PROJECT 4: DESIGNING A VIRTUAL MEMORY MANAGER

Some slides are adopted from PPTs offered with textbook <Operating Systems Concepts>, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne , WILEY.

OUTLINE

- Note: Project description in file:
  - VirtualMemoryManager-ProgrammingAssignment.pdf
- Class PPT on handling page fault - 1 page
- Useful C library functions - 3 pages
- An overview of the project – 3 pages
- Details on the paging system described in the above PDF file – 14 pages
- On TLB replacement in Part II (Modifications)

HANDLING PAGE FAULT WITH PAGE REPLACEMENT

1. If there is a reference to a page, first reference to that page will trap to operating system (0 in present bit)
   - Page fault
2. Operating system looks at page table to decide present or not
3. // find a free frame

   If there is a free frame, use it
   If there is no free frame, use a page replacement algorithm to select a **victim frame;** write the victim frame to disk if dirty
4. Swap the page in disk into the (newly) free frame via scheduled disk operation
   (the disk address of this page is known to OS)
5. Reset tables to indicate the page now in memory
   (set frame # and present bit = **1**)
6. Restart the instruction that caused the page fault

- Note 3: now potentially 2 page disk transfers for a page fault:

  increasing Effective Access Time

PROJECT: NEW C LIBRARY FUNCTIONS (SYS CALLS)

- MMAP
- #include <sys/mman.h>
- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

- starting address addr for the new mapping, the length of the mapping,
- the file (or other object) referred to by the file descriptor fd; start at offset
- Prot memory protection (PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE)
- Flags visible to other processes mapping the same region, MAP_SHARED, MAP_PRIVATE

mmap: creates a new mapping in the virtual address space of the calling process.

mmap maps memory pages directly to bytes on disk. With mmap, whenever there is a page fault, the kernel pulls the page directly from disk into the program's memory space.

- The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.
- The address of the new mapping is returned as the result of the call.

PROJECT: NEW C LIBRARY FUNCTIONS (SYS CALLS)

- MEMCPY
- #include < string.h >
- void *memcpy(void *dest, const void *src, size_t n);

Copies n bytes from memory area src to memory area dest. The memory areas must not overlap.

returns a pointer to dest.

memcpy - copy memory area //copy pages from backing fi into memo

// dest: frame location in physical memory ,

// src: the page in the backing store

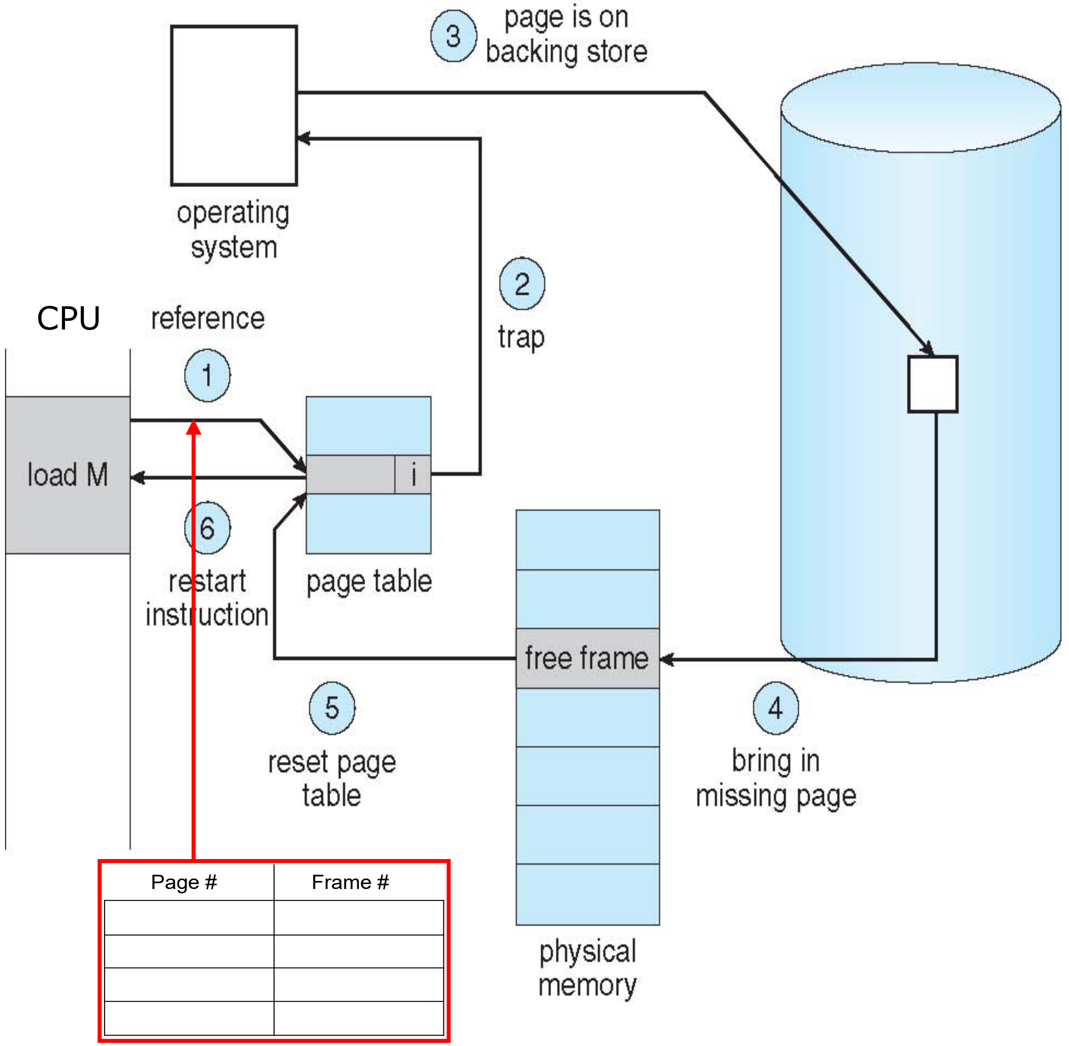
PROJECT: NEW C LIBRARY FUNCTIONS (SYS CALLS)

