Memory Management - 2

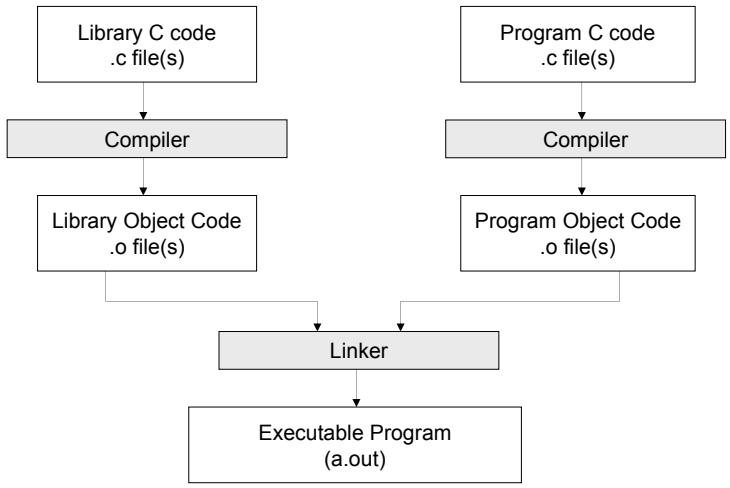
CS 342 – Operating Systems

Ibrahim Korpeoglu
Bilkent University
Department of Computer Engineering

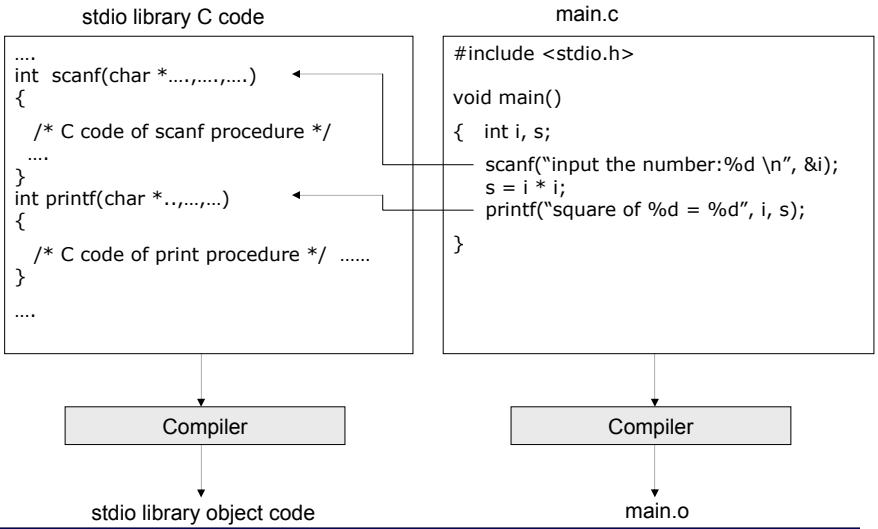
Relocation and Protection

- Multiprogramming introduces two problems:
  - Relocation
  - Protection
- Relocation
  - When a program is linked
    - User-written program and library procedures are compiled into a single address space.

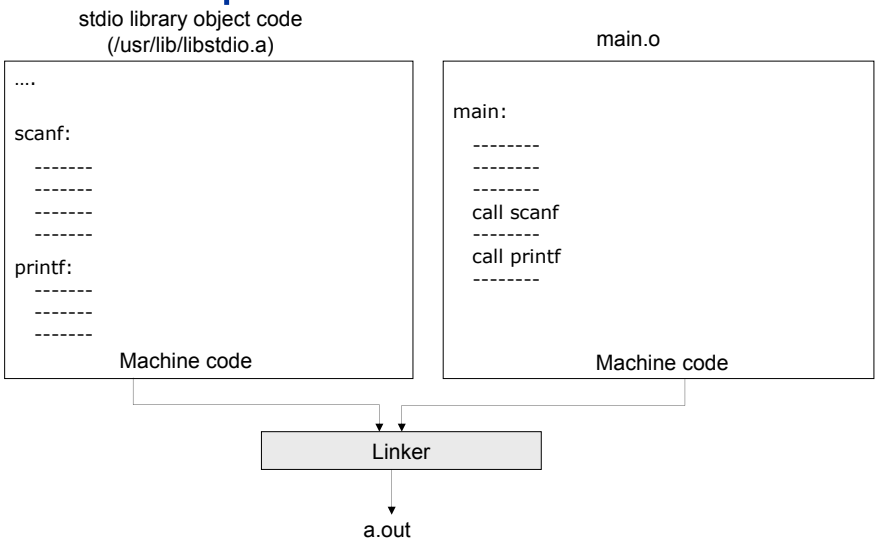
Linking



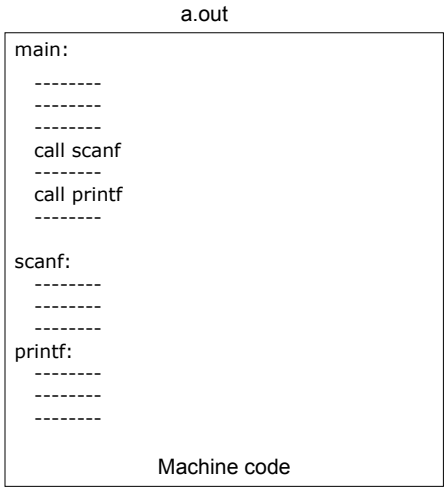
Compiler Input/Output



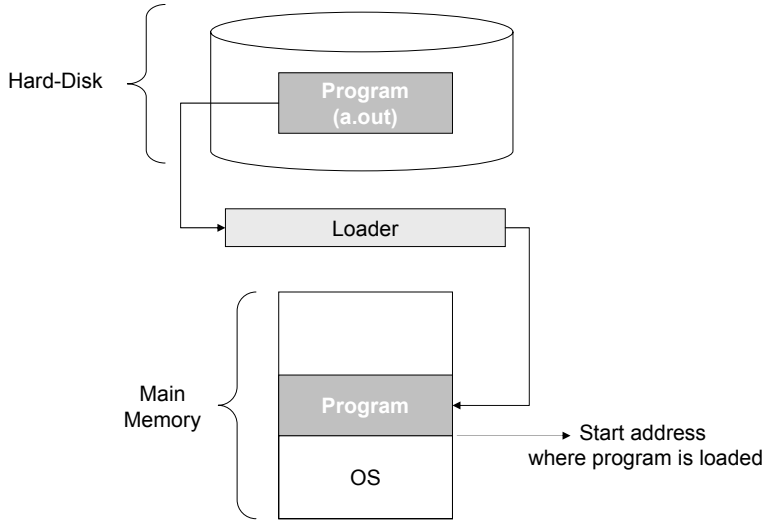
Linker Input



Linker Output



Loader



Linker and Relocation

- Linker produces an executable file which memory references for *variables* and *procedures*
  - Example: variables i and s in the previous slides
    procedures scanf and printf
- The linker does not know in advance where actually a program will be loaded in memory
  - Linker produces output that assumes that the program will be loaded to a place in memory starting with *memory address zero*.

Relocation Problem

- But due to multiprogramming, the physical start address in memory where program will be loaded into may be different than zero.
- In this case all memory references (based on zero start address) in program will be wrong when program is loaded.

Relocation Solutions

- 1) Modify the instructions of the program that make memory reference when a program is first loaded into memory
  - Linker must include a list together with the binary program telling which program words are memory addresses.
  - Does not solve the protection problem.
- Protection Problem is:
  - a user program should not be able to access the part of memory that belongs to some other user program or OS

Relocation Solutions

- 2)
  - Divide memory into blocks of 2 KB bytes and assign a 4-bit protection code to each block.
  - The PSW register contains a 4-bit protection code for the running program
  - The running program can access only the blocks of memory where the protection code in PSW and protection codes of blocks match

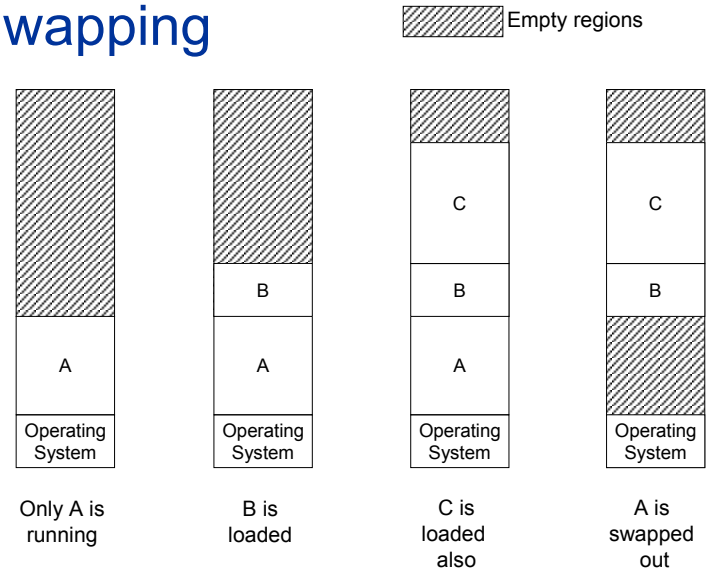
Relocation Solutions

- 3)
  - A hardware solution
  - Have 2 extra special registers:
    - Limit and Base registers
      - Base register contains the start address of program in memory
      - Limit register contains the size of the program in memory
  - When a memory reference is generated by CPU, the base address is added to the memory reference to find the actual physical memory address.
  - When a memory reference is made beyond *base+limit*, and error is returned.
  - This solution also solves the problem of protection.

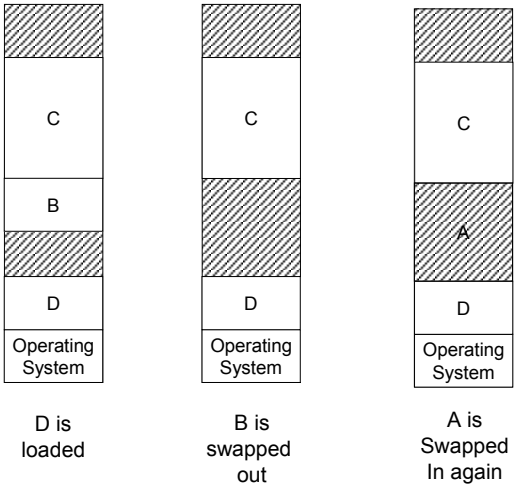
Swapping and Paging

- Sometimes there not enough memory space to keep *all* the running programs in memory
- Two strategies to deal with this case
  - 1) Swapping
  - 2) Paging
- *Swapping*:
  - *Bring* each process as a whole into memory, *run it* for a while, then put it back *on the disk*.
- *Paging*:
  - A process does not have to brought up to the memory as a whole when it needs to be executed. It allows the programs to run even through they are partially in memory.

Swapping



Swapping



Swapping

- This causes variable partitions to be made dynamically in memory.
- May provide better CPU utilization
- Requires data structures to keep track of which parts of memory are used and which parts are free (holes).
- Memory compaction can be used to combine small sized holes into big ones.

Processes and memory

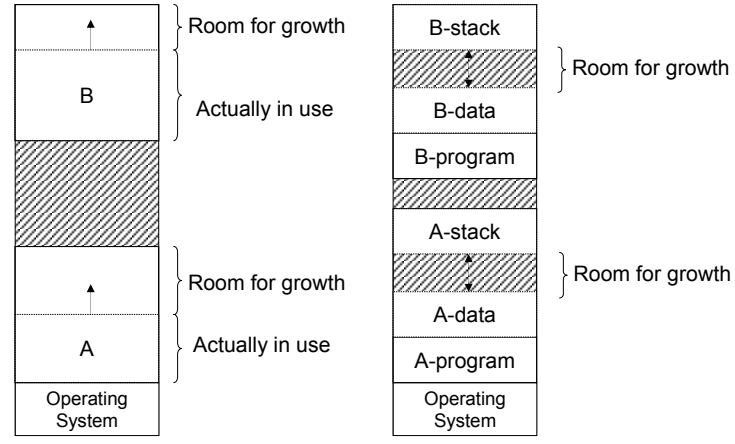
- A process should reside in consecutive memory addresses
  - i.e some part of program can not be loaded to a different address
    - Otherwise relocation would be very difficult
- Memory required to load a program
  - Depends on whether the program has a fixed size or variable size (may allocate more memory during execution).

Processes and Memory Requirements

- Processes data segments may grow
  - by allocate more memory dynamically from a heap
  - C, Pascal allows this for example.
- If a process wants to grow, there are two strategies to follow:
  - 1) An adjacent hole can be used to grow.
  - 2) Program can be relocated to a different larger hole

Processes and Memory Requirements

- A process may have two growing segments
  - Data segment: variables dynamically created.
  - Stack segment: local variable, etc.
- When a process is loaded into memory, it is a good idea to allocate *more space* for the process than it initially needs.



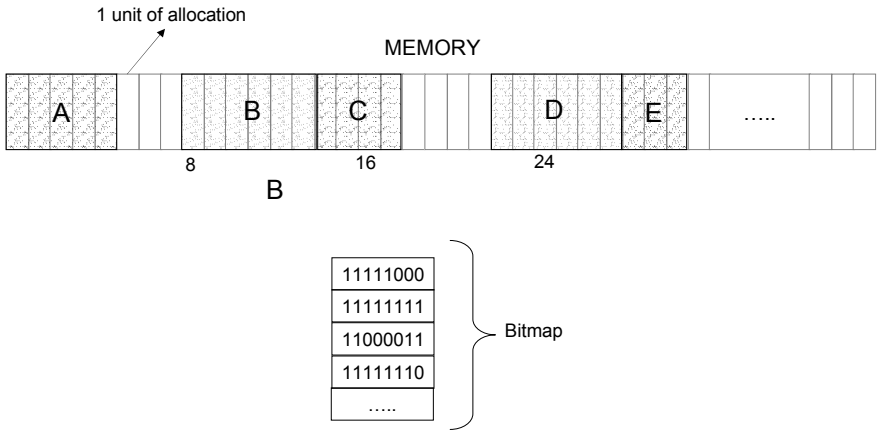
Keeping Track of Memory Usage

- We need to manage the allocated and free memory space
  - Bitmaps
  - Free Lists

Bitmaps

- Memory is divided into allocation units
  - Units can be
    - A few words
    - ….
    - Several kilobytes
- We use a bit for each unit, telling if unit is free or allocated.
- Size of allocation unit
  - Affects the size of the bitmap
    - Each unit is represented with a bit
    - Number of Bits Required = (Size of Memory) / (Allocation Size)
  - Affects the unusable memory space
    - The last unit allocated to a process may not be filled up.
    - The larger the allocation unit, the bigger is the unused space in the last unit.

