CS 31: Intro to Systems
Virtual Memory

Martin Gagne
Swarthmore College
April 6, 2017
Recap

• We can’t load processes into memory on every context switch like we do with CPU state.

• Finding a place to put processes when we allocate memory for them is not simple.
  – Solution: Divide process memory into pieces

• The compiler doesn’t know where, in physical memory, the process will reside.
  – Solution: Make the process think, via a virtual addressing abstraction, that it has a private address space. Translate virtual addresses to physical addresses on access.
These challenges are why we need virtual memory.

Getting the benefits of solving these problems justifies the complexity of virtual memory.

• Finding a place to put processes when we allocate memory for them is not simple.
  – Solution: Divide process memory into pieces

• The compiler doesn’t know where, in physical memory, the process will reside.
  – Solution: Make the process think, via a virtual addressing abstraction, that it has a private address space. Translate virtual addresses to physical addresses on access.
Recall from Tuesday

We support virtual addressing by translating addresses at runtime.

It’s hard achieve this using base and bound registers unless each process’s memory is all in one block.

Having process in one block leads to fragmentation
Recall from Tuesday

Our solution to fragmentation is to split up a process’s address space into smaller chunks.
Paging Vocabulary

• For each process, the **virtual** address space is divided into fixed-size **pages**.

• For the system, the **physical** memory is divided into fixed-size **frames**.

• The size of a page is equal to that of a frame.
  – Often 4 KB in practice.
Main Idea

• ANY virtual page can be stored in any available frame.
  – Makes finding an appropriately-sized memory gap very easy – they’re all the same size.

• For each process, OS keeps a table mapping each virtual page to physical frame.
Main Idea

- ANY virtual page can be stored in any available frame.
  - Makes finding an appropriately-sized memory gap very easy – they’re all the same size.

Implications for fragmentation?

- External: goes away. No more awkwardly-sized, unusable gaps.
- Internal: About the same. Process can always request memory and not use it.
Addressing

• Like we did with caching, we’re going to chop up memory addresses into partitions.

• Virtual addresses:
  – High-order bits: page #
  – Low-order bits: offset within the page

• Physical addresses:
  – High-order bits: frame #
  – Low-order bits: offset within the frame
Example: 32-bit virtual addresses

• Suppose we have 8-KB (8192-byte) pages.
• We need enough bits to individually address each byte in the page.
  – How many bits do we need to address 8192 items?
Example: 32-bit virtual addresses

- Suppose we have 8-KB (8192-byte) pages.
- We need enough bits to individually address each byte in the page.
  - How many bits do we need to address 8192 items?
  - \(2^{13} = 8192\), so we need 13 bits.
  - Lowest 13 bits: offset within page.
Example: 32-bit virtual addresses

- Suppose we have 8-KB (8192-byte) pages.
- We need enough bits to individually address each byte in the page.
  - How many bits do we need to address 8192 items?
  - \(2^{13} = 8192\), so we need 13 bits.
  - Lowest 13 bits: offset within page.
- Remaining 19 bits: page number.
Example: 32-bit virtual addresses

Suppose we have 8-KB (8192-byte) pages.
We need enough bits to individually address each byte in the page.
  - How many bits do we need to address 8192 items?
    - $2^{13} = 8192$, so we need 13 bits.
    - Lowest 13 bits: offset within page.
Remaining 19 bits: page number.
Address Partitioning

Virtual address:
We’ll call these bits $p$.

Physical address:
We’ll call these bits $i$.

Once we’ve found the frame, which byte(s) do we want to access?

We’ll (still) call these bits $i$. 
Address Partitioning

Virtual address:

- We’ll call these bits $p$. [Green bits]
- We’ll call these bits $i$. [Blue bits]

OS Page Table For Process

- Where is this page in physical memory? (In which frame?)

Physical address:

- We’ll call these bits $f$. [Red bits]
- We’ll (still) call these bits $i$. [Blue bits]

Once we’ve found the frame, which byte(s) do we want to access?
Address Translation

Logical Address

Page $p$  Offset $i$

Page Table

V R D Frame Perm ...

Physical Memory
Address Translation

Logical Address

Page $p$  Offset $i$

Page Table

V R D Frame Perm ...

Physical Memory
Address Translation

Logical Address

Page $p$  Offset $i$

Page Table

V | R | D | Frame | Perm | ...

Physical Address

Physical Memory
Page Table

- One table per process
- Table entry elements
  - V: valid bit
  - R: referenced bit
  - D: dirty bit
  - Frame: location in phy mem
  - Perm: access permissions
- Table parameters in memory
  - Page table base register
  - Page table size register
Address Translation

• Physical address = frame of $p$ + offset $i$
• First, do a series of checks
Check if Page $p$ is Within Range

Logical Address

Page $p$ | Offset $i$

PTBR
PTSR

$p < PTSR$

V | R | D | Frame | Perm | ...

Physical Address
Check if Page Table Entry $p$ is Valid

Logical Address

Page $p$ | Offset $i$

PTBR
PTSR

$V == 1$

Physical Address
Check if Operation is Permitted
Translate Address

Logical Address

Page $p$    Offset $i$

PTBR
PTSR

V | R | D | Frame | Perm | ...

concat

Physical Address
Physical Address by Concatenation

Logical Address

Page $p$  Offset $i$

PTBR  PTSR

V R D Frame Perm ...

Physical Address

Frame $f$  Offset $i$
Sizing the Page Table

Number of bits $n$ specifies max size of table, where number of entries $= 2^n$

Number of bits needed to address physical memory in units of frames

Number of bits specifies page size
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
How many entries (rows) will there be in this page table?

A. $2^{12}$, because that’s how many the offset field can address
B. $2^{20}$, because that’s how many the page field can address
C. $2^{30}$, because that’s how many we need to address 1 GB
D. $2^{32}$, because that’s the size of the entire address space
Example of Sizing the Page Table

<table>
<thead>
<tr>
<th>Page $p$: 20 bits</th>
<th>Offset $i$: 12 bits</th>
</tr>
</thead>
</table>

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

20 bits to address $2^{20} = 1$ M entries

How big is a frame?
What will be the frame size, in bytes?

A. $2^{12}$, because that’s how many bytes the offset field can address
B. $2^{20}$, because that’s how many bytes the page field can address
C. $2^{30}$, because that’s how many bytes we need to address 1 GB
D. $2^{32}$, because that’s the size of the entire address space
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

Page $p$: 20 bits  
Offset $i$: 12 bits

- 20 bits to address $2^{20} = 1$ M entries
- Page size = frame size = $2^{12} = 4096$ bytes

How many bits in frame nb?
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
How big is an entry, in bytes? 
(Round to a power of two bytes.)

Page $p$: 20 bits
Offset $i$: 12 bits

20 bits to address $2^{20} = 1$ M entries

$2^{30} / 2^{12} = 2^{18}$ so 18 bits

Page size = frame size = $2^{12} = 4096$ bytes

Size of an entry?

• Given: 32 bit virtual addresses, 1 GB physical memory
  – Address partition: 20 bit page number, 12 bit offset
• A: 1    B: 2    C: 4    D: 8
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

### Page $p$: 20 bits

- 20 bits to address $2^{20} = 1$ M entries

### Offset $i$: 12 bits

- $2^{30} / 2^{12} = 2^{18}$ so 18 bits

### Page size = frame size = $2^{12} = 4096$ bytes

### Total size of page table?

- 24 (1+1+1+18+3+...) bits so 4 bytes needed
Example of Sizing the Page Table

- **Page p**: 20 bits
- **Offset i**: 12 bits

20 bits to address $2^{20} = 1$ M entries

$2^{30} / 2^{12} = 2^{18}$ so 18 bits

24 (1+1+1+18+3+...) bits so 4 bytes needed

Page size = frame size = $2^{12} = 4096$ bytes

Table size = $1$ M x 4 = 4 MB

- **4 MB of bookkeeping for every process?**
  - 300 processes -> 1.2 GB just to store page tables...
Concerns

• We’re going to need a ton of memory just for page tables…

• Wait, if we need to do a lookup in our page table, which is in memory, every time a process accesses memory…
  – Isn’t that slowing down memory by a factor of 2?
Multi-Level Page Tables

(You’re not responsible for this. Take an OS class for the details.)

