Memory Management

- Linking & loading programs: assigning memory addresses
- Logical vs. physical address spaces
- Process swapping
- Allocation mechanisms
  - » Contiguous allocation
  - » Paging
- Segmentation
- Combining segmentation and paging

Basic Steps in Running a Program

- Before being run, a program must be
  - » Brought into memory
  - » Placed within a process
- For batch systems, there's a list of programs to be run
  - » Input queue: collection of programs waiting to be run
  - » System picks the next program to execute
- For interactive systems, users specify processes to run
  - » System runs whichever program is specified

OS Responsibilities for Memory

- Assigning memory locations to instructions & data
  - » Programs can be loaded anywhere
  - » Individual instructions & data locations must be associated with specific memory locations
  - » Actual instructions may change depending on where the program and data are loaded
- Finding physical memory into which to place the program
  - » Find memory space not currently in use for other things
  - » Manage memory and allocate it appropriately to processes and other memory users (I/O devices, OS)
- Protect processes from one another
  - » Don’t allow one process to read or write memory that isn’t its own
  - » Allow sharing for efficiency or usability

Binding Instructions & Data to Memory

- OS must assign addresses to all instructions and data in a program
- Compile time
  - » Set starting location (and all others) when program compiled
  - » Recompile to change location in memory
- Load time
  - » Compiler generates offsets from the start of the program
  - » Loader sets all memory locations when program is loaded
  - » Program may not be relocated once it’s been loaded
- Execution time
  - » Compiler generates offsets from the start of the program
  - » Hardware provides registers to point to start of program
  - » Program may be moved during execution by changing regs

Linking & Loading

- Goals
  - » Provide locations for all instructions & data
  - » Propagate this information to all other instructions & data
- Step 1: compute offset from start of program
  - » Store computed values in symbol table
  - » Keep symbolic names in program
- Step 2: substitute actual values using actual locations
  - » Use symbol table to look up symbol values
  - » For relocatable code, use offsets from either 0 or a “base” register set by the operating system
- Step 3: OS loads program into memory and sets base register (if used)

Dynamic Loading

- Keep as little in memory as possible
  - » Don’t load a routine until it’s actually called
  - » May even reclaim space from routine after using it
- Useful when program has lots of code that’s used infrequently
  - » Error handling code
  - » Code for many different unusual cases
- No special OS help required
  - » Call to a routine first loads it into memory, then calls it
  - » Routine could “unload” itself upon finishing

Dynamic Linking

- Postpone linking (final resolution of addresses) until execution time
- Stub (small piece of code) used to locate a piece of OS code
  - » Ensures that the desired procedure is in memory
  - » Replaces call to stub with call to actual procedure
  - » Executes the procedure
- OS needed
  - » Trap calls to stub
  - » Load code into memory if needed
  - » Make sure process can read the code: code must be in process' address space

Overlays

- Divide program into several sections, including a "master" section
- Keep in memory only those sections (instructions & data) needed at a particular time
  - » Allows process to use less memory
  - » Implemented by programmer (not OS)
  - » Needs no special OS support
- May be useful for programs with several phases
  - » Compiler requires two passes: only need space for pass one or pass two (not both at once)
  - » Microsoft Word: code for table editor, graphical editor, and printer not all required at once

Logical & Physical Address Spaces

- Two different views of memory:
  - » Program view: logical address space
  - » Hardware view: physical address space
- Logical address (also called virtual address)
  - » Used by the process: process never sees physical addresses
  - » Generated by the CPU
- Physical address
  - » Memory management unit translates virtual to physical
  - » Memory hardware (chips) sees physical addresses
- Logical vs. physical addresses
  - » Same in compile-time and load-time binding schemes
  - » Different in execution-time address binding schemes

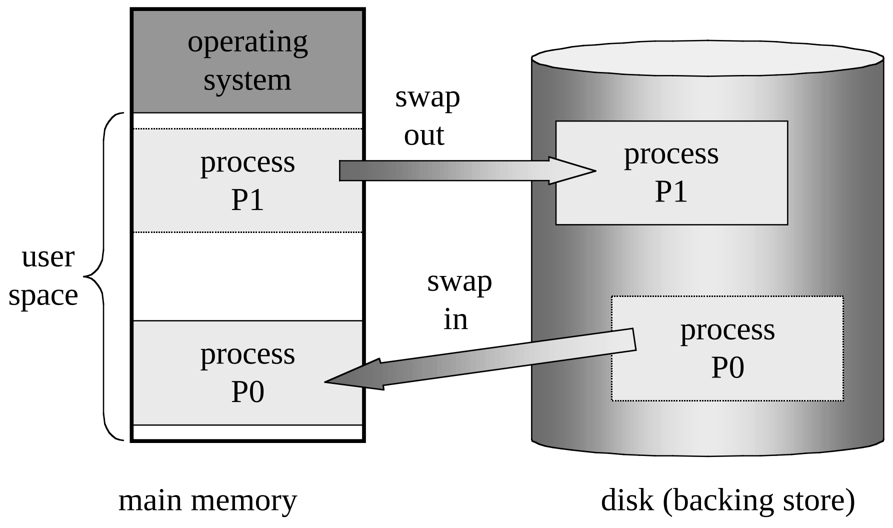
Memory Management Unit (MMU)

- Piece of hardware that maps virtual addresses to physical addresses
  - » May use several different methods to do this
  - » Relocation register: value in hardware register added to each address generated by a user process before it's sent to physical memory
  - » Page tables: more on them in a bit…
- User program only uses logical addresses
  - » Program can't tell where in physical memory it's loaded
  - » Program may be relocated in physical memory as long as MMU keeps logical addresses the same

Process Swapping

- Inactive processes consume memory, if not CPU time
- A process can be temporarily moved to a backing store, and brought back when it's ready to run again
- Backing store (usually a disk)
  - » Sufficient space to store copies of all user processes
  - » Direct (random) access to all of the images
- Swapping takes time
  - » Time to seek to correct location (relatively small)
  - » Time to transfer process to or from disk (relatively large): 10 MB process @ 5 MB/sec = 2 seconds!
- Swapping (or something very similar) is found everywhere
  - » UNIX / Linux
  - » Microsoft Windows & Macintosh OS

Schematic View of Swapping

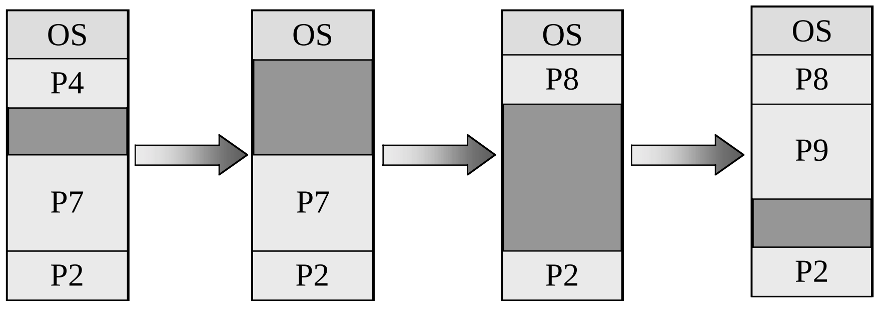


Contiguous Memory Allocation

- Divide main memory into two partitions
  - » Operating system (always resident), often in “low” memory (lower addresses)
  - » User processes in “high” memory
  - » Hardware protects operating system?
- Single-partition allocation
  - » Relocation register scheme protects other processes and and the operating system from the current process
  - » Relocation registers:
    - – Base: smallest physical address in the process (mapped to a 0 logical address)
    - – Limit (bounds): maximum logical address for the process
    - – Accesses greater than the limit are disallowed (cause an exception to be handled by the OS)

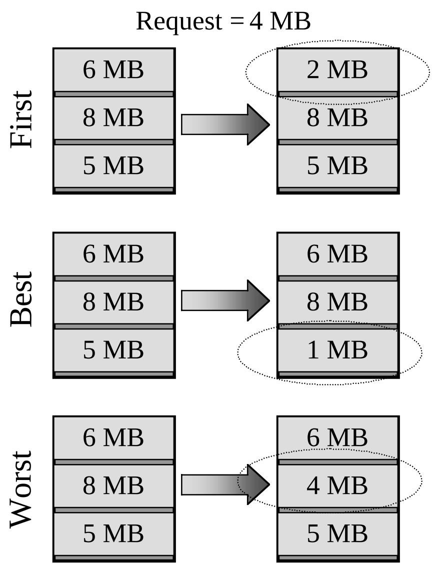
Multiple-Partition Allocation

- Blocks of available memory (called holes) are scattered throughout user memory
- Processes are allocated memory from a sufficiently large hole
- Operating system keeps track of
  - » Allocated partitions (and which process owns them)
  - » Free partitions (holes)



