Chapter 8: Details of Page Based Memory Management

Demand Paging

The idea is to load parts of the program that are not in memory *on demand*

- Better memory utilization

Hardware For Demand Paging

- Add valid bit to each PTE
- *Page Fault Exception* happens when we request a page that is not in memory yet

Figure 8.1: Page Table Entry

- Means processor has to be capable of restarting an instruction whose execution has been suspended in the middle due to a page fault

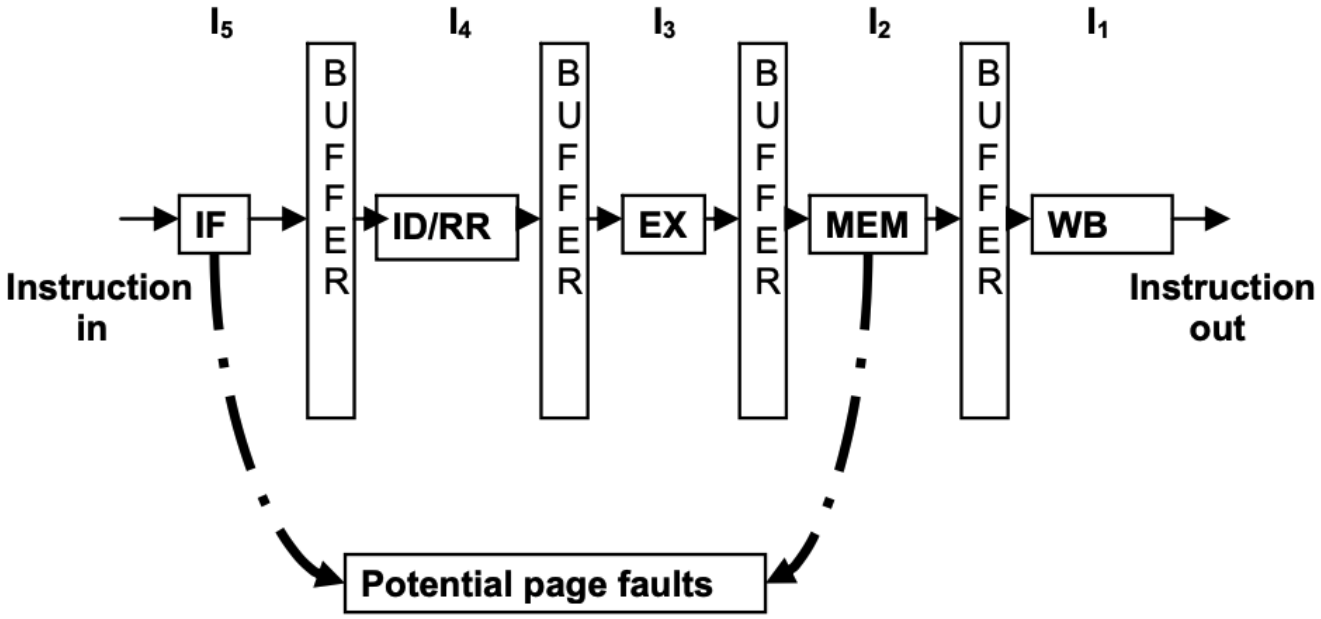


Figure 8.2: Potential Page Faults In Processor Pipeline

- Interesting thing to note is that the pipeine registers will contain the PC value of the instruction in the event there is an exception while executing the instruction

Page Fault Handler

1. Find a free page frame
2. Load the faulting virtual page from the disk into the free page frame
3. Update the page table for the faulting process
4. Place the PCB of the process back in the ready queue of the scheduler

Data Structures For Demand-Paged Memory Management

1. Free-list of Page Frames: Contains information about the currently unused page frames. Each node of the free list only contains the PFN

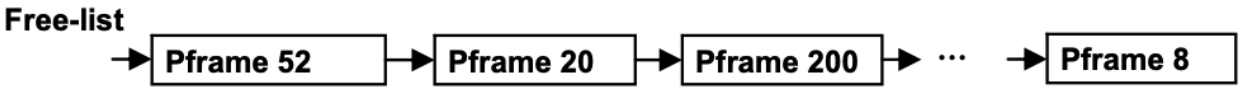


Figure 8.3: Free-list of Page Frames

2. Frame Table (FT): Contains the reversee mapping. Given a frame number, it gives the Process ID (PID) and the virtul page number that currently occupies the page frame

