Paging and Page Replacement Algorithms

Section 3.4 Tanenbaum's book

Kartik Gopalan

OS Involvement with Page Table Management

Four times when OS deals with page-tables

1. Process creation
   - create page table

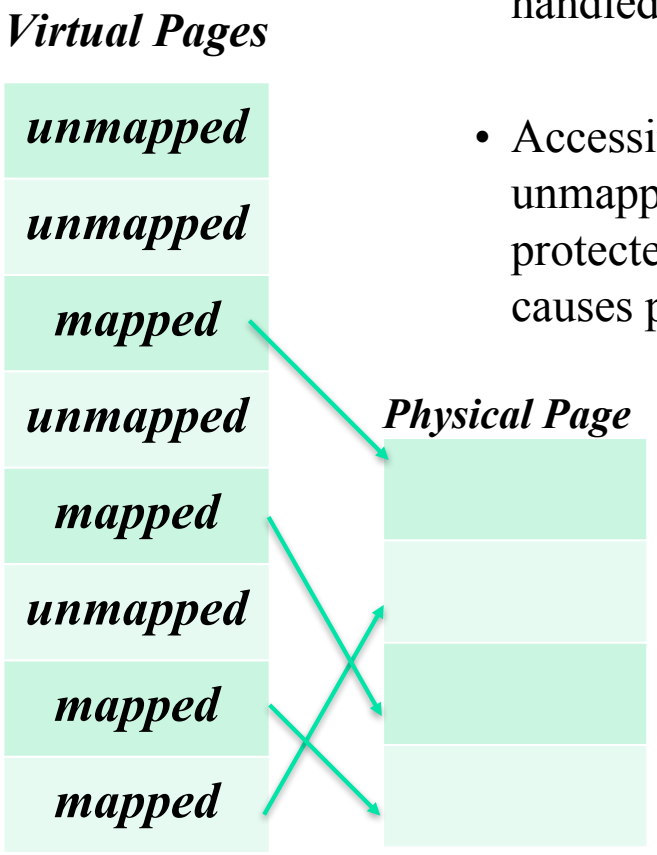
2. Upon context switch
   - Load MMU context for a new process (e.g. load CR3 register with base address of page table)
   - TLB may be flushed

3. Page fault time
   - determine the virtual address causing fault (read CR2 register for faulting address)
   - swap target page out, bring needed page in

4. Process termination time
   - release page table and other pages

Page Fault

- Page Fault is a hardware exception
- It occurs when a process accesses a virtual page
  1. that is currently not resident in physical memory OR
  2. for which it doesn't have the correct access permissions OR
  4. which hasn't yet been allocated by the operating system



- Translation for mapped pages are handled by MMU
- Accessing unmapped or protected pages causes page fault

Page-fault handling

1. Process accesses unmapped/protected virtual address
2. MMU hardware raises trap
    - ❑ Because it cannot find a valid page-table entry
3. Process is paused
    - ❑ Program counter and Registers saved
4. OS runs page-fault handler (PFH)
    - ❑ PFH examines the virtual address which caused the fault
    - ❑ Is this a write to an unallocated page?
        - ❑ PFH allocates a new physical page
    - ❑ Is this an already allocated page?
        - ❑ PFH reads the page from the swap device (page-in)
        - ❑ PFH updates the page-table entry.
    - ❑ Is this a disallowed access? E.g. writing to a read-only page?
        - ❑ PFH terminates the process
5. Process resumes from where it left off (unless terminated by PFH).
    - ❑ Program counter and Registers restored

❑ OS may periodically page-out victim page to disk to create space

Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Problem
  - Process P1 issues call for read from device into buffer
  - While waiting for I/O, another processes P2 starts up
  - P2 has a page fault
  - Buffer for the P1 may be chosen as victim to be paged out, resulting in a DMA error.
- Need to specify some pages as “locked” or “pinned” in memory
  - These pages are exempted from being victim pages

Page Replacement

- Resident pages must be occasionally evicted from memory to make room for new pages
- Pages that have not been modified can be simply discarded i.e. overwritten with new content
- Modified pages must be saved before reuse
  - unmodified pages are simply overwritten
- Ideally, we'd like to evict pages that are least needed.
- Goal of page replacement algorithm
  - minimize the number of page faults
  - E.g. if we evict an often used page, we may need to bring back in soon resulting in more page faults.

