Virtual Memory: Mechanisms

CS439: Principles of Computer Systems
March 6, 2019
Bringing It Together

• Processes are the OS’s main abstraction for protection
• A process defines an address space---but this is a virtual/logical address space
• Virtual address spaces are mapped onto physical address spaces in physical memory
• All virtual addresses range from 0->max for that process
• Relocation techniques place a process contiguously in physical memory
  – Virtual addresses must be translated to physical addresses-
  – Static relocation does this at load time
  – Dynamic does this at execution time
• The *-fit algorithms are used to choose *where* to allocate a process’s address space in physical memory
  – All had problems with external fragmentation
• Compaction reduces external fragmentation
• Swapping allows more processes to run than would otherwise fit in memory
Today’s Agenda

• Add one more problem...
  – Single process too big to fit into memory
  – Solution: overlays

• Solve the problems we had last time!
  – Eliminate external fragmentation
  – Allow more processes to execute at once
    • A higher degree of multiprogramming!
  – Enable sharing

• Mechanisms to support our solution
  – Address Translation
  – Page Tables
Yet Another Memory Problem...

• Problem: Processes that are each too large to fit into memory

• First Solution: Overlays
  – Programmer manually divided the program into pieces, or overlays
  – Overlay manager swaps overlays in and out during execution
    – Only necessary pieces are in memory at a given time

• But wait...
Virtual Memory: Today’s Version

• Process uses a virtual address space
  – May be much larger than available physical memory

• Only portions of the virtual address space are in physical memory at any one time
  – Virtual address space is automatically divided into equal pieces
  – Needed pieces are automatically brought into the memory (from where?)

• This technique is known as paging
Paging

1. Divides a process’s virtual address space into fixed-sized *pages*
2. Stores a copy of that address space on disk
3. Views the physical memory as a series of equal-sized *page frames* (or *frames*)
4. Moves the pages into frames in memory
   – *when* they are moved is a policy decision
5. Manages the pages in memory
   – Is a page still in use? is it written? ...
Virtual Memory: Pages

A process’s virtual address space is partitioned into equal sized *pages*.

A virtual address is a pair \((p, o)\)

- \(p\) — page number \((p_{\text{max}}\) pages\)
- \(o\) — page offset \((o_{\text{max}}\) bytes/pages\)

Virtual address = \(o_{\text{max}} \times p + o\)

\[2^{n-1} = (p_{\text{MAX}}-1, o_{\text{MAX}}-1)\]
Page Frames

Physical memory is partitioned into equal sized page frames.

\[ |\text{page}| = |\text{page frame}| \]

The system’s page size is equivalent to its frame size.

A memory address is a pair \((f, o)\)
- \(f\) — frame number \((f_{\text{max}}\) frames)
- \(o\) — frame offset \((o_{\text{max}}\) bytes/frames)

Physical address = \(o_{\text{max}} \times f + o\)
Physical Addresses (Frame/Offset Pairs)

Example: A 16-bit address space with $(o_{max} =) 512$-byte page frames

Addressing location $(3, 6) = 1,542$

PA:

```
0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0
```

1,542

Physical Memory

$(0, 0) \rightarrow 1,542$
iClicker Question

Paging has what advantage over relocation?

A. Easier to manage transfer of pages from disk
B. Requires less hardware support
C. No external fragmentation
From Virtual to Physical: Allocation Policy

- Pages map to frames
- Pages are contiguous in a VAS...
  - But pages are arbitrarily located in physical memory, and
  - Not all pages mapped at all times
    - Where are the others?
Problem: How do we find addresses when pages are not allocated contiguously in memory?

Solution: A page table keeps track of the mapping of pages to page frames

- You can think of the page table as a set of relocation registers, one for each frame
- Mapping is invisible to the process
- Protection is provided with the same mechanisms as used in dynamic relocation
A *page table* maps virtual pages to physical frames.
So... How are we doing?

Wanted:

– reduce or eliminate external fragmentation
– the ability to grow processes easily
– to allow processes to use more memory than that which is physically available
– easy to allocate and de-allocate memory to/from processes
– ability to easily share memory between processes
– protection
Advantages of Paging: Sharing

• Paging allows shared memory
  – the memory used by a process no longer needs to be contiguous

• A shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address
Virtual Address Translation: Protection and Consistency

1 table per process
Part of process’s state

Page Table Entry Contents:
- Flags: dirty bit, resident bit, clock/reference bit
- Frame number

Page Table:

Virtual Addresses

Physical Addresses
A system with 16-bit addresses
- 32 KB of physical memory
- 1024 byte pages

Virtual Address Space

Page Table

Physical Memory

Physical Addresses

Virtual Addresses

CPU
Performance Issues: Speed

Problem: Virtual memory references require two memory references!
  – One access to get the page table entry
  – One access to get the data
Back to Architecture

