Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Virtual Memory

Schedule

- **Today and Monday**
  - Chapter 6 – Virtual memory
- **Next Wednesday**
  - Chapter 7 – Input / Output Systems
- **Next Friday (March 30th)**
  - **Quiz #5** – Chapter 6 (Cache & Virtual Memory)

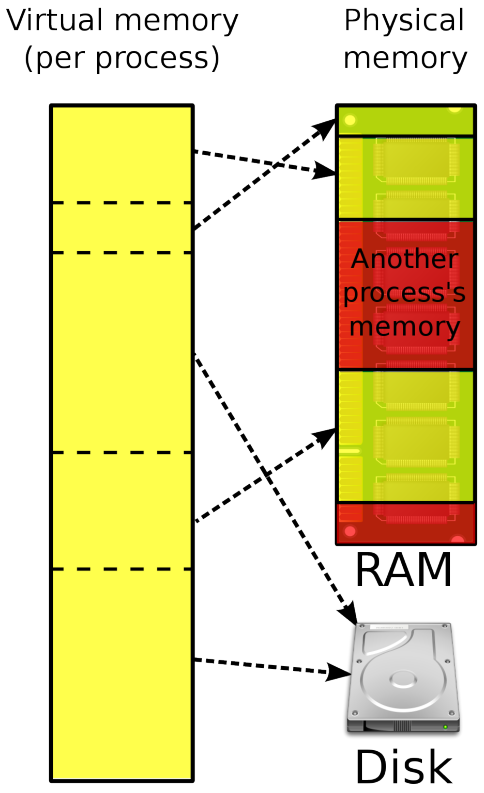
Virtual Memory

Virtual Memory

Virtual Memory is a BIG LIE!

- We **lie** to your application and tell it that the system is simple:
  - Physical memory is infinite! (or at least huge)
  - You can access *all* of physical memory
  - Your program starts at *memory address zero*
  - Your memory address is *contiguous* and *in-order*
  - Your memory is *only RAM* (main memory)

What the System Really Does



Why use Virtual Memory?

- We want to run multiple programs on the computer concurrently *(multitasking)*
  - Each program needs its own separate memory region, so physical resources must be divided
  - The amount of memory each program takes could vary dynamically over time (and the user could run a different mix of apps at once)
- We want to use multiple types of storage (main memory, disk) to increase performance and capacity
- We don't want the programmer to worry about this
  - Make the processor architect handle these details

Cache Memory vs Virtual Memory

- Goal of cache memory
  - Faster memory access speed (**performance**)
- Goal of virtual memory
  - Increase memory **capacity** without actually adding more main memory
    - Data is written to disk
    - If done carefully, this can improve performance
    - If overused, performance suffers greatly!
  - Increase system flexibility (as previously discussed)

Pages and Virtual Memory

- Main memory is divided into "blocks" called **pages** for virtual memory
  - Why use the term *page* instead of *block?*
    - This is a different concept than cache blocks!
    - Pages are **larger** – generally 4kB in size
    - Blocks are 64 bytes in size (on modern Intel)
  - Data is moved between main memory and disk at a page granularity
    - i.e. we don't move single bytes around, but rather big groups of bytes

Virtual Memory

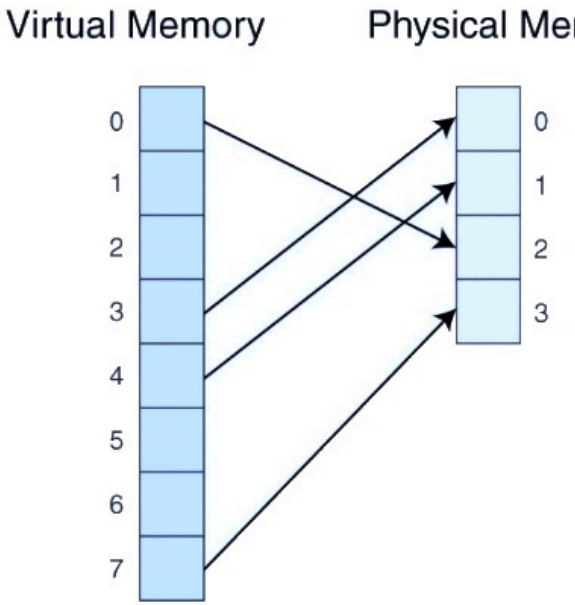
- Main memory and virtual memory are divided into equal sized pages
- The entire address space required by a process need not be in memory at once
  - Some pages can be on disk
    - Push the unneeded parts out to slow disk
  - Other pages can be in main memory
    - Keep the frequently accessed pages in faster main memory
- The pages allocated to a process do not need to be stored contiguously-- either on disk or in memory

Virtual Memory Terms

- **Physical address** – the actual memory address in the *real* main memory
- **Virtual address** – the memory address that is seen in your program
  - We need some special hardware/software to map between virtual addresses and physical addresses!
- **Page faults** – a program accesses a virtual address that is not currently resident in main memory (at a physical address)
  - The data must be loaded from disk!
- **Pagefile –** The file on disk that holds memory pages
  - Usually twice the size of main memory

Mapping: Virtual → Physical Address

- **Page Table** tracks location of each page (whether on disk or in memory)
  - One page table for each active process (application)



Page Table

Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

Mapping: Virtual → Physical Address

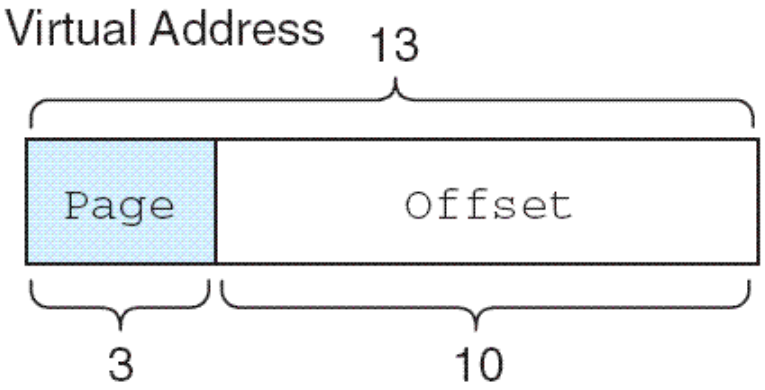
- A process – like your program – generates a virtual address (aka "*logical address*")
- The operating system translates the virtual address into a physical memory address
- Virtual address is divided into two fields
  - **Page field** – Page location of the address
  - **Offset field** – Location of the address within the page
- The **logical page number** (from the virtual address) is translated into a **physical frame number** through a lookup in the page table
  - Page number = part of virtual address
  - Frame number = part of physical address

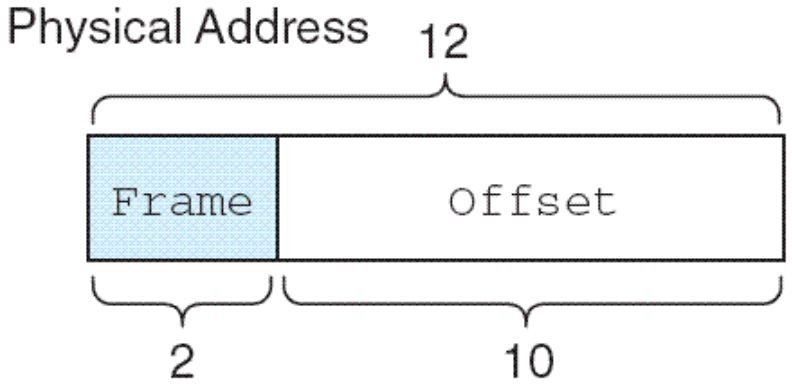
Mapping: Virtual → Physical Address

- Check the valid bit in the page table entry!
  - Valid bit = 0
    - **Page fault!**
    - Page is not in memory and must be fetched from disk
    - If necessary, a page is evicted from memory and is replaced by the page retrieved from disk, and the valid bit is set to 1
  - Valid bit = 1
    - Page is in main memory, and we know where!
    - Replace virtual page number with the physical frame number from the page table
    - Data can be accessed by adding the offset to the physical frame number

Mapping: Virtual → Physical Address

- Example:
  - Byte-addressable system with 1024 byte pages
  - Virtual address space of 8K; Physical address space of 4K
- **What do we know?**
  - We have $2^{13}/2^{10} = 2^3 = 8$ pages in virtual memory
  - Virtual address has 13 bits ($8K = 2^{13}$): 3 bits for **page** and 10 bits for **offset**
  - Physical address has 12 bits: 2 for frame and 10 bits for offset





Mapping: Virtual → Physical Address

- Suppose this system has the following page table:
  - **What happens when program generates address $5459_{10} = 1010101010011_2 = 1553_{16}$?**

Page Table

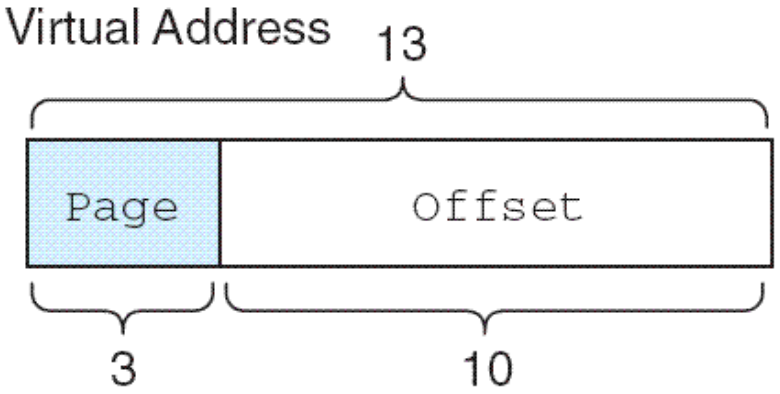
Page	Frame	Valid Bit
0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Addresses

Page	Base 10	Base 16
0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

Mapping: Virtual ➔ Physical Address

- What happens when the program generates address $5459_{10} = 1010101010011_2 = 1553_{16}$?



The high-order 3 bits of the virtual address are 101 (5_{10})
This is the page number to lookup in the page table

Mapping: Virtual ➔ Physical Address

- Virtual address 1010101010011_2
- Physical address $0101010010011_2 = 1363_{16}$
  - Page field 101 is replaced by frame number 01 through a lookup in the page table

Page Table

Page	Frame	Valid Bit
0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Addresses

Page	Base 10	Base 16
0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

Mapping: Virtual → Physical Address

- **What happens when the program generates address 1000000000100_2?**

Page Table

Page	Frame	Valid Bit
0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Addresses

Page	Base 10	Base 16
0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

Relationships *(for Homework problem)*

- If data exists in main memory, it must have a valid entry in the page table
  - Entry not valid? Data must be paged to disk
- You can't have an entry in the cache that doesn't exist in main memory
  - i.e. if data gets paged out to disk, it is also removed from the cache
  - This makes sense – we only page out infrequently accessed data to disk anyway!

Effective Access Time (again)

Effective Access Time

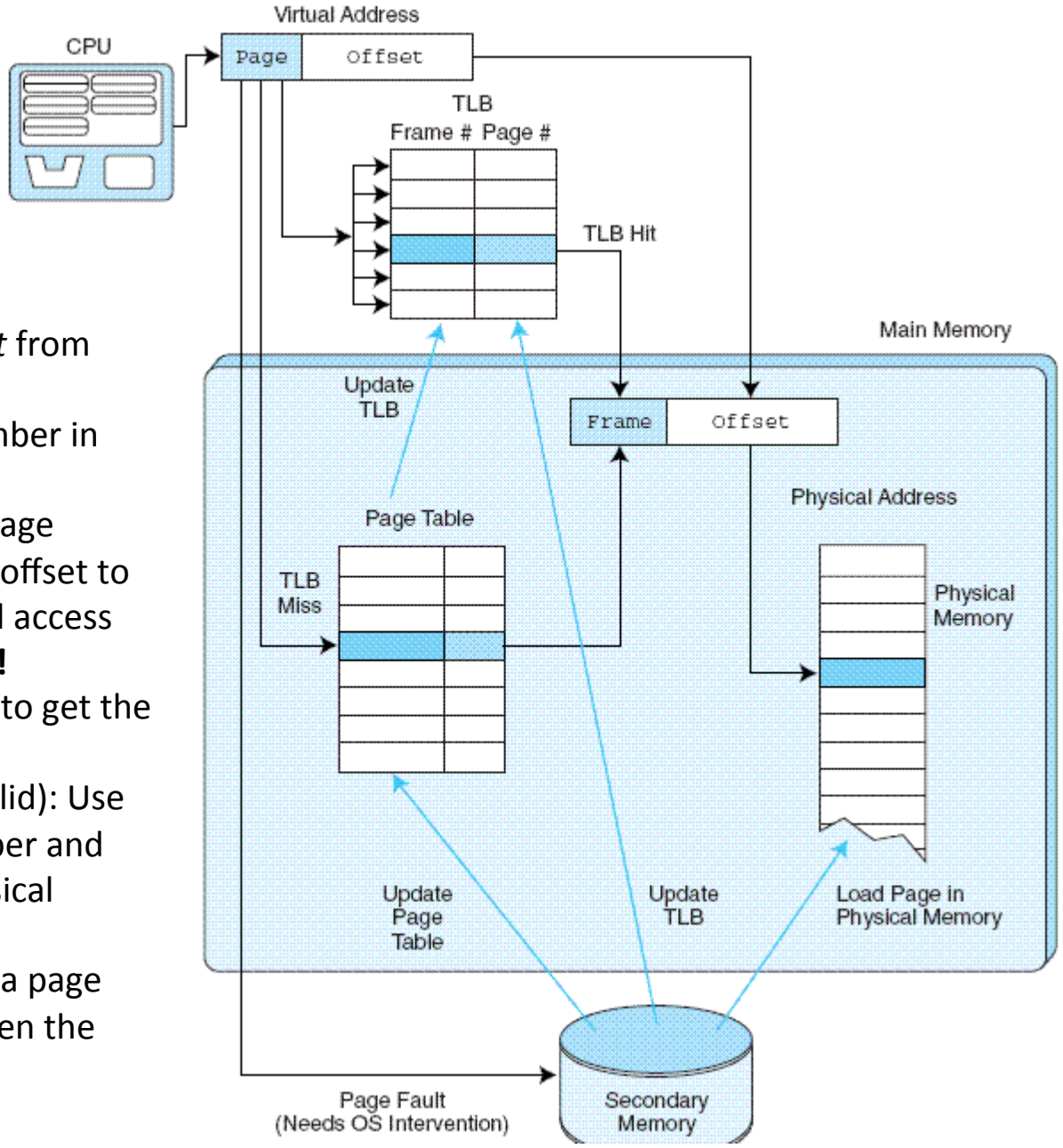
- Effective access time (EAT) takes all levels of memory into consideration
  - Previously we only included *cache* and *main memory*
  - Now we add *page table translation* and *virtual memory* (disk)…
- Example: Suppose a main memory access takes 200ns, the page fault rate is 1%, and it takes 10ms to load a page from disk
  - EAT = %mem(mem time) + %disk(disk time)
  - EAT = 0.99(200ns + 200ns) + 0.01(10ms) = 100,396ns
  - **Why is the memory time 200+200ns?**

