VIRTUAL MEMORY

Chapter 4.3

VIRTUAL MEMORY: TOPICS

- Paging
- Page Tables
- Example of paging hardware
- Associative Memory

VIRTUAL MEMORY: INTRODUCTION

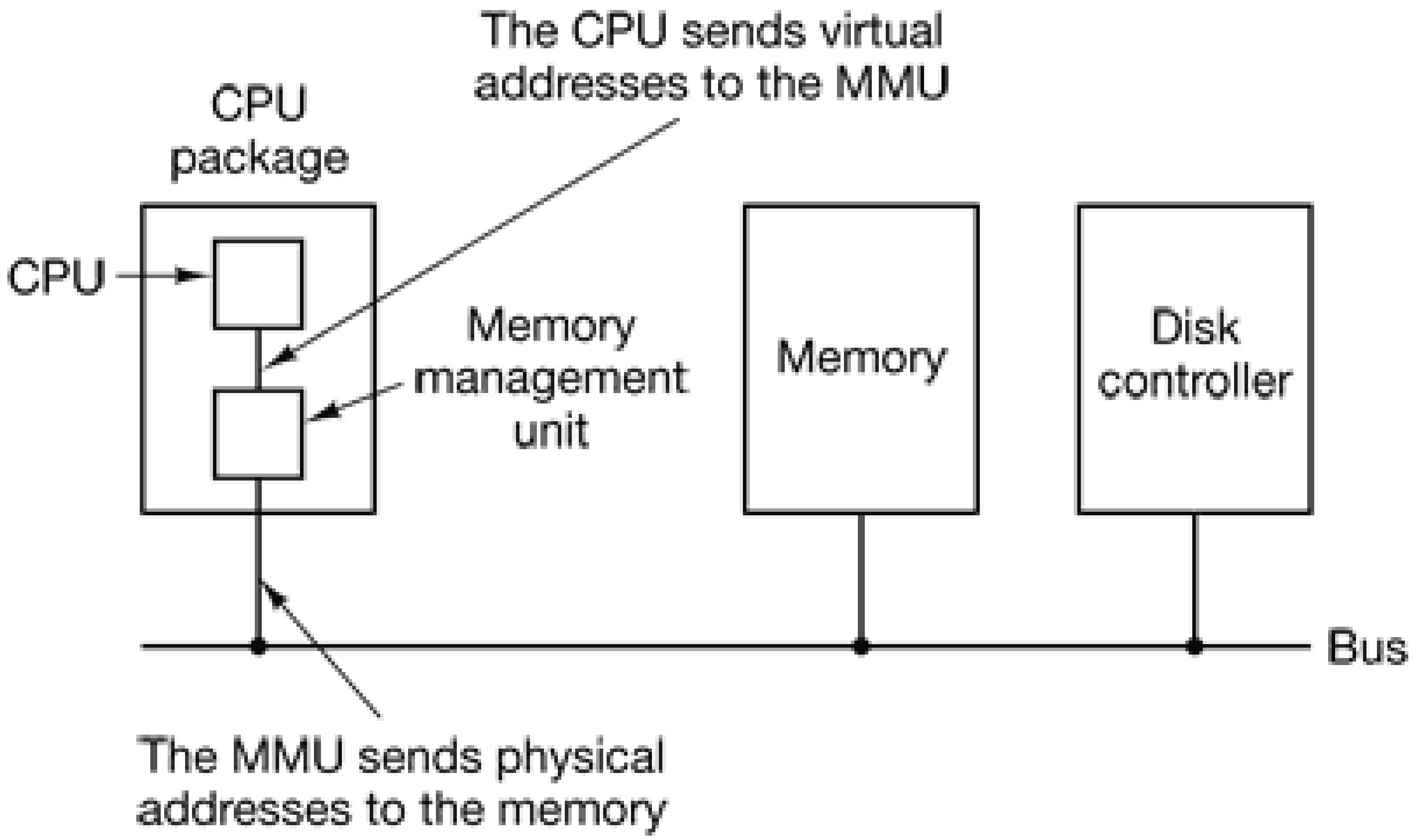
- Combined size of the program, data, and stack may exceed the amount of physical memory available for it.
- The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk.
- For example, a 16-MB program can run on a 4-MB machine by carefully choosing which 4 MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.
- Virtual memory can be implemented by two most commonly used methods :
  - *Paging* and
    Segmentation or mix of both.

VIRTUAL MEMORY

- **Virtual address space vs. Physical address space**
- The set of all virtual (logical) addresses generated by a program is a ***virtual address space***;
- the set of all physical addresses corresponding to these virtual addresses is a ***physical address space***.

- MMU
  The **run time mapping** from virtual address to physical address is done by hardware devices called *memory-management-unit (MMU)*.

MMU POSITION & FUNCTION


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

PAGING

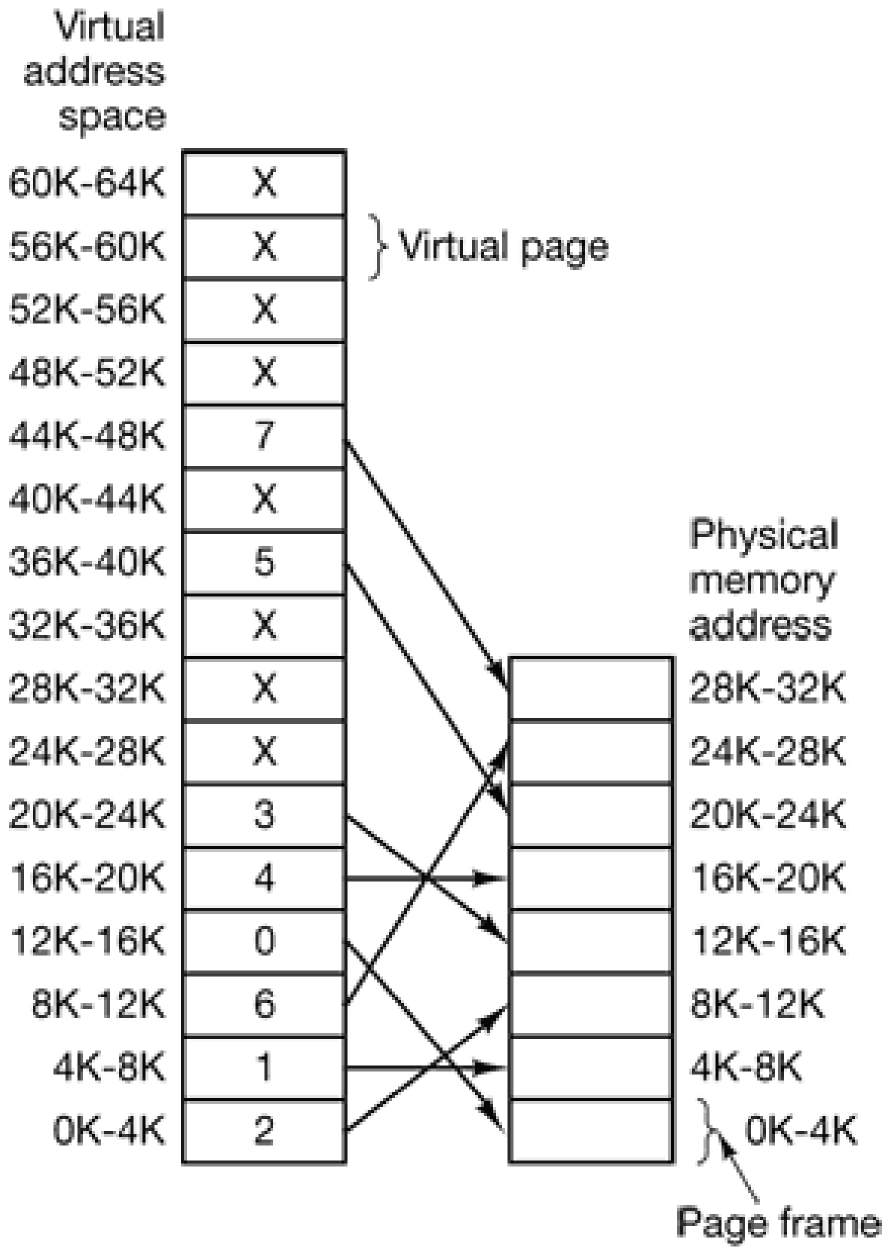
- The virtual address space is divided up into units called **pages**.
- The corresponding units in the physical memory are called **page frames**.
- The pages and page frames are always the same size.
- Page sizes from **512 bytes to 64 KB(in power of 2)** have been used in real systems.
- If 4KB page size used, With 64 KB of virtual address space and 32 KB of physical memory, we get 16 virtual pages and 8 page frames.
- Transfers between RAM and disk are always **in units of a page.**

PAGE MAPPING

- ❖ Computer that can generate 16-bit addresses, from 0 up to 64K
- ❖ has only 32 KB of physical memory
- ❖ MOV REG,0 effectively transformed to page frame starting from 8192.
- ❖ MOV REG,8192 is effectively transformed into MOV REG,24576

What happens if the program tries to use an unmapped page, for example, by using the instruction
MOV REG,32780

Page Fault



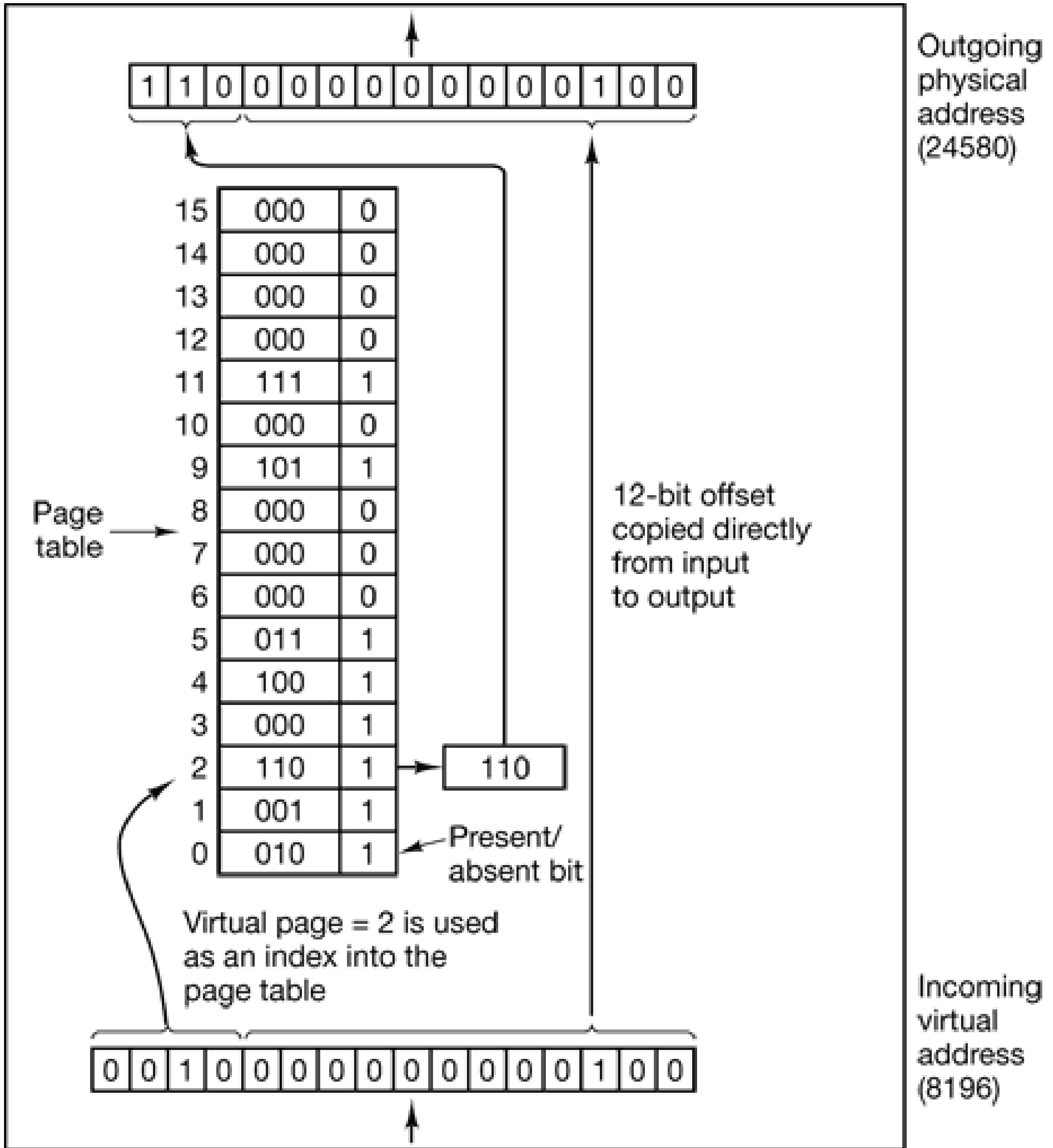
PAGE FAULT

- If the MMU notices that the page is unmapped (indicated by a cross in the figure)
- causes the CPU to trap to the operating system.
- This trap is called a **page fault**.
- The operating system picks a little-used page frame and writes its contents back to the disk.
- It then fetches the page just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

PAGE TABLES

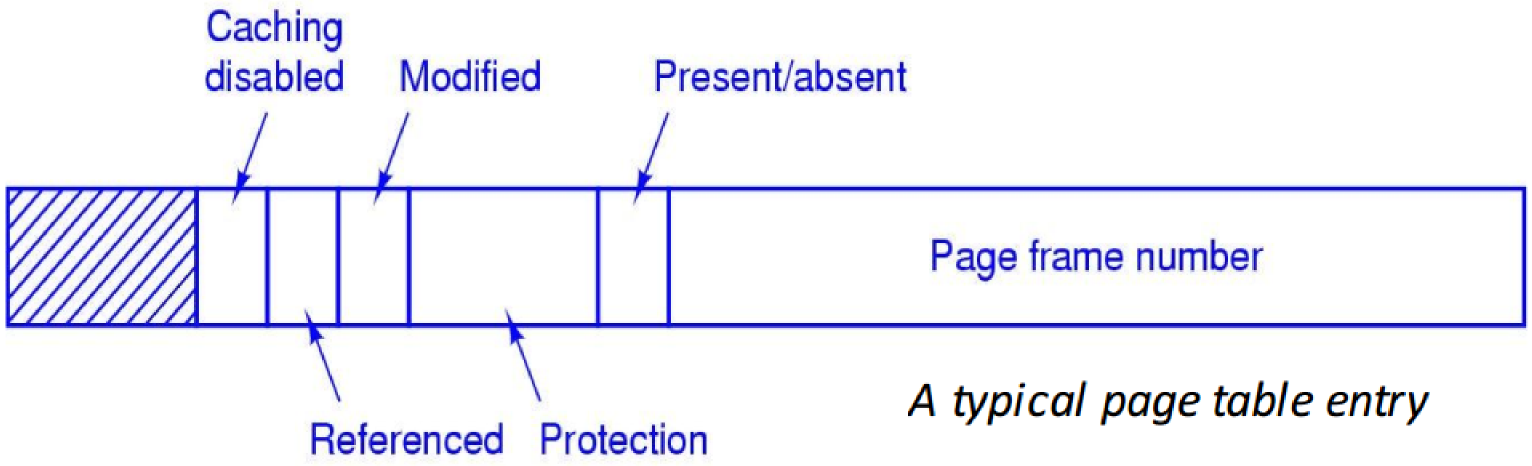
- the mapping of virtual addresses onto physical addresses
- The virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits).
- For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page.

ADDRESS TRANSLATION EXAMPLE



PAGE TABLE STRUCTURE

- The exact layout of page table entry is highly machine dependent, but more common structure for 32-bit system is as



A typical page table entry

PAGE TABLE STRUCTURE

- **Frame number:** The goal is to locate this value.
- **Present/absent bit:** If present/absent bit is present, the virtual addresses is mapped to the corresponding physical address. If present/absent is absent the trap is occur called page fault.
- **Protection bit:** Tells what kinds of access are permitted read, write (0) or read only (1).
- **Modified bit (dirty bit):** Identifies the changed status of the page since last access; if it is modified then it must be rewritten back to the disk.
- **Referenced bit:** set whenever a page is referenced; used in page replacement.
- **Caching disabled:** used for that system where the mapping into device register rather than memory for IO.

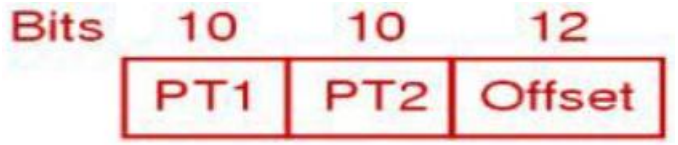
PAGE TABLE ISSUES

- Despite this simple description, two major issues must be faced:
- **The page table can be extremely large.**
  - If computers use virtual addresses of at least 32 bits.
  - With, say, a 4-KB page size, it will have 1 million pages,
  - a 64-bit address space has more than you want to think.
  - With 1 million pages in the virtual address space, the page table must have 1 million entries.
  - And remember that each process needs its own page table (because it has its own virtual address space).
- **The mapping must be fast.**
  - mapping must be done on every memory reference.
  - A typical instruction has an instruction word, and often a memory operand as well.
  - Consequently, it is necessary to make 1, 2, or sometimes more page table references per instruction
  - If an instruction takes, say, 4 nsec, the page table lookup must be done in under 1 nsec to avoid becoming a major bottleneck.

