Page Fault
(*how to minimize)*

Multiple PF Interrupts (from ONE asm inst)

CPU microcode generates THREE virtual addresses for the loading the data

```
                    (Page 7: Code)
// in C: a = b + c
add  $7FF004, $6B00, $66644D
```

(Page 17: Data)

(Page 22: Data)

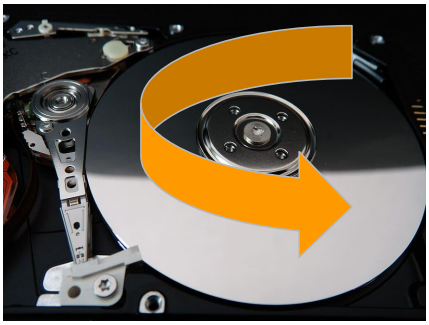
(Page 53: Data)

Handling Page Faults (in details)

- The process experiencing Page Fault Interrupt (PFI) is blocked
- The OS initiates swap disk I/O: read in the missing page
- Direct Memory Access (DMA) handles swap disk I/O operation (while the OS performs other tasks)
- DMA notifies I/O completion (via interrupt)
- The OS handles interrupt and resumes the PF handling (update the process page table and other data structures)
- Resume the faulty process from the instruction that causes PFI

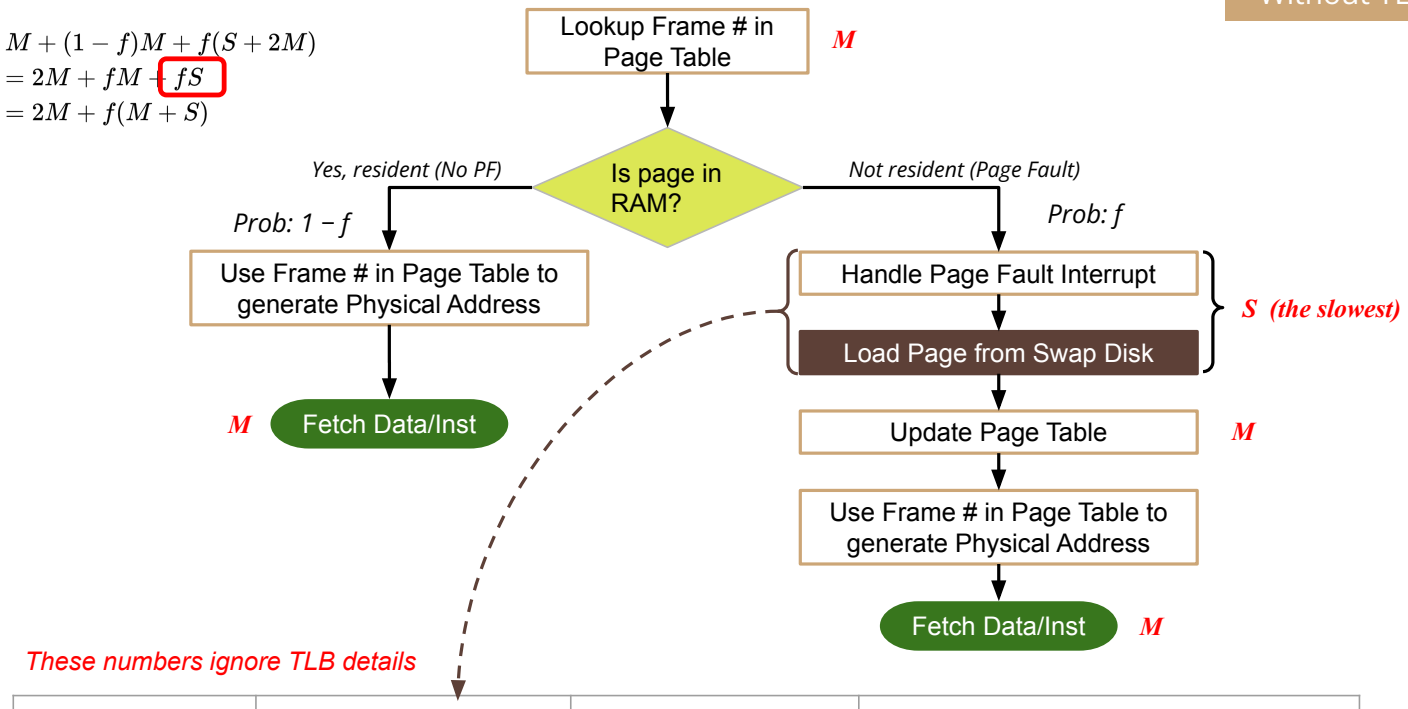
Demand Paging Effective Access Time

- RAM access time (*faster is preferred*)
- TLB access time and TBL hit/TLB miss (*higher hit ratio is preferred*)
- Page fault rate (*low PageFault rate is preferred*)
- Page fault service time (*short PageFault service time is preferred*)
  - Time to run page fault interrupt handler
  - Time to perform **I/O to/from swap disk (the slooooooowest)**
  - Time to update Page Table



Without TLB

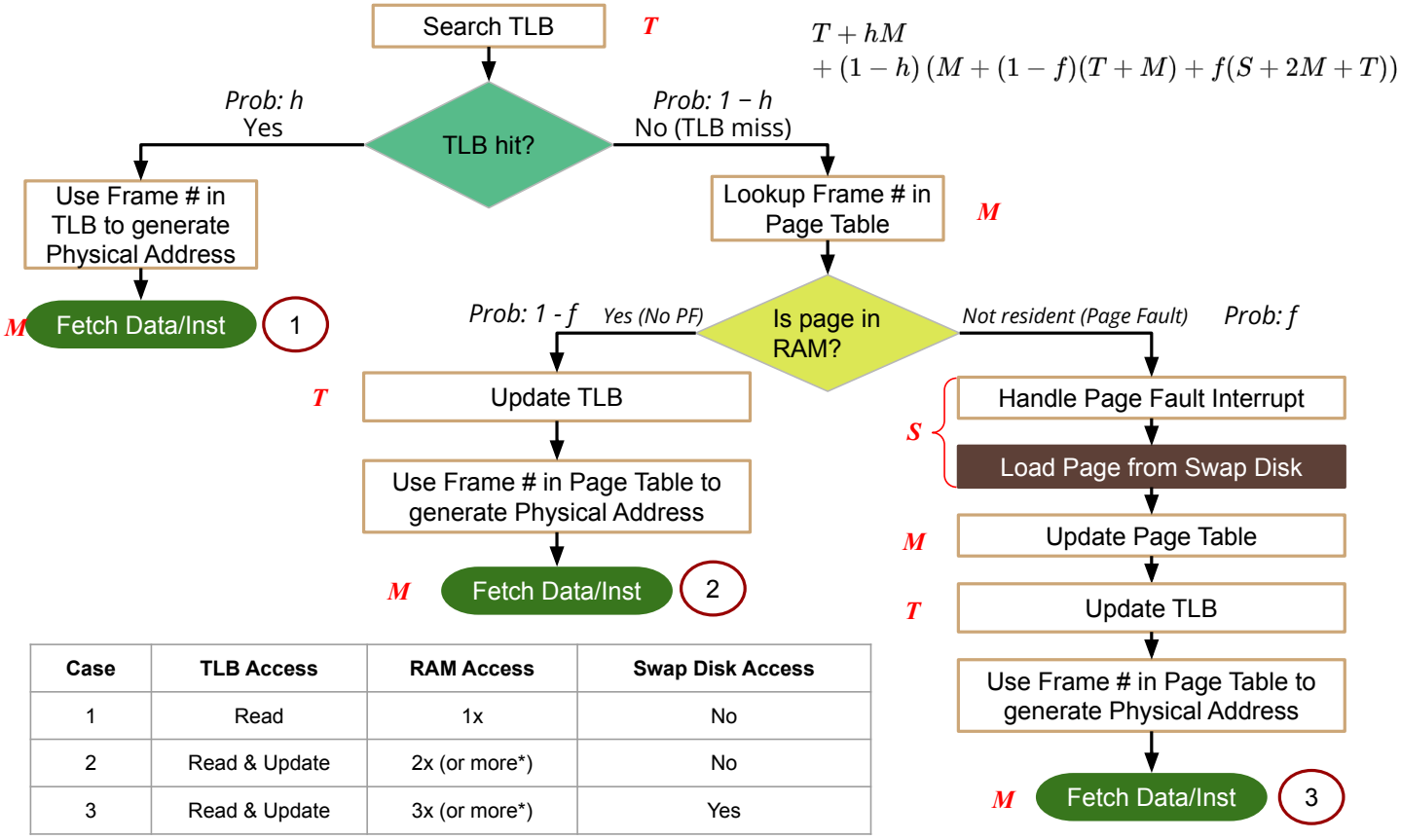
$$M + (1-f)M + f(S+2M)$$
$$= 2M + fM + fS$$
$$= 2M + f(M+S)$$