Effective Access Time

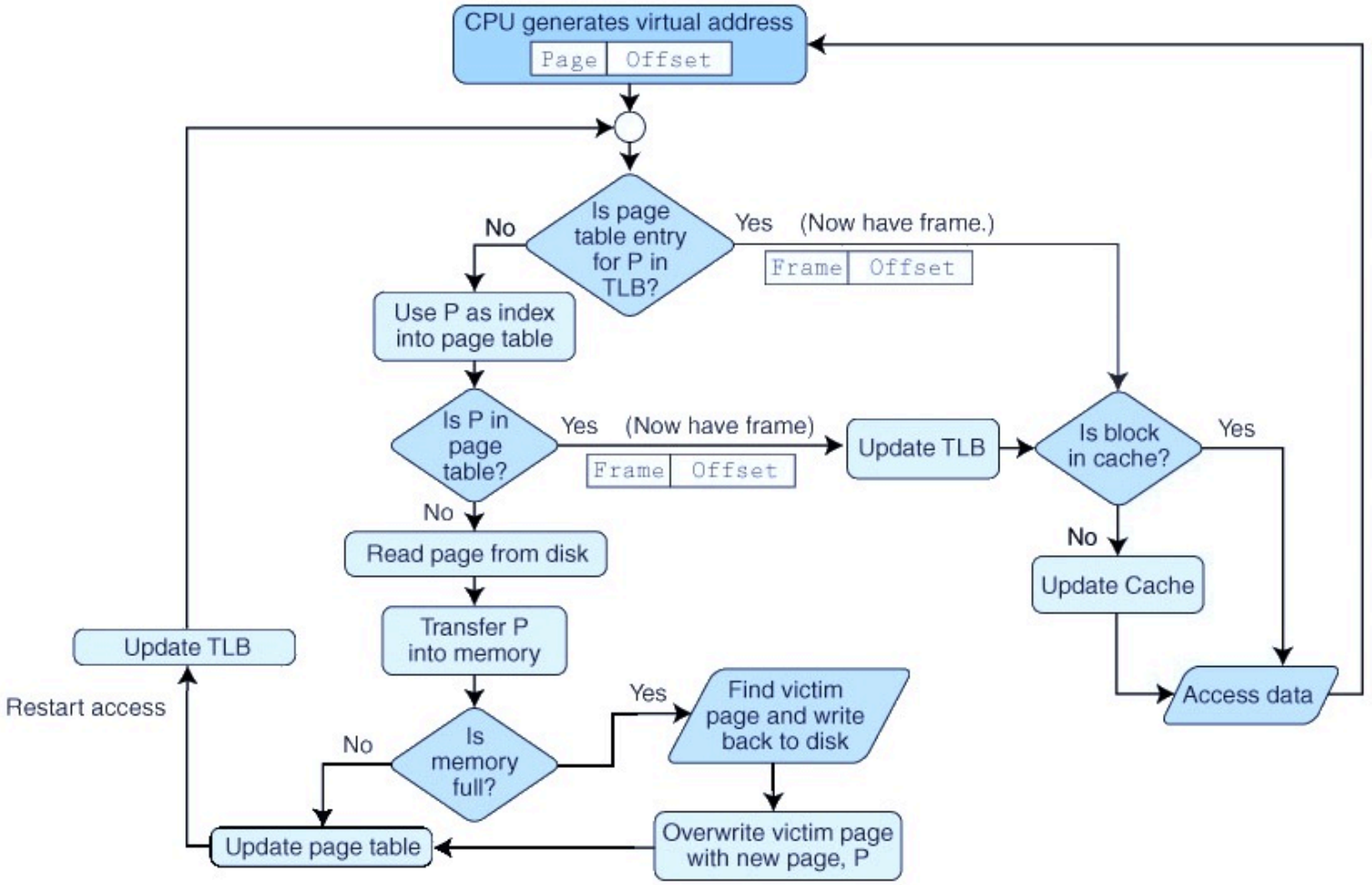
- **Why was the memory time 200+200ns?**
  - Even if we had no page faults, the EAT would be 400ns because memory is always read twice
    - **First** to access the page table
    - **Second** to load the page from memory.
- Observation: **Page table is read for every memory access! (Yikes!!)**
- **Suggestions to improve this?**
  - Make a special **cache** just for page table data
  - **Translation look-aside buffer (TLB)**
    - **Fully associative cache** that *only* stores the mapping of virtual pages to physical pages

