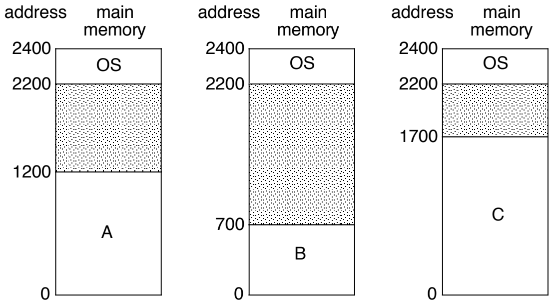
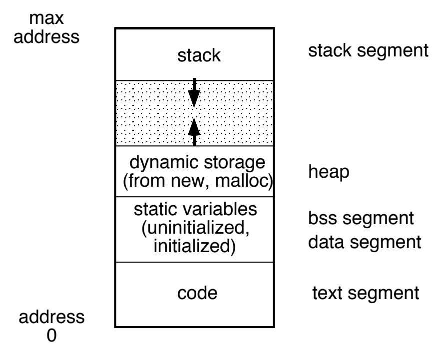
Memory Management in a Uniprogrammed System



- 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

Classifying Information Stored in Memory

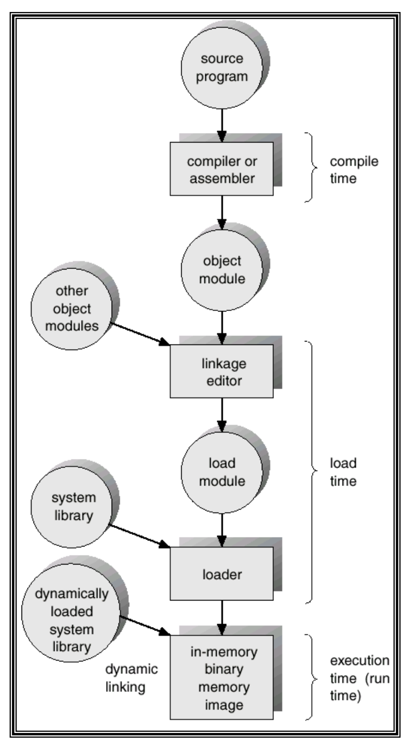
- Binding time (when is space allocated?):
  - Static: before program starts running
    - Program code, static global variables (initialized and uninitialized)
  - Dynamic: as program runs
    - Procedure stack, dynamic storage (space allocated by malloc or new)
- UNIX view of a process's memory (doesn't consider threads):



Segments of a Process

- Process' memory is divided into logical *segments* (text, data, bss, heap, stack)
- Who assigns memory to segments?
  - *Compiler* and *assembler* generate an *object file* each *source file*
  - *Linker* combines all the object files for a program into a single executable object file, which is complete and self-sufficient
    - Regroup all the segments from each file together (one big data segment, etc.)
    - Adjust addresses to match regrouping
    - Result is an executable program
  - *Loader* (part of OS) loads an executable object file into memory at location(s) determined by the operating system
  - *Program* (as it runs) uses new and malloc to dynamically allocate memory, gets space on stack during function calls

Processing a User Program



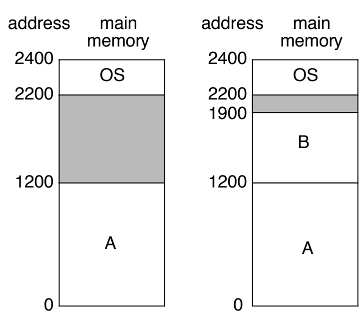
Why is Linking Difficult?

- When assembler assembles a file, it may find *external references* — symbols it doesn't know about (e.g., printf, scanf)
  - Compiler just puts in an address of 0 when producing the object code
  - Compiler records external symbols and their location (in object file) in a *patch list*, and stores that list in the object file
  - Linker must *resolve* those external references as it links the files together
- Compiler doesn't know where program will go in memory (if multiprogramming, always 0 for uniprogramming)
  - Compiler just assumes program starts at 0
  - Compiler records *relocation information* (location of addresses to be adjusted later), and stores it in the object file

