Memory Management

Using Tanenbaum's Modern Operating Systems (3rd edition)

Memory Hierarchy

- The OS must juggle the user requests to let the user feel like it has **infinite**, **fast**, and **nonvolatile** memory.

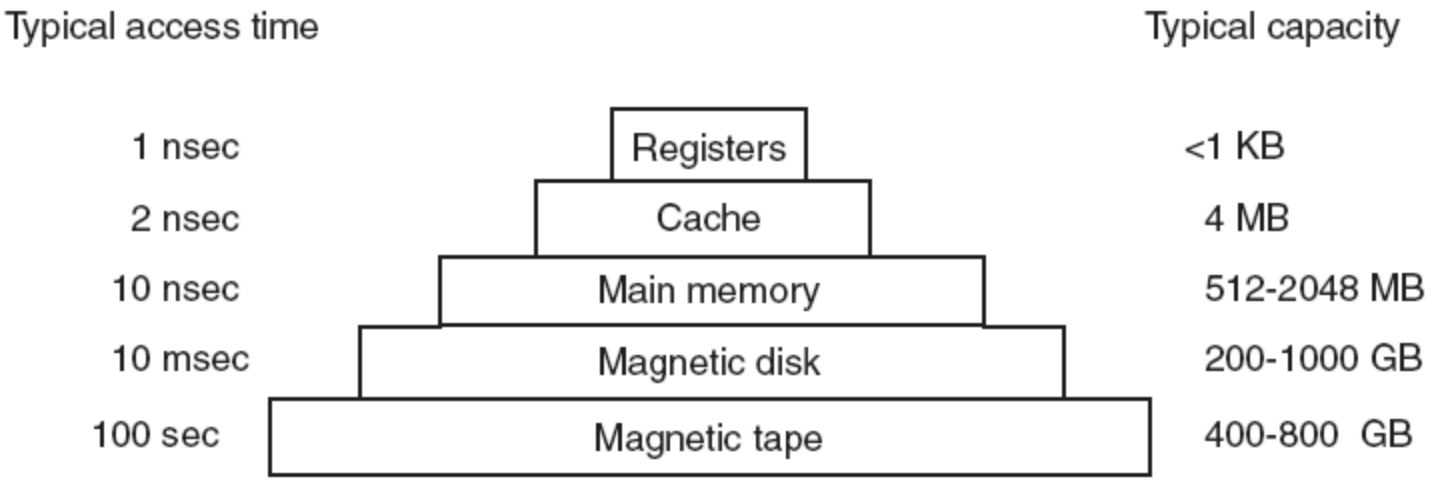


Figure 1-9. A typical memory hierarchy. The numbers are very rough approximations

OS Memory Manager

- Manages the memory hierarchy
- Keeps track of used and free parts
- Enables to allocate and de-allocate memory
- Manages swapping and paging

Memory Management Schemes

- Memory management schemes are divided into two:
  - Those that move processes between RAM and Disk
  - Those that do not

- Swapping and paging are motivated by lack of RAM
  - When RAM will grow big enough, swapping and paging might become obsolete
  - However, SW is growing faster than HW

- The simplest memory management schemes are still in use in the embedded OS's

Mono-Programming w/o Swapping & Paging

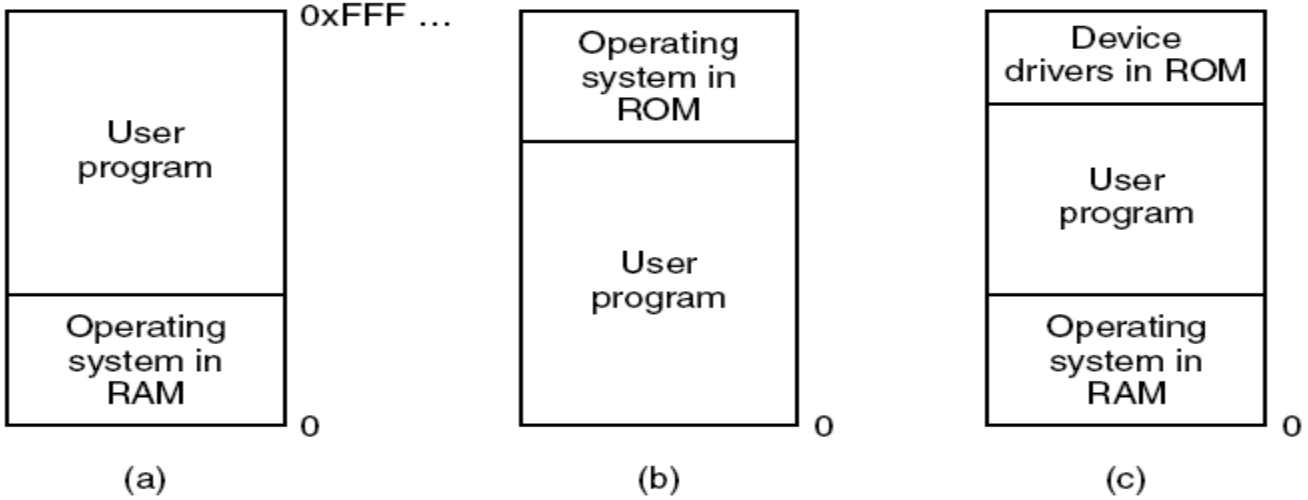


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

- The first was formally used by mainframes.
- The second still used by some embedded.
- The third was used in early PC's.

Multi-programming with Fixed Partitions (I)

- The easiest way to achieve multiprogramming, is to partition the memory into N (possibly of different size) partitions,
- But:
  - Unused part of a partition is wasted.
  - New job may wait for the smallest partition that may hold it.
  - Many small jobs may compete for the small partition while the big one remains empty.

Multiprogramming with Fixed Partitions (II)

Problems:

1. How to solve conflicts when two process access the same physical address?

2. How the hard-coded address translated to the real one?

Protection-code

1) To prevent one process from interfering with another:
   - Each memory block & process has a protection number (e.g. 4-bit code saved in PSW)
   - HW throws an exception, if process accesses memory with wrong protection code

Static Relocation

2) Multiprogramming requires **relocation**:
   - A program, which is loaded to an arbitrary memory segment, needs to change all its addresses accordingly
   - The loader uses a list or bitmap, made by the linker, to know which addresses must be adjusted

Pros/Cons:
   - Slows down loading
   - Needs an extra data in the executable (why?)

