Module 9: Memory Management

- Background
- Logical versus Physical Address Space
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging
Background

- Program must be brought into memory and placed within a process for it to be executed.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being executed.
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*).
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.
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 address.
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.
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.
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 and Microsoft Windows.
Schematic View of Swapping

1. Swap out
2. Swap in
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.
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

logical memory

page table

frame number

physical memory

page 0

page 1

page 2

page 3

0  1
1  4
2  3
3  7
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 registers* or *translation look-aside buffers* (TLBs)
Associative Register

- Associative registers – 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
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.
Two-Level Page-Table Scheme
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>10</td>
<td>10</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.
Address-Translation Scheme

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

[Diagram showing the address-translation scheme with logical address, outer-page table, page of page table, and a flow from logical address to page of page table through outer-page table and page table.]
Multilevel Paging and Performance

- Since each level is stored as a separate table in memory, covering a logical address to a physical one may take four memory accesses.
- Even though time needed for one memory access is quintupled, caching permits performance to remain reasonable.
- Cache hit rate of 98 percent yields:
  \[ \text{effective access time} = 0.98 \times 120 + 0.02 \times 520 \]
  \[ = 128 \text{ nanoseconds}. \]
  which is only a 28 percent slowdown in memory access time.
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. The diagram includes:
- CPU
- Logical address
- Physical address
- Physical memory
- Page table
- pid, p, d
- i, d
- Search

The logical address is mapped to the page table, which is then used to find the physical address.
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
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,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays
Logical View of Segmentation

user space

physical memory space

1
2
3
4

1
4
2
3
Segmentation Architecture

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

- **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
• 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
Logical memory:

Process $P_1$:
- Segment 0
  - Editor
  - Data 1

Segment table for $P_1$:

<table>
<thead>
<tr>
<th>Limit</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25286</td>
</tr>
<tr>
<td>1</td>
<td>4425</td>
</tr>
</tbody>
</table>

Process $P_2$:
- Segment 0
  - Editor
  - Data 2

Segment table for $P_2$:

<table>
<thead>
<tr>
<th>Limit</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25286</td>
</tr>
<tr>
<td>1</td>
<td>8850</td>
</tr>
</tbody>
</table>
Segmentation with Paging – MULTICS

- 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

logical address

s d

STBR

segment table

segment length page–table base

≥ yes

no

d

 trap

p d’

memory

page table for segment s

f d’

physical address

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

- Hardware support
- Performance
- Fragmentation
- Relocation
- Swapping
- Sharing
- Protection
9.03

CPU

logical address 346

relocation register 14000

MMU

physical address 14346

memory
9.06

CPU

logical address

<

limit register

no

trap; addressing error

relocation register

yes

physical address

memory
9.07

<table>
<thead>
<tr>
<th>job queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
</tr>
<tr>
<td>P₁</td>
</tr>
<tr>
<td>P₂</td>
</tr>
<tr>
<td>P₃</td>
</tr>
<tr>
<td>P₄</td>
</tr>
<tr>
<td>P₅</td>
</tr>
</tbody>
</table>

Operating system:
- 0 to 400K
- 400K to 2160K
- 2160K to 2560K

Memory:
- 0 to 2560K
next request is for 18,462 bytes

\{ \text{hole of 18,464 bytes} \}

$P_7$

operating system

$P_{43}$
9.13

logical memory

page table

frame number

physical memory

page 0
page 1
page 2
page 3

0 1
1 4
2 3
3 7

0
1
2
3
4
5
6
7

page 0
page 2
page 1
page 3
9.14

logical memory

page table

physical memory

0  a
1  b
2  c
3  d
4  e
5  f
6  g
7  h
8  i
9  j
10 k
11 l
12 m
13 n
14 o
15 p

0      5
1      6
2      1
3      2

4  i
  j
  k
  l

8  m
  n
  o
  p

12
16
20  a
   b
   c
   d

24  e
   f
   g
   h

28
9.15

(a) free-frame list
14
13
18
20
15

new process
page 0
page 1
page 2
page 3

(b) 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

13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
9.16

The diagram illustrates the process of a logical address being translated into a physical address in a virtual memory system. It consists of the following components:

- **CPU**: The central processing unit, which generates the logical address.
- **Page Table**: Contains page number and frame number information.
- **TLB (Translation Lookaside Buffer)**: A cache for rapid translation of logical to physical addresses.
- **Physical Memory**: Stores the actual data.

The process flow is as follows:

1. **Logical Address**: CPU generates a logical address that includes page number (p) and frame number (d).
2. **Page Table**: The logical address is first checked in the page table. If it is a **TLB miss** (no match), the page table is consulted to obtain the frame number (f).
3. **TLB**: The frame number is then used to check the TLB. If it is a **TLB hit** (match), the physical address is derived and the data is accessed.
4. **Physical Memory**: The physical address is used to access the data stored in physical memory.

This process ensures efficient address translation and data retrieval in virtual memory systems.
9.62

process $P_1$

| ed 1 | 3 |
| ed 2 | 4 |
| ed 3 | 6 |
| data 1 | 1 |

Page table for $P_1$

process $P_2$

| ed 1 | 3 |
| ed 2 | 4 |
| ed 3 | 6 |
| data 2 | 7 |

Page table for $P_2$

process $P_3$

| ed 1 | 3 |
| ed 2 | 4 |
| ed 3 | 6 |
| data 3 | 2 |

Page table for $P_3$
9.22

logical address space

subroutine

stack

symbol table

main program

Sqrt
9.24
9.25

logical memory process $P_1$

logical memory process $P_2$

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

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

43062

68348

72773

90003

98553

Physical memory

editor

data 1

data 2