Operating Systems

Chapter 3 Memory Management

Tien Pham Van, Dr. rer. nat.

(*Lecture compiled with reference to other presentations*)

Concept

- Memory management is the act of managing computer memory.
- It involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed.
- The management of main memory is critical to the computer system, particular embedded systems

Memory Management Requirements

- Relocation
  - Programmer does not know where the program will be placed in memory when it is executed
  - While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
  - Memory references must be translated in the code to actual physical memory address

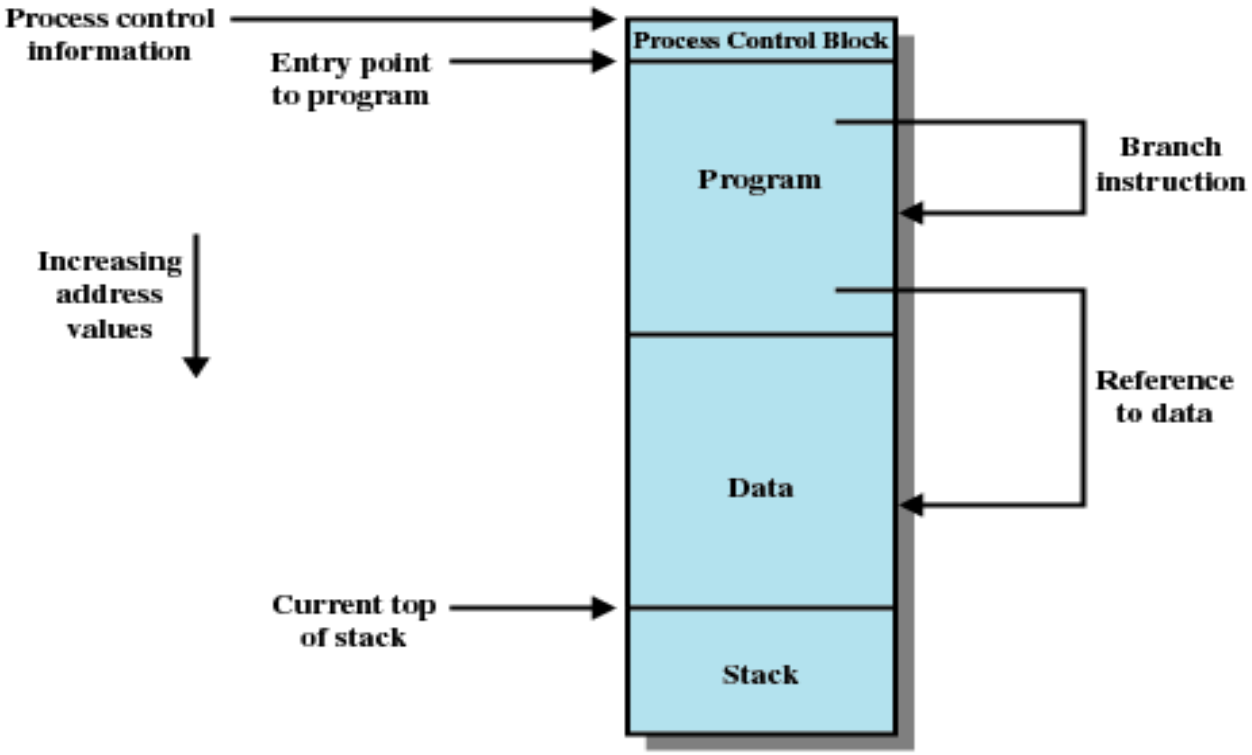


Figure 7.1 Addressing Requirements for a Process

Memory Management Requirements

- Protection
  - Processes should not be able to reference memory locations in another process without permission
  - Impossible to check absolute addresses at compile time
  - Must be checked at run time
  - Memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software)
    - Operating system cannot anticipate all of the memory references a program will make

Memory Management Requirements

- Sharing
  - Allow several processes to access the same portion of memory
  - Better to allow each process access to the same copy of the program rather than have their own separate copy

Memory Management Requirements

- Logical Organization
  - Programs are written in modules
  - Modules can be written and compiled independently
  - Different degrees of protection given to modules (read-only, execute-only)
  - Share modules among processes

- Operating Systems 2 - Memory Manager - YouTube

https://www.youtube.com/watch?v=qdkxXygc3rE&t=195s

Memory Management Requirements

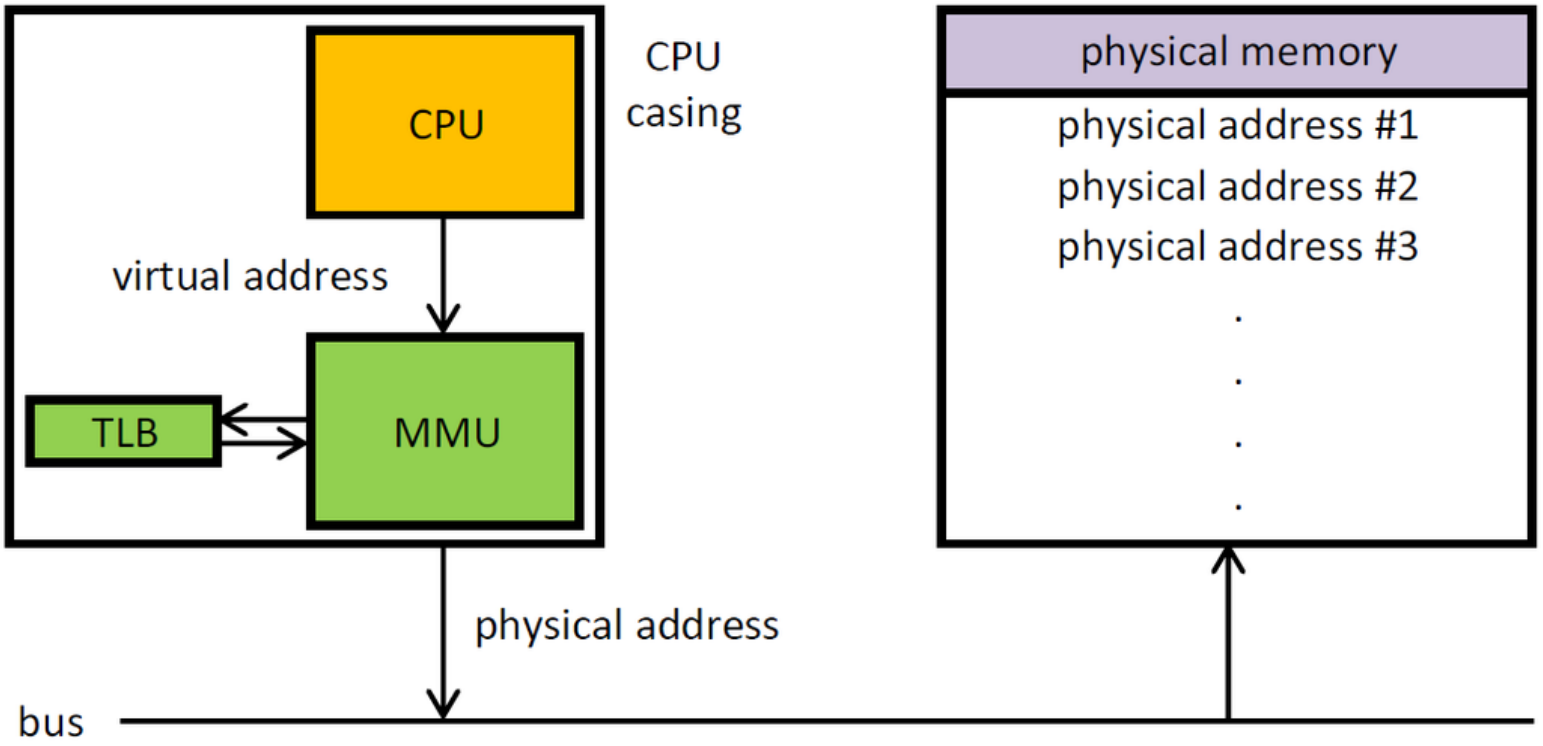
- Physical Organization
  - Memory available for a program plus its data may be insufficient
    - Overlaying allows various modules to be assigned the same region of memory
  - Programmer does not know how much space will be available

Overview clip

- Address Binding,Address Translation and Memory Management Unit Tutorial-2 - YouTube

https://www.youtube.com/watch?v=_kPCbBGRl1o&list=PLskQvPDUk0sJnmLgi4qBRyshlmHydbsAJ&index=3

MMU



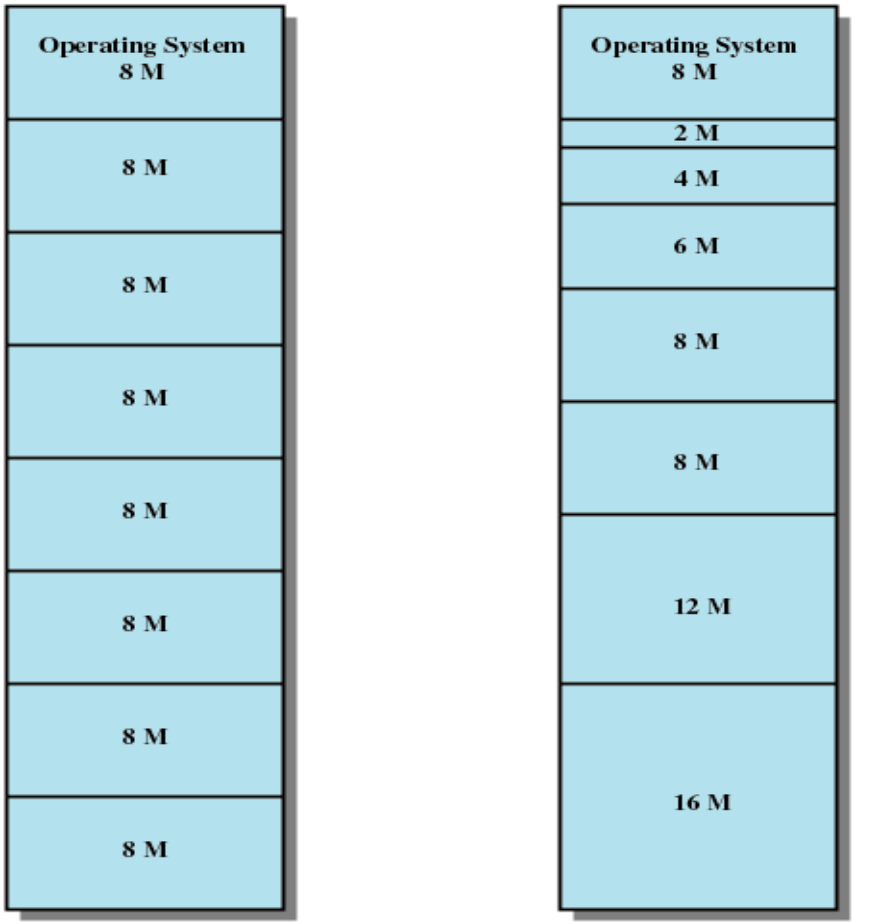
CPU: Central Processing Unit

MMU: Memory Management Unit

TLB: Translation lookaside buffer

Fixed Partitioning

- Equal-size partitions
  - Because all partitions are of equal size, it does not matter which partition is used
  - Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation.
- Unequal-size partitions
  - Can assign each process to the smallest partition within which it will fit
  - Queue for each partition
  - Processes are assigned in such a way as to minimize wasted memory within a partition



(a) Equal-size partitions

(b) Unequal-size partitions

Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
- Eventually get holes in the memory. This is called external fragmentation
- Must use compaction to shift processes so they are contiguous and all free memory is in one block

