Main Points

• Address Translation Concept
  • Converting virtual addresses to physical addresses

• Flexible Address Translation
  • Base and bound
  • Segmentation
  • Paging
  • Multilevel translation
Address Translation Concept

- Translator converts (virtual) addresses generated by programs into physical memory addresses.
Address Translation Goals

- **Memory protection**
  - Limit access of processes to certain regions of memory
  - E.g., prevent a process from accessing others’ memory

- **Memory sharing**
  - Allow processes to shared selected regions of memory
  - E.g., enable shared libraries, interprocess communication

- **Sparse addresses**
  - Allow memory regions to dynamically change in size
  - E.g., enable dynamically sized heap and stack
Address Translation Goals (cont.)

- Flexibility
  - Allow OS to place process anywhere in physical memory
- Efficiency
  - Translate addresses faster than executing instructions
- Compact translation tables
  - Minimize space overhead of translation
- Portability
  - Design a portable kernel (hardware-independent data structures) to allow different hardware architectures to implement translations differently
**Virtually Addressed Base and Bounds**

- Virtual address is added to base to generate physical address.
- Bound is checked against the virtual address to prevent a process from reading/writing outside of its memory region.
- Only OS changes base and bound; otherwise, no protection.
Virtually Addressed Base and Bounds

• **Pros**
  • Provides protection, so it is safe
  • Has low overhead and is simple
    • Two registers, one adder, and one comparator
  • Can move processes transparently
    • E.g., if program needs to grow beyond its bounds, stop the program, copy bits, change base and bound registers, restart!

• **Cons**
  • Does not prevent program from overwriting its own code
  • Does not share code/data with other processes
  • Has high overhead to allow growing stack and heap
Process Regions or Segments

• Process has logical regions or segments
  • E.g., code, data, heap, stack, dynamic library, memory mapped file, ...
• Physical memory for each segment is stored contiguously
• Different segments can be stored at different locations in physical memory
• Segmented memory has gaps in both virtual and physical address space
  • Accessing these gaps generates exception (segmentation fault in UNIX)
Virtual address is divided to a segment number and a segment offset.

Segment number indexes into the **segment table** (in hardware).

- Each entry has base, bound, and access permission of a segment in physical memory.

Processes can have **restricted rights** to certain segments.

- E.g., to prevent writes to the code segment.
Memory Sharing

- Suppose two processes sharing a code segment
- Each process uses the same virtual addresses
- These virtual addresses are mapped to the same region of physical memory for code and different regions of physical memory for data
UNIX fork and Copy-on-Write

- UNIX fork makes a complete copy of a process
- Segments allow a more efficient implementation
  - Copy segment table into child
  - Mark parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (ex: stack, heap)
    - Trap into kernel
    - Make a copy of the segment and resume
Zero-on-Reference

- Size of heap could vary from a few KB to several GB
- OS zeroes a few kilobytes of allocated heap memory
  - Avoid accidental information leakage
- OS sets the bound in segment table to limit the program to just the zeroed part
- If program expands its heap, it will take an exception
  - Allocate some more memory
  - Zero the allocated memory
  - Modify segment table
  - Resume process
Upsides of Memory Segmentation

• Can share code/data segments between processes
• Can protect code segment from being overwritten
• Can transparently grow stack/heap as needed
• Can detect if need to copy-on-write
Downside: Complex Memory Management

- Large # of variable sized and dynamically growing segments
- Variable sized free spaces between allocated segments
  - How much memory should we set aside for a program’s heap?
  - Should we put new process in the largest free space or smallest?
- External fragmentation
  - There might be enough free space available but not contiguous!
  - May need to rearrange memory to make room for new segment or growing segment
Paged Translation

- Split physical memory into fixed-size page frames
- Assign page-size aligned block of virtual addresses to each frame
- Allocate free space easily by representing memory as a bit map
  - One bit per page, e.g., 0011010
Paged Translation (Implementation)

- Virtual address is divided to a **page number** and an **offset** within the page.
- Page number indexes into the page table to yield a physical page.
- Physical address is physical page concatenated with the page offset.
- Page table is more compact compared to segment table.
  - Stores only upper bits of page address.
- Page table is stored in physical memory.
  - Hardware registers store pointer to page table start and length of page table.
Paging Questions