Picking a Free Block of Memory

- Given a list of holes, which hole do we allocate?
- First-fit: allocate the first hole in the list that's large enough
- Best-fit
  - » Allocate the smallest hole that's big enough
  - » Leaves a small leftover hole
- Worst-fit
  - » Allocate the largest hole
  - » Leaves a large leftover hole
- First-fit & best-fit are better in
  - » Speed
  - » Memory utilization

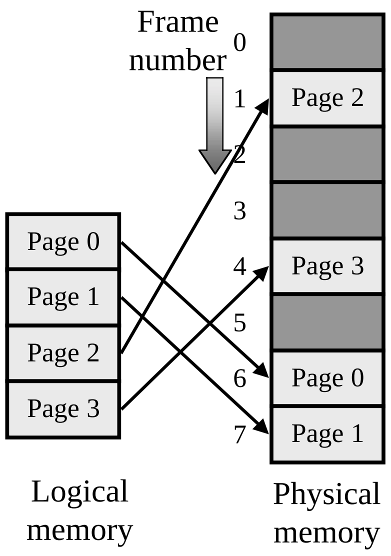


Problem: Fragmentation

- Fragmentation: there's enough memory available in the system, but it can't be used to satisfy the request
- External fragmentation: there's enough free space, but it's not contiguous
- Internal fragmentation
  - » Process isn't using all of the memory in its partition
  - » Unused memory is within a partition, not outside it
- Compaction can reduce external fragmentation
  - » Memory contents are shuffled to combine all free memory into one large block
  - » Compaction requires that relocation is dynamic and done at execution time (probably needs hardware help)
  - » Processes can't have outstanding I/O requests to user memory when they're moved, so do I/O only into OS buffers

Solution: Paging

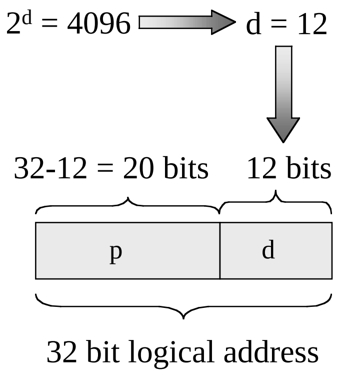
- Paging allows the logical address space of a process to be non-contiguous
- Process is allocated more physical memory when needed
- Physical memory divided into fixed-size blocks - frames
- Logical memory divided into fixed-size blocks - pages
- Keep track of free frames
- Allocate as many frames as a process has pages
- Use page table to map logical to physical addresses



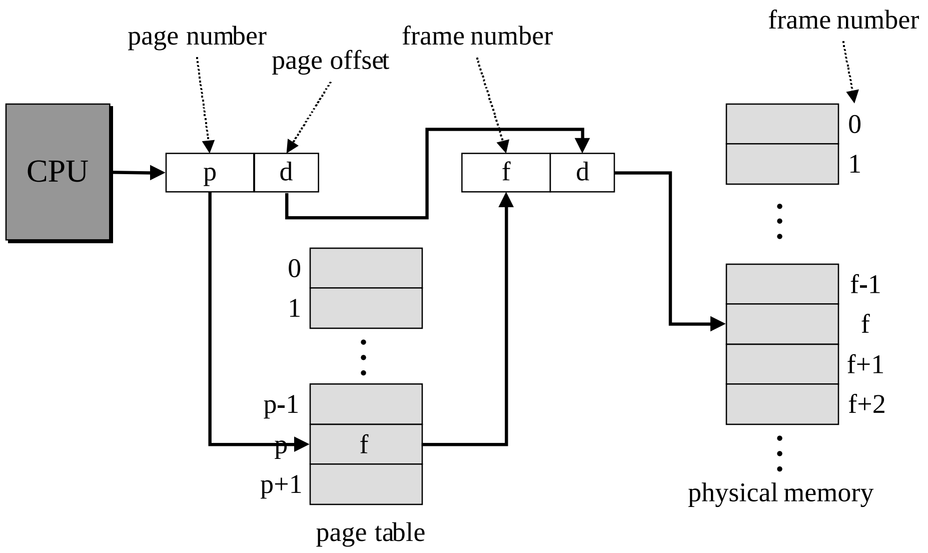
Breaking Up a Logical Address

- Split address from CPU into two pieces
  - » Page number (*p*)
  - » Page offset (*d*)
- Page number
  - » Index into page table
  - » Page table contains base address of page in physical memory
- Page offset
  - » Added to base address to get actual physical memory address
- Page size = 2^d bytes