Page Replacement Algorithms

- Optimal Page Replacement
- Not Recently Used (NRU)
- FIFO
- Second Chance
- Clock
- Least Recently Used (LRU)
- Not Frequently Used (NFU)
- Aging
- Working Set
- WSClock

Optimal Page Replacement Algorithm (OPR)

- Replace the page that's NOT needed for the longest time in the future

Page access timeline

1	2	3	4	1	5	2	4

Assume we have 3 frames in physical memory

Eviction Decisions	t0	t1	t2	t3	t4	t5	t6	t7
Frame 0	1	1	1	1	1	5	5	5
Frame 1		2	2	2	2	2	2	2
Frame 2			3	4	4	4	4	4

Optimal Page Replacement Algorithm (OPR)

- Why is it optimal?
  - Here, "Optimal" means that no other algorithm can give fewer page faults than OPR.
  - OPR is optimal but unrealizable
    - Because we cannot predict all future memory accesses
  - But OPR can serve as a useful baseline to measure the effectiveness of other algorithms

- How do we know its optimal?
  - Try proof by contradiction
  - What if you can find another algorithm which does not pick the page needed farthest point in the future and can still yield fewer page faults than OPT?

- Other algorithms try to approximate OPR
  - Estimate the pages not needed for a long time by recording the sequence of pages used in the past.

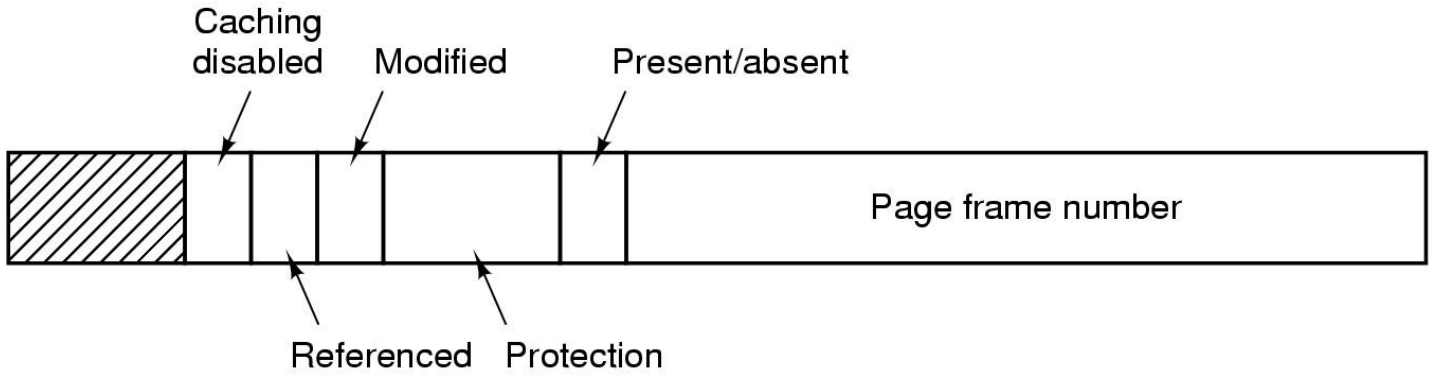
Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - throw out page that has not been used for longest time in the past

- Ideally: Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !! Not practical.

- Aging: Alternatively keep counter in each page table entry
  - Counter tracks the number of references to a page
  - Choose page with lowest value counter
  - Periodically zero the counter
  - Note: This only approximates LRU
  - Implementation details in Tanenbaum Section 3.4.7

(Recall) Page Table Entry (PTE)



- Referenced bit: Whether the page was accessed since last time the bit was reset.
- Modified bit: Also called "Dirty" bit. Whether the page was written to, since the last time the bit was reset.
- Present/Absent bit: Whether the PTE contains a valid page frame #. Used for marking swapped/unallocated pages.

