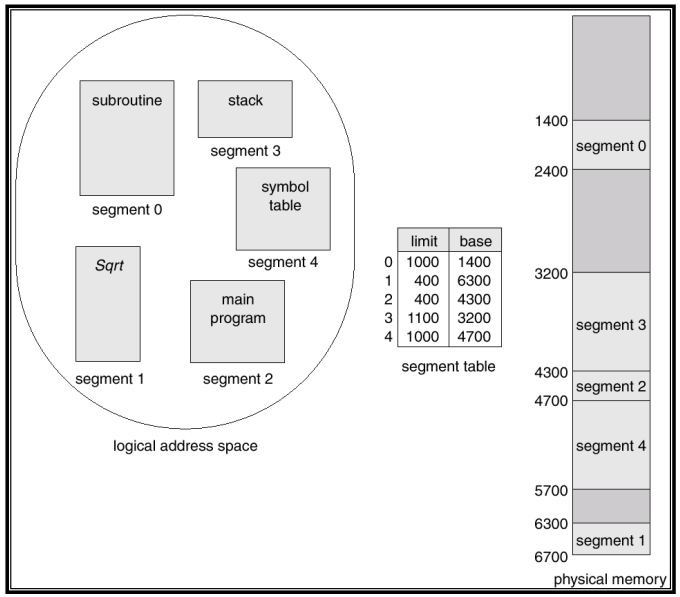
Segmentation (Walker 1 of 1)

- Basic idea — using the programmer's view of the program, divide the process into separate *segments* in memory
- Each segment has a distinct purpose:
  - Example: text/code, static data, heap, stack
  - Possibly divide the code up further, such as a separate segments for each (or some) procedures or data elements
  - Segments can be identified by the compiler, possibly also by the programmer
- The whole process is still loaded into memory, but the segments that make up the process ***do not*** have to be loaded contiguously into memory – each one can be anywhere in memory where there is sufficient free space for that segment
  - Space within a segment is still contiguous, and has to be allocated with first fit, best fit, etc.
  - The whole process must still be swapped out
- Segments may be of different sizes

Logical View of Segmentation



Segmentation Example (Walker 1 of 1)



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

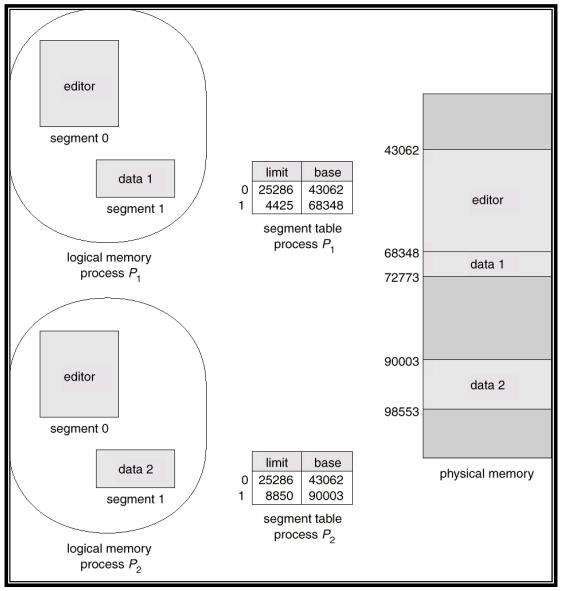
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location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

  segment number ***s*** is legal if ***s*** < **STLR**

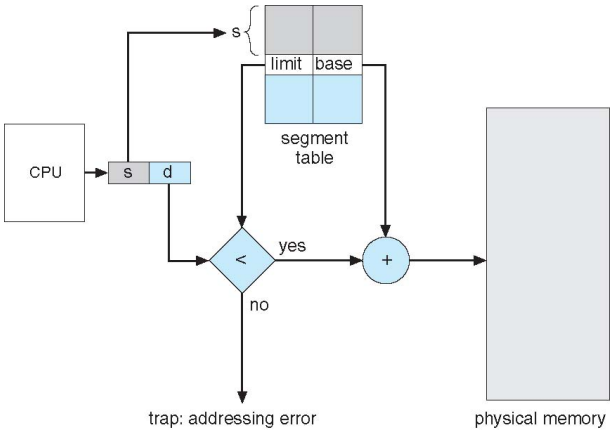
Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Sharing Segments (Walker 1 of 1)



