Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging
Background

- Program must be brought into memory and placed within a process for it to be run
- Input queue -- collection of processes on the disk that are waiting to be brought into memory to run the program
- User programs go through several steps before being run

Memory Management Principles

- Memory is central to the operation of a modern computer system
- Memory is a large array of words/bytes
- CPU fetches instructions from memory according to the value of the program counter
- Instructions may cause additional loading from and storing to specific memory addresses
Some basic concepts

- Need LARGE address space
- Ability to execute partially loaded programs
- Dynamic relocatability
- Support for Sharing

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*).
**Logical Address Space**

- Operating system
- Process

**Address Protection with Base and Limit Registers**

- CPU
- Address
- Base
- Base + Limit
- Trap to operating system monitor—addressing error
- Memory
Address Binding

- Addresses in source programs are symbolic.
- Compiler binds symbolic to relocatable addresses.
- Linkage editor/loader binds relocatable addresses to absolute addresses.

Binding can be done at any step:
- i.e., compiler may generate absolute code (as for MS-DOS .COM programs).

Multistep Processing of a User Program
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 vs. Physical Address Space

- Address generated by CPU is called a logical address
- Memory unit deals with physical addresses
- compile-time and load-time address-binding:
  - Logical and physical addresses are identical
- execution-time address-binding:
  - Logical addresses are different from physical addresses
  - Logical addresses are also called virtual addresses
  - Run-time mapping from virtual to physical addresses is done by Memory Management Unit (MMU) – a hardware device
- The concept of a logical address space that is bound to a different physical address space is central to Memory Management!
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.
  - The MMU is part of the processor
  - Re-programming the MMU is a privileged operation that can only be performed in privileged (kernel) mode

- 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.
Dynamic relocation using a relocation register

- CPU
  - Logical address: 642
- MMU
  - Relocation register: 7000
  - Physical address: 7642
- Memory

Dynamic relocation using a relocation register

- CPU
  - Logical address: 346
- MMU
  - Relocation register: 14000
  - Physical address: 14346
- Memory
Dynamic relocation using a relocation register

Dynamic Loading

- A routine is not loaded until it is called
- All routines are kept on disk in a relocatable load format
- When a routine calls another routine:
  - It checks, whether the other routine has been loaded
  - If not, it calls the relocatable linking loader to load desired routine
  - Loader updates program’s address tables to reflect change
  - Control is passed to newly loaded routine
- Better memory-space utilization
  - Unused routines are never loaded
- No special OS support required
Dynamic Linking

- Similar to dynamic loading:
  - Rather than loading being postponed until run time, linking is postponed
  - Dynamic libraries are not statically attached to a program’s object modules (only a small stub is attached)
  - The stub indicates how to call (load) the appropriate library routine
- All programs may use the same copy of a library (code) (shared libraries - .DLLs)
- Dynamic linking requires operating system support
  - OS is the only instance which may locate a library in another process’s address space

Memory Allocation Schemes

- Main memory must accommodate OS + user processes
  - OS needs to be protected from changes by user processes
  - User processes must be protected from each other
- Single partition allocation:
  - User processes occupy a single memory partition
  - Protection can be implemented by limit and relocation register (OS in low memory, user processes in high memory, see below)
Memory Allocation Schemes (contd.)

- Multiple-Partition Allocation
  - Multiple processes should reside in memory simultaneously
  - Memory can be divided in multiple partitions (fixed vs. variable size)
  - Problem: What is the optimal partition size?

- Dynamic storage allocation problem
  - Multiple partitions with holes in between
  - Memory requests are satisfied from the set of holes

- Which hole to select?
  - First-fit: allocate the first hole that is big enough
  - Best-fit: allocate the smallest hole that is big enough
  - Worst-fit: allocate the largest hole (produces largest leftover hole)
  - First-fit & best-fit are better than worst-fit (time & storage-wise)
  - First-fit is generally faster than best-fit

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
### Overlays

- Size of program and data may exceed size of memory
- Concept:
  - Separate program in modules
  - Load modules alternatively
  - Overlay driver locates modules on disk
  - Overlay modules are kept as absolute memory images
  - Compiler support required

**Example:** multi-pass compiler

![Diagram of Overlays]

### Segmentation

- What is the programmer’s view of memory?
  - Collection of variable-sized segments (text, data, stack, subroutines, …)
  - No necessary ordering among segments
  - Logical address: \(<\text{segment-number}, \text{offset}>\)
- Hardware:
  - Segment table containing base address and limit for each segment

![Diagram of Segmentation]

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

### 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** – 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)
### Contiguous Allocation

- 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
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register
Hardware Support for Relocation and Limit Registers

Contiguous Allocation (Cont.)