- **Random Access**
- Randomly seek to certain positions of the file for reading.
- Standard C library I/O functions, `fopen()`, `fread()`, `fseek()`, and `fclose()`.

- `int fseek(FILE *stream, long offset, int whence);`
- Sets the file position indicator for the stream pointed to by stream.
- The new position (in bytes), by adding `offset` bytes to the position `whence`.
- `whence` can be: `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, for the `start of the file`, `the current position indicator`, or `end-of-file`, respectively.

PROJECT 4 OVERVIEW

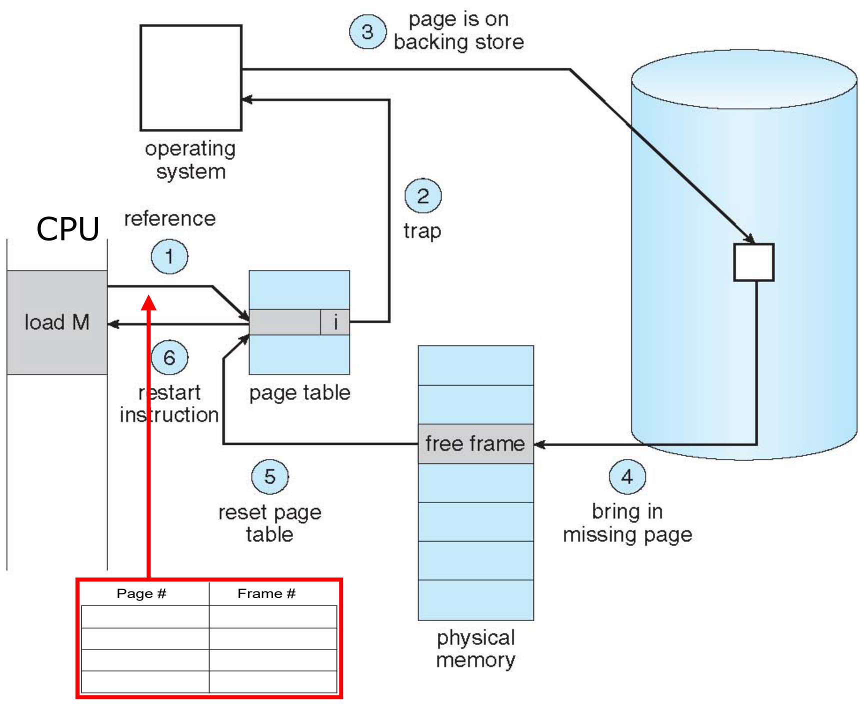
- Project 4 will be on paging system,
  - Page table, TLB, Page Faults
- TLB entries can be replaced when there is a need.
  - The replacement algorithm used by TLB!
- Part I: enough physical memory
  - Focusing on paging
- Part II: NOT enough physical memory
  - Adding page replacement



PROJECT 4: STEPS OF PURE DEMANDING PAGING WITH TLB - SUMMARY

Part I – **enough** physical memory

- Pure demand paging!
  - Starting with the first page: Page fault
- Build page table
- Build TLB



Read a page (256 bytes) and store in physical memory (update page table, TLB entry, if needed)

- Always has free physical frames in this part
- Pages are stored in a file. File I/O uses random access.
- Physical memory is an array

Address translation (p, d):
If p is in TLB, get frame #;
Otherwise, get frame # from page table in memory (a TLB miss); also load the page entry to TLB
TLB replacement using FIFO

PROJECT 4: STEPS OF PURE DEMANDING PAGING WITH TLB - SUMMARY

Part II "Modifications" – **NOT enough** physical memory

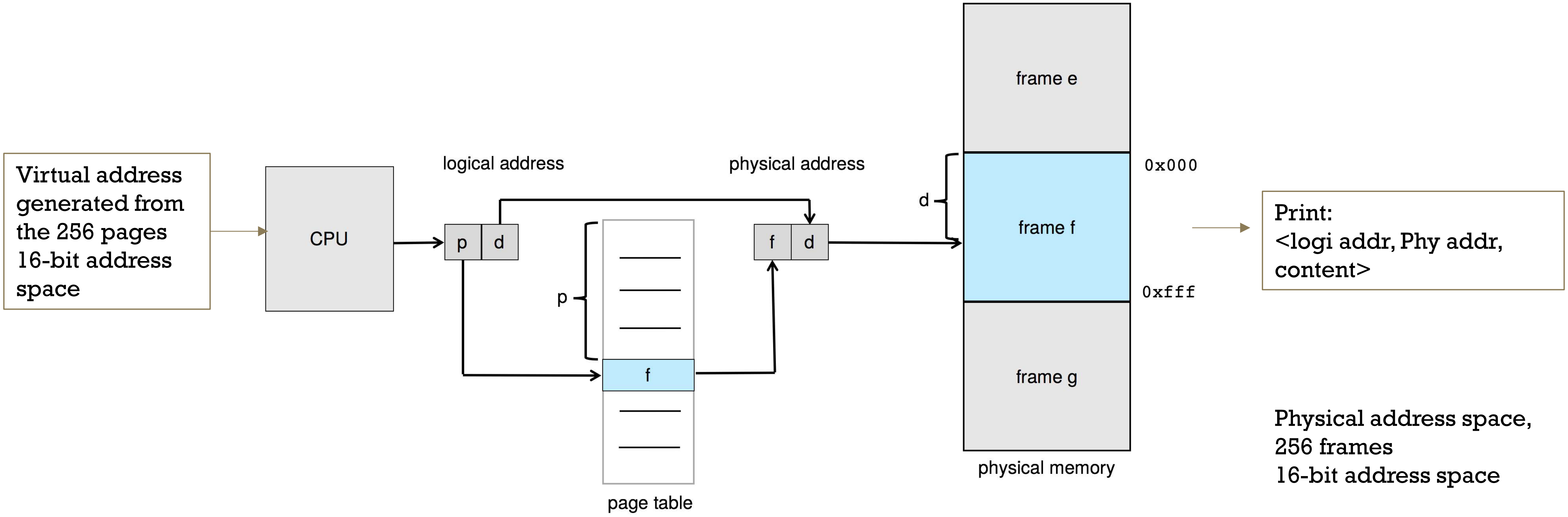
- Apply Page Replacement to Part I
- Page replacement using LRU

