Memory Management

• Background
  • Programs in the Von-Neumann memory model assume to be operating on values (data and instructions) stored in memory
  • This memory is shared by multiple processes and is limited in size
  • Further, the actual programming prior to compilation uses symbolic representations of these locations which get translated into actual (or physical) memory locations
Objectives

• Memory management is concerned with providing efficient mechanisms for
  • Mapping program derived names into actual memory locations; possibly via an intermediate level of logical memory addresses
  • Utilizing the limited physical memory to bring the logical memory objects back and forth
Binding Program Names

- The compiler translates the original mnemonic (alphanumeric) "storage references" such as variable names, into logical addresses.
- This results in the user program having a set of locations 1, 2, ... L that can be viewed as a (sparse) array.
- These addresses may be bound to actual locations in the machine's memory at various times:
  - compile time
  - load time
  - run time
At Compile Time

• If the location of the process and its storage is known and fixed ahead of time
  • In this case, the mapping of logical to physical addresses can be done statically
  • A change in the physical address map will require a recompilation of the program
• This is rare for general programs, but sometimes done for OS components
At Load Time

- The compiler generates relocatable code
- The binding is done by the loader
  - when the program is brought into memory for execution
- A change in the starting address of the program will require a reload
  - for this reason, is rarely done in modern OSs
At Run-time

- Binding must be delayed until the program actually executes
  - This is because parts of the program and its storage move around all the time
  - Special hardware support for "page management" and so on are needed to accomplish this efficiently
- We will study this later on
Logical v Physical Addresses

• A *logical address* is one that is visible to the program at the time it is executing; the address in the instruction

• In the case of run-time binding of addresses, the address in the instruction gets “translated” on the fly as the operand is being fetched

• Permits actual storage and hence *physical addresses* to be different from their logical counterparts
  • the most common case
Other Issues

- Multiprogramming
- I/O
- Storage allocation
- Aiding the user
  - Overlay management
  - Dynamic loading
  - Large address spaces
Multiprogramming

- In general
  - Memory requirements of all the processes cannot be simultaneously met
  - Only a subset of the processes will be resident in (main) memory
  - These processes need to be “dynamically” moved out and possibly moved back in as dictated by the scheduler
- Consider a round-robin policy for example
  - The memory transfer required for the swap contribute to the context-switch time and hence limit its frequency; RR is only used for resident processes
I/O and Swapping

- Swapping a process A out that is currently in the middle of an I/O induced wait state can cause problems
  - The completion of I/O might store the values in the space meant for A in the space now occupied by a new process B
- The solutions are
  - Never swap out a process while in a wait state induced by I/O requests
  - All I/O interactions are via a special set of buffers that are controlled by the OS and are part of its space; not swapped out
Storage allocation algorithms

• Operations
  • p = malloc(n)
  • free (p)

• View memory as sequence of *blocks* and *voids*
  • blocks are in use
  • voids are available
    • neighboring voids are coalesced
Data-structures

- Free blocks marked; linked together in memory order
Freeing a block

- given
  - pointer to beginning
  - length
- look for next free block
  - using block lengths
  - (about 1/3 of the blocks will be free in equilibrium!)
- find previous free block
  - from next free block
- coallesce is possible
- add to free list
Allocation Policies

• First fit: Allocate space from the first void in the list that is big enough
  • Fast and good in terms of storage utilization
• Best fit: Allocate space from a void such that the remaining space is a minimum
  • Very good storage utilization
• Worst fit: Allocate a void such that the remaining space is a maximum
  • Requires peculiar memory loads to perform well in terms of storage utilization
• (Performance is measured via simulations)
Inefficiencies

• External Fragmentation
  • void space between blocks that does not serve any useful purpose
  • can be avoided by compaction (requires relocation)
    • needs run-time bindings
    • induces overhead in adjusting mapping

• Internal Fragmentation
  • it is not worth maintaining memory that leaves very small voids --- order of bytes --- between used regions
  • occurs more obviously when unit of allocation is large (e.g. disks)
Aiding the user

• Overlay management
  • for segmented programs too large for memory
  • alternative to user-defined overlays
• Dynamic loading
  • (relocateable) procedures are loaded “on demand”
• Large address spaces
  • allow space for objects to grow
Memory Mapping Schemes

• Objectives
  • memory protection
    • users from other users
    • system from users
  • efficient use of memory
    • time
    • space
  • programmer convenience
    • large virtual memory space
Partitioning 1

- Memory protected by a “key”
  - CPU register contains a key
  - each memory block specifies a key
  - at run time, CPU key matched with keys in memory blocks
    - on mismatch, generates an interrupt, handles to OS
- Two schemes
Partitioning 2

