CS 370: OPERATING SYSTEMS

MEMORY MANAGEMENT

Shrideep Pallickara
Computer Science
Colorado State University

October 30, 2018

Frequently asked questions from the previous class survey

- Compaction
  - What if there is no space for a process after compaction?
- Are segments placed in logical or physical memory?
- Is it common to have 32-bit addresses vs 64-bit?

Topics covered in this lecture

- Paging
- Translation look-aside buffers (TLB)
- Memory Protection in paged environments
- Shared Pages
- Page sizes

The Paging memory management scheme

- Physical address space of process can be non-contiguous
- Solves problem of fitting variable-sized memory chunks to backing store
  - Backing store has fragmentation problem
  - Compaction is impossible

Basic method for implementing paging

- Break memory into fixed-sized blocks
  - Physical memory: frames
  - Logical memory: pages
  - Same size
- Backing store is also divided the same way

What will seem odd, and perhaps cool, about paging

- While a program thinks of its memory as linear ...
  - It is usually scattered throughout physical memory in a kind of abstract mosaic
- The processor will execute one instruction after another using virtual addresses
  - The virtual addresses are still linear
  - However, an instruction located at the end of one page will be located in a completely different region of physical memory from the next instruction at start of another page
What will seem odd, and perhaps cool, about paging [2/2]
- Data structures appear to be contiguous using virtual addresses
- But a large matrix is scattered across many physical page frames

Paging: Analogy
- Shuffling several decks of cards together
- A single process in its virtual address page sees the cards of a single deck in order
  - A different process sees a completely different deck, but it will also be in order
- In physical memory, however, the decks of all processes currently running will be shuffled together, apparently at random
- Page tables are the magician's assistant in locating cards from the shuffled decks

Paging: Logical and Physical Memory

Paging Hardware: Performing address translation

Page size
- A power of 2
  - Typical sizes: 512 bytes – 16 MB
- Size of logical address: $2^m$
- Page size: $2^n$

Paging and Fragmentation
- No external fragmentation
  - Free frame available for allocation to other processes
- Internal fragmentation possible
  - Last frame may not be full
  - If process size is independent of page size
    - Internal fragmentation = $\frac{1}{2}$ page per process
Page sizes
- Processes, data sets, and memory have all grown over time
- Page sizes have also increased
- Some CPUs/kernels support multiple page sizes

Paging: User program views memory as a single space
- Program is scattered throughout physical memory
- User view and physical memory reconciled by
  - Address-translation hardware
  - Process has no way of addressing memory outside of its page table

OS manages the physical memory
- Maintains frame-table; one entry per frame
  - Free or allocated?
  - If allocated: Which page of which process
- Maintains a page table for each process
  - Used by CPU dispatcher to define hardware page table when process is CPU-bound
  - Paging increases context switching time

Example: 32-bit address space
- Page size = 4K
- Logical address = 0x23FA427
- What's the offset within the page?
  - 0x427
- What's the page number?
  - 0x23FA
- Page table entry maps 0x23FA to frame 0x12345, what is the physical memory address for the logical address?
  - 0x12345427

Example: 32-bit address space
- Page size = 1K
- Logical address = 0x23FA427
- What's the offset within the page?
  - 0x427
- What's the page number?
  - 0x23FA

Hardware support for Paging
- All accesses to memory must go through a map. Efficiency is important.
The purpose of the page table is to map virtual pages onto physical frames

- Think of the page table as a function
  - Takes virtual page number as an argument
  - Produces physical frame number as result
- Virtual page field in virtual address replaced by frame field
- Physical memory address

Two major issues facing page tables

- Can be extremely large
  - With a 4 KB page size, a 32-bit address space has 1 million pages
  - Also, each process has its own page table
- The mapping must be fast
  - Virtual-to-physical mapping must be done on every memory reference
  - Page table lookup should not be a bottleneck

Implementing the page table: Dedicated registers

- When a process is assigned the CPU, the dispatcher reloads these registers
- Feasible if the page table is small
  - However, for most contemporary systems entries are greater than $10^6$

Implementing the page table in memory

- Page table base register (PTBR) points to page table
  - 2 memory accesses for each access
    - One for the page-table entry
    - One for the byte

Process and its page table: When is the page table entirely in memory?

- A pointer to the page table is stored in the page table base register (PTBR) in the PCB
  - Similar to the program counter
- Often there is also a register which tracks the number of entries in the page table
- Page table need not be memory resident when the process is swapped out
  - But must be in memory when process is running

Cash is king. — Pehr Gyllenhammar

**Translation Look-aside Buffers**
Observation

- Most programs make a large number of references to a small number of pages
- Not the other way around
- Only a small fraction of the page table entries are heavily read
- Others are barely used at all

Translation look-aside buffer (TLB): Small, fast-lookup hardware cache

- Number of TLB entries is small (64 ~ 1024)
- Contains few page-table entries
- Each entry of the TLB consists of 2 parts
  - A key and a value
- When the associative memory is presented with an item
  - Item is compared with all keys simultaneously

Using the TLB with page tables [1/2]

- TLB contains only a few page table entries
- When a logical address is generated by the CPU, the page number is presented to the TLB
  - When frame number is found (TLB hit), use it to access memory
  - Usually just 10-20% longer than an unmapped memory reference

Using the TLB with page tables [2/2]