Translation
Lookaside
Buffer/Memory
Management Unit
Improving Speed:
Translation Lookaside Buffers (TLBs)

Cache recently accessed page-to-frame translations in a TLB
- For TLB hit, physical page number obtained in 1 cycle
- For TLB miss, translation is updated in TLB
- Has high hit ratio (why?)
Performance Issues: Space

• Two sources:
  – data structure overhead (the page table)
  – fragmentation (*How large should a page be?*)

• Page table can be very large!
  – 32-bit addresses, 4096-byte pages... 1 million pages!
    • 32-bit addresses imply a VAS of $2^{32}$ bytes
    • Divide VAS into pages ($2^{32}/2^{12}$)
    • 1 million pages ($2^{20}$)
Dealing With Large Page Tables: Multi-Level Paging

Add additional levels of indirection to the page table by sub-dividing page number into $k$ parts
- Create a “tree” of page tables
- TLB still used, just not shown
- The architecture determines the number of levels of page table

Virtual Address

First-Level Page Table

Second-Level Page Tables

Third-Level Page Tables
Multi-Level Paging: Example

Example: Two-level paging

- CPU
- Physical Addresses
- Virtual Addresses
- Memory
- First-Level Page Table
- Second-Level Page Table
- PTBR
64-bit Addresses

• Multi-level page tables work decently well for 32-bit address spaces... but what happens when we get to 64-bit?
• Too cumbersome (5 levels?)
• Rethink the page table:
  – Instead of making tables proportional to the size of the VAS, make them proportional to the size of the physical address space
  – One entry for each physical page---and hash!

Result: Inverted Page tables
Inverted Page Tables

• A table where each entry represents a frame
  – As opposed to representing a page, like in a page table

• Entry contains:
  – Residence bit: is the frame occupied?
  – Occupier: page number of the page in the frame
  – Protection bits
Inverted Page Tables: Virtual Address Translation

• As the CPU generates virtual addresses, where is the physical page?
  – Must search entire page table
    • 4KB of entries for 16MB RAM

• How can we solve that problem?
  – Cache it! (How did you know?)
  – Use the TLB to hold all the heavily used pages
  – But, um... the TLB is limited in size
  – So what about a miss in the TLB?
Use a Hash Table!

• Page $i$ is placed in slot $f(i)$ where $f$ is an agreed-upon hash function
  – Deal with collisions using linked list or rehashing

• To look up page $i$, perform the following:
  – Compute $f(i)$ and use it as an index into the table of page entries
  – Extract the corresponding page entry
Hashed Inverted Page Table

Hash page numbers to find corresponding frame number
– Page frame number is not explicitly stored (1 frame per entry)
– Protection, dirty, used, resident bits also in entry

$h(PID, p)$
iClicker Question

Why use hashed/inverted page tables?

A. Regular (forward-mapped) page tables are too slow
B. Forward-mapped page tables don’t scale to larger virtual address spaces
C. Inverted page tables have a simpler lookup algorithm, so the hardware that implements them is simpler
D. Inverted page tables allow a virtual page to be anywhere in physical memory
The BIG picture

CPU \[\text{vaddr}\] \rightarrow \text{Translator Box} \[\text{paddr}\] \rightarrow \text{Physical memory}
The BIG picture

CPU

Virtually addressed cache

Segment and page table lookup

PTBR (per process)

vpage

Main memory

Vaddr
data

if match

if no match

vpage

ppage

if match

if no match

Paddr
data

if match

if no match

dictionary

Physically addressed cache

TLB

if match

if no match
Virtual Memory: The Bigger Picture

- A process’s VAS contains its code, data, and stack
- Code pages are stored in a user’s file on disk
  - Some are currently residing in memory; most are not
- Data and stack pages are also stored in a file
  - Although this file is typically not visible to users
  - File only exists while a program is executing
  - Located on swap
- OS determines which portions of a process’s VAS are mapped in memory at any one time
Page Faults

References to non-mapped pages generate a page fault

Page fault handling steps:
1. Processor runs the interrupt handler
2. Page fault handler determines the reason for the page fault (exception or not resident)
3. If exception, process is killed
4. If not resident, it is a true page fault
5. Handler locates unmapped page and initiates the read of that page
6. Faulting process is blocked and another process is scheduled
7. Read of page completes, page is mapped into physical memory
8. Faulting process is unblocked
Summary

Paging is a commonly used virtual memory mechanism

– Virtual address space is treated as pages
– Physical address space is treated as page frames
– Address translation is necessary
– Page tables are used: TLBs speed them up
  • Multilevel and Inverted
Announcements

• Discussion section Friday! Problem Set 5 is posted.

• Project 2 stack check due Monday
  – Sign up schedule will be posted (watch Piazza or Project page)
  – Be certain to follow instructions and provide all information (CS logins! NOT EIDs!)

• Canvas group sign ups must be completed by tonight