Example:
• 4 KB (=4096 byte) pages
• 32 bit logical addresses



Address Translation Architecture



Memory & Paging Structures

Logical memory (P0)
Page 0
Page 1
Page 2
Page 3
Page 4
Page 5

Page table (P0)
10
0
4
7
2
6

Logical memory (P1)
Page 0
Page 1

Page table (P1)
8
1

Physical memory

Frame number	
0	Page 1 (P0)
1	Page 1 (P1)
2	Page 4 (P0)
3	▨
4	Page 2 (P0)
5	▨
6	Page 5 (P0)
7	Page 3 (P0)
8	Page 0 (P1)
9	▨
10	Page 0 (P0)
11	▨

Implementing Page Tables in Hardware

- Page table resides in main (physical) memory
- CPU uses special registers for paging
  - » Page table base register (PTBR) points to the page table
  - » Page table length register (PTLR) contains length of page table: restricts maximum legal logical address
- Translating an address requires two memory accesses
  - » First access reads page table entry (PTE)
  - » Second access reads the data / instruction from memory
- Reduce number of memory accesses
  - » Can't avoid second access (we need the value from memory)
  - » Eliminate first access by keeping a hardware cache (called a translation lookaside buffer or TLB) of recently used page table entries

Translation Lookaside Buffer (TLB)

- Search the TLB for the desired logical page number
  - » Search entries in parallel
  - » Use standard cache techniques
- If desired logical page number is found, get frame number from TLB
- If desired logical page number isn't found
  - » Get frame number from page table in memory
  - » Replace an entry in the TLB with the logical & physical page numbers from this reference

Logical page #	Physical frame #
8	3
unused	
2	1
3	0
12	12
29	6
22	11
7	4

Example TLB

Handling TLB Misses

- If PTE isn't found in TLB, OS needs to do the lookup in the page table
- Lookup can be done in hardware or software
- Hardware TLB replacement
  - » CPU hardware does page table lookup
  - » Can be faster than software
  - » Less flexible than software, and more complex hardware
- Software TLB replacement
  - » OS gets TLB exception
  - » Exception handler does page table lookup & places the result into the TLB
  - » Program continues after return from exception
  - » Larger TLB (lower miss rate) can make this feasible

How Long Do Memory Accesses Take?

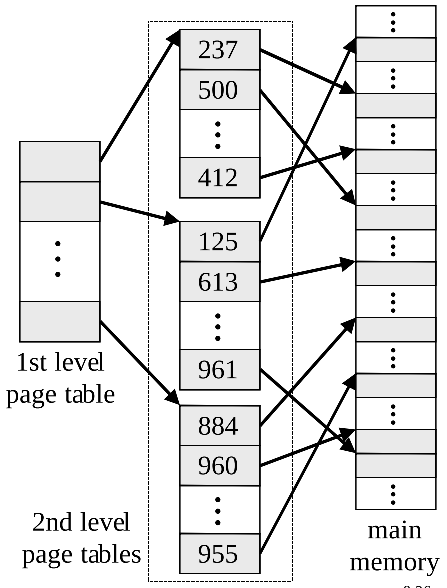
- Assume the following times:
  - » TLB lookup time = *a* (often zero - overlapped in CPU)
  - » Memory access time = m
- Hit ratio (*h*) is percentage of time that a logical page number is found in the TLB
  - » Larger TLB usually means higher *h*
  - » TLB structure can affect *h* as well
- Effective access time (an average) is calculated as:
  - » $EAT = (m + a)h + (m + m + a)(1-h)$
  - » $EAT = a + (2-h)m$
- Interpretation
  - » Reference always requires TLB lookup, 1 memory access
  - » TLB misses also require an additional memory reference

Protecting Memory

- Associate protection bits with each page table entry
  - » Store bits along with physical frame number
- Valid bit
  - » "valid" => page is in the process' logical address space, so access to it is OK
  - » "invalid" => page isn't currently accessible
    - – Page not in process' address space?
    - – Page not in memory?
- Writeable bit
  - » "writeable" => writes to this page are OK
  - » "non-writeable" => this page is read-only
- Executable bit: if set, instructions may come from this page
- Access must pass *all* checks to be allowed