Loading

- The *loader* loads the completed program into memory where it can be executed
  - Loads code and initialized data segments into memory at specified location
  - Leaves space for uninitialized data (bss)
  - Returns value of start address to operating system
- Alternatives in loading
  - *Absolute loader* — loads executable file at fixed location
  - *Relocatable loader* — loads the program at an memory location specified by OS
    - Assembler and linker assume program will start at location 0
    - When program is loaded, loader modifies all addresses by adding the real start location to those addresses
    - *Static Relocation* vs. *dynamic relocation*

Static Relocation



- Put the OS in the highest memory
- Compiler and linker assume each process starts at address 0
- At load time, the OS:
  - Allocates the process a segment of memory in which it fits completely
  - Adjusts the addresses in the processes to reflect its assigned location in memory

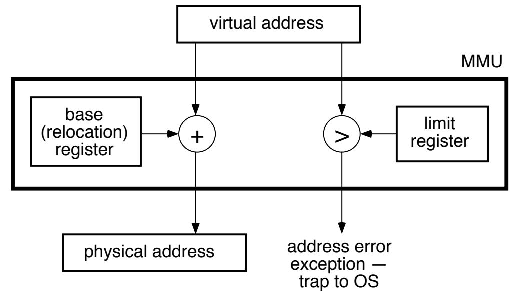
Static vs. Dynamic Relocation

- Problems with static relocation:
  - Safety — not satisfied — one process can access / corrupt another's memory, can even corrupt OS's memory
  - Processes can not change size (why…?)
  - Processes can not move after beginning to run (why would they want to?)
  - Used by MS-DOS, and early versions of Windows and Mac OS
- An alternative: dynamic relocation
  - The basic idea is to change each memory address dynamically <u>as the process runs</u>
  - Translation done by hardware — between the CPU and the memory is a *memory management unit* (MMU) that converts logical addresses to physical addresses
    - This translation happens for every memory reference the process makes

Dynamic Relocation

- There are now two different views of the address space:
  - The *physical address space* — seen only by the OS — is as large as there is physical memory on the machine
  - The *logical address space* —seen by the process — can be as large as the instruction set architecture allows
    - For now, we'll assume it's much smaller than the physical address space
  - Multiple processes share the physical memory, but each can see only its own logical address space
- 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)

Implementing Dynamic Relocation



- MMU protects address space, and translates logical addresses
  - *Base register* holds base physical address of process, *limit register* holds highest logical address of process
  - Translation:
    physical address = logical address + base
  - Protection:
    if logical address > limit, then trap to the OS with an address exception

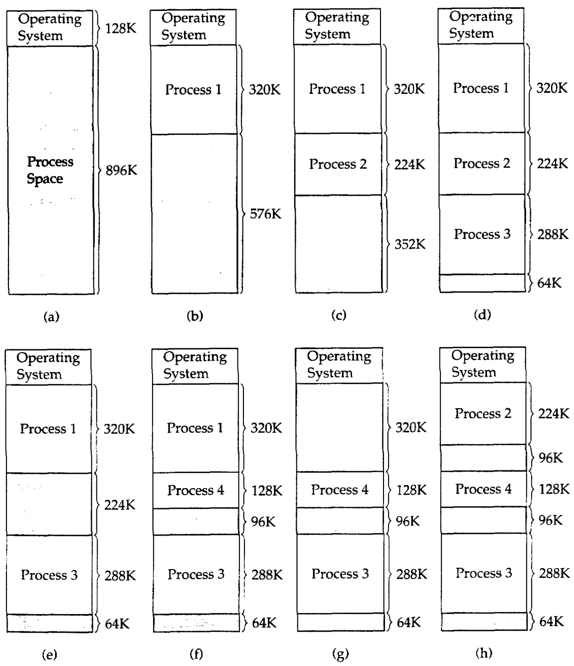
Dynamic Relocation — OS vs. User Programs

- User programs (processes) address their own logical memory
  - Run in relocation mode — indicated by a bit in the PSW — and in user mode
    - User programs can not change the relocation mode
- OS directly addresses physical memory
  - OS runs with relocation turned off, and in kernel mode
- When user program makes a system call:
  - CPU atomically goes into kernel mode, turns off relocation, traps to trap handler
  - OS trap handler accesses physical memory and does whatever is necessary to service the system call
  - CPU atomically turns on relocation, goes into user mode, returns to user program