  - What would be the performance, if such a large table have to load at every mapping.?

MULTILEVEL PAGE TABLES

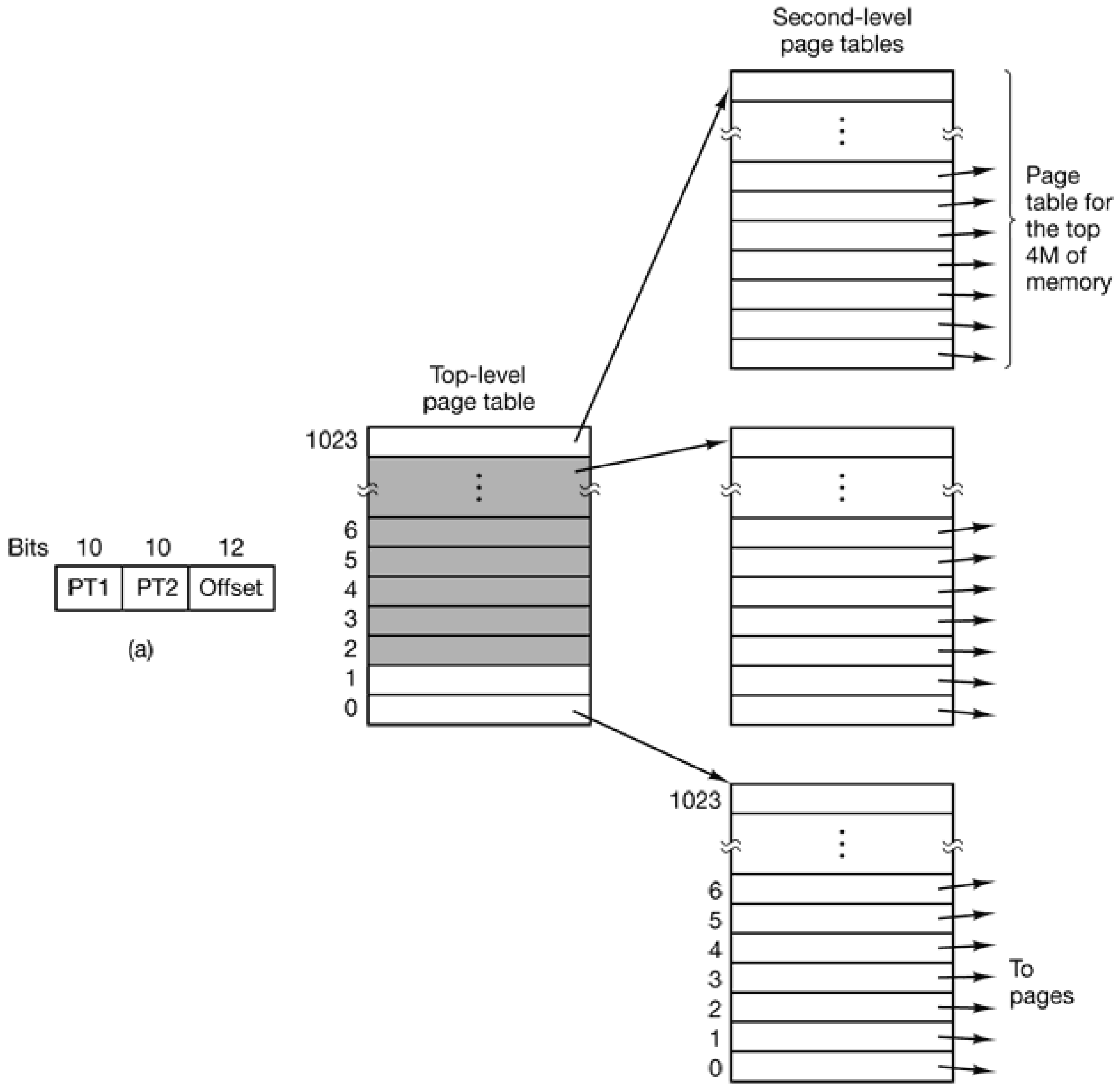
- To get around the problem of having to store huge page tables in memory all the time.
- a 32-bit virtual address that is partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field.



- The top level have 1024 entries, corresponding to PT1. At mapping, it first extracts the PT1 and uses this value as an index into the top level page table. Each of these entries have again 1024 entries , the resulting address of top-level yields the address or page frame number of second-level page table.

MULTILEVEL PAGE TABLES

- a



WHY HARDWARE SOLUTION?

- Large page tables are kept in memory
  - Two references on memory per instruction
    - Firstly for page table
    - Then for instruction address
  - enormous impact on performance.
- solution is based on the observation that
  - most programs tend to make a **large number of references to a small number of pages**, and not the other way around.
  - Thus only a **small fraction of the page table entries are heavily read**; the rest are barely used at all.
  - **TLBs—Translation Lookaside Buffers**

TLBS—TRANSLATION LOOKASIDE BUFFERS

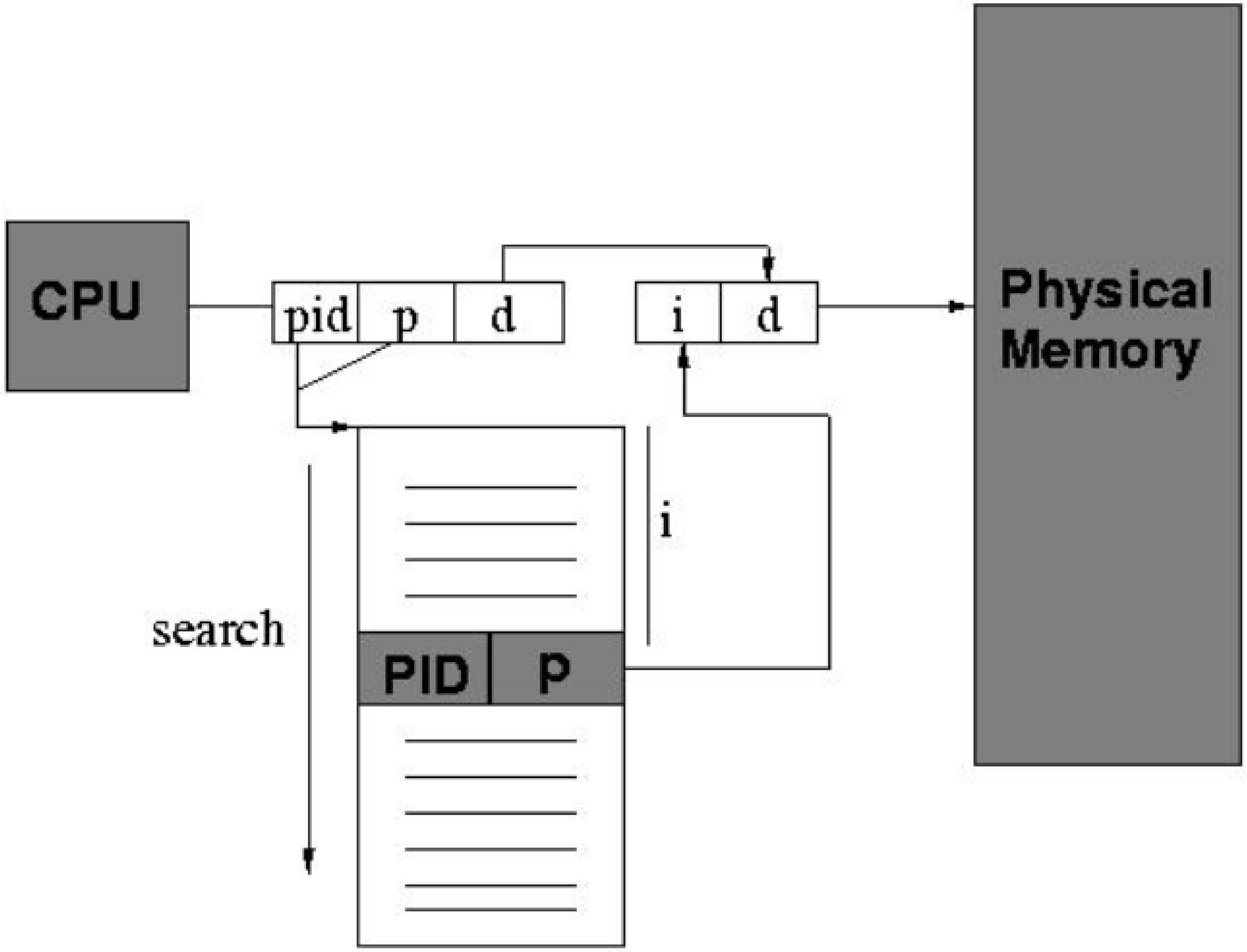
- TLB/Associative memory/address-translation cache
- Equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table
- TLB Hit/TLB Miss
- If the valid bit is set, the page is in physical memory else page is on disk.

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

INVERTED PAGE TABLES

- *A common approach for handling address space larger than 32-bit.*
- One entry per page frame, rather than one entry per page in earlier tables.
- Ex: 256MB RAM with 4KB page requires only 65,536 entries in table
- *Virtual address consists three fields:*
  - *[process-id, page-number, offset].*
- *The inverted page table entry is determined by*
  - *[process-id, page-number].*
- *The page table is search for the match, say at entry i the match is found, then the physical address [i, offset] is generated.*

INVERTED PAGE TABLES



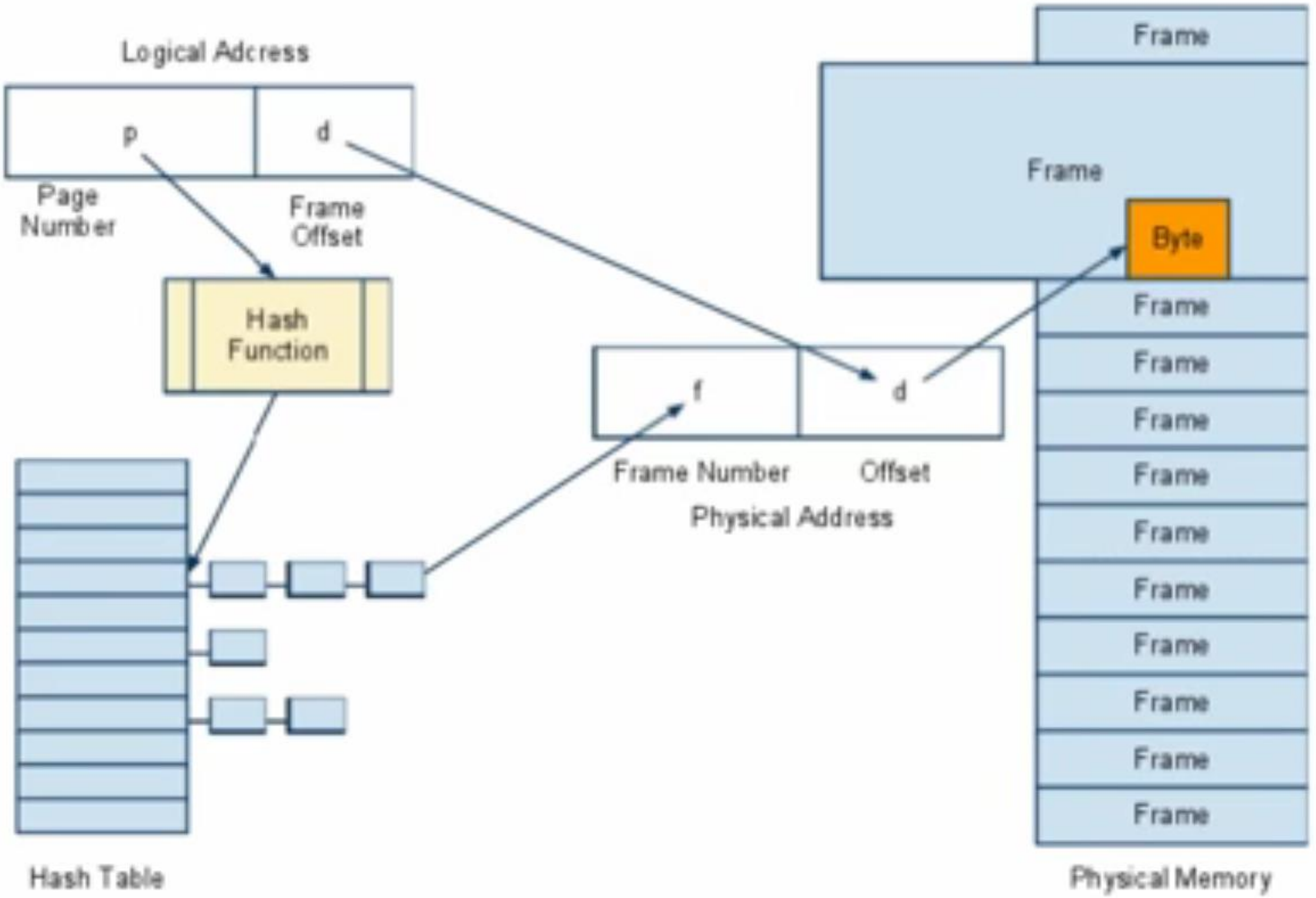
INVERTED PAGE TABLES

- When process *pid* references virtual page *p*, the hardware can no longer find the physical page by using *p* as an index into the page table.
- Instead, it must search the entire inverted page table for an entry (*pid*, *p*).
- Search must be done on every memory reference
- Searching a 64K table on every memory reference is not the way to make your machine blindingly fast.
- The way out of this dilemma is to use the **TLB**.
  - If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables.
  - On a TLB miss, however, the inverted page table has to be searched in software.
  - One feasible way to accomplish this search is to have a **Hash Table** hashed on the virtual address

HASH TABLE

- *A common approach for handling address space larger than 32-bit.*
- The hash value is the virtual-page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location.
- Each element consists of three fields: virtual-page-number, value of mapped page frame, and a pointer to the next element.
- *The virtual address is hashed into the hash table, if there is match the corresponding page frame is used, if not, subsequent entries in the linked list are searched.*

Hashed Address Translation



GIVEN 16 BIT ADDRESS SPACE, PAGE SIZE 4KB, IF YOU USE 4 BYTES PER PAGE TABLE ENTRY, FIND TABLE SIZE IN SINGLE TABLE ENTRY MODE AND MULTI LEVEL ENTRY MODE WITH 2 LEVELS.

GIVEN 16 BIT ADDRESS SPACE, PAGE SIZE 4KB, IF YOU USE 4 BYTES PER PAGE TABLE ENTRY, FIND TABLE SIZE IN SINGLE TABLE ENTRY MODE AND MULTI LEVEL ENTRY MODE WITH 2 LEVELS.

- Page size 4kb = page frame size = 4096 bytes
- Ie 2^{12} = 12 bits requi red for offset.
- Remaining bits = 4 = 2^4
- Total no of pages = 2^4 =16
- Total size of table = 16 * 4 bytes = 64bytes

with 2 level page table:

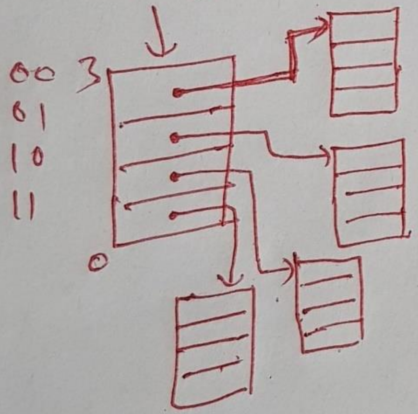
page size 2^{12}

remaining bits $2^{16} - 2^{12} = 2^4$

1st level table 2^2

2nd level table 2^2

PT 1	PT 2	offset
2 bits	2 bits	12 bits



Total # of pages in each table = 4

Size of 1 table = 4×4 = 16 Kb

IF AN INSTRUCTION TAKES 10NSEC AND A PAGE FAULT TAKES AN ADDITIONAL N SEC, GIVE FORMULA FOR THE EFFECTIVE INSTRUCTION TIME IF PAGE FAULTS OCCUR EVERY K INSTRUCTIONS.

