Previous class...

- What is logical address? Who use it?
  - Describes a location in the logical memory address space
  - Compiler and CPU use it

Some slides are courtesy of Dr. Thomas Anderson and Gustavo Duarte
Process switch

• Upon process switch what is updated in order to assist address translation?
  – Contiguous allocation: base & limit registers
  – Segmentation: register pointing to the segment table (recall that each process has its own segment table)
  – Paging: register pointing to the page table; TLB should be flushed if the TLB does not support multiple processes
How is Shared Memory implemented in the Paging allocation?

- The same set of frames are pointed to by page tables of multiple processes
- Example: share memory region (*recall project 1*), kernel, shared library code, and program code
How is fork() implemented in Paging?

• Copy-on-write
  – Copy page table of parent into child process
  – Mark all writable pages (in both page tables) as read-only
  – Trap into kernel on *write* (by child or parent)
    • Consult ground truth information about the write permission
    • Allocate a frame and copy the page content, and mark it writable
    • Resume execution

• What is the benefit of CoW?
  – If a page is never modified, it will not be copied; i.e., pages are copied only when necessary
Calculation in Paging

• Given a 64-bit system, if we use the scheme of 48-bit virtual address space and 4KB page size, calculate n (# of bits for a page), and # of bits for page number
  – n = 12
  – Page # bits = 48 – 12 = 36

• How many page table entries are needed if we use array based page table? If each entry needs 4 bytes, how much memory does the page table need?
  – $2^{48}/2^{12} = 64$G entries => 256G memory
  – Multi-level page table is the solution
Multilevel Page Table

Virtual Address

10 bits 10 bits 12 bits

Root page table
(contains 1024 PTEs)

Frame # Offset

4-kbyte page table (contains 1024 PTEs)

Page Frame

Program Paging Mechanism Main Memory
What is TLB, and how does it work?

• Translation Lookaside Buffer (TLB): cache specialized for speeding up address translation
  – each entry stores a mapping from page # to frame #
• Translate the address “page # : offset”
  – If the page # is in TLB cache, get frame # out
  – Otherwise get frame # from page table, and load the (page #, frame #) mapping into TLB
Outline

• Locality and Memory hierarchy
• CPU cache
• Page cache (also called Disk Cache) and swap cache
  – Demand paging and page fault handling
  – Memory mapping
Locality (of memory reference)

- Temporal locality
  - Programs tend to reference the same memory locations multiple times
  - Example: instructions executed due to a loop

- Spatial locality
  - Programs tend to reference nearby locations
  - Example: call stack

- Due to locality, the working set of a program is usually a small fraction of the memory needed during its lifecycle
Cache

- **Cache**
  - Storing a copy of data that is faster to access than the original
  - Hit: if cache has a copy
  - Miss: if cache does not have a copy

- **Locality explains high hit ratio of cache**
  - Temporal locality: a copy of data in cache will be referenced multiple times
  - Spatial locality: when you bring data copy into cache, you also bring in the nearby data, e.g., per 64 bytes

- **Cache may refer to**
  - A generic idea Caching implemented as CPU cache, page cache, web page cache, etc.
  - Or, just CPU cache
CPU cache and page cache

• CPU cache
  – Memory in CPU that stores the recently accessed data, code, or page tables
  – It is mostly managed by hardware
  – Three types: data, code, TLB cache

• Page cache
  – Portion of main memory that stores page-sized chunks of disk data
  – It is managed by kernel
  – It is not some dedicated h/w but a good use of main memory
Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng
### Memory Hierarchy (i7 as an example)

<table>
<thead>
<tr>
<th>Cache</th>
<th>Hit Cost</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st level cache/first level TLB</td>
<td>1 ns</td>
<td>64 KB</td>
</tr>
<tr>
<td>2nd level cache/second level TLB</td>
<td>4 ns</td>
<td>256 KB</td>
</tr>
<tr>
<td>3rd level cache</td>
<td>12 ns</td>
<td>2 MB</td>
</tr>
<tr>
<td>Memory (DRAM)</td>
<td>100 ns</td>
<td>10 GB</td>
</tr>
<tr>
<td>Data center memory (DRAM)</td>
<td>100 μs</td>
<td>100 TB</td>
</tr>
<tr>
<td>Local non-volatile memory</td>
<td>100 μs</td>
<td>100 GB</td>
</tr>
<tr>
<td>Local disk</td>
<td>10 ms</td>
<td>1 TB</td>
</tr>
<tr>
<td>Data center disk</td>
<td>10 ms</td>
<td>100 PB</td>
</tr>
<tr>
<td>Remote data center disk</td>
<td>200 ms</td>
<td>1 XB</td>
</tr>
</tbody>
</table>

