Paging

# 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** chunks
* Diagram

  Description automatically generatedPhysical address space divided into **fixed size** chunks of same size.
* This allows chunks of a program to slot anywhere into memory.
* Solves fragmentation problem

## Address Translation Scheme

* We need to be able to map logical chunks to physical chunks
  + Logical chunks are called **pages**.
  + Physical chunks called **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

### Diagram Description automatically generatedExample

* **Page table**: array of page table entries. p indexes into table: stored in RAM
* **Page table entry** (**PTE**): frame base address and bits/flags
* We obtain the frame by matching with the page number, and use the offset to specify where in the frame we want to access.

A screenshot of a computer program

Description automatically generated with medium confidence

* Logical address 4 translates to bytes 24-27:
  + Address 4 corresponds to
    - page = 1
    - offset = 3
  + Page 1 in page table gives frame 6
  + Offset e is added, resulting in bytes 24-27

## 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 long translation table

## Free Frames

* All free frames kept in a list

Diagram

Description automatically generated

## Many Processes

**Diagram

Description automatically generated**

* Each process has its own page table.
  + One implementation that doesn’t need a table per process

# Page Tables

* Page tables are kept in main memory
* The CPU needs to know where the page table is.
  + **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 PCU**
* 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

### Diagram Description automatically generatedPaging Hardware with TLB

### 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**
* Consider , 100ns for memory access
* Consider , 100ns for memory access
* As you can see the TLB can have a significant impact on access time.

## Memory Protection

* Page tables have many PTEs (page table entries)
* Memory protection accomplished by **protection bits** in each PTE. These can detail:
  + What **type of access** is allowed? Read, read-write, execute only?
  + 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

## 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
    - Huge portion only for the page table
    - Need to allocate contiguously in physical memory
    - Costs a lot for small memories
      * 0.1% of the physical address space with 4GB
* 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:
  + 32-bit page number
  + 10 bit offset
* Page table is paged, but 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

Graphical user interface, application, table

Description automatically generated

* (go and use book to refine notes)

### 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

* 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)
* 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
* How to implement shared memory?
  + Need OS intervention

Diagram

Description automatically generated

# Why is Paging Important

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence