Chapter 8: Memory Management
Chapter 8: Memory Management

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

- Examine **basic** (not virtual) **memory management**
  - Swapping
  - Contiguous allocation
  - Paging
  - Segmentation

- Prepare for study of **virtual memory**

- Discuss what **CAN** be done without virtual memory
Program **must be brought into memory** and placed within a process for it to be run.

- Main memory and registers are only storage CPU can access directly
- Can access registers in one clock cycle or less
- Accessing main memory may take several cycles
- Cache(s) sits between main memory and CPU registers

Main memory is of a **finite size**

- In a multiprogramming environment, processes must **SHARE** space in main memory

**Input queue** – **(job 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.
User (or system) usually **does not know** where the data that makes up a program eventually **will** reside in memory.

- Assume it **logically resides contiguously**, starting at address 0 (although some code / situations **require** absolute addressing)
Address binding (mapping from logical to physical addresses) 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. *(absolute addressing)*
  - If do as described above, code can run at only one spot in memory
    - Required for some low level OS code, but little else
  - Instead, usually compile as if will run at address 0, and bind later

- **Load time:** Must generate *relocatable code* if memory location is not known at compile time. *(relative addressing)*
  - Can load code in different places in memory and it will run.
  - Loader handles binding (linker will have resolved addresses)

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

- **Source Program**
  - **Compiler or Assembler**
  - **Object Module**
    - **Linkage Editor**
      - **Load Module**
        - **System Library**
          - **Loader**
            - **In-Memory Binary Memory Image**
              - **Dynamic Linking**

Timelines:
- **Compile Time**
- **Load Time**
- **Execution Time (Run 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 (HW)

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 virtual 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 referenced.

- The user program deals with logical addresses; it never sees the real physical addresses.
  - This is similar in concept to the base and limit registers, discussed in Ch. 2
  - Technically, base and limit registers usually refer to real addresses, but their use is so similar to that of a relocation register, that they are often discussed as if they were the same thing.
Dynamic relocation using a relocation register

Logical address: 346
Relocation register: 14000
Physical address: 14346

CPU -> MMU -> memory
Of course, if you write (or have a good compiler that generates) relocatable code, then much of this is unnecessary

(Do you remember what relocatable code is ???)
BUT – What if the Program's too BIG?

Three possibilities (what was old is new again)

■ **Dynamic loading**
  - Keep functions/procedures/methods as separate, loadable routines
  - Only load a routine when it is needed (*for user code*)

■ **Dynamic linking**
  - Not often supported
  - Only link (resolve addressing of) system library routines when called (*for system code*)

■ **Overlays**
  - Quite common when memory constrained (e.g., in the past)
  - Separate the “common” portions of program from portions executed at different times
  - After one part of program is executed, load another part into that same space, and **OVERLAY** the first
  - Much work for programmer – system does not help you – tools may
Dynamic Loading – More Details

- **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 – More Details

- 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 needs to check if routine is in processes’ memory address space

- Dynamic linking is particularly useful for libraries
  - System also known as shared libraries
Overlays

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

- Useful when process is larger than amount of memory allocated to it and paging is not possible / supported

- **Implemented by user**, no special support needed from operating system
  - Programming design of overlay structure is complex
  - Good tool support can help

- Necessary in some new technology/architectures, too, when they have constrained resources – e.g., cell processor
Overlays for a Two-Pass Assembler
BUT – what if the program is *still* too big ???

OR – if you are in a multiprogramming environment and can't get enough code of any program in main memory to run (when have pieces of all processes in memory)

**One way to solve is SWAPPING**

(a memory management technique, not multiprogramming technique, used to change the mix of jobs)
Swapping

- A process can be **swapped** temporarily out of memory to a **backing store**, and then brought back into memory for continued execution.
  - Sometimes called “swap space,” “swap partition,” “swap disk”

- **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 data swapped.

- Modified versions of swapping are **found on many systems**, i.e., UNIX, Linux, and Windows.
More on Swapping

- **Use variants of CPU scheduling algorithms** (RR, priority, etc.) to decide which processes swap out

- A job can be swapped out and still be on the Ready Queue
  - So now, **can have more jobs on Ready Queue than will fit in main memory**
  - Must swap a job **IN** if not already in main store when dispatched
  - If late binding, can swap back in anywhere
  - If compile/link time binding, must swap back in to same spot

- **NOTE** – **swapping is not cheap** (in terms of time)

- Is also important to consider whether swap time involves **synchronous or asynchronous I/O**
Schematic View of Swapping

1. swap out
2. swap in
Memory Partitioning

- How do we manage having many jobs and OS all swapping at once
  - **PARTITION MAIN MEMORY**
    - “Partition” just means divide it up somehow – not disk partition, etc.

- No partitions – dedicated system

- Two partitions – system and user
  - Separate with *fence register*
    - Resident OS code and interrupt vector in low memory
    - User processes in high memory
  - Use base/limit registers, assume contiguous allocation

- Multiple / variable-sized partitions
  - Allows loading in extra operating system code
  - Must be able to relocate programs if needed
  - May use system overlays
  - Sometimes build system area up, user area down
  - Also use base/limit registers & relocation register
Review of Base and Limit Registers

■ Purpose
  ● To protect user processes from each other, and from changing operating system code and data.

■ Technique
  ● Base register (sometimes called relocation register) contains value of smallest physical address process can use
  ● Limit register contains range of logical addresses –
    ➢ Each logical address must be less than the limit register
    ➢ Each physical address must be less than the base register + limit register – 1
Use of Base and Limit Registers

![Diagram showing the use of base and limit registers in memory management. The diagram illustrates how memory is divided into regions for the operating system and processes, with base and limit registers indicating the boundaries of each process's memory area.](image-url)
Hardware Support for **Base** and Limit Registers

NOTE: technically applies to **physical / real addresses**

Check **base first**, then validate that within limit
Hardware Support for Relocation and Limit Registers

NOTE: uses logical / virtual addresses

Validate limit first, then add relocation value
Multiple-partition allocation

- Operating system maintains information about:
  a) allocated partitions  
  b) free partitions (hole)

- **Hole** – block of available memory; holes of various size are scattered throughout memory.
  - Holes and partitions are various sizes

- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - OS must make decisions and keep track of all of this
Dynamic Storage-Allocation Problem

Once a job is scheduled and needs to be loaded, the OS must decide \textbf{where to allocate} a request of size $n$ from a list of free holes.

- **First-fit:** Allocate the \textit{first} hole that is big enough.
  - Usually fastest, but not always best use of space

- **Best-fit:** Allocate the \textit{smallest} hole that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole.

- **Worst-fit:** Allocate the \textit{largest} hole
  - Must also search entire list
  - Produces the largest leftover hole.
  - Both first-fit and best-fit often better utilization & speed

**With / Without Skip** – if process selected cannot fit in a free partition, should the OS select a new process from the queue or wait?
PROBLEM: There is almost never an exact fit. This results in fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - “Holes” **between** allocated areas
  - Leftover, available holes are too small to use

- **Internal Fragmentation** – memory is usually allocated in standard sizes (e.g., word boundaries, powers of two, etc.).
  - Allocated memory may be slightly larger than requested memory
  - Wasted / unused space **within** allocated area
External Fragmentation

Parts of memory are unallocated, but are too small to use
Partitions must be multiples of a set size (page size), so wasted / unused space within allocated area.
Reducing Fragmentation

- **Long term scheduler could help**
  - Long term scheduler could order jobs according to block size
  - BUT this may produce results contrary to requirements of other scheduling algorithms.

- **Compaction**
  - Like “garbage collection” routines
  - **Move code blocks together**, to get rid of small blocks of free space and create large ones
  - NOTE, however – this **requires time ! ! !**
  - Also, it can only be done if mapping from logical to physical addresses at run time (e.g., using base/limit registers, etc.)
  - Then, there **can also be I/O-related problems / concerns**
    - Must have job in memory while it is involved in I/O.
    - Do I/O only into OS buffers (not into job’s buffers)
Compaction Example 1

Compaction Example 1
Compaction Example 2

Comparison of some different methods of compaction

Original allocation

Moved 600K

Moved 400K

Moved 200K
Multiple Base Registers

- It would be easier to manage memory if we could break up a program into several sections and put each section into a different memory partition
  
  - **Separate code from data,** so only have one copy of code and many of data (if reentrant)
  
  - Fit into memory more easily; sometimes use less memory
  
  - Often **reduce external fragmentation**
Paging

■ PROBLEM: variable size partitions cause fragmentation

■ SOLUTION: divide physical memory into equal sized partitions called frames (size is power of 2, between ½K and 8K)

- Divide logical memory (programs) into blocks of same size called pages.
- Keep track of all free frames in memory
- To run a program of size $n$ pages, need to find $n$ free frames ANYWHERE (can be non-contiguous) and load program.
- Set up a page table to translate logical to physical addresses.
  - Two parts: page number and page offset
- Requires special hardware !!!
- Internal fragmentation within pages is still possible
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.
    - Easy to calculate, since can just shift right and strip off least significant bits
  - **Page offset** \((d)\) – combined with base address of page to define the physical memory address that is sent to the memory unit.
    - Offset into the page where the data is to be found
    - Can just mask off most significant bits to get this – simple/quick

<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>
Page Table Address Translation

Diagram showing the process of address translation in a computer system, involving logical addresses, page tables, and physical memory. The diagram illustrates how logical addresses are translated into physical addresses through the use of a page table.
Paging Example

You think your code looks like this

But it really looks like this in main memory

Can have one page table per job, or move data around in page table
Most common to have a separate page table for each process
32-byte memory and 4-byte pages
Free Frames

(a) Before allocation

(b) After allocation
Implementation of Page Table

■ Would like to keep page table in registers -- fastest
  ● But only practical if very small – e.g., only for 1 process

■ Instead, often keep page table in main memory.
  ● \textit{Page-table base register} (PTBR) points to the page table.
  ● \textit{Page-table length register} (PRLR) tells 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.
  ● \textbf{Slow} – \textit{Effective Access Time} (E.A.T.) is at least twice memory access time

■ The two memory access problem can be solved by the use of a special fast-lookup hardware cache called \textit{associative memory} or \textit{translation look-aside buffers (TLBs)}
Address translation of “A”

To improve address translation performance, first look at translation lookaside buffer

- If A is in associative register, get frame # back immediately.
- Otherwise have to go to page table in memory to get frame #
Other Page Table Options

Hybrid approach: BOTH memory and registers (most common)

- **Associative Registers -or- Translation Look-Aside Buffer (TLB)**
  - Holds small portion of table -- most recently referenced pages
  - Very fast, but limited size (typically 64-1024 entries)

- **When reference a page number**
  - First search TLB
    - If “hit,” get frame number, and can quit
    - If no hit, search page table in memory, and **when find page, move entry for it to Look Aside Buffer**
  - A good hit ration is > 95%
    - (on AS/400 / iSeries, the hit ratio is > 97%-98%)
  - Some TLBs store **Address-space Identifiers (ASIDs)** in TLB
    - ASID identifies process
    - Allows system to store entries for multiple processes in TLB
Paging Hardware With TLB

CPU

Logical address

p d

Page frame number

TLB hit

f d

Physical address

Physical memory

TLB miss

Page table

p

f

Silberschatz, Galvin and Gagne ©2009

Operating System Concepts
Effective Access Time

**EAT** is how long it takes to resolve an address

- **Associative Lookup time** (required for TLB) = $\varepsilon$ time unit
- Assume **memory cycle time** is 1 microsecond
- **Hit ratio** – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.
  - Hit ratio = $\alpha$

**Effective Access Time (EAT)**

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

- $\alpha$ % of the time, it requires a lookup in TLB + 1 memory access (i.e., the data)
- The rest of the time (1-$\alpha$ % of the time), it requires a TLB lookup + 2 memory accesses (i.e., the page table and then the data)
Other Page Table Questions

■ What happens if the page table gets too large for memory

- Many current systems have large logical address spaces: $2^{32}$ or $2^{64}$
- If $2^{32}$, with page table for each task and 4K pages – would take 4MB for page table for each process
  - Remember, have a page table for each process
- Since systems can have thousands of processes, could fill memory with just page tables

■ SOLUTION: page the page table
Hierarchical Page Tables

- Break up the logical address space into multiple page tables.

- A simple technique is a two-level page table.
  - Partition the page table by High Order part of address
  - 1\textsuperscript{st} level – references to partitions
  - 2\textsuperscript{nd} level – partitions (and pages within them)
  - Partitions (2\textsuperscript{nd} level) can be paged
  - Acceptable for 32-bit architectures
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>
</tbody>
</table>

| 10   | 10   | 12   |

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 for a two-level 32-bit paging architecture
Other Multi-Level Page Tables

- Memory impacts of larger address space size
  - Would require 16GB to store just the 1st level of a 2-level page table for 2^64 bit addressing

- 3- and 4-level page tables
  - Significant performance impacts
  - Expand scheme further

- Also, consider other alternatives
## Three- and Four-Level Paging Schemes

<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.
  - Traversing the chain (presuming it is in memory) is much quicker than multiple I/Os

- 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

Hash ‘p’, then traverse chain to find ‘p’, then read its associative value, ‘r’, and use that as the page number.
Ordinarily have a page table for each process – natural
- Problem: many page tables are mostly empty

Solution: Inverted Page Table
- Single page table, single entry for each real page
- Entry expanded to consist of
  - Virtual address of page
  - Process ID of owning process
- Hash virtual address to find Inverted Page Table entry
  - Use associative registers (TLB) to speed up
- Significantly decreases memory needed to store each page table
  - But possibly slight increase in time needed to search table when a page reference occurs.
Inverted Page Table Architecture

Inverted Page Table Architecture
Another advantage of paging

- **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 its own private code (if any) and data.
  - The pages for the private code and data can appear anywhere in the logical address space.
Shared Pages Example

Process $P_1$
- ed 1
- ed 2
- ed 3
- data 1

Page Table for $P_1$
- 3
- 4
- 6
- 1

Process $P_2$
- ed 1
- ed 2
- ed 3
- data 2

Page Table for $P_2$
- 3
- 4
- 6
- 7

Process $P_3$
- ed 1
- ed 2
- ed 3
- data 3

Page Table for $P_3$
- 3
- 4
- 6
- 2

Data
- data 1
- data 3
- ed 1
- ed 2
- ed 3
- data 2
- data 3
- data 1
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

- In some respects, this is similar to irregularly sized partitions, although in this case, the partitions are usually made up of equal sized pages.
User’s View of a Program

- subroutine
- stack
- symbol table
- main program

logical address

Sqrt
Logical View of Segmentation

user space

physical memory space
Segmentation Architecture

- 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 $s < STLR$
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:
  - Valid 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

CPU → s d

s

limit base

segment table

< no yes +

trap: addressing error

physical memory
Example of Segmentation

- **Subroutine**
- **Stack**
- **Symbol Table**
- **Main Program**

Logical address space:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Limit</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>1400</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>6300</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>4300</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>3200</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>4700</td>
</tr>
</tbody>
</table>

Physical memory:

- Segment 0
- Segment 1
- Segment 2
- Segment 3
- Segment 4
Sharing of Segments

logical memory
process $P_1$

logical memory
process $P_2$

editor

segment 0

data 1

segment 1

limit | base
------|------
0 25286 | 43062
1 4425 | 68348

editor

data 1

segment table
process $P_1$

segment 0

data 2

segment table
process $P_2$

limit | base
------|------
0 25286 | 43062
1 8850 | 90003

physical memory

68348
72773
90003
98553
Another Option Using Segments

- Each process can be assigned its own logical segment of virtual addresses
  - It can only address memory within this segment
  - Protects other processes and system code without relying on conventional base/limit address validation
  - Makes communication with other processes and system code somewhat challenging
Example: The Intel Pentium

- 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
Logical to Physical Address Translation in Pentium

```
| CPU | logical address | segmentation unit | linear address | paging unit | physical address | physical memory |
```

- **Page Number:**
  - $p_1 = 10$
  - $p_2 = 10$

- **Page Offset:**
  - $d = 12$
Intel Pentium Segmentation

logical address

| selector | offset |

descriptor table

segment descriptor

32-bit linear address
Pentium Paging Architecture

(logical address)

page directory  page table  offset

31  22  21  12  11  0

page directory

page table

4-KB page

4-MB page

CR3 register

Operating System Concepts

Silberschatz, Galvin and Gagne ©2009
Linux on Intel 80x86

- 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 (logical descriptor table)

- Uses 2 protection levels:
  - Kernel mode
  - User mode
Linear Address in Linux

Broken into four parts:

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

Operating System Concepts
Three-level Paging in Linux

![Diagram of three-level paging in Linux]

- Global directory
- Middle directory
- Page table
- Offset

CR3 register

Global directory entry

Middle directory entry

Page table entry

Page frame

(linear address)