Each core has its own 1st & 2nd level cache
3rd level cache is (2MB per core) shared among cores in a processor
Intel i7

Integrated Memory Controller - 3 Ch DDR3

Core 0  Core 1  Core 2  Core 3

Shared L3 Cache
Intel i7

32k L1 I-cache
32k L1 D-cache
32K L1 I-cache
32K L1 D-cache
32k L1 I-cache
32k L1 D-cache
32k L1 I-cache
32k L1 D-cache

256K L2 cache
data + inst.
256K L2 cache
data + inst.
256K L2 cache
data + inst.
256K L2 cache
Data + inst.

8 MB L3 cache

For all applications to share
Inclusive cache policy to minimize traffic from snoops
Intel i7

- The terms inside the table will be explained

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Instruction TLB</th>
<th>Data DLB</th>
<th>Second-level TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>128</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
<td>4-way</td>
<td>4-way</td>
</tr>
<tr>
<td>Replacement</td>
<td>Pseudo-LRU</td>
<td>Pseudo-LRU</td>
<td>Pseudo-LRU</td>
</tr>
<tr>
<td>Access latency</td>
<td>1 cycle</td>
<td>1 cycle</td>
<td>6 cycles</td>
</tr>
<tr>
<td>Miss</td>
<td>7 cycles</td>
<td>7 cycles</td>
<td>Hundreds of cycles to access page table</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>32 KB I/32 KB D</td>
<td>256 KB</td>
<td>2 MB per core</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way I/8-way D</td>
<td>8-way</td>
<td>16-way</td>
</tr>
<tr>
<td>Access latency</td>
<td>4 cycles, pipelined</td>
<td>10 cycles</td>
<td>35 cycles</td>
</tr>
<tr>
<td>Replacement scheme</td>
<td>Pseudo-LRU</td>
<td>Pseudo-LRU</td>
<td>Pseudo-LRU but with an ordered selection algorithm</td>
</tr>
</tbody>
</table>
Definitions

• Separate vs. unified CPU cache
  – Separate: instruction and data cache are separate
  – Unified: the cache can store both instruction and data
Cache structure

• Cache line (or, cache block)
  – “Cell” (think about a cell in an excel spreadsheet)
  – It is the unit of cache storage, e.g., 64 bytes, 128 bytes
  – # of cache blocks = cache size / cache block size

• Way
  – “Column”
  – # of ways (or, set associativity): # of choices for caching data
  – E.g., 4-way cache (or, 4-way set associative cache)
  – 1-way cache is called direct mapped

• Set
  – “Row”
  – # of sets: # of cache blocks / # of ways
  – 1-set cache is called fully associative cache
Index and tag and their use for cache search

• Index (or, set index)
  – determines which set ("row") in cache stores the data
  – # of bits for index = \( \log_2(\text{# of sets}) \)

• Tag
  – after the set (i.e., row) is located, determine which cache block it is by comparing the tags
  – # of bits for tag = bits for physical address – bits for offset – bits for index

| tag | index | Block offset |
Cache structure

64 rows * 64 bytes
4KB per way

4KB * 8 = 32KB total
Question

- Given 32KB 8-way cache with cache block = 64 bytes, calculate
  - # of cache blocks and # of sets
  - How to split the 36-bit physical address?

- # of cache blocks = cache size / size of basic block
  \[= \frac{2^{15}}{2^6} = 2^9\]

- # of sets = # of cache blocks / # of ways
  \[= \frac{2^9}{2^3} = 2^6\]

| 24 bits for tag | 6 bits for index | 6 bits for block offset |
Step 1: locate the set using set index

L1 Cache - 32KB, 8-way set associative, 64-byte cache lines

1. Pick cache set (row) by index

36-bit memory location as interpreted by the L1 cache:

Physical Page Address (24-bit tag), aligned to 4KB | Set Index | Offset into cache line

<table>
<thead>
<tr>
<th>Directory 0</th>
<th>Way 0</th>
<th>Dir 1/ Way 1</th>
<th>...</th>
<th>Dir 6/ Way 6</th>
<th>Dir 7/ Way 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>64-byte line</td>
<td>tag</td>
<td>line</td>
<td>tag</td>
<td>line</td>
</tr>
<tr>
<td>62</td>
<td>64-byte line</td>
<td>tag</td>
<td>line</td>
<td>tag</td>
<td>line</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>64-byte line</td>
<td>tag</td>
<td>line</td>
<td>tag</td>
<td>line</td>
</tr>
<tr>
<td>2</td>
<td>64-byte line</td>
<td>tag</td>
<td>line</td>
<td>tag</td>
<td>line</td>
</tr>
<tr>
<td>1</td>
<td>64-byte line</td>
<td>tag</td>
<td>line</td>
<td>tag</td>
<td>line</td>
</tr>
<tr>
<td>0</td>
<td>64-byte line</td>
<td>tag</td>
<td>line</td>
<td>tag</td>
<td>line</td>
</tr>
</tbody>
</table>