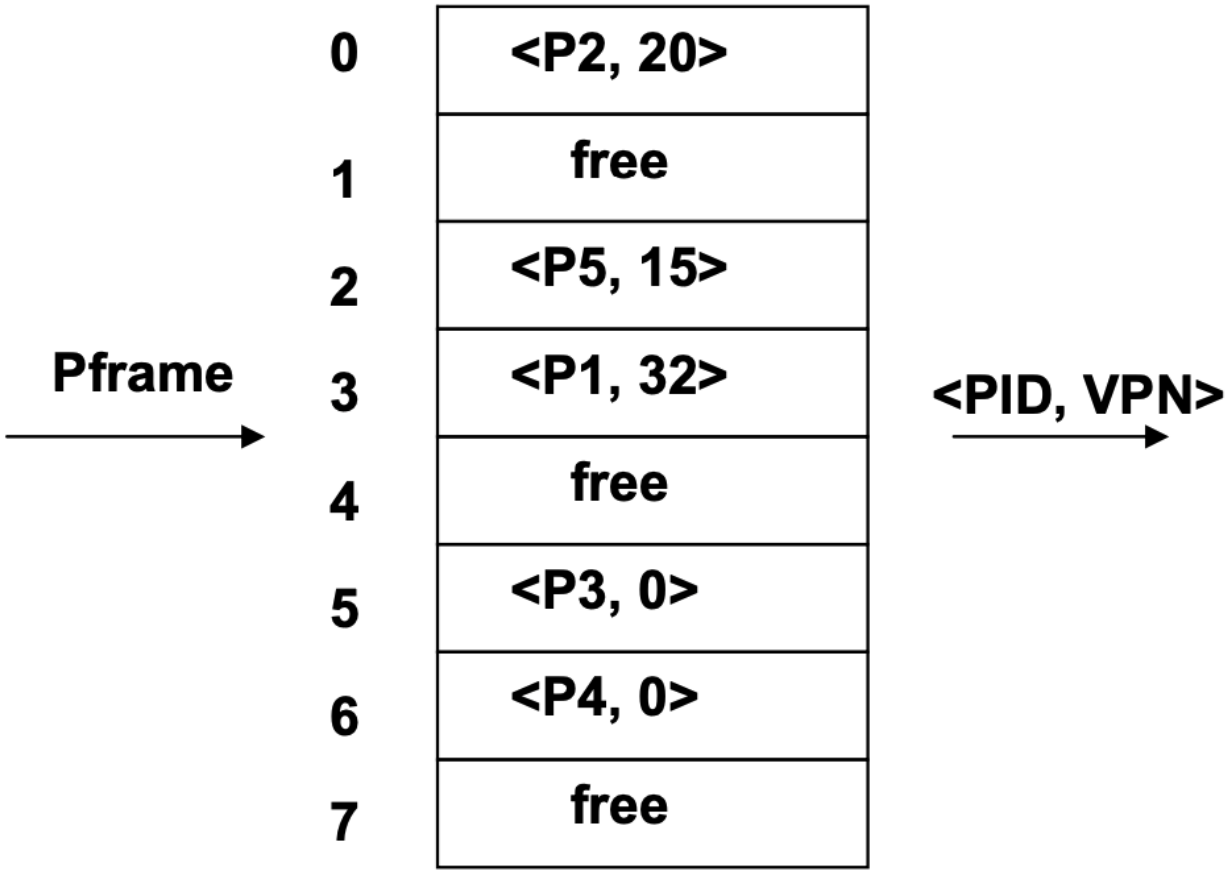


Figure 8.4: Frame Table

3. Disk Map (DM): DM maps the process virtual space to location on the physical disk that contain the contents of the pages. This is the disk version of the page table

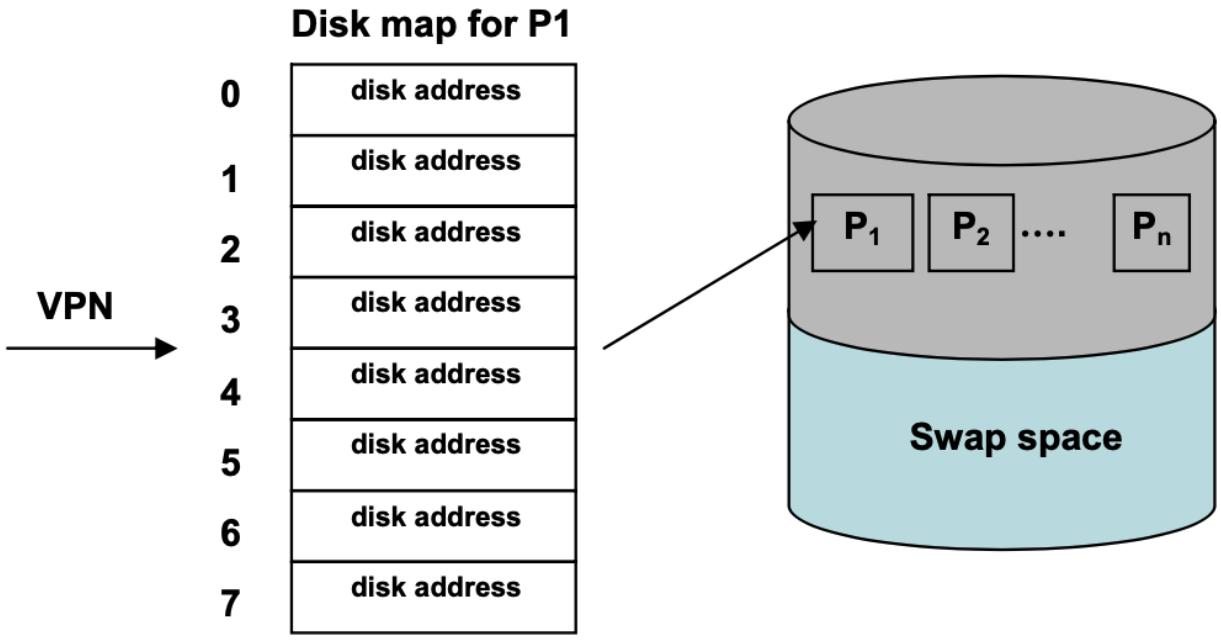


Figure 8.5: Disk Map

Anatomy of a Page Fault

Find a free frame

- Handler (part of the memory manager) looks up free-list
- If list is empty (all physical frames are in use), we select a *victim* physical frame to make room for the faulting page

