Lecture 9

Main Memory II

Lecture Information

Ceng328 *Operating Systems* at April 20, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Address Binding I

- The process of associating program instructions and data to physical memory addresses is called *address binding*, or *relocation*.
- Addresses may be represented in different ways during these steps.
  - Addresses in the source program are generally symbolic (such as *count*).
  - A compiler will typically bind these symbolic addresses to **relocatable addresses** (such as "14 bytes from the beginning of this module").
  - The linkage editor or loader will in turn bind the **relocatable addresses** to **absolute addresses** (such as 74014).
  - Each binding is a mapping from one address space to another.

Address Binding II

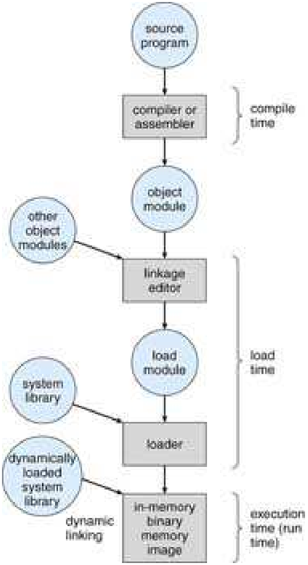


Figure: Multistep processing of a user program.

Logical Versus Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address**,
- Whereas an address seen by the memory unit -that is, the one loaded into the memory-address register of the memory- is commonly referred to as a **physical address**.
- The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- However the execution-time address-binding scheme results in differing logical and physical addresses.
- In this case, we usually refer to the logical address as a **virtual address**.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit** (MMU).

Logical Versus Physical Address Space

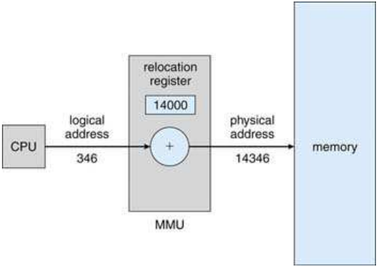


Figure: Dynamic relocation using a relocation register.

- A simple MMU scheme, which is a generalization of the base-register scheme (see Fig. 2)).
  - The base register is now called a **relocation register**.
  - 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 concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Swapping I

- A process can be **swapped** temporarily out of memory to a backing store (disk) and then brought back into memory for continued execution.
- A round-robin CPU-scheduling algorithm; when a quantum expires (see Fig. 3),

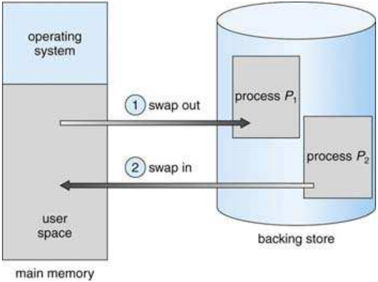


Figure: Swapping of two processes using a disk as a backing store.

- The quantum must be large enough to allow reasonable amounts of computing to be done between swaps.

Swapping II

- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.
- This restriction is dictated by the method of address binding.
  - If binding is done at assembly or load time, then the process cannot be easily moved to a different location.
  - If execution-time binding is being used, however, then a process can be swapped into a different memory space.
- Context-switch time; to get an idea of the context-switch time,
  - Let us assume that the user process is 10 MB in size and the backing store is a standard hard disk with a transfer rate of 40 MB per second.
  - The actual transfer of the 10-MB process *to* or *from* main memory takes

    `10000 KB/40000 KB per second = 1/4 second`
    `= 250 milliseconds.`
  - Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds. (swap out + swap in = 516 msec)

Swapping III

- For efficient CPU utilization, we want *the execution time for each process to be long relative to the swap time*.
- Thus, the time quantum should be substantially larger than 0.516 seconds.
- Notice that the major part of the swap time is transfer time.
- Generally, swap space is allocated as a chunk of disk, separate from the file system, so that its use is as fast as possible.
- Currently, standard swapping is used in few systems. A modification of swapping is used in many versions of UNIX.
  - Swapping is normally disabled but will start if many processes are running and are using a threshold amount of memory.
  - Swapping is again halted when the load on the system is reduced.

Memory Mapping and Protection I

- We need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
- In the contiguous memory allocation, each process is contained in a single contiguous section of memory.
- With **relocation** and **limit** registers, each logical address must be less than the limit register;
- The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (see Fig. 4).

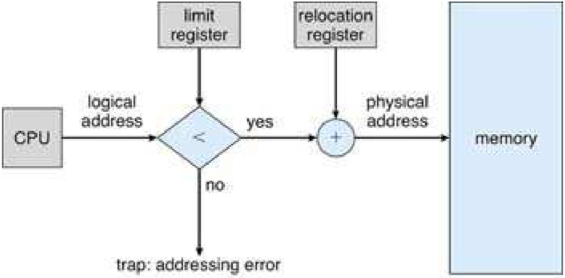


Figure: Hardware support for relocation and limit registers.

Memory Mapping and Protection II

- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
- The relocation-register scheme provides an effective way to allow the OS size to change dynamically.
- For example, the OS contains code and buffer space for device drivers.
  - If a device driver (or other OS service) is not commonly used, we do not want to keep the code and data in memory.
  - Such code is sometimes called **transient** OS code; it comes and goes as needed.
  - Thus, using this code changes the size of the OS during program execution.

Memory Allocation I

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.
- Thus, the *degree of multiprogramming* is bound by the number of partitions.
- In this **multiple-partition** method,
  - When a partition is free, a process is selected from the input queue and is loaded into the free partition.
  - When the process terminates, the partition becomes available for another process.
- This method is no longer in use.

Memory Allocation II

- The next method is a generalization of the fixed-partition scheme (called MVT, Multiprogramming with Variable Partitions).
- In the fixed-partition scheme,
  - The OS keeps a table indicating which parts of memory are available and which are occupied.
  - Initially, all memory is available for user processes and is considered one large block of available memory, a hole.
  - When a process arrives and needs memory, we search for a hole large enough for this process.
  - If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.
- At any given time, we have a list of available block sizes and the input queue.
- The OS can order the input queue according to a scheduling algorithm.

Memory Allocation III

- This procedure is a particular instance of the general **dynamic storage-allocation** problem, which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem.
  - **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
  - **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the *smallest leftover hole*.
  - **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the *largest leftover hole*, which may be more useful than the smaller leftover hole from a best-fit approach.
- Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.
- Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Address Binding

Logical Versus Physical Address Space

Swapping

Contiguous Memory Allocation
Memory Mapping and Protection
Memory Allocation
Fragmentation

Paging
Basic Method
Protection
Shared Pages

Segmentation
Basic Method
Hardware

Fragmentation I

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**.
- External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.
- Storage is fragmented into a large number of small holes.
- Memory fragmentation can be **internal** as well as external.
    - Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.
    - Suppose that the next process requests 18,462 bytes.
    - If we allocate exactly the requested block, we are left with a hole of 2 bytes.
    - The difference between these two numbers is internal fragmentation; memory that is internal to a partition but is not being used.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.

Fragmentation II

- One solution to the problem of external fragmentation is **compaction**.
- The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be **non-contiguous**, thus allowing a process to be allocated physical memory wherever the latter is available.
- Two complementary techniques achieve this solution:
  - paging
  - segmentation
- These techniques can also be combined.

Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
- Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store.
- The backing store also has the fragmentation problems discussed in connection with main memory, except that access is much slower, so compaction is impossible!
- Because of its advantages over earlier methods, paging in its various forms is commonly used in most OSs.
- Traditionally, support for paging has been handled by hardware.
- However, recent designs have implemented paging by closely integrating the hardware and OS, especially on 64-bit microprocessors.

Basic Method I

- The basic method for implementing paging involves
  - breaking physical memory into fixed-sized blocks called frames
  - breaking logical memory into blocks of the same size called pages.
- *The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.*

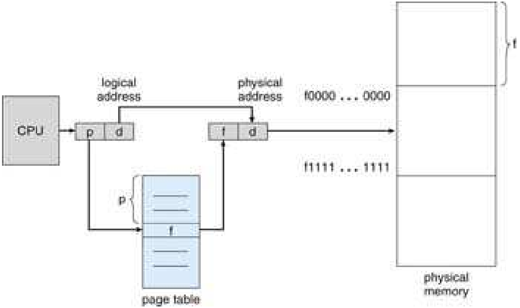


Figure: Paging hardware.

- The hardware support for paging is illustrated in Fig. 5.

Basic Method II

- Every address generated by the CPU is divided into two parts: a **page number** (*p*) and a **page offset** (*d*).
- The page number is used as an index into a page table.
- The page table contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the physical memory address.
- The paging model of memory is shown in Fig. 6.

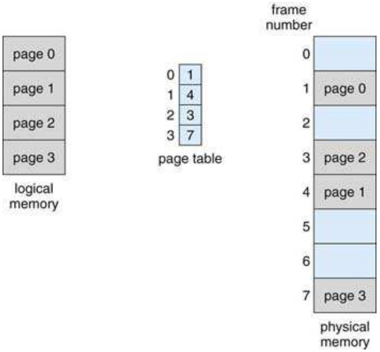


Figure: Paging model of logical and physical memory.

Basic Method III

- The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.
- Consider the memory in Fig. 7. It is shown that how the user's view of memory can be mapped into physical memory.
    - Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).
    - Logical address 0 is page O, offset O. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0).
    - Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 x 4) + 3).
    - Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 x 4) + 0).
    - Logical address 13 maps to physical address 9.

Basic Method IV

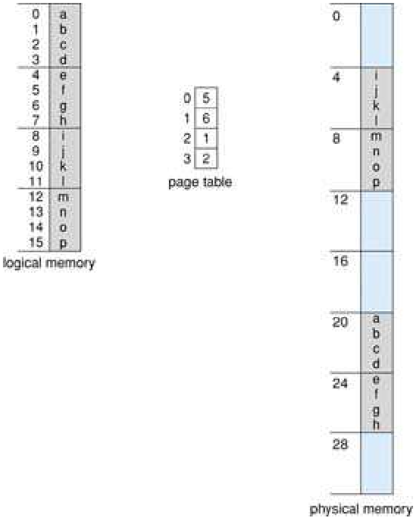


Figure: Paging example for a 32-byte memory with 4-byte pages.

Basic Method V

- Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.
- When we use a paging scheme, we have no external fragmentation:
  - Any free frame can be allocated to a process that needs it.
- However, we may have some internal fragmentation.
- If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
- For example, if page size is 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes.
- It would be allocated 36 frames, resulting in an internal fragmentation of 2,048 - 1,086 = 962 bytes.
- In the worst case, a process would need *n* pages plus 1 byte. It would be allocated $n + 1$ frames, resulting in an internal fragmentation of almost an entire frame.

Basic Method VI

- What about page size?
- Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.
- Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes.
- Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames.
- If frame size is 4 KB, then a system with 4-byte entries can address $2^{44}(4KB * 2^{32})$ bytes (or 16 TB) of physical memory.

Basic Method VII

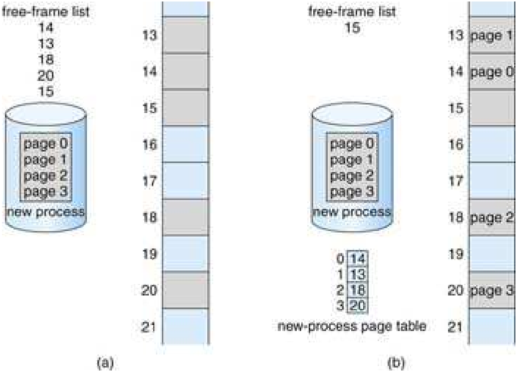


Figure: Free frames (a) before allocation and (b) after allocation.

Basic Method VIII

- An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.
- The logical addresses are translated into physical addresses by the address-translation hardware.
- This mapping is hidden from the user and is controlled by the OS.
- The user process has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.
- Since the OS is managing physical memory, it must be aware of the allocation details of physical memory
  - which frames are allocated,
  - which frames are available,
  - how many total frames there are, and so on.
- This information is generally kept in a data structure called a **frame table**.
- The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and,
- if it is allocated, to which page of which process or processes.

Protection I

- Memory protection in a paged environment is accomplished by **protection bits** associated with each frame.
- These bits are kept in the page table. One bit can define a page to be read-write or read-only.
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit.
  - When this bit is set to "valid", the associated page is in the process's logical address space and is thus a legal (or valid) page.
  - When the bit is set to "invalid", the page is not in the process's logical address space.
- Illegal addresses are trapped by use of the valid-invalid bit.

Protection II

- Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.
    - Given a page size of 2 KB (with 6 pages $2048 * 6 = 12288$).
    - item Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
    - Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the OS (invalid page reference).
- Because the program extends to only address 10468, any reference beyond that address is illegal.
- However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid.
- Only the addresses from 12288 to 16383 are invalid.
- This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Protection III

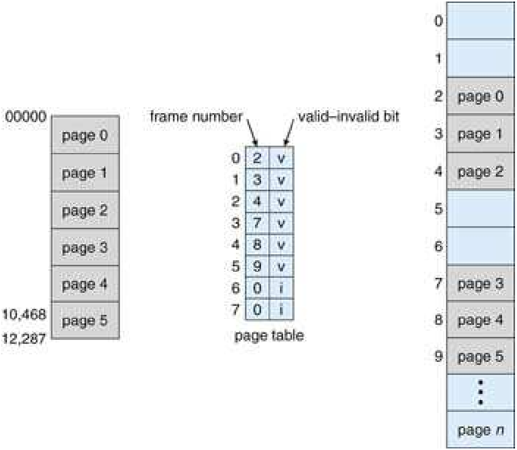


Figure: Valid (v) or invalid (i) bit in a page table.