- **Fixed Partitions**
  - memory is divided into a number of fixed size partitions
  - each partition is allotted to a single process
  - no longer in use

- **Variable Partitions**
  - contiguous memory is allocated on loading
  - released on termination

- **Used in early IBM 360 models**
Problems with Partitions

• Programs must execute in the same place
  • allocation is inefficient
  • swapping is very constrained
  • changing memory requirements not provided for
  • no user assistance at all
Relocation Registers

• Dynamic Relocation
  • The process occupies locations 0, 1, 2, ... N in logical memory
  • Suppose that it is actually relocated into locations 1000, 1001, ... (1000 + N) in physical memory
  • The OS loads a value of 1000 into the relocation register and after that whenever a memory location is accessed:
    • It computes physical-address = logical-address + relocation register
    • Fetches the value from the resulting memory location
  • The stream of physical addresses are seen only by the memory unit
Protection & Relocation

• Base-Limit registers
  • mapping
    • logical address is compared with limit register
      • if higher, error trap to OS
    • logical address is added to contents of base register
      • result used as physical address
Pros & Cons of BL registers

• Pros
  • Allows
    • relocation
    • compaction
    • swapping

• Cons
  • Does not allow selective memory sharing
  • Compaction requires “move”
Memory Management (con’d)

• Paging
  • Motivation
    • Initially proposed as a method for
      • minimizing external fragmentation
      • allowing use of non-contiguous memory
    • Subsequently played a major role in virtual memory
      design
The Idea

• Simple approach involves
  • Viewing physical memory as a set of fixed size blocks called frames
  • A frame is the physical memory allocation unit
  • The (contiguous) virtual memory space of the user is broken into non-overlapping segments of the same size as the size of a frame
  • Each of these logical units is called a page (of virtual memory)
  • Each (virtual) page is allotted to a (physical) frame
Properties

• Allocation need not preserve the contiguity of logical memory
  • Pages 1, 2, 3, 4 can be allotted to frames 3, 7, 9 and 14
  • The frames in between the above such as 4, 5, 6 and so on can be assigned to a different process using a disjoint virtual memory space.

• Avoids external fragmentation?

• The First Step Towards Virtual Memory
  • A distinct separation between the meaning of a location in the user's virtual space and its actual physical storage is accomplished
Memory mapping

• The mapping is hidden from the user and is controlled via the OS
• The OS maintains a *map* of the available and allotted pages via a structure called a frame table
• This data structure indicates for each physical frame, its status, i.e.
  • Whether it is allocated or not
  • If allocated, to which page of which process
Address Translation

- Translating Logical to Physical Addresses
  - Performed on *each and every step*
  - Crucial to perform *extremely* efficiently if overall performance must not degrade

- The Scheme
  - Frame sizes are of size $2^n$
  - For each logical address of $a = m + n$ bits
    - The higher order $m$ bits indicate the page number $p_i$ and
    - The remaining $n$ bits indicate the offset $w_i$ into the page
Page Table Lookup

- The mapping is maintained by the *page table*
- The page number \( p_i \) is used to *index* into the \( p_i \)’th entry of the (process’) page table where the corresponding frame number \( f_i \) is stored
Hardware Support

• Needed for Page Table Maintainence
  • Fast on-chip registers that can be used to translate memory addresses \textit{on the fly} to virtual addresses
  • These registers are
    • loaded and saved with each process at context switch time
    • constructed from fast and \textit{expensive} logic to enable rapid translation
  • Reasonable if the size of the page table is \textit{small}, i.e. in the range of 256 entries or so
  • Not reasonable if we have over a \textit{million} entries which modern systems support and need
An Obvious Solution

- The page table is maintained in memory
- There is a single *page table base* register that
  - is loaded in with the process
  - points to the beginning of the page table
  - pi is now the offset into this table
- Problem
  - Requires two accesses to memory for each value
  - Impossibly slow
A Better Solution

- A special *hardware* memory unit called the *translation lookaside buffer* (TLB) is constructed using associative registers
- A portion of the page table is stored in the TLB
TLB operation

- TLB is searched for page table entry
  - uses (expensive) parallel hardware
  - is extremely fast
  - TLB sizes of the order of 2K are reasonable
- If a value is a hit in the TLB
  - there is little degradation in performance
- If not
  - a memory access is needed to load the value into the TLB
  - an existing value must be flushed if the TLB is full
Multi-level lookup

Logical address

PID

p1#
p2#
dis

s#

size base-addr

Primary Table

size base-addr

Secondary Table
OS involvement

- **OS must**
  - Allocate memory
  - Maintain page tables
  - Set up registers, e.g.
    - page-table base register
    - PID register