Dynamic Relocation and Partitioning

- Physical memory is divided into *partitions*
  - A process is loaded into a free partition (a "hole" in the memory space)
- Fixed-size partitions:
  - Memory is divided into a predetermined number of fixed-size partitions
    - Partitions may be either of equal size, or of different (although fixed) sizes
  - Use first-fit, best-fit, etc. to keep track of holes (see upcoming slide)
  - Number of partitions limits the *degree of multiprogramming* — number of active processes
- Dynamic (variable-size) partitions:
  - When a process gets brought into memory, it is allocated a partition of exactly the right size

Effect of Dynamic Relocation with Dynamic Partitioning



Managing the Free List

- Dynamic relocation and partitioning maintains a *free list* to keep track of all the holes
- Algorithms to manage the free list:
  - Best fit
    - Keep linked list of free blocks
    - Search the whole list at each allocation
    - Choose the hole that comes the closest to matching the request size
      - Any unused space becomes a new (smaller) hole
    - When freeing memory, combine adjacent holes
    - Any way to do this efficiently?
  - First fit
    - Scan the list for the first hole that is large enough, choose that hole
    - Otherwise, same as best fit
  - Which is better? Why??

Swapping (Medium-Term Scheduling)

- If there isn't room enough in memory for all processes, some processes can be swapped out to make room
  - OS *swaps a process out* by storing its complete state to disk
  - OS can reclaim space used (not really…) by ready or blocked processes
- When process becomes active again, OS must *swap* it 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 relocation and limit registers)
- Swapping and dynamic relocation make it easy to increase the size of a process and to compact memory (although slow!)

UNIX Process Model (From Lecture 06)

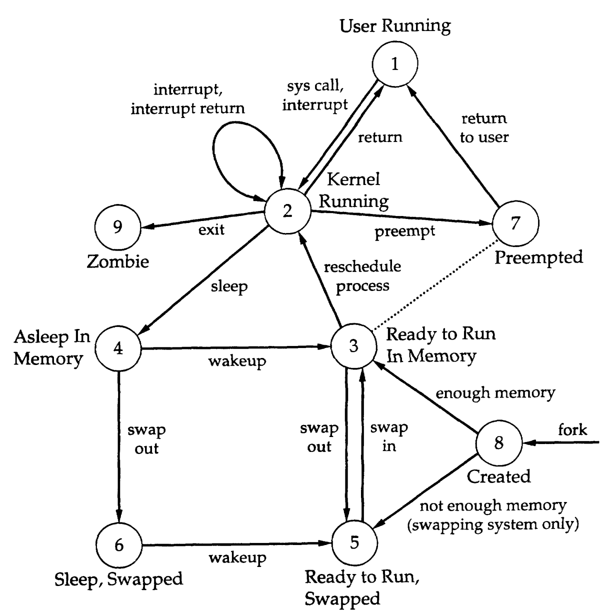


FIGURE 3.16 UNIX process state transition diagram [BACH86]

Figure from *Operating Systems*, 2nd edition, Stallings, Prentice Hall, 1995

Original diagram from *The Design of the UNIX Operating System*, M. Bach, Prentice Hall, 1986

Evaluation of Dynamic Relocation

- Advantages:
  - OS can easily move a process
  - OS can allow processes to grow
  - Hardware changes are minimal, but fairly fast and efficient
  - ➥Transparency, safety, and efficiency are all satisfied, although there is some small overhead to dynamic relocation
- Disadvantages:
  - Compared to static relocation, memory addressing is slower due to translation
  - Memory allocation is complex (partitions, holes, fragmentation, etc.)
  - If process grows, OS may have to move it
  - Process limited to physical memory size
  - Not possible to share code or data between processes

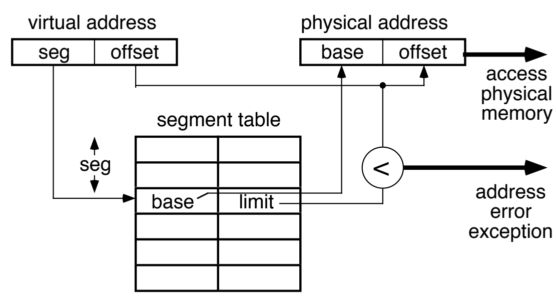
Segmentation

- 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: code, static data, heap, stack
      - Maybe a separate segment for each function or object
    - Segments may be of different sizes
    - Stack and heap don't conflict
  - 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
    - Space within a segment is contiguous
- Each segment has *protection bits*
  - Read-only segment (code)
  - Read-write segments (data, heap, stack)
  - Allows processes to share code and data

Segment Addresses

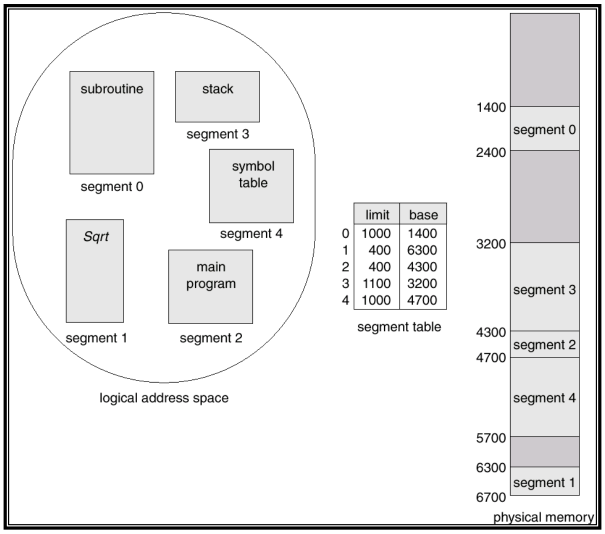
- Logical address consists of:
  - Segment number
  - Offset from beginning of that segment
  - Both are generated by the assembler
- What is stored in the instruction?
  - Simple method:
    - Top bits of address specify segment
    - Bottom bits of address specify offset
  - Implicit segment specification:
    - Segment is selected implicitly by the instruction being executed (code vs. data)
    - Examples: PDP-11, Intel 386/486
  - Explicit segment specification:
    - Instruction prefix can request that a specific segment be used
    - Example: Intel 386/486…
    - Most common technique

Implementing Segments



- A *segment table* keeps track of every segment in a particular process
  - Each entry contains base and limit
  - Also contains protection information (sharing allowed, read vs. read/write)
- Additional hardware support required:
  - Multiple base and limit registers, or
  - Segment table base register (points to a segment table stored in a PCB)

Segmentation Example



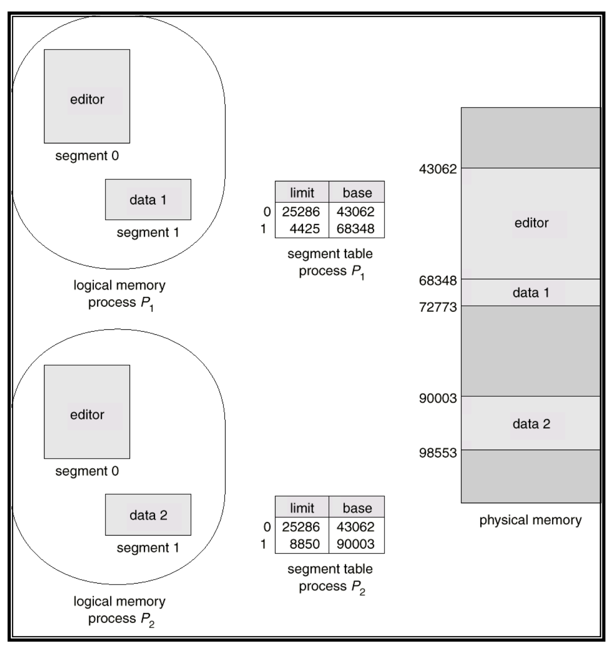
Managing Segments

- When a process is loaded into memory:
  - Allocate space in physical memory for all of the process's segments
  - Create a (mostly empty) segment table, and store it in the process's PCB
- When a context switch occurs:
  - Update the segment table base register to point to the segment table in the new process's PCB
- If there's no space in physical memory:
  - Compact memory (move segments, update bases) to make contiguous space
    - Tradeoff efficiency for overhead
  - Swap one or more segments out to disk
    - To run that process again, swap *all* of its segments back into memory

Managing Segments (cont.)

- To enlarge a segment:
  - If space above the segment is free, OS can just update the segment's limit and use some of that space
  - Move this segment to a larger free space
  - Swap the segment above this one to disk
  - Swap this segment to disk, and bring it back into a larger free space
- Advantages of segmentation:
  - Segments don't have to be contiguous
  - Segments can be swapped independently
  - Segments allow sharing
- Disadvantages of segmentation:
  - Complex memory allocation (first-fit, etc.)
  - External fragmentation

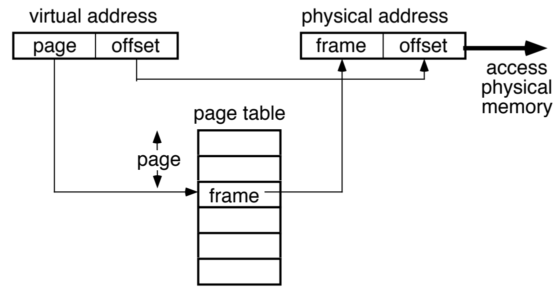
Sharing Segments



Paging

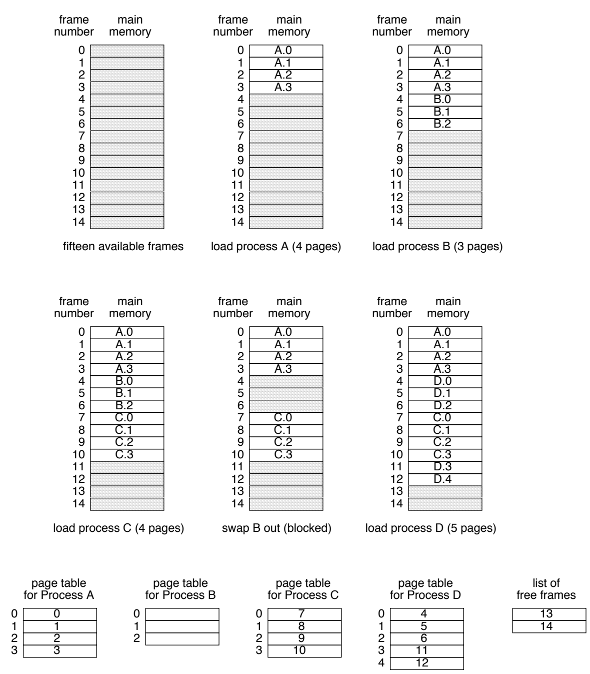
- Compared to segmentation, paging:
  - Makes allocation and swapping easier
  - No external fragmentation
- Each process is divided into a number of small, fixed-size partitions called *pages*
  - Physical memory is divided into a large number of small, fixed-size partitions called *frames*
  - Pages have nothing to do with segments
  - Page size = frame size
    - Usually 512 bytes to 16K 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
  - Logical address consists of page number and offset from beginning of that page

Implementing Paging

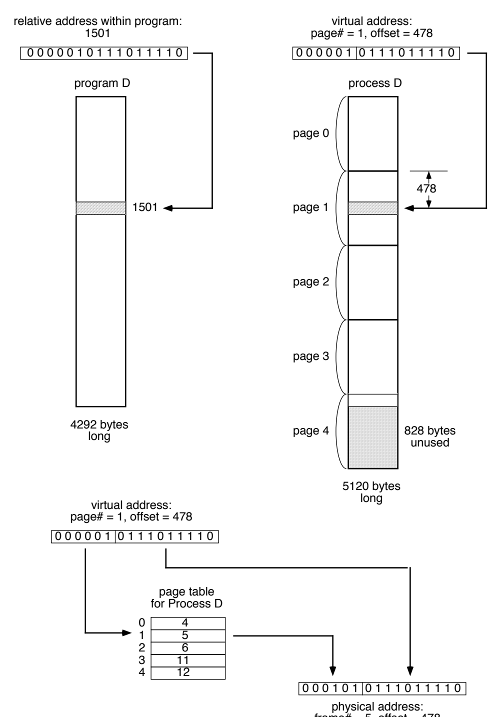


- A *page table* keeps track of every page in a particular process
  - Each entry contains the corresponding frame in main (physical) memory
  - Can add protection bits, but not as useful
- Additional hardware support required is slightly less than for segmentation
  - No need to keep track of, and compare to, limit. Why not?

Paging Example



Paging Example (cont.)