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

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into blocks of same size called pages.
- Keep track of all free frames
- 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
- Internal fragmentation

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

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$d$</td>
</tr>
<tr>
<td>$m - n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
Paging Hardware

CPU

logical address

physical address

p d

f d

page table

physical memory

10000 ... 0000

11111 ... 1111

Paging Example

frame number

page table

logical memory

page 0

page 1

page 2

page 3

0 1

1 4

2 3

3 7

page 0

page 1

page 2

page 3

physical memory
### Paging Example

#### Page Table

<table>
<thead>
<tr>
<th>Global Memory</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>j</td>
</tr>
<tr>
<td>10</td>
<td>k</td>
</tr>
<tr>
<td>11</td>
<td>l</td>
</tr>
<tr>
<td>12</td>
<td>m</td>
</tr>
<tr>
<td>13</td>
<td>n</td>
</tr>
<tr>
<td>14</td>
<td>o</td>
</tr>
<tr>
<td>15</td>
<td>p</td>
</tr>
</tbody>
</table>

**Page Table**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

### Free Frames

#### Free Frame List

<table>
<thead>
<tr>
<th>Free-frame List</th>
<th>Free-frame List</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>13</td>
<td>13 page 1</td>
</tr>
<tr>
<td>18</td>
<td>14 page 0</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

#### New Process Page Table

<table>
<thead>
<tr>
<th>New Process Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New Process Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
</tbody>
</table>
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-look-up hardware cache called associative memory or translation look-aside buffers (TLBs)

Associative Memory

- Associative memory – parallel search

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

Address translation (A′, A″)
- If A′ is in associative register, get frame # out
- Otherwise get frame # from page table in memory
Paging Hardware With TLB

Effective Access Time

- Associative Lookup = \( \varepsilon \) time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers
- Hit ratio = \( \alpha \)
- Effective Access Time (EAT)

\[
EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha
\]
Memory Protection

- 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.

---

Valid (v) or Invalid (i) Bit In A Page Table

![Diagram showing valid-invalid bit in a page table]

- The diagram illustrates how valid-invalid bits are associated with each page in a page table.
Page Table Structure

- 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
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

<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>( d )</td>
<td></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 outer page table.

Two-Level Page-Table Scheme
Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

VAX address Architecture

<table>
<thead>
<tr>
<th>section</th>
<th>page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>21</td>
<td>9</td>
</tr>
</tbody>
</table>
### 64-bit Address Space with Two Level Paging

<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>

### Three Level Paging

<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>

---

Operating System Concepts

8.51

Operating System Concepts

8.52
**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.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
Inverted Page Table

- 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

Inverted Page Table Architecture

[Diagram showing the architecture of an inverted page table]
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes

- **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 Pages Example

```
<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ed 1</td>
<td>ed 1</td>
<td>ed 1</td>
</tr>
<tr>
<td>ed 2</td>
<td>ed 2</td>
<td>ed 2</td>
</tr>
<tr>
<td>ed 3</td>
<td>ed 3</td>
<td>ed 3</td>
</tr>
<tr>
<td>data 1</td>
<td>data 2</td>
<td>data 3</td>
</tr>
</tbody>
</table>
```

Page table for P1:
```
<table>
<thead>
<tr>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ed 1</td>
<td>ed 2</td>
<td>ed 3</td>
<td>data 1</td>
</tr>
</tbody>
</table>
```

Page table for P2:
```
<table>
<thead>
<tr>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ed 1</td>
<td>ed 2</td>
<td>ed 3</td>
<td>data 2</td>
</tr>
</tbody>
</table>
```

Page table for P3:
```
<table>
<thead>
<tr>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ed 1</td>
<td>ed 2</td>
<td>ed 3</td>
<td>data 3</td>
</tr>
</tbody>
</table>
```
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

1
2
3
4

physical memory space

1
2
3
4

Logical address consists of a two tuple:
<segment-number, 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.)

- **Relocation.**
  - dynamic
  - by segment table

- **Sharing.**
  - shared segments
  - same segment number

- **Allocation.**
  - first fit/best fit
  - external fragmentation

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
Segmentation Hardware

Example of Segmentation
The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.

Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a page table for this segment.
MULTICS Address Translation Scheme

Example – Intel Pentium

- CPU generates a logical address given to the Segmentation Unit
- Segmentation Unit produces a linear address which is given to the paging unit
- Paging unit generates a physical address
Pentium Segmentation

- Pentium Segmentation
  - Max Segment Size – 4 GB (32 bit address)
  - Number of Segments – 16K (14 bit address)
    - 8K private - Kept in Logical Descriptor Table
    - 8K Shared – Kept in Global Descriptor Table
  - Two bit field – p – deals with protection

<table>
<thead>
<tr>
<th>s</th>
<th>g</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
### Pentium Paging

- Page Size can be 4 KB (12 bits) or 4 MB (22 Bits)
- For 4 KB pages, uses two level paging scheme

<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>
Segmentation with Paging – Intel 386

As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

Intel Address Translation
Linux on Intel

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
  - Kernel code
  - Kernel data
  - User code (shared by all user processes, using logical addresses)
  - User data (likewise shared)
  - Task-state (per-process hardware context)
  - LDT
- Uses 2 protection levels:
  - Kernel mode
  - User mode

Three-level paging model in Linux

<table>
<thead>
<tr>
<th>global directory</th>
<th>middle directory</th>
<th>page table</th>
<th>offset</th>
</tr>
</thead>
</table>

Operating System Concepts 8.77
Three-level Paging in Linux

(global address)

CPR register

global directory

global directory entry

middle directory

middle directory entry

page table

page table entry

page frame