Pick the victim page

- Manager determines the victim process that currently owns it using the *frame table*
- ***Clean Page*** - page **HAS NOT** been modified by the program since the time it was brought into memory from the disk
- ***Dirty Page*** - page **HAS** been modified by the program since the time it was brought into memory from the disk
    - Need to flush (write page back to disk)
    - Use *disk map* to locate the victim process

Load the Faulting Page

- Use the *disk map* for the faulting process and read in the page from the disk into the selected page frame

Restart Faulting Process

- Place PCB back onto the ready queue

Interaction Between Process Scheduler and Memory Manager

1. Hardware Timer interrupts CPU, resulting in upcall (function call from the lower levels of system software to higher levels)
2. Process incurs a page fault resulting in an upcall to the memory manager that results in a page fault handling
3. Process makes a system call (e.g. I/O operation) resulting in another subsystem getting an upcall to take the necessary action

All 3 of these have the same PCB structure

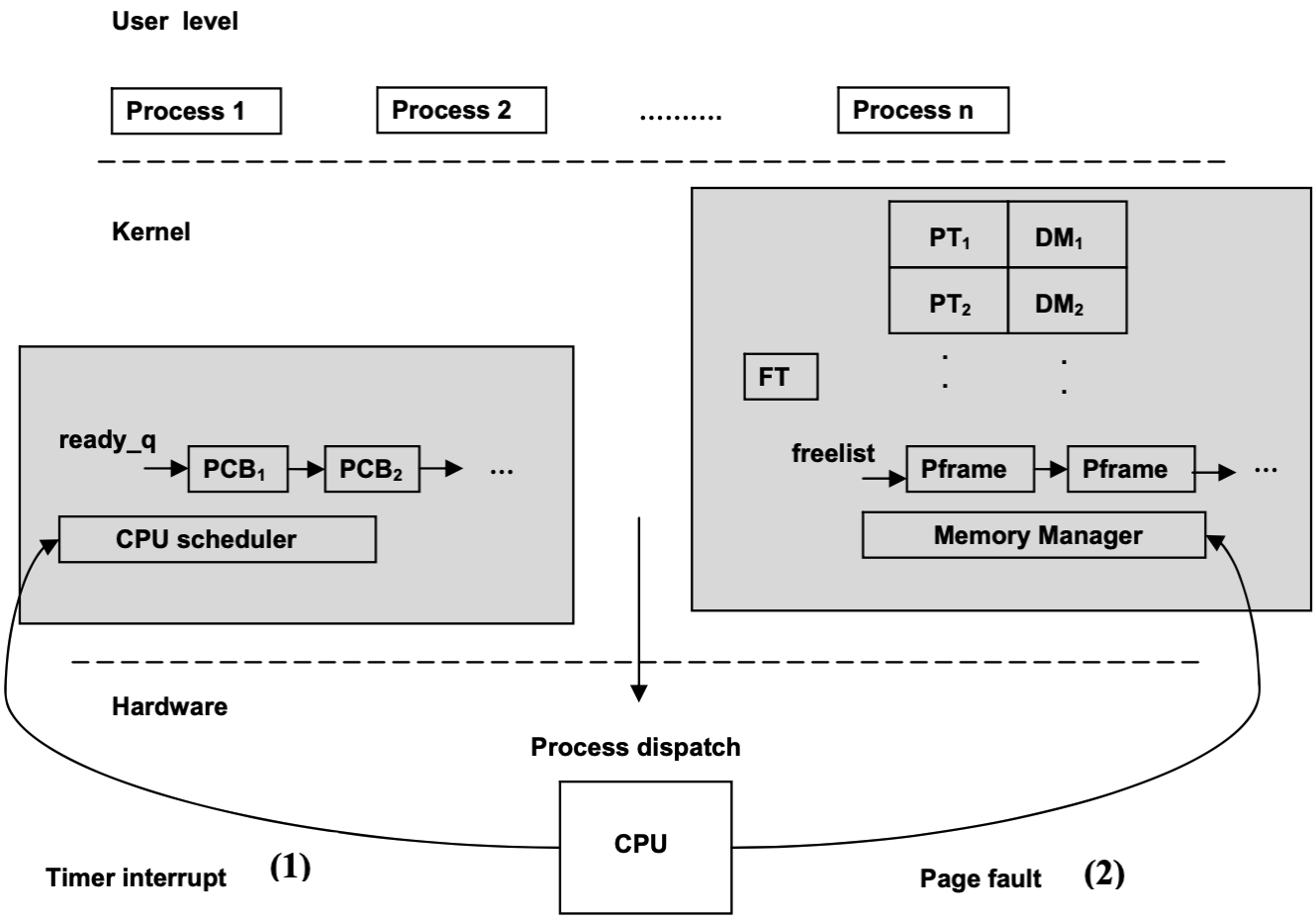


Figure 8.6: Interaction Between Scheduler and MMU

Page Replacement Policies

- Local victim selection: steal a physical frame from the faulting process itself
  - Eliminates need for frame table
  - Poor memory utilization
- Global victim selection: steal a physical frame from any process
  - Good memory utilization
  - Requires frame table

