Computer Systems  CEN591(502)
Fall 2011

Sandeep K. S. Gupta
Arizona State University
16th lecture
-Virtual Memory
(Slides adapted from CSAPP)
Previous class summary

- Processes
  - What is the process?
  - Systems calls for process creation and termination in Linux.

- This class:
  - Virtual Memory
Agenda

- Virtual memory
  - What is the virtual memory?
  - VM as a tool for caching
  - VM as a tool for memory management
  - VMAddress translation
Example (using wait for synchronizing Childs and parent and checking the Childs' status)

- Use fork to parallelize the summation over an array

```c
pid_t pid[2];
if (((pid[0] = fork()) == 0) { /* Child runs user job */
    for(i=0;i<5;i++)
        sum=sum+a[i];
    exit(sum);
}
else{
    if (((pid[1] = fork()) == 0) { /* Child runs user job */
        for(i=5;i<10;i++)
            sum=sum+a[i];
        exit(sum);
    }
    else{
        int child_status;
        for (i = 1; i >= 0; i--) {
            pid_t wpid = waitpid(pid[i], &child_status, 0);
            if (WIFEXITED(child_status)) {
                sum=sum+WEXITSTATUS(child_status);
                printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
            }
            else
                printf("Child %d terminated abnormally\n", wpid);
        }
        printf("Parent process: The sum of array is:%d\n",sum);
    }
}
```

How can child communicate with the parent?
- exit status
- shared memory
- files
...

First child to the sum on the first part of the array

Second child to the sum on the second part of the array

Collect the results and calculate the final result (child pass their results by their status)

Sample input: 1 1 1 1 2 2 2 2
Sample output:
Child 18005 terminated with exit status 10
Child 18004 terminated with exit status 5
Parent process: The sum of array is: 15
Virtual memory
A System Using Physical Addressing

- Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames.

Diagram:
- CPU
- Physical address (PA)
- Main memory
- Data word
- M-1
A System Using Virtual Addressing

- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science
Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:
  \{0, 1, 2, 3 \ldots \}

- **Virtual address space:** Set of \( N = 2^n \) virtual addresses
  \{0, 1, 2, 3, \ldots, N-1\}

- **Physical address space:** Set of \( M = 2^m \) physical addresses
  \{0, 1, 2, 3, \ldots, M-1\}

- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory:
  one physical address, one (or more) virtual addresses
Why Virtual Memory (VM)?

- 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

- Isolates address spaces
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information
VM as a tool for caching
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)

Virtual memory

<table>
<thead>
<tr>
<th>Virtual Pages (VPs) stored on disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0</td>
</tr>
<tr>
<td>Unallocated</td>
</tr>
<tr>
<td>Cached</td>
</tr>
<tr>
<td>Uncached</td>
</tr>
<tr>
<td>Unallocated</td>
</tr>
<tr>
<td>Cached</td>
</tr>
<tr>
<td>Uncached</td>
</tr>
<tr>
<td>VP 1</td>
</tr>
<tr>
<td>Unallocated</td>
</tr>
<tr>
<td>Cached</td>
</tr>
<tr>
<td>Uncached</td>
</tr>
<tr>
<td>VP ( 2^{n-p-1} )</td>
</tr>
<tr>
<td>Uncached</td>
</tr>
</tbody>
</table>

Physical memory

<table>
<thead>
<tr>
<th>Physical Pages (PPs) cached in DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 0</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>PP 1</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>Empty</td>
</tr>
<tr>
<td>PP ( 2^{m-p-1} )</td>
</tr>
<tr>
<td>Empty</td>
</tr>
</tbody>
</table>

CEN591 Fall 2011
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about $10x$ slower than SRAM
  - Disk is about $10,000x$ 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
Page Tables

- 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

![Diagram of page tables](image)
Page Hit

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

![Diagram of virtual memory and page tables]

**Virtual address**

**Physical page number or disk address**

**Memory resident page table (DRAM)**

**Virtual memory (disk)**

**Physical memory (DRAM)**

- PP 0
  - VP 1
  - VP 2
  - VP 7
  - VP 4

- PP 3
  - VP 1
  - VP 2
  - VP 3
  - VP 4
  - VP 6
  - VP 7
Page Fault

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

![Diagram of page fault and memory management](image)

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

Virtual address

![Diagram of memory resident page table (DRAM) and physical memory (DRAM)]

Physical page number or disk address

Memory resident page table (DRAM)

Virtual memory (disk)

Physical memory (DRAM)

Valid

PTE 0

0 1 1 1 0 0 0 1

null

PTE 7

0 0 1

null
Locality to the Rescue Again!

