Paging

Chapters: 9.3, 9.4, 9.6 (9.7 are optional, 9.4.4 excluded)

# Contiguous Memory Management

* Previously looked at contiguous memory allocation
* Allocation granularity entirely within logical address space
* We request physical space for the entire program at once
* Contiguous in logical and physical memory
* Issues:
  + Fragmentation: sum of free spaces is significant but space spread among disk
  + Long compaction times
  + Long swap times

# Non Contiguous Memory Allocation

* Techniques that allow us to not have to search entire logical address space
* Involves splitting logical address space into chunks
* Request physical space for each program chunk at the time
  + Contiguous in logical memory, non contiguous in physical

# Segmentation

* Involves partitioning address space into variable size chunk/units
* Logical units can be :
  + Stack heap, data, code, subroutines
  + With associated segment number
  + A logical address is then composed of <segment number, offset>
* This technique facilitates sharing and reuse
  + Diagram

    Description automatically generatedsegment is a natural unit of sharing
    - e.g. shared library
* Every address above is built with the segment ID + offset
* Code can be segmented and refer to other segments
* Every time we have a logical address, it goes to the MMU and is translated into a physical address
* We however can place every segment wherever we want in memory, allowing us to make better use of the physical space
* The compiler, supported by the CPU keeps track of all the offsets
* To keep track of all the segments, a **segment table** is used.
* Each segment has a base/limit pair
* Segments named by segment number used as **index** into the table
  + Logical address: <segment # , offset>
  + Diagram

    Description automatically generatedphysical address = logical address + segment base address

## Pros and Cons

* Allows non-contiguous physical addresses
  + Allocated “chunks” are smaller than entire program address space
  + Reduces fragmentation by exploiting varying sized holes
* Enables sharing
  + Same segment shared across multiple processes
* Process’s view and physical memory **very different**
* By implementation, process can only access its own memory
* Rarely used today

# Paging

* How modern OSs manage memory
* Logical address space divided into **fixed size** blocks of the same size called **pages**
* Diagram

  Description automatically generatedPhysical address space divided into **fixed size** blocks of the same size called **frames**.
* This allows chunks of a program to slot anywhere into memory.
* Solves fragmentation problem and allows virtual address space to be larger than physical address space.

## Address Translation Scheme

* We need to be able to map each page to a frame or cluster of frames.
* Chunks are at a predefined fixed (logical and physical) address.
* Table

  Description automatically generatedFor translation, the logical address is divided into:
  + Page number (p): Indexes into a page table which contains frame’s base address
  + Page Offset (d): summed to frame base address to become **physical** memory address.
  + Think of address like sequence of bits:
    - Page offset is from less significant to most significant bit
    - bit is page number
* Logical address space size: bytes
* Page and frame size: bytes
* Page size and frame size are both defined by the hardware. It is **always** a **power of 2**.

### Diagram Description automatically generatedPage Tables

* **Page table**: array of page table entries. p indexes into table: stored in RAM
* **Page table entry** (**PTE**): frame base address and bits/flags
* To translate a logical address into a physical address:

1. Extract page number and use it to index the page table.
2. Extract the corresponding frame number from page table.
3. Replace the page number in the logical address with frame number .

* Logical address 0 (page 0, offset 0) maps to physical address 20
* Logical address 3 (page 0, offset 3) maps to physical address 23
* Logical address 4 maps to 24.
* A screenshot of a computer

  Description automatically generated with low confidenceIn logical example table, LHS corresponds o

## Paging Advantages

* No external fragmentation
* Internal fragmentation depends on page size
  + Page size: 2048 byes. process size 72,766
  + Remainder of 72,766/2048 gives us total fragmentation
  + 72,766/2048 = 35 (full) pages + 1086 bytes
  + Total fragmentation:2048 – 1086 = 962 bytes
* Worst case scenario: 1 frame with only 1 byte
* Average fragmentation: 0.5 frame size
* Are small frames desirable?
  + Advantage: Smaller frames mean less internal fragmentation
  + Disadvantage: Small frame size however results in overhead from long translation table

## Free Frames

* All free frames kept in a list
* If a process requires frames, there must be free frames available in memory.
* Every time a free page is allocated, the frame number is put in the page table for this particular process.
* Note that the user will interpret this as a contiguous space. This is handled by address translation hardware, and controlled by the OS.
* **Frame Table**: entry per physical frame, showing if it is free or allocated to a particular process.

Diagram

Description automatically generated

## Many Processes

**Diagram

Description automatically generated**

* OS maintains a copy of page table for each process, just like it maintains a copy of the instruction counter and register contents.
* This copy is used to translate the logical addresses to physical addresses whenever the OS must map this manually.
* Also used by CPU dispatcher to define the hardware page table when aa process is to be allocated to the CPU.
* Hence time of context switches increase.

# Page Tables

* Page tables are kept in main memory.

If page table was set up as a set of physical addresses, this would be fast but increase context switch time, as all the registers would need changed every time.

* Instead we use a pointer in the PCB to page table with other register values:
  + **Page-table base register** (**PTBR**): Points to the page table, and is **part of the CPU**.
  + **Page-table length register** (**PTLR**): Indicates size of page table, **part of CPU.**
  + Substantially reduced Context switch time.
* Every data/code requires two memory accesses as a result:
  + To fetch **page table entry** from memory (translation).
  + To fetch actual **memory content** (data/code).

## TLB – Translation Look-Aside Buffers

* The solution to this is the **TLB**: **Translation look-aside buffers**
  + This is a fast lookup hardware cache
  + Entries take the form of <page #, frame #>.
  + Typically small (64 or 1024 entries)
* TLB hits use it, and TLB misses fetch it from memory
* Maintains translations for subsequent memory access:
  + **replacement policies** needed
  + Some entries can be **wired down** for permanent fast access
* Some TLBs store **address-space identifiers** (**ASIDs**): uniquely identifies each process and provides **address space protection** for that process.
  + When TLB attempts to resolve virtual page number, it ensures the ASID for the currently running process matches the ASID associated with the virtual page.
  + If they do not match, the attempt is treated as a TLB miss.
  + Also allows TLB to contain entries for several different processes simultaneously as it uses process identification.
* Diagram

  Description automatically generatedWithout ASIDs, TLB cache must be flushed every time a new page table is selected (e.g. every context switch).

### Effective Access Time

* We can do simple math to understand how much time our application takes to access memory:
* **Memory Access Time**: time CPU must wait to access main memory directly
* **Hit Ratio** : Percentage of times page number is found in TLB
* **Miss Ratio**
* If memory access time is t. Then a TLB miss will take 2t as we must access memory to find the page table, and then again to find the desired byte in memory.
* We want to find the **effective memory-access time**:
* Consider , 100ns for memory access
* Consider , 100ns for memory access
* As you can see the TLB can have a significant impact on access time. 99% TLB miss produces only 1 percent slowdown in access time.

## Memory Protection

* Page tables have many PTEs (page table entries)
* Memory protection accomplished by **protection bits** in each PTE:
  + **Read**, **read-write**, **execute only** bits
  + **Valid-Invalid** bit: Does a **translation exist**?
    - **Valid**: indicates associated page is in process’s logical address space, thus legal
    - **Invalid**: indicates associated page is not in process’s logical address space
* Violations to memory protection are trapped to the OS kernel.
* Ensures that processes are only accessing pages within their logical address spaces
* **Page-table length register** also ensures that reference address is in valid range for a process.

## Sharing Pages

* We can use paging to **share code or data**
  + One copy of read-only (**re-entrant**) code can be shared among processes
  + Read-Write **data pages** shared among processes for **communication**
* **Private code and data** (weird point from lectures)
  + Each process keeps a separate copy of the code and/or data
  + Pages that for private/shared code can appear **anywhere**  in logical address space.