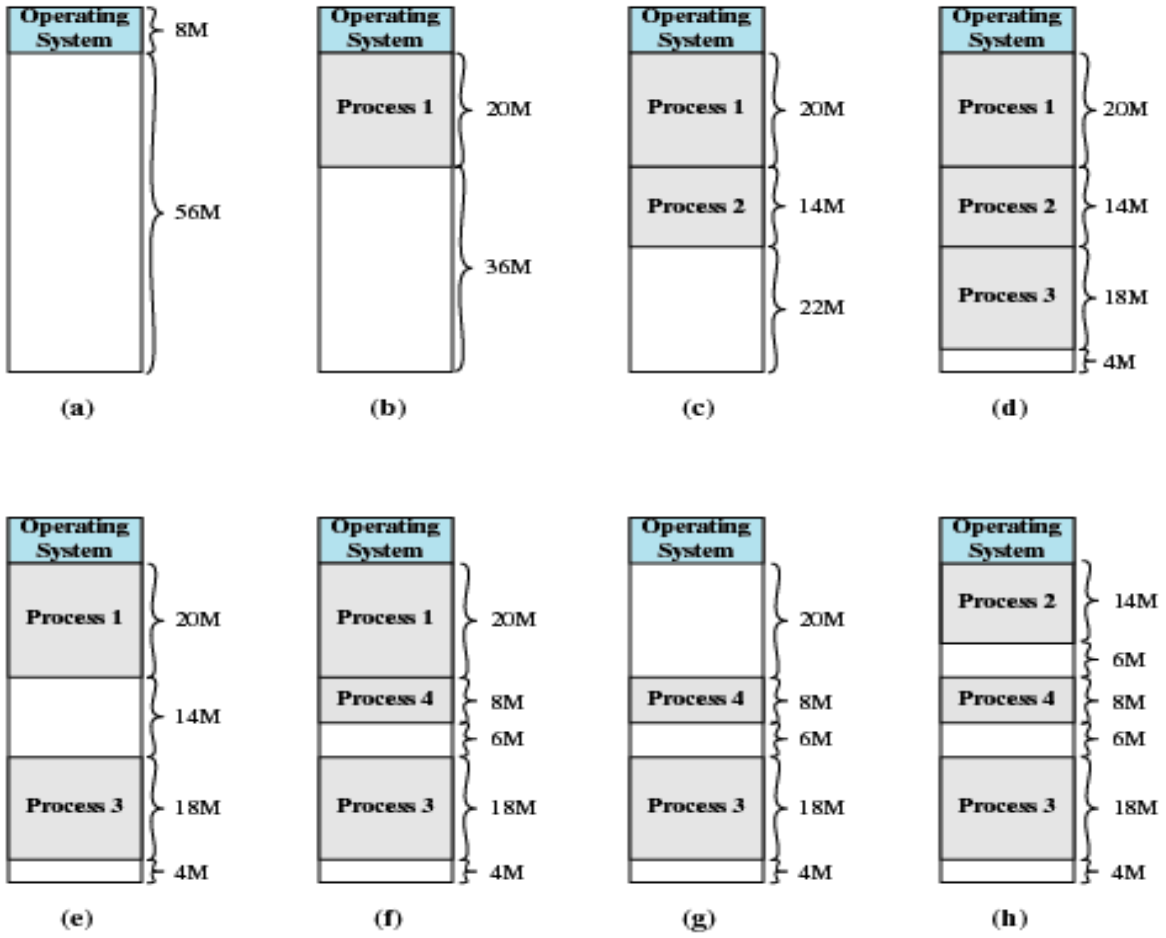


Figure 7.4 The Effect of Dynamic Partitioning

Dynamic Partitioning Placement Algorithm

- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often

Dynamic Partitioning Placement Algorithm

- First-fit algorithm
  - Scans memory form the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many process loaded in the front end of memory that must be searched over when trying to find a free block

Dynamic Partitioning Placement Algorithm

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory

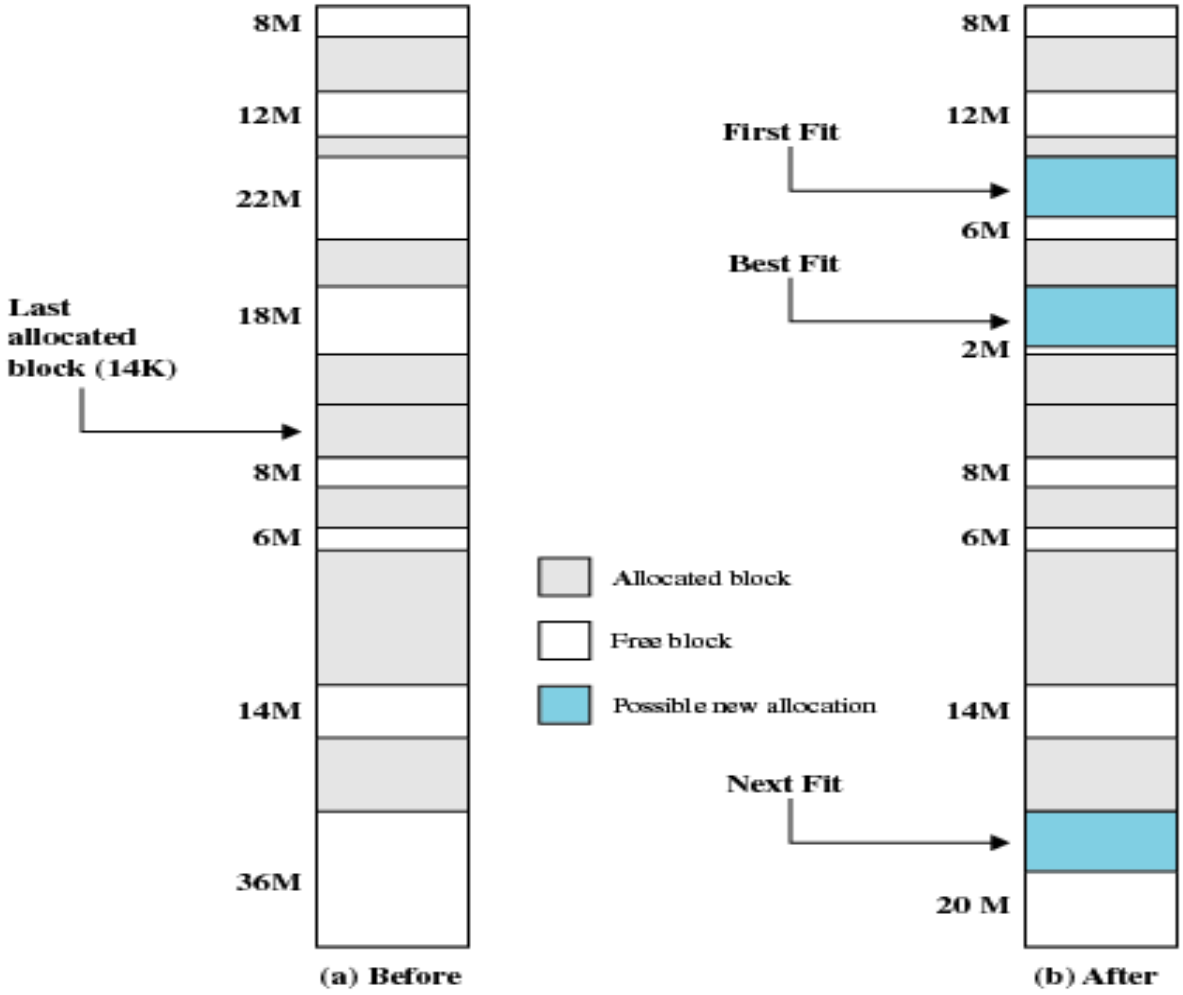


Figure 7.5 Example Memory Configuration Before and After Allocation of 16 Mbyte Block

- https://www.youtube.com/watch?v=ueV1gcYJSZg

Embedded Networking Research Group
Email: tien.phamvan1@hust.edu.vn
School of Elec. and Telecom - Hanoi University of Science and Technology
C9-411, Dai Co Viet str. 1, HBT, Hanoi
Tel: +84-243-8693596

Buddy System

- Entire space available is treated as a single block of 2^U
- If a request of size s such that $2^{U-1} < s <= 2^U$, entire block is allocated
  - Otherwise block is split into two equal buddies
  - Process continues until smallest block greater than or equal to s is generated

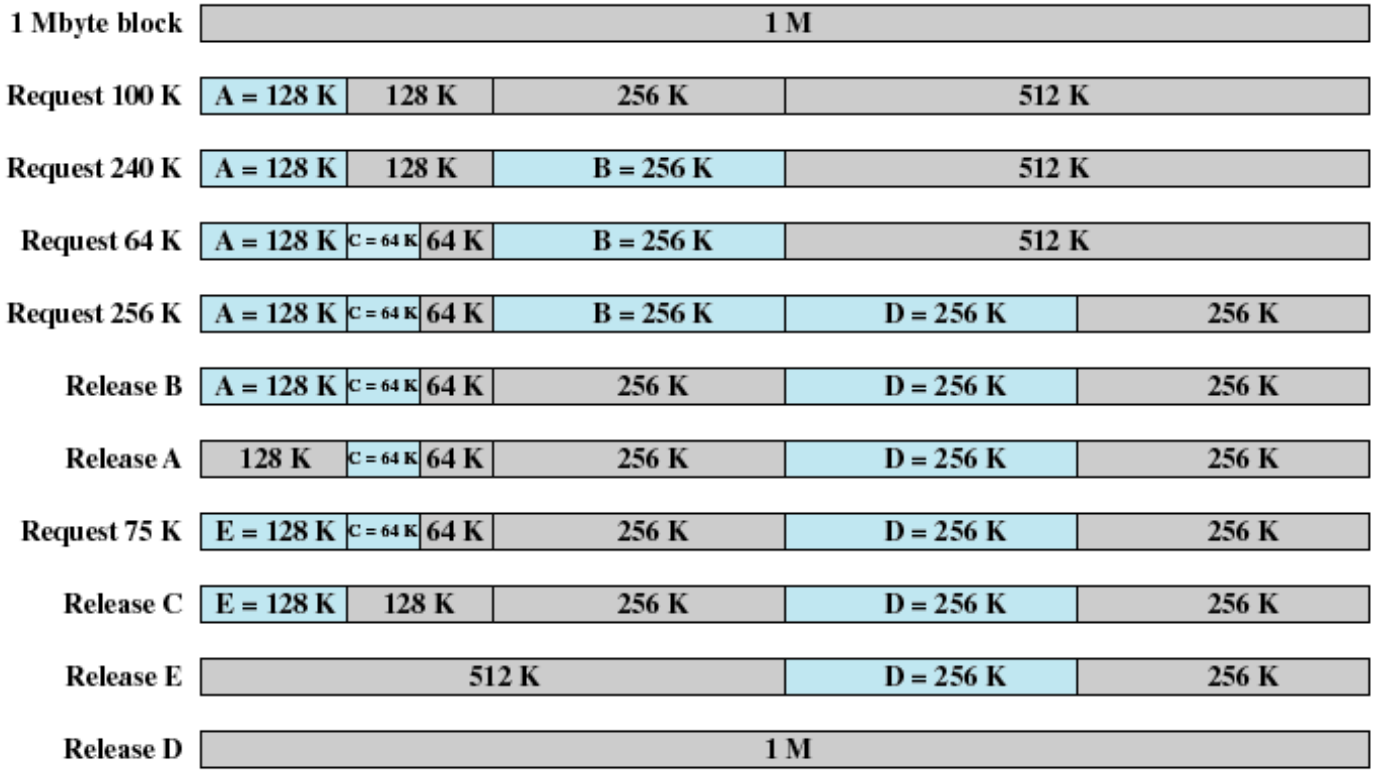


Figure 7.6 Example of Buddy System

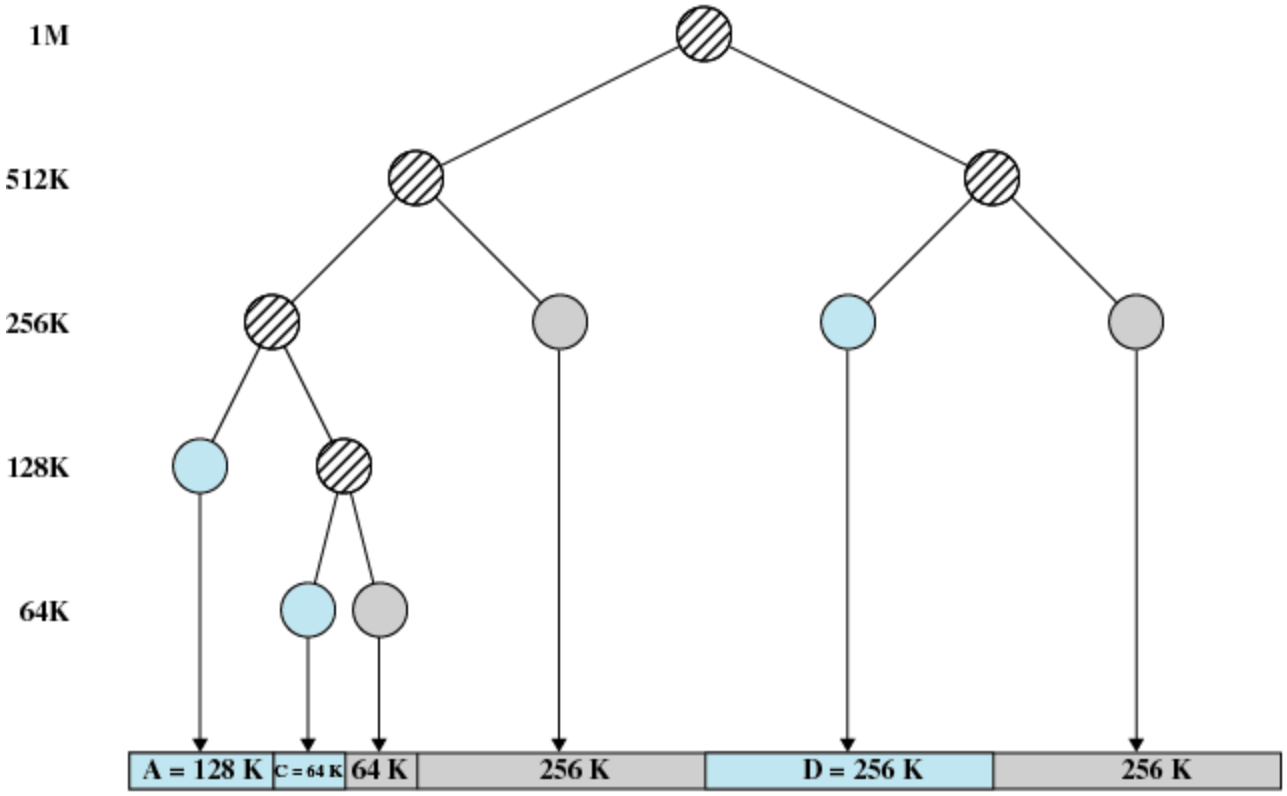


Figure 7.7 Tree Representation of Buddy System

Addresses

- Logical
  - Reference to a memory location independent of the current assignment of data to memory
  - Translation must be made to the physical address
- Relative
  - Address expressed as a location relative to some known point
- Physical
  - The absolute address or actual location in main memory

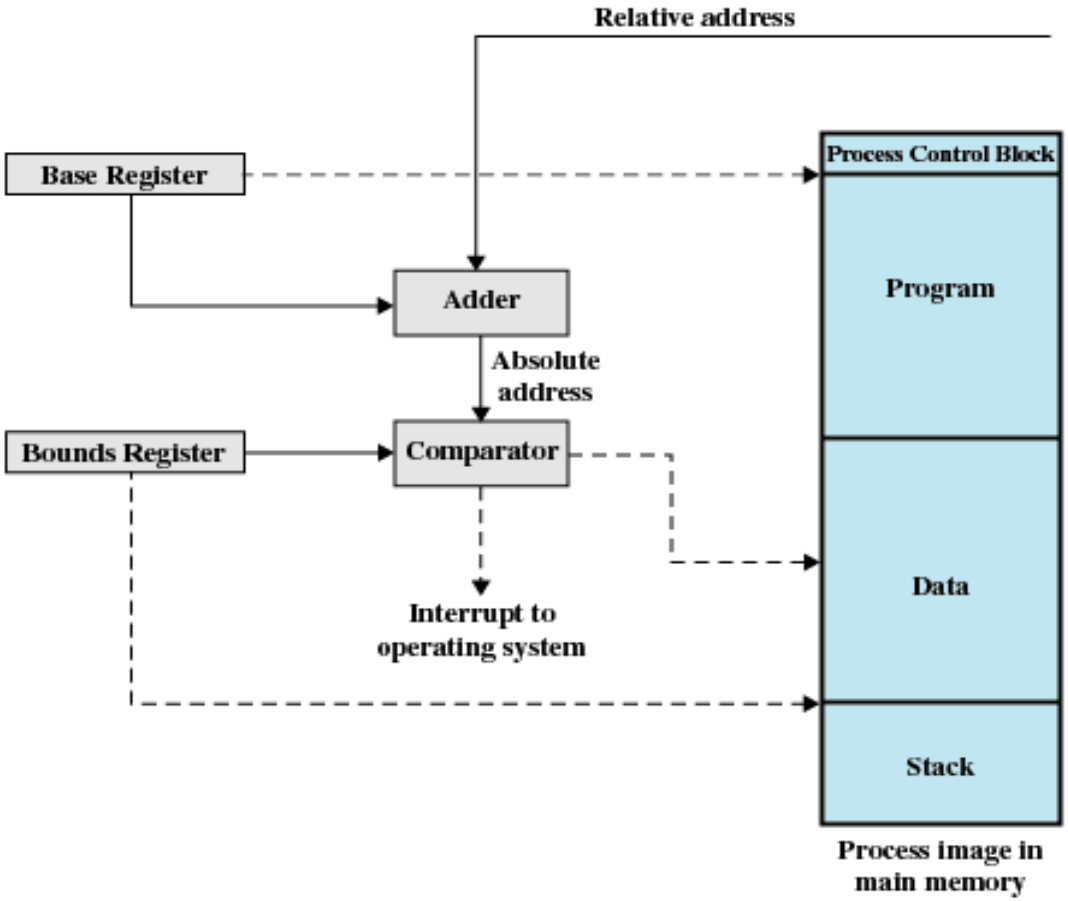


Figure 7.8 Hardware Support for Relocation

Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called pages and chunks of memory are called frames
- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a page number and offset within the page

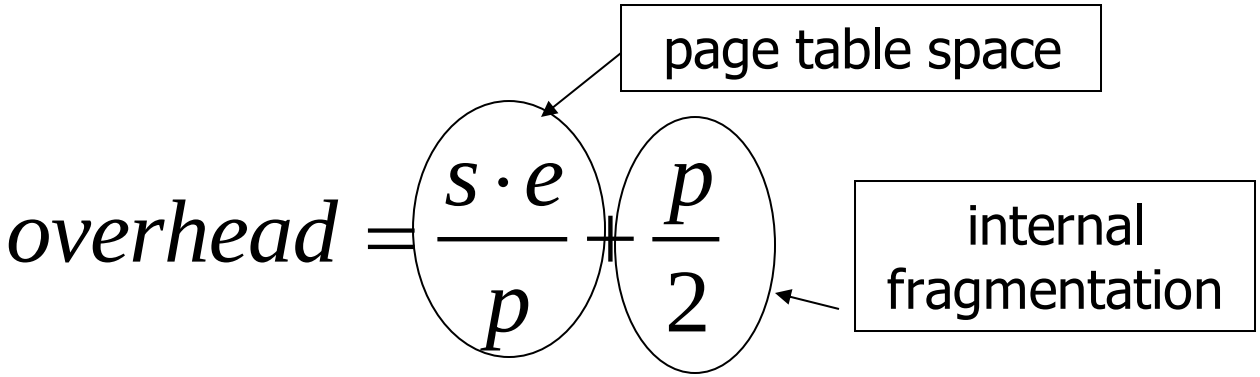
Page Size (1)

Small page size

- Advantages
  - less internal fragmentation
  - better fit for various data structures, code sections
  - less unused program in memory
- Disadvantages
  - programs need many pages, larger page tables

Page Size (2)

- Overhead due to page table and internal fragmentation



- Where
  - s = average process size in bytes
  - p = page size in bytes
  - e = page entry

Optimized when

$$p = \sqrt{2se}$$

Page size (3)

Architecture	Smallest page size	Larger page sizes
x86-64[18]	4 KiB	2 MiB, 1 GiB (only when the CPU has PDPE1GB flag)
Power ISA[21]	4 KiB	64 KiB, 16 MiB, 16 GiB
SPARC v8 with SPARC Reference MMU[22]	4 KiB	256 KiB, 16 MiB
UltraSPARC Architecture 2007[23]	8 KiB	64 KiB, 512 KiB (optional), 4 MiB, 32 MiB (optional), 256 MiB (optional), 2 GiB (optional), 16 GiB (optional)
ARMv7[24]	4 KiB	64 KiB, 1 MiB ("section"), 16 MiB ("supersection") (defined by a particular implementation)

Page size (4)

```
#include <stdio.h>
#include <unistd.h> /* sysconf(3) */

int main(void)
{
        printf("The page size for this system is %ld bytes.\n",
                sysconf(_SC_PAGESIZE)); /* _SC_PAGE_SIZE is OK too. */

        return 0;
}
```

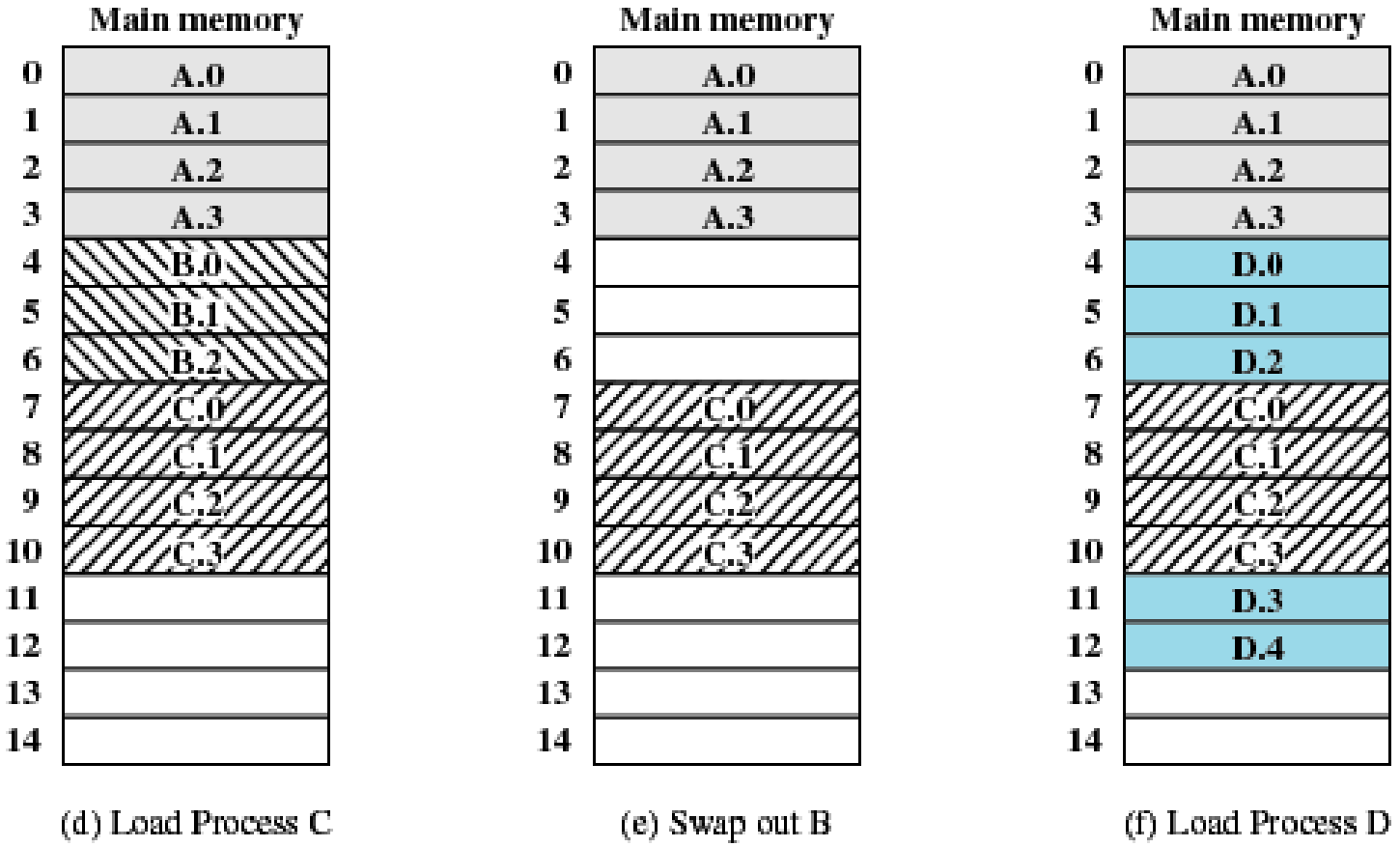


Figure 7.9 Assignment of Process Pages to Free Frames

- https://www.youtube.com/watch?v=AKGtJAi4wGo&ab_channel=Xoviabcs

https://www.youtube.com/watch?v=AKGtJAi4wGo

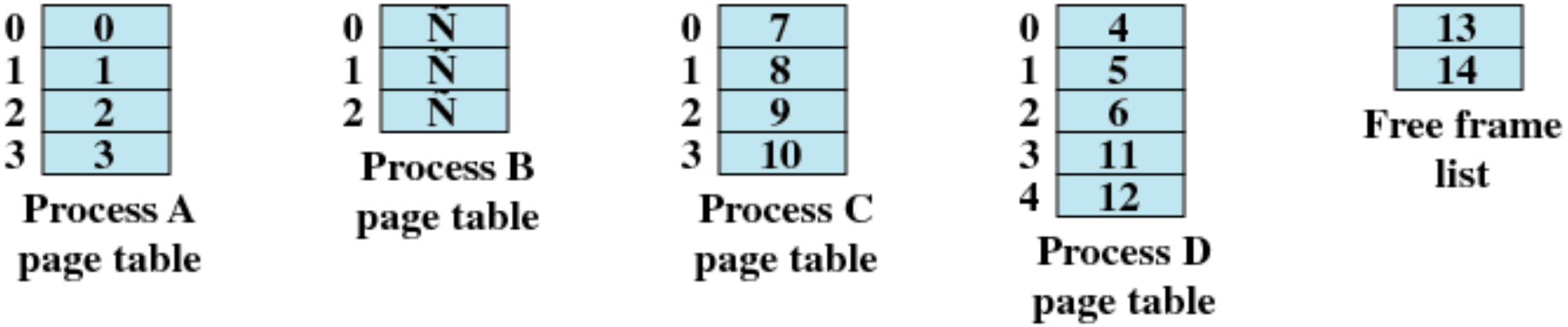
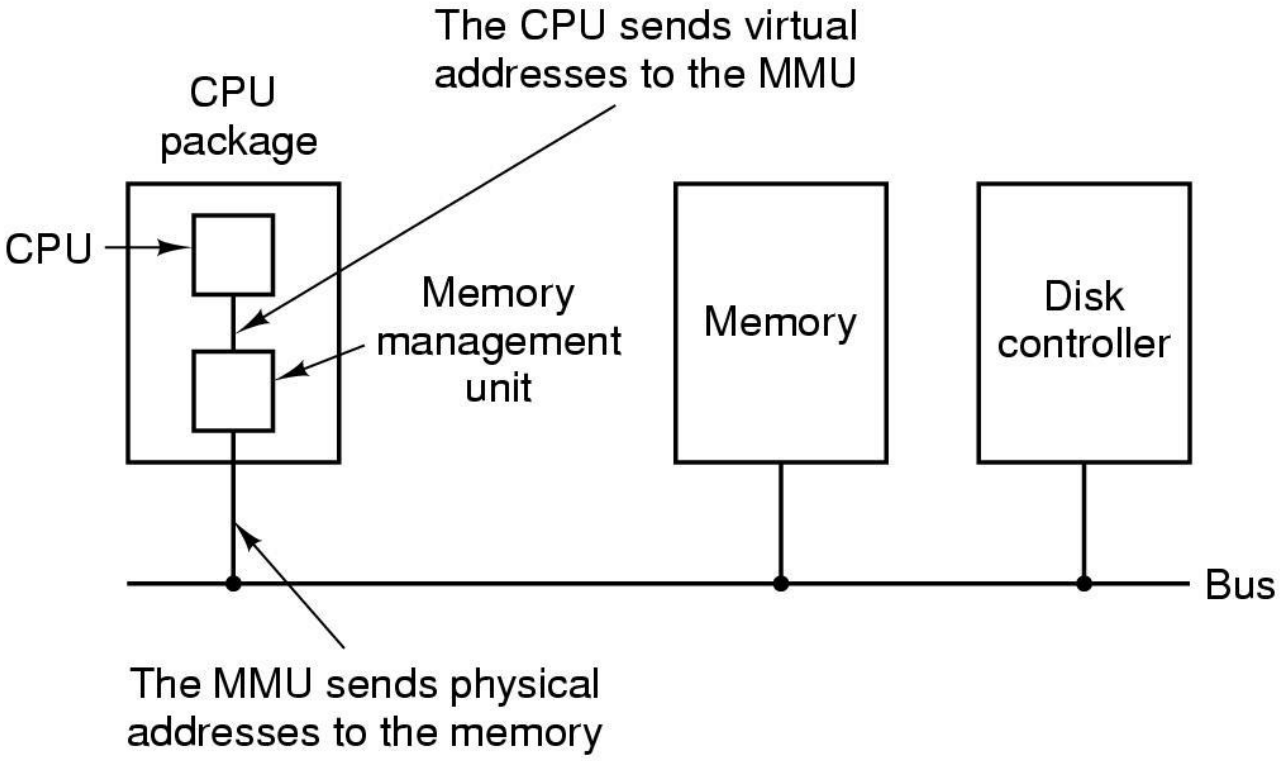


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

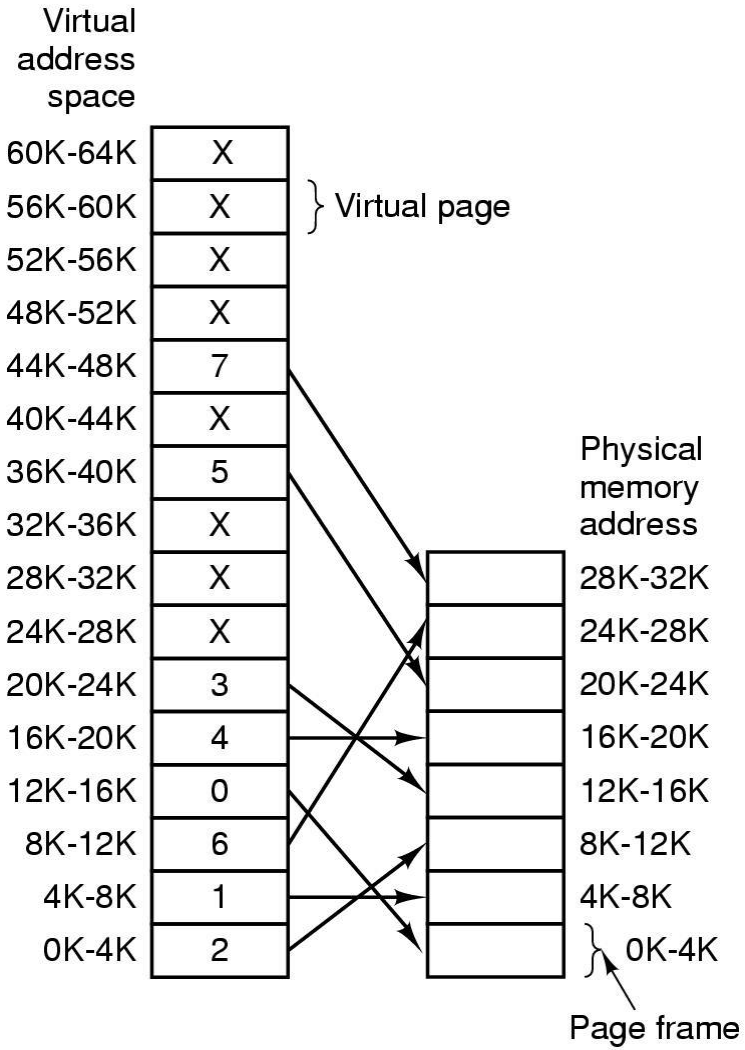
Virtual Memory Paging



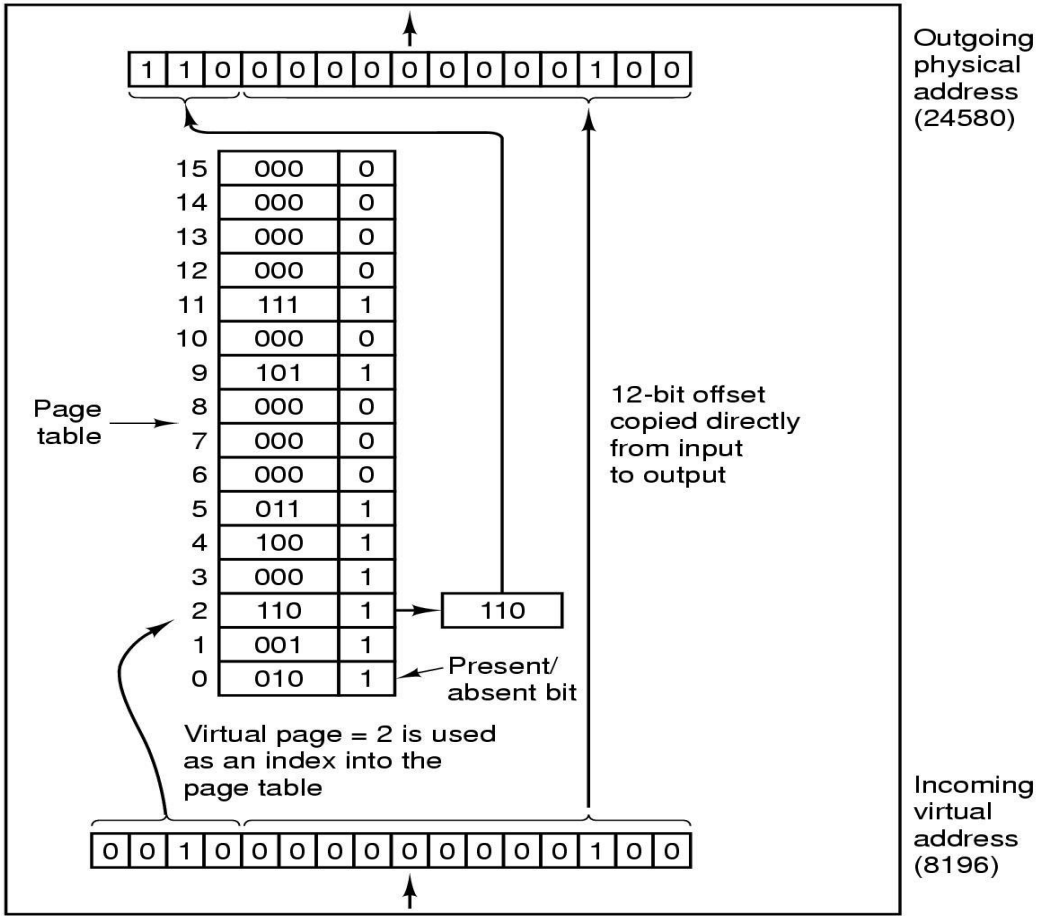
The position and function of the MMU

Paging and translation

The relation between virtual addresses and physical memory addresses given by page table



Internal Operation of MMU with 16 4 KB Pages*



16 bit addresses => address space size: 2^{16}

Page size ~4K ~ 2^{12} => $2^{16} / 2^{12} = 2^4 = 16$ pages.

Segmentation

- All segments of all programs do not have to be of the same length
- There is a maximum segment length
- Addressing consist of two parts - a segment number and an offset
- Since segments are not equal, segmentation is similar to dynamic partitioning

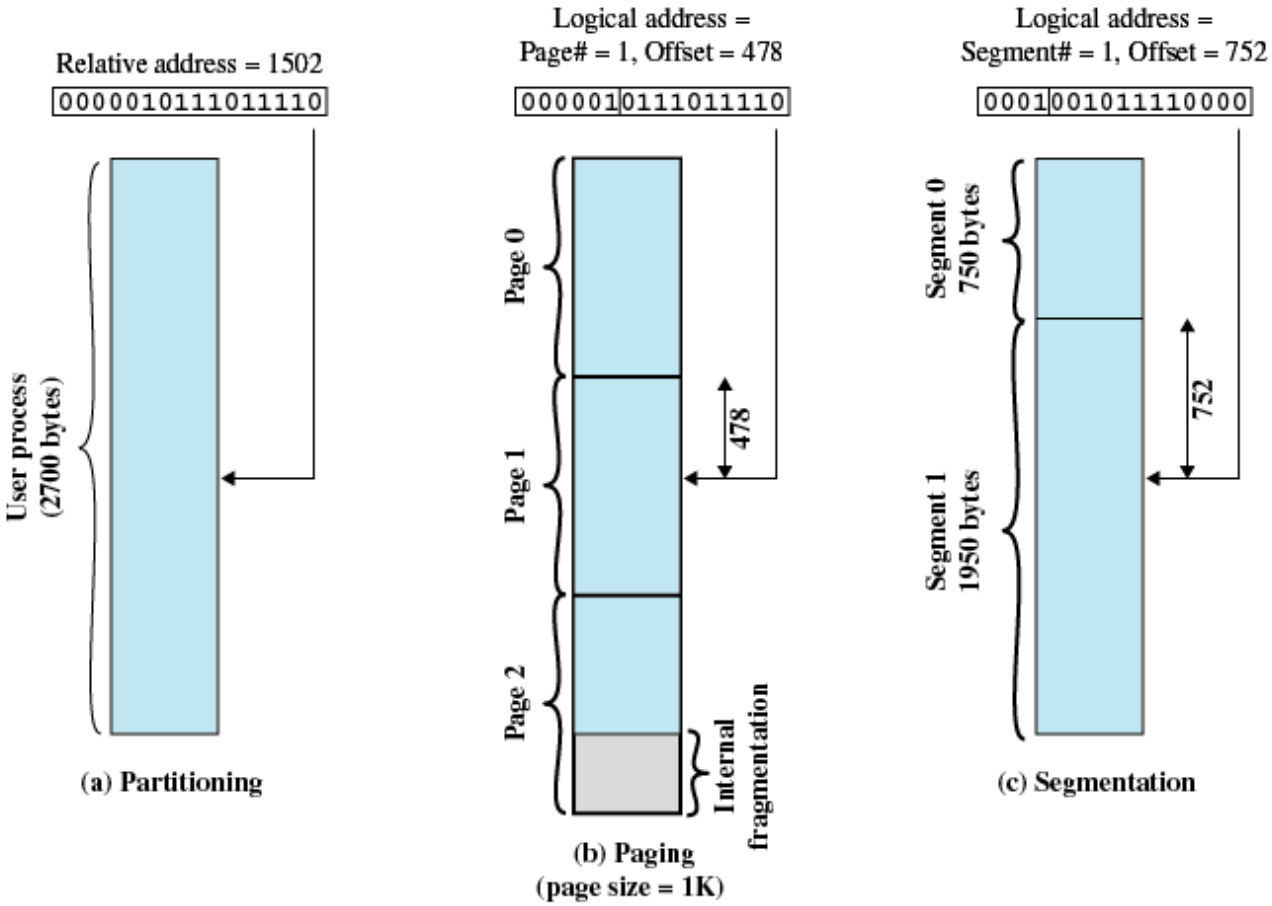
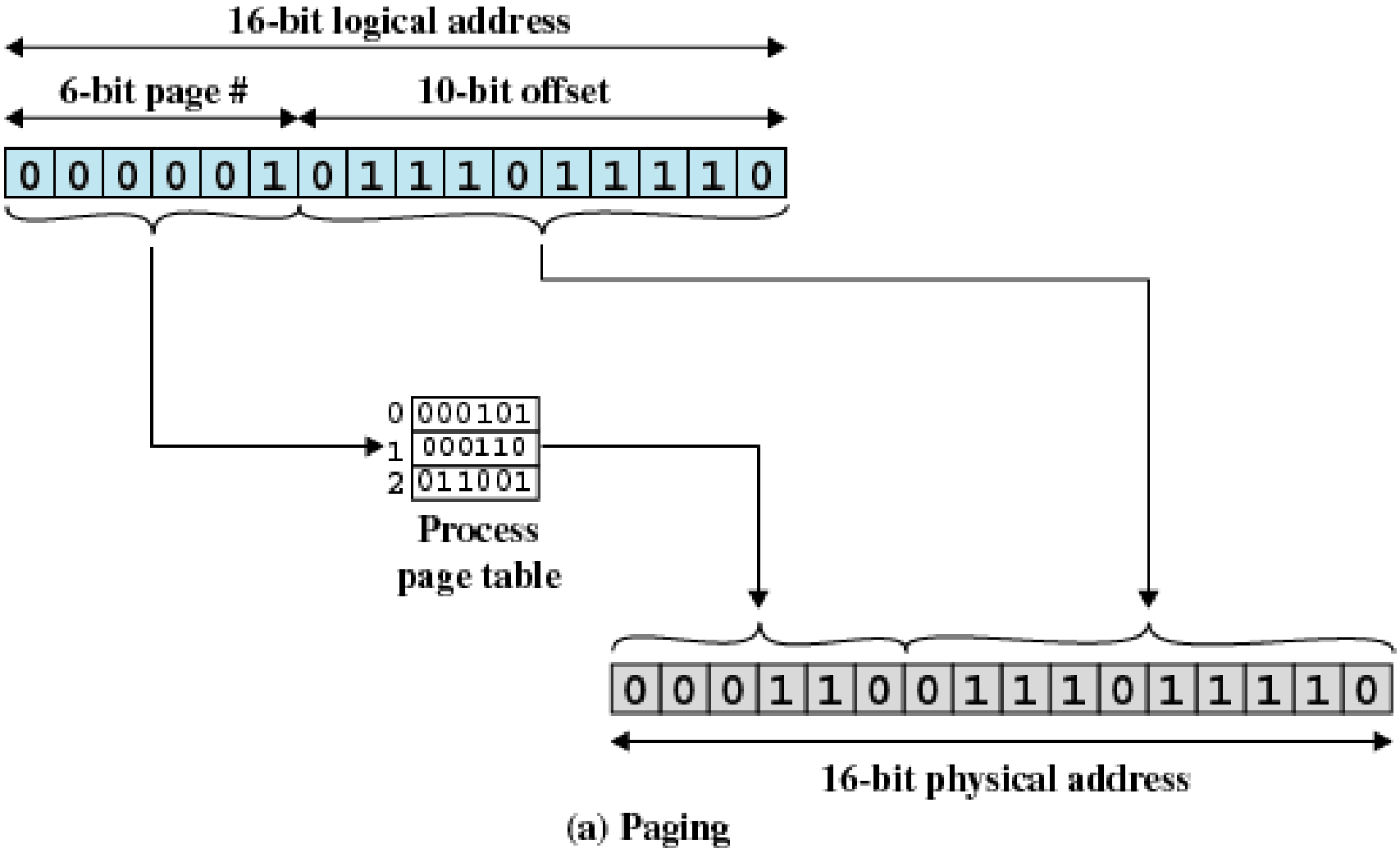


Figure 7.11 Logical Addresses


16-bit logical address
6-bit page #
10-bit offset
0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 0
0 000101
1 000110
2 011001
Process page table
0 0 0 1 1 0 0 1 1 1 0 1 1 1 1 0
16-bit physical address

(a) Paging

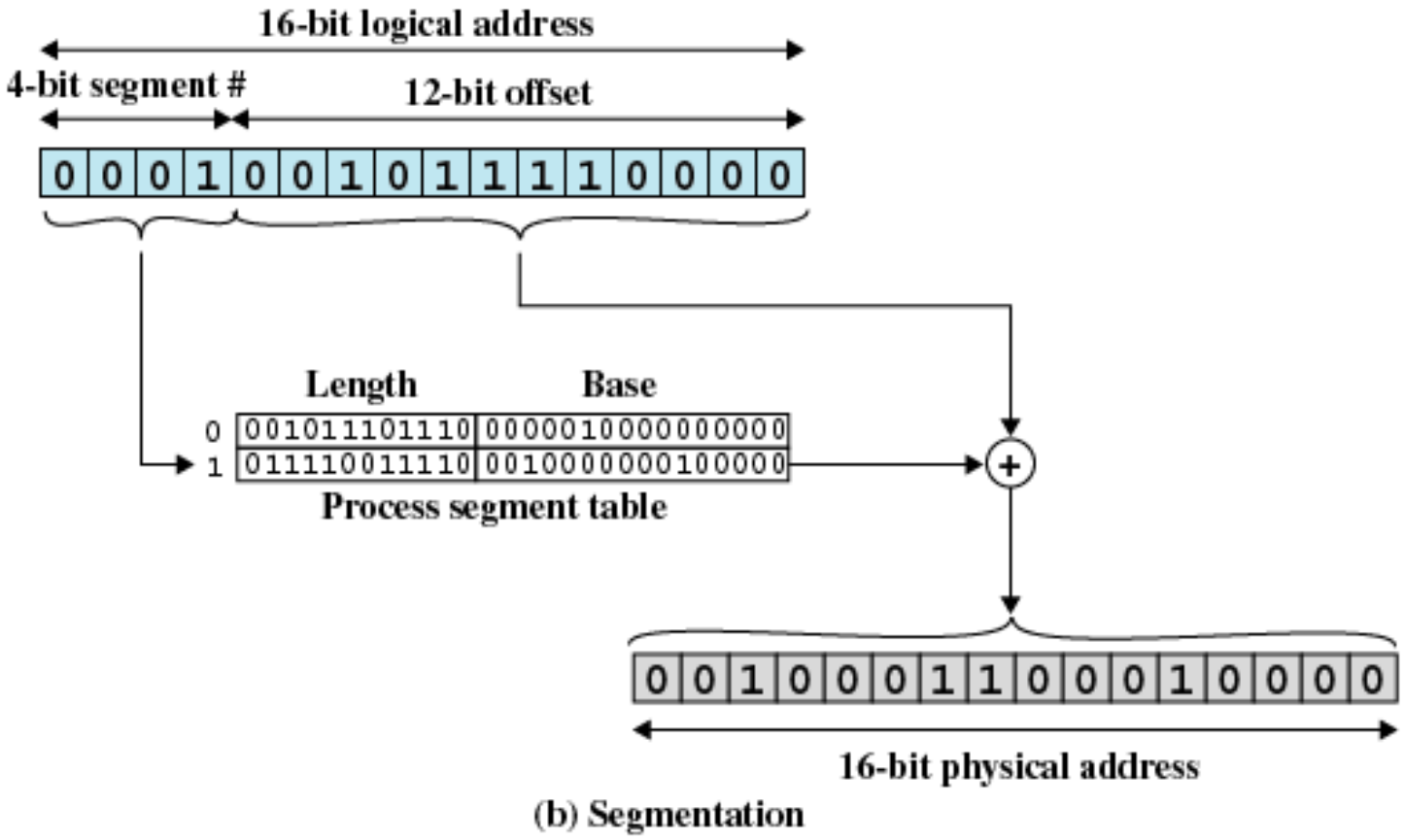
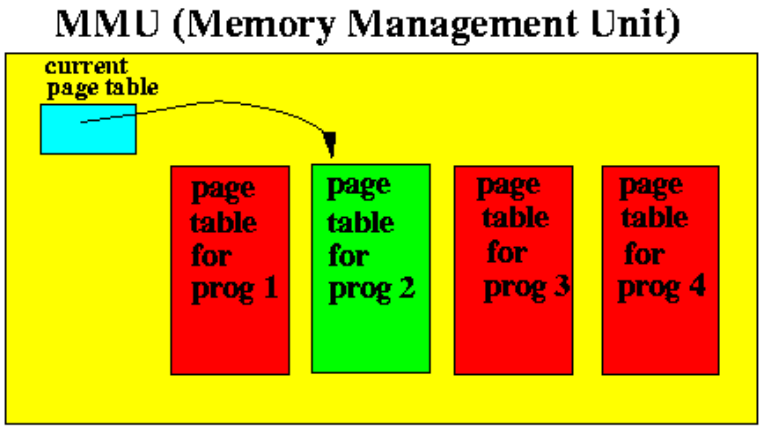
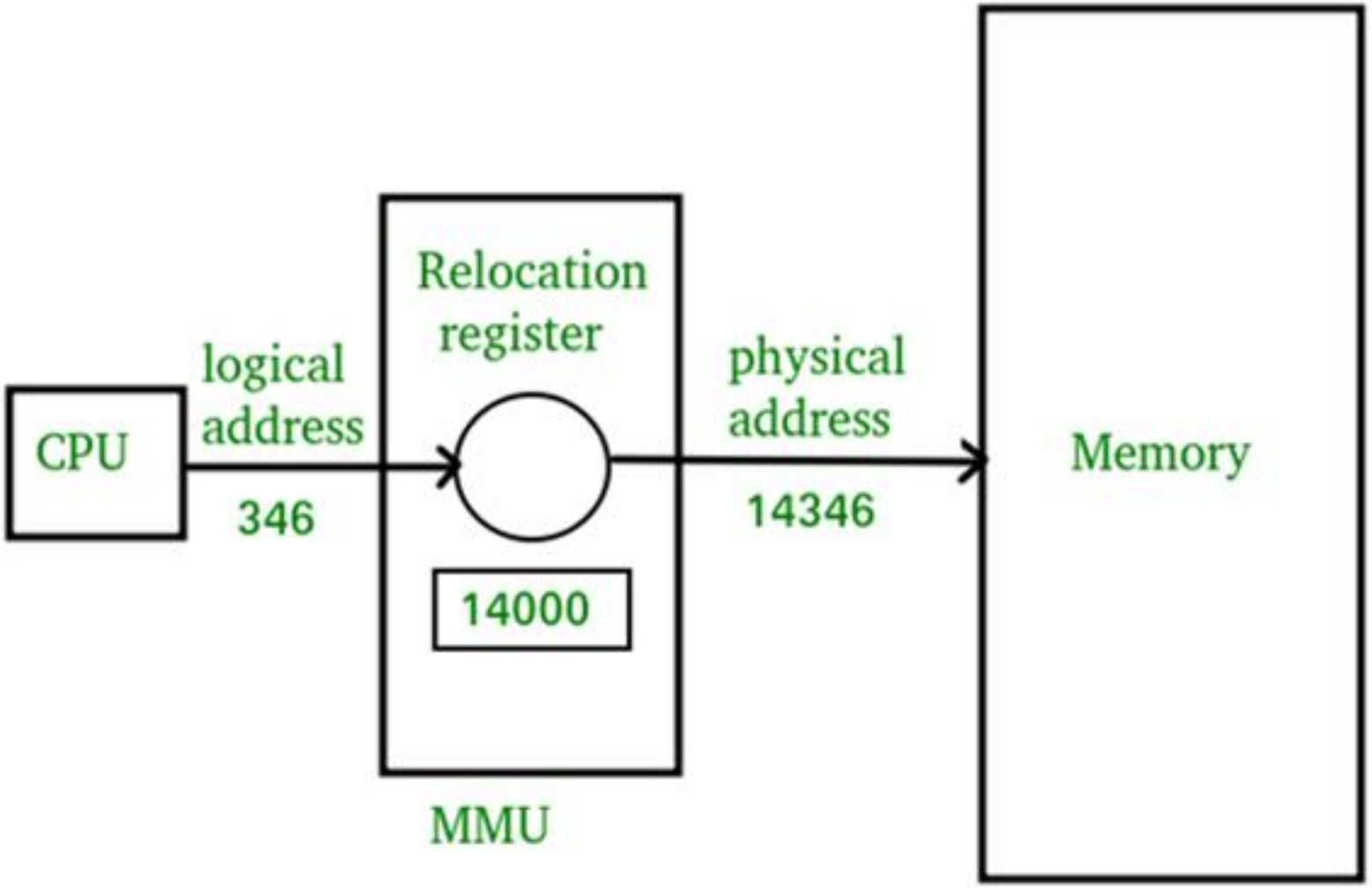


Figure 7.12 Examples of Logical-to-Physical Address Translation

Hardware Support

- For each process, a pointer to the page table is stored with the other register values (like the instruction counter) in the PCB
- When the dispatcher starts a process, it must reload all registers and copy the stored page table values into the hardware page table in the MMU.




Relocation
register
logical
address
CPU
346
physical
address
14346
Memory
14000
MMU

Solutions to Large Page Table Problems

The MMU contains only a **Page-Table Base Register** which points to the page table. Changing page tables requires changing only this one register, substantially reducing context switch time. However this is very slow! The problem with the PTBR approach, where the page table is kept in memory, is that TWO memory accesses are needed to access one user memory location: one for the page-table entry and one for the byte. This is **intolerably slow** in most circumstances. Practically no better than swapping!

Solutions to Large Page Table Problems (cont.)

Multilevel page tables avoid keeping one huge page table in memory all the time: this works because most processes use only a few of its pages frequently and the rest, seldom if at all. Scheme: the page table itself is paged.

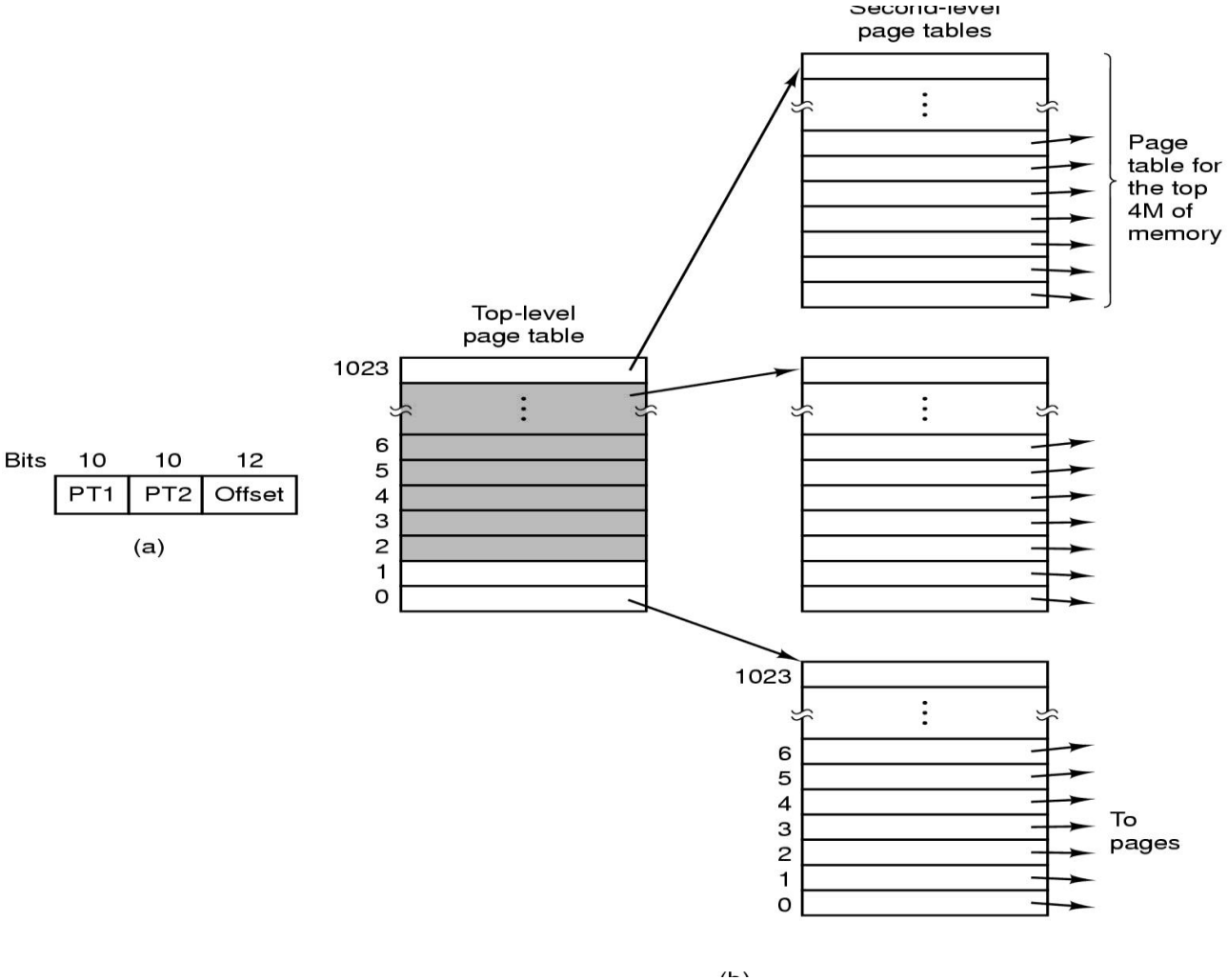
Example: Using 32 bit addressing:

The top-level table contains 1,024 pages (indices). The entry at each index contains the page frame number of a 2nd-level page table. This index (or page number) is found in the 10 highest (leftmost) bits in the virtual address generated by the CPU.

The next 10 bits in the address hold the index into the 2nd-level page table. This location holds the page frame number of the page itself.

The lowest 12 bits of the address is the offset, as usual.

Two-level Page Tables



32 bit address with 2 page table fields

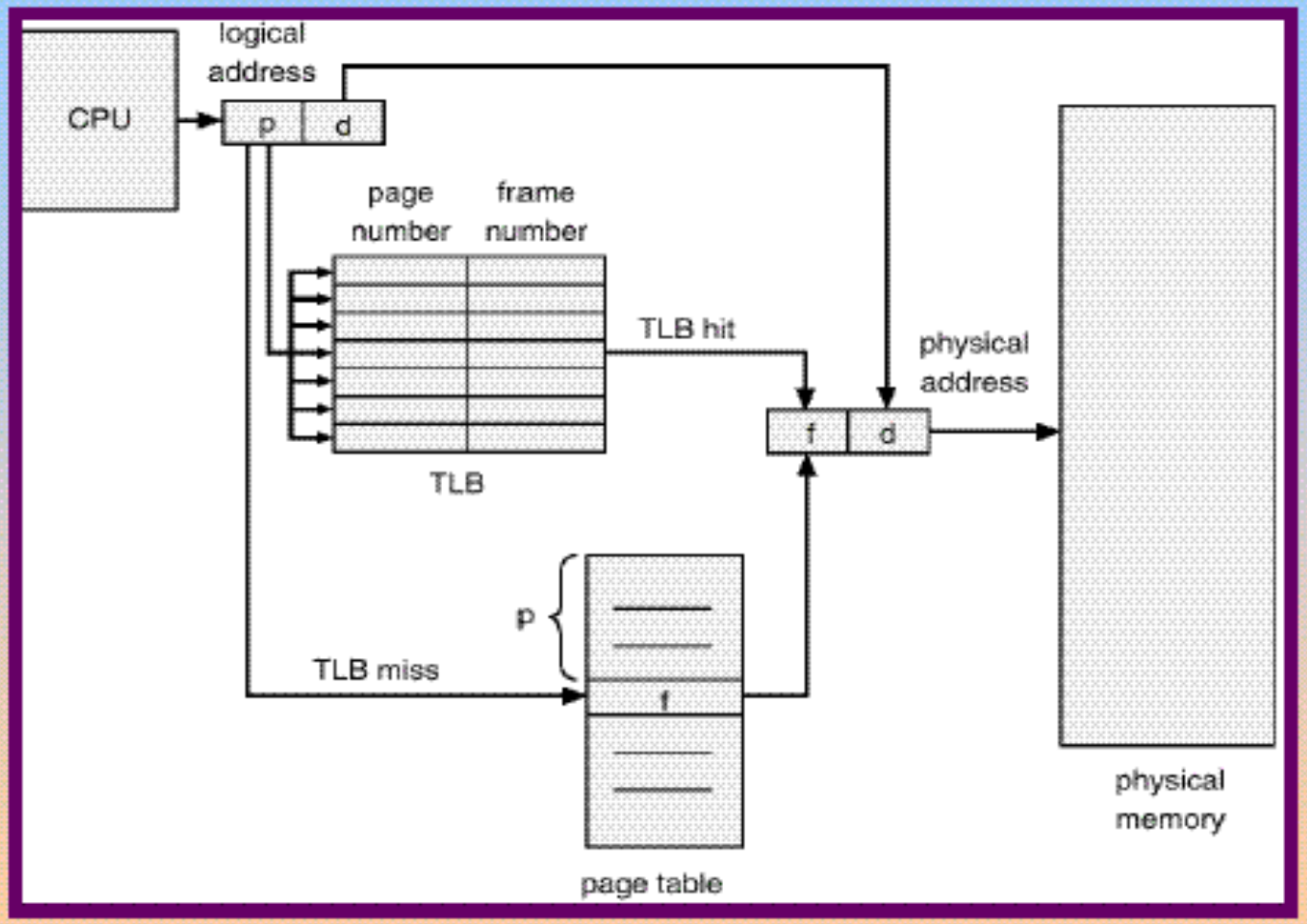
Revisit to clip: https://www.youtube.com/watch?v=AKGtJAi4wGo

Solutions to Large Page Table Problems (cont.)

A small, fast lookup cache called the **TRANSLATION LOOK-ASIDE BUFFER (TLB)** or ASSOCIATIVE MEMORY.

The TLB is used along with page tables kept in memory. When a virtual address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame is immediately available and used to access memory. If the page number is not in the TLB (a miss) a memory reference to the page table must be made. This requires a trap to the operating system. When the frame number is obtained, it is used to access memory AND the page number and frame number are added to the TLB for quick access on the next reference. This procedure may be handled by the MMU, but today it is often handled by software; i.e. the operating system.

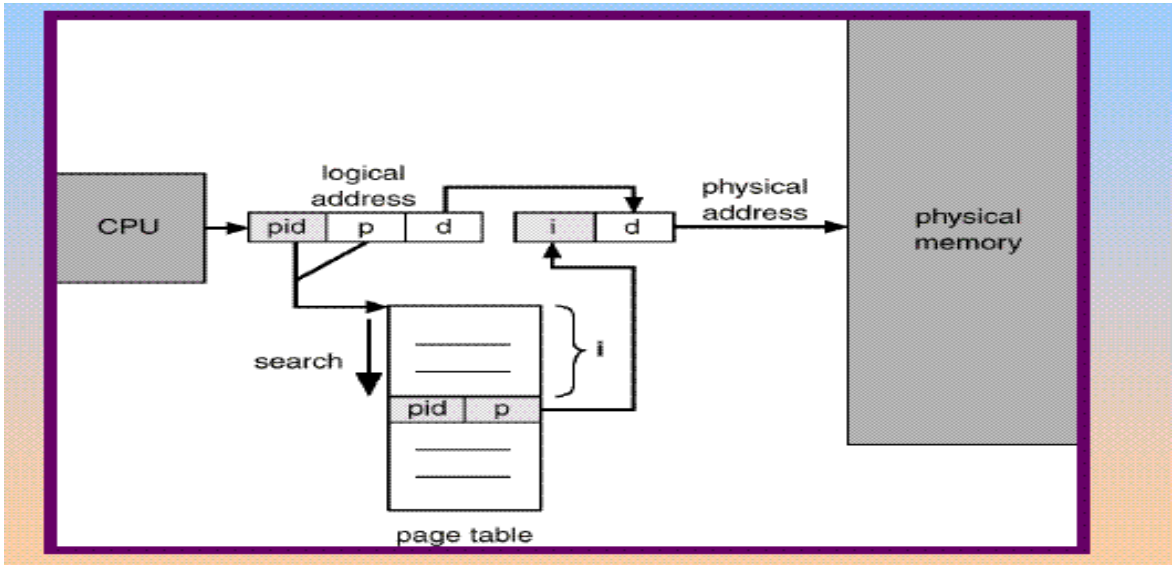
Paging Hardware With TLB



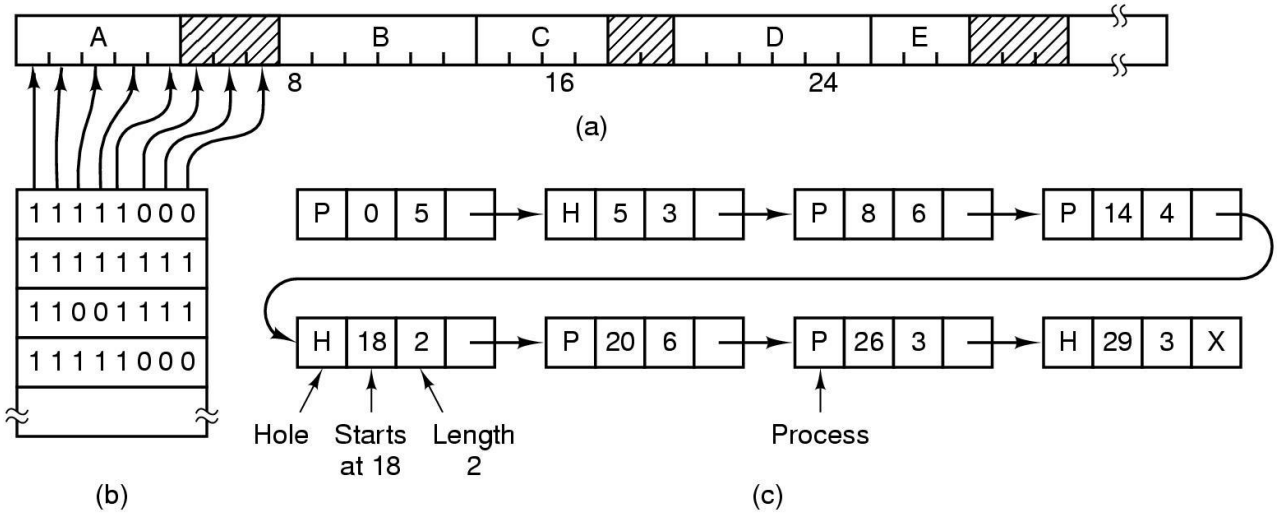
Solutions to Large Page Table Problems (cont.)

- For larger addressing, such as 64 bits, even multi-level page tables are not satisfactory: just too much memory would be taken up by page tables and references would be too slow.

- One solution is the **Inverted Page Table**. In this scheme there is not one page table for each process in the system, but only one page table for all processes in the system. This scheme would be very slow alone, but is workable along with a TLB and sometimes a hash table.

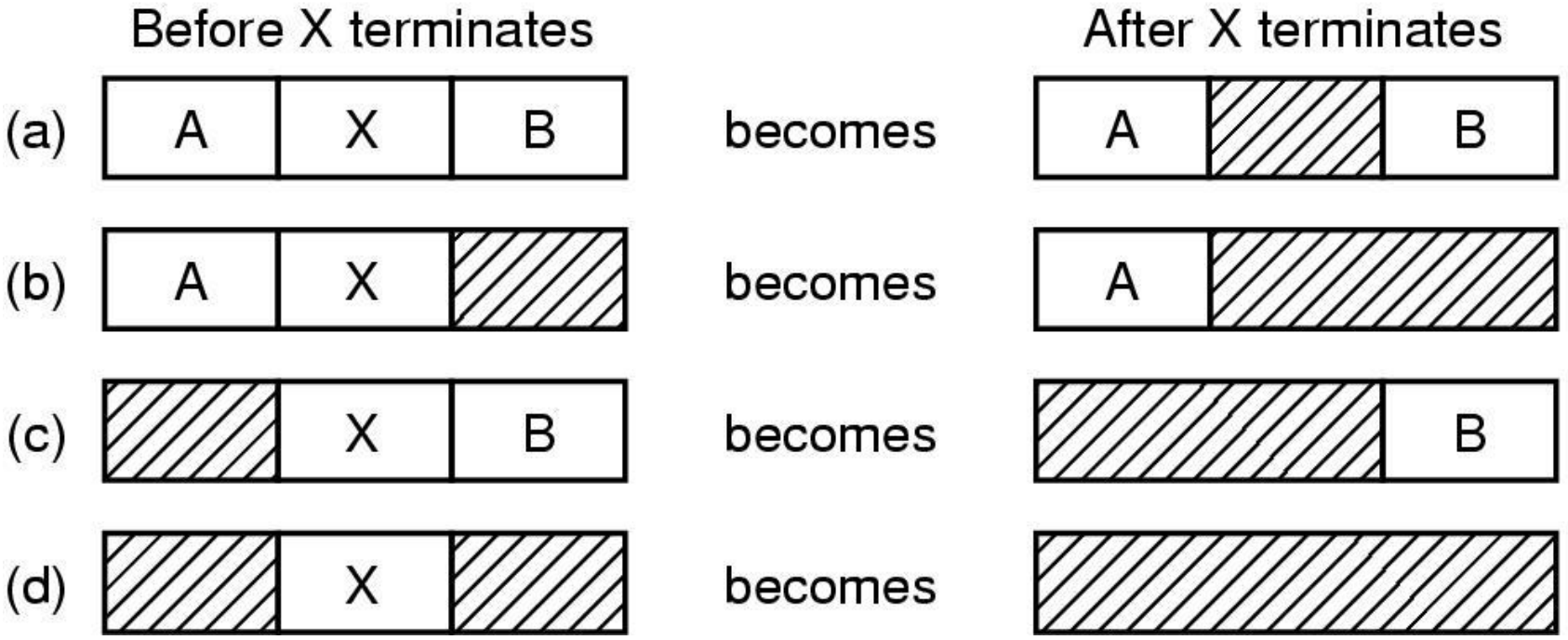


Memory Management with Bit Maps



- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list

Memory Management with Linked Lists



Four neighbor combinations for the terminating process X

Page Replacement Algorithms

When a page fault occurs, the operating system must choose a page to remove from memory to make room for the page that has to be brought in.

- On the second run of a program, if the operating system kept track of all page references, the "**Optimal Page Replacement Algorithm**" could be used:

replace the page that will not be used for the longest amount of time. This method is **impossible** on the first run and not used in practice. It is used in theory to evaluate other algorithms.

Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by …
  - logging page use on previous runs of process
  - although this is impractical

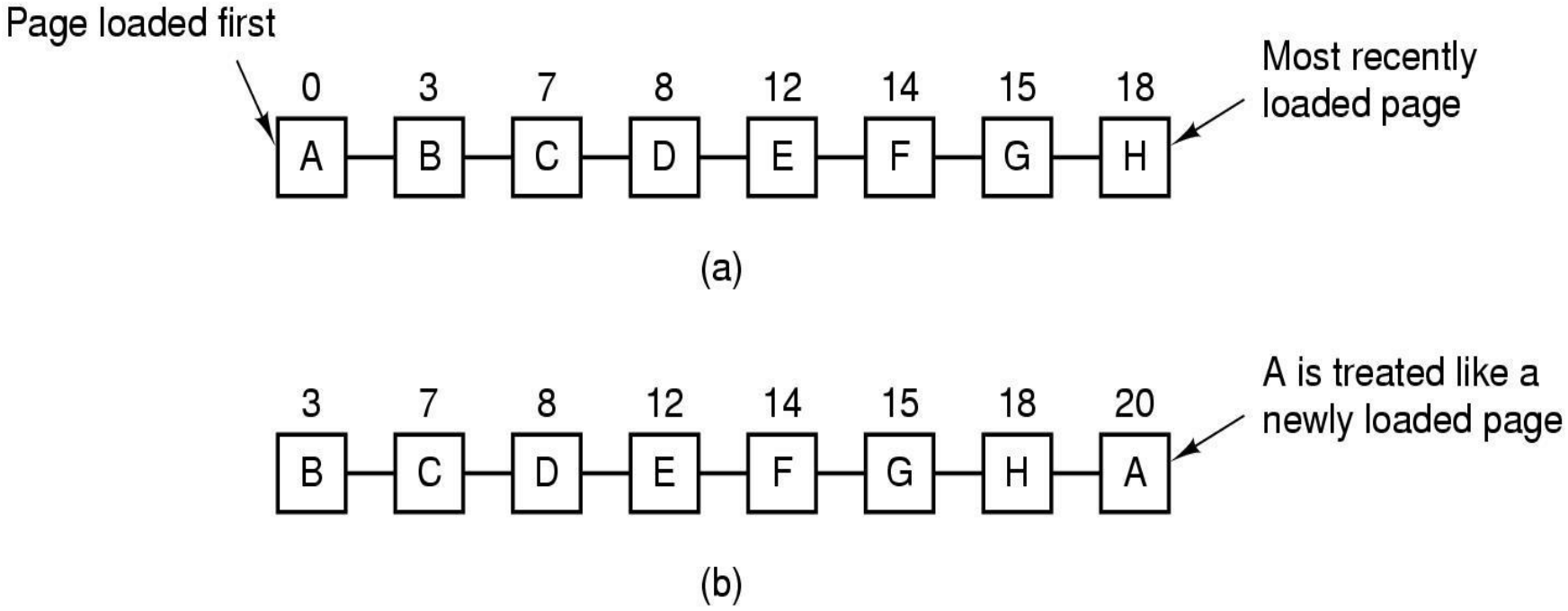
Not Recently Used Page Replacement Algorithm

- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
- Pages are classified
  - not referenced, not modified
  - not referenced, modified
  - referenced, not modified
  - referenced, modified
- NRU removes page at random
  - from lowest numbered non empty class

FIFO Page Replacement Algorithm

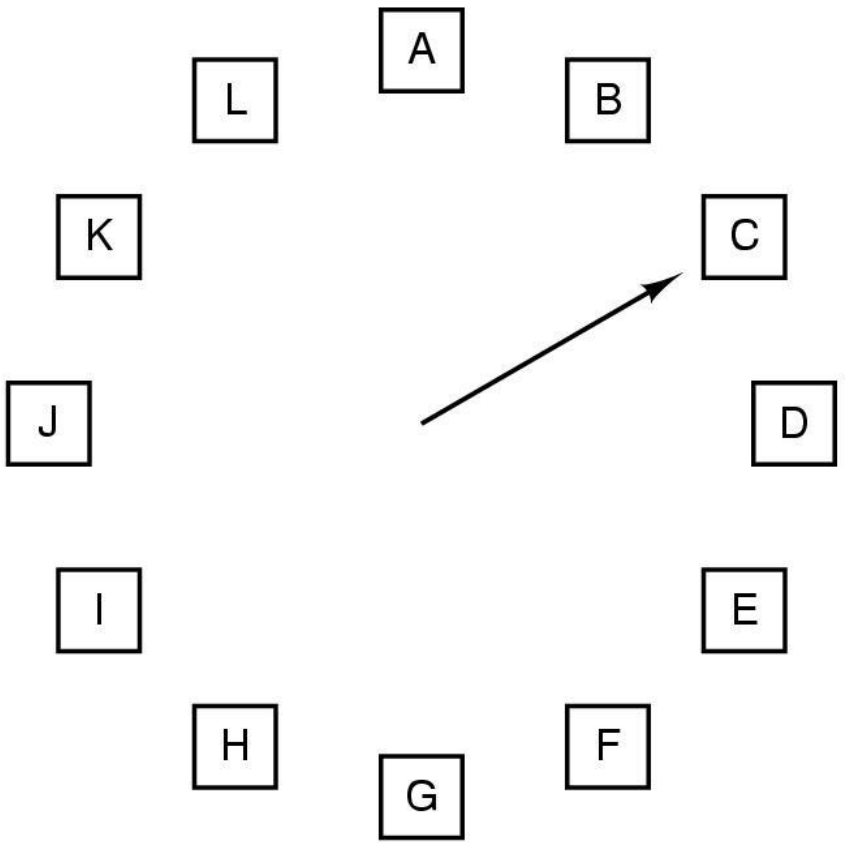
- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
  - page in memory the longest may be often used

Second Chance Page Replacement Algorithm



- Operation of a second chance
  - pages sorted in FIFO order
  - Page list if fault occurs at time 20, *A* has *R* bit set (numbers above pages are loading times)

The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

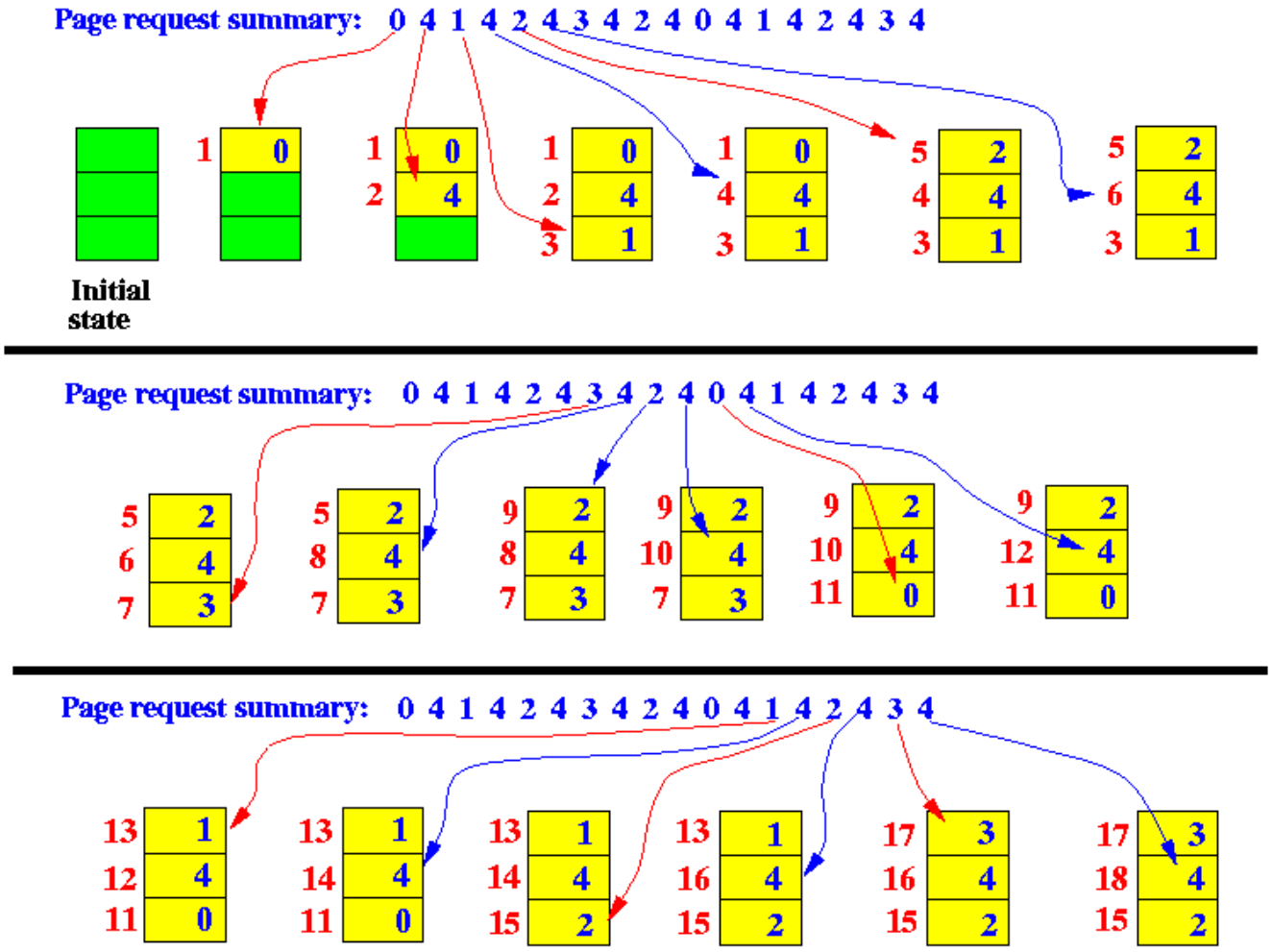
R = 0: Evict the page

R = 1: Clear R and advance hand

Least Recently Used (LRU)

- Assume pages used recently will used again soon
  - throw out page that has been unused for longest time
- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter

LRU example



- Add a register to every page frame - contain the last time that the page in that frame was accessed
- Use a "logical clock" that advance by 1 tick each time a memory reference is made.
- Each time a page is referenced, update its register

Least-Frequently-Used (LFU) Page Replacement

- LFU page replacement
  - Replaces page that is least intensively referenced
  - Based on the heuristic that a page not referenced often is not likely to be referenced in the future
  - Could easily select wrong page for replacement
    - A page that was referenced heavily in the past may never be referenced again, but will stay in memory while newer, active pages are replaced

The Working Set Page Replacement Algorithm(1)

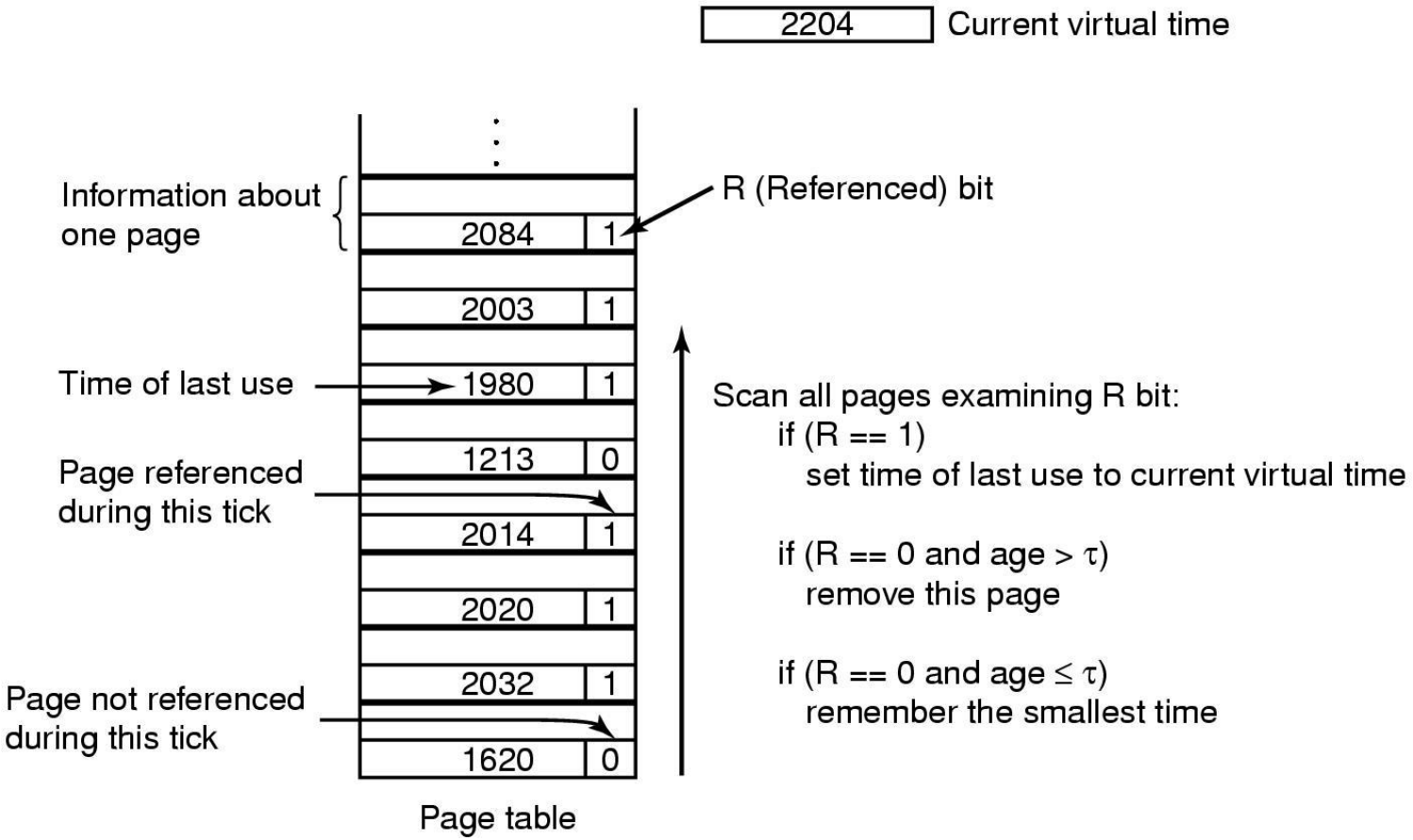
The 'working set' is represented by w(k,t). This is the set of pages, where 't' is any instant in time and 'k' is a number of recent memory references. The 'working set' set changes over time but slowly. When a process must be suspended (due to an I/O wait or lack of free frames), the w(k, t) can be saved with the process. In this way, when the process is reloaded, its entire w(k,t) is reloaded, avoiding the initial large number of page faults. This is called **'PrePaging'**.

The operating system keeps track of the working set, and when a page fault occurs, chooses a page not in the working set for replacement. This requires a lot of work on the part of the operating system. A variation, called the '**WSClock Algorithm'**, similar to the 'Clock Algorithm', makes it more efficient.

- Clock Page Replacement - YouTube

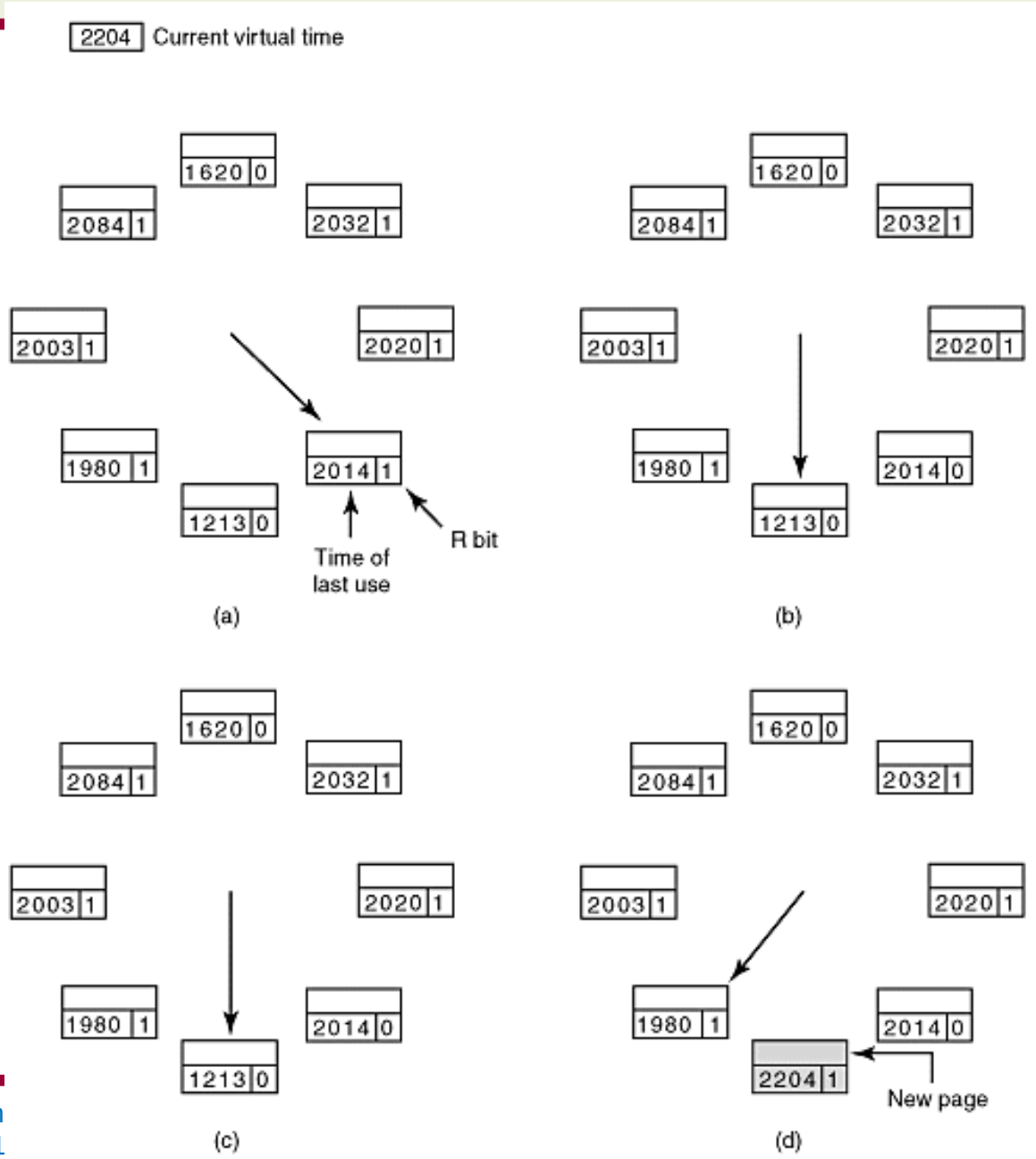
https://www.youtube.com/watch?v=b-dRK8B8dQk

The Working Set Page Replacement Algorithm(2)



The working set algorithm

The WSClock Page Replacement Algorithm



- CSE 312 Spring 2021 May 03 WSClock algo, Design issues with memory management - YouTube

https://www.youtube.com/watch?v=DAyoL-tNnwo

Review of Page Replacement Algorithms

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
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

References

- Ebook: "Linux Kernel Source Code – Heavily commented"
  - **Chapter 4. Protection mode and its programming**
  - **Chapter 5. Linux kernel architecture**
  - **Chapter 13. Memory management**