Segmentation Hardware



Evaluation of Dynamic Relocation

(Review from Earlier in This Chapter) (Walker 1 of 1)

- Advantages:
  - OS can easily move a process (swap it out, swap it in elsewhere)
  - A process cannot access another's memory or the OS's memory
  - Hardware changes are minimal, but fairly fast and efficient **SEGMENTATION DOES HAVE A MORE COMPLEX MMU**
- Disadvantages:
  - ~~Entire process must be contiguous in memory~~
  - ~~Compared to static relocation, memory addressing is slower due to translation~~ **STILL THERE, BUT WE DON'T REALLY CARE**
  - Memory allocation is complex (partitions, holes, fragmentation, etc. as we will see later) **STILL A PROBLEM**
  - ~~Not possible to share code or data between processes~~
  - Process are limited to physical memory size **STILL A PROBLEM**

STRIKETHROUGH = ELIMINATED WITH SEGMENTATION

Evaluation of Segmentation (Walker 1 of 1)

- Advantages shared with Dynamic Relocation:
  - OS can easily move a process (swap it out, swap it in elsewhere)
  - A process cannot access another's memory or the OS's memory (we have address protection)
  - Hardware changes are minimal, and fairly fast and efficient (although we do add a memory access to reach the Segment Table)
- Advantages over Dynamic Relocation:
  - The entire process does not have be contiguous in memory (although individual segments must be contiguous)
  - Segments may be shared between processes
- Disadvantages shared with Dynamic Relocation:
  - Memory allocation (finding free space) is complex
  - External fragmentation is likely
  - Process are limited to physical memory size

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

Paging (Walker 1 of 1)

- The logical memory space of each process is divided into a number of small, fixed-size partitions called *pages*
  - Physical memory is divided into a large number of small, fixed-size partitions called *frames*
  - Segments no longer exists; processes are divided into pages
  - Page size = frame size
    - Usually 4K bytes to 16M bytes
  - The whole process is still loaded into memory, but the pages of a process do not have to be loaded into a contiguous set of frames
  - Allocation is easy; one frame is as good as another
  - Logical address consists of page number and displacement (offset) from the beginning of that page
- Compared to segmentation, paging:
  - Makes memory allocation and swapping easier (no first fit, etc.)
  - Does not have external fragmentation

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

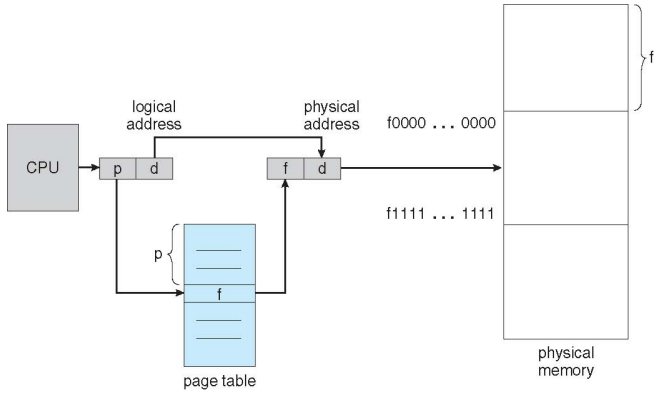
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit

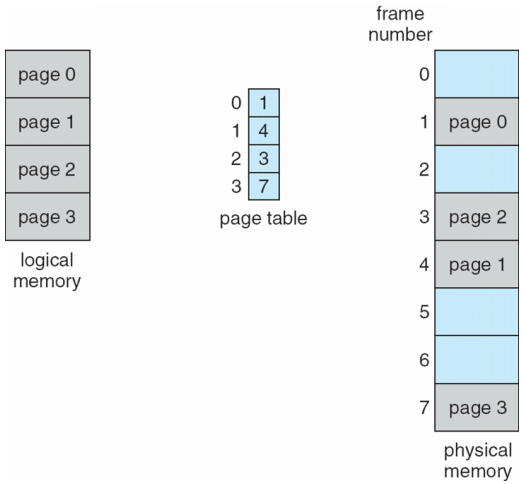
page number	page offset
p	d
m -n	n

  - For given logical address space 2^m and page size 2^n

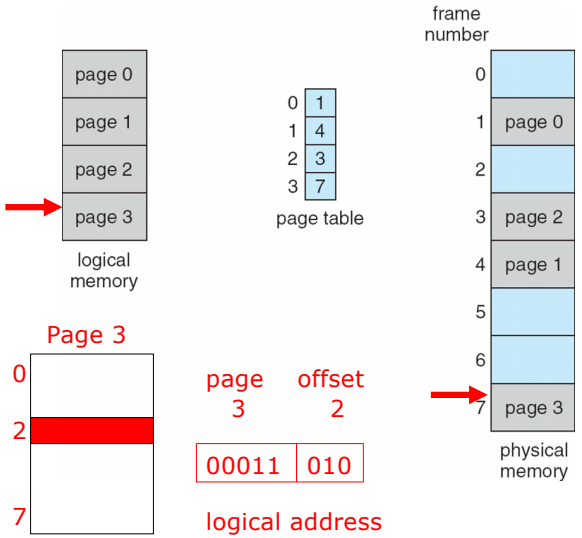
Paging Hardware



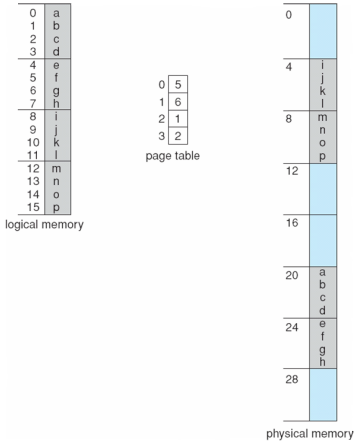
Paging Model of Logical and Physical Memory



Paging Model of Logical and Physical Memory
(Walker 1 of 1)

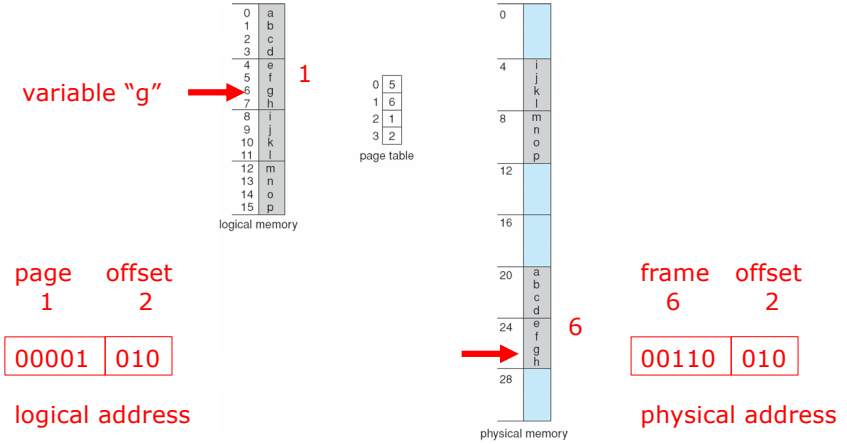


Paging Example



n=2 and *m*=4 32-byte memory and 4-byte pages

Paging Example (Walker 1 of 1)

