Outline

• Announcements
  – Lab 4 due back on April 5th
  • Demos on April 5th and 6th
  – Questions?

• Memory Management
  – logical versus physical address space
  – swapping
  – allocation schemes
    • Partitioning
    • Paging
    • Segmentation

[Silberschatz/Galvin/Gagne, Chapter 9]

Background

• Programs operate on data and instructions stored in memory (von Neumann model)
  – 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

• Memory management: Providing efficient mechanisms for
  – binding: mapping program names into actual memory locations
  – mapping: utilizing the limited physical memory to bring logical memory objects (belonging to multiple processes) back and forth

  • Lectures 13 and 14: allocation of physical memory to processes
    – assume that the entire process fits in physical memory
  • Lectures 14 and 15: supporting virtual memory in allocated physical memory
    – process data and instructions need not all fit into physical memory

Binding Program Names: Logical to Physical

at compile-time
  – mapping of logical-to-physical addresses is done statically
  • changes in the physical address map require recompilation
  – rare for general programs, sometimes for OS components

at load-time
  – binding done by the loader when program is brought into memory for execution
  – change in the starting address only requires a reload

at run-time
  – binding is delayed until the program actually executes
  • special hardware support needed to accomplish this
  – more details in the rest of the lecture
Process Memory Requirements

- So far, we have assumed that the entire process and data need to fit into memory for the program to execute
  - Many techniques to reduce amount that needs to fit at any time

**Explicit management** by the programmer

- **dynamic loading**
  - load procedures “on demand”
- **overlays**
  - keep in memory only those instructions/data that are needed at any given time
  - rewrite portions of the address space with new instructions/data as required

**Implicit management** by the OS

- **Dynamic linking**
  - typically used with shared system libraries that are loaded on demand
  - calls resolved using an “import table”: initially point to the loading stub

Multiprogramming and Swapping

- Problem: Memory requirements of all the processes cannot be simultaneously met

**Solution: Swapping**

- “Dynamically” move a process out of memory into a backing store (and back in) as dictated by the medium-term scheduler
  - backing store is typically a fast disk
  - choice of which processes to swap out/in
    - can be influenced by short-term scheduling policy (e.g., priority-driven)
    - knowledge of process’ actual memory requirements
      - requires the process to reserve, commit, decommit, and release memory

**Swapping: Issues**

- **High context-switch times**
  - assume a user process of size 100 KB
  - backing store is a standard hard disk with transfer rate of 5 MB/s
  - actual transfer of 100 KB from and to memory takes
    2 × (100 KB / 5000 KB/s) = 2 × (1/50 second)
    = 2 × (20 ms) = 40 ms + disk time
  - helps to know exactly how much memory is being used
  - also, determines frequency

- **Swapping out a process that is currently in the middle of I/O**
  - I/O completion might store values in memory, now occupied by a new process
  - common solutions
    - 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
Memory Mapping Schemes

- **Goal**: Allocate physical memory to processes
  - translate process logical addresses into physical memory addresses

- **Objectives**
  - memory protection
    - users from other users, system from users
  - efficient use of memory
  - programmer convenience
    - large virtual memory space

- **Three schemes**
  - Partitioning
  - Paging
  - Segmentation (if time permits)

---

Memory Mapping (1): Partitioning

- **Idea**: Divide memory into partitions

- **Protection**
  - each partition protected with a “key”
  - at run time, process key (stored in a register) matched with partition key
    - on mismatch, generates a trap

- **Allocation**
  - fixed partitions
    - memory is divided into a number of fixed size partitions
    - each partition is allotted to a single process
    - used in the early IBM 360 models
    - no longer in use
  - variable partitions
    - contiguous memory is allocated on loading
    - released on termination
    - this is what you are using in Nachos Lab 4

---

Memory Mapping: Partitioning (cont’d)

- Partitioning for statically-bound programs
  - programs must execute in the same place
  - allocation is inefficient, and swapping is very constrained
  - no provision for changing memory requirements

- Partitioning for dynamically-bound programs
  - relocation registers
    - a CPU register keeps track of the starting address where the process is loaded
    - whenever a memory location is accessed:
      - the system 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 MMU
  - how to prevent a process from accessing addresses outside its partition?
Memory Allocation and Scheduling

Partitioning Policies

- Memory is viewed as sequence of blocks and voids (holes)
  - blocks are in use
  - voids are available: neighboring voids are coalesced to satisfy request

- Question: Given a request for process memory and list of current voids, how to satisfy the request
  - 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 to leave minimum remaining space
    • 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

Partitioning Policies (contd.)