These numbers ignore TLB details

RAM Access (M)	Page Fault Service (S)	Page Fault Ratio (f)	Total Access Time
200 ns	8 ms (= 8,000,000 ns)	0.001	2x200 + 0.001x8,000,200 = 8,400 ns
200 ns	8 ms (= 8,000,000 ns)	0.00001	2x200 + 0.00001x8,000,200 = 1,200 ns

$$T + hM + (1-h)\,(M + (1-f)(T+M) + f(S+2M+T))$$



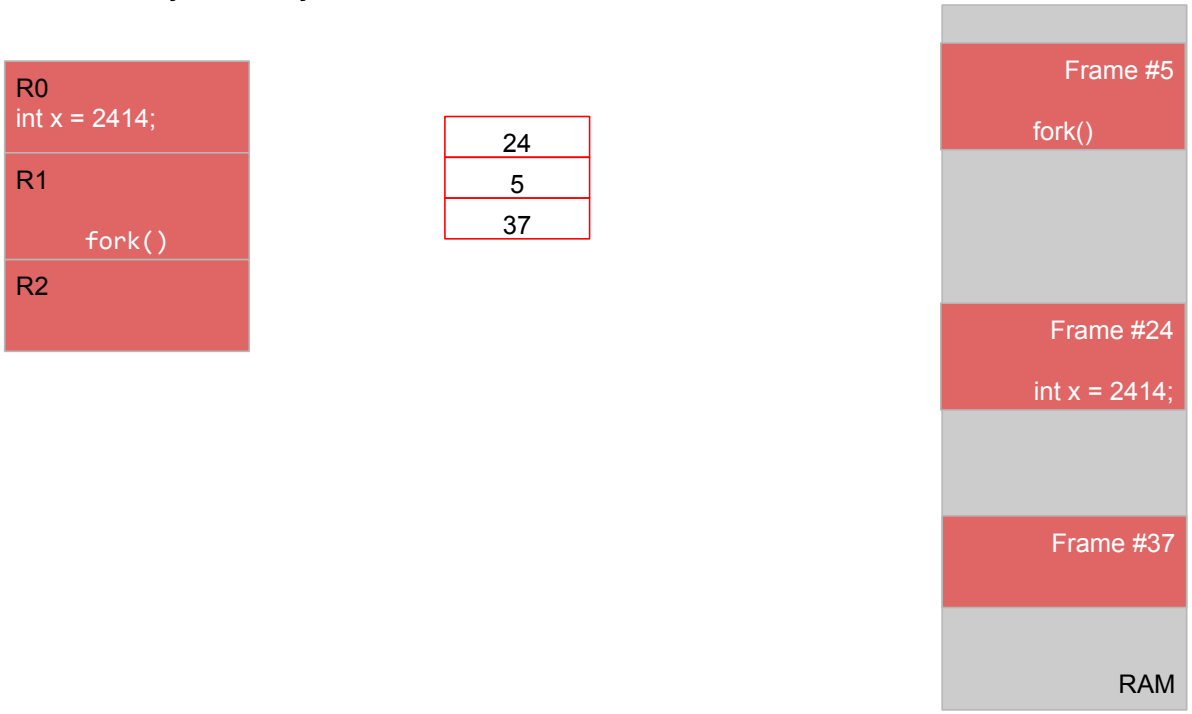
Case	TLB Access	RAM Access	Swap Disk Access
1	Read	1x	No
2	Read & Update	2x (or more*)	No
3	Read & Update	3x (or more*)	Yes