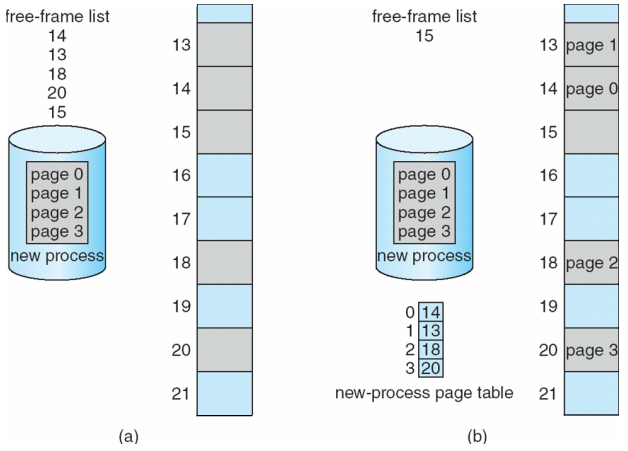


n=2 and *m*=4 32-byte memory and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames



Before allocation

After allocation

Managing Pages & Frames

(Walker 1 of 1)

- Keep page tables in memory, use *Page Table Base Register* (register in Memory Management Unit) to point to page table of active process
  - Page Table is stored in Process Control Block (PCB)
  - Page Table Base Register (in the MMU) points to the page table of the process that is on the CPU
  - The PTBR value must be copied from the PCB to the MMU when the process is placed onto the CPU (during a context switch)
  - Page Table also contains other bits for memory protection and for demand paging (discussed in Chapter 9)
- OS usually keeps track of free frames in memory using a *bit map*
  - A bit map is just an array of bits
    - 1 means the frame is free
    - 0 means the frame is allocated to a page
  - To find a free frame, look for the first 1 bit in the bit map
    - Most modern instruction sets have an instruction that returns the offset of the first 1 bit in a register

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

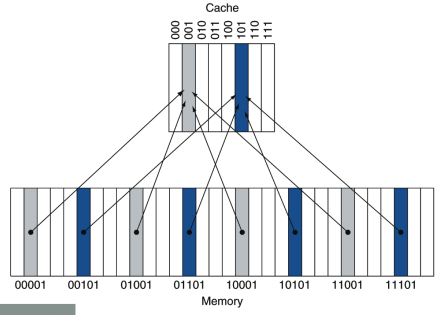
Cache Operation (Walker 1 of 1) (Review)

- When the CPU reads data from memory, it takes 200-300 times longer to get the data from memory than reading it from a general purpose register in the CPU
- To help improve performance, a small cache is placed between the CPU and the memory
- Whenever the CPU reads data from memory, it first checks the cache
  - If the data *is* in the cache, we say there was a cache "hit"
    - Reading data from the cache is only 2-3 times longer than reading from a general purpose register — so much faster than reading from memory
  - If the data *is not* in the cache, we say there was a cache "miss", and the CPU has to (slowly) read the data from the memory
    - But.. after the CPU reads the data, it also stores it in the cache, so the next time that data is needed, if it's still in the cache, it can be retrieved much faster
- As the cache gets full, data has to be replaced, but the caches tries to keep the most recently used data there so it can be used again

Cache Implementation (H&P, 1 of 4)

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

Cache Implementation (H&P, 2 of 4)

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Cache Implementation (H&P, 3 of 4)

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

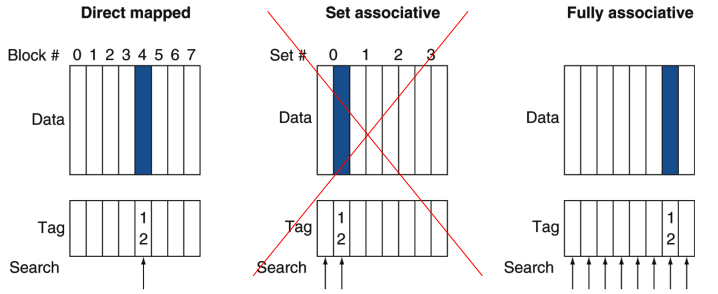
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

To look for a value in the cache, we go to that direct-mapped location, and see if the tag is there. If the tag is there, we can retrieve the data.