Part I and II: Evaluation

Number of TLB misses

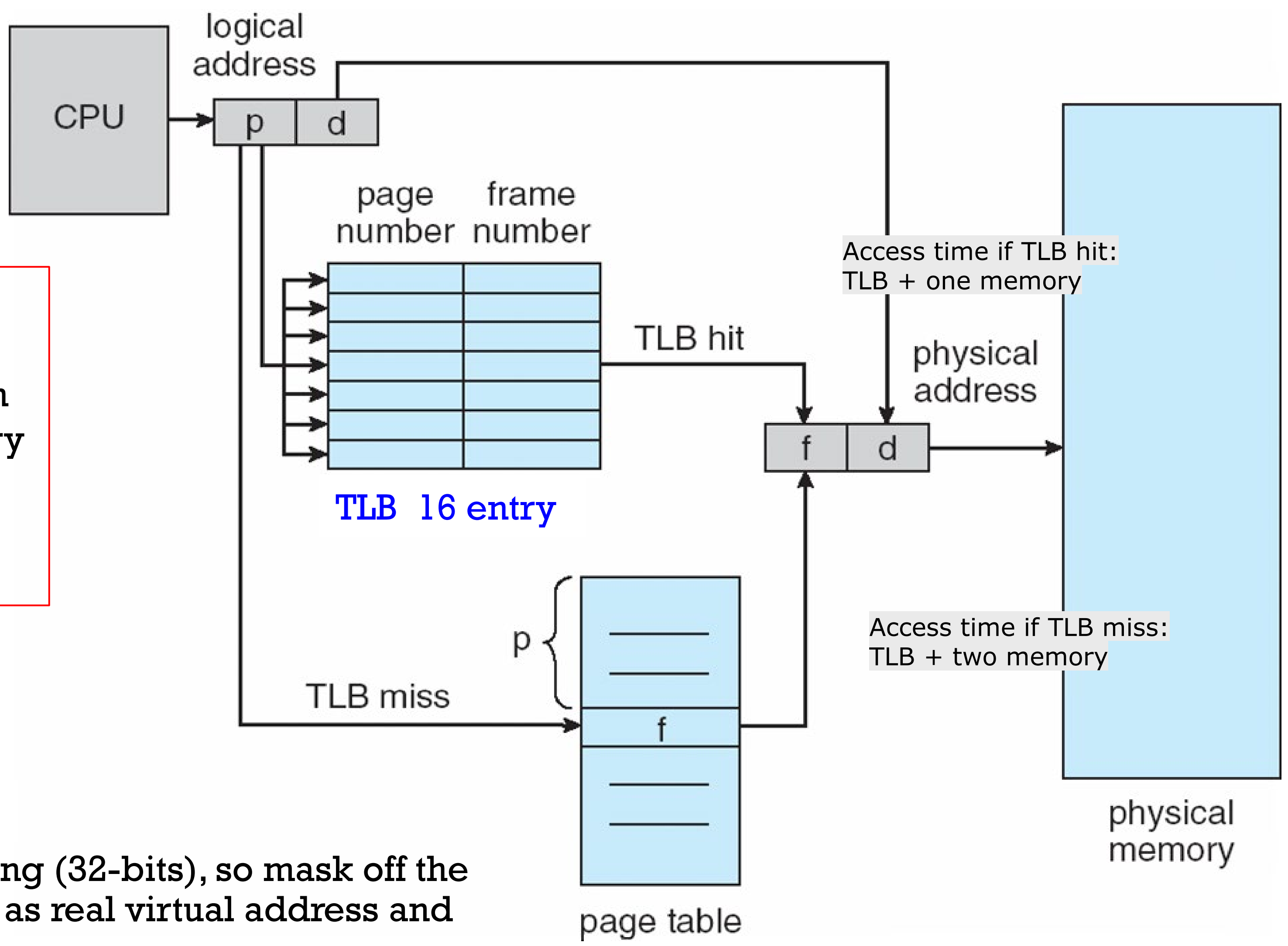
Number of page faults

PROJECT: STEPS OF PAGING (1) WITHOUT TLB FIRST



Implement and test with only paging first!
Add TLB only when paging part is correct.

PROJECT: STEPS OF PAGING WITH TLB (2)



Address translation (p, d):
If p is in TLB, get frame #;
Otherwise, get frame # from page table in memory (a TLB miss); load the page entry to TLB

TLB replacement using FIFO

- Addresses: the inputs are too long (32-bits), so mask off the first 16bits, use the later 16 bits as real virtual address and physical

PROJECT: STEPS OF PURE DEMANDING PAGING WITH TLB - SUMMARY

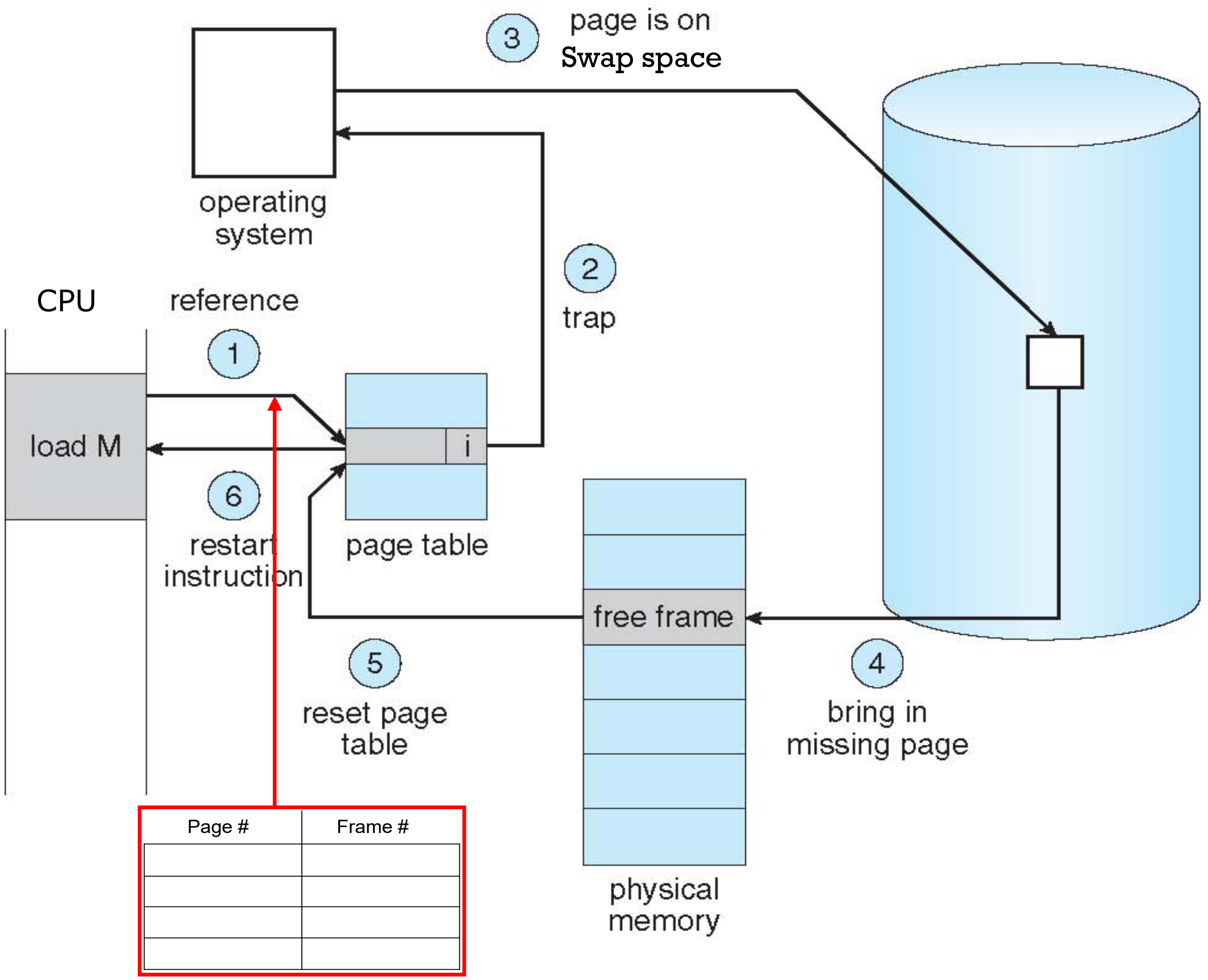
- Pure demand paging!
  - Starting with the first page: Page fault

Read a page (256 bytes) and store in physical memory (update page table, TLB entry, if needed)

- Always has free physical frames in this part

File I/O uses random access.

Address translation (p, d):
If p is in TLB, get frame #;
Otherwise, get frame # from page table in memory (a TLB miss)
load the page entry to TLB
TLB replacement using FIFO



PROJECT: THE SYSTEM

program is stored in the backing store.

Effective bits for addresses: 16 bits [8 bits, 8bits]
256 pages, page size 256 B (each address stores one byte data)
The total virtual memory space of the program: 256 x 256 B = 2^16 B = 64KB (i.e. 65,536B)

Virtual address space

How many entries in the page table?

page table 256 entries

The inputs of addresses are too long: 32-bits!

So mask off the first 16bits, **use the later 16 bits as real virtual address**

Physical address space, 256 frames, frame size 256 B
16-bit address space

In Part 2 Modification, physical memory will be less.
E.g., only 128 frames, or 32 frames
Start Part 2 after Part 1, copy code from Part 1, then revise.

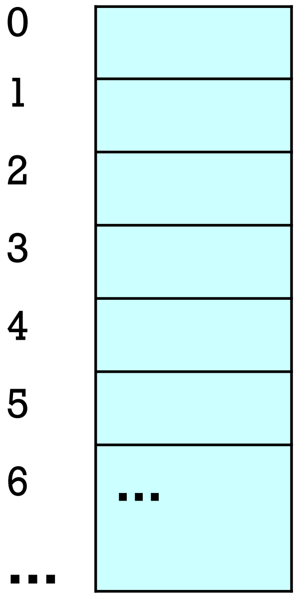
PROJECT: THE SYSTEM

Add TLB

The backing store

TLB is smaller than the page table. What to do?

Virtual address space, 256 pages
16-bit address space



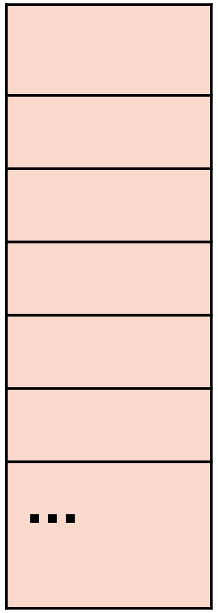
TLB 16 entry

Page #	Frame #

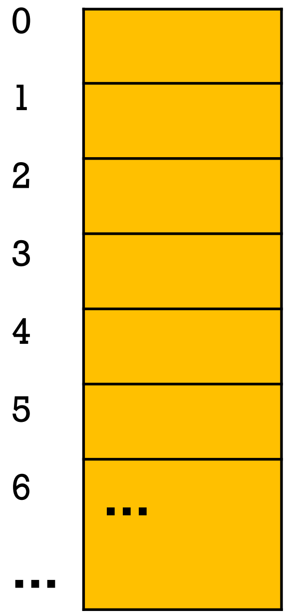
TLB replacement: a new page entry replaces an existing page entry in TLB

Use FIFO (first in first out), the same a FCFS

page table 256 entry



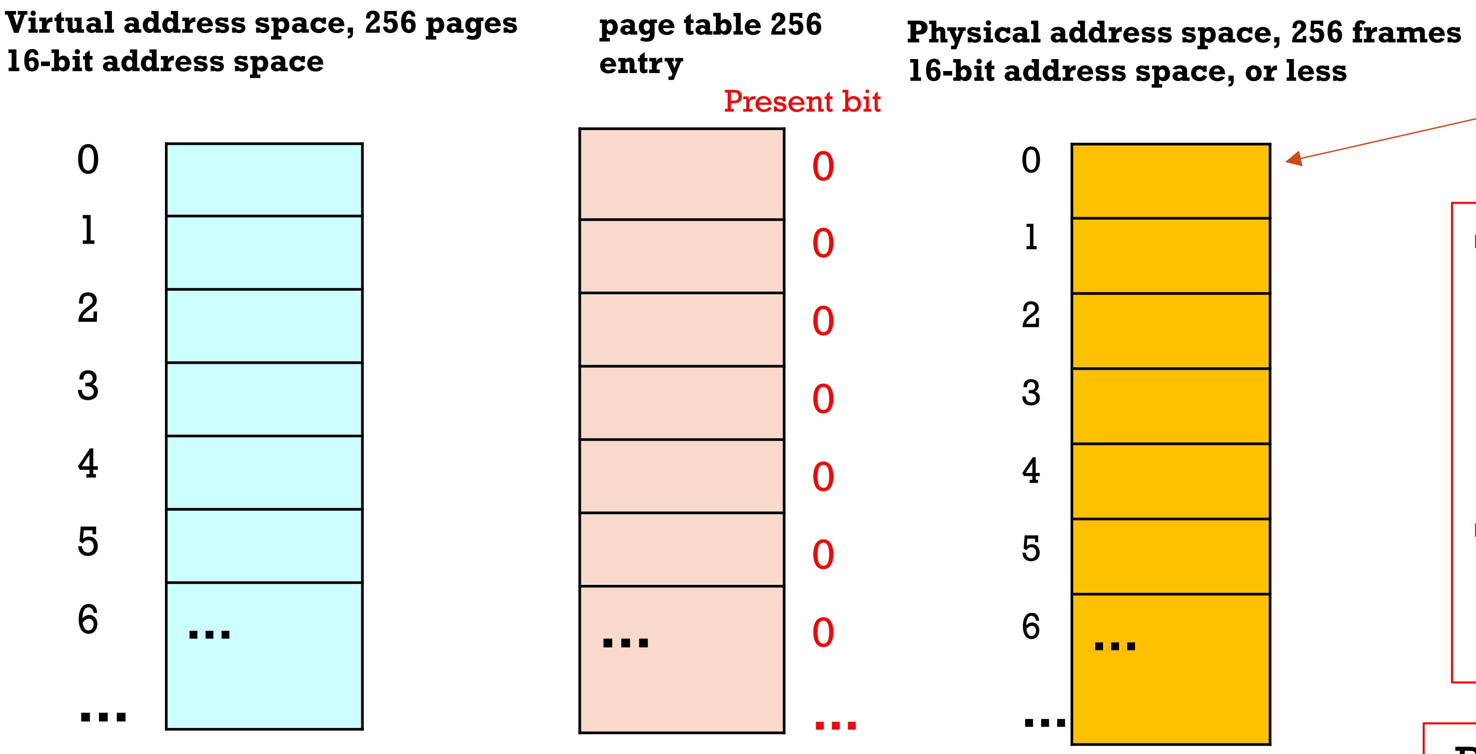
Physical address space, 256 frames
16-bit address space, or less



