Memory Management

"Multitasking without memory management is like having a party in a closet."

– Charles Petzold. *Programming Windows 3.1*

"Programs expand to fill the memory that holds them."

Preparing a program for execution

- Development of programs
  - *Source* program
  - Compilation/Assembly to get *Object* program
  - Linking / Linkage editors to get *Relocatable load module*
  - Loading to get *Load module*
- Memory
  - Large array of words (or bytes)
  - Unique address of each word
  - CPU fetches from and stores into memory addresses
- Instruction execution cycle
  - Fetch an instruction (opcode) from memory
  - Decode instruction
  - Fetch operands from memory, if needed
  - Execute instruction
  - Store results into memory, if necessary
- Memory unit sees only the addresses, and not how they are generated (instruction counter, indexing, direct)
- Address Binding
  - Binding – Mapping from one address space to another
  - Program must be loaded into memory before execution
  - Loading of processes may result in relocation of addresses
    - Link external references to entry points as needed
  - User process may reside in any part of the memory
  - Symbolic addresses in source programs (like `i`)
  - Compiler *binds* symbolic addresses to relocatable addresses, assumed to start at location zero
  - Linkage editor or loader *binds* relocatable addresses to absolute addresses
  - Types of binding
    - Compile time binding
      - Binding of absolute addresses by compiler
      - Possible only if compiler knows the memory locations to be used
      - MS-DOS .com format programs
    - Load time binding
      - Based on relocatable code generated by the compiler
      - Final binding delayed until load time

        - If change in starting address, reload the user code to incorporate address changes
    - Execution time binding
        - Process may be moved from one address to another during execution
        - Binding must be delayed until run time
        - Requires special hardware

- Relocation
    - Compiler may work with *assumed* logical address space when creating an object module
    - Relocation – Adjustment of operand and branch addresses within the program
    - Static Relocation
        - Similar to compile time binding
        - Internal references
            - References to locations within the same program address space
            - Earliest possible moment to perform binding at the time of compilation
            - If bound at compile time, compiler must have the actual starting address of object module
            - Early binding is restrictive and rarely used, but may result in better run-time performance, specially in loops

```
int x[5], i, j;
for ( i = 0; i < n; i++ )
{
    for ( j = 0; j < 5; j++ )
        x[j] = /* some computation */

    /* Other part of loop using x  */
}
```

```
for ( int i ( 0 ); i < n; i++ )
{
    int x[5];
    for ( int j ( 0 ); j < n; j++ )
        x[j] = /* some computation */

    /* Other part of loop using x  */
}
```

        - External references
            - References to locations within the address space of other modules or functions
            - More practical
            - All modules to be linked together must be known to resolve references
            - Linking loader
            - With static relocation, load module is not relocatable and the loader simply copies the load module into memory
        - Separate linkage editor and loader
            - More flexible
            - Starting address need not be known at linking time
            - Absolute physical addresses bound only at the time of loading
            - Relocatable physical addresses bound by relocating the complete module
            - The program gets relocated twice – once for linking and then for loading
    - Dynamic Relocation
        - All object modules kept on disk in relocatable load format
        - Relocation at runtime immediately precedes *each* storage reference
        - Invisible to all users (except for system programmers)
        - Forms the basis for *virtual memory*
        - Permits efficient use of main storage
        - Binding of physical addresses can be delayed to the last possible moment
        - When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded into memory
        - Relocatable linking loader can load the new routine if needed

    - Unused routine is never loaded
    - Useful to handle large amount of infrequently used code (like error handlers)

- Linking
  - Allows independent development and translation of modules
  - Compiler generates *external symbol table*
  - Resolution of external references
    - Chain method
      - Using chain of pointers in the module
      - Chain headed by corresponding entry in the external symble table
      - Resolution by linking at the end (when the address is known) through external symbol table
      - Last element in the chain is `NULL`
      - External symbol table not a part of the final code
    - Indirect addressing
      - External symbol table a permanent part of the program
      - Transfer vector approach
      - Each reference is set to point to the entry in the external symbol table
      - External symbol table reflects the actual address of the reference when known
      - Linking faster and simpler than chaining method but two memory references required at run time to complete the reference
  - *Static Linking*
    - All external references are resolved before program execution
  - *Dynamic Linking*
    - External references resolved during execution
    - *Dynamically linked libraries*
      - Particularly useful for system libraries
      - Programs need to have a copy of the language library in the executable image, if no dynamic linking
      - Include a *stub* in the image for each library-routine reference
      - Stub

        Small piece of code

        Indicates the procedures to locate the appropriate memory resident library routine

        Upon execution, stub replaces itself with the routine and executes it

        Repeated execution executes the library routine directly

        Useful in library updates or bug fixes

        A new version of the library does not require the programs to be relinked

        Library version numbers to ensure compatibility

Implementation of memory management

- Done through *memory tables* to keep track of both real (main) as well as virtual (secondary) memory
- Memory management unit (MMU)
  - Identifies a memory location in ROM, RAM, or I/O memory given a physical address
  - Does *not* translate physical address
  - Physical address is provided by an address bus to initiate the movement of code or data from one platform device to another

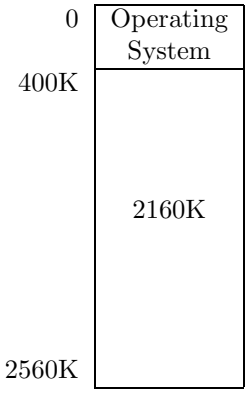
  - Physical address is generated by devices that act as *bus masters* on the address buses (such as CPU)
  - Frame buffers and simple serial ports are *slave devices* that respond to the addresses
- In Unix, memory can be accessed as a device
  - Entails a larger overhead in doing so
  - You have to go through the I/O system through a system call to talk to memory, rather than doing it directly by one machine instruction
  - Useful only when trying to access otherwise protected memory
  - `ps` digs up information about other processes by reading `/dev/kmem` (kernel memory) and `/dev/mem` (physical memory)

Simple Memory Management Schemes

- Shortage of main memory due to
  - Size of many applications
  - Several active processes may need to share memory at the same time
- Fixed Partition Memory Management
  - Simplest memory management scheme for multiprogrammed systems
  - Divide memory into fixed size *partitions*, possibly of different size
  - Partitions fixed at system initialization time and may not be changed during system operation
  - Single-Partition Allocation
    - User is provided with a bare machine
    - User has full control of entire memory space
    - Advantages
      - Maximum flexibility to the user
      - User controls the use of memory as per his own desire
      - Maximum possible simplicity
      - Minimum cost
      - No need for special hardware
      - No need for operating system software
    - Disadvantages
      - No services
      - OS has no control over interrupts
      - No mechanism to process system calls and errors
      - No space to provide multiprogramming
  - Two-Partition Allocation
    - Memory divided into two partitions
      - Resident operating system
      - User memory area
    - Linux divides the memory into kernel space and user space
    - OS placed in low memory or high memory depending upon the location of interrupt vector
    - Need to protect OS code and data from changes by user processes
      - Protection must be provided by hardware
      - Can be implemented by using base-register and limit-register
    - Loading of user processes

        - First address of user space must be beyond the base register
        - Any change in base address requires recompilation of code
        - Could be avoided by having relocatable code from the compiler
        - Base value must be *static* during program execution
        - OS size cannot change during program execution
          Change in buffer space for device drivers
          Loading code for rarely used system calls
          Transient OS code
    * Handling transient code
        - Load user processes into high memory down to base register
          Allows the use of all available memory
        - Delay address binding until execution time
          Base register known as the *relocation register*
          Value in base register added to every address reference
          User program never sees the real physical addresses
          User program deals only with logical addresses

- Multiple-Partition Allocation
  - Necessary for multiprogrammed systems
  - Allocate memory to various processes in the wait queue to be brought into memory
  - Simplest scheme
    * Divide memory into a large number of fixed-size partitions
    * One process to each partition
    * Degree of multiprogramming bound by the number of partitions
    * Partitions allocated to processes and released upon process termination
    * Originally used by IBM OS/360 (MFT)
    * Primarily useful in a batch environment
  - Variable size partitions – Basic implementation
    * Keep a table indicating the availability of various memory partitions
    * Any large block of available memory is called a *hole*
    * Initially the entire memory is identified as a large hole
    * When a process arrives, the allocation table is searched for a large enough hole and if available, the hole is allocated to the process
    * Example
        - Total memory available – 2560K
        - Resident OS – 400K
        - User memory – 2160K



Job Queue

Process	Memory	Time
p_1	600K	10
p_2	1000K	5
p_3	300K	20
p_4	700K	8
p_5	500K	15

  - Set of holes of various sizes scattered throughout the memory
  - Holes can grow when jobs in adjacent holes are terminated
  - Holes can also diminish in size if many small jobs are present
  - Problem of *fragmentation*
    * Division of main memory into small holes not usable by any process
    * Enough total memory space exists to satisfy a request but is fragmented into a number of small holes
    * Possibility of starvation for large jobs
  - Used by IBM OS/MVT (multiprogramming with variable number of tasks, 1969)
  - Dynamic storage allocation problem
    * Selects a hole to which a process should be allocated
    * First-fit strategy
      · Allocate first hole that is big enough
      · Stop searching as soon as first hole large enough to hold the process is found
    * Best-fit strategy
      · Allocate the smallest hole that is big enough
      · Entire list of holes is to be searched
      · Search of entire list can be avoided by keeping the list of holes sorted by size
    * Worst-fit strategy
      · Allocate the largest available hole
      · Similar problems as the best-fit approach
  - Memory Compaction
    * Shuffle the memory contents to place all free memory into one large hole
    * Possible only if the system supports dynamic relocation at execution time
    * Total compaction
    * Partial compaction
    * Dynamic memory allocation in C
      · C heap manager is fairly primitive
      · The `malloc` family of functions allocates memory and the heap manager takes it back when it is `free`d
      · There is no facility for heap compaction to provide for bigger chunks of memory
      · The problem of fragmentation is for real in C because movement of data by a heap compactor can leave incorrect address information in pointers
      · Microsoft Windows has heap compaction built in but it requires you to use special memory handles instead of pointers
      · The handles can be temporarily converted to pointers, after locking the memory so the heap compactor cannot move it

- Overlays
  - Size of process is limited to size of available memory
  - Technique of *overlaying* employed to execute programs that cannot be fit into available memory
  - Keep in memory only those instructions and data that are needed at any given time
  - When other instructions are needed, they are loaded into space previously occupied by instructions that are not needed
  - A 2-pass assembler

Pass 1	70K	Generate symbol table
Pass 2	80K	Generate object code
Symbol table	20K	
Common routines	30K	

Total memory requirement – 200K
Available memory – 150K

- Divide the task as into overlay segments

Overlay 1	Overlay 2
Pass 1 code	Pass 2 code
Symbol table	Symbol table
Common routines	Common routines
Overlay driver (10K)	

- Code for each overlay kept on disk as absolute memory images
- Requires special relocation and linking algorithms
- No special support required from the OS
- Slow to execute due to additional I/O to load memory images for different parts of the program
- Programmer completely responsible to define overlays

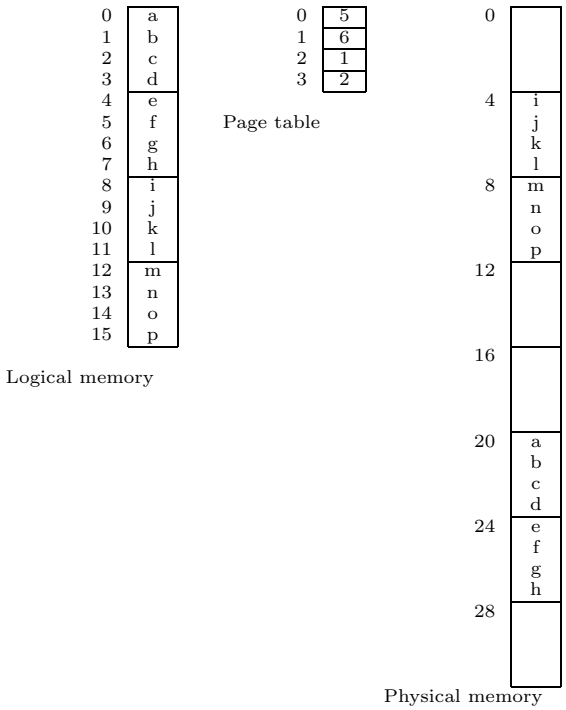
Principles of Virtual Memory

- Hide the real memory from the user
- Swapping
  - Remove a process temporarily from main memory to a *backing store* and later, bring it back into memory for continued execution
  - Swap-in and Swap-out
  - Suitable for round-robin scheduling by swapping processes in and out of main memory
    - Quantum should be large enough such that swapping time is negligible
    - Reasonable amount of computation should be done between swaps
  - *Roll-out, Roll-in Swapping*
    - Allows to preempt a lower priority process in favor of a higher priority process
    - After the higher priority process is done, the preempted process is continued
  - Process may or may not be swapped into the same memory space depending upon the availability of execution time binding
  - Backing store
    - Preferably a fast disk
    - Must be large enough to accommodate copies of all memory images of all processes
    - Must provide direct access to each memory image
    - Maintain a ready queue of all processes on the backing store
    - Dispatcher brings the process into memory if needed
  - Calculation of swap time
    - User process of size 100K
    - Backing store – Standard head disk with a transfer rate of 1 MB/sec
    - Actual transfer time – 100 msec
    - Add latency (8 msec) – 108 msec
    - Swap-in + Swap-out – 216 msec

  - Total transfer time directly proportional to the amount of memory swapped
  - Swap only completely idle processes (not with pending I/O)

- Multiple Base Registers
  - Provides a solution to the fragmentation problem
  - Break the memory needed by a process into several parts
  - One base register corresponding to each part with a mechanism to translate from logical to physical address

- Paging
  - Permits a process' memory to be noncontiguous
  - Avoids the problem of fitting varying-sized memory segments into backing store
  - First used in ATLAS computer in 1962
  - Physical memory is divided into a number of equal-sized contiguous blocks, called *page frames*
    * In the past, common block sizes were 512 bytes or 1K
    * Common block sizes have been increasing in size, with Solaris allowing for a frame size of 8K
    * Block size in Unix can be determined by the C library function `getpagesize(3C)`
  - Hardware requirements
    * Logical memory broken into blocks of same size as page frame size, called *pages*
    * To execute, pages of process are loaded into frames from backing store
    * Backing store divided into fixed size blocks of the same size as page or frame
    * Every address generated by the CPU divided into two parts
      · Page number p
      · Page offset d
    * Page number used as index into a *page table*
      · Page table contains the base address of each page in memory
    * Page offset defines the address of the location within the page
    * Page size 2^n bytes
      · Low order n bits in the address indicate the page offset
      · Remaining high order bits designate the page number
    * Example – Page size of four words and physical memory of eight pages



- Scheduling processes in a paged system
  - Each page an instance of memory resource
  - Size of a process can be expressed in pages
  - Available memory known from the list of unallocated frames
  - If the process' memory requirement can be fulfilled, allocate memory to the process
  - Pages loaded into memory from the list of available frames
  - Example

free-frame list			free-frame list		
14	13	unused	15	13	page 1
13	14	unused		14	page 0
18	15	unused		15	unused
20	16			16	
15	17			17	
	18	unused		18	page 2
new process	19		new process	19	
page 0	20	unused	page 0	20	page 3
page 1	21		page 1	21	
page 2			page 2		
page 3			page 3		

new process page table

0	14
1	13
2	18
3	20

  - No external fragmentation possible with paging
  - Internal fragmentation possible – an average of half a page per process
  - Paging allows programs larger than available main memory to be executed without using overlay structure, using a technique called *demand paging*

  - Page size considerations
    * Small page size ⇒ More overhead in page table plus more swapping overhead
    * Large page size ⇒ More internal fragmentation
  - Implementation of page table
    * Simplest implementation through a set of dedicated registers
      · Registers reloaded by the CPU dispatcher
      · Registers need to be extremely fast
      · Good strategy if the page table is very small (< 256 entries)
      · Not satisfactory if the page table size is large (like a million entries)
    * *Page-Table Base Register*
      · Suitable for large page tables
      · Pointer to the page table in memory
      · Achieves reduction in context switching time
      · Increase in time to access memory locations
    * *Associative registers*
      · Also called *translation look-aside buffers*
      · Two parts to each register
        1. key
        2. value
      · All keys compared simultaneously with the key being searched for
      · Expensive hardware
      · Limited part of page table kept in associative memory
      · *Hit ratio* – Percentage of time a page number is found in the associative registers
      · Effective memory access time

- Shared pages
  - Possible to share common code with paging
  - Shared code must be reentrant (pure code)
    * Reentrant code allows itself to be shared by multiple users concurrently
    * The code cannot modify itself and the local data for each user is kept in separate space
    * The code has two parts
      1. Permanent part is the instructions that make up the code
      2. Temporary part contains memory for local variables for use by the code
    * Each execution of the permanent part creates a temporary part, known as the *activation record* for the code
    * Requirements for a function to be classified as reentrant
      1. All data is allocated on the stack
      2. No global variables
      3. Function can be interrupted at any time without affecting its execution
      4. Function doesn't call any other function that is not reentrant
  - Separate data pages for each process
  - Code to be shared – text editor, windowing system, compilers

- Memory protection in paging systems
  - Accomplished by protection bits associated with each frame
  - Protection bits kept in page table

  - Define the page to be read only or read and write
  - Protection checked at the time of page table reference to find the physical page number
  - Hardware trap or memory protection violation
  - *Page table length register*
    * Indicates the size of the page table
    * Value checked against every logical address to validate the address
    * Failure results in trap to the OS

- Logical memory vs Physical memory
  - Logical memory
    * Provides user's view of the memory
    * Memory treated as one contiguous space, containing only one program
  - Physical memory
    * User program *scattered* throughout the memory
    * Also holds other programs
  - Mapping from logical addresses to physical addresses hidden from the user
  - System could use more memory than any individual user
  - Allocation of frames kept in *frame table*

- Segmentation
  - User prefers to view memory as a collection of variable-sized segments, like arrays, functions, procedures, and main program
  - No necessary order in the segments
  - Length of each segment is defined by its purpose in the program
  - Elements within a segment defined by their offset from the beginning of segment
    * First statement of the procedure
    * Seventeenth entry in the symbol table
    * Fifth instruction of the `sqrt` function
  - Logical address space considered to be collection of segments
  - A name and length for each segment
  - Address – Segment name and offset within the segment
    * Segment name to be explicitly specified unlike paging
  - The only memory management scheme available on Intel 8086
    * Memory divided into *code*, *data*, and *stack* segments
  - Hardware for segmentation
    * Mapping between logical and physical addresses achieved through a *segment table*
    * Each entry in segment table is made up of
      · Segment *base*
      · Segment *limit*
    * Segment table can be abstracted as an array of *base-limit register pairs*
    * Two parts in a logical address
      1. Segment name/number s
         · Used as an index into the segment table
      2. Segment offset d

            - Added to the segment base to produce the physical address
            - Must be between 0 and the segment limit
            - Attempt to address beyond the segment limit results in a trap to the OS
    - Implementation of segment tables
        - Kept either in registers or in memory
        - Segment table base register
        - Segment table length register
        - Associative registers to improve memory access time
    - Protection and sharing
        - Segments represent a semantically defined portion of a program
        - Protection and sharing like paging
        - Possible to share parts of a program
            - Share the `sqrt` function segment between two independent programs
    - Fragmentation
        - Memory allocation becomes a dynamic storage allocation problem
        - Possibility of external fragmentation
            - All blocks of memory are too small to accommodate a segment
        - Compaction can be used whenever needed (because segmentation is based on dynamic relocation)
        - External fragmentation problem is also dependent on average size of segments

- Paged Segmentation
    - Used in the MULTICS system
    - Page the segments
    - Separate page table for each segment
    - Segment table entry contains the base address of a page table for the segment
    - Segment offset is broken into page number and page offset
    - Page number indexes into page table to give the frame number
    - Frame number is combined with page offset to give physical address
    - MULTICS had 18 bit segment number and 16 bit offset
    - Segment offset contained 6-bit page number and 10-bit page offset
    - Each segment limited in length by its segment-table entry
      Page table need not be full-sized; Requires only as many entries as needed
    - On an average, half a page of internal fragmentation per segment
    - Eliminated external fragmentation but introduced internal fragmentation and increased table space overhead