Dynamic Relocation

3) Alternative to both relocation and protection is to add **base** and **limit** registers that are accessible only to the OS:
   - Base is the beginning of the partition, and limit is the length of it
   - The HW protect access using the base and limit

Pros/Cons:

   - Need to do addition and comparison on each memory access.

Swapping

The Problem

- Background
  - On a typical Win/Linux machine – 40-60 processes may be started while booting (each one uses 5-10 MB)
  - Heavy user application needs 50-200MB
  - RAM is only 4G
- The Problem
  - There is not enough RAM to hold all the running processes!
- Solution:
  - Swapping

Swapping

Swapping is the action of loading **entire** process, running it, and swapping it back to disk

- Dynamic partition allocation improves flexibility (Is there a problem?)

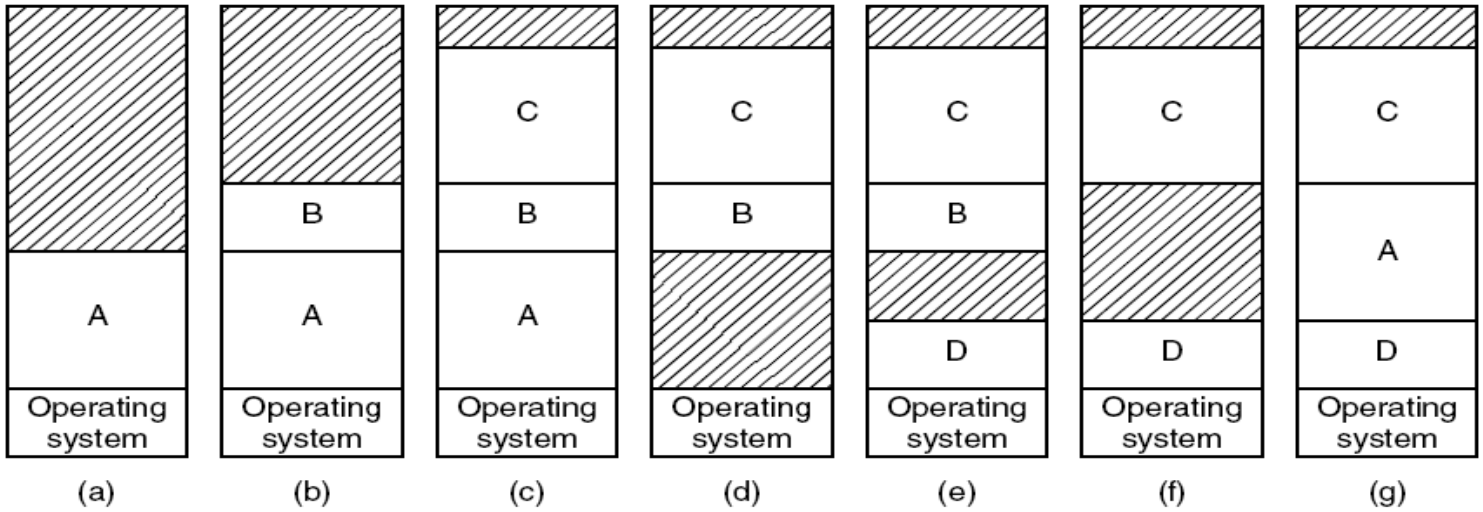


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory

Virtual Memory

The Motivation

- The Problem (Address space vs. Memory)
  - Systems must support multiple programs (simultaneously), each fits in memory but collectively exceed memory

- Swapping – not an option (Why?)
- Overlays(‘60) – Split the program into little pieces.
  - Splitting the program into pieces (by the programmer)
  - The pieces (=overlays) kept on the disk
  - Swap in/out by the overlay manager

Virtual Memory

What is ***Virtual Memory***?

- The virtual address space is divided into units called ***pages***
- Corresponding unit in physical memory are called ***page frames***.
- The pages and page frames are of the same size
  - Different for different CPU architectures
- The OS (using HW) performs the mapping on the fly
- VM is an abstraction of physical memory

Paging

- Programs and data are split to ***page*** size (4KB).
  - RAM and disk use the same page size
  - OS swaps pages from RAM to disk
  - No programmer intervention

- User calls address in virtual address space
- MMU maps virtual address space to pages in RAM or disk
- If page is on the disk – a page fault occurs, and the process is blocked until the page is loaded into RAM

MMU

- The code is accessing virtual memory address
- The **MMU**, translates this address to a real address in physical memory

