Chapter 9: Main Memory / Part One

- for a program to be run, it must be brought (from disk) into memory and placed within a process.
- CPU can directly access the main memory and registers only.
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing stall.
- 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.
- programs on disk, ready to be brought into memory to execute from an input queue
  - must be loaded to address 0000 without support
- addresses represented in different ways at different stages of a program life
  - Sourse code address usually symbolic
  - Compiled code addresses bind to relocatable addresses
  - linker or loader will bind relocatable addresses to absolute addresses
  - Each binding maps one address space to another.

- Stages of address binding of instruction and data to memory addresses:
  - Compile time: if memory location known 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 memory segment to another.
    Need hardware support for address maps (base and limit reg)
- Logical address: generated by the CPU (or 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. they differ in execution time address-binding schemes.
- Logical address space: the set of all logical addresses generated by a program.
- Physical address space: the set of all physical addresses generated by a program
- MMU (Memory Management Unit): Hardware device that at run time maps virtual to physical address

- Base register can be called relocation register when adding its value to every address generated by a user process at the time it is sent to memory.

- The user program only deals with logical 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.
  - no special support is required.
    - implemented through program design
    - OS can help by providing libraries to implement dynamic loading
  - Useful when large amount of code are needed to handle infrequently occuring cases.

- Static Linking: System libraries and program code combined by the Loader into the binary program image.
  - Consider applicability to patching system libraries (versioning may be needed)
- Dynamic Linking: linking postponed until execution time.
  - Particularly useful for libraries. known as Shared libraries
  - Stub (small piece of code)
    - Used to locate the appropriate memory-resident library routine.
    - replaces itself with the address of the routine, and executes it.
  - OS checks if routine is in processes' memory address, if not in address space, it will be added to it.

- 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 processes is swapped out so higher-priority processes 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.

- Swapped out processed do not need to be swapped in ~~into~~ to same physical address, it depends on address binding method.

- Context switch can be very high if next process to be put on CPU is not in memory.
  - Can reduce if reduce size of memory swapped
    [by knowing how much memory really being used]

- System calls to inform OS of memory use:
  1) request_memory() 2) release_memory()

Chapter 9: Main Memory / Part two

- Swapping constraints:
  - Pending I/O
    - ↳ Can't swap out as I/O would occur to wrong process
  - always transfer I/O to kernal space, then to I/O device (double buffering)
    - ↳ adds overhead.

- Standard swapping not used in modern OS

- modified version of swapping → 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
  - It uses other methods to free memory if needed (low)

- Main memory must accommodate both the OS and the various user processes, so therefore we need to allocate it in most efficient way.

- Memory is divided into two partitons, one for the resident OS and one for the user processes

- The OS can be placed in either low or high memory. the major factor affecting this decision is the location of the interrupt vector.
  - → the interrupt vector is often in low memory, thus the OS is usually held in low memory. → then user processes are held in high memory

- In contiguous memory allocation, each process is contained in a single contiguous section of memory.

- Relocation registers used to protect user processes from each other, and from changing OS 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.
  - Now actions can be allowed.
    ex: kernal code being transient - kernal changing size

↓ Fixed-partitions

- Multiple-partition allocation: memory is divided into several fixed-sized partitions, Each partition may contain exactly one process
  ⟹ the degree of multiprogramming is bound by the number of partitions.

- When a ~~process~~ partition is free, a process is selected from the input queue and is loaded into the free partition.
- If the process is terminated, its partitions become available for other processes.

↳ fixed-partitions

- Fixed-partitions is no loger used, variable-partiton sizes is now used for efficiency [Sized to a given process' needs]

- Hole: block of available memory
  ↳ Holes of various size are scattered throughout memory.

- Initially, all memory is available for user processes and considered as one large block of available memory.
- when a process arrives, it is allocated memory a hole large enough to accommodate it.
- Process exiting frees its partition
- Operating System maintains information about:
  - allocated partitions - free partitions (holes)
- Solutions for the dynamic Storage-Allocation problem:
  - 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"
  - Worst-fit: Allocate the largest hole
    Must Search entire list / produces the largest leftover hole.
- First-fit and worst-fit better than worst-fit in terms of Speed and storage utillization
- External Fragmentation: total memory space exists to satisfy a request, but it is not contiguous.
- Internal Fragmentation: allocated memory may be slightly larger than request memory
  [this size difference is memory internal to partitions, but not being used
- 50-percent rule:
  First-fit analysis reveals that given N blocks allocated, 0.5N blocks lost to fragmentation. (1/3 may be unusable)

- do I/O only into OS buffer

- 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
- Segmentation: a memory-managment scheme that ~~reduces~~ 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/global variables - common block
    - symbole table - arrays

- Segment Architecture
  - Logical address consists of a two tuple: <segment-number, offset>
  - Segment table: maps two-dimensional physicad addresses, each table entiry has:
    - base: contains the starting physical address where the Segment reside in memory.
    - limit: specifies the length of the segment.