Two-Level Page Tables

- Problem: page tables can be too large
  - » 2^{32} bytes in 4KB pages need 1 million PTEs
- Solution: use multi-level page tables
  - » "Page size" in first page table is large (megabytes)
  - » PTE marked invalid in first page table needs no 2nd level page table
- 1st level page table has pointers to 2nd level page tables
- 2nd level page table has actual frame numbers in it



More on Two-Level Page Tables

- Tradeoffs between 1st and 2nd level page table sizes
  - » Total number of bits indexing 1st and 2nd level table is constant for a given page size and logical address length
  - » Tradeoff between number of bits indexing 1st and number indexing 2nd level tables
    - – More bits in 1st level: fine granularity at 2nd level
    - – Fewer bits in 1st level: maybe less wasted space?
- All addresses in table are physical addresses
- Protection bits kept in 2nd level table
- Only PTEs from 2nd level table (actual logical -> physical translations) are cached in TLB

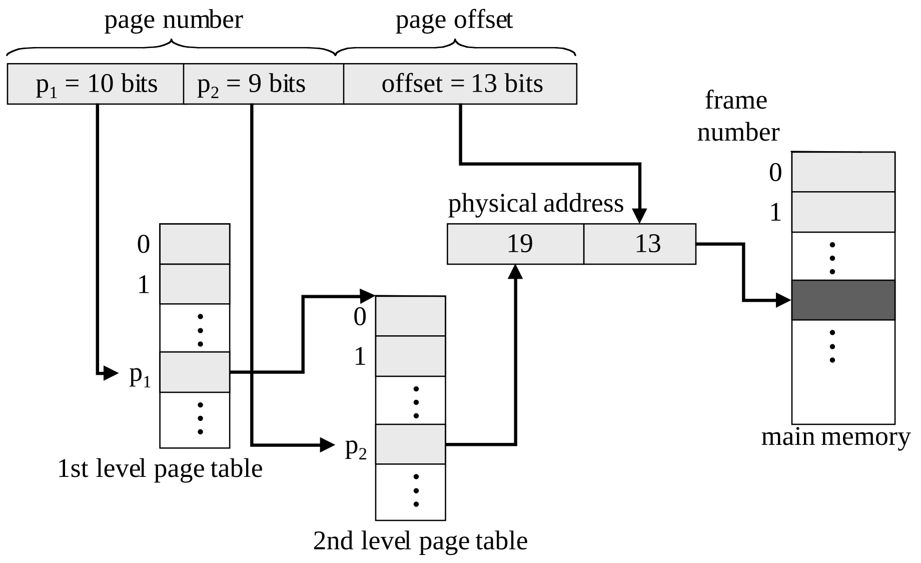
Two-Level Paging: Example

- System characteristics
  - » 8 KB pages
  - » 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - » 10 bit page number
  - » 9 bit page offset
- Logical address looks like this:

$$\overbrace{\boxed{p_1 = 10\text{ bits}}\boxed{p_2 = 9\text{ bits}}}^{\text{page number}}\overbrace{\boxed{\text{offset} = 13\text{ bits}}}^{\text{page offset}}$$

  - » p_1 is an index into the 1st level page table
  - » p_2 is an index into the 2nd level page table pointed to by p_1

