CSE120
Principles of Operating Systems

Prof Yuanyuan (YY) Zhou
Memory Management
Next few lectures are going to cover memory management

- **Goals of memory management**
  - To provide a convenient abstraction for programming
  - To allocate scarce memory resources among competing processes to maximize performance with minimal overhead

- **Mechanisms**
  - Physical and virtual addressing (1)
  - Techniques: partitioning, paging, segmentation (1)
  - Page table management, TLBs, VM tricks (2)

- **Policies**
  - Page replacement algorithms (3)
Lecture Overview

- Virtual memory warm-and-fuzzy
- Survey techniques for implementing virtual memory
  - Fixed and variable partitioning
  - Paging
  - Segmentation
- Focus on hardware support and lookup procedure
  - Next lecture we’ll go into sharing, protection, efficient implementations, and other VM tricks and features
Virtual Memory

- The abstraction that the OS will provide for managing memory is virtual memory (VM)
  - Virtual memory enables a program to execute with less than its complete data in physical memory
    - A program can run on a machine with less memory than it “needs”
    - Can also run on a machine with “too much” physical memory
  - Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
  - OS will adjust amount of memory allocated to a process based upon its behavior
  - VM requires hardware support and OS management algorithms to pull it off
- Let’s go back to the beginning…
In the beginning (1960’s)...

- Rewind to the days of “second-generation” computers
  - Programs use physical addresses directly
  - OS loads job, runs it, unloads it

- Multiprogramming changes all of this
  - Want multiple processes in memory at once
    - Overlap I/O and CPU of multiple jobs
  - Can do it a number of ways
    - Fixed and variable partitioning, paging, segmentation
  - Requirements
    - Need protection – restrict which addresses jobs can use
    - Fast translation – lookups need to be fast
    - Fast change – updating memory hardware on context switch
Virtual Addresses

- To make it easier to manage the memory of processes running in the system, we’re going to make them use **virtual addresses** (logical addresses)
  - Virtual addresses are independent of the actual physical location of the data referenced
  - OS determines location of data in physical memory
  - Virtual addresses are translated by hardware into physical addresses (with help from OS)
  - The set of virtual addresses that can be used by a process comprises its **virtual address space**
Many ways to do this translation…
- Start with old, simple ways, progress to current techniques
Analogy: iShelf

- Suppose 5 students each has 20-60 books, they want to share a “intelligent” book shelf called iShelf
  - A student only needs to tell iShelf a book #k, iShelf will return him/her the correct book
  - iShelf would never return a person other people’s book

- If iShelf can hold 300 books, you are hired to design iShelf
  - how should you manage the space?
  - How can you tell iShelf to return the right book if a student #j asks for book #k?
Old Way(1): Fixed Partitions

- Physical memory is broken up into fixed partitions
  - Hardware requirements: base register
  - Physical address = virtual address + base register
  - Base register loaded by OS when it switches to a process
  - Size of each partition is the same and fixed
  - How do we provide protection?

- Advantages
  - Easy to implement, fast context switch
    - Why?

- Problems
  - Internal fragmentation: memory in a partition not used by a process is not available to other processes
  - Partition size: one size does not fit all (very large processes?)
Fixed Partitions

Virtual Address + Offset = P4’s Base

Base Register

Physical Memory

P1
P2
P3
P4

Internal fragmentation
Old Way (2): Variable Partitions

- Natural extension – physical memory is broken up into variable sized partitions
  - Hardware requirements: base register and limit register
  - Physical address = virtual address + base register
  - Why do we need the limit register? Protection
    - If (physical address > base + limit) then exception fault

- Advantages
  - No internal fragmentation: allocate just enough for process

- Problems
  - External fragmentation: job loading and unloading produces empty holes scattered throughout memory
Variable Partitions

External fragmentation (too small to allocate to any process)

Virtual Address

Offset

Base Register

P3’s Base

Limit Register

P3’s Limit

Yes?

No?

Protection Fault

P1

P2

P3
Variable Partitions and Fragmentation

Memory wasted by External Fragmentation

Step 1
Monitor | Job 1 | Job 2 | Job 3 | Job 4 | Free

Step 2
Monitor | Job 1 | | Job 3 | Job 4 | Free
| Job 5 |

Step 3
Monitor | Job 1 | Job 5 | Job 3 | Job 4 | Free

Step 4
Monitor | Job 5 | Job 3 | Job 4 | Job 6 |

Step 5
Monitor | Job 7 | Job 5 | Job 3 | | Job 6 |

Do you know about disk de-fragmentation?
It can improve your application performance!
Compaction (Similar to Garbage Collection)

- Assumes programs are all relocatable
- Processes must be suspended during compaction
- Need be done only when fragmentation gets very bad

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Job 7</th>
<th>Job 5</th>
<th>Job 3</th>
<th>Job 8</th>
<th>Job 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Job 7</th>
<th>Job 5</th>
<th>Job 3</th>
<th>Job 8</th>
<th>Job 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Job 7</th>
<th>Job 5</th>
<th>Job 3</th>
<th>Job 8</th>
<th>Job 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Job 7</th>
<th>Job 5</th>
<th>Job 3</th>
<th>Job 8</th>
<th>Job 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Job 7</th>
<th>Job 5</th>
<th>Job 3</th>
<th>Job 8</th>
<th>Job 6</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How do you implement variable partitions?

Method 1: Memory Management with Linked Lists

- Use a linked list of allocated and free memory segments (called hole)
  - sorted by the address or by the size

Four neighbor combinations for the terminating process X

<table>
<thead>
<tr>
<th>Before X terminates</th>
<th>After X terminates</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) A X B</td>
<td>A ![Shaded] B</td>
</tr>
<tr>
<td>(b) A ![Shaded] X</td>
<td>A ![Shaded]</td>
</tr>
<tr>
<td>(c) ![Shaded] X B</td>
<td>![Shaded] B</td>
</tr>
<tr>
<td>(d) ![Shaded] X</td>
<td>![Shaded]</td>
</tr>
</tbody>
</table>
Method 2: bitmap

- Use bitmaps for free lists.
- Memory is divided into allocation units.
  - One allocation unit corresponds to 1 bit in the bitmap
  - 0: free, 1: allocated
- Size of allocation unit
  - The smaller the allocation unit, the larger the bitmap.
- Problem: allocation
  - When a new process arrives, the manager must find consecutive 0 bits in the map.
  - Searching a bitmap for a run of a given length is a slow operation.
Storage Placement Strategies

- **Analogy:**
  - Shoe Fitting
  - Valet parking

- **Best Fit**
  - Use the hole whose size is equal to the need, or if none is equal, the hole that is larger but closest in size.
  - Problem: Creates small holes that can't be used.

- **First Fit**
  - Use the first available hole whose size is sufficient to meet the need.
  - Problem: Creates average size holes.

- **Next Fit**
  - Minor variation of first fit: search for the last hole stopped.
  - Problem: slightly worse performance than first fit.

- **Worst Fit**
  - Use the largest available hole.
  - Problem: Gets rid of large holes making it difficult to run large programs.

- **Quick Fit**
  - maintains separate lists for some of the more common sizes requested.
  - When a request comes for placement it finds the closest fit.
  - This is a very fast scheme, but a merge is expensive. If merge is not done, memory will quickly fragment in a large number of holes into which no processes fit.
Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory.
User/Process Perspective

- Users (and processes) view memory as one contiguous address space from 0 through N
  - Virtual address space (VAS)
- In reality, pages are scattered throughout physical storage
  - Difference from variable partition
- The mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of its VAS
  - The address “0x1000” maps to different physical addresses in different processes
Question

- Why pages size is always a power of 2?
  - Example: 4096=4K, 8192=8K

- Why not 1000 or 2000?

- Hint: given a virtual address, how do you compute its virtual page number?
Paging

- Translating addresses
  - Virtual address has two parts: virtual page number and offset
  - Virtual page number (VPN) is an index into a page table
  - Page table determines page frame number (PFN)
  - Physical address is PFN::offset

- Page tables
  - Map virtual page number (VPN) to page frame number (PFN)
    - VPN is the index into the table that determines PFN
  - One page table entry (PTE) per page in virtual address space
    - Or, one PTE per VPN
Page Lookups

Virtual Address
Page number  Offset

Page Table
Page frame No.

Physical Address
Page Frame No.  Offset

Physical Memory
Paging Example

- Pages are 4K
  - VPN is 20 bits ($2^{20}$ VPNs), offset is 12 bits
- Virtual address is 0x7468
  - Virtual page is 0x7, offset is 0x468
- Page table entry 0x7 contains 0x2000
  - Page frame number is 0x2000
  - Seventh virtual page is at address 0x2000 (2nd physical page)
- Physical address = 0x2000 + 0x468 = 0x2468
Page Table Entries (PTEs)

- Page table entries control mapping
  - The **Modify** bit says whether or not the page has been written
    - It is set when a write to the page occurs
  - The **Reference** bit says whether the page has been accessed
    - It is set when a read or write to the page occurs
  - The **Valid** bit says whether or not the PTE can be used
    - It is checked each time the virtual address is used
  - The **Protection** bits say what operations are allowed on page
    - Read, write, execute
  - The **page frame number** (PFN) determines physical page
Paging Advantages

- Easy to allocate memory
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem

- Easy to swap out chunks of a program
  - All chunks are the same size
  - Use valid bit to detect references to swapped pages
  - Pages are a convenient multiple of the disk block size
Paging Limitations

- Can still have internal fragmentation in the last page
  - But not a big deal
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (more later)
- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = \(2^{20}\) PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - Solution – page the page tables (more later)
A variation of paging: Segmentation

- Segmentation is a technique that partitions memory into logically related data units
  - Module, procedure, stack, data, file, etc.
  - Virtual addresses become <segment #, offset>
  - Units of memory from user’s perspective

- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process

- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table
Segment Lookups

Virtual Address

Segment Table

Physical Memory

Segment #
Offset

<
Yes?
No?
Protection Fault

limit
base

+
Segment Table

- **Extensions**
  - Can have one segment table per process
    - Segment #s are then process-relative (why do this?)
  - Can easily share memory
    - Put same translation into base/limit pair
    - Can share with different protections (same base/limit, diff prot)

- **Problems**
  - Cross-segment addresses
    - Segments need to have same #s for pointers to them to be shared among processes
  - Large segment tables
    - Keep in main memory, use hardware cache for speed
  - Large segments
    - Internal fragmentation, paging to/from disk is expensive
Can combine segmentation and paging
  - The x86 supports segments and paging

Use segments to manage logically related units
  - Module, procedure, stack, file, data, etc.
  - Segments vary in size, but usually large (multiple pages)

Use pages to partition segments into fixed size chunks
  - Makes segments easier to manage within physical memory
    - Segments become “pageable” – rather than moving segments into and out of memory, just move page portions of segment
    - Need to allocate page table entries only for those pieces of the segments that have themselves been allocated

Tends to be complex…
Swapping

- Move a part of or the whole process to disk
- Allows several processes to share a fixed partition
- Processes that grow can be swapped out and swapped back in a bigger partition
- Real life analogy?
  - Putting things from your shelf to your parents house
Swapping

Monitor

User Partition

Disk

User 1

CSE 120 – Lecture 9 – Memory Management
Swapping

Monitor

User 1

User Partition

Disk

User 1

User 2

CSE 120 – Lecture 9 – Memory Management
Swapping

Monitor

User Partition

User 2

User 1

User 2

Disk
Swapping

Monitor

User Partition

User 2

User 1

User 2

Disk

CSE 120 – Lecture 9 – Memory Management

10/31/18
Swapping

Monitor

User 1

User Partition

Disk

User 1

User 2
External Resources

- Virtual Memory: introduction
  - https://www.youtube.com/watch?v=qcBIvnQt0Bw
- Three problems with Virtual memory
  - https://www.youtube.com/watch?v=eSPFB-xF5iM
- What is Virtual Memory?
  - https://www.youtube.com/watch?v=qlH4-oHnBb8
- How Does Virtual Memory Work?
  - https://www.youtube.com/watch?v=59rEMnKWoS4
- Page Table
  - https://www.youtube.com/watch?v=KNUJhZCQZ9c
- Address translation
  - https://www.youtube.com/watch?v=ZjKS1ibiGDA
- An example
  - https://www.youtube.com/watch?v=6neHHkI0Z0o
- Page fault
  - https://www.youtube.com/watch?v=bShqyf-hDfg

- Memory Protection
  - https://www.youtube.com/watch?v=uDzXXnNy544
- Making virtual memory fast (TLB)
  - https://www.youtube.com/watch?v=uyrSn3qbZ8U
- TLB Example
  - https://www.youtube.com/watch?v=95QpHJX55bM
- Multi-level Page Table
  - https://www.youtube.com/watch?v=Z4kSOv49GNc
- TLB and caches
  - https://www.youtube.com/watch?v=3sX5obQCHNA
- Summary
  - https://www.youtube.com/watch?v=FRvzCrWcFZA
Summary

- **Virtual memory**
  - Processes use virtual addresses
  - OS + hardware translates virtual address into physical addresses

- **Various techniques**
  - Fixed partitions – easy to use, but internal fragmentation
  - Variable partitions – more efficient, but external fragmentation
  - Paging – use small, fixed size chunks, efficient for OS
  - Segmentation – manage in chunks from user’s perspective
  - Combine paging and segmentation to get benefits of both

- **Swapping**
- **Protection**