- 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
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 and Process 2](image)

- 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)

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Address translation

Physical Address Space (DRAM)

(e.g., read-only library code)
Simplifying Linking and Loading

- **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

---

**Memory Map**

- Kernel virtual memory
- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by `malloc`)
- Read/write segment (.data, .bss)
- Read-only segment (.init, .text, .rodata)
- Unused

---

CEN591 Fall 2011
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)

### Process i:

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<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>

### Process j:

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<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>
VM Address Translation

- Virtual Address Space
  - \( V = \{0, 1, \ldots, N-1\} \)

- Physical Address Space
  - \( P = \{0, 1, \ldots, M-1\} \)

- Address Translation
  - \( MAP: V \rightarrow P \cup \emptyset \)
  - For virtual address \( a \):
    - \( MAP(a) = a' \) if data at virtual address \( a \) is at physical address \( a' \) in \( P \)
    - \( MAP(a) = \emptyset \) if data at virtual address \( a \) is not in physical memory
      - Either invalid or stored on disk
Summary of Address Translation Symbols

- **Basic Parameters**
  - \( N = 2^n \): Number of addresses in virtual address space
  - \( M = 2^m \): Number of addresses in physical address space
  - \( P = 2^p \): Page size (bytes)

- **Components of the virtual address (VA)**
  - \textbf{TLBI}: TLB index
  - \textbf{TLBT}: TLB tag
  - \textbf{VPO}: Virtual page offset
  - \textbf{VPN}: Virtual page number

- **Components of the physical address (PA)**
  - \textbf{PPO}: Physical page offset (same as VPO)
  - \textbf{PPN}: Physical page number
  - \textbf{CO}: Byte offset within cache line
  - \textbf{CI}: Cache index
  - \textbf{CT}: Cache tag
Address Translation With a Page Table

Virtual address

Virtual page number (VPN)  Virtual page offset (VPO)

Physical address

Physical page number (PPN)  Physical page offset (PPO)

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

Page table address for process

Page table base register (PTBR)
Address Translation: Page Hit

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
Address Translation: Page Fault

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
Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay

- Solution: *Translation Lookaside Buffer* (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Multi-Level Page Tables

- Suppose:
  - 4KB ($2^{12}$) page size, 48-bit address space, 8-byte PTE

- Problem:
  - Would need a 512 GB page table!
    - $2^{48} \times 2^{-12} \times 2^3 = 2^{39}$ bytes

- Common solution:
  - Multi-level page tables
  - Example: 2-level page table
    - Level 1 table: each PTE points to a page table (always memory resident)
    - Level 2 table: each PTE points to a page (paged in and out like any other data)
A Two-Level Page Table Hierarchy

**Level 1 page table**
- PTE 0
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8
- (1K - 9) null PTEs

**Level 2 page tables**
- PTE 0
- ... PTE 1023

**Virtual memory**
- VP 0
- ... VP 1023
- VP 1024
- ... VP 2047

- 2K allocated VM pages for code and data
- 6K unallocated VM pages
- 1023 unallocated pages
- 1 allocated VM page for the stack

32 bit addresses, 4KB pages, 4-byte PTEs
Simple Memory System Example

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes

```
13 12 11 10 9 8 7 6 5 4 3 2 1 0
VPO
PPN
VPN
VPO

Virtual Page Number
Physical Page Number

13 12 11 10 9 8 7 6 5 4 3 2 1 0
PPO

Virtual Page Offset
Physical Page Offset
```
Simple Memory System Page Table

Only show first 16 entries (out of 256)

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Simple Memory System TLB

- 16 entries
- 4-way associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>-</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>-</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>-</td>
<td>0</td>
<td>04</td>
<td>-</td>
<td>0</td>
<td>0A</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>-</td>
<td>0</td>
<td>08</td>
<td>-</td>
<td>0</td>
<td>06</td>
<td>-</td>
<td>0</td>
<td>03</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>-</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

![Diagram of memory system cache]

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Address Translation Example #1

Virtual Address: 0x03D4

Physical Address

CEN591 Fall 2011
Address Translation Example #2

Virtual Address: $0x0B8F$

<table>
<thead>
<tr>
<th>TLBI</th>
<th>TLBT</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

VPN: $00101111000111111$

VPO: $02E$

VPN: N
TLBI: 2
TLBT: 0
TLB Hit?: N
Page Fault?: Y
PPN: TBD

Physical Address

<table>
<thead>
<tr>
<th>CO</th>
<th>CI</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

PPN: N
PPO: N

Hit?: N
Byte: TBD
Address Translation Example #3

Virtual Address: 0x0020

Physical Address

CEN591 Fall 2011
Summary

- Programmer’s view of virtual memory
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- System view of virtual memory
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions