Chapter 8: Memory Management
Chapter 8: 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 or 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
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
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.
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 sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
Dynamic relocation using a relocation register

![Diagram of CPU, MMU, and memory with logical and physical address calculations]

- CPU sends logical address 346 to the MMU.
- The relocation register contains the value 14000.
- The MMU calculates the physical address as 14346, which is then sent to memory.
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).
Schematic View of Swapping

1. swap out
2. swap in
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
Transfer of a Paged Memory to Contiguous Disk Space

- Program A
- Program B
- Main Memory

Swap out: 0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23

Swap in:
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault
Page Table When Some Pages Are Not in Main Memory
Steps in Handling a Page Fault

1. Reference
2. Trap
3. Page is on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

- Operating system
- Page table
- Free frame
- Physical memory
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times
Memory Allocations
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

- Each (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

CPU

- address

- \( \geq \)
- no

- \( < \)
- no

- trap to operating system monitor—addressing error

- base

- base + limit

- yes

- yes

- memory
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)

![Diagram showing memory allocation and processes]
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
Example

Given free memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)?

Which algorithm makes the most efficient use of memory?
Fragmentation

- **External Fragmentation** – Gaps between allocated contiguous memory - 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

CPU

logical address

physical address

f0000 ... 0000

f1111 ... 1111

physical memory

page table

V

V

V

f
Paging Example

logical memory

page table

0 1
1 4
2 3
3 7

frame number

0 0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory
Paging Example
Free Frames

Before allocation

After allocation

Free Frames list

Free Frames list

page 0
page 1
page 2
page 3
new process

page 0
page 1
page 2
page 3
new process

new-process page table

(a) (b)
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-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>
<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

- CPU
- Logical address
- Page frame number
- TLB
- TLB hit
- TLB miss
- Page table
- Physical memory

Associate Memory
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; ratio related to number of associative registers
- Hit ratio = $\alpha$
- **Effective Access Time (EAT)**

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

1. Swap out
2. Swap in
Virtual Memory

- Elusion to expand memory view onto the secondary storage.
- Moving pages – frames between memory and HD
- Demand paging
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
Transfer of a Paged Memory to Contiguous Disk Space

Diagram showing the transfer of memory pages from main memory to disk space. The diagram includes:
- Program A
- Program B
- Main memory
- Disk space

Arrows indicate the swap out and swap in operations.
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (1⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:

    | Frame # | valid-invalid bit |
    |---------|-------------------|
    | 1       | 1                 |
    | 1       | 1                 |
    | 1       | 1                 |
    | 1       | 1                 |
    | 0       | 0                 |
    | ...     |                   |
    | 0       | 0                 |
    | 0       |                   |

- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault
Page Table When Some Pages Are Not in Main Memory
Steps in Handling a Page Fault

1. trap
2. page is on backing store
3. load M
4. bring in missing page
5. reset page table
6. restart instruction

Operating System Concepts
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times
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_i )</td>
<td>( p_2 )</td>
</tr>
</tbody>
</table>

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
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

A hashed page table uses a hash function to map logical addresses to physical addresses. The hash function takes the page number (p) and offset (d) of a logical address and produces a hash value. This hash value is then used to index into the hash table, which contains entries for each page in memory. The entry for the page in the hash table contains the physical address (r) of the page in memory. The figure illustrates the flow from a logical address to a physical address through the use of a hash table.
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
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

Page table for $P_1$

Page table for $P_2$

Page table for $P_3$
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

- subroutine
- stack
- symbol table
- main program

logical address

$\text{Sqrt}$
Logical View of Segmentation

user space

physical memory space
Stop here
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
Example of Segmentation
Sharing of Segments

Logical memory:
- Process $P_1$:
  - Segment $S_1$: Data 1
  - Segment $S_2$: Data 2
- Process $P_2$:
  - Segment $S_1$: Data 1

Segment Table:
- $P_1$:
  - Segment Table Entry 1: Limit 25286, Base 43082, Offset 68348
- $P_2$:
  - Segment Table Entry 1: Limit 25286, Base 8850, Offset 90003

Physical memory:
- Editor: 43082
- Data 1: 68348
- Data 2: 90003
- Physical memory: 98563
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

logical address -> selector + offset

descriptor table

segment descriptor

+ -> linear address

directory + page + offset

page frame

physical address

page directory

directory entry

page table

directory entry

page table entry

page directory

base register
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

- Uses 2 protection levels:
  - Kernel mode
  - User mode
End of Chapter 8