Not Recently Used Page Replacement Algorithm

- NRU periodically scans all PTEs

- After each scan, pages are classified as
  1. not referenced, not modified (0,0)
  2. not referenced, modified (0,1)
  3. referenced, not modified (1,0)
  4. referenced, modified (1,1)

- When selecting a victim page
  - remove page from the lowest numbered non-empty class
  - Also reset all referenced bits
    - Implies that pages are moved from 3➔1 and from 4➔2

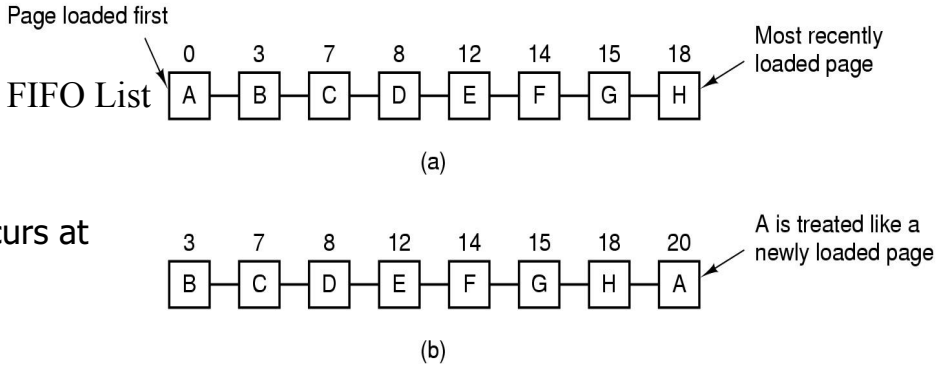
- Before the next NRU scan, some pages may be referenced or modified again
  - Implies some pages may move back from 1–>3, 1–>4, 2–>4, 3–>4

- Advantage: Simple to implement
- Disadvantage: Doesn't distinguish among pages within the same class

FIFO Page Replacement Algorithm

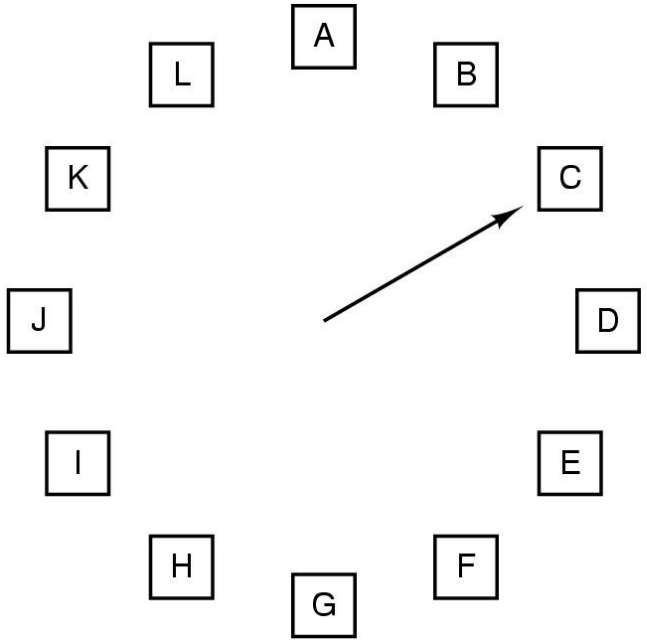
- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
  - page in memory for the longest time may be often used

Second Chance Page Replacement Algorithm



- Pages are sorted in FIFO order
- Basic idea: Select the oldest page that has not been recently used.
- How it works: When selecting a victim page:
  1. Look at the oldest page in the list.
  2. If R bit of the page is set, then set R=0 and move the page to the front of the FIFO list; go back to step 1
  3. Select the oldest page with R=0 as the victim page.
  4. If no page with R=0 (unlikely), then select the oldest page (FIFO)