- Criterion for evaluating a policy: Fragmentation

- External fragmentation
  - void space between blocks that does not serve any useful purpose
  - statistical analysis of first-fit: ~0.5N blocks will be lost due to fragmentation
  - can be avoided by compaction
    • Swap out a partition
    • Swap it back into another part of memory: requires relocation

- Internal fragmentation
  - it is not worth maintaining memory that leaves very small voids (e.g., a few bytes) between used regions
    • occurs more obviously when unit of allocation is large (e.g., disks)
  - Happens when memory request is smaller than the smallest partition size

Memory Compaction: Reducing Fragmentation

- Moving partitions around can group the voids together
  - increase likelihood of their being used to satisfy a future request

- Many ways of doing this:
Memory Mapping (2): Paging

- Motivation: Partitioning suffers from large external fragmentation

Paging
- view physical memory as composed of several fixed-size frames
  - a “frame” is a physical memory allocation unit
- view logical memory as consisting of blocks of the same size: pages
- allocation problem
  - put “pages” into “frames”
    - a page table maintains the mapping
  - allocation need not preserve the contiguity of logical memory
    - e.g., pages 1, 2, 3, 4 can be allocated to frames 3, 7, 9, 14
  - how does this avoid external fragmentation?
- paging played a major role in virtual memory design
  - separation between the meaning of a location in the user’s virtual space and its actual physical storage

Paging (example)

- Mapping of pages to frames
  - the mapping is hidden from the user and is controlled via the OS
- Allocation of frames to processes (Nachos Lab 4)
  - the OS maintains a map of the available and allotted frames via a structure called a frame table
    - whether a frame is allocated or not
    - if allocated, to which page of which process
- Address translation
  - performed on every memory access
  - must be performed extremely efficiently so as to not degrade performance
  - typical scheme
    - frames (and pages) are of size $2^k$
    - 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

- Mapping between pages and frames is maintained by a 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 $f_j$ is stored

- All of this requires hardware support
  - since performed on every memory access
Page Table Structure

- Page table typically stored in memory
  - a single page table base register that
    • points to the beginning of the page table
    • $p_i$ is now the offset into this table
    - problem
      • requires two accesses to memory for each value
      • even with caches, can become very slow

- Solution: Translation Lookaside Buffer (TLB)
  - a portion of the page table is cached in the TLB
    • little performance degradation if a value is a hit in the TLB
    • 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
  - E.g.: Average memory access time for a system with 90% hit rate in TLB
    $= 0.9*(\text{Access}_{\text{TLB}} + \text{Access}_{\text{mem}}) + 0.1*(\text{Access}_{\text{mem}} + \text{Access}_{\text{mem}})$
    $= 1.1*(\text{Access}_{\text{mem}})$

Multi-level Page Tables

- Rationale: Modern systems support a very large logical address space
  - page tables themselves become very large
    • e.g., for a system with 32-bit logical addresses and 4K pages
      - we need $2^{32}$ page table entries (4 bytes per PTE implies 4 MB of space)
  - Solution: page the page table itself
    - cost: additional memory accesses (but caching helps)

Page Tables and Sharing

- Page tables permit different virtual addresses (frames of different processes) to map to the same physical address
  - convenient sharing of common code (dynamically-linked system libraries)
  - shared data segments for IPC

Inverted Page Tables

- Observation
  - usually, only a portion of all the pages from the system's memory can be stored in the physical memory
  - so while the required page table for all of logical memory might be massive, only a small subset of it contains useful mappings

- We can take advantage of this fact in both TLB and page table design
Inverted Page Tables (cont’d)

• 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

• Main disadvantage of inverted page tables: sharing
  – each process that shares an object will have its own (disjoint) space where the shared object is mapped
  – not possible to maintain with standard inverted page tables
    • since space for only one <PID, page number> tuple

Protection Issues with Paging

• Partition protection scheme
  – Check that address lies between base and base+limit
  – Cannot be used on page-based systems: WHY?