- **OS may**
  - Load TLB
Inverted Page Tables

• An Observation
  • Usually, only a portion of all the pages from the system's memory can be stored in the physical memory
  • So while the logical memory might be massive --- needing a very large page table --- only have a small subset of it contains information about useful mappings

• How can we take advantage of this fact?
  • Alternative TLB design
  • Alternative page table design
Inverted TLBs

- Contains mappings between frames and the pages they contain
- Mapping based on frame numbers rather than on page numbers
Inverted Page Tables

• Efficiency Considerations
  • The inverted page table is organized based on physical addresses via frame numbers
    • Searching for the frame number can be very slow
  • Use a hash table based on
    • The PID and logical page number as keys
  • Recently located entries of the inverted page table can be stored in a TLB like structure based on associative registers
Problems with Inverted PTs

- **Sharing**
  - Example, an editor whose code is the same for all users
  - This code can be maintained as one copy only
  - All the page tables of the processes using this code translate to the same address
  - Each process will have its own (disjoint) space where its data that uses the shared code is maintained

- Not possible to maintain with standard inverted page tables
Protection Issues

• Special bits are used by the hardware to indicate
  • Accessibility - whether a page is
    • readable, writable, executable
  • A valid/invalid bit to indicate whether a page is in the user's space
    • The hardware may support a page-table length register
      • Trailing invalid pages can be eliminated
      • Especially useful when processes are using a very small fraction of available physical space
Segmentation

• A Segment is
  • A *logical* piece of a program
  • Examples
    • an editor
    • a file
    • a data-structure

• Logical memory is broken into such segments

• Segments are of variable size
Accessing a Segment

• The logical address is regarded as two-dimensional
  • A segment pointer to an entry in the segment table
  • A displacement into the segment itself
• This is nothing to do with the two parts of an address in paging
Memory Allocation

- Each segment also has a well-defined size
  - called *segment length*
  - A segment is analogous to a single base-limit pair
  - The length attribute is helpful in protection since it prevents programs from accessing locations that are beyond their *segment space*

- The CPU
  - uses the segment address and length to access an address in memory

- The OS
  - sets the segment address and length
Segment Table Lookup

- The mapping is maintained by the segment table
- The segment number $s\#$ is used to index into the (process’) segment table where the corresponding segment size and base address are stored
Support Hardware

• Segment registers
  • In some hardware designs (e.g. Intel X86), segment registers are used to identify segment numbers; loading a segment register loads a (hidden) segment specification register from the segment table; construction of the logical address is then done entirely in the CPU.

• TLBs
  • In some designs, such as the MIPS 2000, the only hardware provided is a TLB; the OS is responsible for loading this, and doing appropriate translation
Traditional

• Multi-level lookup
  • Older designs divide the address into segment and displacement components
  • Similar concerns as in the case of paging with maintaining *segment tables* (as opposed to page tables)
  • Associative memory is used to speed access as before
  • If there is no associative store (such as a TLB) and if the segment table is *very large*, then a *segment-table base register* is used to point to the appropriate segment-table in memory
  • This register is reloaded with each context switch
Pros & Cons of Segments

• Pros
  • Protection in terms of ensuring that illegal address accesses are avoided, comes for free; the segment length check plays an important role here
  • Sharing segments across programs is straightforward by loading identical segment table base register values

• Cons
  • External fragmentation is potentially a big problem
  • Contrast this with paging where only internal fragmentation is possible
Segmentation & Paging

- Schemes that overlay a segmentation scheme on a paging environment are proposed and used
  - originally proposed for GE 645 / Multics
  - e.g. Intel X86 uses segment registers to generate 32-bit logical addresses, which are then translated to physical addresses by an optional multi-level paging scheme
- These contain the problem of external fragmentation
Examples 1

• Multics (c. 1965)
  • address: 8-bit major segment, 10-bit minor segment, 6-bit page, 10-bit offset
  • 3-level paging: segment table contains
    • base address and length of segment table
  • TLB: 16 entries; key = 24-bit (seg# & page#); value = frame#
Examples 2

- **OS/2 (on Intel 386+)**
  - address: 14-bit segment selector, 2-bit protection, 32-bit offset
  - 6 segment registers; 6 48-bit segment descriptor registers
  - descriptor specifies 24-bit base address, and 24-bit limit
  - units of base and limit may be 4 or 8 bits
  - base added to 32-bit offset
  - result is regarded as 10-bit page directory, 10-bit page#, 12-bit offset
  - page table is paged