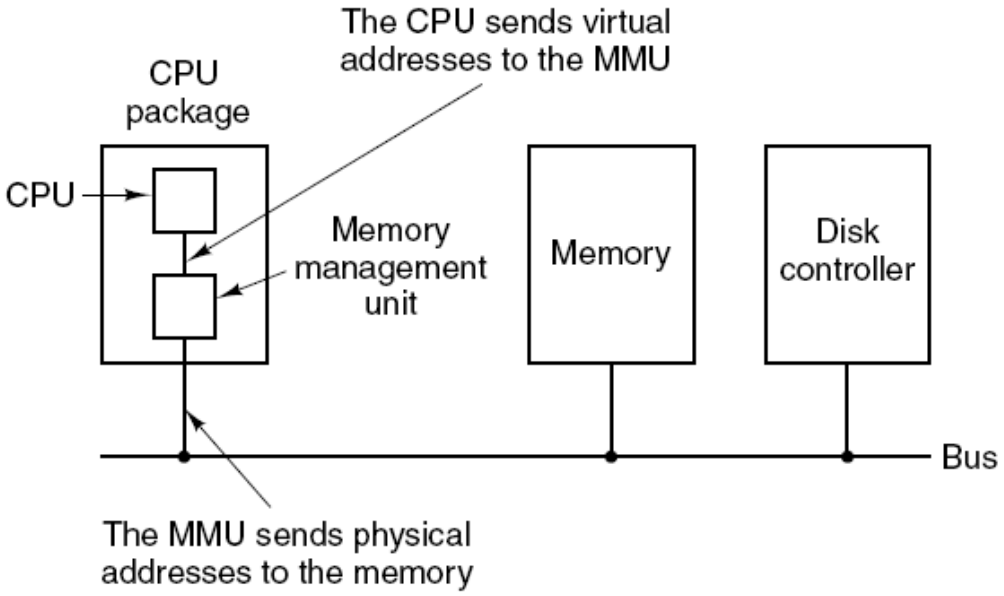
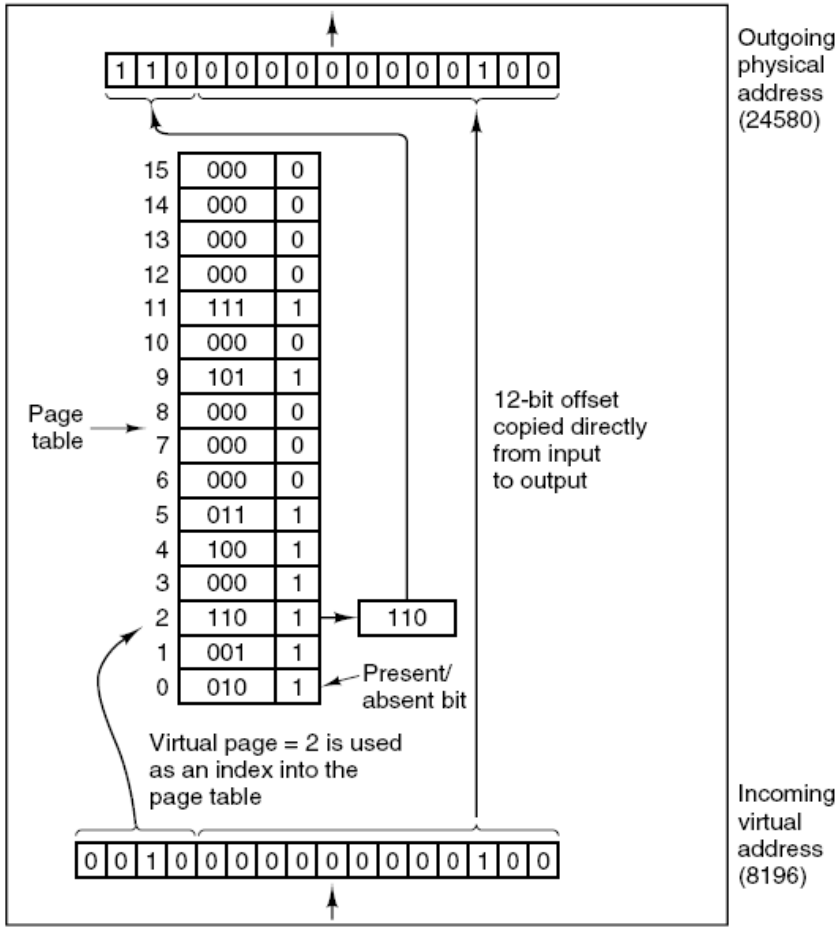


Figure 3-8. The position and function of the MMU Shown as being a part of the CPU chip as it commonly is nowadays. Logically it could be a separate chip, as it was in years gone by

Virtual Memory Address

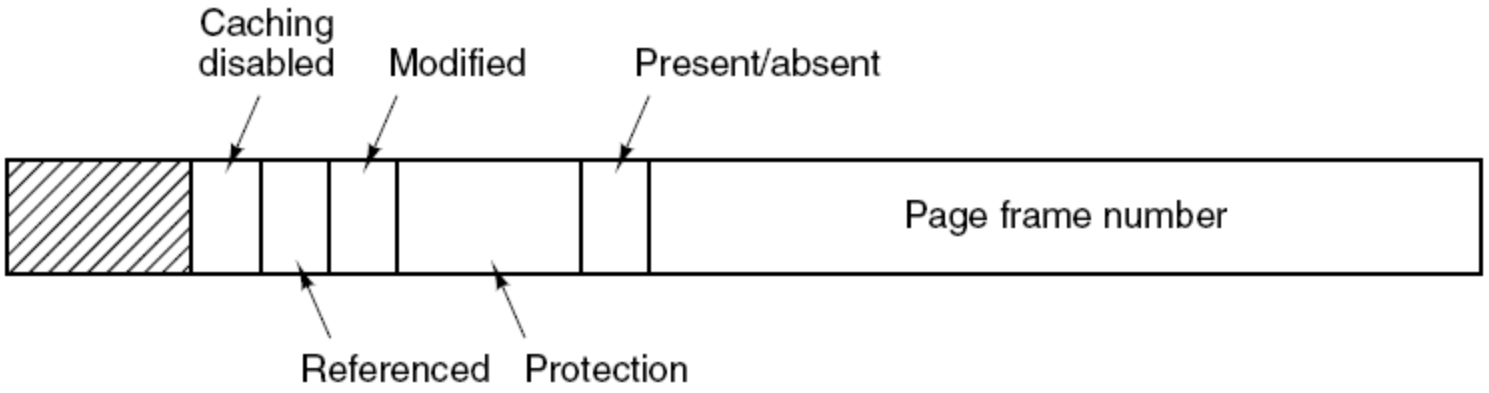
- ***Present/Absent*** bit keeps track of pages in memory.
- The n bits virtual memory address is split into:
  - ***x*** bits for page number, and
  - ***(n-x)*** bits for page offset
- The page number is the index into the page table.
- If the page is absent (absent bit = 0), a trap to the operating system is caused.