The Clock Page Replacement Algorithm



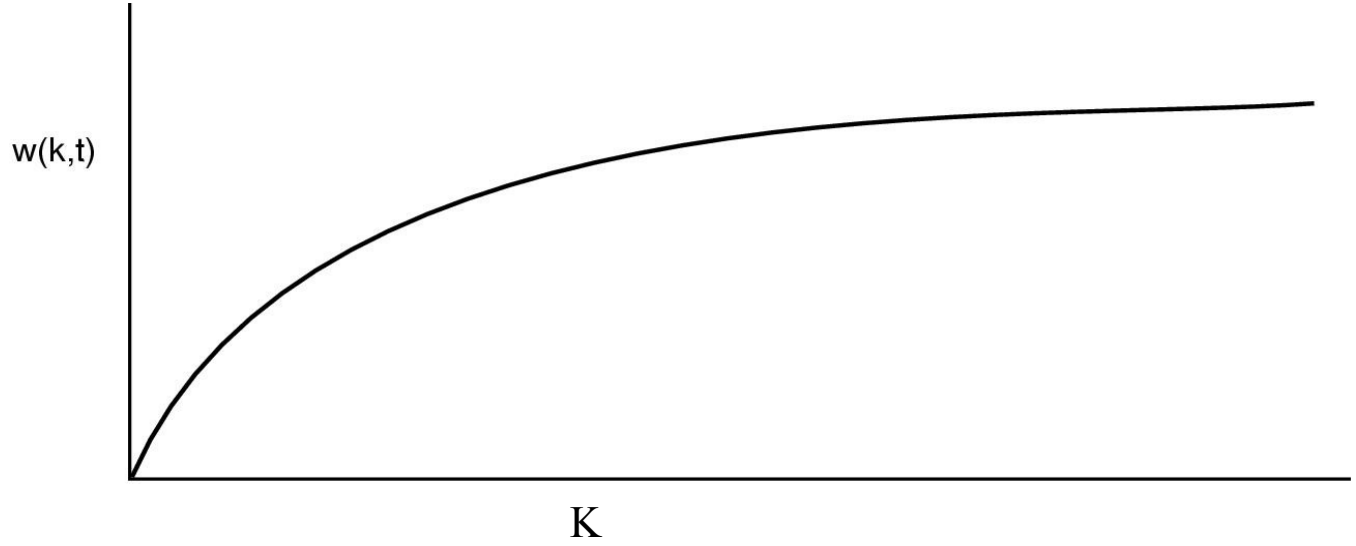
When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page; Put new page in its position

R = 1: Clear R and advance hand

Logically the same as Second Chance, except that it doesn't keep moving pages around the linked list. Hence it is more memory-efficient.

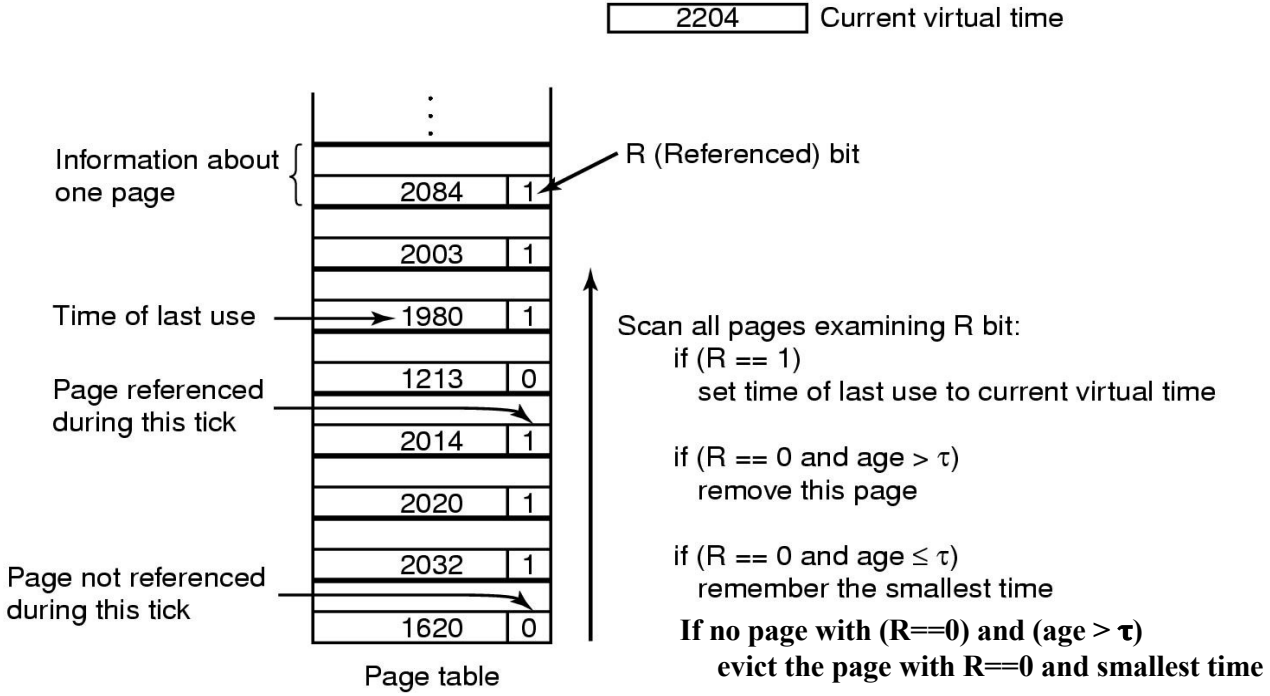
The Working Set Page Replacement Algorithm (1)



