Topics covered in this lecture

- Paging
- Hardware support for paging

Contiguous Memory Allocation: Fragmentation

- As processes are loaded/removed from memory
  - Free memory space is broken into small pieces
- External fragmentation
  - Enough space to satisfy request; BUT
  - Available spaces are not contiguous

Fragmentation: Example

Process P₅ cannot be loaded because memory space is fragmented

Fragmentation can be internal as well

- Memory allocated to process may be slightly larger than requested
- Internal fragmentation
  - Unused memory is internal to blocks
Compaction: Solution to external fragmentation

- **Shuffle** memory contents
  - Objective: Place free memory into large block
- Not possible if relocation is static
  - Load time
- Approach involves moving:
  1. Processes towards one end
  2. Gaps towards the other end

Compaction: Example

Memory compaction is time intensive and is usually not done

- Let's consider a machine with 1 GB of RAM
- The machine can copy 4 bytes in 20 nanoseconds
- Time to compact all the memory?
  \[ 10^9 \times \left(\frac{20 \times 10^{-9}}{4}\right) = 5 \text{ seconds (approximately)} \]
  
  Note: 1 GB is approximately \(10^9\) bytes.

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

PAGING

Noncontiguous memory management
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 a page will be located in a completely different region of physical memory from the next instruction at start of another page

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: $2^n$

Logical address

Page number
Page offset
$0 \ldots 0$
$1 \ldots 1$
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 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?
  - 0x100010 0111
- What's the page number?
  - 0000 0010 0011 1111 1010 01
All accesses to memory must go through a map. Efficiency is important.

**HARDWARE SUPPORT FOR PAGING**

October 26, 2017

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

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 lookaside buffer
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)

- 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, use to access memory
  - Usually just 10-20% longer than an unmapped memory reference

Using the TLB with page tables (2)

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

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

PAGE SIZES
Paging and page sizes

- On average, 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 = Greater fragmentation

**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

**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 x 15 = 960 msec to load 64 512-bytes pages
  - 4 x 25 = 100 msec to load 4 8KB pages
- Here, large pages make sense

**Overheads in paging: Page table and internal fragmentation**

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

\[ p = \sqrt{2se} \]

\[ p = \sqrt{2se} \]

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