Lecture 12/13: Paging/Virtual Memory (1)
A System Using Physical Addressing

- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames
Virtual Addresses

Many ways to do the mapping…
- MMU: Memory management unit
  - Need hardware support and OS management algorithms
  - MMU is part of the CPU

Requirements
- Need protection – restrict which addresses jobs can use
- Fast translation – lookups need to be fast
- Fast change – updating memory hardware on context switch
Virtual Addresses

- To make it easier to manage the memory of processes running in the system, we’re going to make them use virtual addresses (logical addresses)
  - Virtual addresses are independent of the actual physical location of the data referenced
  - OS determines location of data in physical memory

- Instructions executed by the CPU issue virtual addresses
  - Virtual addresses are translated by hardware into physical addresses (with help from OS)
  - The set of virtual addresses that can be used by a process comprises its virtual address space
Virtual address space/Linux

- **Linking**
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address

- **Loading**
  - `execve()` allocates virtual pages for `.text` and `.data` sections = creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system

<table>
<thead>
<tr>
<th>Kernel virtual memory</th>
<th>User stack (created at runtime)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory-mapped region for shared libraries</td>
</tr>
<tr>
<td></td>
<td>Run-time heap (created by <code>malloc</code>)</td>
</tr>
<tr>
<td></td>
<td>Read/write segment (.data, .bss)</td>
</tr>
<tr>
<td></td>
<td>Read-only segment (.init, .text, .rodata)</td>
</tr>
<tr>
<td></td>
<td>Unused</td>
</tr>
</tbody>
</table>

Memory invisible to user code

%esp (stack pointer)

`brk` Loaded from the executable file

- `0xc0000000`
- `0x40000000`
- `0x08048000`
- `0`
HOW TO SHARE MEMORY?
Fixed Partitions

- Physical memory is broken up into fixed partitions
  - Size of each partition is the same and fixed
  - Hardware requirements: base register
  - Physical address = virtual address + base register
  - Base register loaded by OS when it switches to a process
Fixed Partitions

How do we provide protection?
Fixed Partitions

- **Advantages**
  - Easy to implement
    - Need base register
    - Verify that offset is less than fixed partition size
  - Fast context switch

- **Problems?**
  - Internal fragmentation: memory in a partition not used by a process is not available to other processes
  - Partition size: one size does not fit all (very large processes?)
Variable Partitions

- Natural extension – physical memory is broken up into variable sized partitions
  - Hardware requirements: base register and limit register
  - Physical address = virtual address + base register

- Why do we need the limit register?
  - Protection: if (virtual address > limit) then fault
Variable Partitions

Virtual Address

Offset

Base Register
P3’s Base

Limit Register
P3’s Limit

Virtual Address + Offset = Result

Yes?

No?

Protection Fault

P1

P2

P3
Variable Partitions

- Advantages
  - No internal fragmentation: allocate just enough for process

- Problems?
  - External fragmentation: job loading and unloading produces empty holes scattered throughout memory
Paging

- New Idea: split virtual address space into multiple partitions
  - Each can go anywhere!

Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory

But need to keep track of where things are!
Page Lookups

Virtual Address
- Page number
- Offset

Page Table
- Page frame

Physical Address
- Page frame
- Offset

Physical Memory
Paging Advantages

- Easy to allocate memory
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem
    » All pages of the same size

- Simplifies protection
  - All chunks are the same size
  - Like fixed partitions, don’t need a limit register

- Simplifies virtual memory – later
Paging Limitations

- Can still have internal fragmentation
  - Process may not use memory in multiples of a page

- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - What can we do?

- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = $2^{20}$ PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - What can we do?
Segmentation

- Segmentation: partition memory into logically related units
  - Module, procedure, stack, data, file, etc.
  - Units of memory from user’s perspective

- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process
  - Fixed partition: Paging :: Variable partition: Segmentation

- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table
  - Virtual addresses become <segment #, offset>
Segment Lookups

Segment Table

Segment #  Offset

Physical Memory

Virtual Address

Virtual Address

Protection Fault

If the virtual address is less than the limit, the segment table is searched. If the segment table is reached, a protection fault occurs.
Summary of sharing memory

- Single partition
  - Good: easier translation. Bad: Inflexible
  - Fixed size: internal fragmentation
  - Variable size: external fragmentation

- Multiple partitions
  - Good: flexible. Bad: need a page/segment table
  - Fixed size: paging
  - Variable size: segmentation
Introducing Virtual Memory (VM)

- Virtual memory is paging with a new ingredient
  - Allow pages to be on disk
    » In a special partition (or file) called swap

- Motivation?
  - Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space

- Simplifies memory management
  - Each process gets the same uniform linear address space
  - With VM, this can be big!
**VM as a Tool for Caching**

- *Virtual memory* is an array of $N$ contiguous bytes stored on disk.
- The contents of the array on disk are cached in *physical memory (DRAM cache)*
  - These cache blocks are called *pages* (size is $P = 2^p$ bytes)

![Diagram showing virtual and physical memory mapping]

- **Virtual memory**
  - VP 0
  - VP 1
  - VP $2^{n-p}-1$

- **Physical memory**
  - PP 0
  - PP 1
  - PP $2^{m-p}-1$
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about $10^1$ slower than SRAM
  - Disk is about $10^4$ slower than DRAM

- Consequences
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from CPU caches
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through
A **page table** is an array of page table entries (PTEs) that maps virtual pages to physical pages.

- Per-process kernel data structure in DRAM
**Page Hit**

- **Page hit**: reference to VM word that is in physical memory (DRAM cache hit)

![Diagram of Page Hit]

- **Virtual address**
- **Physical page number or disk address**
- **Valid**
- **Memory resident page table (DRAM)**
- **Physical memory (DRAM)**
- **Virtual memory (disk)**

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- VP 0
- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6
- VP 7
**Page Fault**

- **Page fault**: reference to VM word that is not in physical memory (DRAM cache miss)
Handling Page Fault

- Page miss causes page fault (an exception)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)

Virtual address

Physical page number or disk address

Physical memory (DRAM)

Virtual memory (disk)

Memory resident page table (DRAM)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!
Locality to the Rescue!

- Virtual memory works because of locality

- At any point in time, programs tend to access a set of active virtual pages called the **working set**
  - Programs with better temporal locality will have smaller working sets

- If (working set size < main memory size)
  - Good performance for one process after compulsory misses

- If (SUM(working set sizes) > main memory size)
  - **Thrashing**: Performance meltdown where pages are swapped (copied) in and out continuously
VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well chosen mappings simplify memory allocation and management

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Address translation

Physical Address Space (DRAM)

(e.g., read-only library code)
VM as a Tool for Memory Management

- Memory allocation
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)
Sharing

- Can map shared memory at same or different virtual addresses in each process’ address space
  - Different:
    » 10th virtual page in P1 and 7th virtual page in P2 correspond to the 2nd physical page
    » Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  - Same:
    » 2nd physical page corresponds to the 10th virtual page in both P1 and P2
    » Less flexible, but shared pointers are valid
Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()

- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in parent and child
    » Reads happen as usual
    » Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction

- How does this help fork()?
Execution of fork()
fork() with Copy on Write

When either process modifies Page 1, page fault handler allocates new page and updates PTE in child process.

Parent process’s page table:
- Page 1
- Page 2

Child process’s page table:
- Page 1
- Page 2

Physical Memory:

Protection bits set to prevent either process from writing to any page.
VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

Physical Address Space

- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11
Address Translation With a Page Table

Virtual address

Page table base register (PTBR)

Page table address for process

Valid bit = 0: page not in memory (page fault)

Physical address

Valid

Physical page number (PPN)

Physical page offset (PPO)
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction
Integrating VM and Cache

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Elephant(s) in the room

- Problem 1: Translation is slow!
  - Many memory accesses for each memory access
  - Caches are useless!

- Problem 2: Page table can be gigantic!
- We need one for each process
- All your memory are belong to us!

“Unfortunately, there’s another elephant in the room.”