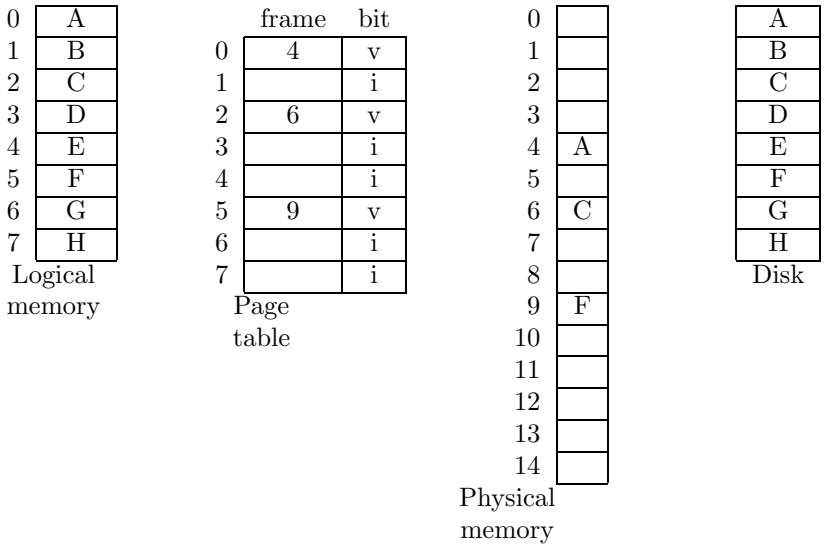
Implementation of Virtual Memory

- Allows the execution of processes that may not be completely in main memory
- Programs can be larger than the available physical memory
- Motivation
    - The entire program may not need to be in the memory for execution
    - Code to handle unusual conditions may never be executed
    - Complex data structures are generally allocated more memory than needed (like symbol table)

  - Certain options and features of a program may be used rarely, like text-editor command to change case of all letters in a file

- Benefits of virtual memory
  - Programs not constrained by the amount of physical memory available
  - User does not have to worry about complex techniques like overlays
  - Increase in CPU utilization and throughput (more programs can fit in available memory)
  - Less I/O needed to load or swap user programs

- *Demand Paging*
  - Similar to a paging system with swapping
  - Rather than swapping the entire process into memory, use a "lazy swapper"
  - Never swap a page into memory unless needed
  - Distinction between *swapper* and *pager*
    * Swapper swaps an entire process into memory
    * Pager brings in individual pages into memory
  - In the beginning, pager guesses the pages to be used by the process
  - Pages not needed are not brought into memory
  - Hardware support for demand paging
    * An extra bit attached to each entry in the page table – *valid-invalid bit*
    * This bit indicates whether the page is in memory or not
    * Example



  - For *memory-resident pages*, execution proceeds normally
  - If page not in memory, a *page fault trap* occurs
  - Upon page fault, the required page brought into memory
    * Check an internal table to determine whether the reference was valid or invalid memory access
    * If invalid, terminate the process. If valid, page in the required page
    * Find a free frame (from the free frame list)
    * Schedule the disk to read the required page into the newly allocated frame
    * Modify the internal table to indicate that the page is in memory

    - Restart the instruction interrupted by page fault
  - *Pure demand paging* – Don't bring even a single page into memory
  - *Locality of reference*
- Performance of demand paging
  - Effective access time for demand-paged memory
    - Usual memory access time (m) – 10 to 200 nsec
    - No page faults $\Rightarrow$ Effective access time same as memory access time
    - Probability of page fault $= p$
    - p expected to be very close to zero so that there are few page faults
    - Effective access time $= (1 - p) \times m + p \times$ page fault time
    - Need to know the time required to service page fault
- What happens at page fault?
  - Trap to the OS
  - Save the user registers and process state
  - Determine that the interrupt was a page fault
  - Check that the page reference was legal and determine the location of page on the disk
  - Issue a read from the disk to a free frame
    - Wait in a queue for the device until the request is serviced
    - Wait for the device seek and/or latency time
    - Begin the transfer of the page to a free frame
  - While waiting, allocate CPU to some other process
  - Interrupt from the disk (I/O complete)
  - Save registers and process state for the other user
  - Determine that the interrupt was from the disk
  - Correct the page table and other tables to show that the desired page is now in memory
  - Wait for the CPU to be allocated to the process again
  - Restore user registers, process state, and new page table, then resume the interrupted instruction
- Computation of effective access time
  - Bottleneck in read from disk
    - Latency time – 8 msec
    - Seek time – 15 msec
    - Transfer time – 1 msec
    - Total page read time – 24 msec
  - About 1 msec for other things (page switch)
  - Average page-fault service time – 25 msec
  - Memory access time – 100 nanosec
  - Effective access time
    $$\begin{aligned} &= (1 - p) \times 100 + p \times 25,000,000 \\ &= 100 + 24,999,900 \times p \\ &\approx 25 \times p\text{msec} \end{aligned}$$
  - Assume 1 access out of 1000 to cause a page fault
    - Effective access time – 25 μsec

    - Degradation due to demand paging – 250%
  - For 10% degradation

$$\begin{aligned} 110 &> 100 + 25,000,000 \times p \\ 10 &> 25,000,000 \times p \\ p &< 0.0000004 \end{aligned}$$

    Reasonable performance possible through less than 1 memory access out of 2,500,000 causing a page fault