2-Level Address Translation Example



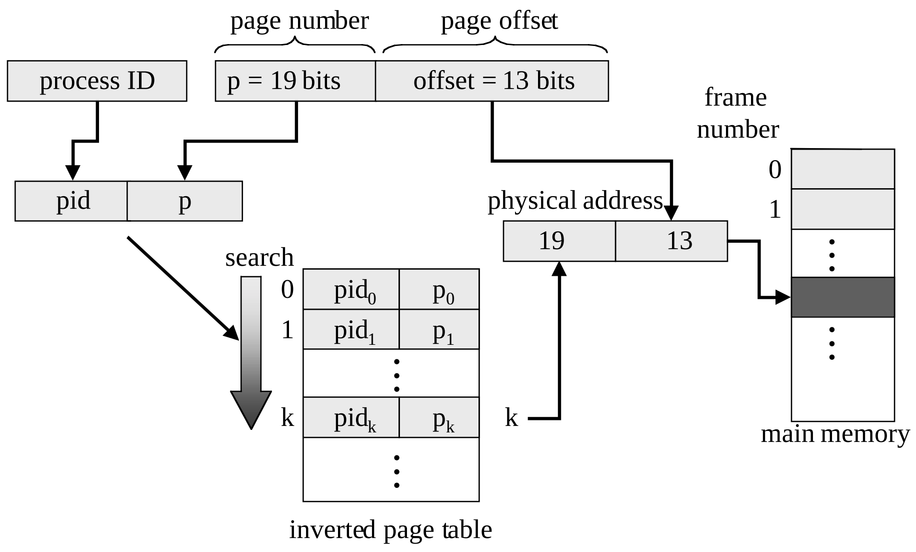
Multilevel Paging Performance Issues

- Each level requires another table lookup
  - » 2-level paging requires 3 accesses for each reference
  - » *N*-level paging requires *n*+1 accesses per reference
- Using a TLB can make this much faster
  - » TLB miss rate of 0.5% (actually a bit high for a modern CPU)
  - » Memory access time of 100 ns
  - » No penalty for using TLB
  - » Access time = 0.995 * 100 + 0.005 * 300 = 101 ns
  - » Only a 1% slowdown!
- Even handling in software is OK!
  - » TLB miss requires 2 us (2000 ns)
  - » Access time = 0.995 * 100 + 0.005 * 2000 = 109.5 ns
  - » Exception handler results in a 10% slowdown

Inverted Page Table

- Reduce page table size further: keep one entry for each frame in memory
- PTE contains
  - » Virtual address pointing to this frame
  - » Information about the process that owns this page
- Search page table by
  - » Hashing the virtual page number and process ID
  - » Starting at the entry corresponding to the hash result
  - » Search until either the entry is found or a limit is reached
- Frame number in physical memory is the index of the PTE in which the correct virtual page number is found
- Improve performance by using more advanced hashing algorithm

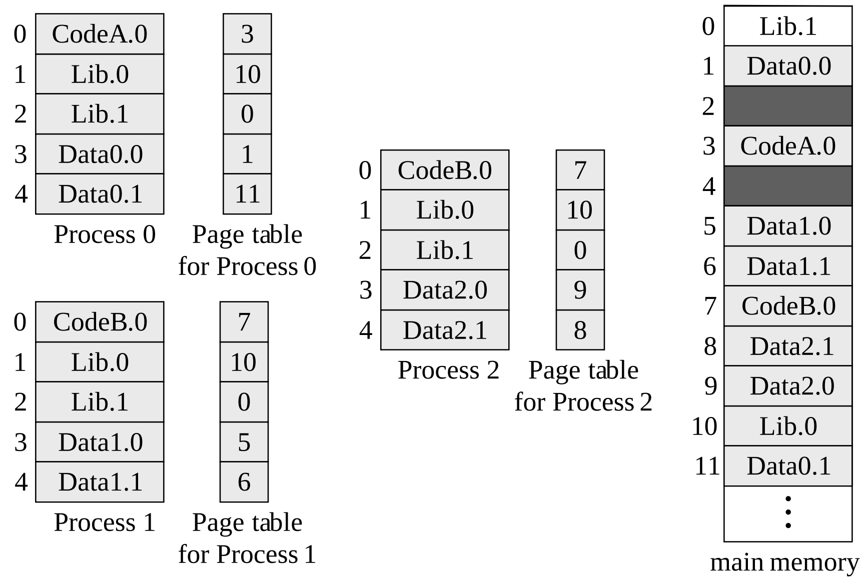
Inverted Page Table Architecture



Sharing Pages of Physical Memory

- Processes often want to share information with other processes
  - » Shared code used in several processes: saves space by loading only a single copy of the code for multiple processes
  - » Shared data between processes
- Shared pages should appear at same virtual address in each process
  - » Not a requirement, but makes life easier
  - » Pointers can be shared between processes
- Processes can also have private code & data
  - » Some PTEs point to shared pages (code & perhaps data)
  - » Other PTEs point to private pages (code & data)