TLB Lookup Flow Chart

1. Extract *page number* and *offset* from virtual address
2. Search for the virtual page number in the TLB (cache)
3. TLB Hit: If the (virtual page #, page frame #) pair is found, add the offset to the physical frame number and access the memory location. **Finished!**
4. TLB Miss: Go to the page table to get the necessary frame number.
5. Page in memory (page table valid): Use the corresponding frame number and add the offset to yield the physical address.
6. Page not in memory: generate a page fault and restart the access when the page fault is complete



TLB + Page Table + Main Memory + Cache



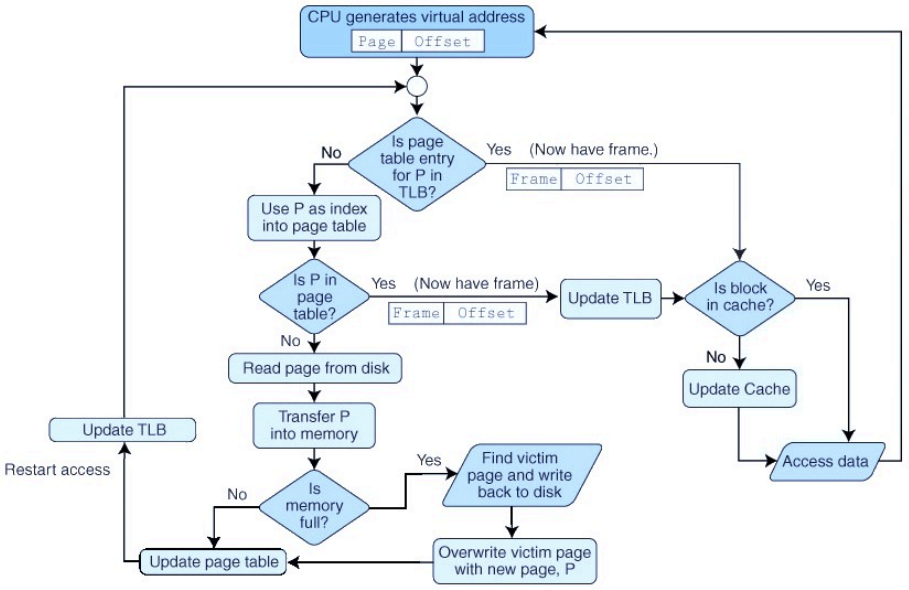
Example – Access Time

- Suppose we have a virtual memory (VM) system with a TLB, cache, and page table. Also assume:
  - A TLB hit takes 10ns, and has a hit ratio of 92%
  - A cache hit takes 20ns (hit ratio 98%)
  - A physical memory reference takes 45ns (page fault rate 0.025%)
  - A disk reference takes 150ms (including loading the page table and TLB)
- For a page fault, the page is loaded from disk and TLB is updated, and memory access restarts
- **How long does it take to access a word if it is not in physical memory?**

Example – Access Time

- Page in TLB (10ns) → no
- Page in physical memory (45ns) → no
- Read page from disk into memory (150ms) → restart
  - Assumes a free page
- Page in TLB (10ns) → yes
- Block in cache (20ns) → no
- Load block from physical memory (45ns) → done
- Total: 10ns + 45ns + 150ms + 10ns + 20ns + 45ns = **150,000,130ns**
  - Aka a **long time**

How long does it take to access a word if it is not in physical memory?



Example – Access Time

- **How frequently does this happen? (i.e. how often must we go to disk?)**
  - TLB (10ns, 92%), Cache (20ns, 98%), Memory (45ns, 1-0.025%), Disk (150ms)
- Process
  - Page in TLB → 92%
  - Page not in TLB (requires access to page table) → 8%
  - Page not in page table (i.e. not in memory) → 0.025%
  - Page not in TLB **and** not in page table:
    - 8% * 0.025% = .08 * .00025 = 0.00002 (or 0.002%)
- **What is the *effective access time* for a page fault?**
  - .00002 * 150,000,130ns = 3,000.0026ns