Page Table

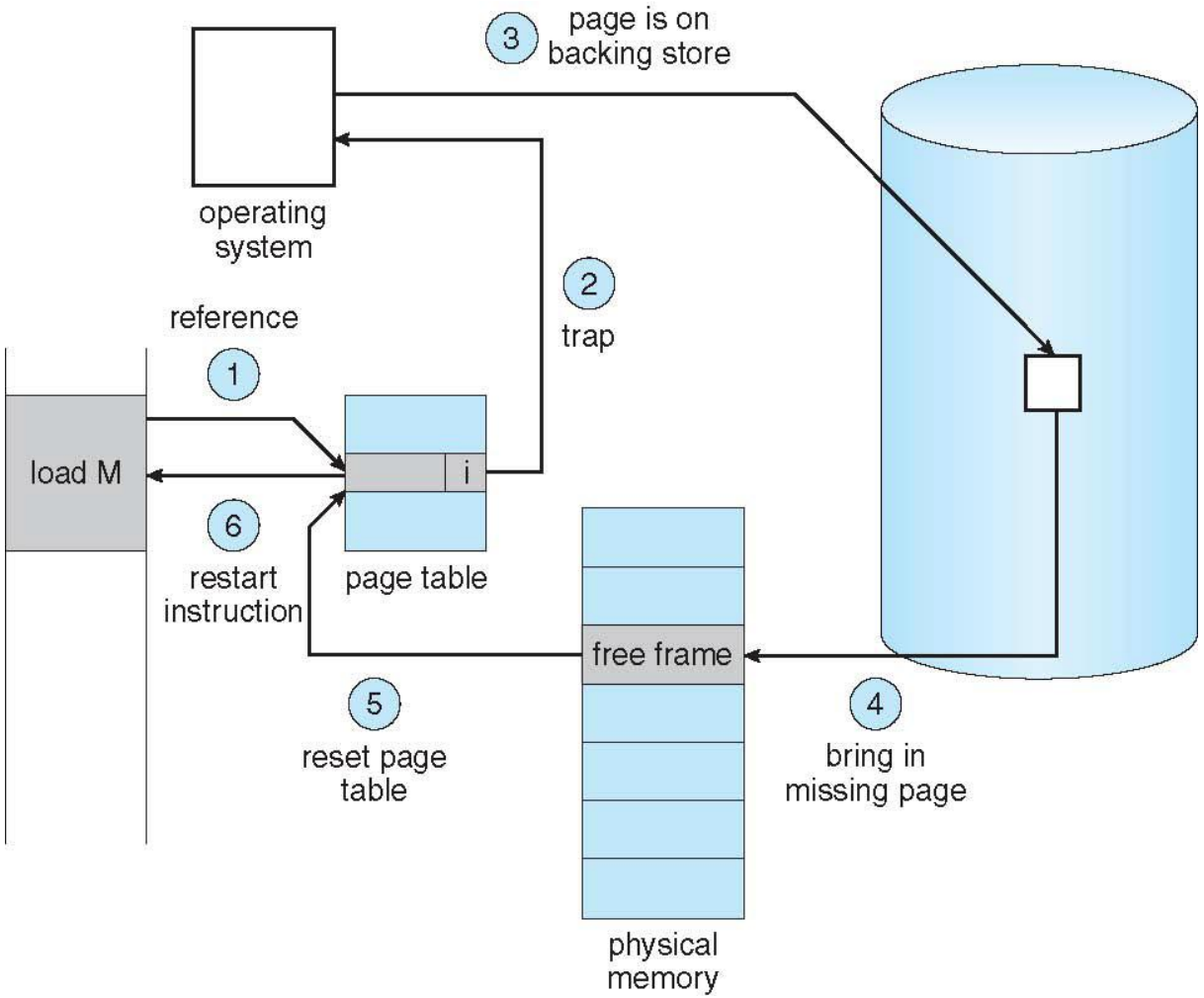


Page Table Entry

- **Protection bits** – page is R/W or R/O.
- **Modified (Dirty) bit** – page has been changed, and must be written back to the disk.
- **Referenced bit** – page is still needed in RAM.
- **Caching disabled bit** – page is used by an external device – do not cache it!



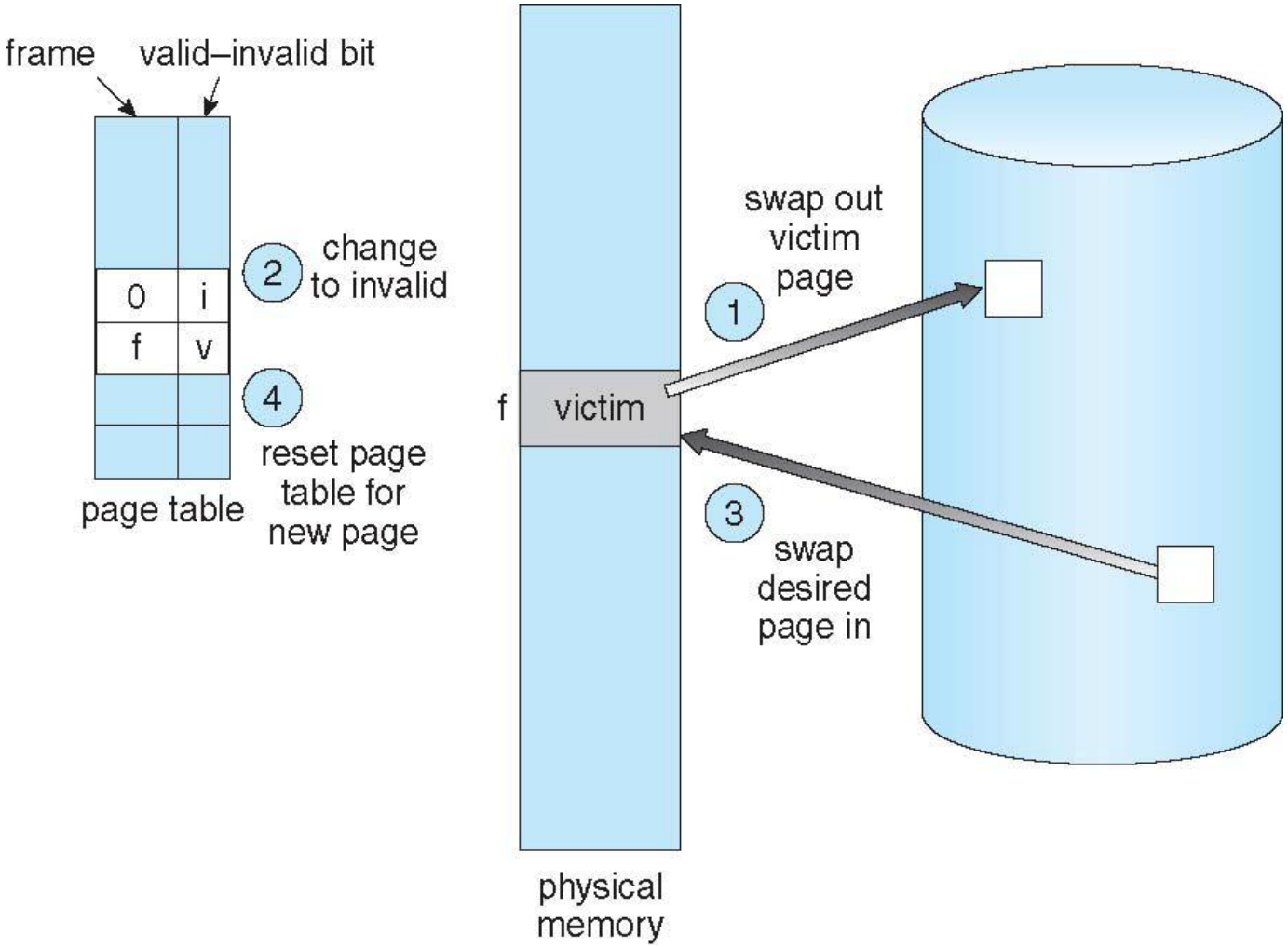
Page Fault



Page Fault Handling

- The HW traps the kernel:
  - Saves the current process state
- The OS examines the chosen page:
  - If the dirty bit is on – the page must be saved on a disk
- The OS allocates a page frame, finds the page on disk and, and starts loading it
- The process is blocked until an interrupt from disk says that the page is loaded and ready
- The OS updates the page table, and moves the process to the Ready queue

Page Replacement



Instruction Backup

- The instruction that caused the page fault must be executed.

Problem:

the PC usually points already to the next instruction, and the previous one might not be the previous one in memory…

- Some CPU's solve it by an instruction register (IR).

Swap Area

- File system used entirely for swapping page frames
- Different partition
- Each block size is exactly the page size (4K)

Improving Paging Performance

Implementation Issues

- Page table may be extremely large:
  - With 32-bit address, 4KB page size, one must map $2^{32}/(4*2^{10})=2^{20}$ virtual memory pages onto page frames
  - Each process has 2^{20} entries in its memory manager
- Mapping must be fast:
  - Machine instruction takes ~1nsec – mapping shall be done in under 0.2nsec.
    - Storing the entire page table in registers is too expensive.
    - Storing the entire page table in memory is too slow.
- So, where should the page table be located?!

Implementation Solutions

- Fast mapping - using TLB (HW):
  - **T**ranslation **L**ook-aside **B**uffer (Associative memory)
  - Compare all the entries (more than 64) in parallel
  - TLB hit/miss

- Large memory – using Multi-level page table:
  - Avoid keeping all the page tables in memory all the time
  - Keep page tables for the used part of the different memory segments

Using TLB

- A typical program, mostly accesses only a small number of pages (locality of reference)
- TLB is a HW buffer that does not go through the paging tables
- The MMU HW first checks the TLB
- When a program access a page that is not in TLB, the MMU looks the page up in the paging table, and inserts it into the TLB instead of some other entry.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging

Using TLB (II)

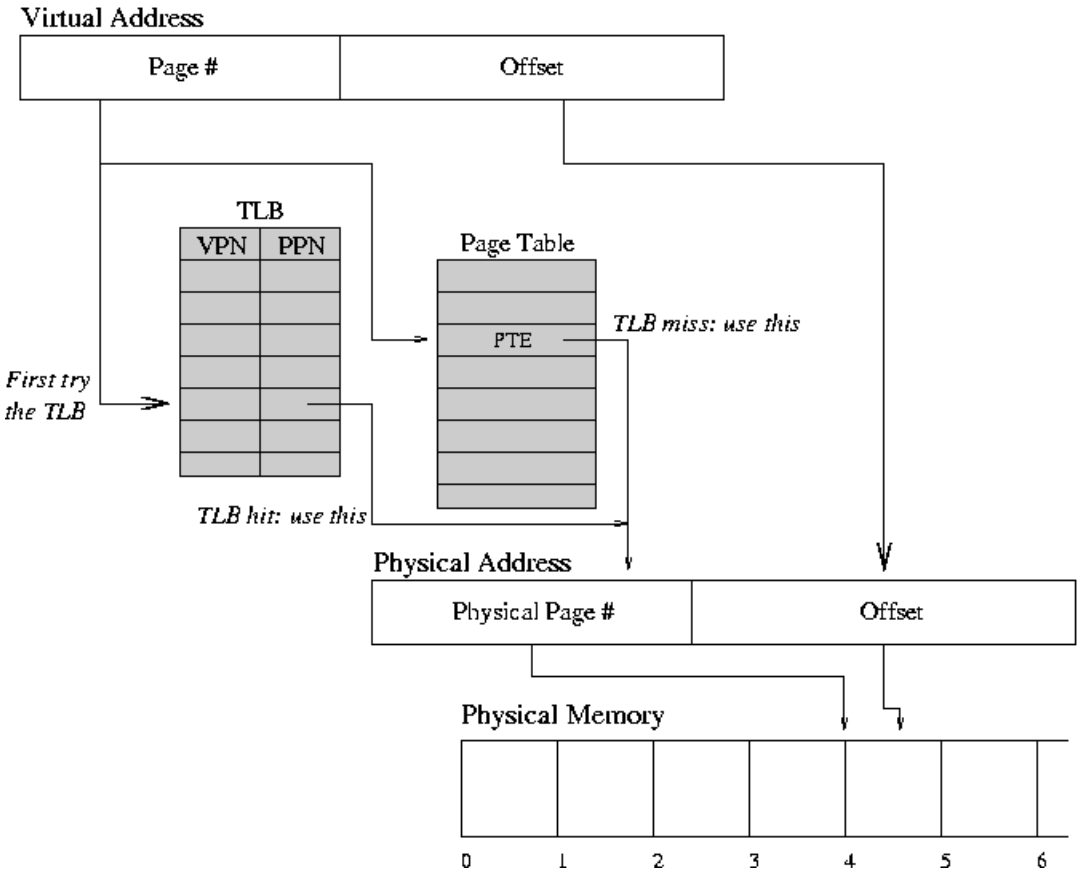


Image from: http://pages.cs.wisc.edu/~bart/537/lecturenotes/s17.html

Multi-Level Page Table

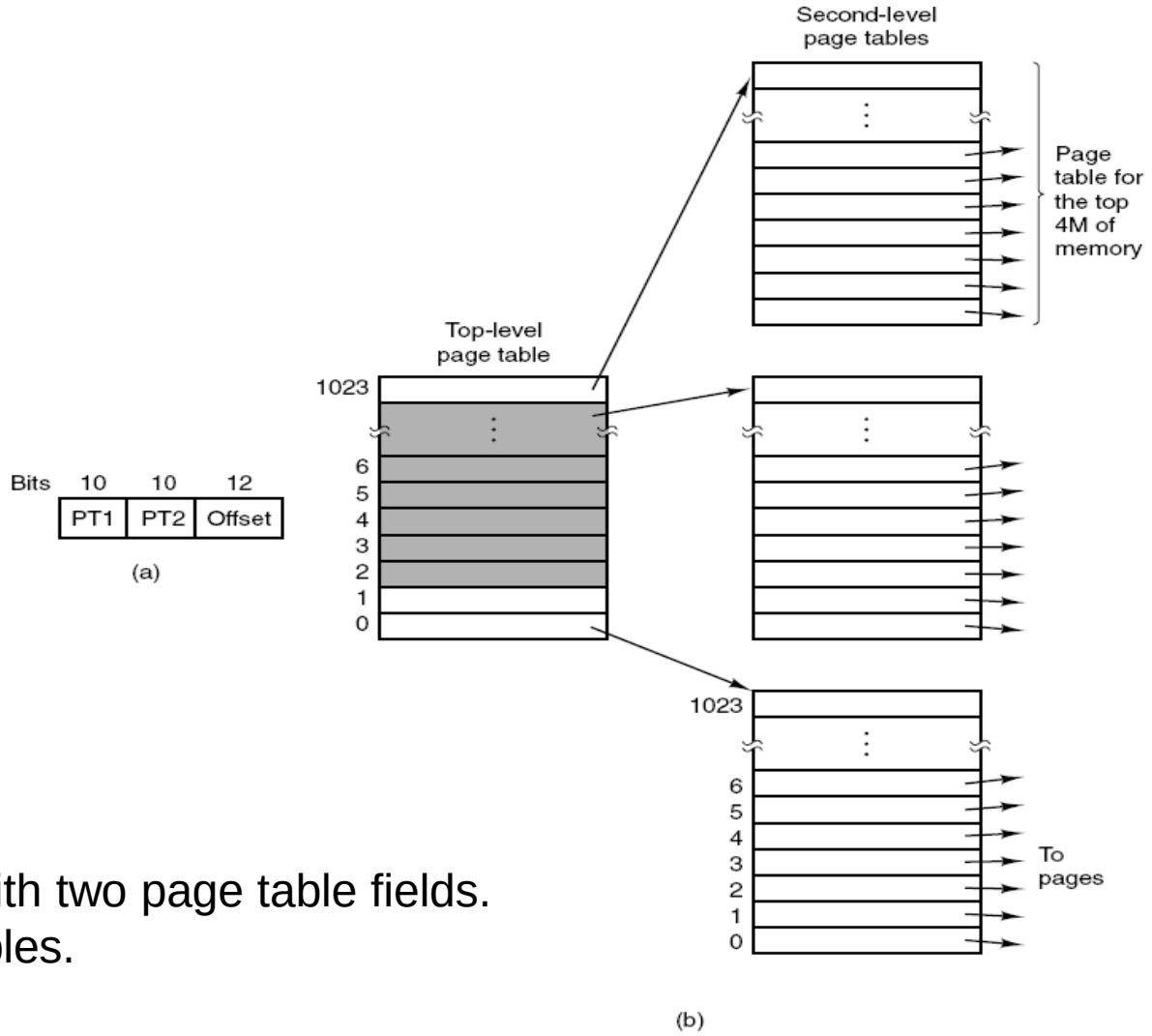


Figure 3-13.
(a) A 32-bit address with two page table fields.
(b) Two-level page tables.

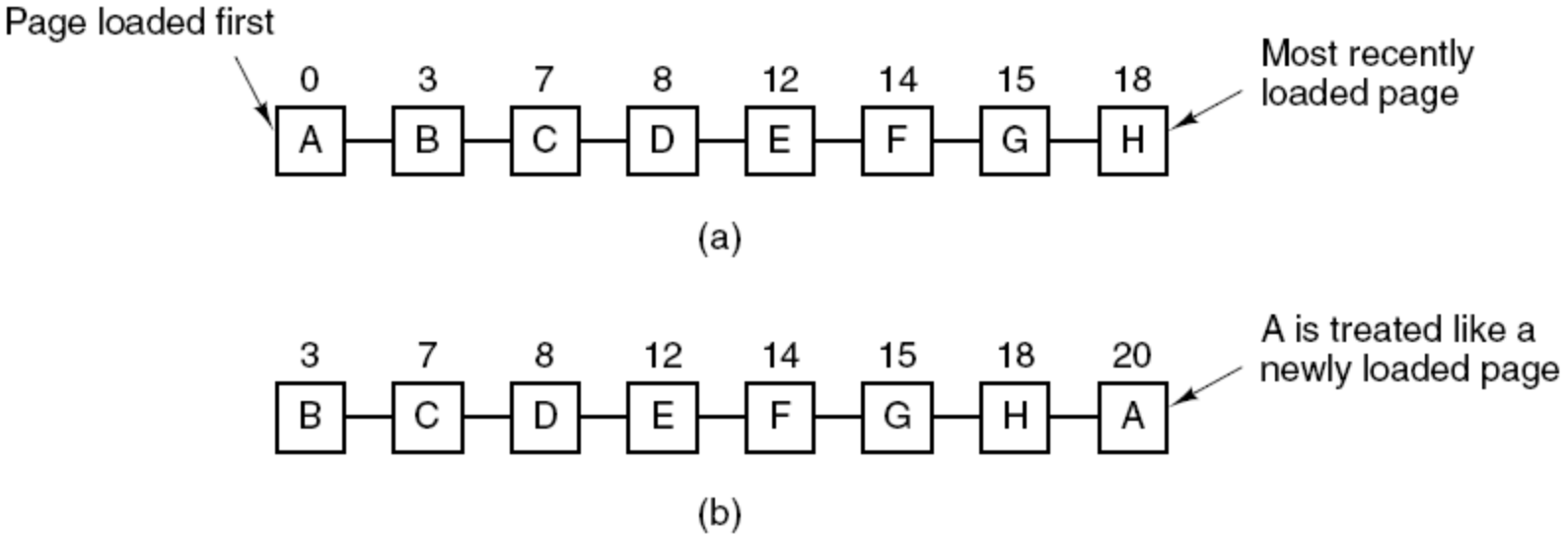
Page Replacement Algorithms

NRU

- **N**ot **R**ecently **U**sed (NRU)
  - Every period (typically ~20 msec) the referenced bit is zeroed.
  - It is set back when referencing this page.
  - One of the pages of the lowest class is replaced:
    - Class 0: not referenced, not modified.
    - Class 1: not referenced, modified.
    - Class 2: referenced, not modified.
    - Class 3: referenced, modified.

