# R

بسم الله الرحمن الرحیم

# سیستم عامل

جلسه هجدهم – الگوریتم‌های جایگزینی صفحه (۳)

# جلسه‌ی گذشته

# FIFO Algorithm

- Replace the page that was first brought into memory
- Example: Memory system with 4 frames:

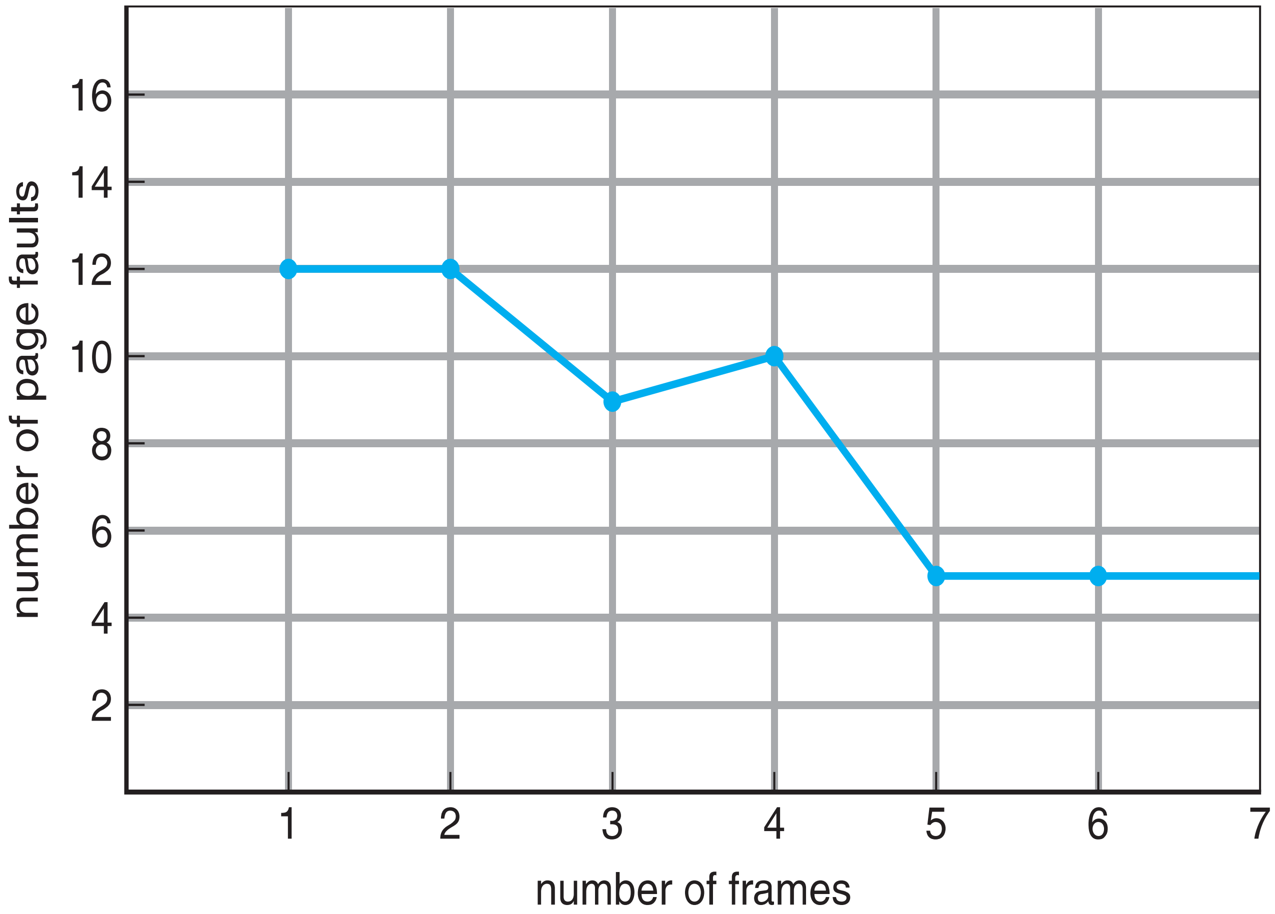
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Time | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Requests | | | c | a | d | b | e | b | a | b | c | a |
| Page Frames | 0 | a | | a | a | a | a | a | a | a | c | c |
| | 1 | b | | | | b | b | b | b | b | b | b |
| | 2 | c | c | c | c | c | e | e | e | e | e | e |
| | 3 | d | | | d | d | d | d | d | d | d | a |
| Page faults | | | | | | | x | | | | x | x |

# FIFO Algorithm

- Always replace the oldest page.
  - *Replace the page that has been in memory for the longest time*
- Implementation
  - *Maintain a linked list of all pages in memory*
  - *Keep it in order of when they came into memory*
  - *The page at the tail of the list is oldest*
  - *Add new page to head of list*

# Belady's Anomaly

- Adding more frames can cause more page faults!



# آزمونک قبل

**پرسش ۱ :** فرض کنید ~~صفحه‌ای~~ آدرس به شماره‌ی 7F35 (آدرس در مبنای ۱۶)، در حافظه‌ی مجازی داریم. فرض کنید از یک صفحه‌بندی یک مرحله‌ای استفاده کردیم و اندازه‌ی صفحات ۲ کیلوبایت هستند. (۸ نمره)

**الف.** Page number و offset را برای این آدرس مجازی مشخص کنید. (۱ نمره)

| P# | PF# | P/A present / Absent |
|---|---|---|
| 0 | 4 | 0 |

**ب.** جدول روبرو، Page table ما است که ستون اول آن، page number، ستون دوم Page Frame Number و ستون سوم آن valid bit است. براساس آن، شماره‌ی آدرس فیزیکی متناظر با آدرس مجازی 7F35 را محاسبه کنید. (۲ نمره)

| P# | PF# | P/A present / Absent |
|---|---|---|
| 0 | 4 | 0 |
| 1 | 3 | 1 |
| 2 | 2 | 0 |
| 12 | 14 | 1 |
| 13 | 20 | 1 |
| 14 | 11 | 1 |
| 15 | 18 | 1 |

**پ.** براساس این جدول صفحه، حافظه‌ی مجازی چند بیتی است؟

(۱ نمره)

| P# | PF# | P/A present / Absent |
|---|---|---|
| 0 | 4 | 0 |
| 1 | 3 | 1 |
| 2 | 2 | 0 |
| 30 | 2 | 0 |
| 31 | 4 | 0 |

**ت.** آدرس یا آدرس‌های مجازی متناظر با آدرس فیزیکی 1A0E را بیابید.
(۲ نمره)

| P# | PF# | P/A present / Absent |
|---|---|---|
| 0 | 4 | 0 |
| 1 | 3 | 1 |
| 2 | 2 | 0 |
| 3 | 3 | 0 |
| 4 | 7 | 1 |

| | | |
|---|---|---|
| 31 | 4 | 0 |

**ث.** اگر حافظه‌ی مجازی ۵۰ بیتی بود، برای هر سطر از جدول صفحه ۱۰ بایت ذخیره می‌کردیم و ۲۰ پردازه‌ی در حال اجرا داشتیم، آیا این روش صفحه بندی مناسب است؟ برای کم کردن سربار، چه راه حل‌هایی مطرح است؟ (۲ نمره)

**پرسش ۲ :** می‌خواهیم عمل‌کرد دو سیستم صفحه‌بندی با داشتن TLB را مقایسه کنیم. فرض کنید زمان دسترسی به یک خانه از حافظه‌ی اصلی 2 micro second باشد. (۴ نمره)

۱. سیستم S1، دارای TLB با زمان پاسخ 120nano seconds،  که در ۶۰٪ مواقع صفحه در TLB است و از صفحه‌بندی ۱ مرحله‌ای استفاده می‌کند.

۲. سیستم S2، دارای TLB بزرگتر با زمان پاسخ 150 nano seconds، که در ۸۰٪ مواقع صفحه در TLB است و از صفحه بندی ۳ مرحله‌ای استفاده می‌کند.

در سیستم S1، یک درخواست همزمان به TLB و page table می‌خورد ولی در سیستم S2، ابتدا درخواست به TLB می‌خورد و در صورت TLB miss، درخواست به page table می‌رود.

برای هر کدام از این دو سیستم، کمینه و بیشینه زمان انتظار برای خواندن مقدار یک خانه از حافظه را به دست آورید.