64 rows * 64 bytes
4KB per way

4KB * 8 = 32KB total
Step 2: locate the cache line using tag

2. Search for matching tag in the set

36-bit memory location as interpreted by the L1 cache:

Physical Page Address (24-bit tag), aligned to 4KB

Set Index

Offset into cache line

24-bit tag 64-byte line

Directory 0 Way 0

match

Dir 1/ Way 1

... Dir 6/ Way 6

Dir 7/ Way 7

Cache hit
Upon cache miss

• Due to cache misses, CPU waste cycles, called *stalls*, which hurts performance. What does CPU do in this situation?
  – Out-of-order execution: execute instructions that does not rely on the instruction waiting for data
  – Hyper-threading: allows an alternate thread to use the CPU core while the first thread waits for data
Next, we will discuss demand paging
Demand Paging

- A frame is not allocated for a page until the page is accessed (*i.e.*, upon a page fault)
- As opposed to Anticipatory Paging, which needs the kernel to actively guess which pages to map
- A process begins execution with no pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory
- Lazy to save resource
Page fault handling

1. TLB miss
2. Page table walk
3. Page fault (e.g., permission violation, present bit = 0)
4. Trap to kernel
5. Kernel checks if the memory reference is valid. If not, a signal is generated (illegal memory access). Otherwise, there are multiple possibilities
   - Write to a CoW page
   - Stack growth
   - **Page is in disk** (i.e., present bit = 0; let's assume the current page fault is due to this reason)
6. Convert virtual address to file + offset
7. Allocate page frame
   - Evict an old page if free memory runs low
8. Initiate disk block read into page frame
9. Disk interrupt when DMA complete
10. Mark page as valid
11. Resume process at faulting instruction
12. TLB miss (again 😞)
13. Page table walk for address translation (now TLB entry is added by the h/w)
14. Submit the addr to memory bus
File-backed pages vs. anonymous pages

• File-backed pages are associated with disk files
  – Such as code, pdf, jpg

• Anonymous pages are not
  – Such as heap and stack
  – When they are to be evicted for the first time, the system allocates space for them from swap. They thus become swap-backed anonymous pages
  – Now, the system can evict them (by copying them to swap area), and free the memory
Page cache for file-backed pages

- A lot of systems actually use *read-ahead* demand paging for *file-backed pages* (e.g. code, jpg, pdf)
  - Instead of merely reading in the page being referenced, the system reads in several consecutive pages to exploit spatial locality
  - Page frames used for such purpose are *page cache*
- When a page fault occurs due to accessing a disk file, the system will first check whether the target page is already stored in page cache. If so, the system need not sleep the process to wait for a new disk read
- Page cache is to speed up access to disk files, just like CPU cache is to speed up access to memory
Questions

• Will the pages for code of a program be swapped out to the swap area of the disk?
  No. Code is read-only. There is no need to swap out code pages. The kernel can directly reclaim the page frames and use for other purposes.

• Will the pages for your PDF file be swapped out to the swap area of the disk?
  No. Such pages are backed by the PDF file on the disk. When memory runs low, the kernel can either reclaim the corresponding page frames directly (if the pages stored there are not modified) or write them back to the file space on the disk before reclaiming the memory.
Memory region types

- File-backed vs. anonymous: associated with disk files or not
- Private vs. shared: “private” means when you write, you will write to your own copy
  - In case it is shared, CoW will be enforced upon the write; e.g., fork()
Let’s rock!

• You will know the secrets about
  – Compare the two types of file I/O
    • Regular file I/O: open() => read(fd, buf, size)
    • Memory mapped file I/O: open() => p = mmap(fd) => p[i]
  – What does memory management do for main()?
  – How does heap/stack grow?
Regular file I/O

- Copy-based file I/O
  - read(): data copied from page cache to user buffer
    - Certainly, before the buffer copy, the file content on disk should be copied to page cache first
  - write(): data copied from user buffer to page cache
    - Then, the kernel will automatically flush the modified page cache to the disk
- Observations:
  - You need read()/write() system calls
  - You need memory copy from page cache to your buffers
  - Assume you need to read in a large amount of data; you either allocate a small buffer and copy it piece by piece (hence, many system calls), or you allocate a large buffer (hence, large memory overhead at once)
Regular file I/O (e.g., to read in 512 bytes)

1. Render asks for 512 bytes of scene.dat starting at offset 0.

2. Kernel searches the page cache for the 4KB chunk of scene.dat satisfying the request. Suppose the data is not cached.

3. Kernel allocates page frame, initiates I/O requests for 4KB of scene.dat starting at offset 0 to be copied to allocated page frame

4. Kernel copies the requested 512 bytes from page cache to user buffer, read() system call ends.

```
render
read(scene.dat, into heap buffer, 512);
Kernel

Kernel
Find scene.dat #0
libc.so #14
free
/bin/ls #2
libc.so #10
/bin/vim #4

render
Copy 512 bytes
disk

libc.so #14
scene.dat #0
/bin/ls #2
libc.so #10
/bin/vim #4
```
Regular file I/O, e.g., reading 12kB

- The copy-based I/O wastes not only CPU time but also memory
- Why don’t we operate on page cache directly?
  - Is it possible?
Memory mapped file I/O

- Memory mapped I/O
  - \( fd = \text{open}() \Rightarrow p = \text{mmap}(fd) \Rightarrow p[i] \)
  - Now you can operate on “p” as if it points to an array
  - Open file as a memory segment
  - Program accesses it like accessing an array

- Three steps
  - File data loaded into page cache upon page fault handling
  - Virtual address area then points to page cache
  - Program directly operate on page cache
Effect of memory mapped file I/O

- It is faster for two reasons:
  - No system calls such as read()/write() due to access
  - Zero-copy: operate on page cache directly
- It also saves memory for two reasons:
  - Zero-copy: no need to allocate memory for buffers
  - Lazy loading: page cache is only allocated for data being accessed (compared to allocating a big buffer for storing the data to be accessed)
Questions

• How to improve the performance of Project 1 in terms of reading the file of content from the disk?
• Use “p = mmap(fd)” and then use “p” as a big array that stores the file content
How is main() invoked – memory part

• Program code is accessed using memory mapped I/O

• Demand paging
  – Initially, no page cache is allocated for it; no page table entry allocated either
  – When the first instruction is being executed, it triggers a page fault
  – Page fault handling: read file data to page cache, then allocate page table entries to point to the page cache
  – Resume the execution of the first instruction
How does heap/stack grow?

1. Program calls brk() to grow its heap

   Heap
   Size: 8KB, Rss: 8KB
   
   free
   anonymous
   free
   anonymous
   free

2. brk() enlarges heap VMA. New pages are not mapped onto physical memory.

   Heap
   Size: 16KB, Rss: 8KB
   
   free
   anonymous
   free
   anonymous
   free

3. Program tries to access new memory. Processor page faults.

4. Kernel assigns page frame to process, creates PTE, resumes execution. Program is unaware anything happened.

   free
   anonymous
   free
   anonymous
   free

   Size: 16KB, Rss: 8KB

   free
   anonymous
   anonymous
   anonymous
   anonymous
   free

   Size: 16KB, Rss: 12KB
Transition between page status

- Code (.text)
- Initialized global data (.data)
- File-backed mmap

- File-backed page (Page cache)

- Copy-on-write, e.g., due to write to a global variable, or write to a MAP_PRIVATE area

- Stack/heap
- Uninitialized global data (.bss)
- Anonymous mmap

- Anonymous non-swap-backed page

- Anonymous swap-backed page (Swap cache)

- To be evicted

- Copy-on-write, e.g., due to write after fork()

- Disk

- Swapping
The diagram illustrates the layout of page frames in memory for two render instances. The page frames are divided into sections for stack, libc.so, scene.dat, heap, and text. Each section contains entries for anonymous memory pages and specific files such as /lib/libc.so, scene.dat, and render. The diagram shows how memory is allocated and managed between the two render instances.
Page cache vs. Disk buffer

- Page cache is inside main memory, while disk buffer is inside the hard disk or SSD
- Page cache is managed by kernel, while disk buffer is managed by the disk controller h/w
- Both are implemented using RAM, though
- They play similar roles
  - To bridge the speed gap between memory and disk
  - Perform pre-fetch (read-ahead) and buffer writes
Summary

- Locality and cache
- Memory hierarchy
- Page fault handling and demand paging
- Page cache and file-backed pages
- Swap cache and anonymous pages
Reference

