Lecture 11: Virtual Memory

CS343 – Operating Systems Branden Ghena – Fall 2020

Some slides borrowed from: Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS61C and CS162

Northwestern

Resources the OS manages

- Processor
 - Scheduling
- Memory
 - Virtual Memory
- Devices
 - Device Drivers
- Files
 - File systems

Today's Goals

• Discuss OS management of process memory with virtual memory

• Understand two virtual memory mechanisms: segmentation and paging

• Explore optimizations to memory paging

Outline

Address Spaces

- Methods of address translation
 - Segmentation
 - Paging
- Paging improvements
 - Improving translation speed
 - Improving table storage size

The reality of memory in a computer



A process's view of the memory

• The Address Space of the process

- The illusion:
 - Processes run alone on the computer
 - They have full access to every memory address
 - 2⁶⁴ bytes of memory available to them
- The reality:
 - There are many processes
 - There is only so much RAM available



Virtual memory enables this illusion

Virtual Addresses Physical Addresses ~ FFFF FFFF_{hex} ~ FFFF FFFF_{hex} stack Space **Process A** RAM Address **Process B** heap static data code OS ~ 0_{hex} ~ 0_{hex}

Why is this illusion important?

- We want to compile our programs at set addresses
 - There are alternatives to this, such as Position Independent code
 - But those alternatives often have performance costs

- But we can't know which addresses will be available
 - How would Google know which addresses Chrome could use safely?
- Plus, the amount of RAM on systems varies widely
 - Old laptop with 512 MB, Desktop with 16 GB, Server with 256 GB
 - If they run x86-64 Linux, the same program will work on all of them
 - Specialized systems, like embedded, might not need this requirement

Goals of virtual memory

- 1. Independence from other programs running
- 2. Independence from machine hardware

- 3. Security
 - Applications shouldn't be able to even *read* other memory much less write

4. Efficiency

- Allow reuse of some parts of memory
 - Code sections for threads, duplicate processes, or shared libraries
- Don't slow down the system too much by enabling the above

Virtual memory is how the OS controls memory accesses

- I/O operations are controlled by system calls
- CPU usage is controlled by the scheduler (and interrupts)

- How can the OS control memory accesses?
 - Context switch for each memory read/write is too high of a cost
- 1. Translation: hardware support for common case reads/writes
- 2. Faults: trap to OS to handle uncommon errors
 - Both will be the performed by the Memory Management Unit (MMU)

Hardware Memory Management Unit supports virtual memory



Outline

Address Spaces

- Methods of address translation
 - Segmentation
 - Paging
- Paging improvements
 - Improving translation speed
 - Improving table storage size

Share memory by splitting between whole processes

Virtual Addresses

Physical Addresses



Address translation with a base register

- Divide RAM into segments, each with a separate "**base**" address
 - Processes each get their own individual segment
 - Takes advantage of processes usually being smaller than RAM
- To get a physical address from a virtual one, add to base value



Adding protection creates "Base and Bound" translation

- Add a "bound" register with maximum value of the segment
 - Memory accesses greater than bound trigger a fault
 - No need to worry about lower bound, since minimum address is 0+base



Base and bounds evaluation

- Advantages
 - Provides protection between address spaces
 - Supports dynamic relocation of processes (even at runtime)
 - Simple, inexpensive hardware implementation
- Disadvantages
 - Process must be allocated contiguous physical memory
 - Including memory between sections that might never be used
 - Large allocations end up wasting a lot of space through fragmentation
 - No partial sharing of memory

Check your understanding – base and bound

- What are the results of the following memory reads? (16-bit)
 - Base: 0xC000 Bound: 0x1FFF
 - Read 0x0010
 - Read 0x1400
 - Read 0xD000

Check your understanding – base and bound

- What are the results of the following memory reads? (16-bit)
 - Base: 0xC000 Bound: 0x1FFF
 - Read 0x0010 -> **0xC010**
 - Read 0x1400 -> **0xD400**
 - Read 0xD000 -> Fault

What if we split the code into multiple base/bound segments?