- Considering k instructions
  - K-1 instructions execute without page fault
    - 10*(k-1) nsec total time for k-1 instructions
  - 1 instruction with page fault
    - (10+n) * 1 nsec for remaining 1 instruction
- Average
  - ((10*(k-1) + (10+n) * 1)/k
  - (10k -10 +10 +n)/k
  - (10k+n)/k
  - 10+n/k nsec

EFFECTIVE ACCESS TIME

- Memory Access Time:
  - the time from the start of one storage device access to the time when the next access can be started.
  - Access time consists of latency (the overhead of getting to the right place on the device and preparing to access it) and transfer time.
- Effective Access Time:
  - Total Access time to access memory with TLB hit and miss percentage.
  - EAT = h * (TLB Lookup time) + (1- h) * (TLB lookup time + memory access time)
    - Where h is hit ratio

HIT RATIO

- Probability to find desired page number in TLB
- The percentage of times that a particular page number is found in TLB is called Hit Ratio
- 80 percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.
- If it takes 20 ns to search TLB and 100ns to access memory, then mapped memory access takes 120 ns when page number is in TLB.
- If we fail to find the page number in TLB(20ns) then we must first access memory for the page table and frame number(100ns) then access the desired byte from memory(100ns) for total of 220ns.
- To find effective memory access time, we weight each case by its probablity
  - EAT = 0.8*120 + 0.20 * 220 = 140ns.
  - In this example we suffer 40% slow down in memory access time (from 100 to 140ns)
  - (140-100)/100   * 100%   = 40%

HOW SLOW IS YOUR SYSTEM WHEN HIT RATIO IS 98%.

- TLB look up time 20ns
- Memory lookup time 100ns

- A computer whose process have 1024 pages in their address space keeps its page tables in memory. The overhead required for reading a word from the page table is 500ns. To reduce this overhead, the computer has an associative memory which hold 32(virtual page, physical page frame) pairs and can do look up in 100ns. What hit rate is needed to reduce the mean overhead to 200ns.

  - EAT = 100 h + 500 (1- h)
  - 200 = 100 h + 500 (1- h)

PAGE REPLACEMENT ALGORITHMS

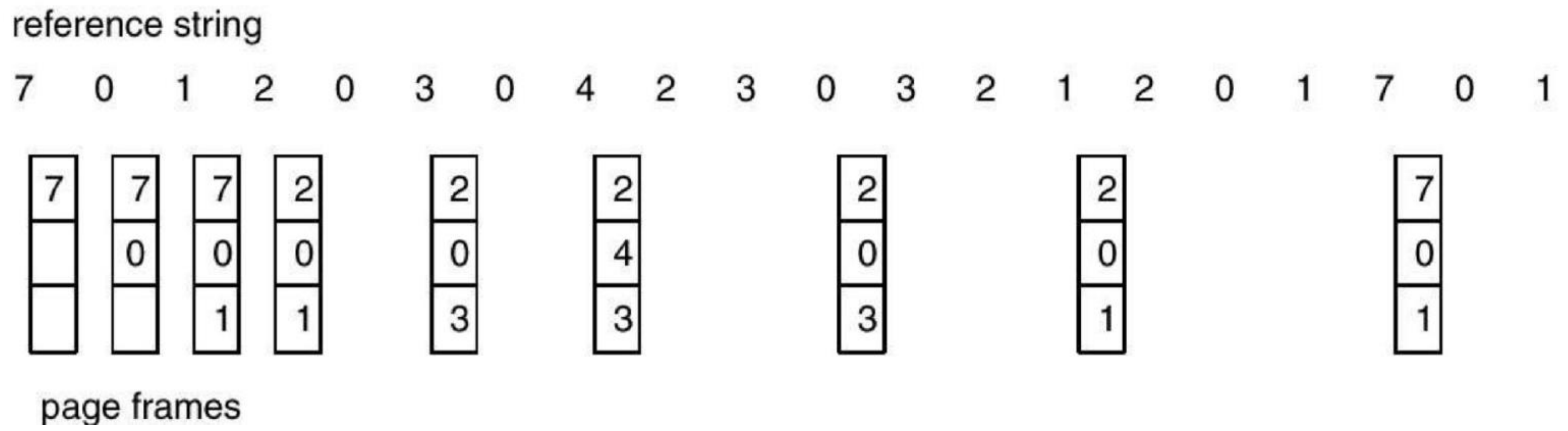
- When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in.
- Which one page to be removed ?
- What happen if the page that required next, is removed?
- For better **system performance**: pick a random page ? not heavily used ? Page just used
  - **The Optimal Page Replacement Algorithm**
  - **The Not Recently Used Page Replacement Algorithm**
  - **The First-In, First-Out (FIFO) Page Replacement Algorithm**
  - **The Second Chance Page Replacement Algorithm**
  - **The Clock Page Replacement Algorithm**
  - **The Least Recently Used (LRU) Page Replacement Algorithm**

THE OPTIMAL PAGE REPLACEMENT ALGORITHM

- Replace the page that will not be used for longest period of time.
- pages may not be referenced until 10, 100, or perhaps 1000 instructions later
- page with the **highest label should be removed**.
- Advantage:
  - Optimal page replacement algorithm
  - It guarantees the lowest possible page fault rate
- Problem:
  - **Unrealizable**
    - The best possible page replacement algorithm is easy to describe but impossible to implement

THE OPTIMAL PAGE REPLACEMENT ALGORITHM

Ex: For 3 - page frames and 8 pages system the optimal page replacement is as:

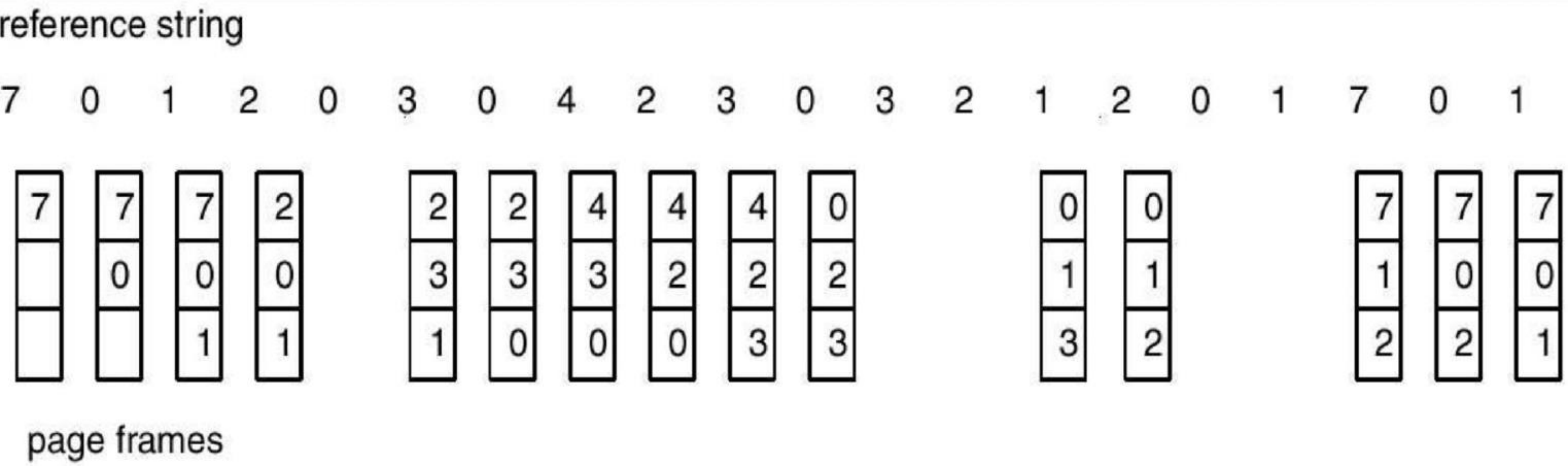


NOT RECENTLY USED PAGE REPLACEMENT ALGORITHM

- *Pages not recently used are* ***not likely to be used in near future*** *and they must be replaced with incoming pages*.
- To keep statistics about which pages are being used
  - **have two status bits** associated with each page
  - – referenced **R** and modified **M**
  - **R:** set whenever the page is referenced
  - **M**: set when the page is written
- *Class 0: not referenced, not modified.*
  Class 1: not referenced, modified.
  (*R* bit cleared by a clock interrupt)
  Class 2: referenced, not modified.
  Class 3: referenced, modified
- Pages in the lowest numbered class should be replaced first.
- Pages within the same class are randomly selected.

THE FIRST-IN, FIRST-OUT (FIFO) PAGE REPLACEMENT ALGORITHM

- The OS **maintains a list of all pages** currently in memory, with the page at the **head of the list the oldest** one and the page at the **tail the most recent arrival.**



7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

BÉLÁDY'S ANOMALY

- **Bélády's anomaly** is the phenomenon in which increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns. This phenomenon is commonly experienced when using the first-in first-out (FIFO) page replacement algorithm.