Belady's Min

Optimal Replacement Algorithm: replace the frame that is not referenced for the longest time in the future

Random Replacement

- No hardware support or book-keeping

- Serves as a lower bound for performance

First In First Out

- Affix a timestamp when a page is brought in to memory
- If a page has to be replaced, choose longest resident page as the victim
- No hardware assist needed
- Memory manager simulates the "timestamp" using a queue for recording the order of the arrival of pages into physical memory
    - In a circular queue, the longest resident page is just at the head
    - Serves the purpose of the free list and frame table simultaneously

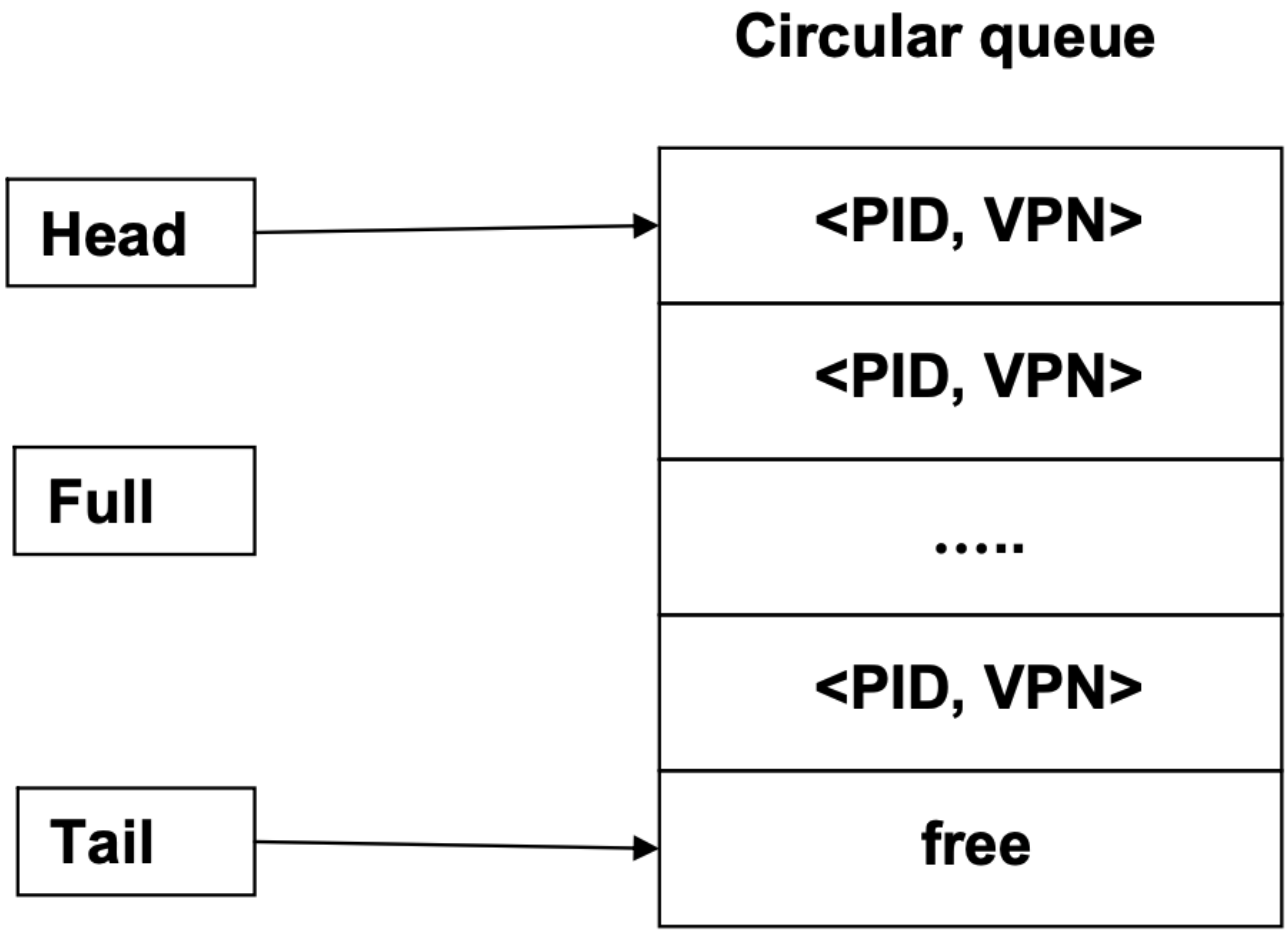


Figure 8.7: Circular Queue for FIFO Page Replacement

Least Recently Used (LRU)

- Choose the page that has not been used for the longest time
- Uses a push-down stack

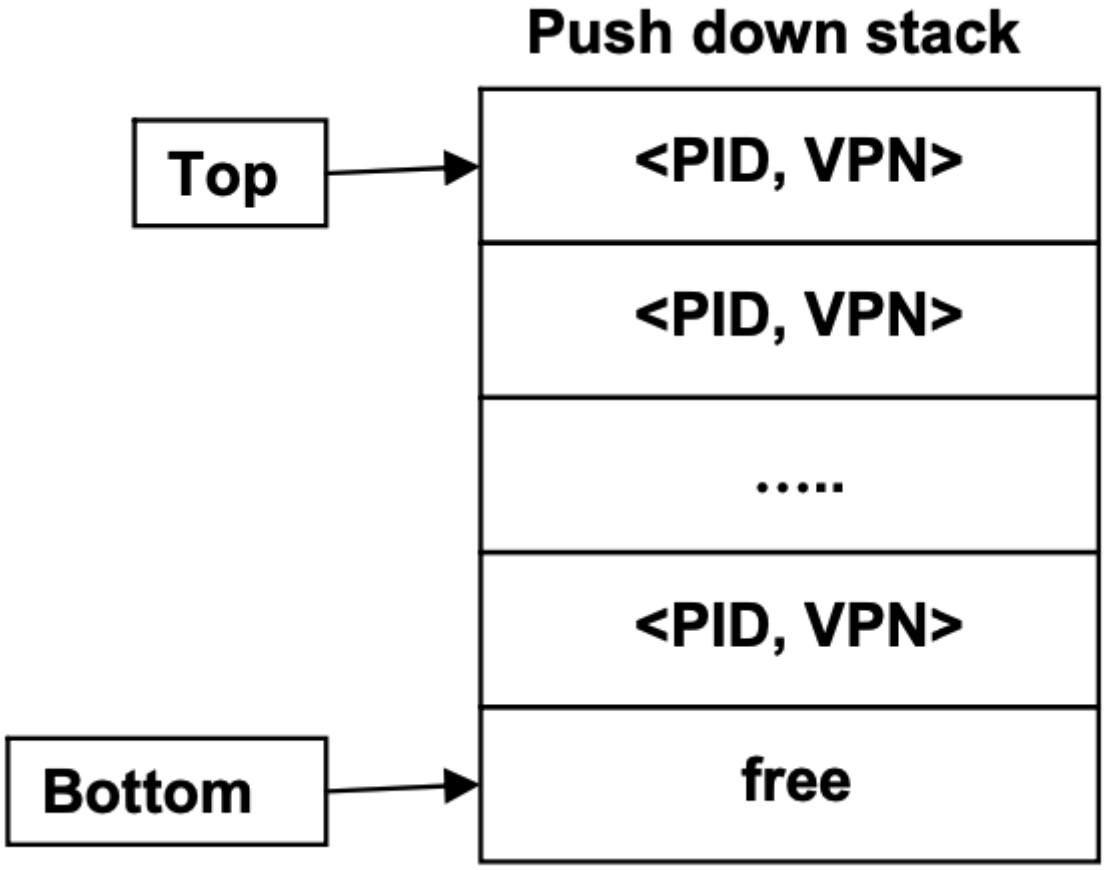


Figure 8.8: Push Down Stack for LRU

- On every access, push process onto top of stack
- Bottom of the stack is the least recently used page

- Stack needs to have as many entries as physical frames
- On every access, hardware has to modify the stack

- Keep a small stack (e.g. 16) that maintains the last n referenced pages
- Algorithm protects the n most recent pages from replacement in the stack

Approximating LRU: Reference Bit

- Memory Manager maintains a *bit vector* per page frame called a reference counter
- Periodically read the reference bits of all the page frames and dumps them in the *most significant bit* of the corresponding per frame *reference counters*. After reading the bits, the memory manager clears them
- Repeats this every time quantum with a **paging daemon**

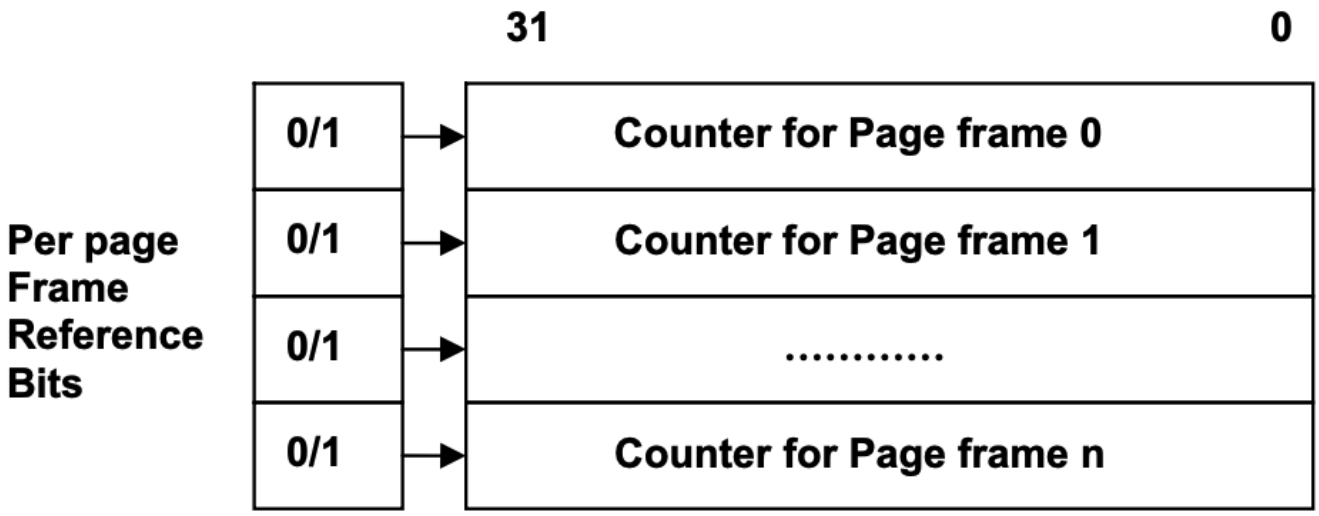
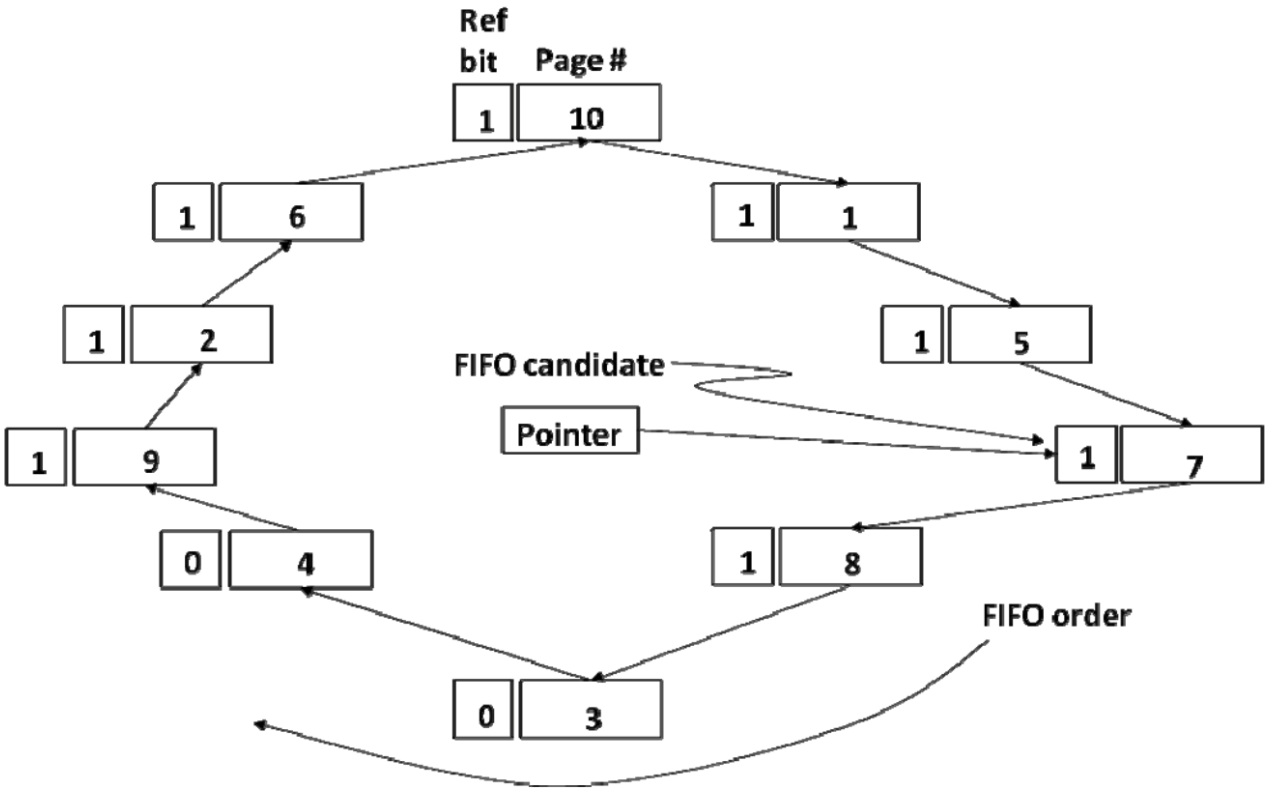


Figure 8.9: Page Frame Reference Counters

- Reference counter with the **largest absolute value** is the most recently referenced
- Reference counter with the **smallest absolute value** is the least recently referenced

Second Chance Replacement Algorithm

- Hardware sets reference bits for pages referenced by program
- If a page has to be replaced, memory manager chooses replacement candidate in FIFO
- If the chosen replacement candidate has a refernece bit set, then the manager clears the reference bit, gives it a new arrival time, and repeats step 1 (moves this to the end of the FIFO queue)
- Victim is the first candidate in FIFO order whose reference bit is not set
- In the case that all reference bits are set, the algorithm degenerates to a simple FIFO



