Memory Management
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
Address binding of instructions and data to memory addresses can happen at three different stages:

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

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

- **Execution time**: Binding delayed until runtime 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

![Diagram of multistep processing of a user program]

- Source program
- Compiler or assembler
- Object module
- Other object modules
- Linkage editor
- Load module
- System library
- Dynamically loaded system library
- Dynamic linking
- In-memory binary memory image
- Loader
- Execution time (run time)
- Compile time
- Load time
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 the execution-time address-binding scheme
Memory-Management Unit (MMU)

- Hardware device that maps virtual/logical to physical address
- The MMU consists of one or more “relocation registers”
- The value in an MMU relocation register is added to every address generated by a user process at the time it is sent to memory
- As such, the user program deals with logical addresses; it never sees the real physical addresses
Dynamic relocation using a relocation register

![Diagram showing dynamic relocation using a relocation register]

- CPU
  - Logical address: 346
- MMU
  - Relocation register: 14000
  - Physical address: 14346
- Memory
Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; an 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

- 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 needed to check if routine is in processes’ memory
- Dynamic linking is particularly useful for libraries
Swapping

- A process can be swapped temporarily out of memory to a backing store (e.g. disk), 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).
Schematic View of Swapping

1. Swap out
2. Swap in
Contiguous Allocation

- 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

- 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
A base and a limit register define a logical address space.
HW address protection with base and limit registers

![Flowchart](attachment:flowchart.png)
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
Address Translation Architecture
Paging Example

The image shows a diagram of a paging example with a page table and frame number association. The page table is labeled as follows:

- Page 0
- Page 1
- Page 2
- Page 3

The frame number column maps to:

- Frame 0: Page 0
- Frame 1: Page 1
- Frame 2: Page 2
- Frame 3: Page 3

This diagram is used to illustrate how pages are mapped to physical memory in a paging system.
Free Frames

(a) Before allocation
- Free-frame list: 14, 13, 18, 20, 15
- New process
- Page 0
- Page 1
- Page 2
- Page 3

(b) After allocation
- Free-frame list: 15
- New process
- Page 0
- Page 1
- Page 2
- Page 3
- New-process page table
- 0: 14
- 1: 13
- 2: 18
- 3: 20
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.
• This two-memory access problem can be solved by the use of a special, fast-lookup, 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>
<tr>
<td></td>
<td></td>
</tr>
<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 units
- Assume memory cycle time is $\tau$ time units
- Hit ratio $\alpha$ – fraction of time that a page number is found in the associative registers; ratio related to number of associative registers
- Assume simultaneous query of TLB and page table entry; cancel read of page table entry if TLB hit

**Effective Access Time (EAT)**

$$t_{\text{eff}} = (\tau + \varepsilon) \alpha + 2\tau(1 - \alpha)$$

$$= \alpha\tau + \alpha\varepsilon + 2\tau - 2\alpha\tau$$

$$= 2\tau - \alpha\tau + \alpha\varepsilon$$

$$= (2 - \alpha + \alpha\varepsilon/\tau) \tau$$

- Typical value for $\varepsilon/\tau$ is 1/5
- Look at limiting cases
  - if $\alpha = 0$, $t_{\text{eff}} = 2\tau$
  - if $\alpha = 1$, $t_{\text{eff}} = 1.2\tau$
Memory Protection

- Memory protection implemented by associating a 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

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Frame Number</th>
<th>Valid-Invalid Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>page 0</td>
<td>0</td>
<td>v</td>
</tr>
<tr>
<td>page 1</td>
<td>1</td>
<td>v</td>
</tr>
<tr>
<td>page 2</td>
<td>2</td>
<td>v</td>
</tr>
<tr>
<td>page 3</td>
<td>3</td>
<td>v</td>
</tr>
<tr>
<td>page 4</td>
<td>4</td>
<td>v</td>
</tr>
<tr>
<td>page 5</td>
<td>5</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>i</td>
</tr>
</tbody>
</table>

Frame numbers range from 0 to 9, with pages 0 to 5 specifically shown.
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 a 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>$p_i$</td>
<td>$p_2$</td>
</tr>
</tbody>
</table>

10 10 12

where $p_i$ 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

![Diagram of address translation scheme](image.png)
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.
Hashed Page Table

Diagram showing the process of mapping a logical address to a physical address through a hash table.
Inverted Page Table

- One entry for each real page (i.e. frame) 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

The diagram illustrates the process of mapping a logical address to a physical address using an inverted page table. The logical address is processed in the CPU, which then queries the page table for the corresponding physical address. The diagram shows the flow of data from CPU to page table to physical memory, highlighting the inverted structure where the page index (i) is used to look up the page directory (d), and then the page directory is used to find the physical page (p).
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

![Diagram](image-url)
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

Diagram:
- subroutine
- stack
- symbol table
- main program
- logical address

CIS 415, Spring 2016 Memory Management
Logical View of Segmentation

user space

physical memory space
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.)

- **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
Address Translation Architecture

CPU -> s d

- Segment Table:
  - Limit
  - Base

- Comparison:
  - <
    - Yes
    - +
    - Physical Memory
  - No
    - Trap: Addressing Error
Example of Segmentation

![Diagram showing segmentation in memory management with segments 0 to 4 and their respective limits and bases.](image)
Sharing of Segments

- Editor
- Segment 0
- Data 1
- Segment 1
- Logical memory process $P_1$

- Editor
- Segment 0
- Data 2
- Segment 1
- Logical memory process $P_2$

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>25286</td>
<td>43062</td>
</tr>
<tr>
<td>4425</td>
<td>68348</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>25286</td>
<td>43062</td>
</tr>
<tr>
<td>8850</td>
<td>90003</td>
</tr>
</tbody>
</table>

Physical memory

- Editor
- Data 1
- Data 2
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
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 30386 Address Translation

The diagram illustrates the address translation process for the Intel 30386 processor. It starts with a logical address, which consists of a selector and an offset. This logical address is used to access the descriptor table, where segment descriptors are stored. The segment descriptor is used to determine the page frame, which contains the physical address of the requested memory location.

The address translation process involves several steps:
1. The selector is used to look up the segment descriptor in the descriptor table.
2. The segment descriptor contains the base address and the page size.
3. The linear address is constructed by combining the selector, offset, and segment descriptor information.
4. The linear address is then used to access the page directory.
5. The page directory entry contains the base address of the page table.
6. The page table entry contains the physical address of the memory location.
7. The physical address is combined with the offset to get the final memory address.