(*) with hierarchical paging

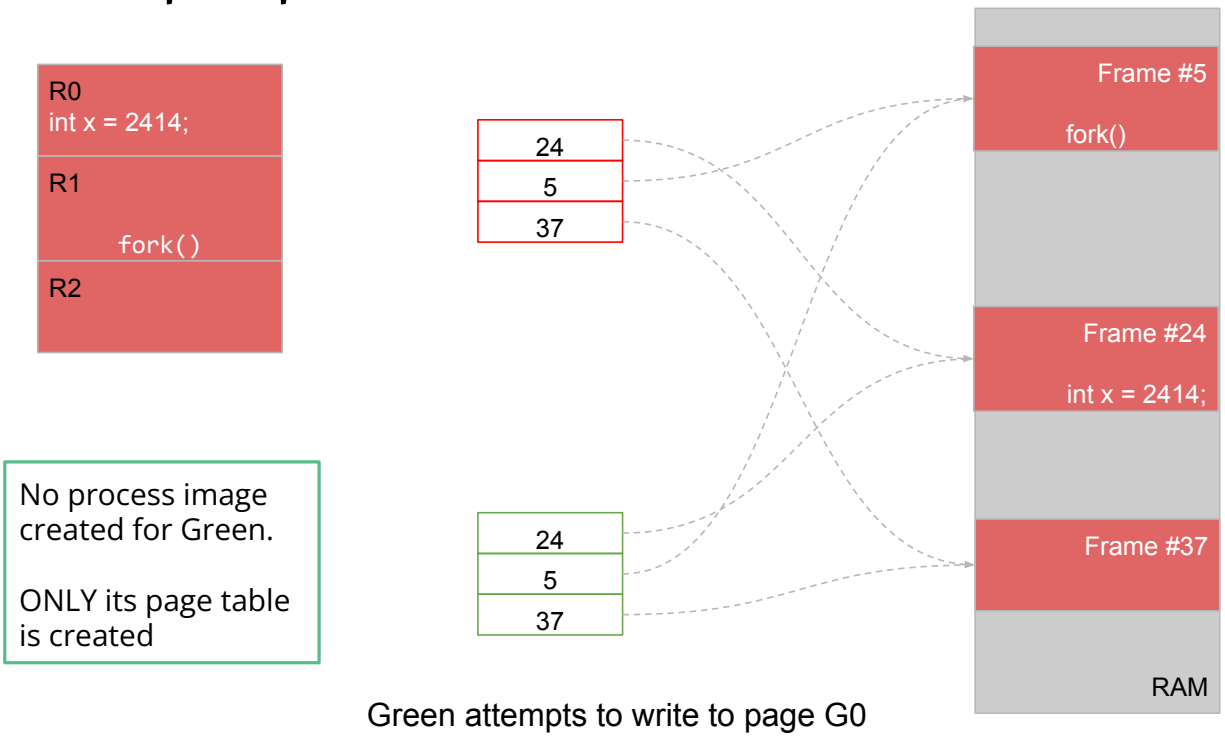
COW: Copy-On-Write

- Recall that fork() creates a new child process, whose process image is a twin image of the parent
- To speed up “spawning”, use “**lazy duplication**”. Let the child process share the parent process image
  - Create the actual duplicate when the child is accessing in R/W mode
  - Duplicate the entire process image?
  - Duplicate only the page being written by child?

COW: before fork()



COW: after fork()

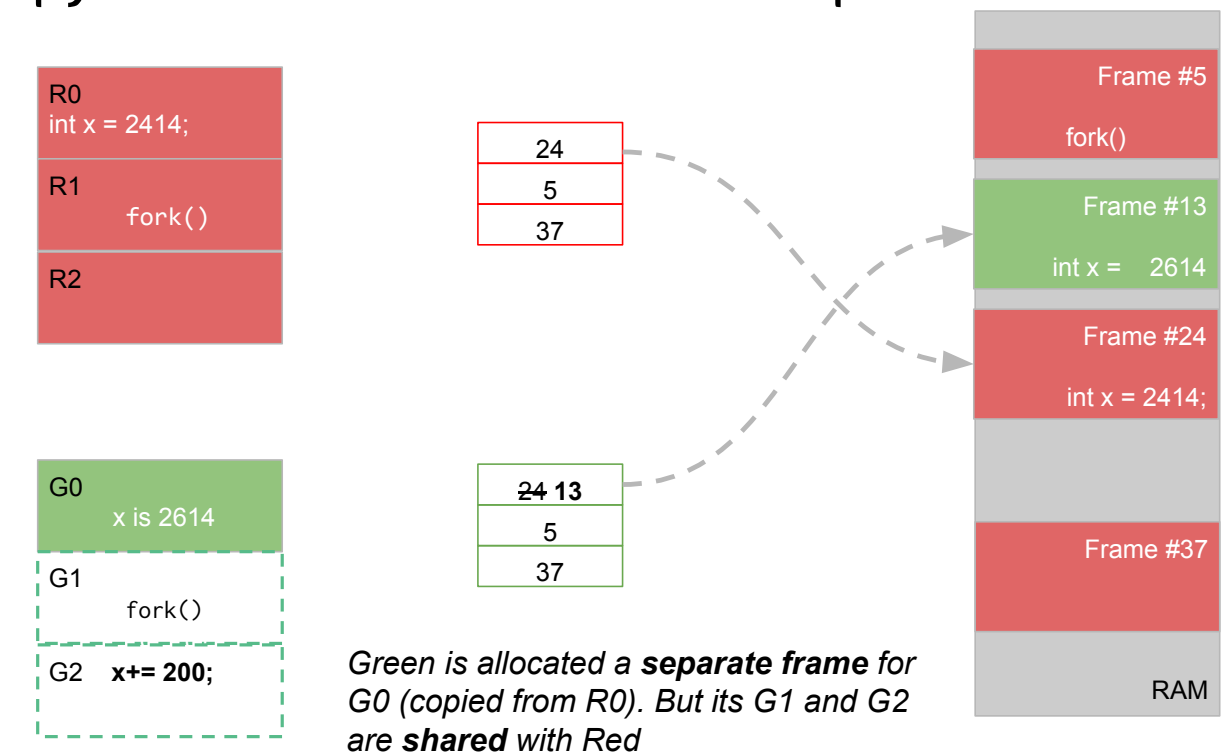


No process image created for Green.