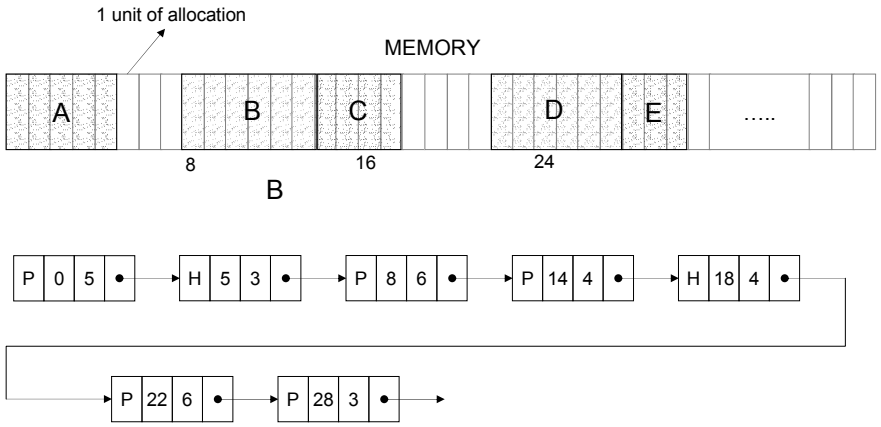
Bitmaps



Linked Lists

- Linked list of allocated and free memory segments
  - A segment is a process as a whole, or
  - A segment is a hole (free)
- Segment list could be kept sorted with respect to the address
- Needs to be updated when processes come and go.

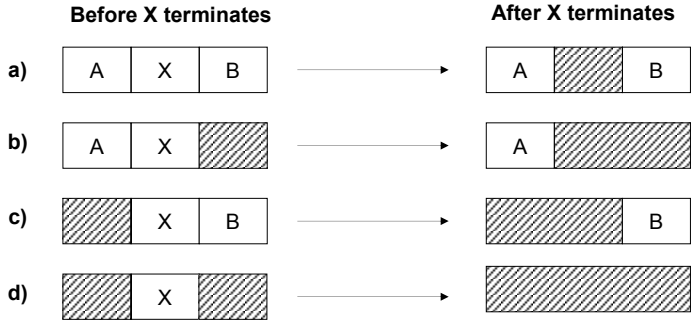
Linked Lists



Linked Lists – process termination

- When a process terminated (or swapped out) the corresponding entry must be deleted from linked list.

4 cases to consider



Linked Lists – process creation

- When a program is to be loaded into memory, a hole must be selected for the program
- There are various strategies
  - First Fit
  - Next Fit
  - Best Fit
  - Worst Fit

First Fit

- Select the first whole from start of the list, that can hold the program
- The hole is broken into pieces
  - An allocated piece for the program
  - A new hole (remaining of the original hole)
- It is a fast algorithm

Next Fit

- Similar to first fit, except,
  - It remembers where it stopped in the previous search for a fit.
  - It starts from that point for the next search of a fit for a new program
- Performs slightly worse than first fit.

Best Fit

- Search the entire list for a hole that is smallest and that can hold the new program
- *Slower* than the previous methods
- Causes more *waste of memory* than first or next fits.
  - Best fit generated tiny holes that are useless.
  - First fit generated larger holes.

Worst Fit

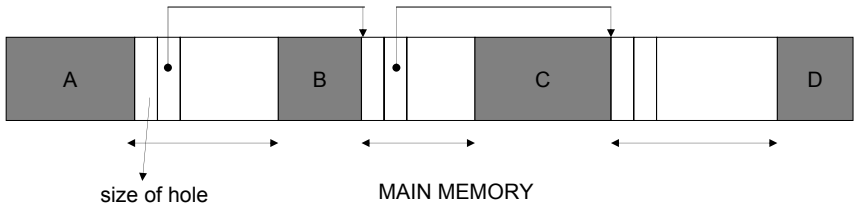
- Searched the entire list for a hole that has the largest size.
  - Simulation shows that this is not a very good idea either.

Speeding up the algorithms

- Keep separate lists of allocated segments and holes.
- Search for a hole will be done on a *smaller list.*
- De-allocation of a used segment is more difficult.
  - Requires searching the neighbors of segment which can be holes or processes.
- The hole list could be *sorted* according to the hole size.

Speeding up the algorithms

- If hole list is separate, a further optimization is possible.
  - Do not keep an explicit separate list for keeping holes.
  - Instead implement the list as part of the holes themselves.



