CS 370: Operating Systems

[Memory Management]

Computer Science
Colorado State University

** Lecture slides created by: SHRIDEEP PALICKARA

Instructor: Louis-Noel Pouchet
Spring 2024
Topics covered in this lecture

- Contiguous memory allocations
- Fragmentations
  - External and Internal
- Paging
- Hardware support for paging
Splitting and Fusing Memory spaces
Dynamic Storage Allocation Problem

- Satisfying a request of size $n$ from the set of available spaces
  - First fit
  - Best fit
  - Worst fit
First fit

- Scan list of segments until you find a memory-hole that is big enough

- Hole is broken up into two pieces
  - One for the process
  - The other is unused memory
Best Fit

- Scan the entire list from beginning to the end
- Pick the smallest memory-hole that is adequate to host the process
Comparing Best Fit and First Fit

- Best fit is slower than first fit

- Surprisingly, it also results in more wasted memory than first fit
  - Tends to fill up memory with tiny, useless holes
Worst fit

- How about going to the other extreme?
  - Always take the largest available memory-hole
  - Perhaps, the new memory-hole would be useful

- Simulations have shown that worst fit is not a good idea either
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 $P_5$ 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
  - 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?
  
  \[ 10^9 \times \left(20 \times 10^{-9} / 4\right) = 5 \text{ seconds (approximately)} \]

  Note: 1 GB is approximately \(10^9\) bytes.
Summarizing the pure Swapping based approach

- Bring in each process in its *entirety* into memory

- Run process for a while before eviction due to:
  - Space being needed for another process
  - Process becomes idle
    - Idle processes should not take up space in memory
Overview of how mapping of logical and physical addresses is performed

MMU may access Physical Memory to perform translations
\{PageTable may be stored there\}
Noncontiguous memory management

PAGING
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 pages

- Break memory into **fixed-sized** blocks
  - Physical memory: **frames**
  - Logical memory: **pages**

- Backing store is also divided the same way
Paging Hardware: Paging is a form of dynamic relocation

Logical Address

Page number

Page offset

CPU

p d

Logical Address

Page Table

Physical Address

Frame f

000...000

f111...111

p
d

f

Page size

- Usually a power of 2
  - 512 bytes – 16 MB
- Size of logical address: $2^m$
- Page size: $2^n$

Page number

$m - n$

Page offset

$n$

$m$ bits

Logical address
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?
  - 01 | 00 0010 0111

- What’s the page number?
  - 0000 0010 0011 1111 1010 01
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
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
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
The contents of this slide-set are based on the following references