- With paging, what is saved/restored on a process context switch?
  - Pointer to page table, size of page table
  - Page table itself is in main memory
- What if page size is very small?
  - Huge space is taken up by page table entries
- What if page size is very large?
  - Internal fragmentation: if we don’t need all of the space inside a fixed size chunk
Paging and Memory Optimizations

- Memory sharing between processes
  - Set entries in both page tables point to same pages
  - Need core-map: track processes pointing to each page
- UNIX fork with copy-on-write
  - Copy page table of parent into child process
  - Mark all pages (in new and old page tables) as read-only
  - Trap into kernel on write (in child or parent)
  - Copy the page and mark page table entry as writeable
- Zero-on-reference
  - Set page table entry at the top of the stack to invalid
  - Trap into the kernel when the page is referenced for first time
  - Extend the stack only as needed
Fill-on-Demand

- OS can start running a program before it is completely loaded in memory
  - Set all page table entries to invalid
  - Trap into kernel when a page is referenced for first time
  - Kernel brings page in from disk
  - Resume execution
  - Remaining pages can be transferred in the background while program is running
Downside: Sparse Address Spaces

- Page table could be huge
  - 32-bits virtual address, 4KB pages: $2^{20}$ page table entries
  - 64-bits virtual address, 16KB pages: $2^{50}$ page table entries

- Virtual address space is sparse
  - Gaps between heap, per-thread stacks, memory-mapped files, dynamically linked libraries

- Entire page table has to be stored
  - Most pages entries are invalid
Multi-level Translation

- To store a sparse key space, tree is more efficient than array.
- Only top-level page table must be filled in; the lower levels of tree are allocated only if used.
- Many systems use tree-based address translation:
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- Leaves of the tree represent fixed-size pages:
  - Efficient physical memory management (compared to segments)
  - Compact lookup table for sparse addresses (compared to paging)
  - Efficient disk transfers (making page size a multiple of disk sectors)
  - Efficient core-map implementation (from physical to virtual)
Paged Segmentation

- Memory is segmented
- Segment table entry:
  - Pointer to page table
  - # of pages in segment
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at page or segment-level
Multilevel Paging

- Virtual address is divided to create multiple paging levels
- OS saves space by omitting sub-tree if no valid addresses
Multilevel Paged Segmentation (x86)

- Global Descriptor Table (segment table)
  - Pointer to page table for each segment
  - Segment length
  - Segment access permissions

- Context switch
  - Change global descriptor table register (GDTR, pointer to global descriptor table)

- Multilevel page table
  - 32-bit: two level page table (per segment)
  - 64-bit: four level page table (per segment)
• 4KB pages; each level of page table fits in one page
**x86 64-bit Virtual Address**

- Fourth level table maps 2MB, and third level table maps 1GB of data
- If 2 MB covered by a fourth level table is contiguous in physical memory, then entry one third level can directly point to this region instead of pointing to a forth level page table
Multilevel Translation

• **Pros**
  • Allocate/fill only page table entries that are in use
  • Simple memory allocation
  • Share at segment or page level

• **Cons**
  • Space overhead (if all virtual address space is used)
  • Two (or more) lookups per memory reference
Portability

• Many OSs keep their own translation data structures
  • List of memory objects (segments)
    • E.g., when process starts, kernel can check if code is in memory
  • Mapping from virtual page to physical page frame
    • E.g., when invalid page is accessed, kernel can check if it is truly invalid, or simply not loaded yet (in the case of fill-on-reference)
  • Mapping from physical page frame to set of virtual pages
    • E.g., when page’s status changes, kernel can also update every page table entry that refers to that physical page
Kernel Page Transition

• Example 1: inverted page table (Apple OS X)
  • Hash from virtual page to physical page
  • Space proportional to number of physical pages

• Example 2: virtual/shadow page table (Linux)
  • Kernel tables mirror x86 structure, even in ARM!
Acknowledgment

• This lecture is a slightly modified version of the one prepared by Tom Anderson