Quick Fit

- Again separate lists are maintained for allocates spaces and holes.
- The list for holes can be further refined.
  - Keep separate lists of holes for common size.
  - Assume there are n different hole sizes, such as 4KB, 8KB, 12KB, etc.
- When a program is to be loaded, the list closest to the size of the program is selected.

Virtual Memory

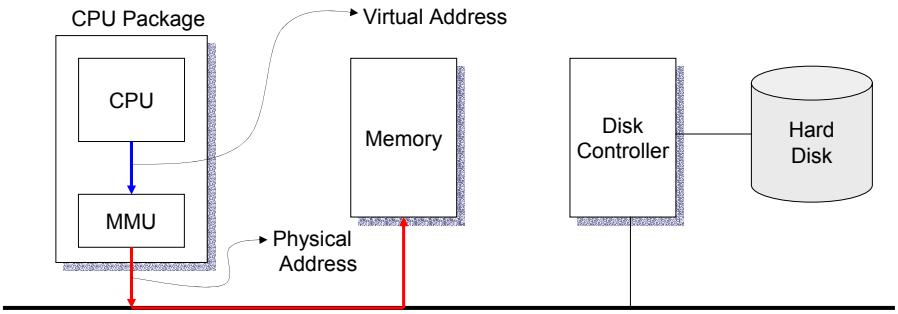
- Execute programs whose size is greater than physical memory size.
- The idea is:
  - OS keeps only the necessary part of the program in the memory, the rest in the hard disk
  - When a memory reference is made to the other part in hard-disk, that part is brought into memory.
  - An unused part is stored back to disk (if it is modified).

Paging

- Virtual Memory systems use a technique called *paging* to realize this idea.
- There are two type addresses
  - Virtual addresses
    - Memory addresses that are generated by a program
    - This addresses is not used directly to retrieve a word from physical memory
  - Physical address
    - Addresses that are used actually to address a word in physical memory.
    - This is the address that is put on the bus between CPU and Memory.
- Virtual addresses are generated by program instruction like the following:
  - MOV REG, 1000 /* move into register REG the content of memory address 1000 */

Memory Management Unit

- Virtual Memory systems have a unit called Memory Management Unit, that does the address translation from virtual addresses to physical addresses.



Page Table

- Memory Management Unit uses a table called *page table* to map virtual addresses into physical memory addresses.
- Page table is index by virtual address (or by some portion of it).
- A *page table entry* contains information to reach the word stored in physical memory

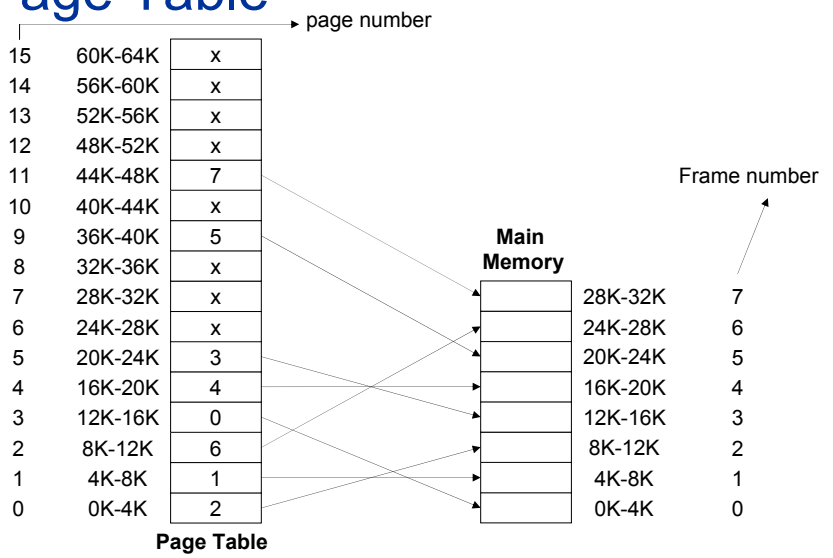
Page Table

- A Virtual address space is divided into fixed units called *pages*.
  - Page size is fixed
    - Can change from 512 bytes to 64 KB
- For each virtual page that should be a corresponding physical page in memory if that page is used.
  - The physical page is called *page frame*.
- Page and page frames have always the same size
  - When a corresponding page frame for a page is not found in memory, it is brought from hard-disk.