- Working Set: The set of all pages that a process is currently using.
- In theory, if the entire working set is in memory, then there'll be no page faults.
- In practice, predicting the exact working set is impossible.
- Assume, the working set is the set of pages used by the K most recent memory references
- For typical processes w(K,t) — the size of the working set at time t — becomes constant over time

The Working Set Page Replacement Algorithm (2)

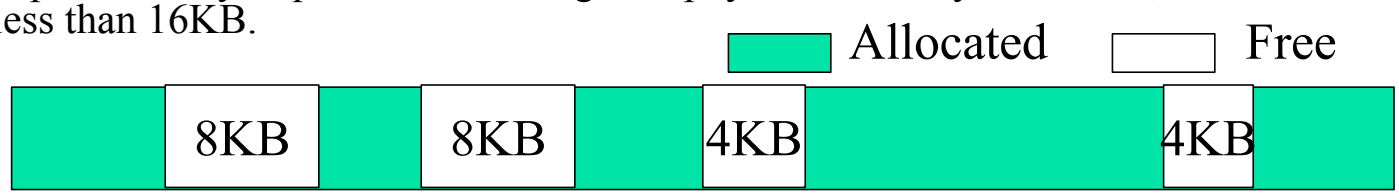
- Keep track of the working set of a process
- When its time to evict a page, choose one which is not in the working set of the process.



Internal versus External Fragmentation

- Internal Fragmentation
  - Occurs when part of memory allocated to a process remains unused.
  - E.g. a process may be allocated a 4KB page, but it uses only 1-byte in the page.

- External fragmentation
  - Occurs when none of the available free memory fragments is big enough to satisfy a new memory allocation request.
  - E.g. a process may request 16KB contiguous physical memory allocation, but all available free memory is of size less than 16KB.



- Can a virtual memory paging system have
  - Internal fragmentation? Yes
  - External fragmentation? No. Why?

Periodic Eviction Policy

- Need for a background process, paging daemon
  - periodically checks memory pressure and evicts pages from memory

- When to evict?
  - Based on Low and High “watermarks”.
    - Two thresholds of memory usage

- When memory usage is above “High” watermark, say 90%, indicating high memory pressure
  - Start evicting pages till memory usage falls below “Low” watermark, say 80%

- Why two thresholds?
  - To enable hysteresis
  - So that the system state doesn’t fluctuate rapidly around a single threshold.