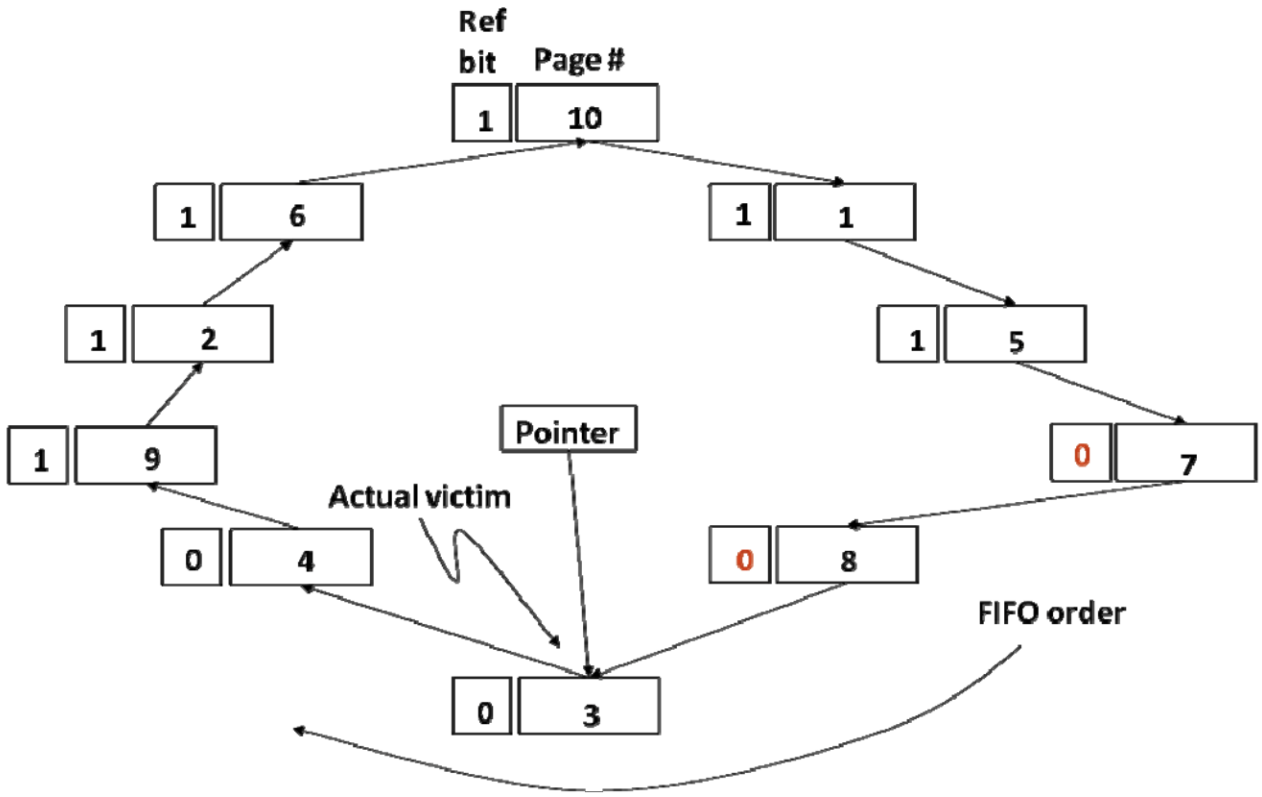


Figure 8.10: Second Chance Replacement

Optimizing Memory Management

Pool of Free Page Frames

- Don't wait until all pages are gone
- Keep a pool of free pages ready
- Paging Daemon wakes up periodically

Overlapping I/O With Processing

- If the victim is dirty, then we need to flush to the disk
- Instead of doing it right away, we can schedule a write I/O for a dirty victim page frame before adding it to the free-list
    - Before the memory manager uses the dirty victim page, the write I/O must be complete
    - Memory manager may skip over a dirty apge on the free-list to satifsfy a page fault
- Delay the necessity to wait on a write I/O at the time of a page fault

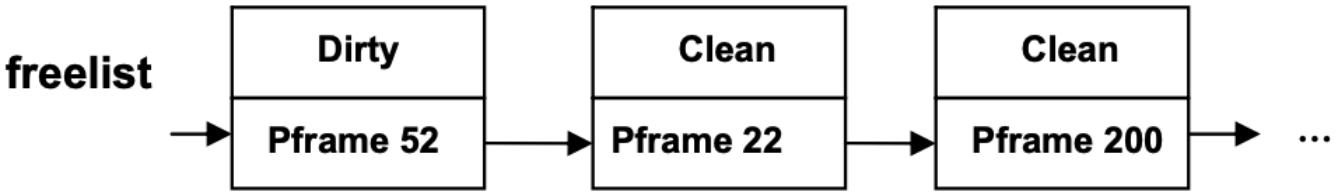


Figure 8.11: Free List – upon a page fault, manager may choose pframe 22 over pframe 52 since the former is clean

Reverse Mapping to Page Tables

If the memory manager has not yet reassigned the page frame to a different process, then we can retrieve it from the *free-list* and give it to the faulting process

- Augment free-list with reverse mapping (like frame table), showing the virtual page it housed last

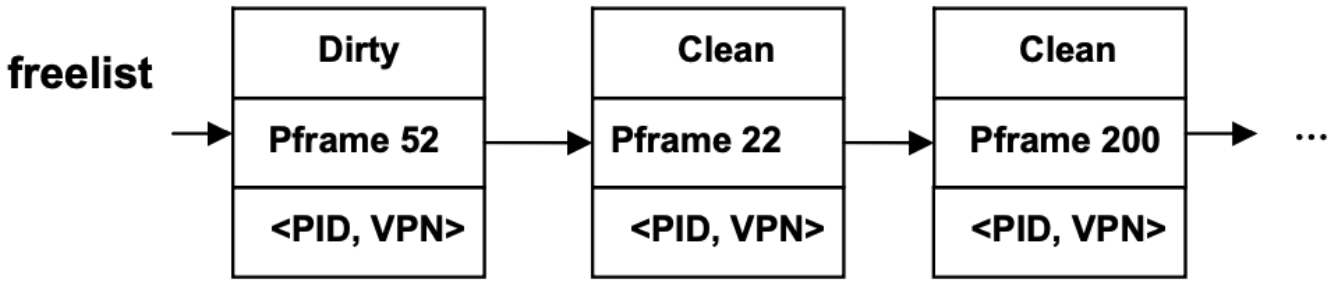


Figure 8.12: Reverse Mapping

Thrashing

Thrashing - system is not getting useful work done and is spending too much time switching contexts

