Chapter 8. Memory Management

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

1. Background

- Memory
  - A large array of words or bytes, each with its own **address**
- Program
  - Must be brought into memory and placed within a process for it to run.
  - The first address of the user process cannot be zero.
- Instruction-execution cycle:
  - Fetching an instruction from memory
  - Decoding
  - Fetching operands from memory
  - Executing
  - Storing results in memory
- Addresses are generated by instructions.
- ☺ ***Then, programmers need to know how to map addresses in source programs?***

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at four different stages:

- **Editing time**
  - Addresses in source programs are generally symbolic.
- **Compile time**
  - If memory location known a priori, ***absolute code*** can be generated; must recompile code if starting location changes.
  - Must generate ***relocatable code*** if memory location is not known at compile time; we assume in the followings that the memory location is not known.
    - Logical addresses in an object file use start from zero.
    - Object files are linked using ***symbol table*** => One logical address space.
- **Load time**
  - Final binding could be delayed until load time.
- **Execution time**
  - Need hardware support for address maps (e.g., ***relocation registers – base ...***)
    - Logical addresses + ***relocation register*** => physical addresses
    - By **MMU (Memory Management Unit)**
- See Figure 8.3.

Binding of Instructions and Data to Memory

- Types of addressing modes:
  - Immediate
    - Operand in instruction
  - Direct
    - Memory pointed by address field in instruction
    - EA = A
  - Indirect
    - Memory pointed by memory pointed by address field in instruction
    - EA = (A); EA = ((A)); …
  - Register
    - Register pointed by register field instruction
    - EA = R
  - Register indirect
    - Memory pointed by register pointed by register address field in instruction
    - EA = (R)
  - Displacement (indexed)
    - EA = A + (R)

Binding of Instructions and Data to Memory - continued

- Types of addressing modes - continued
  - Indexed
    - EA = A + R
  - Combined
    - Postindex
      - EA = (A) + (R)
    - Preindex
      - EA = (A + (R))
  - Stack

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*
    - One logical address space per process
    - Typically start at zero, but not necessarily
    - Space may consist of several segments
  - **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 and physical addresses differ in execution-time address-binding scheme.

Memory-Management Unit (MMU)

- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
- Hardware device that maps **logical to physical address**

- In MMU scheme, the value in the ***relocation register*** is added to every address generated by a user process.
  - A separate value in the relocation register for each process
  - The value in the relocation register is changed at the time of context switch.
  - ☺ *Where is the relocation register for a process?*
- See Figure 8.4.

Dynamic Loading

- The entire program might not fit to memory.
- 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.

Dynamic Linking

- Static linking
  - System libraries are linked into the binary program image when compile time.

- **Dynamic linking**
  - ***Shared library*** or ***DLL (Dynamic Link Library)*** – memory-resident library routines
  - Linking postponed until execution time.
  - ☺ *How?*
  - 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.
  - Dynamic linking is particularly useful for libraries.

2. Swapping

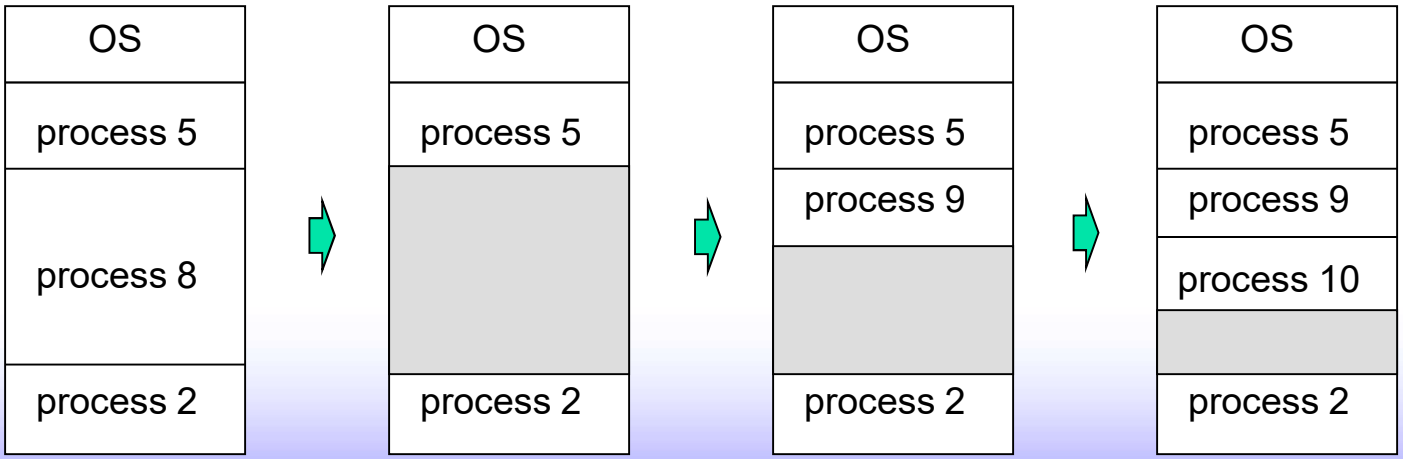
- A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- **Roll out, roll 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).
- See Figure 8.5.

3. Contiguous Allocation

- Main memory usually has two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Each program consists of one contiguous segment.
- 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.
- See Figure 8.6.

Contiguous Allocation - continued

- Single-partition allocation
- Multiple-partition allocation
  - *Hole* – block of available memory; holes of various size are scattered throughout memory.
  - When a process arrives, it is allocated from a hole large enough to accommodate it.
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Contiguous Allocation - continued

There could be several holes.