• Physical memory can only be accessed through page table mappings
  – all addresses are interpreted by the MMU
  – OS intervention required to manipulate page tables and TLBs

• Special bits in the page table entry enforce per-frame protection
  – an accessibility bit
    • whether a page is invalid, readable, writable, executable
  – a valid/invalid bit to indicate whether a page is in the user's (logical) space

• Sometimes, the hardware may support a page-table length register
  – specifies size of the process page table
    • trailing invalid pages can be eliminated
    • useful when processes are using a small fraction of available address space

Memory Mapping (3): Segmentation

• A segment is a logical piece of the program
  – e.g., the code for the program functions, its data structures, symbol tables

• Segmentation views logical memory as broken into such segments
  – segments are of variable size (unlike pages)

• 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

• Allocating a segment
  – a segment is a partition with a single base-limit pair
    • the limit attribute stores the segment length
    • prevents programs from accessing locations outside the segment space
    • differs from partitioning in that there can be multiple segments/process

Memory Mapping: Segment Table Lookup

• Mapping logical addresses to physical addresses
  – 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
Memory Mapping: Segmentation Hardware

- Segment registers
  - some designs (e.g., Intel x86) provide registers to identify segments
    - loading a segment register loads a (hidden) segment specification register from the segment table
    - construction of the logical address is done explicitly
- TLBs
  - some designs, such as the MIPS 2000, only provide a TLB
    - the OS is responsible for loading this, and doing appropriate translation
- Traditional approach: Store the segment table in memory
  - segment table base register (STBR), segment table length register (STLR)
    - saved and restored on each context switch
  - translation of address (s, d)
    - check that s is valid: s < STLR
    - Look up base address, limit: segment table entry at address (STBR + s)
    - check that offset d is valid: d < length
    - compute physical address

Segmentation: Pros and Cons

- 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
    - Caveat: How do instructions refer to addresses within segments?
      - Relative addressing works well with sharing
      - Absolute addressing does not: requires same segment number
- Cons
  - external fragmentation is potentially a big problem
  - contrast this with paging where only internal fragmentation is possible

Memory Mapping: Segmentation and Paging

- Overlay a segmentation scheme on a paging environment
  - several examples
    - originally proposed for GE 645 / Multics
    - Intel x86 uses segment registers to generate 32-bit logical addresses, which are translated to physical addresses by an optional multi-level paging scheme
  - alleviates the problem of external fragmentation

Memory Mapping: Examples

Multics (c. 1965)

- 34-bit logical address
  - 18-bit segment number, 16-bit offset
  - [8-bit major segment, 10-bit minor segment], [6-bit page, 10-bit offset]
  - Both the segment table and segment itself are paged!
- Segmentation structure
  - Segment table is pagged
  - major segment number indexes page table for segment table
  - minor segment number is offset within the page of the segment table
    - this gives the page table of the desired segment and the segment length
- Paging structure
  - one-level page table, 1KB pages
- TLB
  - 16 entries; key = 24-bit (seg# & page#); value = frame#
Memory Mapping: Examples (cont’d)

- OS/2 (on Intel 386+): Segmentation with paging

OS/2 (on i386+) Memory Mapping (cont’d)

- Very flexible addressing scheme
  - pure paging
    - All segment registers set up with the same selector
      - Descriptor for this selector has base = 0, limit = MAXVAL
      - Offset becomes the address
    - pure segmentation
      - How can this be done?
      - options in between