Paging

- Allows physical address space of a process to be non-contiguous
- Page: Contiguous chunk of memory addresses
  - Process virtual memory is divided up into equal-size pages
- Frame: Physical memory is also divided up into same-sized chunks
- Page Table: Mapping of virtual pages to physical frames
  - Each process has its own page table

Memory Allocation

- Paging
- Segmentation

Paging : Virtual Memory Mapping
**Paging: Benefits**
- Process does not have to be placed contiguously
  - OS can assign any available frame
- No external fragmentation
  - May have internal fragmentation
- Can share memory across processes
  - Map to same physical page
  - E.g.: shared libraries

**Paging: Address Translation**
- Virtual address: Consists of two parts
  - Page number (p): Used as index into page table
  - Page offset (d): Offset within the page
- Physical address has:
  - Frame number (f): Page table entry indexed by p
  - Frame offset (d): Same as page offset
- m-bit address:
  - Lower n-bit: Page offset => Page size = $2^n$
  - Upper (m-n)-bit: Page/frame number: Determines page table size

**Hardware Support for Paging**
- How is page table maintained?
- How to speed up page translation?
- How to protect address spaces?

**Page Table Implementation**
- Option 1: Dedicated registers hold page table entries
  - Context switch: Load page table entries for new process, save those of old process in PCB
- Benefit?
- Problem?
**Page Table Implementation**
- Option 2: Hold page table in memory
- Page-Table Base Register (PTBL): Points to base of page table
  - Context switch: Load PTBL for new process, save PTBL for old process in PCB
- Benefit?
- Problem?

**Translation-Lookaside Buffer (TLB)**
- Fast cache for page table entries
- Consists of a set of recent address translations
- TLB hit: Get the address translation directly
- TLB miss:
  - Fetch page table entry, add entry to TLB
  - Replace some entry if no space
  - Some entries can be wired down, e.g., kernel addresses

**TLB: Context switch**
- On context switch:
  - TLB must be flushed
  - Do we need to load new process’s entries?
- Some TLBs have support for:
  - Address-space identifiers (ASIDs). E.g.: pid
  - Allows translation entries for multiple processes to remain together in TLB
  - Benefit?

**Memory Protection**
- Page table entries contain access protection bits:
  - Read-only, RW, execute
  - Corresponds to the access restriction on page
- Some processes may not use all their address space
  - Valid/invalid bit: Whether page is in virtual address space of process
- Page Table Length Register (PTLR): Specifies size of page table
**Structuring Page Tables**

- How many page table entries would be there for:
  - A 32-bit system with 4KB page size?
  - A 64-bit system with 4KB page size?
- Problem: For many modern systems, page tables are too large
  - Cannot allocate the whole page table contiguously
  - Space is wasted if page table is sparse (only a few entries are used)

**Page Table Structures**

- Hierarchical Page Tables
- Hashed Page Tables
- Inverted Page Tables

**Hierarchical Paging**

- Two-level paging:
  - Page table itself is paged
  - Two levels of page tables: inner and outer
- Inner page table: Contains page translation entries
  - Page table divided into multiple inner page tables
  - Each inner page table fits into one page
- Outer page table: Contains entries for inner page table locations
  - Also called Forward-mapped page table

**Hierarchical Paging: Translation**

- Page number (p) further divided into 2 parts:
  - p1 (index into outer page table)
  - p2 (offset into page corresponding to inner page table)
- Address Translation:
  - Use p1 to find page containing inner page table
  - Use p2 to find entry for page in inner page table
  - Translate to frame number and apply offset d to get physical address
  - Each translation requires two memory accesses
Hierarchical Paging: Problems

- How big is the outer page table for:
  - A 32-bit system with 4KB page size? Assume each page table entry is 4 bytes
  - A 64-bit system with 4KB page size? Assume each page table entry is 4 bytes
- Suppose we increased to 3-level page tables?
- 64-bit systems:
  - Would need several levels of hierarchy for efficient page table allocation
  - Problem?

Hashed Page Tables

- Page table is now a hash table
  - Hash of virtual page number is index into page table
  - Each entry is a linked list of page translation entries
  - Linked list entry: <page no., frame no., next ptr>
- Address Translation:
  - Hash on page number
  - Traverse linked list to find the right entry for given page number
  - Convert to frame number to get physical address
  - Problem: Each entry is large – wastes space

Clustered Page Tables

- Page block: Collection of multiple (e.g., 16) consecutive pages
  - Virtual block number = Base of virtual page block
  - Block offset = Page number inside the block (e.g., 0-15)
- Page table is a hashed page table
- Each linked list entry has:
  - mapping information for a block of pages
  - only single tag (virtual block number) and next ptr
- Index into page table = Hash of virtual block number
- Index into linked list entry = Block offset

Address Translation:

- Hash on virtual block number, find the right entry for given page block in linked list
- Index based on block offset to find entry for page
- Convert to frame number to get physical address

Benefits:

- Good for sparse address spaces: don’t need to keep entries for all virtual pages
- Less space usage than hashed page tables
Inverted Page Tables

- Inverted page table: Has one entry for each physical frame
  - Entry: \(<\text{pid}, \text{virtual page no.}>\)
- Address Translation:
  - Traverse the page table until find entry for virtual page
  - Index \(i\) of the entry = frame number
  - Use frame number to get physical address

Inverted Page Tables: Pros & Cons

- Benefits:
  - Much fewer entries than regular page table
  - Low space requirements
- Problems:
  - Lookup is slow: Can be speeded up using hashing
  - Many virtual page entries may be missing (for pages that are not currently in memory)
  - Difficult to map shared pages

Segmentation: Motivation

- User's view of memory
  - Not a linear, contiguous array of addresses
  - A collection of memory "segments": main program, libraries, stack, data structures
- Logical Address = element in a segment
  - 10\(^{th}\) instruction in main program
  - 49\(^{th}\) element of array

Segmentation

- Supports user's view of memory
  - Logical address = \(<\text{segment number (s)}, \text{offset (d)}>\)
- Compiler creates multiple segments
  - E.g.: code, stack, heap, global vars, libs, etc.
- Segment Table:
  - Each entry corresponds to a segment number
  - Has a Segment Base (SB) and Segment Limit (SL)
- Address Translation:
  - Index into segment table using \(s\) to get (SB, SL)
  - Check \(0 < d < SL\)
  - Physical Address = SB+d
Combining Paging and Segmentation

- Logical address = <segment number, offset>
- Two-step address translation:
  - Logical-to-virtual: Translate using segment table to get a linear virtual address
  - Virtual-to-physical: Translate linear virtual address using paging to get physical address
- Example: Intel x86/Linux