Logical Address

1st-level Page \(d\)  \(\text{2nd-level Page } p\)  Offset \(i\)

Points to (base) frame containing 2nd-level page table

Reduces memory usage SIGNIFICANTLY: only allocate page table space when we need it. More memory accesses
Cost of Translation

• Each lookup costs another memory reference
  – For each reference, additional references required
  – Slows machine down by factor of 2 or more

• Take advantage of locality
  – Most references are to a small number of pages
  – Keep translations of these in high-speed memory
    (a cache for page translation)
TLB: Translation Look-aside Buffer

- Fast memory keeps most recent translations
  - Fully associative hardware lookup
- If page matches, get frame number
  else wait for normal translation (in parallel)
TLB Design Issues

• The larger the TLB
  – the higher the hit rate
  – the slower the response
  – the greater the expense
  – the larger the space on-chip

• TLB has a major effect on performance!
  – Must be flushed on context switches
  – Alternative: tagging entries with PIDs
Problem Summary: Addressing

- General solution: OS must translate process’s VAS accesses to the corresponding physical memory location.

When the process tries to access a virtual address, the OS translates it to the corresponding physical address.

OS must keep a table, for each process, to map VAS to PAS. One entry per divided region.
Problem: Storage

• Where should process memories be placed?
  – Topic: “Classic” memory management

• How does the compiler model memory?
  – Topic: Logical memory model

• How to deal with limited physical memory?
  – Topics: Virtual memory, paging
Recall “Storage Problem”

• We must keep multiple processes in memory, but how many?
  – Lots of processes: they must be small
  – Big processes: can only fit a few

• How do we balance this tradeoff?

Locality to the rescue!
VM Implications

• Not all pieces need to be in memory
  – Need only piece being referenced
  – Other pieces can be on disk
  – Bring pieces in only when needed
• Illusion: there is much more memory
• What’s needed to support this idea?
  – A way to identify whether a piece is in memory
  – A way to bring in pieces (from where, to where?)
  – Relocation (which we have)
Virtual Memory based on Paging

- Before
  - All virtual pages were in physical memory
Virtual Memory based on Paging

• Now
  – All virtual pages reside on disk
  – Some also reside in physical memory (which ones?)
• Ever been asked about a swap partition on Linux?
Sample Contents of Page Table Entry

<table>
<thead>
<tr>
<th>Valid</th>
<th>Ref</th>
<th>Dirty</th>
<th>Frame number</th>
<th>Prot: rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Valid: is entry valid (page in physical memory)?
• Ref: has this page been referenced yet?
• Dirty: has this page been modified?
• Frame: what frame is this page in?
• Protection: what are the allowable operations?
  – read/write/execute
A page fault occurs. What must we do in response?

A. Find the faulting page on disk.
B. Evict a page from memory and write it to disk.
C. Bring in the faulting page and retry the operation.
D. Two of the above
E. All of the above
Address Translation and Page Faults

• Get entry: index page table with page number
• If valid bit is off, page fault
  – Trap into operating system
  – Find page on disk (kept in kernel data structure)
  – Read it into a free frame
    • may need to make room: page replacement
  – Record frame number in page table entry, set valid
  – Retry instruction (return from page-fault trap)
Page Faults are Expensive

• Disk: many orders magnitude slower than RAM
  – Very expensive; but if very rare, tolerable
• Example
  – RAM access time: 50 nsec
  – Disk access time: 10 msec
  – $p = \text{page fault probability}$
  – Effective access time: $50 + p \times 10,000,000 \text{ nsec}$
  – If $p = 0.1\%$, effective access time = $10,050 \text{ nsec}$ !
Handing faults from disk seems very expensive. How can we get away with this in practice?

A. We have lots of memory, and it isn’t usually full.

B. We use special hardware to speed things up.

C. We tend to use the same pages over and over.

D. This is too expensive to do in practice!
Principle of Locality

• Not all pieces referenced uniformly over time
  – Make sure most referenced pieces in memory
  – If not, thrashing: constant fetching of pieces

• References cluster in time/space
  – Will be to same or neighboring areas
  – Allows prediction based on past
Page Replacement

• Goal: remove page(s) not exhibiting locality

• Page replacement is about
  – which page(s) to remove
  – when to remove them

• How to do it in the cheapest way possible
  – Least amount of additional hardware
  – Least amount of software overhead
Basic Page Replacement Algorithms

• FIFO: select page that is oldest
  – Simple: use frame ordering
  – Doesn’t perform very well (oldest may be popular)
• OPT: select page to be used furthest in future
  – Optimal, but requires future knowledge
  – Establishes best case, good for comparisons
• LRU: select page that was least recently used
  – Predict future based on past; works given locality
  – Costly: time-stamp pages each access, find least
• Goal: minimize replacements (maximize locality)
Up Next

Multi-threading