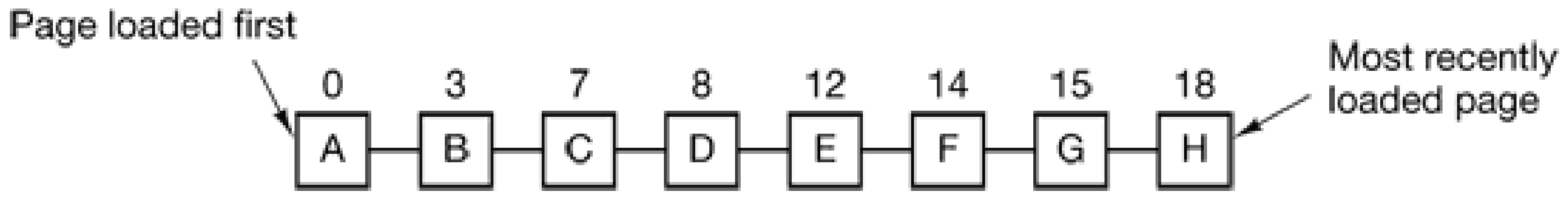
Page requests 6(3 frames)	3	2	1	0	3	2	4	3	2	1	0	4
Newest page Total:9 Page faults	3	2	1	0	3	2	4	3	2	1	0	0

Page requests (4 frames)	3	2	1	0	3	2	4	3	2	1	0	4
Newest page Total:10 Page faults	3	2	1	0	0	0	4	3	2	1	0	4

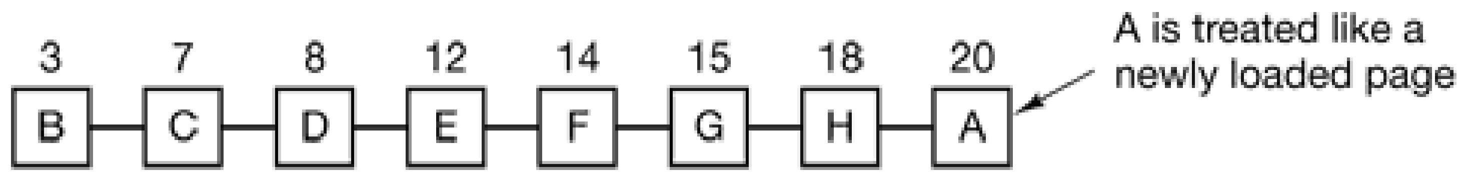
THE SECOND CHANCE PAGE REPLACEMENT ALGORITHM

- *to avoid the replacing of heavily used pages* in FIFO
- inspect the *R* bit of the oldest page
  - If the R bit is 0: the page is both old and unused, replaced immediately
  - If it is 1 clear the bit and look for another page with R bit set to 0.
  - its arrival time is reset to the current time i.e. move the page to end of the list.
- **Advantages:**
  Big improvement over FIFO.
- **Problems:**
  If all the pages have been referenced, second chance degenerates into pure FIFO.

OPERATION OF SECOND CHANCE



(a)



(b)

- (a) Pages sorted in FIFO order.
- (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their loading times.

GIVEN REFERENCES TO THE FOLLOWING PAGES BY A PROGRAM, 0,9,0,1,8,1,8,7,8,7,1,2,8,2,7,8,2,3,8,3. HOW MANY PAGE FAULTS WILL OCCUR IF THE PROGRAM HAS THREE PAGE FRAMES FOR EACH OF THE FOLLOWING ALGORITHMS?

- 1. FIFO: (Only showing page fault cases)

0	8	8	8	3	3							
9	9	7	7	7	8							
1	1	1	2	2	2							

Page faults: 3 1 1 1 1 1 = Total Page Faults 8

- 2. Optimal
- 3. Second Chance LRU
- 4. LRU

GIVEN REFERENCES TO THE FOLLOWING PAGES BY A PROGRAM, 0,9,0,1,8,1,8,7,8,7,1,2,8,2,7,8,2,3,8,3. HOW MANY PAGE FAULTS WILL OCCUR IF THE PROGRAM HAS THREE PAGE FRAMES FOR EACH OF THE FOLLOWING ALGORITHMS?

- 2. Optimal: (Only showing page fault cases)

0	8	8	8	8								
9	9	7	7	3								
1	1	1	2	2								

Page faults: 3 1 1 1 1 = Total Page Faults 7

- 3. Second Chance LRU
- 4. LRU

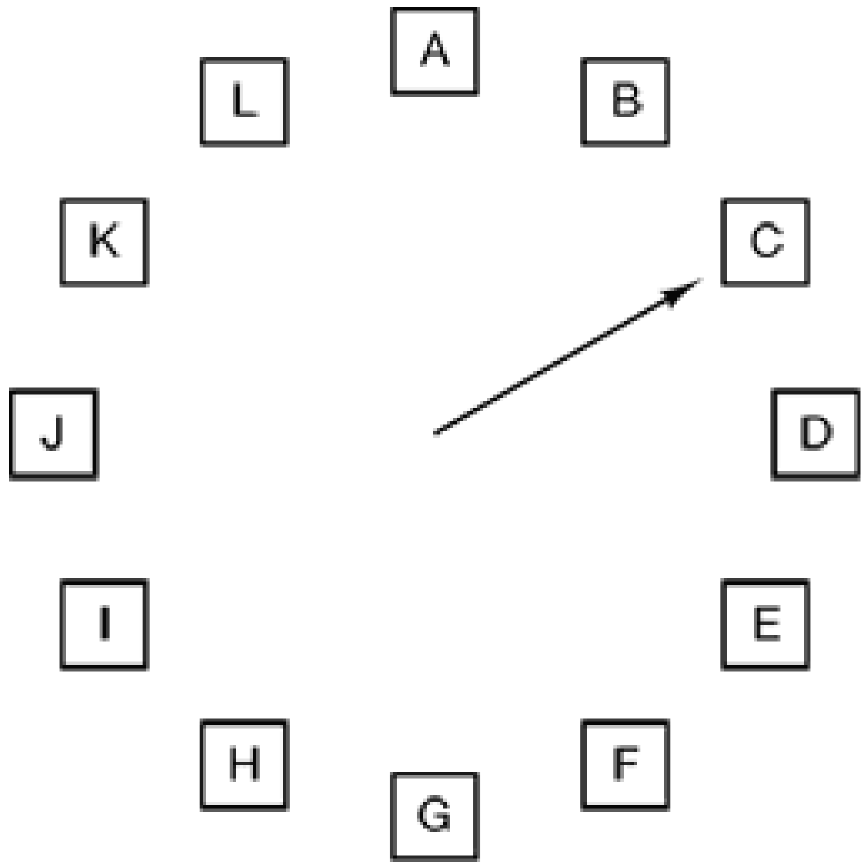
GIVEN REFERENCES TO THE FOLLOWING PAGES BY A PROGRAM, 0,9,0,1,8,1,8,7,8,7,1,2,8,2,7,8,2,3,8,3. HOW MANY PAGE FAULTS WILL OCCUR IF THE PROGRAM HAS THREE PAGE FRAMES FOR EACH OF THE FOLLOWING ALGORITHMS?

- Second Chance:
- Initially all frames were empty so we fill it with three pages [0, 9, 1] and page 0 accessed twice so it will get second chance and array will be {1,0,0} pointer index=0, pf=3
- Pass 5: [0, 8, 1] page 8, since index 0 has reference bit 1 we give it second chance, reference bit will be cleared. Pointer moves to index 1. Page at index 1 is 9 and its reference bit is 0. so it will be replaced and Pointer will be at index 2. {0 0 0} pf=4
- Pass 6: [0,8,1] {0 0 1} Page 1 is already in frame so only reference bit will be set. Pointer unchanged=2 pf=4
- Pass 7: [0,8,1] {0 1 1} Page 8,pointer=2 pf=4
- Pass 8: [7,8,1] {0 1 0} page 7, index 2 second chance, page at index 0 replaced, pointer=1 pf=5
- Pass 9: [7,8,1] {0 1 0} page 8, pointer=1, pf=5
- Pass 10: [7,8,1] {1 1 0} page 7, pointer=1, pf=5
- Pass 11: [7,8,1] {1 11} page 1, pointer=1, pf=5
- Pass 12: [7,2,1] {0 0 0} Page 2,pointer=2, pf=6
- Pass 13: [7,2,8] {0 0 0} page 8,pointer=0, pf=7
- Pass 14] : [7,2,8] {0 1 0} page 2,pointer=0 ,pf=7
- Pass 15: [7,2,8] {1 1 0} page 7, pointer=0, pf=7
- Pass 16: [7,2,8] {111} page 8, pointer=0, pf=7
- Pass 17: [7,2,8] {111} page 2, pointer=0, pf=7
- Pass 18: [3,2,8] {0 0 0} page 3, pointer=1, pf=8
- Pass 19: [3,2,8] {0 0 1} page 8 pointer=1, pf=8
- Pass 20: [3,2,8] {1 0 1} page 3 pointer=0 ,pf=8

THE CLOCK PAGE REPLACEMENT ALGORITHM

- Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is constantly moving pages around on its list.
- A better approach is to keep all the page frames on a circular list in the form of a clock. A hand points to the oldest page.
- Differ from second chance only in implementation.
- **Advantages:** More efficient than second chance.