- Segment-table base register (STBR): points the segment table's location in memory.

- Segment-table length register (STLR): indecates the number of segment used by a program.

Segments number s is legal if $s < STLR$

Chapter 9 : Main Memory / Part three

- Protection : with each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment.
  - read / write / execute privileges.

  → Protection bit associated with segments, code sharing occurs at sharing level

- Memory allocation is a dynamic storage-allocation problem, because segment vary in length.

- Paging
- physical address space of a process can be noncontiguous
  → process is allocated to physical memory whenever it is available.
  → this will help : • Avoid external fragmentation
  • Avoid problem of varing sized memory chunks.

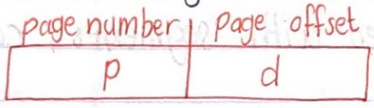
Note: we still have internal fragmentation

- Frames: divide physical memory into fixed-sized blocks.
  - size is power of 2, between 512 bytes and 16 M bytes.
- Pages: divide logical memory into blocks of same size.

keep track of all free frames
loading a program of size N pages, need to find N free frames.

- Page table: used to translate logical to physical addresses.
- Backing store likewise split into pages.

- Address generated by CPU is divided into:
  - page number (p): used as an index into a page table that 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



Logical address space 2^m and page size 2^n

- 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
- every data/instruction access requires two memory accesses
  - ↳ one for the page table and one for the data/instruction
  - ↳ Can be solved by the use of a special fast-lookup hardware cash called associative memory or translation look-aside buffers (TLBs)
- TLBs
  - ↳ - Some store address-space identifires (ASIDs) in each TLB entry uniquely identifires each process to provide address space protection for that process. / otherwise need to flush at every context switch
  - typically small (64 to 1,024 entries)
  - on TLB miss, value is loaded into the TLB for faster access next time
    - Replacement policies must be considered.
    - Some entries can be ~~down~~ wired down for permenent fast access

- for associative memory, if p is in associative register, get frame # out
  Otherwise get frame # from page table in memory.

- Calculating internal fragmentation:
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  ↳ 35 pages + 1086 bytes

  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - worst case fragmentation = 1 frame - 1 byte
  - On average case fragmentation = 1/2 fram size

  - each page table entry takes memory to track.
  - Page sizes growing over time.
  - process view and physical memory ~~diffs~~ are different
  - By implementation process can only access its own memory.

- Associative lookup = ε time unit.
  ↳ ε < 10% of memory access time.

- Hit ratio : percentage of times that a page number is found in the associative registers or Ratio related to number of associative registers

- Example: Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access.
  Effective access time (EAT)
  EAT = α*(memory access time) + (1-α)*(memory access time)

- Memory Protection: implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed.
  - more bits can be added to indicate page execute-only, and so on.

- Valid-Invalid bit : is attached to each entry in the page table to indicate:
  - "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 kernal

- Shared code: One copy of read-only (reentrant) code shared among processes (i.e: text editors, compilers, window systems) accessed many times
  - Similar to multiple threads sharing the same process space.
  - 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.

Chapter 9: Main Memory / Part four

- Structure of the page table
  - Memory structures for paging can get huge using straight-forward method.
    - Consider a 32-bit logical address space as on modern compute
    - Page size of 4KB (2^{12})
    - Page table would have ($2^{32}/2^{12}$) entries
    - If each entry is 4 bytes; 4MB 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
- Could be fixed using:
  - Hierarchical paging
  - Hash page tables
  - Inverted page tables

- Hierarchical ~~paging~~ page table
  - ~~Breack~~ Break up the logical address space into multiple page table
  - A simple technique is a two-level page table.
  - we page the page table.

- Hashed page tables.
  - common in address spaces > 32 bits.
  - The virtual page number is hashed into a page table.
  - the page table contains a chain of elements hashing to the same location (entries in)
  - Each element in page table contains:
    - the virtual page number
    - the value of the mapped page frame
    - 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 hased 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 scatterd)

- Inverted page table.
  - Rather than each process having a page table and keeping track of all possible logical ~~space~~ pages, track all physical pages.
  - One entry for each real page of memory.
  - Entry consists of the virtual address of the page stored in the 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 few- page table entries.
    - TLB can accelerate access.
  - How to implement it?
    - One mapping of a virtual address to the shared physical address