Physical Addresses

Segmentation design

- Select some number of segments that processes may have
 - Separate base and bound register for each one
- Need to distinguish which accesses correspond to which segment
 - Solution: use top few bits of the virtual address
 - 00 -> segment 0
 - 01 -> segment 1
 - etc.
 - Only add remaining lower bits to the base register

Memory Management Unit for segmentation

- Similar comparison and addition hardware as before
- New segment table to select correct base and bounds
 - Bits from virtual address decide on the correct segment
 - Segment decides the proper base and bound selection
 - Can also apply permissions to individual segments

Segment	Base	Bound	Permissions	<u>Example</u>
0	0x2000	0x06FF	Read/Execute	Code
1	0x0000	0x04FF	Read/Write	- Stack
2	0x3000	0x0FFF	Read/Write	🔶 Data
3	0x0000	0x0000	None	- Unused

OS management of processes with segmentation

- On context switch
 - Hardware changes to kernel mode (so MMU is deactivated)
 - Save process's segment table with the rest of the process data
 - Load new process's segment table into the MMU
 - Change to user mode and jump to new process
- x86 example
 - No table, but rather registers for each segment
 - Stack Segment, Code Segment, Data Segment
 - Extra Segment, F Segment, G Segment

Segmentation evaluation

- Advantages
 - Sparse allocation of address space
 - Stack and heap segments can grow
 - Different protection for different segments
 - Only execute or write where it makes sense to
 - Still possible to do dynamic relocation and hardware still relatively simple

Disadvantages

- Still results in fragmentation of memory
 - Entire section must fit
 - But sections are irregularly sized

• How many bits are used for the segment?

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x1C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x1200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

- How many bits are used for the segment?
- Three bits (8 choices)
- Placed as most significant bits

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x1C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x1200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

• Lower 13 bits are added to base

• Translate the following

- Read 0x0200
- Read 0x0500
- Write 0x0410

- Read 0x4004
- Write 0x5004

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x3C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x4200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

• Translate the following

- Read 0x0200 -> 0x0200
- Read 0x0500 -> 0x0500
- Write 0x0410 -> Fault (Permission)

	Segment	Base	Bound	Permissions
	0	0x0000	0x06FF	Read/Execute
	1	0x0700	0x02FF	Read/Write
	2	0x3C00	0x01FF	Read/Write
	3	0x1800	0x01FF	Read/Execute
	4	0x4200	0x0400	Read/Execute
	5	0x0000	0x0000	None
)	6	0x0000	0x0000	None
,	7	0x0000	0x0000	None

- Read 0x4004 -> 0x3C04
- Write 0x5004 -> Fault (Bound)

Outline

Address Spaces

- Methods of address translation
 - Segmentation
 - Paging
- Paging improvements
 - Improving translation speed
 - Improving table storage size

Improving upon segmentation

- Segmentation had some good features
 - Address space does not need to be contiguous
 - Segments can grow when needed
- But irregularly-sized segments lead to fragmentation



New process doesn't fit! RAM is available, but only in fragments.

Solution to fragmentation: pages of memory

• Divide memory into small, fixed-sized pages

- Pages of virtual memory map to pages of physical memory
 - Like segments were mapped, but way more of pages than segments
- Processes and their sections can be mapped to any place in memory



Page table translates virtual addresses to physical addresses

- Use topmost bits of virtual address to select page table entry
 - One page table entry per each virtual page
- Add address at page table entry to bottommost bits
 - Actually just concatenate
- Just like segment tables, there will be a different page table for each process



Paging versus segmentation

- Every page of virtual memory maps to a page of physical memory
 - No need for a bound anymore
 - Above a bound would just be the next page
- We don't pick the number of pages, we pick page size
 - Number of pages = Size of memory / Size of Page
- Way more pages than there were segments
 - 4 kB pages with 4 GB of RAM -> ~1 million pages
 - Need to keep page table in memory rather than hardware registers
 - Hardware register points at the base of the page table







Check your understanding – virtual address translation



Do we need to translate the lower bits of a virtual address?
Check your understanding – virtual address translation



Steps to translating virtual addresses with paging



How the OS deals with memory in a paging system

- 1. How do the OS and program agree on addresses?
 - Each program can use any virtual addresses it wants
 - OS controls physical memory layout in RAM and maps the two
- 2. How does the OS move memory around without messing up programs?
 - Just update the record in the page table
 - Process doesn't know the difference
- 3. How to protect OS and process memory from other processes?
 - Ensure that virtual pages from a process never map to physical pages for another
 - But we can share physical pages for threads or shared libraries if we want!

Dealing with processes bigger than memory

- Paging allows the OS to support processes larger than RAM
 - Just leave the virtual pages unmapped
 - When a load occurs to the unmapped page, a fault triggers the OS
 - Which can then load the needed page into RAM from disk
 - (and push some other page onto disk)



OS management of processes with paging

- When loading a process
 - OS places actual memory into physical pages in RAM
 - OS creates page table for the process
 - OS decides access permissions to different pages
 - OS connects to shared libraries already in RAM
- When a context switch occurs
 - OS changes which page table is in use (%CR3 register in x86)
- When a fault occurs
 - OS decides how to handle it. (Invalid access or missing page?)

Paging evaluation

- Advantages
 - Still sparse allocation of address space and growing segments as needed
 - Still different protection for different segments
 - Only execute or write where it makes sense to
 - Still possible to do dynamic relocation and hardware still relatively simple
 - No fragmentation of main memory
 - Pages can fit anywhere they need to
 - Can load processes bigger than main memory!

Paging evaluation (continued)

- Disadvantages
 - More work on the part of the OS to set up a process
 - Only a problem if we create processes frequently
 - Page tables are slow to access
 - Page tables need to be stored in memory due to size
 - MMU only holds the base address of the page table and reads from it
 - Two loads per load!!!
 - Going to have to fix this...
 - Page tables require a lot of storage space
 - Mapping must exist for each virtual page, even if unused
 - Becomes a serious issue on 64-bit systems

Check your understanding – program memory accesses

Assume `a` starts at 0x3000 (virtual)

Ignore instruction fetches and access to `i` and `sum` (they're in registers)

Code

```
int sum = 0;
for(int i=0; i<N; i++){
   sum += a[i];
}
```

Virtual Address Physical Address Accesses Accesses

load 0x3000 load 0x3004 load 0x3008 load 0x300C

- What physical address is the page table for this process at?
- At what physical address does `a` start?

load 0x100C load 0x7000 load 0x100C load 0x7004 load 0x100C load 0x7008 load 0x100C Check your understanding – program memory accesses

Assume `a` starts at 0x3000 (virtual)

Ignore instruction fetches and access to `i` and `sum` (they're in registers)

Code

```
int sum = 0;
for(int i=0; i<N; i++){
   sum += a[i];
}
```

Virtual Address Ph Accesses Ac

load 0x3000

load 0x3004

load 0x3008

load 0x300C

Physical Address Accesses

load 0x100C load 0x7000 load 0x100C load 0x7004 load 0x100C load 0x7008 load 0x100C load 0x700C

- What physical address is the page table for this process at? **0x100C**
- At what physical address does `a` start? 0x7000

Outline

Address Spaces

- Methods of address translation
 - Segmentation
 - Paging
- Paging improvements
 - Improving translation speed
 - Improving table storage size

Caching can speed up page table access

- How do we make page table access faster?
 - How do we make memory access faster?
 - Cache it!
- Code and Stack have very high spatial locality



TLB caches page table entries

- Translation Lookaside Buffer
 - Fully-associative cache (only compulsory misses)
 - Holds a subset of the page table (VPN->PPN mapping and permissions)
- On a TLB miss, go check the real page table (done in hardware)



Address translation with TLB



Context switches with a TLB

- A process must only access its own page table entries in the TLB!
 - Otherwise, the mapping is wrong, and it accesses another process...
 - OS needs to manage the TLB

- Option 1: Flush TLB on each context switch
 - Costly to lose recently cached translations
- Option 2: Track with process each entry corresponds to
 - x86-64 Process Context Identifiers (12-bit -> 4096 different processes)
 - Extra state for the OS to manage if it has more processes than that

Software controlled TLBs

- Some RISC CPUs have a software-managed TLB
 - TLB still used for translation, but a miss causes a fault for OS to handle
 - OS looks in page table for proper entry
 - OS evicts an existing entry from TLB
 - OS inserts correct entry into TLB
 - Special instruction allows OS to write to TLB
 - Hardware is simpler and OS has control over the TLB functionality
 - Can prefetch page table entries it thinks might be important
 - Can flush entries relevant to other processes
 - TLB misses take longer to complete, however