CLOCK PAGE REPLACEMENT ALGORITHM



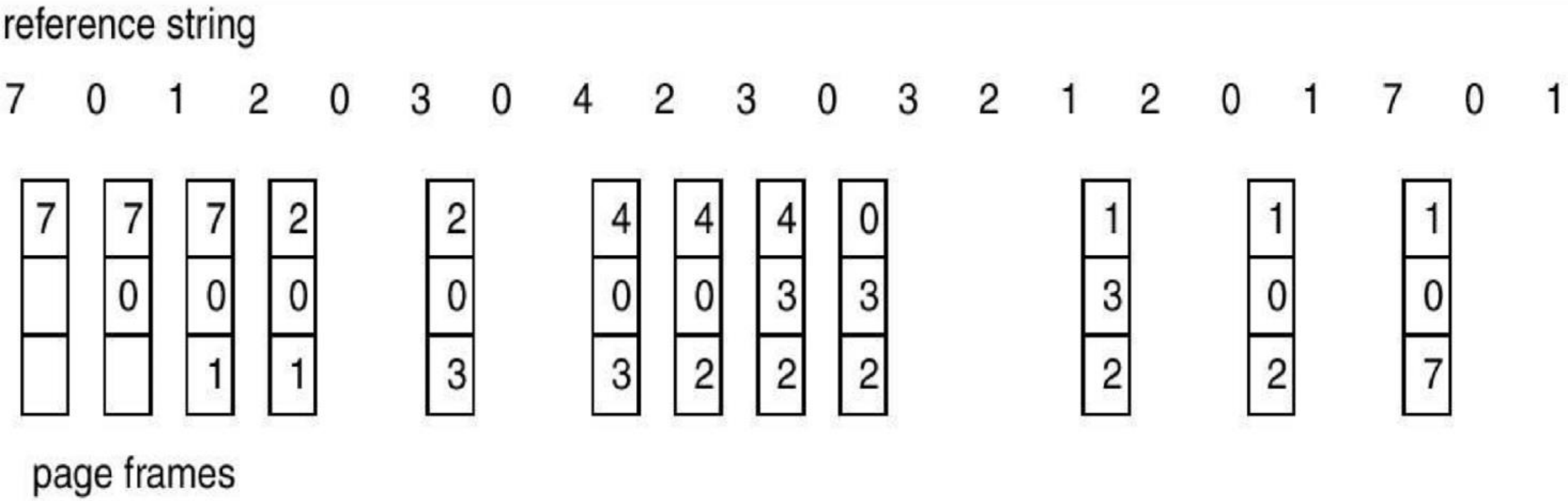
When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

THE LEAST RECENTLY USED (LRU) PAGE REPLACEMENT ALGORITHM

- Throw out the page that has been unused for longest time
- It maintains a linked list of all pages in memory with the most recently used page at the front and least recently used page at the rear.
- The list must be updated on every memory reference.



THE LEAST RECENTLY USED (LRU) PAGE REPLACEMENT ALGORITHM

- Difficulty
  - the list must be updated on every memory reference.
  - Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built).

THE LEAST RECENTLY USED (LRU) PAGE REPLACEMENT ALGORITHM

- ways to implement LRU with special hardware
- **Simple Approach:**
  - requires equipping the hardware with a **64-bit counter**, *C*, that is automatically incremented after each instruction
  - the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.
- **Second hardware LRU algorithm:**
- For a machine with *n* page frames, the LRU hardware can maintain a matrix of $n \times n$ bits, initially all zero.
- Whenever page frame *k* is referenced, the hardware first sets all the bits of row *k* to 1, then sets all the bits of column *k* to 0. At any instant, the row whose binary value is lowest is the least recently used, the row whose value is next lowest is next least recently used, and so forth.

- The workings of this algorithm are given for four page frames and page references in the order
- 0 1 2 3 2 1 0 3 2 3
- After page 0 is referenced, we have the situation of Fig. (a). After page 1 is reference, we have the situation of Fig.(b), and so forth.

$$\begin{array}{c|cccc} & \multicolumn{4}{c}{\text{Page}} \\ & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

(a)

$$\begin{array}{cccc} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}$$

(b)

$$\begin{array}{cccc} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}$$

(c)

$$\begin{array}{cccc} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{array}$$

(d)

$$\begin{array}{cccc} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{array}$$

(e)

$$\begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \\ \hline \end{array}$$

(f)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(g)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(h)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 \\ \hline \end{array}$$

(i)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(j)

DEMAND PAGING

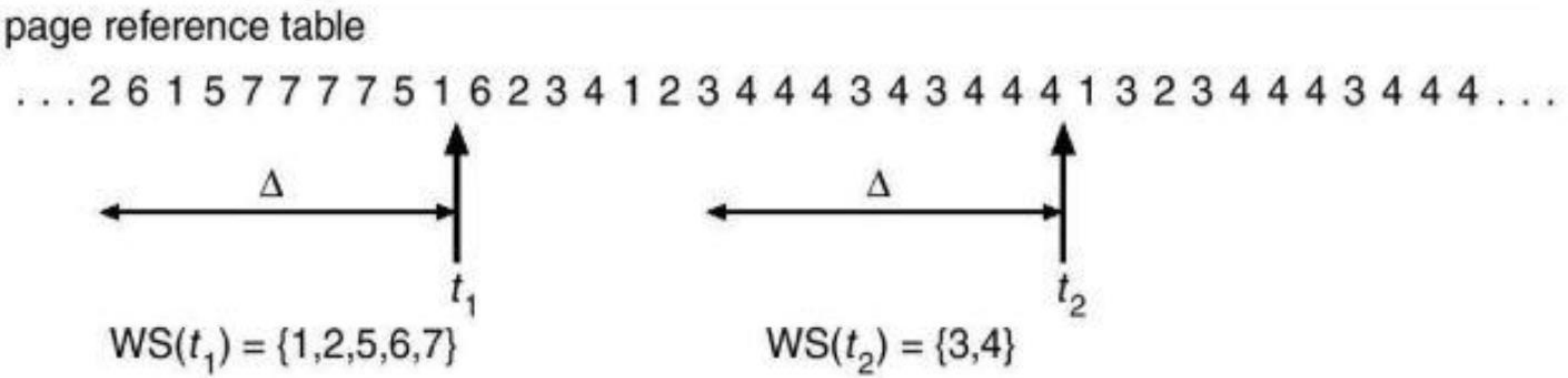
- In the purest form of paging, **processes are started up with none of their pages in memory**.
- As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruction.
- Other page faults for global variables and the stack usually follow quickly.
- After a while, the process has most of the pages it needs and settles down to run with relatively few page faults.
- This strategy is called **demand paging** because pages are loaded only on demand, not in advance.

THE WORKING SET PAGE REPLACEMENT ALGORITHM

- *In multiprogramming, processes are frequently move to disk to let other process have a turn at the CPU.*
- *The set of pages that a process is currently using is called its* ***working set.***
  - the set of pages used in the *k* most recent memory references.
- If the entire working set is in memory, the process will run without causing many faults until it moves into another execution.
  - Otherwise, excessive page fault might occur called ***thrashing***.
- Many paging systems try to keep track of each process' working set and make sure that it in memory before the process run - -
  working set model or prepaging