- Principle of locality: at a reasonable sized window of time, it will only access a small portion of its entire memory footprint

Working Set

Working set is the set of pages that defines the locus of acitivty of a program

- ***working set size*** = number of distinct pages touched by a process in a window of time

Total Memory Pressure = $\sum_{i=1}^{i=n} WSS_i$

Controlling Thrashing

Water Marking

- Memory manager sets a low water mark and a high water mark for page faults
- Observed page fault rate that exceeds high water ⇒ excessive paging and reduce multiprogramming

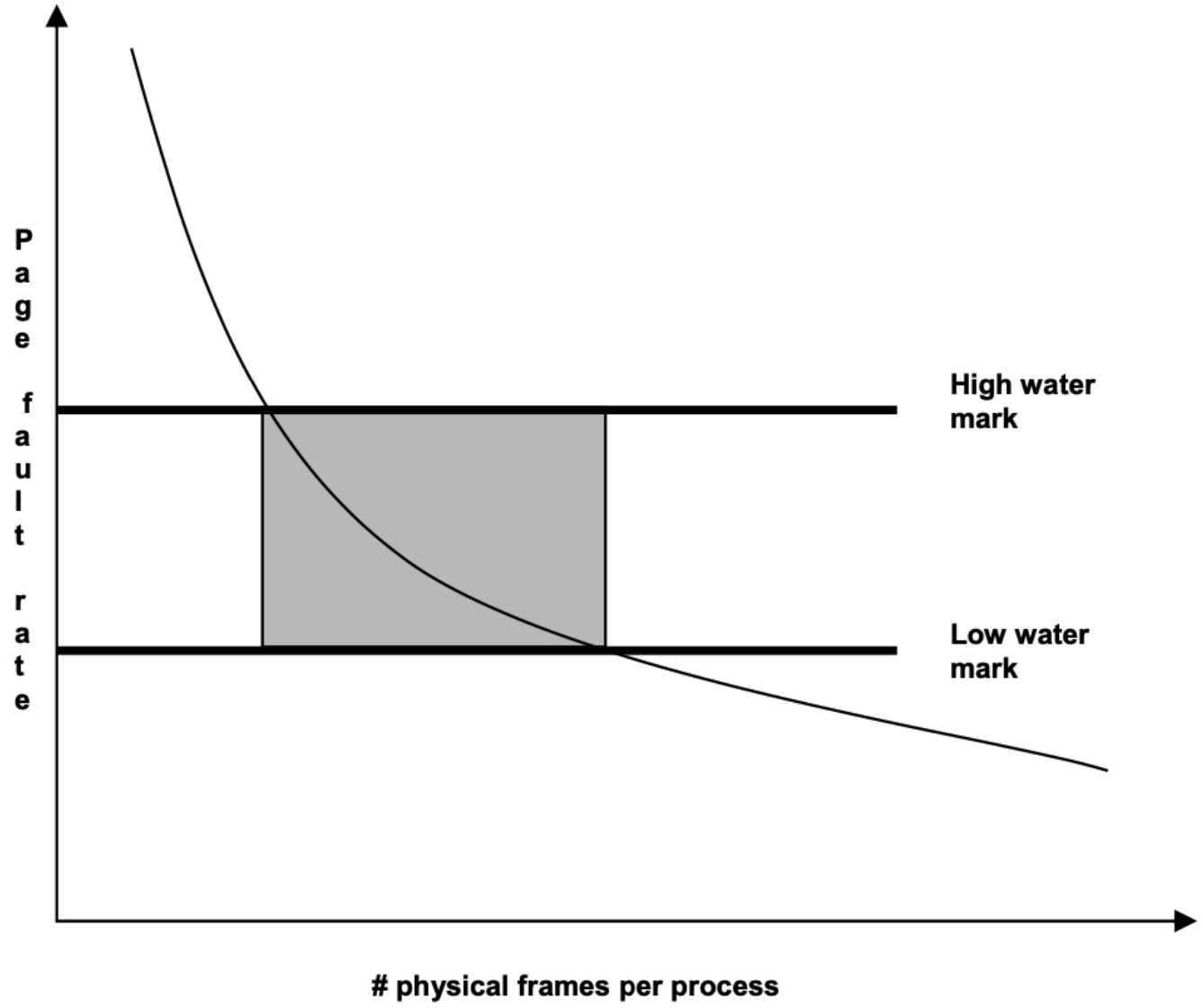


Figure 8.13: Page Fault Rate

Translation Lookaside Buffer (TLB)

- Keep a small page table called TLB
- Separated into user/kernel parts

USER/KERNEL	VPN	PFN	VALID/INVALID
U	0	122	V
U	XX	XX	I
U	10	152	V
U	11	170	V
K	0	10	V
K	1	11	V
K	3	15	V
K	XX	XX	I

Figure 8.14: Translation Look-aside Buffer