Shared Pages I

- An advantage of paging is the possibility of sharing common code.

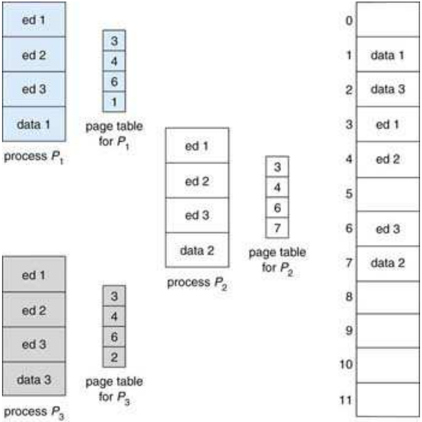


Figure: Sharing of code in a paging environment.

- Consider a system that supports 40 users, each of whom executes a text editor (see Fig. 10).

Shared Pages II

- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users $(40 * (150KB + 50KB))$.
- If the code is reentrant code (or pure code), it can be shared (to be shareable, the code must be reentrant).
- Reentrant code is non-self-modifying code; it never changes during execution.
- Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution.
- Only one copy of the editor need be kept in physical memory.
- Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user.
- The total space required is now 2,150 KB instead of 8,000 KB-a significant savings.

Basic Method I

- Users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments (Figure 8.18).

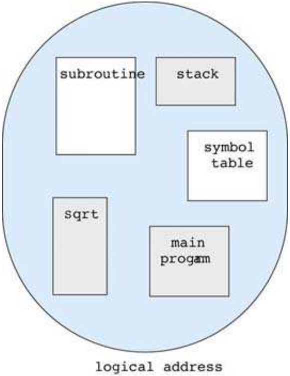


Figure: User's view of a program.

- Segmentation is a memory-management scheme that supports this user view of memory.
- A logical address space is a collection of segments.

Basic Method II

- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.
  - a segment name
  - an offset
- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- Thus, a logical address consists of a two tuple:

  `<segment-number, offset>`
- A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*.

Hardware I

- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes.
- Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.
- This mapping is effected by a **segment table**. Each entry in the segment table has a segment base and a segment limit.
- The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment (see Fig. 12).

Hardware II

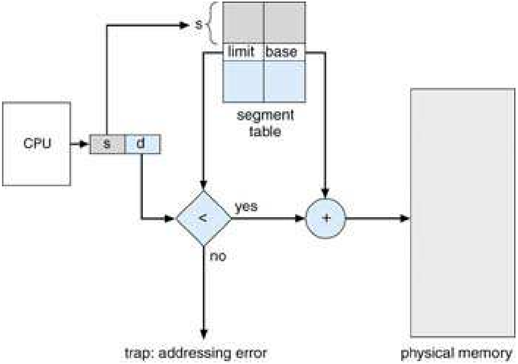
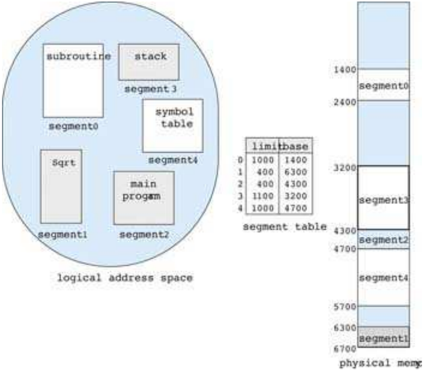


Figure: Segmentation hardware.

Hardware III

- The segment number is used as an index to the segment table.
- The offset *d* of the logical address must be between 0 and the segment limit.
- If it is not, we trap to the OS (logical addressing attempt beyond end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- The segment table is thus essentially an array of base-limit register pairs.
- As an example, consider the situation shown in Fig. 13.

Hardware IV



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Figure: Example of segmentation.

Hardware V

- We have five segments numbered from 0 through 4.
- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- For example, segment 2 is 400 bytes long and begins at location 4300.
- Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.
- A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
- A reference to byte 1222 of segment would result in a trap to the OS, as this segment is only 1,000 bytes long.