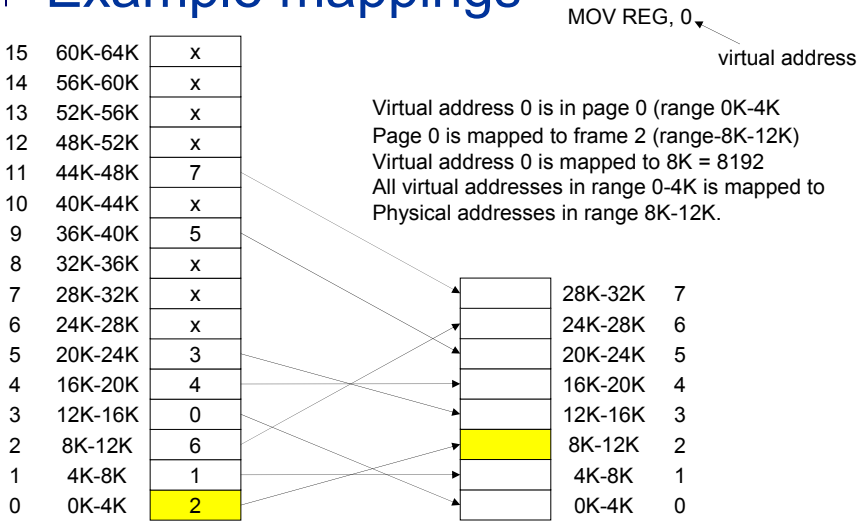
Example

- Assume a system
  - Virtual addresses are 16 bits
    - Virtual address space is 64 KB
  - Physical memory size is 32 KB
  - Page size is 4 KB.
- Given these values
  - The memory can hold at most 8 page frames
  - The number pages in virtual address space is
    - 64 KB / 4 KB = 16 pages.
  - Therefore, the page table size is 16: we need to have 16 entries in the page table. One for each page.

Page Table

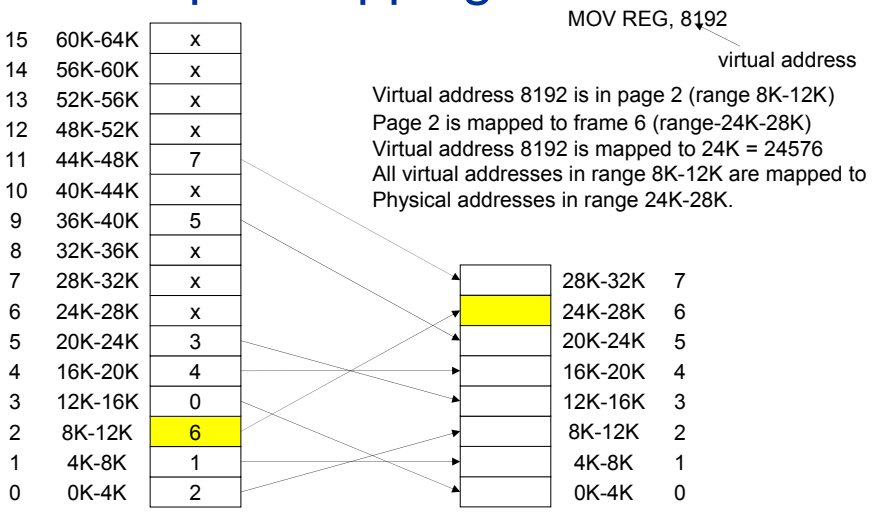


Example mappings

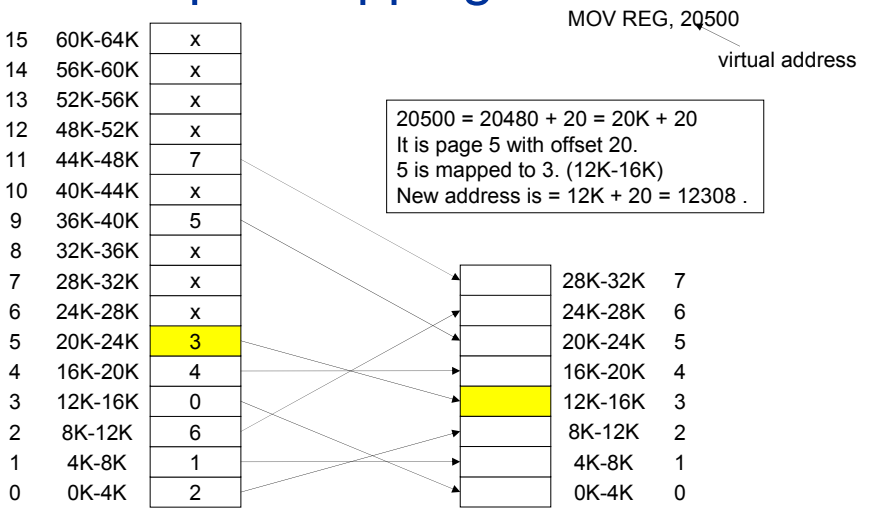


Virtual address 0 is in page 0 (range 0K-4K
Page 0 is mapped to frame 2 (range-8K-12K)
Virtual address 0 is mapped to 8K = 8192
All virtual addresses in range 0-4K is mapped to
Physical addresses in range 8K-12K.

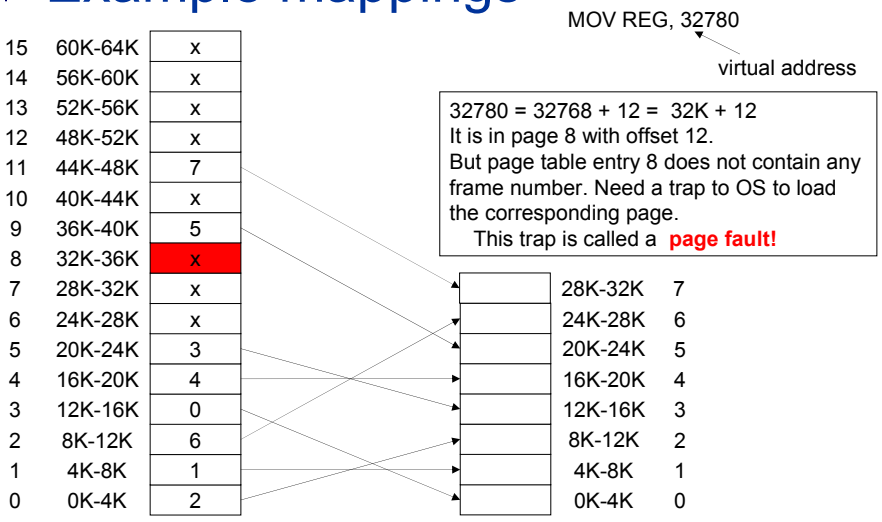
Example mappings



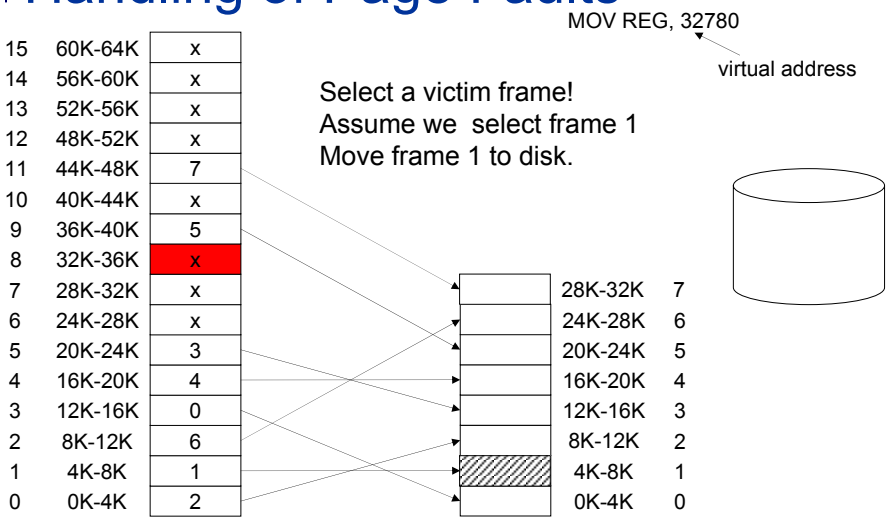
Example mappings



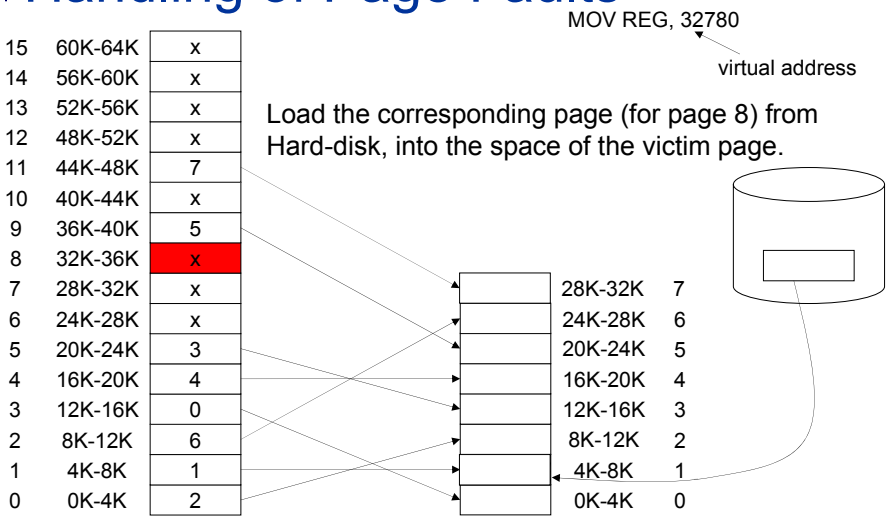
Example mappings



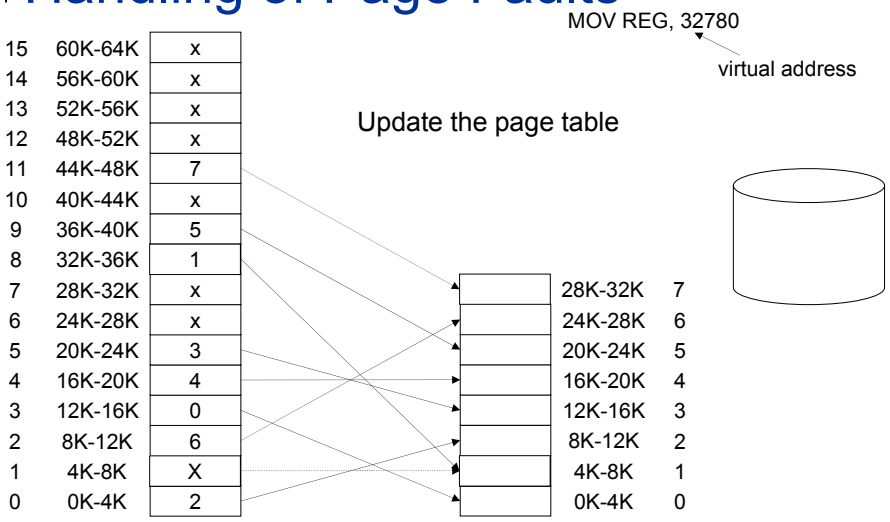
Handling of Page Faults



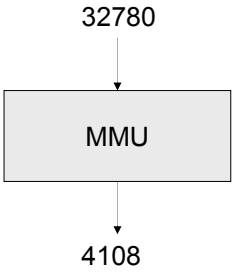
Handling of Page Faults



Handling of Page Faults



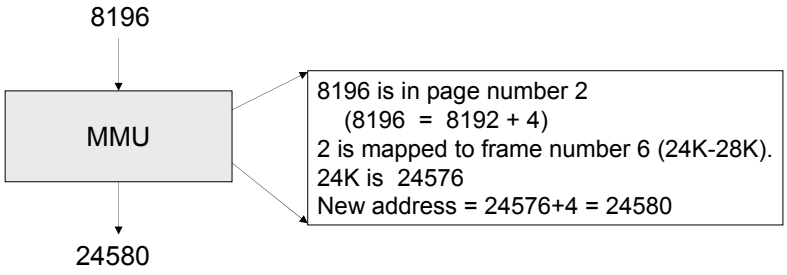
Handling of Page Faults