Outline

Address Spaces

- Methods of address translation
 - Segmentation
 - Paging
- Paging improvements
 - Improving translation speed
 - Improving table storage size

Paging disadvantages

- 1. Page tables are slow to access
 - Memory access for page table before any other memory access
 - TLB can speed this up considerably for common execution
- 2. Page tables require a lot of storage space
 - Mapping must exist for each virtual page, even if unused
 - Becomes a serious issue on 64-bit systems

Why do page tables take so much storage space?

- For every virtual page, there must exist an entry in the page table
 - Even though most virtual addresses aren't used!



- 32-bit address space with 4 kB pages -> 1 million entries
 - At least 8 MB of storage
 - 64-bit address space would require 36 exabytes of page table storage...

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3
2	0	
3	0	
4	0	
5	1	7
6	0	
7	0	

• Collect groups of page table entries (call them "page table entry pages"?)

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3
2	0	
3	0	
4	0	
5	1	7
6	0	
7	0	

- Collect groups of page table entries
- Only keep groups that have valid mappings in them

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3
4	0	
5	1	7

- Collect groups of page table entries
- Only keep groups that have valid mappings in them
- Remaining groups are now separate tables

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3

Virtual Page Number	Valid?	Physical Page Number
4	0	
5	1	7

- Collect groups of page table entries
- Only keep groups that have valid mappings in them
- Remaining groups are now separate tables
- Create a directory of page tables to collect existing page tables

Virtual Page Number Range	Valid?	Page Table Address	
0-1	1		
2-3	0		
4-5	1		
6-7	0		

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3

Virtual Page Number	Valid?	Physical Page Number
4	0	
5	1	7

Multilevel page tables

Multilevel page table logistics

- Virtual address is broken down into three or more parts
 - Highest bits index into highest-level page table
- A missing entry at any level triggers a page fault

- Size of tables in memory approximately equal to number of pages of virtual memory used
 - Small processes can have proportionally small page tables

Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Multilevel page tables can keep nesting

- Even page table directory is often sparse, so break it up too
- x86-64
 - Four levels of page table
 - 48-bit addresses (256 TB RAM ought to be enough for everyone right?)

Intel Ice Lake (2019): 5 layers!!

Figure 2-1. Linear-Address Translation Using 5-Level Paging

Check your understanding – multilevel page table

• How many loads per load are there now?

Figure 2-1. Linear-Address Translation Using 5-Level Paging

Check your understanding – multilevel page table

- How many loads per load are there now?
 - 6
 - As in each memory access takes six times as long
- TLB is *extremely* important

Figure 2-1. Linear-Address Translation Using 5-Level Paging

Additional optimization: large pages

- Always using large pages results in wasted memory
 - Example: 1 MB page where only 1 KB is used
- Always using small pages results in unnecessary page table entries
 - Example: 250 entries in a row to represent 1 MB of memory
- Can we mix in larger pages opportunistically?
 - Small pages normally
 - Large pages occasionally
 - Huge pages rarely

x86-64 allows multiple-sized pages: 4 KB

• Normal x86-64 paging

Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

x86-64 allows multiple-sized pages: 2 MB

- Page Size bit triggers walk to skip next table and go straight to 2 MB page in memory
- Remaining address bits are used as offset into larger page

Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

x86-64 allows multiple-sized pages: 1 GB

Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

- Can also skip straight to 1 GB pages
- With a bit of extra hardware, TLB can hold large page entries
 - Occupies a single TLB entry for 1 GB of data (250000 normal entries)

Other data structures for paging

- If hardware handles TLB misses
 - Need a regular structure it can "walk" to find page table entry
 - x86-64 needs to use multilevel page tables
- If software handles TLB misses
 - OS can use whatever data structure it pleases
 - Example: inverted page tables
 - Only store entries for virtual pages with valid physical mappings
 - Use hash of VPN+PCID to find the entry you need

Outline

Address Spaces

- Methods of address translation
 - Segmentation
 - Paging
- Paging improvements
 - Improving translation speed
 - Improving table storage size