Page Replacement

- Limited number of pages available in memory
- Need to optimize swapping (placement and replacement)
- Increase in multiprogramming through replacement optimization
- Increase in degree of multiprogramming ⇒ overallocation of memory
- Assume no free frames available
  - Option to terminate the process
    - Against the philosophy behind virtual memory
    - User should not be aware of the underlying memory management
  - Option to swap out a process
    - No guarantee that the process will get the CPU back pretty fast
  - Option to replace a page in memory
- Modified page-fault service routine
  - Find the location of the desired page on the disk
  - Find a free frame
    - If there is a free frame, use it
    - Otherwise, use a *page replacement algorithm* to find a *victim* frame
    - Write the victim page to the disk; change the page and frame tables accordingly
  - Read the desired page into the (newly) free frame; change the page and frame tables
  - Restart the user process
- No free frames ⇒ two page transfers
- Increase in effective access time
- *Dirty bit*
  - Also known as *modify bit*
  - Each frame has a dirty bit associated with it in hardware
  - Dirty bit is set if the page has been modified or written into
  - If the page is selected for replacement
    - Check the dirty bit associated with the frame
    - If the bit is set write it back into its place on disk
    - Otherwise, the page in disk is same as the current one
- Page replacement algorithms
  - Aim – to minimize the page fault rate

- Evaluate an algorithm by running it on a particular string of memory references and compute the number of page faults
- Memory references string called a *reference string*
- Consider only the page number and not the entire address
- Address sequence

  0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101,
  0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103,
  0104, 0101, 0609, 0102, 0105

- 100 byte to a page
- Reference string

  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

- Second factor in page faults – Number of pages available
- More the number of pages, less the page faults
- FIFO Replacement Algorithm
  * Associate with each page the time when that page was brought in memory
  * The victim is the oldest page in the memory
  * Example reference string

    $$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

  * With three pages, page faults as follows:

7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

  * May reduce a page that contains a heavily used variable that was initialized a while back
  * Bad replacement choice ⇒ Increase in page fault rate
  * Consider another reference string

    $$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$$

  * With three pages, the page faults are:

1	1	1	4	4	4	5	5	5
	2	2	2	1	1	1	3	3
		3	3	3	2	2	2	4

  * With four pages, the page faults are:

1	1	1	1	5	5	5	5	4	4
	2	2	2	2	1	1	1	1	5
		3	3	3	3	2	2	2	2
			4	4	4	4	3	3	3

  * *Belady's Anomaly* – For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases
- Optimal Page Replacement Algorithm
  * Also called OPT or MIN
  * Has the lowest page-fault rate of all algorithms
  * Never suffers from Belady's Anomaly
  * "Replace the page that will not be used for the longest period of time"
  * Example reference string

    $$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

    * With three pages, page faults as follows:

7	7	7	2	2	2	2	2	7
	0	0	0	0	4	0	0	0
		1	1	3	3	3	1	1

    * Guarantees the lowest possible page-fault rate of all algorithms
    * Requires future knowledge of page references
    * Mainly useful for comparative studies with other algorithms
  - LRU Page Replacement Algorithm
    * Approximation to the optimal page replacement algorithm
    * Replace the page that has not been used for the longest period of time
    * Example reference string

$$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

    * With three pages, page faults as follows:

7	7	7	2	2	4	4	4	0	1	1	1
	0	0	0	0	0	0	3	3	3	0	0
		1	1	3	3	2	2	2	2	2	7

    * Implementation may require substantial hardware assistance
    * Problem to determine an order for the frame defined by the time of last use
    * Implementations based on *counters*
      · With each page-table entry, associate a time-of-use register
      · Add a logical clock or counter to the CPU
      · Increment the clock for every memory reference
      · Copy the contents of logical counter to the time-of-use register at every page reference
      · Replace the page with the smallest time value
      · Times must be maintained when page tables are changed
      · Overflow of the clock must be considered
    * Implementations based on *stack*
      · Keep a stack of page numbers
      · Upon reference, remove the page from stack and put it on top of stack
      · Best implemented by a doubly linked list
      · Each update expensive but no cost for search (for replacement)
      · Particularly appropriate for software or microcode implementations

- Stack algorithms
  - Class of page replacement algorithms that never exhibit Belady's anomaly
  - Set of pages in memory for n frames is always a *subset* of the set of pages in memory with $n + 1$ frames
- LRU Approximation Algorithms
  - Associate a *reference bit* with each entry in the page table
  - Reference bit set whenever the page is referenced (read or write)
  - Initially, all bits are reset
  - After some time, determine the usage of pages by examining the reference bit
  - No knowledge of order of use
  - Provides basis for many page-replacement algorithms that approximate replacement
  - Additional-Reference-Bits Algorithm

    - Keep an 8-bit byte for each page in a table in memory
    - At regular intervals (100 msec), shift the reference bits by 1 bit to the right
    - 8-bit shift-registers contain the history of page reference for the last eight time periods
    - Page with lowest number is the LRU page
    - 11000100 used more recently than 01110111
    - Numbers not guaranteed to be unique
- Second-Chance Algorithm
    - Basically a FIFO replacement algorithm
    - When a page is selected, examine its reference bit
    - If reference bit is 0, replace the page
    - If reference bit is 1, give the page a second chance and proceed to select the next FIFO page
    - To give second chance, reset the reference bit and set the page arrival time to current time
    - A frequently used page is always kept in memory
    - Commonly implemented by a circular queue
    - Worst case when all bits are set (degenerates to FIFO replacement)
    - Performance evaluation with reference string 7, 0, 1, 2, 0, 3, 0, 4, 2 the th reference bit as -1 or -0 at each time instance.

```
7-1 7-1 7-1 7-0 2-1 2-1 2-1 2-1 2-0 4-1 4-1
    0-1 0-1 0-0 0-0 0-1 0-1 0-1 0-0 0-0 0-0
        1-1 1-0 1-0 1-0 3-1 3-1 3-0 3-0 2-0
              +      ++      ++   +   *   *

+    - The bits got reset and there was no page fault.
++   - Page 0 got reference bit changed back to 1; no page fault.
*    - Used FIFO replacement by replacing the oldest page, whose reference bit is 0.
```

- LFU Algorithm
    - Keep counter of number of references made to each page
    - Replace the page with the smallest count
    - Motivation – Actively used page has a large reference count
    - What if a page is heavily used initially but never used again
    - Solution – Shift the counts right at regular intervals forming a decaying average usage count
- MFU Algorithm
    - Page with smallest count was probably just brought into memory and is yet to be used
    - Implementation of LFU and MFU fairly expensive, and they do not approximate OPT very well
- Using *used bit* and *dirty bit* in tandem
    - Four cases
        - (0,0) neither used nor modified
        - (0,1) not used (recently) but modified
        - (1,0) used but clean
        - (1,1) used and modified
    - Replace a page within the lowest class

Allocation of Frames

- No problem with the single user virtual memory systems
- Problem when demand paging combined with multiprogramming

- Minimum number of frames
  - Cannot allocate more than the number of available frames (unless page sharing is allowed)
  - As the number of frames allocated to a process decreases, page fault rate increases, slowing process execution
  - Minimum number of frames to be allocated defined by instruction set architecture
    - Must have enough frames to hold all the different pages that any single instruction can reference
    - The instruction itself may go into two separate pages
  - Maximum number of frames defined by amount of available physical memory
- Allocation algorithms
  - Equal allocation
    - m frames and n processes
    - Allocate m/n frames to each process
    - Any leftover frames given to *free frame buffer pool*
  - Proportional allocation
    - Allocate memory to each process according to its size
    - If size of virtual memory for process p_i is s_i, the total memory is given by $S = \sum s_i$
    - Total number of available frames – m
    - Allocate a_i frames to process p_i where
    $$a_i = \frac{s_i}{S} \times m$$
    - a_i must be adjusted to an integer, greater than the minimum number of frames required by the instruction set architecture, with a sum not exceeding m
    - Split 62 frames between two processes – one of 10 pages and the other of 127 pages
    - First process gets 4 frames and the second gets 57 frames
  - Allocation dependent on degree of multiprogramming
  - No consideration for the priority of the processes

Thrashing

- Number of frames allocated to a process falls below the minimum level ⇒ Suspend the process's execution
- Technically possible to reduce the allocated frames to a minimum for a process
- Practically, the process may require a higher number of frames than minimum for effective execution
- If process does not get enough frames, it page-faults quickly and frequently
- Need to replace a page that may be in active use
- Thrashing – High paging activity
- Process thrashing if it spends more time in paging activity than in actual execution
- Cause of thrashing
  - OS monitors CPU utilization
  - Low CPU utilization ⇒ increase the degree of multiprogramming by introducing new process
  - Use a *global page replacement algorithm*
  - May result in increase in paging activity and thrashing
  - Processes waiting for pages to arrive leave the ready queue and join the wait queue

  - CPU utilization drops
  - CPU scheduler sees the decrease in CPU utilization and increases the degree of multiprogramming
  - Decrease in system throughput
  - At a certain point, to increase the CPU utilization and stop thrashing, we must decrease the degree of multiprogramming
  - Effect of thrashing can be reduced by *local replacement algorithms*
    * A thrashing process cannot steal frames from another process
    * Thrashing processes will be in queue for paging device for more time
    * Average service time for a page fault will increase
    * Effective access time will increase even for processes that are not thrashing
  - Locality model of process execution
    * Technique to guess the number of frames *needed* by a process
    * As a process executes, it moves from locality to locality
    * Locality – Set of pages that are generally used together
  - Allocate enough frames to a process to accommodate its current locality

- Working-Set Model
  - Based on the presumption of locality
  - Uses a parameter Δ to define *working-set window*
  - Set of pages in the most recent Δ page reference is the working set
  - Storage management strategy
    * At each reference, the current working set is determined and only those pages belonging to the working set are retained
    * A program may run if and only if its entire current working set is in memory
  - Actively used page $\in$ working set
  - Page not in use drops out after Δ time units of non-reference
  - Example
    * Memory reference string

      ...
      2 6 1 5 7 7 7 7 5 1 $\quad\Delta_{t_1}$
      6 2 3 4 1
      2 3 4 4 4 3 4 4 4 4 $\quad\Delta_{t_2}$
      1 3 2 3 4 4 4 3 4 4 4
      ...

    * If $\Delta = 10$ memory references, the working set at time t_1 is $\{1, 2, 5, 6, 7\}$ and at time t_2, it is $\{2, 3, 4\}$
  - Accuracy of working set dependent upon the size of Δ
  - Let WSS_i be the working set size for process p_i
  - Then, the total demand for frames D is given by $\sum \text{WSS}_i$
  - Thrashing occurs if $D > m$
  - OS monitors the working set of each process
    * Allocate to each process enough frames to accommodate its working set
    * Enough extra frames $\Rightarrow$ initiate another process
    * $D > m \Rightarrow$ select a process to suspend
  - Working set strategy prevents thrashing while keeping the degree of multiprogramming high
  - Optimizes CPU utilization

  - Problem in keeping track of the working set
    - Can be solved by using a timer interrupt and a reference bit
    - Upon timer interrupt, copy and clear the reference bit value for each page

- Page-fault frequency
  - More direct approach than working set model
  - Measures the time interval between successive page faults
  - Page fault rate for a process too high ⇒ the process needs more pages
  - Page fault rate too low ⇒ process may have too many frames
  - Establish upper and lower-level bounds on desired page fault rate
  - If the time interval between the current and the previous page fault exceeds a pre-specified value, all the pages not referenced during this time are removed from the memory
  - PFF guarantees that the resident set grows when page faults are frequent, and shrinks when the page fault rate decreases
  - The resident set is adjusted only at the time of a page fault (compare with the working set model)

- Prepaging
  - Bring into memory at one time all pages that will be needed
  - Relevant during suspension of a process
  - Keep the working set of the process
  - Rarely used for newly created processes

- Page size
  - Size of page table
    - Smaller the page size, larger the page table
    - Each process must have its own copy of the page table
  - Smaller page size, less internal fragmentation
  - Time required to read/write a page in disk
  - Smaller page size, better *resolution* to identify working set
  - Trend toward larger page size
    - Intel 80386 – 4K page
    - Motorola 68030 – variable page size from 256 bytes to 32K

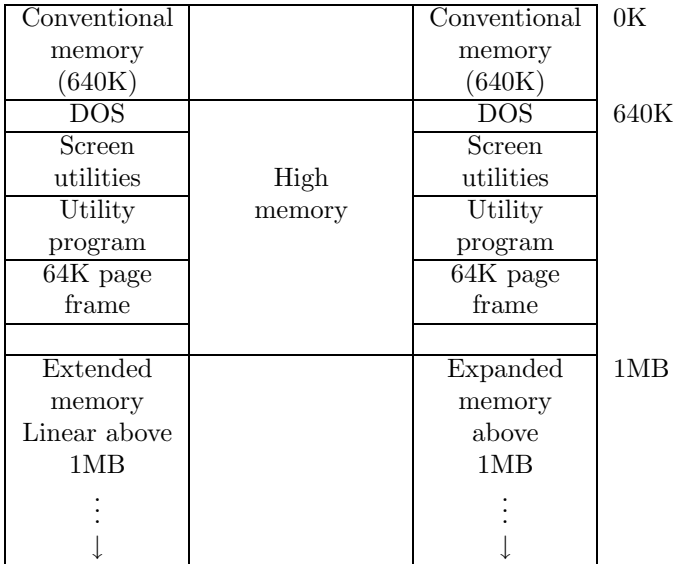
- Program structure
  - Careful selection of data structures
  - Locality of reference

Memory management in MS-DOS

- DOS uses different memory types – Dynamic RAM, Static RAM, Video RAM, and Flash RAM
  - Dynamic RAM (DRAM)
    - Used for the bulk of immediate access storage requirements
    - Data can only be accessed during refresh cycles, making DRAM slower than other memory types
    - Linked to the CPU by local bus to provide faster data transfer than standard bus
    - Chips range in capacity from 32 and 64 Kb to 16Mb

  - Static RAM (SRAM)
    * Does not require constant electrical refreshing to maintain its contents

- Memory allocation in MS-DOS
  - Determined by the amount of physical memory available and the processor
  - 386-based machines can address up to 4 GB of memory but were once limited by MS-DOS to 1 MB
  - Memory is allocated into the following components
    1. **Conventional memory**
       * Ranges from 0 to 640K
       * Used primarily for user programs
    2. **High memory**
       * Ranges from 640K to 1M
       * Used to map ROMs, video, keyboards, and disk buffers
    3. **Extended memory**
       * Memory directly above the 1M limit
       * Most programs cannot directly use this memory without a major rewrite to enable them to address it
         · Problem can be overcome by using extended memory managers, such as `HIMEM.SYS` from Microsoft
         · Such managers allow the programs to run unmodified in extended memory to the limits supported by the processor (4G on 386)
    4. **Expanded memory**
       * Not a preferred way for using additional memory
       * Slower than extended memory
       * Use a 64K "page frame" in high memory to access memory outside DOS's 1M limit
       * Program chunks of 16K are swapped into this page frame as needed, managed by expanded memory manager
       * Emulated in extended memory by using a special program such as Microsoft's `EMM386`



- Blue Screen of Death
  - Page fault (exception `0X0E`)

    - Due to the fact that application or kernel code tries to access memory at a logical address whose corresponding frame is not allocated in memory
    - Normally handled by OS by paging in the required page
    - Exception remains unhandled only if no one claims to own that region of memory, leading to BSOD, possibly due to an invalid pointer in code
        - Could be due to software problem (bug)
        - Also possible due to hardware fault, if some bit is flipped in stored pointer value
    - BSOD happens only due to the above happening in kernel; in case of application, the application is simly terminated
- General protection fault (exception `0X0D`)
    - Most common cause is attempted access to memory overriding protection settings
        - Application trying to access kernel memory
    - Occurs in kernel code if it tries to
        - Execute certain types of invalid code
        - Write to a read-only segment
    - Generally due to kernel bug or bad hardware