Lecture 9: Memory Management

Ding Yuan
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

• 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”
    - Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
  - Processes cannot see the memory of others’
  - 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…

• Rewind to the old days (generally before 1970s)
  • 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
  - Instructions executed by the CPU issue virtual addresses
  - 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
Remember this example?

```c
int myval;
int main(int argc, char *argv[])
{
    myval = atoi(argv[1]);
    while (1)
        printf("myval is %d, loc 0x%lx\n", myval, (long) &myval);
}
```

- Now *simultaneously* start two instances of this program
  - Myval 5
  - Myval 6
- What will the outputs be?
<table>
<thead>
<tr>
<th>Default</th>
<th>D</th>
<th>Thank</th>
</tr>
</thead>
<tbody>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
<tr>
<td>myval is 5, loc 0x2030</td>
<td>myval is 6, loc 0x2030</td>
<td></td>
</tr>
</tbody>
</table>
Virtual Addresses

- Many ways to do this translation…
  - Start with old, simple ways, progress to current techniques
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

- 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

Base Register
P4's Base

Virtual Address
Offset

Physical Memory
P1
P2
P3
P4
P5

Internal fragmentation
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
Variable Partitions and Fragmentation

Memory wasted by External Fragmentation

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

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

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

4. Monitor | Job 5 | Job 3 | Job 4 | Job 6

5. Monitor | Job 7 | Job 5 | Job 3 | Job 6

Do you know about disk de-fragmentation? It can improve your system performance!
Compaction

- Processes must be suspended during compaction
- Need be done only when fragmentation gets very bad
Paging

- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory.
Internal vs. External fragmentation

- How paging can solve fragmentation problems?
  - External fragmentation: can be solved by re-mapping between VA and PA
  - Internal fragmentation: can be solved if the page size is relatively small
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
  • Different from variable partition, where the physical memory for each process is contiguously allocated

• 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

- Page size is always a power of 2
  - Examples: 4096 bytes = 4KB, 8192 bytes = 8KB
  - Why?
  - Why not 1000 or 2000?
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

Physical Address

Page frame  Offset

Physical Memory
Paging Example

- Pages are 4K (Linux default)
  - VPN is 20 bits ($2^{20}$ VPNs), offset is 12 bits
- Virtual address is 0x7468 (hexadecimal)
  - 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 Lookups

Example: how do we ‘load 0x00007468’?

Questions:
1. How large is the RAM?
2. How big is the page table?
3. Besides page frame, what else we need to store in the page table?
Page Table Entries (PTEs)

<table>
<thead>
<tr>
<th>M</th>
<th>R</th>
<th>V</th>
<th>Prot</th>
<th>Page Frame Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>20</td>
<td>20 (determined by the size of physical memory)</td>
</tr>
</tbody>
</table>

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
- If you’re interested: watch the OS lecture scene from “The Social Network” again, see if now you can understand

http://www.youtube.com/watch?v=-3Rt2_9d7Jg
2-level page table

- Single level page table size is too large
  - 4KB page, 32 bit virtual address, 1M entries per page table!
Linear address:

31  24  23  16  15  8  7  0

Page directory

32 bit PD entry

32* CR3

Page table

32 bit PT entry

4K memory page

*) 32 bits aligned to a 4-KByte boundary
Two-Level Page Tables

- Two-level page tables
  - Virtual addresses (VAs) have three parts:
    - Master page number, secondary page number, and offset
  - Master page table maps VAs to secondary page table
  - Secondary page table maps page number to physical page
  - Offset indicates where in physical page address is located

- Example
  - 4K pages, 4 bytes/PTE
  - How many bits in offset? 4K = 12 bits
  - Want master page table in one page: 4K/4 bytes = 1K entries
  - Hence, 1K secondary page tables. How many bits?
  - Master (1K) = 10, offset = 12, inner = 32 – 10 – 12 = 10 bits
Two-Level Page Tables

Virtual Address

Master page number | Secondary | Offset

Page table

Master Page Table

Page frame

Physical Address

Page frame | Offset

Secondary Page Table

Physical Memory
What is the problem with 2-level page table?

Hints:
- Programs only know virtual addresses
- Each virtual address must be translated
  - Each program memory access requires several actual memory accesses
- Will discuss solution in the next lecture
### 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
  - Process may not use memory in multiples of a page

- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
    - Even more for two-level page tables!
  - 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!
    - Remember: each process has its own page table!
  - Solution – 2-level page tables
What if a process requires more memory than physical memory?

- **Swapping**
  - Move one/several/all pages of a process to disk
    - Free up physical memory
    - “Page” is the unit of swapping
  - The freed physical memory can be mapped to other pages
  - Processes that use large memory can be swapped out (and later back in)

- **Real life analogy?**
  - Putting things from your shelf to your parents’ house
Swapping

Physical memory

Disk

Process 1
Swapping process 1’s data into memory
Swapping
Swapping

Physical memory

Disk

Process 1

Process 2

Process 2
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

[Diagram showing the process of segment lookups.]

1. **Virtual Address**
2. **Segment #**
3. **Offset**
4. **Segment Table**
   - limit
   - base
5. **Yes?**
6. **No?**
7. **Protection Fault**
8. **Physical Memory**

The process involves comparing the virtual address with the limit in the segment table. If the address is within the limit, it proceeds to the physical memory. If not, a protection fault is generated.
Segment Table

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

- Problems
  - Large segment tables
    - Keep in main memory, use hardware cache for speed
  - Large segments
    - Internal fragmentation, paging to/from disk is expensive
Segmentation and Paging

- 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...
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