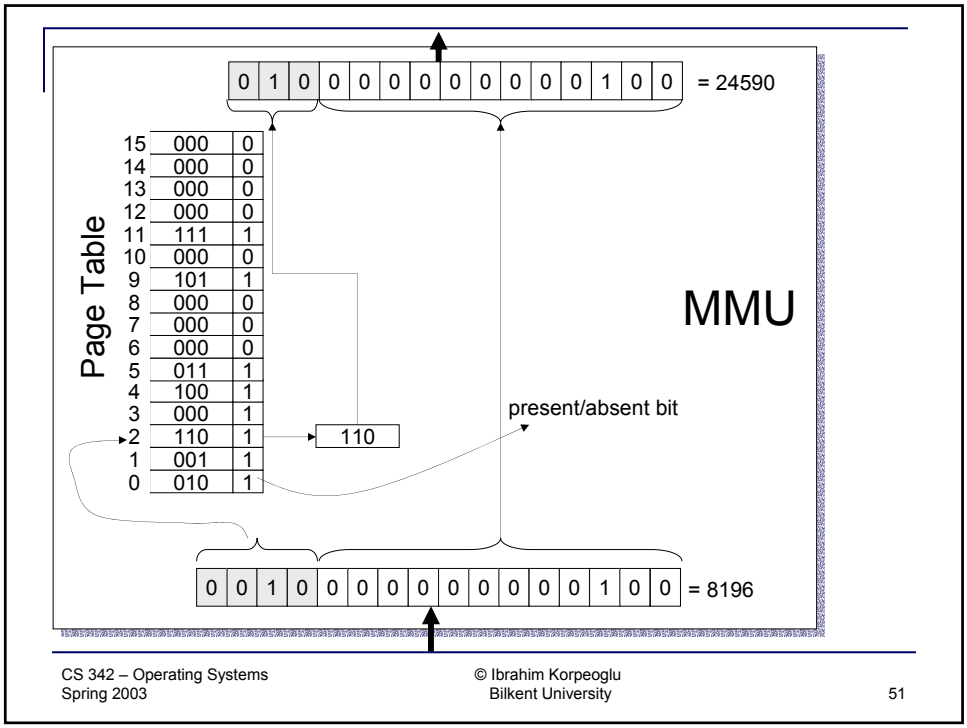


8 is mapped to frame 1 (4K-8K)
4K + 12 = 4096+12 = 4108.

After page fault is handles, the instruction can be continued executing.

MMU Internal Operation




0 1 0 0 0 0 0 0 0 0 0 1 0 0
= 24590
Page Table
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
110
MMU
present/absent bit
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
= 8196