Memory Management – Outline

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Segmentation
- Segmented Paging

Background

- Memory is a large array of bytes
  - memory and registers are only storage CPU can access directly
  - both instructions and data are fetched from memory
  - memory unit just sees a stream on memory addresses
- Accessing memory is slow (why?)
  - register access in one CPU clock (or less)
  - cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
  - isolate processes from one another and from the OS

Process Protection in Hardware

- Protect OS from user processes and processes from each other
  - give each process separate space
  - determine legal range of addresses for each process
- A pair of base and limit registers define the process address space.
- CPU compares each address generated by the process with the base and limit registers.

Address Binding

- 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
    - code can be placed anywhere in memory
  - 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)
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 hardware memory unit

Logical and physical addresses are the same in compile-time and load-time address-binding schemes.

Logical and physical addresses differ in execution-time address-binding scheme.

**Memory-Management Unit (MMU)**

- Hardware device that maps virtual to physical address.
- **Simple MMU scheme**
  - start each process’s logical address from 0
  - base memory address for each process stored in relocation register
  - max. process address stored in limit register
  - relocation register added to every logical address
  - logical: 0 – max correspond to physical: R+0 – R+limit

**Dynamic Loading**

- **Static loading**
  - entire program in memory before program starts
  - program size limited by size of memory

- **Dynamic loading**
  - routine is not loaded until it is called
  - better memory-space utilization; unused routine is never loaded
  - useful when large amounts of code are needed to handle infrequently occurring cases
  - no special support from the operating system is required implemented through program design
Dynamic Linking

- Static linking
  - library modules considered as any other object files
  - linker/loader combines libraries into every binary image in which they may be referenced
  - multiple copies of library code may reside in memory at a time

- Dynamic linking
  - linking postponed until execution time
  - small piece of code, *stub*, used to locate the appropriate memory-resident library routine
  - stub replaces itself with the address of the routine
  - OS checks if routine is in processes’ memory address
  - only one copy of libraries resident in memory at a time
  - such a library is called a *shared* library

Swapping

- What is swapping?
  - moving a process currently resident in memory to a *backing store* and *swapping* another process into the freed space
  - swap individual *pages* of the same process in-out of memory

- When is swapping needed?
  - if there is no free memory for the new process dispatched by the scheduler
  - if some process needs more *physical* memory than is available

- Backing store is generally maintained on disk
  - swapping is costly
  - if swapped out memory needs to be brought back in immediately all the time, then it will lead to memory *thrashing*

- See ‘top’ and ‘swapon -s’ under Linux.

Schematic View of Swapping

Contiguous Memory Allocation

- Efficient allocation of main memory is important
  - memory is limited in size
  - contains the OS and multiple concurrently active user processes

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

- Contiguous memory allocation
  - each process in a single contiguous region of memory
  - allows multiple processes to reside in memory at the same time
  - processes must be protected from each other!
Hardware Support for Protection

- Relocation and limit registers used to protection
  - of user processes from each other, and
  - operating system from the user processes

- Hardware checks
  - base register contains value of smallest physical address
  - each logical address must be less than the limit register
  - MMU maps logical address dynamically

Contiguous Memory Allocation (2)

- Multiple-partition method
  - 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 Storage-Allocation Problem

- Given a list of variable-sized holes, how to allocate memory for a new arriving process of some size, \( n \)

- Memory allocation schemes
  - 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

- Performance
  - first-fit and best-fit better than worst-fit in terms of performance and storage requirement
  - first-fit generally the fastest

Fragmentation

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

- 50-percent rule
  - given \( N \) allocated blocks, \( 0.5N \) blocks will be lost to fragmentation

- Solutions to external fragmentation
  - compaction
    - move allocated blocks to place all holes together
    - compaction is possible only if relocation is dynamic
  - permit non-contiguous allocation of logical address space

- Internal Fragmentation
  - occurs with fixed sized blocks used to reduce cost of managing extremely small holes
  - size of last block may be slightly larger than requested memory
  - what is the greatest memory loss due to internal fragmentation?
Paging

- Solution to external fragmentation and compaction
  - logical address space of a process can be noncontiguous
  - commonly used in most OS
  - frequently depend on hardware support

- Scheme
  - divide physical memory into fixed-sized blocks called frames
    - size is generally power of 2, between 512 bytes and 8,192 bytes
  - divide logical memory into blocks of same size called pages
  - OS keeps track of all free frames
  - a program of size $n$ pages, needs $n$ free frames before loading
  - needs a page table to translate logical to physical addresses

- Problem of internal fragmentation still exists

Address Translation Scheme

- Hardware defines the page size (and frame size).
- Page size, $d$, determines the number of bits required to specify the page offset, $n$.
  - How?
  - $d = 2^n$
- The remaining bits in the logical address specify the page number, $p$.
- For logical address space, $2^m$, and page size $2^n$,

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$d$</td>
</tr>
</tbody>
</table>

  $m - n$ $n$

Paging Hardware

Paging Model of Logical and Physical Memory
Page Table Issues

- Small or large page size?
  - small page size reduces internal fragmentation loss, how?
  - large page size reduces size of page table, how?
  - larger transfers produces more efficient disk transfers
- Each memory load/store goes through page table
  - page table access must be fast
  - store page tables in hardware registers
  - only possible for small tables, up to 256 entries!
- Each process has its own page table
  - page table switching on each context switch must be fast
- Most computers allow large page tables
  - page table is kept in main memory
  - single register points to page table address in memory