Sharing Physical Pages: Example



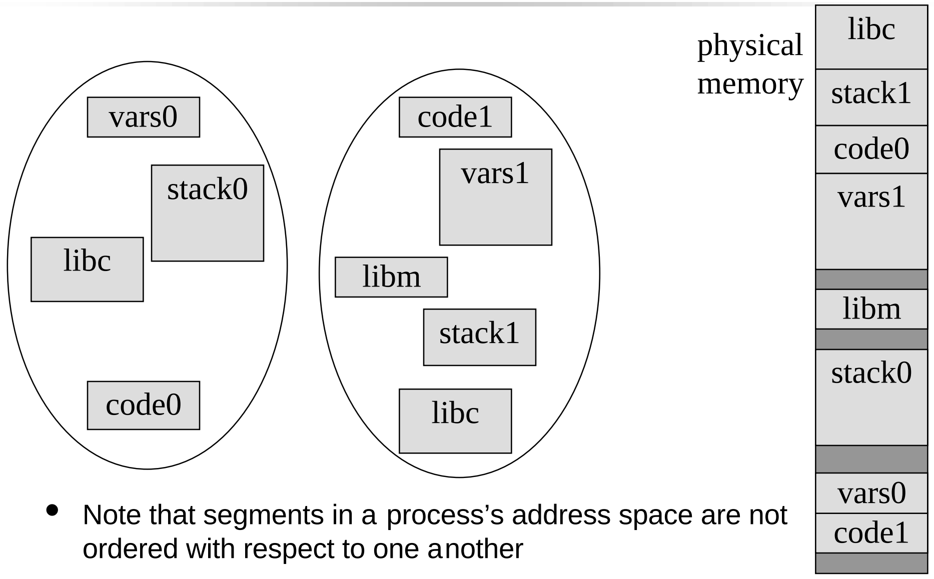
Problems with Sharing Physical Pages

- Sharing pages is good!
  - » Requires less physical memory, particularly for code
  - » Makes programs load faster (use code already in memory)
- Problems with sharing pages
  - » Pages usually have the same address in all processes: leads to difficulties allocating address space
  - » Changes in a single piece of shared code may require a lot of recompilation
- Solution: use segmentation

Segmentation

- Divide address space into segments rather than pages
  - » A <u>segment</u> is a logical unit from the user's point of view
  - » Segments can be any size (large or small)
  - » Segments can be placed at any location in a process's address space (more on that in a bit)
- Processes are composed of one or more segments
- Segments can be
  - » "Private" code to implement process-specific functions such as `main` in your code
  - » Libraries that have procedures shared by many processes
  - » Local variables (or groups of them)
  - » Global variables shared by many processes
  - » Stack

How Segments Fit Into Memory



- Note that segments in a process's address space are not ordered with respect to one another

Implementing Segments

- Logical addresses consist of segment number and offset: <segment number (s), offset (x)>
- Segment table maps logical address into physical address
  - » Base: starting physical address for each segment
  - » Limit: size of the segment
- CPU keeps track of segment table location
  - » Segment table base register (STBR) points to the start of the segment table in physical memory
  - » Segment table length register (STLR) indicates how many segments there are
- Translation is done by:
  - » Check that segment number is less than STLR
  - » Look up base of segment using STBR+s
  - » Add x to base to get physical address

