Background

- Programs operate on values (data and instructions) stored in memory (in the von Neumann model)
  - 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

- Memory management: Providing efficient mechanisms for
  - binding: mapping program names into actual memory locations (possibly via an intermediate level of logical memory addresses)
  - mapping: utilizing the limited physical memory to bring logical memory objects (belonging to multiple processes) back and forth
    - this lecture: allocation of physical memory to processes
    - assume that the entire process fits in physical memory
    - next lecture: supporting process virtual memory in allocated physical memory

Silberschatz/Galvin: Chapter 8

Binding Program Names

Step 1: The compiler translates variable names into logical addresses

Step 2: These addresses are bound to actual locations in memory
  - at compile-time
    - mapping of logical-to-physical addresses is done statically
    - changes in the physical address map require recompilation
    - rare for general programs, but sometimes done for OS components
  - at load-time
    - the compiler generates relocatable code
    - binding done by the loader when program is brought into memory for execution
    - a change in the starting address of the program only requires a reload
  - at run-time
    - binding is delayed until the program actually executes
    - because parts of the program and its storage move around all the time
    - special hardware support for "page management" needed to accomplish this
    - more details in the rest of the lecture
Reducing Process Memory Requirements

- So far, we have assumed that the entire process and data need to fit into memory for the program to execute

Two broad categories of techniques
- Explicit management by the programmer
  - 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
  - dynamic loading
    - load (relocatable) procedures “on demand”
- 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
    - large virtual address spaces
      - more about this in the next lecture

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
  - issue: 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

General Problem: Dynamic Storage Allocation

- Operations at the user level
  \[ p = \text{malloc}(n) \]
  \[ \text{free} (p) \]
- Algorithms view memory as sequence of blocks and voids
  - blocks are in use
  - voids are available: neighboring voids are coalesced to satisfy request
- 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 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

Inefficiencies in Storage Allocation

- 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
  - e.g., statistical analysis of first-fit shows that ~0.5N blocks will be lost due to fragmentation
- 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)
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

Memory Mapping: 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 an interrupt

- Allocation
  - 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 the early IBM 360 models

Memory Mapping: Partitioning (contd.)

- 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 \( \text{physical-address} = \text{logical-address} + \text{relocation register} \)
      - 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 Mapping: Partitioning (contd.)

- Protection and relocation for dynamically-bound programs
  - base-limit registers
    - a pair of registers that store the starting address and maximum offset
    - 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: allows
    - relocation
    - compaction of holes
    - swapping: a process can be swapped back into another location
  - cons
    - does not allow selective memory sharing
    - compaction requires “move”
Memory Mapping: 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”
    - 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

Memory Mapping: Paging (contd.)

- Mapping of pages to frames
  - the mapping is hidden from the user and is controlled via the OS

- Allocation of frames to processes
  - 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

Memory Mapping: 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 $f_i$ is stored

  ![Logical address](#)  [PID]  [pi]  [wi]  [wi]  (Page Table)  [fi]  (Physical address)

- How can we support efficient page table lookups?

Memory Mapping: Page Table Maintenance

- Page table is stored in fast on-chip registers
  - loaded and saved with each process at context switch time
  - constructed from fast and expensive logic to enable rapid translation
    - reasonable if the size of the page table is small (e.g., ~256 entries)
      - modern systems require millions of page table entries

- More reasonable: Store page table in memory
  - 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
    - even with caches, can become very slow
Memory Mapping: Translation Lookaside Buffers

- Improve lookup costs for memory-allocated page tables
  - a special hardware memory unit called the translation lookaside buffer (TLB) is constructed using associative registers
  - sizes of the order of 2K are reasonable
  - 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

Memory Mapping: 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 $2^{20}$ 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)

Memory Mapping: 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 have a small subset of it contains useful mappings

- We can take advantage of this fact in both TLB and page table design

Memory Mapping: Inverted Page Tables (contd.)

- 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
Memory Mapping: Protection Issues with Paging

- All addresses are interpreted by the MMU
- OS intervention required to manipulate page tables and TLBs
- Special bits in the page table entry
  - the MMU interprets these bits
  - an *accessibility* bit
    - whether a page is 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
  - especially useful when processes are using a very small fraction of available physical space

Memory Mapping: 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 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
- 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*

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
    - reloaded on each context switch
Memory Mapping: 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

- **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
    - 8-bit major segment, 10-bit minor segment, 6-bit page, 10-bit offset
  - segmentation structure
    - segment table base register stores segment table base and length
    - major segment number indexes page table for segment base
    - 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 (contd.)

- OS/2 (on Intel 386+)
  - segmentation with paging: upto 16K segments (max 4 GB), 4KB pages
  - logical address space of process is divided into two partitions
    - first partition (upto 8K segments) are private
      - info kept in the local descriptor table (LDT)
    - second partition (upto 8K segments) are shared
      - info kept in the global descriptor table (GDT)
  - logical address: 14-bit segment selector, 2-bit protection, 32-bit offset
  - machine has
    - 6 segment registers; 6 48-bit segment descriptor registers (essentially a TLB)
    - descriptor specifies 24-bit base address, and 24-bit limit (units of 4 or 8 bits)
    - base added to 32-bit offset: result is a *linear address*
      - 10-bit page directory, 10-bit page#, 12-bit offset
  - 2-level page table
Next Lecture

- Virtual Memory
  - background: process working sets
  - demand paging
  - page replacement
  - frame allocation
  - thrashing

Reading

- Silberschatz/Galvin: Chapter 9