Page Table Implementation

- Page-table base register (PTBR) points to page table
  - only this register affected on each context switch
- Every load/store requires two memory accesses
  - one for the page table and one for the data/instruction
- Translation look-aside buffer (TLB)
  - cache to hold popular page table entries
  - access to TLB should be very fast, use associative cache
- However, each process has its own page table
  - on a context switch, OS changes PTBR pointer
  - and, may need to purge the TLB
  - use address-space identifiers (ASIDs) in each TLB entry
    - uniquely identifies each process to provide address-space protection for that process
Memory protection implemented by associating protection bit with each frame

- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process' logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process' logical address space

Memory Protection (2)

- **Scheme**
  - 14 bit address space
  - Program uses addresses 0 – 10468
  - Page size of 2kb
  - Accessing addresses on pages 6 and 7 is invalid → segmentation fault

- **Problems**
  - addresses 10469 – 12287 are still valid
  - page table still contains all rows
  - use page table length register

Shared Pages

- **Shared code**
  - Paging allows the possibility of easily sharing code
  - one copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - each process has private data
  - significant reduction in memory usage possible

- **Example**
  - 3 code pages shared
  - each process has private data
Hierarchical Page Tables

- Sometimes necessary to divide page table into pieces
  - to support a large logical address space ($2^{32}$ to $2^{64}$)
  - example: for page size of 4kb, page table has $2^{20}$ entries
  - contiguous allocation of big page tables not feasible

- Two-level page table (paged page-table)
  - further divide the page number field
  - $p_1$ – index into outer page table
  - $p_2$ – displacement within outer page table

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Hierarchical Paging

- Hashed Page Tables

Inverted Page Tables

Two-Level Page-Table Scheme

Address-Translation Scheme for Two-Level Paging
More Levels in Hierarchical Mapping

- Two level scheme not appropriate for 64-bit machine
  - divide the outer page table further
    - outer page | inner page | offset
      - p₁ | p₂ | d
      - 42 | 10 | 12
  - Even three-level scheme is daunting
    - outer page table still \(2^{34}\) bytes in size
  - We can go on increasing the number of paging levels
    - but, each level requires a memory access
    - two-level page table requires three memory accesses for each load/store
    - n-level requires \(n+1\) memory accesses!
    - need another page table implementation...

Hashed Page Tables

- Common in address spaces > 32 bits
  - virtual page number is the hash
- Each hash table entry contains a linked list of elements
- Each element contains
  - virtual page number
  - value of mapped page frame
  - next pointer
- Algorithm
  - hash virtual page number into hash table
  - compare with each element in linked list until match is found
  - use corresponding page frame to form physical address

Hashed Page Tables (2)

Inverted Page Table

- Attempt to overcome drawbacks of traditional page tables
  - machines with large virtual address spaces have huge tables
  - consume lot of memory
- Inverted page table approach
  - table contains one entry for each real page of memory
  - entry consists of the virtual address of the
  - also contains information on process that owns that page
  - see figure on next page
- Decreases memory needed to store each page table, but
  - linear search of table needed to find physical mapping
  - substantial increase in search time for every memory access
  - Use hash table to limit the search to one, or few page-table entries
**Inverted Page Table (2)**

- Logical address consists of a two tuple: 
  $$\langle \text{segment-number}, \text{offset} \rangle$$

- **Segment table** – maps two-dimensional physical addresses; each table entry has
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment

- **Segment-table base register (STBR)** points to the segment table’s location in memory

- **Segment-table length register (STLR)** indicates number of segments used by a program;
  - segment number $$s$$ is legal if $$s < \text{STLR}$$

---

**Segmentation Architecture**

- Memory-management scheme that supports user view of memory
  - a program is a collection of segments

- **Segmentation Architecture (2)**

- Logical address consists of a two tuple: 
  $$\langle \text{segment-number}, \text{offset} \rangle$$

- **Segment table** – maps two-dimensional physical addresses; each table entry has
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment

- **Segment-table base register (STBR)** points to the segment table’s location in memory

- **Segment-table length register (STLR)** indicates number of segments used by a program;
  - segment number $$s$$ is legal if $$s < \text{STLR}$$
Example of Segmentation

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - given to segmentation unit
    - which produces linear addresses
  - linear address given to paging unit
    - which generates physical address in main memory
    - paging units form equivalent of MMU

Intel Pentium Segmentation

- 32-bit architecture
- Max. number of segments – 16K
  - 8k segments in local descriptor table (LDT)
  - 8k segments in global descriptor table (GDT)
- Max. size of segment – 4GB
- Each segment descriptor is 8 bytes
  - contains segment base and limit
- Each logical address is 48 bits
  - 16 bit segment selector, 32 bit offset
- 16 bit segment selector contains
  - 13 bit segment number, 1 bit table identifier, 2 bits of protection
- Maps 48 bit logical address to 32 bit linear address

Example: The Intel Pentium

- CPU generates logical address
  - given to segmentation unit
    - which produces linear addresses
  - linear address given to paging unit
    - which generates physical address in main memory
    - paging units form equivalent of MMU

Intel Pentium Segmentation (2)

- Each logical address is 48 bits
  - 16 bit segment selector, 32 bit offset
- 16 bit segment selector contains
  - 13 bit segment number, 1 bit table identifier, 2 bits of protection
- Maps 48 bit logical address to 32 bit linear address
Pentium Paging Architecture

- Page size – 4kb or 4mb
- For 4kb pages – two level paging scheme

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$d$</td>
</tr>
<tr>
<td>$p_2$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

- base of page directory pointed to by CR3 register
- 10 high-order bits offset into the page directory
- page directory gives base of inner page table
- inner page table indexed by middle 10 bits of linear address
- page table returns base of page
- low-order 12 bits of linear address used to offset page