PROJECT: PURE DEMAND PAGING!

ADDRESS TRANSLATION, PAGE FAULT, LOADING CONTENTS

E.g., a reference list: 2 1 5 6 1 2 0 …



Free frame list

Page #	Frame #

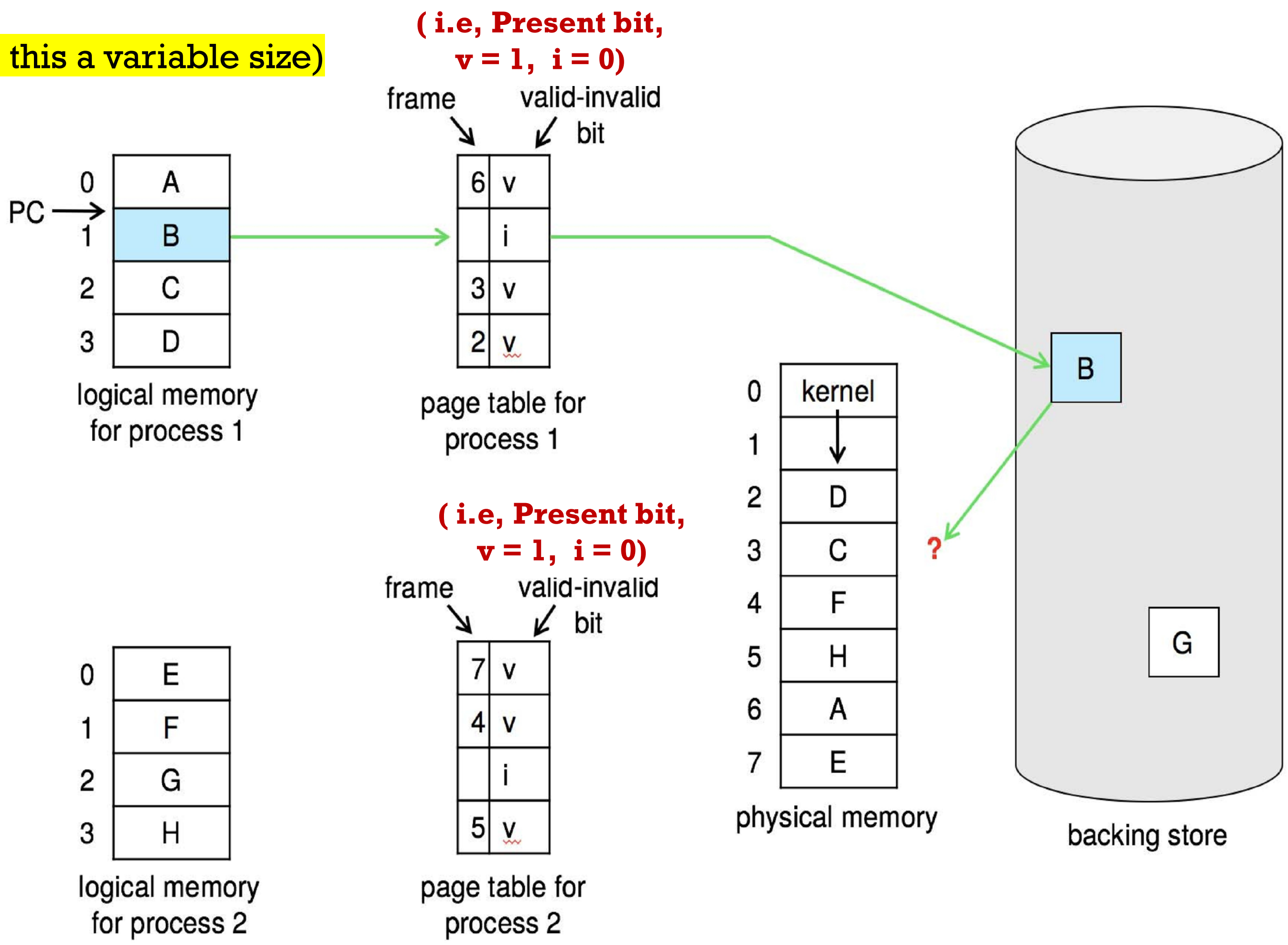
TLB 16 entry

- Pure demand paging!
  - Starting with the first page: Page fault
  - Handle page fault
- Update page table, TLB entry, if needed

Read one page a time (using logical address) store it to a frame: File I/O reading the backing store uses random access.

PROJECT: MODIFICATION - ADDING PAGE REPLACEMENT

- Reduce the physical memory to 128 frames (reduced half) Revised to make this a variable size)
  - At one point, there will be no free frames when referencing to a new page.
  - Pages replacement
  - Also, need to add the ability to keep track of free frames
- Handling Page Fault with Page Replacement (move one page to the backing store, use the freed frame for the new page)
  - Possibly TLB needs update too.
- Page replacement using LRU
- Memory references: read in from an addresses.txt
- Address translation and access. Possibly handle three cases:
  - TLB miss
  - Page fault
  - Page fault and replacement
- Reading in the page from backing store
- Output in one line



PROJECT: TESTING AND STATISTICS

- Memory reference: read in from a file (e.g. addresses.txt)

Reference string

- Address translation and access. Possibly handle two cases:
  - TLB miss
  - Page fault
- Reading in the page content (one byte) from backing store
- Output in one line

<logical address, phy address, content byte>

//types: <integer, integer, signed byte>

Add variables for statistics:

- # of TLB hit; // for calculate TLB hit rate before your program finishes
- # of page fault; // for calculate Page fault rate at the end

Testing cases:

(1) Give a couple logical addresses, manually work out the physical addresses. See whether your program output the same

(2) You can create your own small test file manually.

- e.g., 16 or 32 reference addresses.

(3) One large test file and answer is provided.

- Statistics:
  - Hit rate: # of hits/ total references
  - Page fault rate: # of page fault / total references
- Test each component when it is finished, before moving to next components.
  - Write your own test cases, e.g., a few addresses.
- You are read to work on the project!
- If you started, the remaining pages may not be helpful to you.

PROJECT 2, PART 2, REPLACEMENT

- When the number of frames is smaller than the virtual address space, also smaller thatn the size of the TLB
  - Q: with page replacement, what if the page has an entry in TLB? What to do with it when the page to be evicted, and a new page will bring in?
- The textbook we use doesn't talk about it.
- We will test the # of frames always >= 16 (the TLB size).
- Use whatever you coded in Part 1 regarding TLB for Part 2

- Otherwise, for your reference:
- The TLB must be kept in sync with the page table. When a page in physical memory is replaced, its TLB entry, if there is one for it, must be invalidated.
- Update the TLB entry for the page being replaced to contain the virtual-to-physical mapping for the new page being loaded in. Do not advance the round-robin (FIFO) replacement index in this case, if round-robin replacement is being used for the TLB

https://www.cs.unh.edu/~pjh/courses/cs520/14spr/prog7.html#:~:text=The%20TLB%20must%20be%20kept,new%20page%20being%20loaded%20in.

MORE Q/A

- How page fault impact TLB hit rate?
  - TLB FIFO
  - TLB page size vs phy memory size (when Phy memory is smaller than 16 TLB)
- LRU, associated with free frames or with page table (pages)
- Getting page # and offset: binary op or integer op?
- Part II correct addresses?