Managing Pages and Frames

- 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
- Keep page tables in memory, use page table base register (special register) to point to page table of active process
  - Saved/restored as part of context switch
  - Page table also contains:
    - Other bits for demand paging (discussed next time)

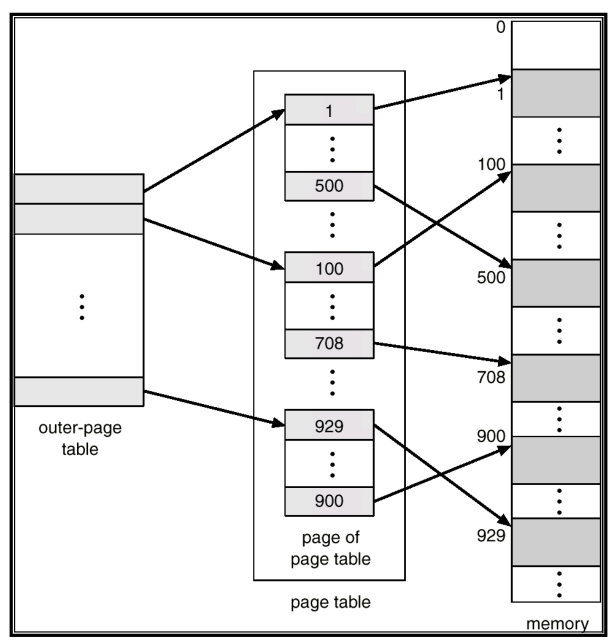
Evaluation of Paging

- Advantages:
  - Easy to allocate memory — keep a list of available frames, and simply grab first one that's free
  - Easy to swap — pages, frames, and often disk blocks as well, all are same size
  - One frame is just as good as another!
- Disadvantages:
  - Page tables are fairly large
    - Most page tables are too big to fit in registers, so they must live in physical memory
    - This table lookup adds an extra memory reference for every address translation
  - Internal fragmentation
    - Always get a whole page, even for 1 byte
    - Larger pages makes the problem worse
    - Average = 1/2 page per process

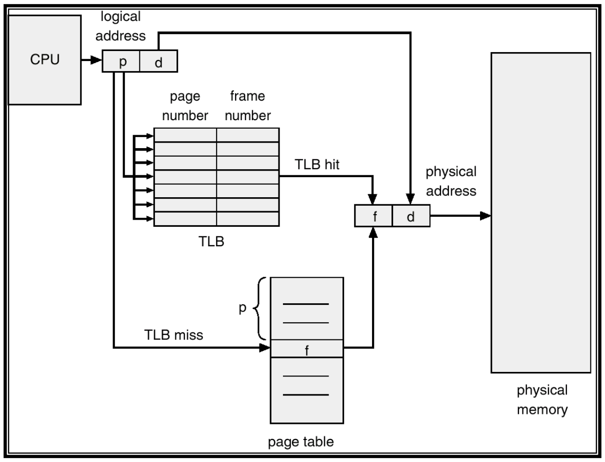
Address Translation, Revisited

- A modern microprocessor and OS has maybe a 32 bit logical address space per process (2^{32} = 4 GB)
  - If page size is 4k (2^{12}), 32–12=20, meaning each page table could have up to 2^{20} (approximately 1 million) page entries, each maybe 4 bytes long = 4MB
  - Problem: page table is too large to store in one page, can't store contiguously
    - Two-level page tables: page tables are also stored in each process' logical memory
  - New problem: memory access time may double since the page tables are now subject to paging
    - (one access to get info from page table, plus one access to get data from memory)
    - New solution: use a special cache (called a Translation Lookaside Buffer (TLB)) to cache page table entries

Two-Level Page Table



Translation Look-Aside Buffer



Paging and Segmentation

- Use two levels of mapping:
  - Process is divided into variable-size segments
    - Segments are logical divisions as before
  - Each segment is divided into many small fixed-size pages
    - Pages are easy for OS to manage
    - Eliminates external fragmentation
  - Logical address = segment, page, offset
  - One segment table per process, one page table per segment
- Sharing at two levels: segment, page
  - Share frame by having same frame reference in two page tables
  - Share segment by having same base in two segment tables
  - Still need protection bits (sharing, r/o, r/w)