### Chart, diagram Description automatically generatedExample of Shared pages

## Structure of Page Tables

* Page tables can end up being massive in size:
  + Consider a 32-bit logical address space
  + Page size of 4 kB ()
  + Page table would have 1 million entries
  + Each entry is 4 bytes (2 for page no, 2 for offset),
  + Total size of page table MB of memory space required per process.
    - Huge portion only for the page table
    - Needs to be allocated **contiguously** in physical memory
    - Costs a lot for small memories
* Alternative constructions include:
  + **Hierarchical** page tables
  + **Hashed** page tables
  + **Inverted** page tables

### Hierarchical Page Tables

* Diagram

  Description automatically generatedBreak up **entire logical space** into multiple **page tables**
* Then construct another table referring to each such page tables
  + e.g. two level page table:
* Advantages:
  + Allows us to only need to store parts of the page table, so we can store the logical address space actually needed.
* Logical address (32 bit machine with 1K page size) divided into:
  + 22-bit page number
  + 10 bit offset
* Page table is paged, but the 22-bit page number further divided into:
  + Table

    Description automatically generated12-bit **outer** page number
  + 10-bit **inner** page number
* Chart, box and whisker chart

  Description automatically generatedNew logical address with:
  + : index for outer page table
  + : index for inner page table
* This does not work in a 64-bit system, as the outer page table is still too long.
* We can instead use a **three-level scheme**:
  + Suppose a page in a system is 4KB large. There are bytes, so 12 bit offset per page
  + With no hierarchy, page table would be entries in the page table.
  + In a two level scheme, inner pages can be 1 page long ( 4 byte entries) , so outer pages would have
  + Using 3 level scheme, we can use another page to reference inner pages, leaving entries for outer page.
* We could keep increasing the levels for the scheme until the outer page is an appropriate size

Graphical user interface, application, table

Description automatically generated

### Hashed Page Tables (not assessed)

* **Logical Page number** is **hashed** into page table index
* Each entry **chains** elements hashing to the same location
* Each element contains:
  + Logical page number
  + Value of mapped page frame
  + Pointer to next element
* Logical page numbers are compared searching for a **match**
  + If match found, corresponding physical frame is extracted
* Variation for 64-bit is **clustered page tables**
  + Similar to hashed but each entry refers to several pages
  + Searching is then required

Chart, diagram

Description automatically generated

### Inverted Page Tables

* Hierarchical page tables had the problem that outer page tables ended up being very large.
* Rather than each process having a page table and keeping track of all **possible** logical pages:
  + **track all physical pages**
* One entry for each physical page in memory.
* Entry consists of
  + Virtual address of the page stored in physical memory location
  + Information about the process that owns the page (protection)
* This often requires an ASID to be stored in each entry of the page table, as it ensures that a logical page for a particular process is mapped to the corresponding physical frame.
* Decreases memory needed to store translations
  + Increases time needed to search the table
  + Use hash table to limit the search to one/few page-table entries
    - Means that memory reference requires 2 real memory reads: one for hash table entry and one for page table.
* How to implement shared memory?
  + Only one virtual address can be mapped to a physical address at any one time.
  + Need OS intervention

Diagram

Description automatically generated

# Examples

## IA-32 Architecture

* Memory management divides into two components: **segmentation** and **paging**

1. CPU generates logical addresses that are given to segmentation unit
2. Segmentation unit produces linear address for each logical address
3. Linear address is given to paging unit, which generates physical address in main memory.

### Segmentation

* Diagram

  Description automatically generatedSegment can be as large as 4GB, with maximum number of segments per process being 16K.
* Logical address space divided into 2 partitions:
  + First partition contains segments that are private to process.
    - Information kept in **local descriptor table** (**LDT**)
  + Second partition contains segments that are shared among all processes.
    - Information kept in **global descriptor table** (**GDT**).
  + Each table contains 8 byte-segment descriptor with detailed information about a particular segment, including:
    - Base Location
    - Limit of segment
* Graphical user interface, application

  Description automatically generated with medium confidenceLogical address is pair (selector, offset), where selector is a 16 bit number.
  + s: segment number
  + g: indicates whether segment is in GDT or LDT
  + p: protection
* Offset is 32-bit number, specifying location of byte within the segment in question.
* Base and limit information used to generate a **linear address**.

### Paging

* Chart, box and whisker chart

  Description automatically generated with medium confidenceIA 32-bit architecture allows page size of 4 KB or 4MB.
* Uses **two level** paging scheme with division shown.

# Why is Paging Important

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence