CSE 120
Principles of Operating Systems

Fall 2021

Lecture 11: TLB, Swapping, Memory Allocation
Yiying Zhang
Announcements

- Midterm graded, appeal on Piazza
- Tomorrow’s discussion section will go over PR2 and some of HW3 (if have time)
- Course survey (Google form in the latest Piazza post)
Page Table Entries (PTEs)

- Page table entries control mapping
  - The **Modify** bit says whether or not the page has been written
    » It is set when a write to the page occurs
  - The **Reference** bit says whether the page has been accessed
    » It is set when a read or write to the page occurs
  - The **Valid** bit says whether or not the PTE can be used
    » It is checked each time the virtual address is used
  - The **Protection** bits say what operations are allowed on page
    » Read, write, execute
  - The **page frame number** (PFN) determines physical page

<table>
<thead>
<tr>
<th>M</th>
<th>R</th>
<th>V</th>
<th>Prot</th>
<th>Page Frame Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

- The **dirty bit** is set
- The **reset bit** is set by OS
# x86 Page Table Entry

<table>
<thead>
<tr>
<th>Page frame number</th>
<th>U</th>
<th>P</th>
<th>Cw</th>
<th>Gl</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>Cd</th>
<th>Wt</th>
<th>O</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Valid (present)
- Read/write
- Owner (user/kernel)
- Write-through
- Cache disabled
- Accessed (referenced)
- Dirty
- PDE maps 4MB
- Global

Reserved
[lec10] Paging implementation – how does it really work?

- Where to store page table?
- How to use MMU?
  - Even small page tables are too large to load into MMU
  - Page tables kept in mem and MMU only has their base addresses
- What happens at context switches?
Managing Page Tables

• How can we reduce page table space overhead?
  ♦ Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)
Managing Page Tables

• How can we reduce page table space overhead?
  ‣ Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)

• How can we be flexible?
Managing Page Tables

• How can we reduce page table space overhead?
  ‣ Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)

• How can we be flexible?
  “All computer science problems can be solved with an extra level of indirection.”
  two-level page tables
Two-Level Page Tables
[lec10] Multiple-level page tables

- Page-map L4
- Page-dir-pointer
- Page-directory
- Page-table
- Page offset

4 KB page

© 2007 Elsevier, Inc. All rights reserved.
[lec10] Multi-level page tables

• 3 Advantages?
Multi-level page tables

• 3 Advantages?
  ♦ L1, L2, L3 tables do not have to be consecutive
  ♦ They do not have to be allocated before use!
  ♦ They can be swapped out to disk!
Multi-level page tables

- 3 Advantages?
  - L1, L2, L3 tables do not have to be consecutive
  - They do not have to be allocated before use!
  - They can be swapped out to disk!

The power of an extra level of indirection!
Multi-level page tables

• 3 Advantages?
  ♦ L1, L2, L3 tables do not have to be consecutive
  ♦ They do not have to be allocated before use!
  ♦ They can be swapped out to disk!

The power of an extra level of indirection!

• Problems?
[lec10] Efficient Translations

- Our original page table scheme already increased the cost of doing memory lookups
  - Two lookups into the page table, another to fetch the data
  - One lookup and one data access for original flat page table
- Now 4-level page tables require five DRAM accesses for one memory operation!
  - Four lookups into the page tables, a fifth to fetch the data
- Solution: reference locality!
  - In a short period of time, a process is likely accessing only a few pages
  - Store part of the page table that is “hot” in a fast hardware unit
Translation Look-aside Buffer (TLB)

- Translation Look-aside Buffers
  - Translate VPNs into PFNs
- TLBs implemented in hardware
  - TLB hit is very fast \( \leq 1 \) CPU cycle
  - Fully associative cache => least conflict misses
  - New entries can be inserted anywhere in the TLB
  - All entries looked up in parallel
  - TLB can’t be made very big, typically 64 – 4096 entries

- Optional (useful) bits
  - ASIDs -- Address-space identifiers (process tags)
Translation Look-aside Buffer (TLB)
Miss handling: Hardware-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    » OS performs fault handling
    » Restart the faulting instruction
Miss handling: Hardware-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    » OS performs fault handling
    » Restart the faulting instruction

- On a TLB miss
  - MMU parses page table and loads PTE into TLB
    » Needs to replace if TLB is full
    » Page table layout is fixed
  - Same as hit …
Miss handling:
Software-controlled TLB

• On a TLB hit, MMU checks the valid bit
  ♦ If valid, perform address translation
  ♦ If invalid (e.g. page not in memory), MMU generates a page fault
    » OS performs page fault handling
    » Restart the faulting instruction
Miss handling: Software-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    » OS performs page fault handling
    » Restart the faulting instruction

- On a TLB miss, HW raises exception, *traps to the OS*
  - OS parses page table and loads PTE into TLB
    » Needs to replace if TLB is full
    » Page table layout can be *flexible*
  - Same as in a hit…
Hardware vs. software controlled

- Hardware approach
  - Efficient – TLB misses handled by hardware
  - OS intervention is required only in case of page fault
  - Page structure prescribed by MMU hardware -- rigid

- Software approach
  - Less efficient -- TLB misses are handled by software
  - MMU hardware very simple, permitting larger, faster TLB
  - OS designer has complete flexibility in choice of MM data structure
Deep thinking
Deep thinking

• Without TLB, how MMU finds PTE is fixed

• With TLB, it can be flexible, e.g. software-controlled is possible

• What enables this?

• TLB is an extra level of indirection!
More TLB Issues

• When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  ♦ Which TLB entry should be replaced?
    » Random
    » LRU

• What happens when changing a page table entry (e.g. because of swapping, change read/write permission)?
More TLB Issues

• When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  ♦ Which TLB entry should be replaced?
    » Random
    » LRU

• What happens when changing a page table entry (e.g. because of swapping, change read/write permission)?
  ♦ Change the entry in memory
  ♦ flush (eg. invalidate) the TLB entry
    » INGLPG on x86
What happens to TLB in a process context switch?

- During a process context switch, cached translations can not be used by the next process.
What happens to TLB in a process context switch?

• During a process context switch, cached translations can not be used by the next process
  ♦ Invalidate all entries during a context switch
    » Lots of TLB misses afterwards
What happens to TLB in a process context switch?

- During a process context switch, cached translations can not be used by the next process
  - Invalidate all entries during a context switch
    » Lots of TLB misses afterwards
  - Tag each entry with an ASID
    » Add a HW register that contains the process id of the current executing process
    » TLB hits if an entry’s process id matches that register
Cache vs. TLB

• Similarities:
  ♦ Both cache a part of the physical memory

• Differences:
  ♦ Associatively
    » TLB is usually fully associative
    » Cache can be direct mapped
  ♦ Coherence
    » No hardware provided coherence between TLB and main memory
    » Software needs to flush TLB entries for coherence
    » Cache: hardware-provided (via snooping bus) coherence across multiple cores and main memory
More on coherence issues
More on coherence issues

• No hardware maintains coherence between DRAM and TLBs:
  ♦ OS needs to flush related TLBs whenever changing a page table entry in memory

• On multiprocessors, when you modify a page table entry, you need to do “TLB shoot-down” to flush all related TLB entries at all the cores

many core 128
Summary so far

- Virtual memory addresses: a level of indirection to decouple static time (compiler) from run time (OS)
- Paging: avoiding external fragmentation, great flexibility
- Single-level page tables are too big
- Multi-level page tables reduce the space overhead (leveraging indirection) but increases the performance overhead
- TLB improves paging performance (leveraging locality)
- But TLB shootdown is costly (esp. on many cores)
We’ll cover more virtual memory topics:

• Optimizations
  ♦ Managing page tables (space) \(\text{last time}\)
  ♦ Efficient translations (TLBs) (time)
  ♦ Demand paged virtual memory (swapping) (space)

• Memory allocation

• Kernel address space (if have time)
[lec9] Sharing main memory

• Simple multiprogramming – 4 drawbacks
  ✓ Lack of protection
  ✓ Cannot relocate dynamically
    ➔ dynamic memory relocation: base&bound
  ✓ Single segment per process
    ➔ dynamic memory relocation: segmentation, paging

Entire address space needs to fit in mem
  » More need for swapping
  » Need to swap whole, very expensive!
The last drawback

• So far we’ve separated the process’s view of memory from the OS’s view using a mapping mechanism
  ♦ Each sees a different organization
  ♦ Allows OS to shuffle processes around
  ♦ Simplifies memory sharing
  ♦ What is the essence of the mechanism that enables this?
The last drawback

• So far we’ve separated the process’s view of memory from the OS’s view using a mapping mechanism
  ♦ Each sees a different organization
  ♦ Allows OS to shuffle processes around
  ♦ Simplifies memory sharing
  ♦ *What is the essence of the mechanism that enables this?*

• But, a user process had to be completely loaded into memory before it could run
The last drawback

- So far we’ve separated the process’s view of memory from the OS’s view using a mapping mechanism:
  - Each sees a different organization
  - Allows OS to shuffle processes around
  - Simplifies memory sharing
  - What is the essence of the mechanism that enables this?

- But, a user process had to be completely loaded into memory before it could run

→ Wasteful since a process only needs a small amount of its total memory at any time (reference locality!)
Virtual Memory

• Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory
Virtual Memory

- Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory.

- Key idea: Virtual address space translated to either
  - Physical memory (small, fast) or
  - Disk/SSD (backing store), large but slow.
Virtual Memory

- Definition: Virtual memory permits a process to run with only some of its virtual address space loaded into physical memory.

- Key idea: Virtual address space translated to either
  - Physical memory (small, fast) or
  - Disk/SSD (backing store), large but slow

- Deep thinking – what made above possible?

\[\text{indirect on}\]
Virtual Memory

• Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory

• Key idea: Virtual address space translated to either
  ♦ Physical memory (small, fast) or
  ♦ Disk/SSD (backing store), large but slow

• Deep thinking – what made above possible?

• Objective:
  ♦ To produce the illusion of memory as big as necessary
Virtual Memory

• “To produce the illusion of memory as big as necessary”
Virtual Memory

• “To produce the illusion of memory as big as necessary”
  ✷ Without suffering a huge slowdown of execution
Virtual Memory

• “To produce the illusion of memory as big as necessary”
  ♦ Without suffering a huge slowdown of execution
  ♦ What makes this possible?
Virtual Memory

• “To produce the illusion of memory as big as necessary”
  ♦ Without suffering a huge slowdown of execution
  ♦ What makes this possible?
  ♦ **Principle of locality**
    » Knuth’s estimation of 90% of the time in 10% of the code
    » There is also significant locality in data references
Virtual Memory Implementation

- Virtual memory is typically implemented via *demand paging*

  - *demand paging*:
    - Load memory pages (from storage or initially allocated) “on demand”
    - paging with swapping, e.g., physical pages are swapped in and out of memory
Demand Paging
(paging with swapping)

- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

• How to detect this?
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

• How to detect this?
  ♦ In software?
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

• How to detect this?
  ♦ In software?

![Diagram]

- CPU
- Translation (MMU)
- Physical memory
- I/O device
- virtual address
- physical address
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

  page fault handler

• Hardware/software cooperate to make things work
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

• Hardware/software cooperate to make things work
  ♦ Include a valid bit (present bit) in each PTE
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

• Hardware/software cooperate to make things work
  ♦ Include a valid bit (present bit) in each PTE
  ♦ Any page not in main memory right now has the valid bit cleared in its PTE
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

• Hardware/software cooperate to make things work
  ¦ Include a valid bit (present bit) in each PTE
  ¦ Any page not in main memory right now has the valid bit cleared in its PTE
  ¦ If valid bit isn’t set, a reference to the page results in a trap by the paging hardware, called page fault
Demand Paging

(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

• Hardware/software cooperate to make things work
  ♦ Include a valid bit (present bit) in each PTE
  ♦ Any page not in main memory right now has the valid bit cleared in its PTE
  ♦ If valid bit isn’t set, a reference to the page results in a trap by the paging hardware, called page fault
  ♦ What needs to happen when page fault occurs?
What happens at virtual memory allocation time and access time?

- **What happens at virtual memory allocation time?**
  - If demand paging (on-demand allocation) is used, the OS allocates a virtual address (more later today) and establishes a PTE with no PFN and with invalid bit set.

- **What happens when the virtual address is first accessed?**
  - The OS should allocate physical memory for it.
  - How to capture the first write to a virtual page?
    - e.g. want to trap into page fault handler
      - Use valid bit
  - In page fault handler, check if the virtual page is allocated (and access permitted)
    - If not, segmentation fault
    - Else allocate physical page and update PTE (and flush TLB)
What happens when main memory is not big enough?

- Processes running on a machine have collectively used more memory than what the physical main memory has.

- Some memory pages need to be put to storage (swap out)

- What happens when a swapped out page is accessed?
  - Need to swap in the page
  - How to detect that a swapped out page is accessed?
Page Fault Handling in Demand Paging

VM subsystem

Page Table (TLB)

physical pages

Load M

mem ref

Inst seq.
Page Fault Handling in Demand Paging

- Load M
- mem ref
- Inst seq.
- Page Table (TLB)
- VM subsystem
- fault
- physical pages
Page Fault Handling in Demand Paging

Load M

mem ref

Inst seq.

Page Table (TLB)

VM subsystem

fault

physical pages
Page Fault Handling in Demand Paging

Load M

mem ref

Inst seq.

Page Table (TLB)

VM subsystem

fault

physical pages
Page Fault Handling in Demand Paging

Load M

mem ref

Inst seq.

Page Table (TLB)

VM subsystem

fault

physical pages

32
Page Fault Handling in Demand Paging

- Load M
- mem ref
- Inst seq.
- Page Table (TLB)
- VM subsystem
- physical pages
Page Fault Handling in Demand Paging

Load M

mem ref

Inst seq.

Page Table (TLB)

VM subsystem

fault

physical pages
Page Fault Handling in Demand Paging

VM subsystem

Load M
Inst seq.

mem ref

Page Table (TLB)

fault

physical pages
Page Fault Handling in Demand Paging

- Load M
- mem ref
- Inst seq.
- Page Table (TLB)
- VM subsystem
- fault
- physical pages
Page Fault Handling in Demand Paging

VM subsystem

Load M

mem ref

Inst seq.

Page Table (TLB)

physical pages

t Fault

vm i v
Page Fault Handling in Demand Paging

VM subsystem

Load M

mem ref

mem ref

fault

Page Table (TLB)

Inst seq.

physical pages
Page fault handling (cont)

• On a page fault
  ♦ Find an unused phy page. If no unused, find a used phy. page (policy on which used one to pick in next lecture)
  ♦ If the phy. page is used
    » If it has been modified (how to know?), write it to disk
    » Invalidate its current PTE and TLB entry (how?)
  ♦ Load the new page from disk
  ♦ Update the faulting PTE and its TLB entry (how?)
  ♦ Restart the faulting instruction
Page fault handling (cont)

• On a page fault
  ◦ Find an unused phy page. If no unused, find a used phy. page (policy on which used one to pick in next lecture)
  ◦ If the phy. page is used
    » If it has been modified (how to know?), write it to disk
    » Invalidate its current PTE and TLB entry (how?)
  ◦ Load the new page from disk
  ◦ Update the faulting PTE and its TLB entry (how?)
  ◦ Restart the faulting instruction

• Supporting data structure that an OS uses
Page fault handling (cont)

- On a page fault
  - Find an unused phy page. If no unused, find a used phy. page (policy on which used one to pick in next lecture)
  - If the phy. page is used
    - If it has been modified (how to know?), write it to disk
    - Invalidate its current PTE and TLB entry (how?)
  - Load the new page from disk
  - Update the faulting PTE and its TLB entry (how?)
  - Restart the faulting instruction

- Supporting data structure that an OS uses
  - For speed: A list of unused physical pages (more later)
  - Data structure to map a phy. page to its pid(s) and virtual address(es)
  - Data structure to remember where a swapped out page is on disk
Address Translation Redux

• We started this topic with the high-level problem of translating virtual addresses into physical addresses
• We’ve covered all of the pieces
  ♦ Virtual and physical addresses
  ♦ Virtual pages and physical page frames
  ♦ Multi-level page tables and page table entries (PTEs)
  ♦ TLBs
  ♦ Demand paging
• Now let’s put it together, bottom to top
The Common Case

- The compiler compiles source code into binaries (containing memory instructions)
- OS loads the executable (a.out) into memory and starts its execution

- Process is executing on the CPU, and it issues a read to an address
  - What kind of address is it, virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the page number of the address
  2. Common case is that the page number matches, returning the physical page frame and protection bits for this address
  3. TLB validates that the protection bits allows reads (in this example)
  4. MMU combines the PFN and offset into a physical address
  5. MMU then reads from that physical address, returns value to CPU

- Note: The above execution is all done by the hardware
TLB Misses

- At this point, two other things can happen
  1. TLB does not have this virtual address
  2. Mapping in TLB, but memory access violates protection bits
- We’ll consider each in turn
Reloading the TLB

- If the TLB does not have mapping, two possibilities:
  1. MMU loads PTE from page table in memory (a page table walk)
     » Hardware managed TLB, OS not involved in this step
  2. Trap to the OS
     » Software managed TLB, OS intervenes at this point
- A machine will only support one method or the other (all modern computers have hardware-managed TLB)
Reloading the TLB

• If the TLB does not have mapping, two possibilities:
  1. MMU loads PTE from page table in memory (a page table walk)
    » Hardware managed TLB, OS not involved in this step
  2. Trap to the OS
    » Software managed TLB, OS intervenes at this point
  ♦ A machine will only support one method or the other (all modern computers have hardware-managed TLB)

• When TLB has PTE, it restarts translation
  ♦ Common case is that the PTE refers to a valid page in memory
    » Hardware just reads PTE from the page table and loads it into TLB
  ♦ Uncommon case is that TLB faults again on PTE because of PTE protection/valid bits (e.g., page is invalid (not in memory))
    » Becomes a page fault…
Page Faults

- PTE can indicate the type of a page fault
  - **Read/write/execute** – operation not permitted on page
  - **Invalid** – page not in physical memory

- TLB traps to the OS (software takes over)
  - **R/W/E** – OS usually will send fault back up to user process, or use for other purposes (e.g., copy on write)
  - **Invalid**
    - Page not in physical memory because this is the first access
      - OS allocates physical frame and sets up the PTE (and flush TLB)
    - Page not in physical memory because it has been swapped out
      - Finds an empty frame in physical memory (if none, need to swap out something first), reads the page from disk, sets up the PTE to point to the new physical frame (and flush TLB)
Memory Allocation

- Virtual memory allocation
- Physical memory allocation
- Who performs these allocations?
- How are the allocations done?
Virtual memory allocation: two general forms

• Stack
  ♦ Restricted
  ♦ Simple and efficient
  ♦ Easy to implement

• Heap
  ♦ More general
  ♦ Less efficient
  ♦ More difficult to implement
Heap organization

• Allocation & freeing are unpredictable
  ♦ For arbitrary, complex data structures

• Memory consists of allocated areas and free areas (holes) → lots of holes inevitable

• Fragmentation problem
  ♦ solution: keep # of holes small, size large
Heap organization
Heap organization

• *Fragmentation*: inefficient use of memory due to holes too small
  ♦ What happens in stack?
Heap organization

• **Fragmentation**: inefficient use of memory due to holes too small
  ♦ What happens in stack?

• Typically, heap allocation uses a *free list (or tree)* of holes
Heap organization

- **Fragmentation**: inefficient use of memory due to holes too small
  - What happens in stack?

- Typically, heap allocation uses a *free list (or tree)* of holes
- Allocation algorithms differ in how to manage the free list
Implementation
Implementation

• Bit map
  ♦ For fixed-size chunks
Implementation

• Bit map
  ♦ For fixed-size chunks
Implementation

• Bit map
  ✷ For fixed-size chunks

• Pools
  ✷ A separate allocation pool for each popular size
  ✷ Fast, no fragmentation
  ✷ But some pools may run out faster than others
Implementation – Segregated Lists

• Basic motivation: applications may use certain types of objects often
  ♦ These types have fixed sizes

• Always keep a few free objects (memory regions) of popular sizes
  ♦ One list of free objects for one size

• Example: Linux kernel slab allocator
Implementation - Buddy Allocation

• Basic idea: coalescing free regions
  ♦ Free space organized in “binary”
  ♦ Only give out power-of-two-sized blocks
  ♦ Neighboring free spaces form a bigger one
Implementation - Buddy Allocation

- Basic idea: coalescing free regions
  - Free space organized in "binary"
  - Only give out power-of-two-sized blocks
  - Neighboring free spaces form a bigger one

Internal Fragmentation!
Reclamation

- When can dynamically-allocated memory be freed?
  - Easy if a chunk is used in one place
  - Hard when a chunk is shared
  - Sharing is indicated by presence of pointers to the data

- Reference counts:
  - Keep track of the number of outstanding pointers to each chunk of memory
  - When this goes to 0, free the memory
malloc, brk, and physical memory allocation

- Who calls malloc?
- What happens at malloc time?

- What is brk?
- Who calls brk?
- What happens at brk time?

- When is physical memory allocated?
malloc and brk / mmap

Application

Allocator (libc)
1. malloc()
   free()
   realloc()
   calloc()

Virtual Memory

Heap

Mappings

Process Address Space

Physical Memory

MMU

4. page fault

lookup

2. brk()

3. mmap()
   munmap()
One last thing...

- Not in exam, but nice to know and very interesting
Kernel Address Space

• Wait…how does the OS virtual address space work?
• We have talked about it as a separate address space
• But it is typically implemented as an extension of the user-level process address space
  ♦ The bottom portion is for the user-level process
  ♦ The top portion is for the operating system/kernel
  ♦ VMS, early Unix: user 2GB, kernel 2GB (32-bit)
  ♦ Linux, Windows: user 3GB, kernel 1GB (32-bit)
Process Address Space

Address space used by process

- Stack
- Heap
- Static Data (Data Segment)
- Code (Text Segment)
Kernel Address Space

- **Stack**
  - User Address Space (3GB)
  - OS Code, Data, Heap, Drivers, ...

- **Heap**
  - User Address Space (3GB)

- **Code (Text Segment)**
  - User Address Space (3GB)

- **Static Data (Data Segment)**
  - User Address Space (3GB)

Address space used by process

Address space used by kernel

Same in all page tables

Trap to kernel
Kernel Address Space

• When CPU is in user mode, a process can only access the user-level portion
• When CPU is in kernel/privileged mode, the OS can access the entire region
• This arrangement is very convenient for the OS
  † The OS can access any memory in the user-level portion of the current process (e.g., copying system call arguments)
  † But the OS region is protected from the process
• As a result, the OS is mapped into every process
  † The upper portion of every process address space is the OS
  † Context switching effectively just switches the bottom portion
• This works well until Meltdown (mitigation: kernel page-table isolation KPTI) read more: [here](#) (Meltdown and KPTI not in exam)
Summary

Paging mechanisms:
- Optimizations
  - Managing page tables (space)
  - Efficient translations (TLBs) (time)
  - Demand paged virtual memory (space)
- Recap address translation
- Memory allocation and reclamation

Next time:
- Advanced Functionality
  - Sharing memory
  - Copy on Write
- Paging policies
Next time...

- Chapter 22
Backup Slides
Mapped Files

• Mapped files enable processes to do file I/O using loads and stores
  ♦ Instead of “open, read into buffer, operate on buffer, …”

• Bind a file to a virtual memory region (mmap() in Unix)
  ♦ PTEs map virtual addresses to physical frames holding file data
  ♦ Virtual address $\text{base + N}$ refers to offset N in file

• Initially, all pages mapped to file are invalid
  ♦ OS reads a page from file when invalid page is accessed
  ♦ OS writes a page to file when evicted, or region unmapped
  ♦ If page is not dirty (has not been written to), no write needed
    » Another use of the dirty bit in PTE
NAME
top
mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
top

#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);

See NOTES for information on feature test macro requirements.

DESCRIPTION
top

mmap() creates a new mapping in the virtual address space of the
calling process. The starting address for the new mapping is
specified in addr. The length argument specifies the length of the
mapping (which must be greater than 0).

If addr is NULL, then the kernel chooses the (page-aligned) address
at which to create the mapping; this is the most portable method of
creating a new mapping. If addr is not NULL, then the kernel takes
it as a hint about where to place the mapping; on Linux, the mapping
will be created at a nearby page boundary. The address of the new
mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping;
see MAP_ANONYMOUS below), are initialized using length bytes starting
at offset offset in the file (or other object) referred to by the

MapViewOfFile function

Maps a view of a file mapping into the address space of a calling process.

To specify a suggested base address for the view, use the MapViewOfFileEx function. However, this practice is not recommended.

Syntax

```cpp
LPVOID WINAPI MapViewOfFile(
    _In_ HANDLE hFileMappingObject,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwFileOffsetHigh,
    _In_ DWORD dwFileOffsetLow,
    _In_ SIZE_T dwNumberOfBytesToMap
);
```
Mapped Files

• Pages of file mapped one-to-one and contiguous into virtual pages in the address space
Mapped Files

- Pages do not have to be contiguous in physical memory
- Not all pages have to be in physical memory at once

Virtual Address Space

Physical Memory

File on Disk
Mapped Files (2)

• File is essentially backing store for that region of the virtual address space (instead of using the swap file)
  ♦ Virtual address space not backed by “real” files also called Anonymous VM

• Advantages
  ♦ Uniform access for files and memory (just use pointers)
  ♦ Less copying

• Drawbacks
  ♦ Process has less control over data movement
    » OS handles faults transparently
  ♦ Does not generalize to streamed I/O (pipes, sockets, etc.)