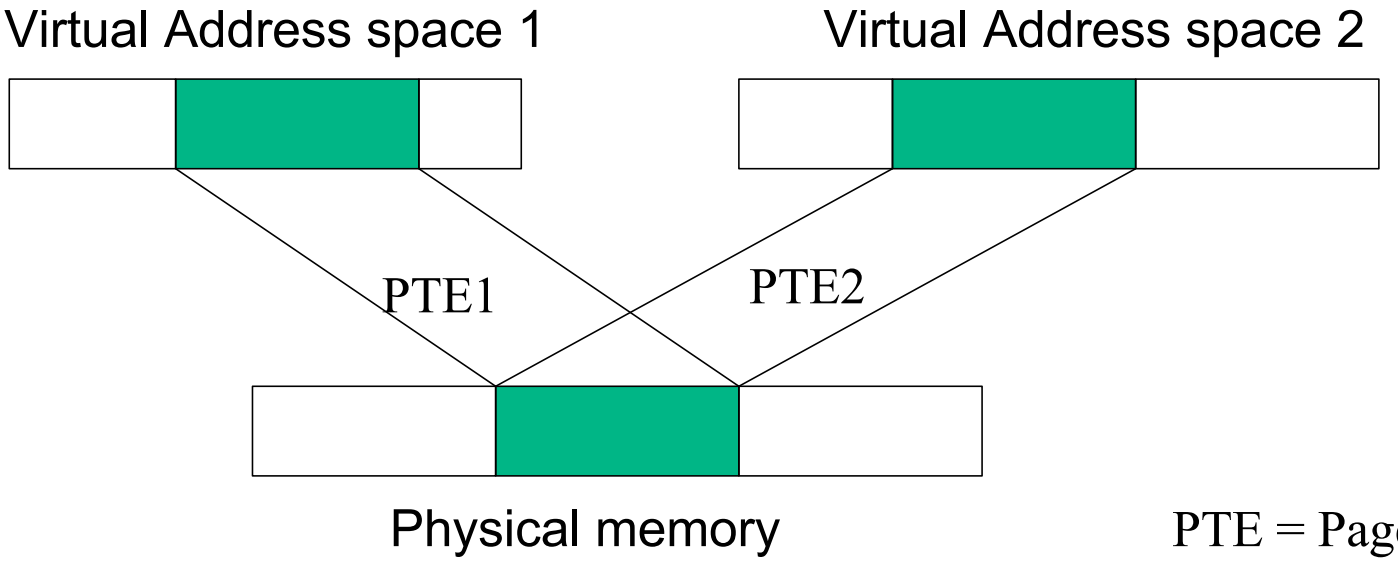
Thrashing

- Despite good designs, system may still thrash when
  - some processes need more memory
  - but no processes need less

- Thrashing
  - When the working set of all processes does not fit in the main memory.
  - Hence, paging daemon has to constantly page-out pages to disk and bring them back in almost immediately.

- Solution : Reduce the degree of multi-programming ➔ number of processes competing for memory
  - swap some less important processes to disk
  - And free up all the pages they hold

Shared Memory between multiple processes



PTE = Page Table Entry

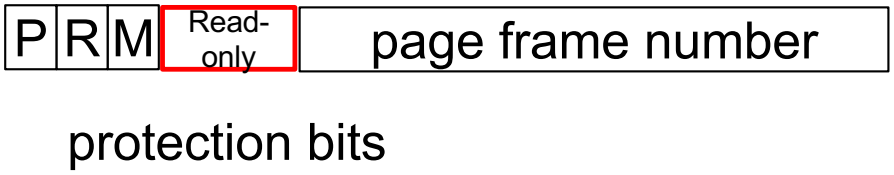
PTEs for virtual pages in different processes map to common physical page frames in DRAM.

Copy-on-write

- Right after fork()
  - Child process has identical memory content as parent.
  - But copying is inefficient, especially if the child calls exec

- Copy-on-write
  - Delays copying memory till either child or parent actually write to a page.

- Pages are shared as "read-only" (RO) between the parent and child
  - Writes cause hardware to trap to OS
  - OS intercepts write faults, copies page, and marks them read/write (RW)



Memory-mapped Files

- Normally, files are accessed with system calls
  - Open, read, write, close
- Memory mapping (using mmap) allows a program to access a file with load/store operations