• Duarte’s series: http://duartes.org/gustavo/blog/post/page-cache-the-affair-between-memory-and-files/

• Tmpfs in linux: https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt

• Swapping and the page cache: http://repo.hackerzvoice.net/depot_madchat/ebooks/Mem_virtuelle/linux-mm/pagecache.html

• Linux emory management: http://www.tldp.org/LDP/tlk/mm/memory.html

• Swap management: https://www.kernel.org/doc/gorman/html/understand/understand014.html
Writing Assignments

• What is the relation between CoW and sharing page frames?
• How does memory mapped IO works? What is the benefit over “open/read/write”?
• How does your memory subsystem help you start process quickly (i.e., your code and data don’t need to be copied to the memory before execution)?
• Do you swap out code pages?
Backup slides
Virtual vs. physical index & tag

- Physically indexed, physically tagged (PIPT)
  - Caches use the physical address (PA) for both the index and the tag.
  - Simple but slow, as the cache access must wait until physical address is generated.
- Virtually indexed, virtually tagged (VIVT)
  - Caches use the virtual address (VA) for both index and tag
  - Very fast as MMU is not consulted
  - Complex to deal with aliasing
    - Multiple virtual addresses and cache blocks point to the same memory cell, which makes it expensive to maintain consistency
Complete Picture of Memory Access

Processor → Virtual Address

Virtual Cache

Virtual Address Miss → TLB

TLB

TLB Miss → Page Table

Page Table

Invalid → Raise Exception

Valid Frame → Physical Cache

Physical Cache

Physical Address Miss → Physical Memory

Physical Memory

Hit Frame → Offset

Offset → Physical Address

Physical Address Hit → Data

Data Hit → Processor
Virtual vs. physical index & tag

• Virtually indexed, physically tagged (VIPT)
  – caches use VA for the index and PA in the tag
  – Faster than PIPT, as TLB translation and indexing into cache sets are parallel
  – After PA is obtained, the tag part is used to locate cache line
  – No need to deal with aliasing and context switch

• Physically indexed, virtually tagged (PIVT)
  – no sense

• Intel i7: L1 cache uses VIPT, and L2 and L3 use PIPT
Consistency issues between TLB and Ground Truth upon permission reduction

• Permission Reduction
  – When a page is swapped out
  – When R/W pages become read-only due to fork()
  – These events lead to Permission Reduction

• What if we do nothing on permission reduction?
  – TLB, as hardware, has no idea about it
  – It will allow your process to access those frames

• So OS has to explicitly invalidate the corresponding TLB entries
Consistency issues between TLB caches on different cores

• On a multicore, upon permission reduction, the OS must further ask each CPU to invalidate the related TLB entries
  – This is called TLB Shootdown
  – It leads to Inter-processor Interrupts. The initiator processor has to wait until all other processors acknowledge the completion, so it is costly
## TLB Shootdown

<table>
<thead>
<tr>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
<tr>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
</tbody>
</table>
Question

• Why is TLB shootdown performed upon permission reduction only?
  – In the case of permission increment, e.g., read-only becomes r/w, when CPU writes to the page, a page fault will be triggered; then the kernel has chance to invalidate the TLB
  – Lazy update
Copy-on-write work in fork()

- Page table entries initially point to file-backed pages and are marked as read-only
- Upon write, a page fault is triggered
  - An anonymous page is allocated and data is copied
- Later, when the page becomes cold (rarely accessed), the system will allocate swap space for it, and it is then backed by swap area
  - I.e., it becomes swap cache. Now the system can evict it
- File-backed pages (or, regular page cache)
  -> not-backed anonymous pages
  -> swap-backed pages (or, swap cache)
Copy-on-write

• All private memory mapping (MAP_PRIVATE), no matter it is backed by a file or not, uses the copy-on-write strategy

• Of course, if the mapped area is read only, a write will lead to program crash to protect the process
1. Two programs map scene.dat privately. Kernel deceives them and maps them both onto the page cache, but makes the PTEs read only.

2. Render tries to write to a virtual page mapping scene.dat. Processor page faults.

3. Kernel allocates page frame, copies contents of scene.dat #2 into it, and maps the faulted page onto the new page frame.

4. Execution resumes. Neither program is aware anything happened.
How is process virtual memory mapped?

• All the sections in executable and shared library files are mapped into process segments using private memory mapping
  – .text: read-only
  – .rodata: same as above
  – .data: global/static variables with initialization values
  – Exception: .bss (global/static variables without initialization values) uses anonymous pages
    • Why? It does not make sense to load a page of zeros from disk, which would be extremely slow

• Heap/stack use anonymous pages