Memory Management
CSCI 444/544 Operating Systems
Fall 2008

Agenda
• Background
• Address space
• Static vs Dynamic allocation
• Contiguous vs non-contiguous allocation

Background
Program must be brought into memory and placed within a process for it to be run

Uniprogramming: one process runs at a time

Multiprogramming: several processes coexist in main memory

Binding of Instructions and Data to Memory
Address binding of instructions and data to memory addresses can happen at three different stages

Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes

Load time: Must generate relocatable code if memory location is not known at compile time

Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).
Multi-step Processing of a User Program

Goals of Memory Management
- **Sharing**: allocate scarce memory resources among competing processes, maximizing memory utilization and system throughput
- **Transparency**: a convenient abstraction for programming (and for compilers, etc.)
- **Protection**: provide isolation between processes
  - we have come to view “addressability” and “protection” as inextricably linked, even though they’re really orthogonal
- **Low overhead**: fast address translation and fast updating in context switching

Address Space
Remember what a process is?
- address space + 1 or more threads

Address space: unit of protection
- memory space that the threads use
- including all the data the program uses as it runs (program code, stack, data segments)

Illusions provided by address spaces
- **address independence**: each process use addresses starting at 0
- **virtual memory**: much larger than available physical memory
- **protection**: process can only access data in its own address space

Logical vs Physical Address Space
The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
- **Logical address** – generated by the CPU; also referred to as virtual address
- **Physical address** – address seen by the memory unit

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
Memory-Management Unit (MMU)

Hardware device that maps logical to physical address

In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory

The user program deals with *logical* addresses; it never sees the *real* physical addresses

---

Static Allocation

Goal: Allow transparent sharing - Each address space may be placed anywhere in memory

- OS finds free space for new process
- Modify addresses statically (similar to linker) when load process

---

Discussion of Static Allocation

Advantages
- Requires no hardware support

Disadvantages
- No protection
  - Process can destroy OS or other processes
  - No privacy
- Address space must be allocated contiguously
  - Allocate space for worst-case stack and heap
- Cannot move address space after it has been placed
  - May not be able to allocate new process

---

Dynamic Allocation

Goal: Protect processes from one another
Requires hardware support

- Memory Management Unit (MMU)

MMU dynamically changes process address at every memory reference

- Process generates *logical or virtual* addresses
- Memory hardware uses *physical or real* addresses

---
Hardware Support for Dynamic Allocation

Two operating modes
• Privileged (protected, kernel) mode: OS runs
  – When enter OS (trap, system calls, interrupts, exceptions)
  – Allows certain instructions to be executed
    • Can manipulate contents of MMU
  – Allows OS to access all of physical memory
• User mode: User processes run
  – Perform translation of logical address to physical address

MMU contains base and limit registers
• base: start location for address space
• limit: size limit of address space

Schemes of Memory Management

Base and limit registers

Swapping

Paging (and page tables and TLBs)

Segmentation (and segment tables)

Page fault handling => Virtual memory

Base and Limit Registers

Translation on every memory access of user process
• MMU compares logical address to limit register
  – if logical address is greater, then generate error
• MMU adds base register to logical address to form physical address

Managing Processes with Base and Limit

Context-switch
• Add base and limit registers to PCB
• Steps
  – Change to privileged mode
  – Save base and limit registers of old process
  – Load base and limit registers of new process
  – Change to user mode and jump to new process

What if don’t change base and limit registers when switch?

Protection requirement
• User process cannot change base and limit registers
• User process cannot change to privileged mode
Base and Limit Discussion

Advantages
- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
  - Can move address spaces
- Simple, inexpensive implementation
  - Few registers, little logic in MMU
- Fast
  - Add and compare can be done in parallel

Disadvantages
- Each process must be allocated contiguously in physical memory
  - Must allocate memory that may not be used by process
- No partial sharing: Cannot share limited parts of address space

Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

Backing store (swap space) – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

Schematic View of Swapping

Main memory usually into two partitions:
- Resident operating system, usually held in low memory with interrupt vector
- User processes then held in high memory

Single-partition allocation
- base-register scheme used to protect user processes from each other, and from changing operating-system code and data
- base register contains value of smallest physical address; limit register contains range of logical addresses

Multiple-partition allocation
- Fixed partition
- Variable partition
Fixed Partitions

Physical memory is broken up into fixed partitions
  • partitioning never changes
    – either each partition has separate input queue
    – or all partitions with a single input queue
  • how do we provide protection?
    – use base + limit registers

Advantages
  • Simple

Problems
  • internal fragmentation: the fixed size partition is larger than what was requested
  • external fragmentation: two small partitions left, but one big job left

Variable (Dynamic) Partitions

Physical memory is broken up into variable-sized partitions
  • partitions are created dynamically, each process is loaded into a partition of exactly the same size as the process

Advantages
  • no internal fragmentation
    – simply allocate partition size to be just big enough for process

Problems
  • external fragmentation
    – as we load and unload jobs, holes are left scattered throughout physical memory
    – slightly different than the external fragmentation for fixed partition systems

Dynamic Partition Allocation

Variable partition allocation
  • Hole – block of available memory; holes of various size are scattered throughout memory
  • When a process arrives, it is allocated memory from a hole large enough to accommodate it
  • Operating system maintains information about:
    a) allocated partitions    b) free partitions (hole)

Dynamic Allocation Problem

How to satisfy a request of size $n$ from a list of free holes

First-fit: Allocate the first hole that is big enough
Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
Worst-fit: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization
Fragmentation

**External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous

**Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Reduce external fragmentation by **compaction**
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible only if relocation is dynamic, and is done at execution time

Non-contiguous Allocation

Paging
- address translation
- page table
- hardware support
  - Translation look-aside buffers (TLBs)

Segmentation
- segmented addressing
- segmentation with paging

Paging

Solve the external fragmentation problem by using **fixed** sized units in both physical and virtual memory
- physical address space of a process can be **noncontiguous**
- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical memory into blocks of same size called **pages**
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

User’s Perspective

Processes view memory as a contiguous address space from bytes 0 through N

In reality, logical pages are scattered across physical memory frames – not contiguous as earlier
- virtual-to-physical mapping (page table)
- this mapping is **invisible** to the program
**Paging View**

Goal: Eliminate external fragmentation
Idea: Divide memory into fixed-sized pages
- Size: 2^n, Example: 4KB
- Physical page: page frame

**Logical View**

**Address Translation**

Logical address generated by CPU is divided into:
- **Page number (p)** – used as an index into a page table which contains base address of each page (page’s corresponding frame number f) in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

Physical address is f+d

**Page Tables**

- managed by the OS
- map logical page number (p) to physical frame number (f)
  - p is simply an index into the page table
- one page table entry (PTE) per page in logical address space
  - i.e., each p has PTE in the table

**Page Table Entries**

<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>M</th>
<th>prot</th>
<th>frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

PTE’s control mapping:
- the **valid** bit says whether or not the PTE can be used
  - it is checked each time a virtual address is used
- the **referenced** bit says whether the page has been accessed
  - it is set when a page has been read or written to
- the **modified** bit says whether or not the page is dirty
  - it is set when a write to the page has occurred
- the **protection bits** control which operations are allowed
  - read, write, execute
- the **frame number** determines the physical frame
  - physical frame start address
Mechanism of Address Translation

Paging Example

Implementation of Page Table

• Page table is kept in main memory
  – Page-table base register (PTBR) points to the page table
  – Page-table length register (PRLR) indicates size of the page table

• In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

• The two memory access problem can be solved by the use of a special fast-lookup hardware cache called translation look-aside buffers (TLBs)

Translation Look-Aside Buffer (TLB)

Goal: Avoid page table lookups in main memory

Idea: Hardware cache of recent page translations
• Typical size: 64 - 2K entries

Why does this work?
• process references few unique pages in time interval
• spatial, temporal locality

On each memory reference, check TLB for translation
• If present (hit): use cached frame number and append page offset
• Else (miss): Use page tables to get frame number
  – Update TLB for next access (replace some entry)
Paging Hardware with TLB

Paging Advantages

Easy to allocate physical memory
  - physical memory is allocated from free list of frames
  - to allocate a frame, just remove it from the free list
  - external fragmentation is not a problem!

Leads naturally to virtual memory
  - entire program need not to be memory resident
  - take page faults using "valid" bit
  - but paging was originally introduced to deal with external fragmentation, not to allow programs to be partially resident

Paging disadvantages

Can still have internal fragmentation
  - process may not use memory in exact multiples of pages

Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - solution: use a hardware cache to absorb page table lookups
    - translation lookaside buffer (TLB)

Memory required to hold page tables can be large
  - need one PTE per page in address space
  - 32 bit AS with 4KB pages = 2^32 PTEs = 1,048,576 PTEs
  - 4 bytes/PTE = 4MB per page table
    - OS’s typically have separate page tables per process
    - 25 processes = 100MB of page tables

Segmentation

Divide address space into logical segments
  - A program is a collection of segments
  - Each segment corresponds to logical entity in address space
    - code, stack, heap

Each segment can independently:
  - be placed separately in physical memory
  - grow and shrink
  - be protected (separate read/write/execute protection bits)
User’s View of a Program

Logical View of Segmentation

Paging vs Segmentation

Paging
- mitigates various memory allocation complexities (e.g., fragmentation)
- view an address space as a linear array of bytes
- divide it into pages of equal size (e.g., 4KB)
- use a page table to map virtual pages to physical page frames
  - page (logical) \Rightarrow page frame (physical)

Segmentation
- partition an address space into logical units
  - stack, code, heap, subroutines, ...
- a virtual address is \langle segment #, offset\rangle

Why Segmentation

More “logical”
- a logical address space is a collection of variable-size segments
- they are really independent, no necessary order among segments

Facilitates sharing and reuse
- a segment is a natural unit of sharing – a subroutine or function

Different protection for different segments
- read-only status for code

A natural extension of variable-sized partitions
- variable-sized partition = 1 segment/process
- segmentation = many segments/process
Segmentation Architecture

Logical address consists of a two tuple:
<segment-number, offset>,

**Segment table** – each table entry has:
- **base** – contains the starting physical address where the segments reside in memory
- **limit** – specifies the length of the segment
- multiple base/limit pairs, one per segment
- segment-number used as index into segment table
- logical address: offset+base -> physical address
- each segment must be allocated **contiguously**

Address Translation

Sharing of Segments

Segmentation with Paging

Use segments to manage logical units
- segments vary in size, but are typically large (multiple pages)

Use pages to partition segments into fixed-size chunks
- each segment has its own page table
  - there is a page table per segment, rather than per user address space
- memory allocation becomes easy once again
  - no contiguous allocation, no external fragmentation

<table>
<thead>
<tr>
<th>Segment #</th>
<th>Page #</th>
<th>Offset within page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset within segment</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>