THE WORKING SET PAGE REPLACEMENT ALGORITHM

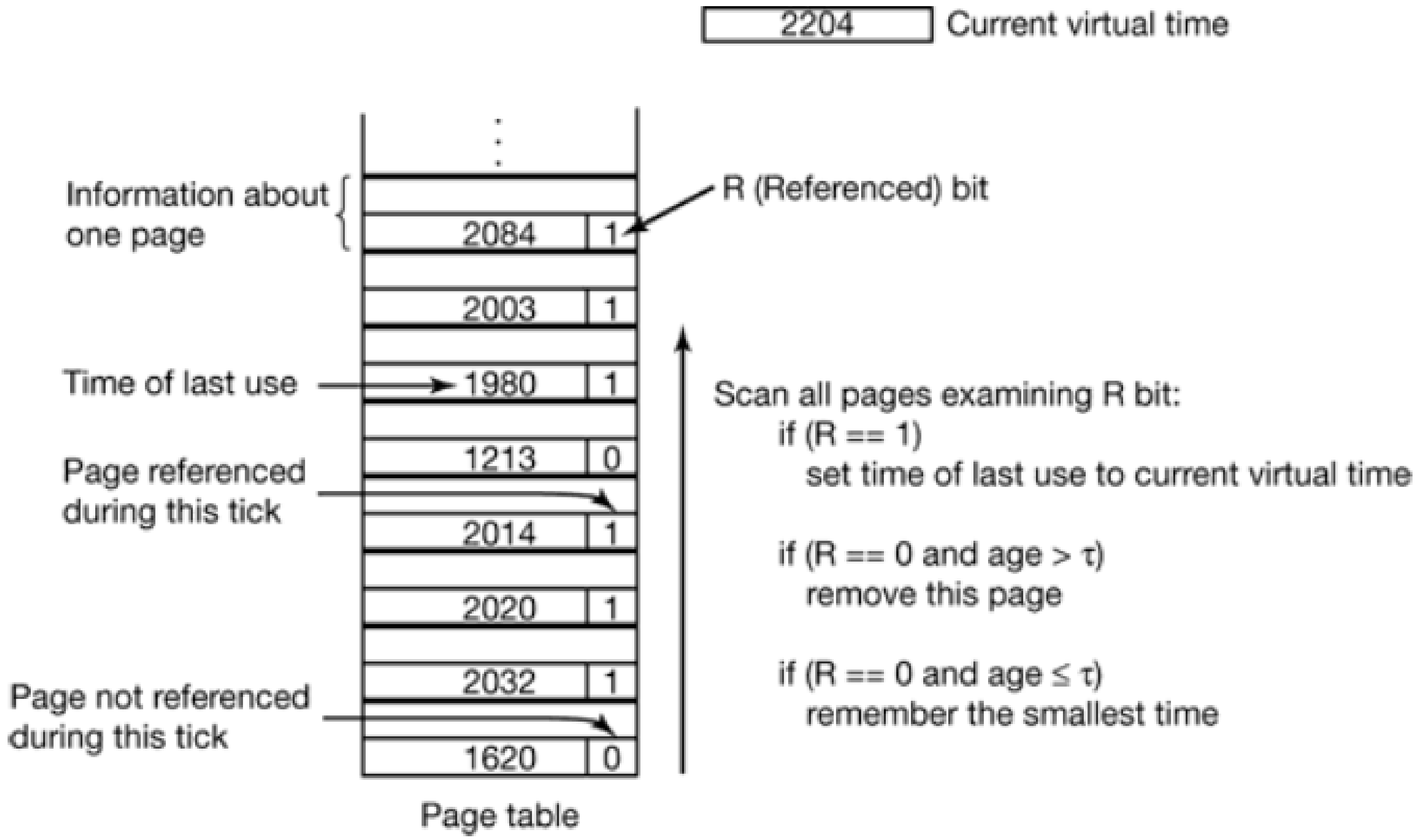
- The working set of pages of process, ws(t, Δ) at time t, is the set of pages referenced by the processes in time interval t-k to t.
- The variable Δ is called working-set-window, the size of Δ is central issue in this model.
- Ex: Working-set-model with Δ = 10



THE WORKING SET PAGE REPLACEMENT ALGORITHM

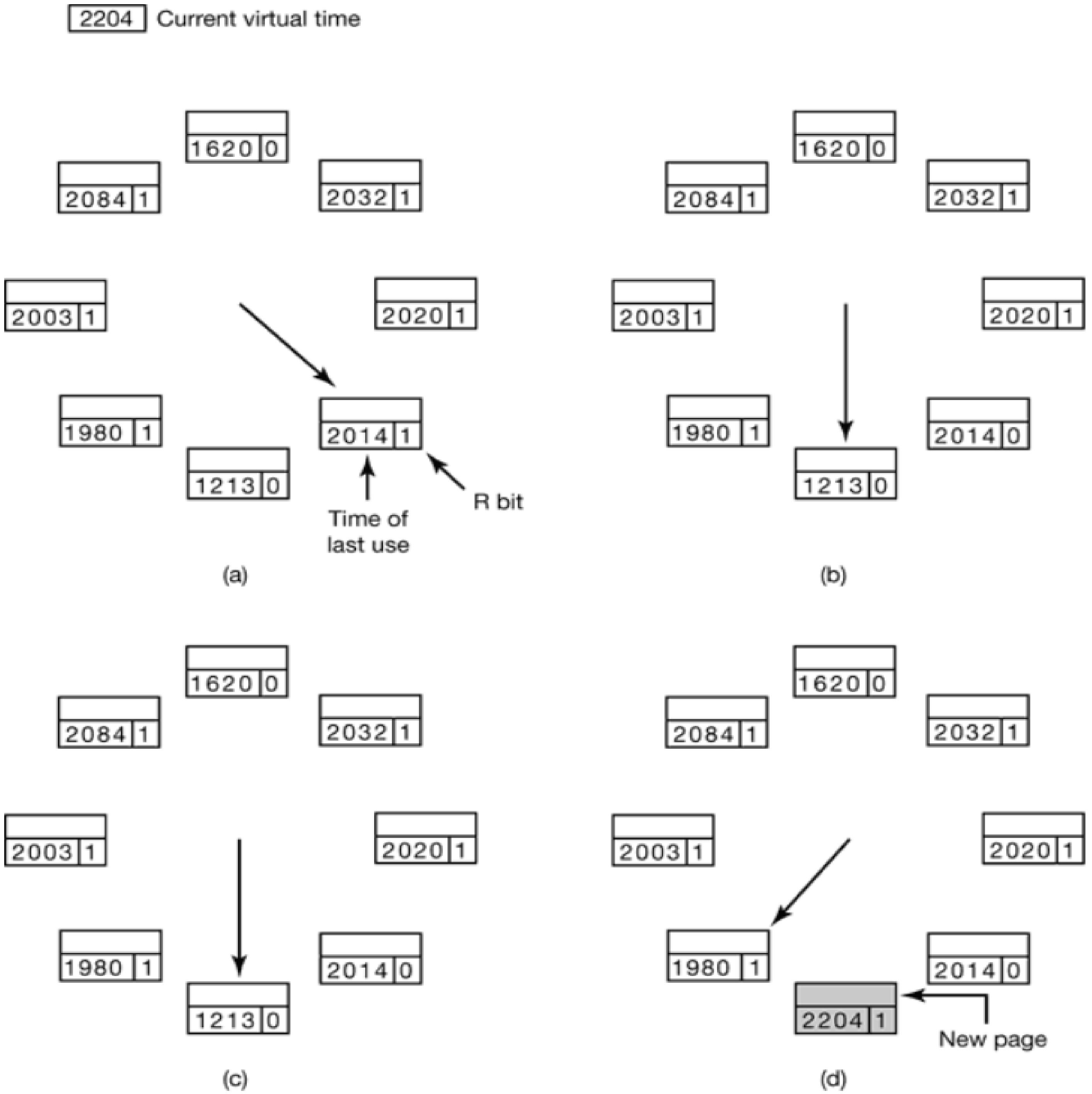
- The amount of CPU time a process has actually used since it started very fist time is often called its **current virtual time.**
- **The basic idea is to find a page that is not in the working set and evict it.**
- *R* bit is examined, if it is 1, the current virtual time is written into the *Time of* ***last use*** **field** in the page table, indicating that the page was in use at the time the fault occurred.
- If *R* is 0, the page has not been referenced during the current clock tick and **may be** a candidate for removal
- If the **age is greater than τ**, the page is no longer in the working set. It is reclaimed and the new page loaded here
  - **Age** is the *current virtual time* **minus** *its Time of last use*

THE WORKING SET PAGE REPLACEMENT ALGORITHM


2204
Current virtual time
Information about one page
R (Referenced) bit
2084
1
2003
1
Time of last use
1980
1
1213
0
Page referenced during this tick
2014
1
2020
1
2032
1
Page not referenced during this tick
1620
0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

THE WSCLOCK PAGE REPLACEMENT ALGORITHM

- The basic **working set algorithm is cumbersome** since the entire page table has to be scanned at each page fault until a suitable candidate is located.
- Each entry **contains the *Time of last use*** field from the basic working set algorithm, as well as the ***R* bit (shown)** and the ***M* bit (not shown)**.
- If the age is greater than τ and the page is clean(not modified), it is not in the working set and a valid copy exists on the disk so it can be removed.



2204
Current virtual time
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 1
1213 0
Time of last use
R bit
(a)
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
1213 0
(b)
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
1213 0
(c)
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
2204 1
New page
(d)

COUNTING-BASED PAGE REPLACEMENT

- keep a counter of the number of references that have been made to each page and develop the following two schemes.
- The **least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.
- A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
- One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- The **most frequently used** (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

SEGMENTATION

- Next Class.