Cache Implementation (H&P, 4 of 4)

Associative Caches

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)



Why Do Caches Work? (Walker 1 of 1)

- Whenever the CPU reads data from memory, it first checks the cache
  - If the data *is* in the cache, we say there was a cache "hit"
  - If the data *is not* in the cache, we say there was a cache "miss", and the CPU reads read the data from the memory, but it also stores it in the cache in case it is needed again later
- Not mentioned before – the CPU does not just fetch one value from memory and store it in the cache, but it retrieves a whole *cache line* (usually 64 bits) of data
  - If the next instruction or data value was physically close to the previous value in memory, it may already be in the cache
- Caches work because of the Principle of Locality of Reference
  - *Spatial Locality* – if one item is needed, nearby items will often be needed sometime soon (e.g., next sequential instruction, next value in the array)
  - *Temporal Locality* – if an item is needed, it may be needed again sometime soon (e.g., instructions in a loop, primary data structures)

Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

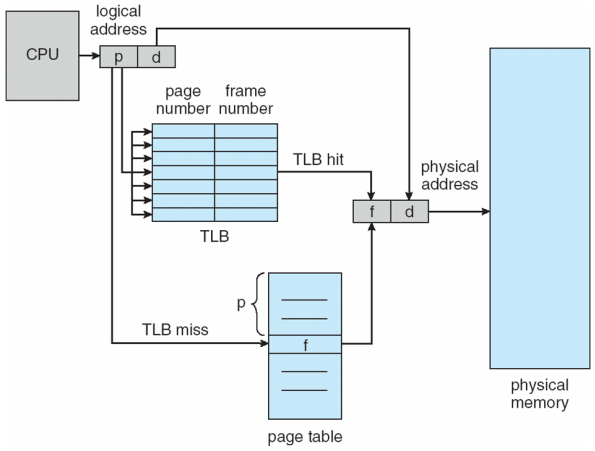
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

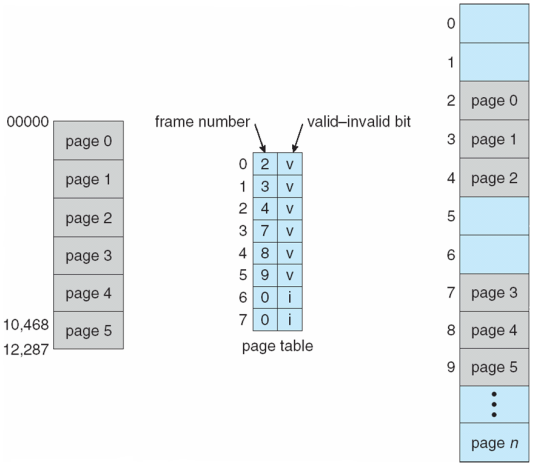
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "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
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

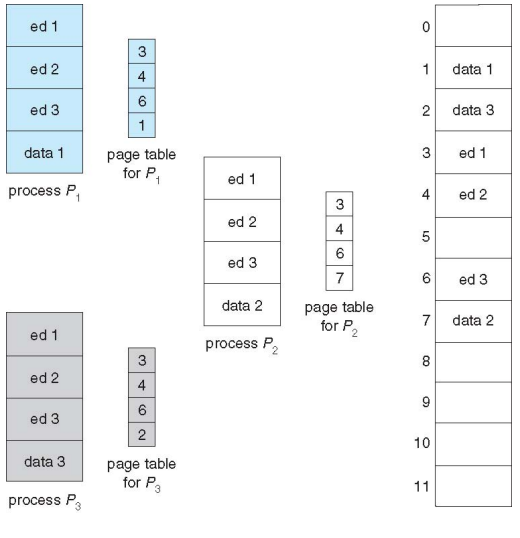
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **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



Evaluation of Paging (Walker 1 of 1)

- Advantages shared with Dynamic Relocation:
  - OS can easily move a process, process cannot access another's memory or the OS's memory (we have address protection)
  - Hardware changes are minimal, and fairly fast and efficient (although the TLB does help avoid accessing the Page Table)
- Advantages over Dynamic Relocation (similar to Segmentation):
  - The entire process does not have be contiguous in memory
  - Pages may be shared between processes
- Advantages over Segmentation:
  - Memory allocation (finding free space) is simple
- Tradeoff with Segmentation
  - Segmentation has external fragmentation; paging has internal fragmentation (although at most in one frame per process)
- Disadvantages shared with Dynamic Relocation & Segmentation:
  - Process are limited to physical memory size

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table – covered lightly in class; will not be on the exam
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture

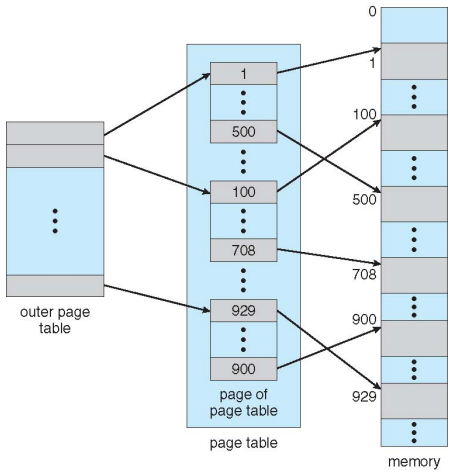
Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



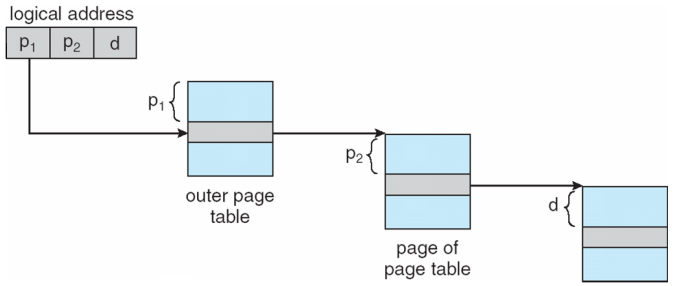
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
12	10	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



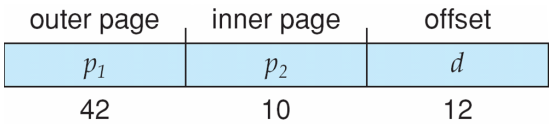
64-bit Logical Address Space

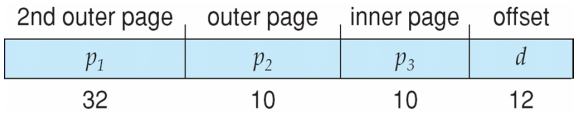
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

outer page	inner page	page offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

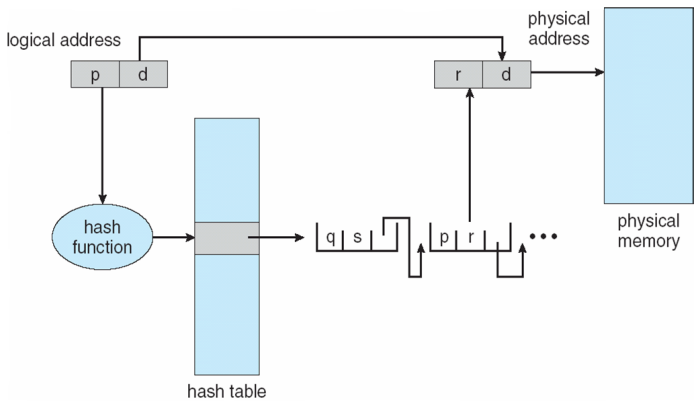




Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

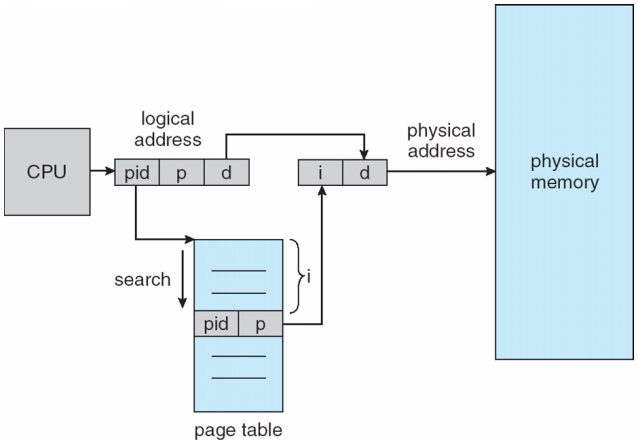
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- 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
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures – not covered in class; will not be on the exam
- Example: ARM Architecture

