CSMC 412
Operating Systems
Prof. Ashok K Agrawala

Memory Management
Memory Management

• Background
• Swapping
• Contiguous Memory Allocation
• Segmentation
• Paging
• Structure of the Page Table
CPU and Memory

• Basic architecture of a computer system requires the CPU and the main memory

• All programs and data accessed by the CPU during the execution of instructions is either in the registers or in the main memory
  • For the discussion here we are going to ignore the presence of Cache Memory which many CPUs have today and whose presence is managed by the hardware transparently

• For executing an instruction
  • Instruction has to be fetched from the memory
  • Operand(s) have to be fetched from the memory – if so required
  • Results may have to be stored in memory – if so required

• CPU may make multiple memory accesses for each instruction
Background

• Program must be brought (from disk) into memory and placed within a process for it to be run
• Main memory and registers are only storage CPU can access directly
• Memory unit only sees a stream of addresses + read requests, or address + data and write requests
• Register access in one CPU clock (or less)
• Main memory can take many cycles, causing a stall
  • Cache sits between main memory and CPU registers
• Protection of memory required to ensure correct operation
View of the memory

• An array of cells

• Each cell can store several bits (cell width)
  • 8- Byte
  • 16 – Half Word
  • 32 – Word
  • ..

• Cells are organized as a linear array with each cell having a unique address

• A memory cell is accessed by the CPU by presenting the address of the cell to the memory controller
Address Space

• The address of a cell consists of say \( n \) bits. This gives \( 2^n \) unique addresses, from 0 to \( (2^n - 1) \)

• We can view this address space in *any logical organization* we desire, treating any number of contiguous cells as a block.

• When the number of such cells in a block is a power of 2 then the address can be decomposed easily into the group number and the cell within the group
Mapping of Address Spaces

• We can map any address space, “A” into any other address space “B” as long as we can get a unique address for one given address for the other.
  • Map one address to the other
  • Address spaces do not have to be of the same size. (Size defined by the number of bits required to specify a unique address.
  • Usually, the cell sizes are the same but if they are not, a mapping there is required also.
    • One organized with cell size of one byte while the other with cell size of a word – 4 bytes

• How to map???
Mapping of address spaces

• How to map (Going from “A” to “B”)
  • Lookup table
    • Organized by cells
      • Each address in A has an entry for the address in B
      • Use A as an index in the array which has the corresponding address B

n bit address

m bit address
Desirable Features

• Very large address space
• Ability to execute partially loaded programs
• Dynamic Relocatability
• Sharing
• Protection

• Achieving these features require a variety of hardware and software support
Binding and Multiple Mappings

• Binding
  • Associating an address to a location in an address space

• Mapping
  • Translating one address to another address
  • Each address is defined in an address space
  • Mapping one address space to another address space

• Mapping is never done on Byte by Byte
  • A contagious portion is mapped on to a contagious portion
Address Binding

• Programs on disk, ready to be brought into memory to execute form an input queue
  • Without support, must be loaded into address 0000
• Inconvenient to have first user process physical address always at 0000
  • How can it not be?
• Further, addresses represented in different ways at different stages of a program’s life
  • Source code addresses usually symbolic
  • Compiled code addresses bind to relocatable addresses
    • i.e. “14 bytes from beginning of this module”
  • Linker or loader will bind relocatable addresses to absolute addresses
    • i.e. 74014
• Each binding maps one address space to another
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)
Multistep Processing of a User Program
Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- 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, and executes the routine
- Operating system checks if routine is in processes’ memory address space
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
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
Overlays

• Keep in memory only those instructions and data that are needed at any given time

• Needed when process is larger than amount of memory allocated to it

• Implemented by user, no special support needed from operating system, programming design of overlay structure is complex
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

• **Logical address space** is the set of all logical addresses that can be generated by a program

• **Physical address space** is the set of all physical addresses that can be generated by a program
Memory-Management Unit (MMU)

• Hardware device that at run time maps virtual to physical address
• Many methods possible
• To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  • Base register now called relocation register
  • MS-DOS on Intel 80x86 used 4 relocation registers
• The user program deals with logical addresses; it never sees the real physical addresses
  • Execution-time binding occurs when reference is made to location in memory
  • Logical address bound to physical addresses
Swapping

• A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  • Total physical memory space of processes can exceed physical memory

• **Backling store** – 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

• System maintains a **ready queue** of ready-to-run processes which have memory images on disk
Swapping (Cont.)

• Does the swapped out process need to swap back in to same physical addresses?
• Depends on address binding method
  • Plus consider pending I/O to / from process memory space
• Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  • Swapping normally disabled
  • Started if more than threshold amount of memory allocated
  • Disabled again once memory demand reduced below threshold
Schematic View of Swapping

1. Swap out
2. Swap in
Context Switch Time including Swapping

- If next processes to be run on the CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`
Context Switch Time and Swapping (Cont.)

• Other constraints as well on swapping
  • Pending I/O – can’t swap out as I/O would occur to wrong process
  • Or always transfer I/O to kernel space, then to I/O device
    • Known as double buffering, adds overhead

• Standard swapping not used in modern operating systems
  • But modified version common
    • Swap only when free memory extremely low
Contiguous Allocation

• Main memory must support both OS and user processes
• Limited resource, must allocate efficiently
• Contiguous allocation is one early method
• Main memory usually into two partitions:
  • Resident operating system, usually held in low memory with interrupt vector
  • User processes then held in high memory
  • Each process contained in single contiguous section of memory
Contiguous Allocation (Cont.)

• Relocation registers 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 – each logical address must be less than the limit register
  • MMU maps logical address *dynamically*
  • Can then allow actions such as kernel code being *transient* and kernel changing size
Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
Hardware Support for Relocation and Limit Registers

![Diagram showing the process of logical to physical address translation with decision points for limit and relocation registers and error handling for addressing errors.]

- CPU
- Logical address
- Limit register
- Yes
- Physical address
- Relocation register
- No
- Trap: addressing error
- Memory
Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- 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
  - OS can help by providing libraries to implement dynamic loading
Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process’ needs)
- **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
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions    
  b) free partitions (hole)
Dynamic Storage-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
  - First-fit and best-fit better than worst-fit in terms of speed and storage utilization

- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole
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

• First fit analysis reveals that given $N$ blocks allocated, $0.5 \times N$ blocks lost to fragmentation
  • $1/3$ may be unusable -> **50-percent rule**
Fragmentation (Cont.)

- 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

- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

- Now consider that backing store has same fragmentation problems
Memory Management Schemes from I

• The logical address space of a process must be resident in the physical memory when this process is executing

Desirable Features:
• Very large address space
• Ability to execute partially loaded programs
  • Dynamic Relocatability
• Sharing
• Protection
Programmers View

- Each Module starts with address 0
- When linking/loading start from 0 again but only one module can be at location 0. Others have to be relocated.
- What if each module was treated independently.
Logical Address Space

• Single linear address space
  • Address is \((d)\) where \(d\) is a number between 0 and \(2^k - 1\), when address uses \(k\) bits.

• Segmented
  • Logical address space consists of a collection of segments where each segment is an independent linear address space
  • Address now consists of \((s,d)\) where \(s\) is the segment identifier and \(d\) an address in the linear address space of the segment
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:
    - main program
    - procedure
    - function
    - method
    - object
    - local variables, global variables
    - common block
    - stack
    - symbol table
    - arrays
User’s View of a Program
Logical View of Segmentation

user space

physical memory space

1
4
2
3

April 2020
Segmentation Architecture

• Logical address consists of a two tuple:
  \(<\text{segment-number}, \text{offset}>\),

• **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 (Cont.)

• Protection
  • With each entry in segment table associate:
    • validation bit = 0 ⇒ illegal segment
    • read/write/execute privileges

• Protection bits associated with segments; code sharing occurs at segment level

• Since segments vary in length, memory allocation is a dynamic storage-allocation problem

• A segmentation example is shown in the following diagram
Example of Segmentation
Sharing of Segments
Paging

• Used to map a linear, contiguous Logical Address Space on to (linear) Physical Address Space

• View physical memory consisting of fixed-sized blocks called frames
  • Size is power of 2, between 512 bytes and 16 Mbytes

• View logical memory consisting of blocks of same size called pages

• To run a program of size $N$ pages, need to find $N$ free frames and load program

• Set up a page table to translate logical to physical addresses
Address Space

• The address of a cell consists of say n bits. This gives $2^n$ unique addresses, from 0 to $(2^n - 1)$

• We can view this address space in *any logical organization* we desire, treating any number of contiguous cells as a block.

• When the number of such cells in a block is a power of 2 then the address can be decomposed easily into the group number and the cell within the group
Paging Model of Logical and Physical Memory

![Diagram of paging model]

- Logical memory:
  - page 0
  - page 1
  - page 2
  - page 3

- Page table:
  - 0 1
  - 1 4
  - 2 3
  - 3 7

- Physical memory:
  - Frame number:
    - 0
    - 1 page 0
    - 2 page 0
    - 3 page 2
    - 4 page 1
    - 5
    - 6
    - 7 page 3
Address Translation Scheme

• 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 in physical memory
  • Page offset \((d)\) – combined with base address to define the physical memory address that is sent to the memory unit

  - \(p\) and \(d\) are split as follows:
    - \(p\) occupies \(m - n\) bits
    - \(d\) occupies \(n\) bits

• For given logical address space \(2^m\) and page size \(2^n\)
Paging Hardware
Paging Example

$n=2$ and $m=4$  32-byte memory and 4-byte pages
Paging (Cont.)

• Calculating internal fragmentation
  • Page size = 2,048 bytes
  • Process size = 72,766 bytes
  • 35 pages + 1,086 bytes
  • Internal fragmentation of 2,048 - 1,086 = 962 bytes
  • Worst case fragmentation = 1 frame – 1 byte
  • On average fragmentation = 1 / 2 frame size
  • So small frame sizes desirable?
  • But each page table entry takes memory to track
  • Page sizes growing over time
    • Solaris supports two page sizes – 8 KB and 4 MB

• Process view and physical memory now very different
• By implementation process can only access its own memory
Free Frames

Before allocation

After allocation
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** (**PTLR**) 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 helped some by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)
Translation Lookaside Buffer

• Keep a list of translations
• Provide means for fast look up
Implementation of Page Table (Cont.)

• Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  • Otherwise need to flush at every context switch
• TLBs typically small (64 to 1,024 entries)
• On a TLB miss, value is loaded into the TLB for faster access next time
  • Replacement policies must be considered
  • Some entries can be **wired down** for permanent fast access
Associative Memory

• Associative memory – parallel search

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Address translation (p, d)
  • If p is in associative register, get frame # out
  • Otherwise get frame # from page table in memory
Paging Hardware With TLB
Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - The time for accessing TLB is often ignored as it is overlapped with memory access.
- **Effective Access Time (EAT)**
  - Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
    - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
  - Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
    - EAT = 0.99 x 100 + 0.01 x 200 = 101ns
Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **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
  - Or use **page-table length register (PTLR)**
- Any violations result in a trap to the kernel
Valid (v) or Invalid (i) Bit In A Page Table
Memory Management Schemes

• Segmentation
• Paging

• Address Translation Mechanisms

Desirable Features:
• Very large address space
• Ability to execute partially loaded programs
• Dynamic Relocatability
• Sharing
• Protection
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space
Shared Code
Shared Pages Example

![Diagram showing shared pages for processes P₁, P₂, and P₃. Each process has a page table with entries for ed 1, ed 2, ed 3, and data.]
Structure of the Page Table

• Memory structures for paging can get huge using straight-forward methods
  • Consider a 32-bit logical address space as on modern computers
  • Page size of 4 KB ($2^{12}$)
  • Page table would have 1 million entries ($2^{32} / 2^{12}$)
  • If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    • That amount of memory used to cost a lot
    • Don’t want to allocate that contiguously in main memory

• Hierarchical Paging
• Hashed Page Tables
• Inverted Page Tables
Hierarchical Page Tables

• Break up the logical address space into multiple page tables
• A simple technique is a two-level page table
• We then page the page table
Two-Level Page-Table Scheme
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits

- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset

- Thus, a logical address is:

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

  - where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the inner page table
  - Known as **forward-mapped page table**
Address-Translation Scheme

[Diagram of address-translation scheme]

logical address
\[ p_1 \quad p_2 \quad d \]

outer page table

page of page table

\[ p_1 \]

\[ p_2 \]

\[ d \]
64-bit Logical Address Space

• Even two-level paging scheme not sufficient
• If page size is 4 KB ($2^{12}$)
  • Then page table has $2^{52}$ entries
  • If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  • Address would look like

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$d$</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

• Outer page table has $2^{42}$ entries or $2^{44}$ bytes
• One solution is to add a 2$^{nd}$ outer page table
• But in the following example the 2$^{nd}$ outer page table is still $2^{34}$ bytes in size
  • And possibly 4 memory access to get to one physical memory location
# Three-level Paging Scheme

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$d$</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd outer page</th>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>$d$</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
Hashed Page Tables

• Common in address spaces > 32 bits
• The virtual page number is hashed into a page table
  • This page table contains a chain of elements hashing to the same location
• Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
• Virtual page numbers are compared in this chain searching for a match
  • If a match is found, the corresponding physical frame is extracted
• Variation for 64-bit addresses is clustered page tables
  • Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  • Especially useful for sparse address spaces (where memory references are non-contiguous and scattered)
Hashed Page Table
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address
Inverted Page Table Architecture
Example: The Intel 32 and 64-bit Architectures

• Dominant industry chips

• Pentium CPUs are 32-bit and called IA-32 architecture

• Current Intel CPUs are 64-bit and called IA-64 architecture

• Many variations in the chips, cover the main ideas here
Example: The Intel IA-32 Architecture

• Supports both segmentation and segmentation with paging
  • Each segment can be 4 GB
  • Up to 16 K segments per process
  • Divided into two partitions
    • First partition of up to 8 K segments are private to process (kept in local descriptor table (LDT))
    • Second partition of up to 8K segments shared among all processes (kept in global descriptor table (GDT))
Example: The Intel IA-32 Architecture (Cont.)

• CPU generates logical address
  • Selector 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
  • Pages sizes can be 4 KB or 4 MB
Logical to Physical Address Translation in IA-32

CPU → segmentation unit → paging unit → physical memory

<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>
Intel IA-32 Segmentation

logical address

selector  offset

descriptor table

segment descriptor

32-bit linear address
Intel IA-32 Paging Architecture
32-bit address limits led Intel to create **page address extension (PAE)**, allowing 32-bit apps access to more than 4GB of memory space.

- Paging went to a 3-level scheme
- Top two bits refer to a **page directory pointer table**
- Page-directory and page-table entries moved to 64-bits in size
- Net effect is increasing address space to 36 bits – 64GB of physical memory
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits

<table>
<thead>
<tr>
<th>unused</th>
<th>page map</th>
<th>page directory</th>
<th>page directory</th>
<th>page table</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>48</td>
<td>47</td>
<td>39 38</td>
<td>30 29</td>
<td>21 20</td>
</tr>
</tbody>
</table>