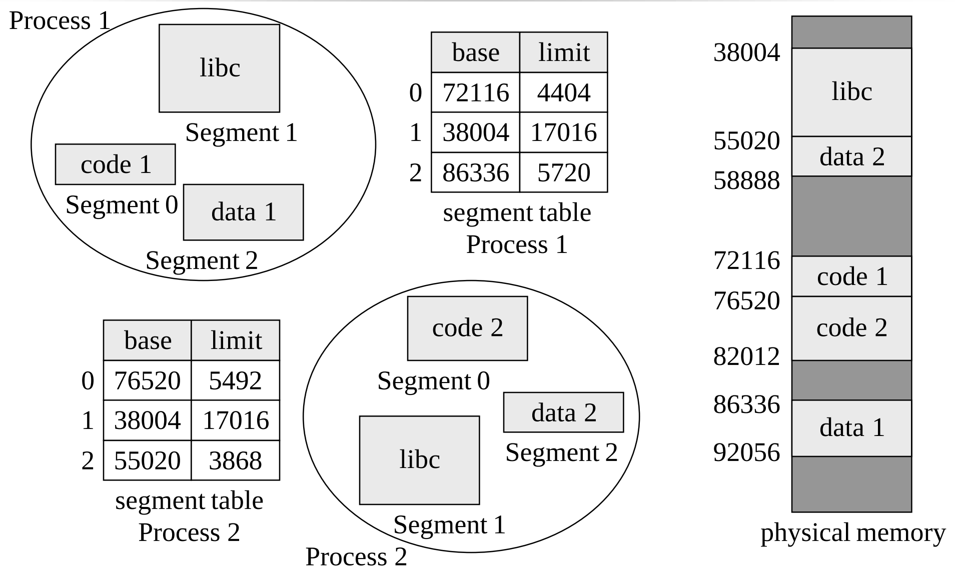
Advantages of Segments

- Relocation is easy
  - » Suspend all processes using the segment
  - » Copy segment to anywhere in memory
  - » Fix up the segment table to point to the new segment base
  - » Resume processes using the segment
- Sharing is easy
  - » All processes use the same segment number for any given segment
  - » Processes can use the segment simply by referring to it
- Allocation may be difficult
  - » Variable-sized objects can lead to external fragmentation
  - » Use first-fit or best-fit to allocate memory
  - » Relocate segments to consolidate memory "holes"

Protecting Segments

- Basic protection bits: each entry in the segment table has
  - » Valid bit: 1 = segment is valid
  - » Read/write/execute bits: indicate whether operation is permissible
- Protection is done on a segment-by-segment basis
  - » Code sharing occurs at the level of segments
  - » Memory with different sharing or permitted operations is split into multiple segments with the same permission bits
- More detailed protection is possible by using a separate segment table for each process
  - » Only include segments the process is allowed to access
  - » Make sure updates cover all of the affected segment tables

Segmentation Example



Segment table, Process 1:

	base	limit
0	72116	4404
1	38004	17016
2	86336	5720

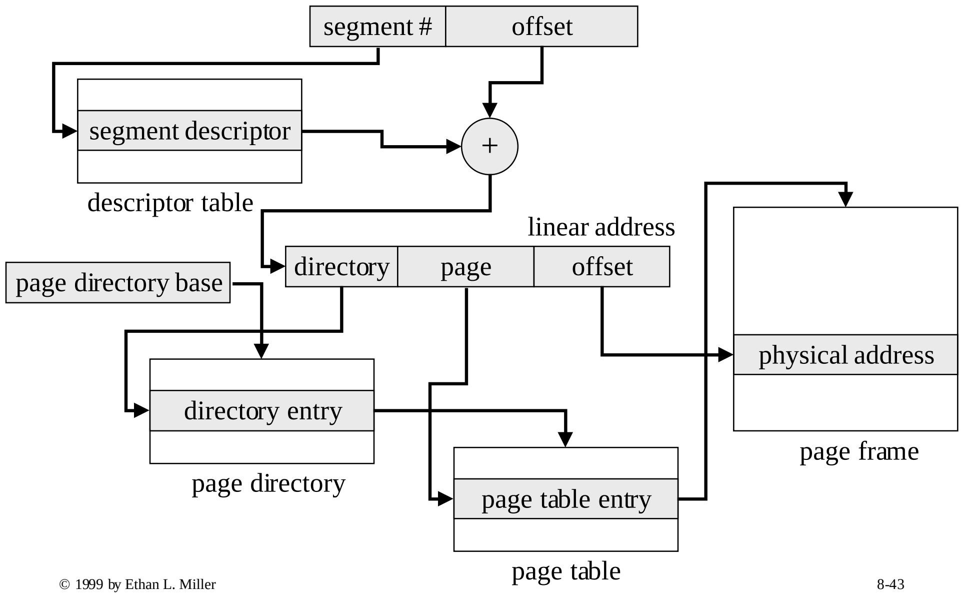
Segment table, Process 2:

	base	limit
0	76520	5492
1	38004	17016
2	55020	3868

Segmentation with Paging

- Segments have advantages
  - » Sharing is easier
  - » Relocatable code is very easy to make
- Paging has advantages
  - » Objects in memory are fixed size, making allocation easier
  - » Fragmentation is greatly reduced
- Use both segmentation and paging to get both advantages
- Two possible solutions
  - » Segment table entry contains pointer to a page table rather than actual segment (MULTICS)
  - » Segment table translates from segmented address to virtual address, which is then translated using page tables (x86)

Segmentation & Paging in the x86



Comparing Memory Management Schemes

- Hardware support: some schemes need special hardware that may not be available on a particular platform
- Performance: the more complex the scheme, the slower it usually runs
- Fragmentation: how much memory is wasetd?
- Relocation: how easy is it to move information around in memory, perhaps to reduce fragmentation?
- Sharing: can memory be shared between processes, reducing total memory usage?
- Protection: how are individual pages protected, particularly if sharing is possible?
- Swapping: how easy is it to move processes in and out of memory?