Example: The Intel 32 and 64-bit Architectures

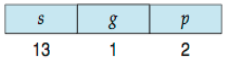
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))

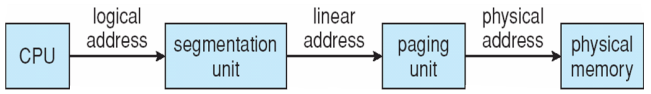
Example: The Intel IA-32 Architecture (Cont.)

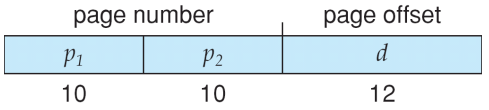
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



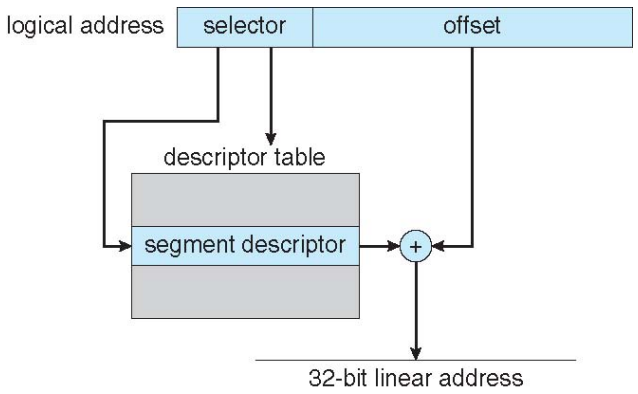
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32

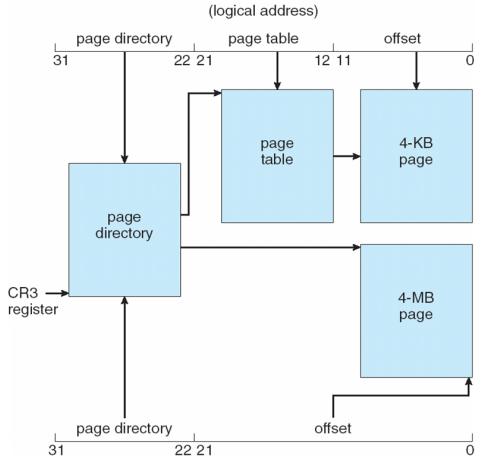




Intel IA-32 Segmentation

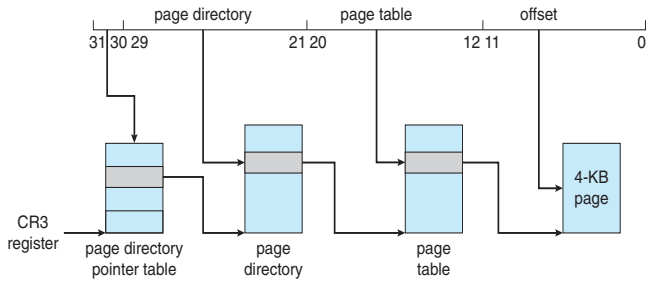


Intel IA-32 Paging Architecture



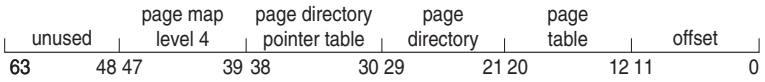
Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits

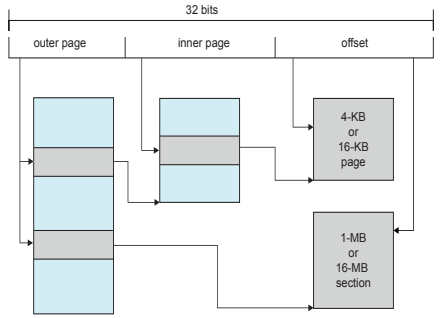


Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture – not covered in class; will not be on the exam

Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



End of Chapter 8