ONLY its page table is created

Green attempts to write to page G0

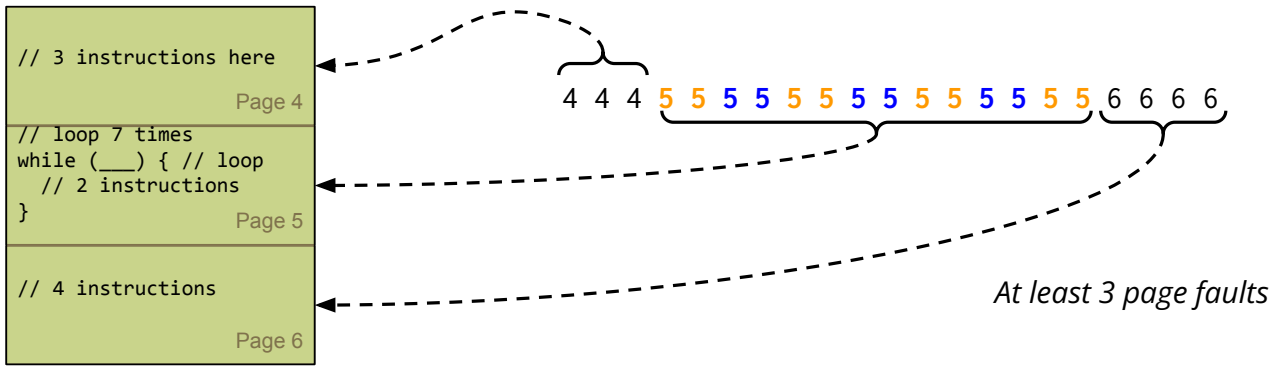
Copy-on-Write: Frame 24 is copied to Frame 13

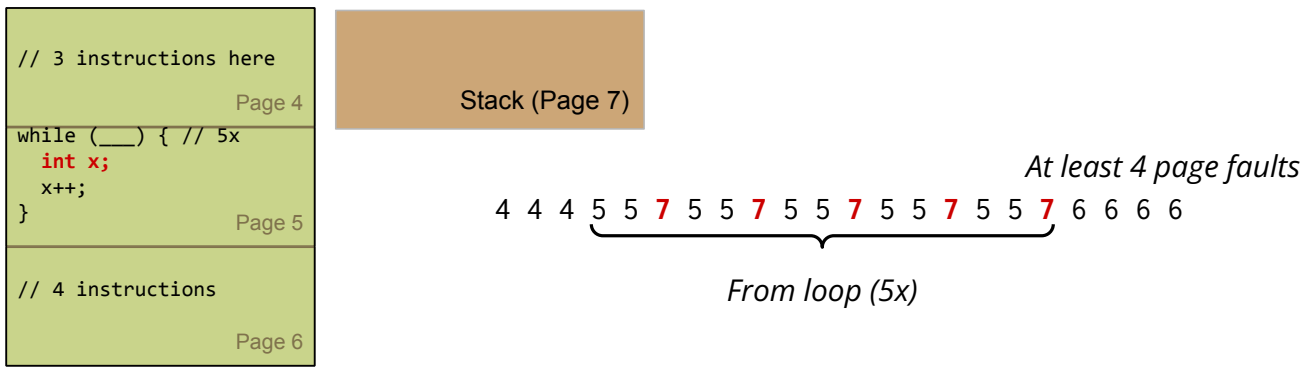


*Green is allocated a **separate frame** for G0 (copied from R0). But its G1 and G2 are **shared** with Red*

Memory Reference Pattern

Code With Loop and Local Variables





Reference Locality

- Memory access pattern of a program is NOT **random.** It follows some patterns of locality

```
Ref #1: 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3

Ref #2: 1 1 1 2 2 7 2 2 7 2 2 7 2 2 7 2 2 7 3 3 3 3
```

- **Spatial Locality** (*clustered in space*): *when a program accesses a particular memory location X, it may also access* ***other locations nearby X***

  (visiting other nearby stores)

- **Temporal Locality** (*clustered in time*): *when a program accesses a particular memory location X, it may* ***reference X again in the short future***

  (returning to a favorite store a week later)

Other examples of Locality

- Spatial Locality (*access other locations nearby X*)
  - Executing sequential instructions
  - Accessing several local variables in the **same** stack frame
  - Adjacent array elements
- Temporal Locality (*repeated access to the same location X*)
  - A loop that refers to the same set of global variables / heap variables
  - Multiple calls to same function

Locality: Space-Time Diagram

Spatial locality: nearby locations are referenced about the same time
Temporal locality: one location is referenced (again many tines) in the future