- What if there is a TLB miss?
  - Memory reference to page table is made
  - Replacement policies for the TLB entries
  - Some TLBs allow certain entries to be wired down
    - TLB entries for kernel code are wired down

Paging Hardware with a TLB

TLB and Address Space Identifiers (ASIDs)

- ASID uniquely identifies each process
  - Allows TLB to contain addresses from several different processes simultaneously
- When resolving page numbers
  - TLB ensures that ASIDs match
  - If not, it is treated as a TLB miss
Without ASIDs TLB must be flushed with every context switch

- Each process has its own page table
- Without flushing or ASIDs, TLB could include old entries
  - Valid virtual addresses
  - But incorrect or invalid physical addresses from previous process

Effective memory access times

- 20 ns to search TLB
- 100 ns to access memory
- If page is in TLB, access time = 20 + 100 = 120 ns
- If page is not in TLB:
  \[ 20 + 100 + 100 = 220 \text{ ns} \]

Access TLB
Access memory to retrieve frame number

Effective access times with different hit ratios

- 80%:
  \[ 0.80 \times 120 + 0.20 \times 220 = 140 \text{ ns} \]
- 98%:
  \[ 0.98 \times 120 + 0.02 \times 220 = 122 \text{ ns} \]
- When hit rate increases from 80% to 98%
  - Results in ~13% reduction in access time

TLB in practical settings

- Hit time: 0.5 – 1 clock cycle
- Miss penalty: 10 – 100 clock cycles
- Miss rate: 0.01 - 1%

Memory Protection in Paged Environments

Protection bits are associated with each frame

- Kept in the page table
- Bits can indicate
  - Read-write, read-only, execute
  - Illegal accesses can be trapped by the OS
- Valid-invalid bit
  - Indicates if page is in the process’s logical address space
Protection Bits: Page size = 2K; Logical address space = 16K

Program restricted to 0 - 10468

Page Table

Physical Memory

Reentrant Code

- A computer program or subroutine is called reentrant if:
  - It can be interrupted in the middle of its execution and
  - Then safely called again ("re-entered") before its previous invocations complete execution

Shared Pages

- System with N users
  - Each user runs a text editing program
- Text editing program
  - 150 KB of code
  - 50 KB of data space
- 40 users
  - Without sharing: 8000 KB space needed
  - With sharing: 150 + 40 x 50 = 2150 KB needed

Reentrant Code

- Non-self-modifying
  - Does not change during execution
- Two or more processes can:
  1. Execute same code at same time
  2. Will have different data
- Each process has:
  - Copy of registers and data storage to hold the data

Shared Paging

- 0
- Data 1
- Page Table
- Process P1
- ed 1
- ed 2
- ed 3
- Data 1
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- Page n

Physical Memory

Process P2
- ed 1
- ed 2
- ed 3
- Data 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- Page n

Process P3
- ed 1
- ed 2
- ed 3
- Data 3
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- Page n
Shared Paging

- Other heavily used programs can be shared
  - Compilers, runtime libraries, database systems, etc.
- To be shareable:
  1. Code must be reentrant
  2. The OS must enforce read-only nature of the shared code

Paging and page sizes

- On average, \( \frac{1}{2} \) of the final page is empty
  - Internal fragmentation: wasted space
- With \( n \) processes in memory, and a page size \( p \)
  - Total \( np/2 \) bytes of internal fragmentation
  - Greater page size \( \Rightarrow \) Greater fragmentation

Transfers to-and-from disk are a page at a time

- Primary Overheads: Seek and rotational delays
- Transferring a small page almost as expensive as transferring a big page
  - \( 64 \times 15 = 960 \) msec to load 64 512-byte pages
  - \( 4 \times 25 = 100 \) msec to load 4 8KB pages
  - Here, large pages make sense

But having small pages is not necessarily efficient

- Small pages mean programs need more pages
  - Larger page tables
  - 32KB program needs 4 8KB pages, but 64 512-byte pages
- Context switches can be more expensive with small pages
  - Need to reload the page table

Overheads in paging:
Page table and internal fragmentation

- Average process size \( = s \)
- Page size \( = p \)
- Size of each page entry \( = e \)
- Pages per process \( = s/p \)
  - \( s/e \): Total page table space
- Total Overhead \( = s/e + p/2 \)

PAGE SIZES

- Other heavily used programs can be shared
  - Compilers, runtime libraries, database systems, etc.
- To be shareable:
  1. Code must be reentrant
  2. The OS must enforce read-only nature of the shared code
Looking at the overhead a little closer

- Total Overhead = se/p + p/2
  - Increases if p is small
  - Increases if p is large
- Optimum is somewhere in between
- First derivative with respect to p
  \[-se p^2 + \frac{1}{2} = 0\]
  i.e. \[p^2 = 2se\]
  \[p = \sqrt{2se}\]

---

Optimal page size: Considering only page size and internal fragmentation

- \[p = \sqrt{2se}\]
- \[s = 128KB\] and \(e = 8\) bytes per entry
- Optimal page size = 1448 bytes
  - In practice we will never use 1448 bytes
  - Instead, either 1K or 2K would be used
  - Why? Pages sizes are in powers of 2 i.e. \(2^x\)
  - Deriving offsets and page numbers is also easier

---

Pages sizes and size of physical memory

- As physical memories get bigger, page sizes get larger as well
  - Though not linearly
- Quadrupling physical memory size rarely even doubles page size

---

The contents of this slide-set are based on the following references