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

Contiguous Allocation - continued

- Problem
  - **External Fragmentation** – total free memory space exists to satisfy a request, but it is not contiguous.
  - 50% rule – waste of external fragmentation
- ☺ *How to reduce external fragmentation???*
- **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.
  - Very expensive operation
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

4. Paging

- Logical address space of a process can be noncontiguous.
- Fragmentation problem in contiguous allocation
- ☺ *How to solve???*
- **Paging**
  - 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.
    - Pointed by **page table base register**
  - A page table per process

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.
  - E.g., the size of the logical address space is 2^m, and a page size is 2^n, then p = the high-order $m - n$ bits of a logical address; d = the n low-order bits.

- See Figure 8.7.

Paging Example

- See Figure 8.8.
  - ☺ *Logical page 2 -> physical frame number?*
- See Figure 8.9.
  - ☺ *Logical address 5 -> physical address?*

Free Frames

- See Figure 8.10.

Implementation of Page Table

- ☺ *Where?*
- Page table is kept in main memory.
- A page table per process
- *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).**
- Expensive

Associative Memory

- Associative memory – parallel search

Page #	Frame #

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

- See Figure 8.11.

Effective Access Time

- Assume memory cycle time is 1 time unit.
- Associative Lookup = ε time unit $<< 1$
- **Hit ratio** – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.
- Hit ratio = α
- **Effective Access Time** (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

Example: 80% => EAT = $2 + \varepsilon - 0.8 = 1.2 + \varepsilon$

Page Table Structure

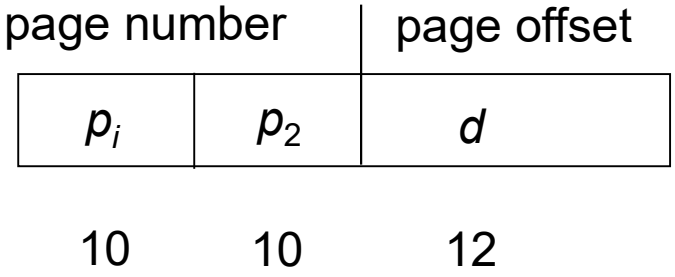
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

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:



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.

- See Figure 8.14.

Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture
- See Figure 8.15.

Hashed Page Table

- 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.
- Figure 8.16.

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.

- See Figure 8.17.

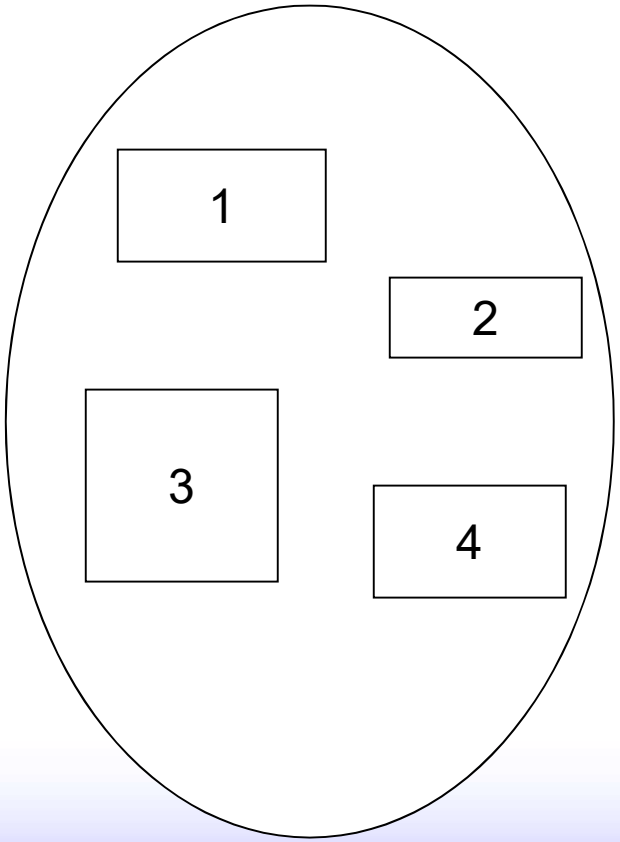
5. Segmentation

- Paging works well when programs consist of one contiguous address space.
- But, a program is a collection of segments. A segment is a logical unit such as:

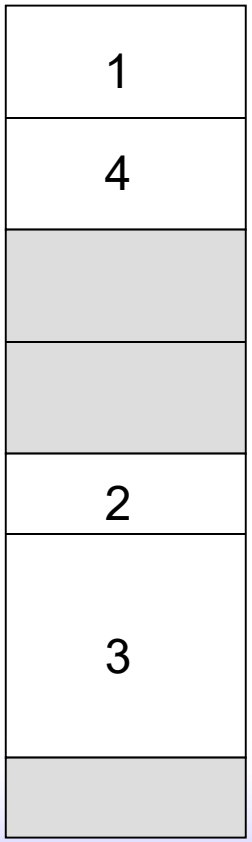
  main program,
  procedure,
  function,
  method,
  object,
  local variables, global variables,
  common block,
  stack,
  symbol table, arrays
- In order to use paging, the use of large page tables is unavoidable.
- **Memory-management scheme that supports user view of memory.**

User/Logical View of Segmentation

- See Figure 8.18, “User’s view of a program”.
- Logical view of a program.



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 problem

Segmentation Hardware

- See Figure 8.19 – 8.20.

Example of Segmentation

- See Figure 8.21.
  - ☺ *What physical address space is used for main program?*