FIFO

- FIFO – a LL in FIFO order. The oldest page is dropped.
- Second chance algorithm – a FIFO variation: If the referenced bit is set – the page is moved to the back of the list, the bit is zeroed, and another page is dropped.



LRU

- **L**east **R**ecently **U**sed
  - When page faults – the page that has been unused for the longest time is thrown out
  - A large 64-bit counter may be used per page table entry, (incremented after each memory reference)
  - HW shall find the smallest value, after page fault occurs

Page Replacement Algorithms Summary

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Workingset	Somewhat expensive to implement
WSClock	Good efficient algorithm

Sharing by means of Paging

Shared Pages

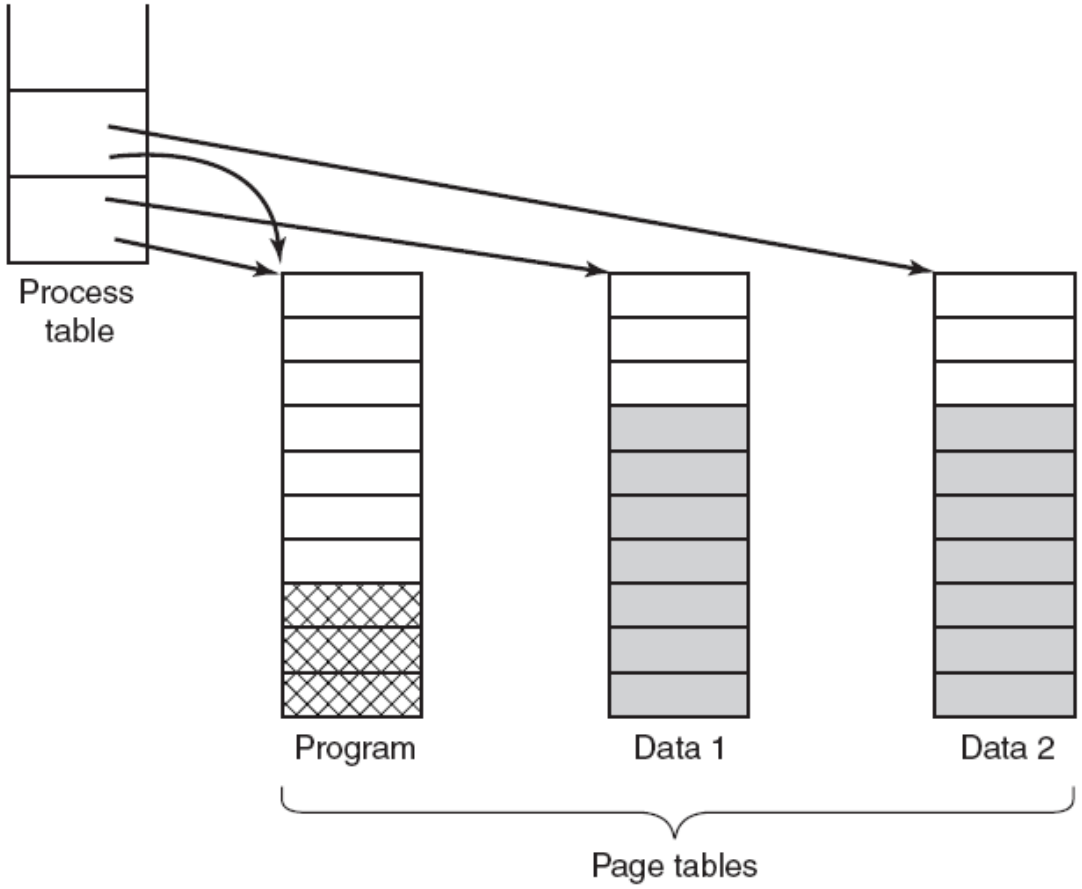


Figure 3-26. Two processes sharing the same program sharing its page table

Shared Pages (II)

- Both PTs (of each process) point to the same page frames (Text + Data) signed as R/O.
- At fork() time:
  - No coping of pages is done.
  - All data pages marked as R/O.
- **Lazy copy**
  - While updating Data (by either process), a copy of the Data is made (signed as R/W)
  - Also called COW – Copy On Write

Shared Libraries

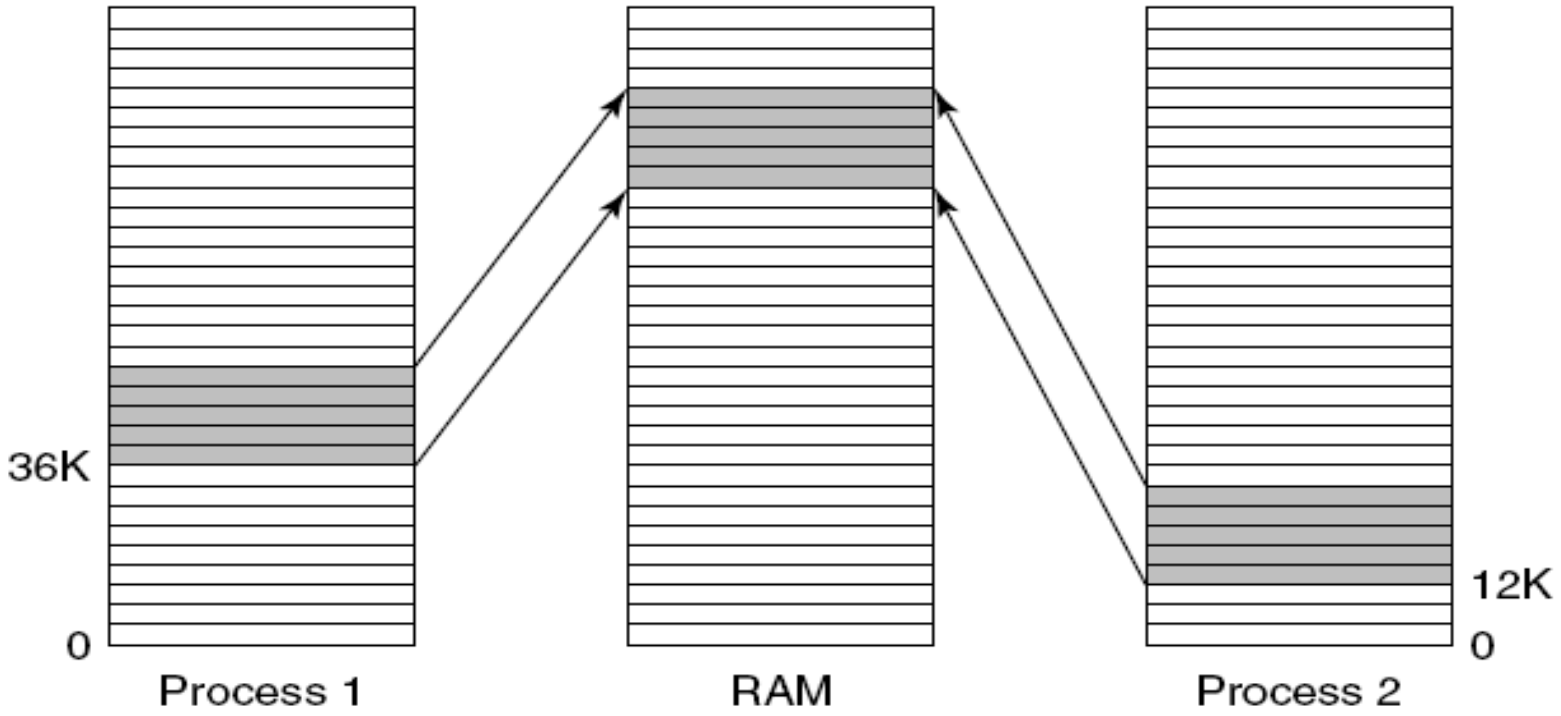


Figure 3-27. A shared library being used by two processes

Shared Libraries (II)

- The linker includes a small stub routine that binds to the called function at run-time
- Problem:
  The library is located at different address in each process. So how it can handle the command: jmp to address 16 ?
- Solution:
  - Relocation on the fly – no option (why?)
  - **Position-Independent Code**:
    Avoid use of absolute address – use relative offset

Caching

Cache – HW assistance

- What is Cache Memory?
  - A smaller, faster memory.
  - Stores copies of the most frequently used data (from main memory).
- Typical usage:
  - Instruction cache – to speed up executable instruction fetch.
  - Data cache – to speed up data fetch and store.
  - Translation Look-aside Buffer – to speed up virtual-to-physical address translation for both executable instructions and data.

Cache (II)

- Cache levels:
  - Level 1 (L1) – Inside the CPU, 16 KB, No delay
  - Level 2 (L2) – Outside the CPU (AMD/Intel), several MB, 1-2 clock cycles delay
- Cache lines:
  - Main memory is divided up into cache lines
  - Typically 64 bytes each one
  - e.g. 4096 cash lines of 64 bytes
- Cache hit/miss

Associative Memory

- **MMU** checks for cache hit
  - In how many places should it check ?
- Associativity ~= number of places to check
  - Checking more places:
    Takes more power, chip area, and potentially time.
  - Fewer misses.
- Cheap to expensive:
  - Direct mapped
  - 2-way set
  - 4-way set associative cache
  - Fully associative cache

Processor & Cache Architectures

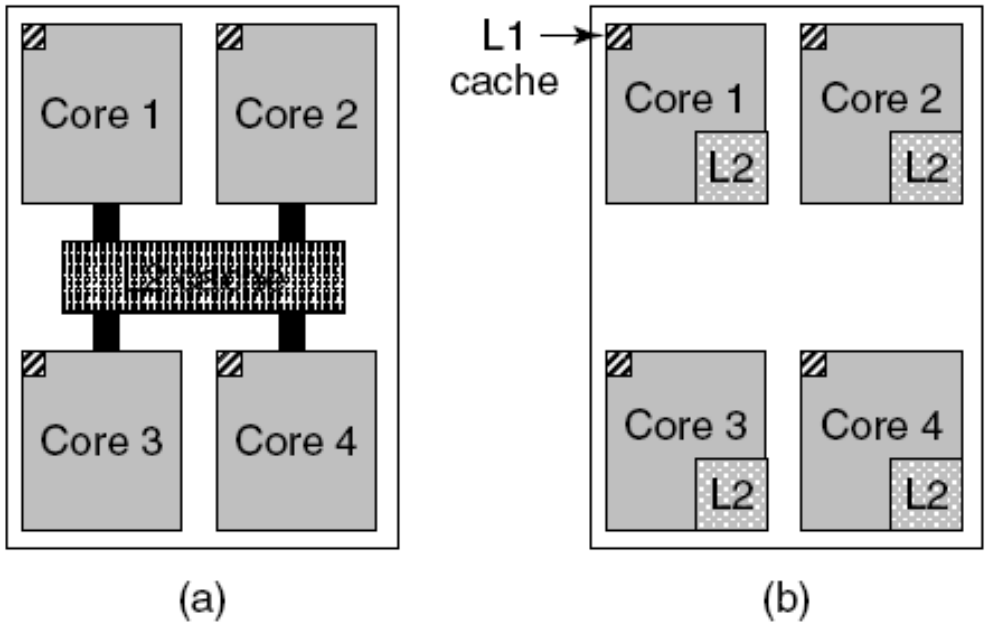


Figure 1-8. (a) A quad-core chip with a shared L2 cache.
(b) A quad-core chip with separate L2 caches.