# جلسه‌ی جدید

# NRU ALGORITHM

# How Can We Do Better?

- Need an approximation of how likely each frame is to be accessed in the future
  - *If we base this on past behavior we need a way to track past behavior*
  - *Tracking memory accesses requires hardware support to be efficient*

# مسئله: یکی از فریم‌هایی که اخیرا استفاده نشده رو بریزیم دور

- چطوری انتخاب کنیم کدوم فریم؟
- علی‌الحساب فرض کنید نمی‌خوایم داده‌ساختار جدیدی نگه داریم...
- مثلا از اطلاعات توی page table استفاده کنیم.
- چه اطلاعاتی داریم؟
- Reference bit
- Dirty bit

# Referenced and Dirty Bits

- Each page table entry (and TLB entry!) has a
  - *Referenced bit - set by TLB when page read / written*
  - *Dirty / modified bit - set when page is written*
  - *If TLB entry for this page is valid, it has the most up to date version of these bits for the page*
  - *OS must copy them into the page table entry during fault handling*
- Idea: use the information contained in these bits to drive the page replacement algorithm

# Referenced and Dirty Bits

- Some hardware does not have support for the dirty bit
- Instead, memory protection can be used to emulate it
- Idea:
  - *Software sets the protection bits for all pages to "read only"*
  - *When program tries to update the page...*
    - A trap occurs
    - Software sets the Dirty Bit in the page table and clears the ReadOnly bit
    - Resumes execution of the program

# Not Recently Used Algorithm

- Uses the Referenced Bit and the Dirty Bit
- Initially, all pages have
  - *Referenced Bit = 0*
  - *Dirty Bit = 0*
- Periodically... (e.g. whenever a timer interrupt occurs)
  - *Clear the Referenced Bit*
  - *Referenced bit now indicates "recent" access*

# Not Recently Used Algorithm

- When a page fault occurs...
- Categorize each page...
  - *Class 1:* Referenced = *0* Dirty = *0*
  - *Class 2:* Referenced = *0* Dirty = *1*
  - *Class 3:* Referenced = *1* Dirty = *0*
  - *Class 4:* Referenced = *1* Dirty = *1*
- Choose a victim page from …
  - *class 1 … why?*
- If none, choose a page from …
  - *class 2 … why?*
- If none, choose a page from …
  - *class 3 … why?*
- If none, choose a page from …
  - *class 4 … why?*

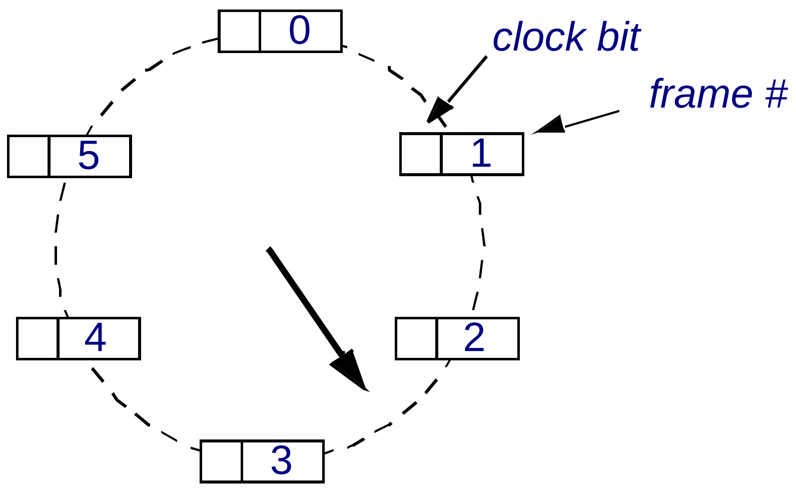
# SECOND CHANCE ALGORITHM

# Second Chance Algorithm

- An implementation of NRU based on FIFO
- Pages kept in a linked list (oldest at the front)
- Look at the oldest page
  - *If its "referenced bit" is 0...*
    - Select it for replacement
  - *Else*
    - It was used recently; don't want to replace it
    - Clear its "referenced bit"
    - Move it to the end of the list
  - *Repeat*
- What if every page was used in last clock tick?

# Implementation of Second Chance

- Maintain a circular list of pages in memory
- Set a bit for the page when a page is referenced
- Search list looking for a victim page that does not have the referenced bit set
  - *If the bit is set, clear it and move on to the next page*
  - *Replaces pages that haven't been referenced for one complete clock revolution*



# Enhanced Second Chance algorithm

- همزمان از هر دو ایده‌ی قبلی استفاده کنیم...
- در ۴ گروه براساس reference bit و dirty bit در نظر بگیریم.
- حالا second chance رو براساس گروه‌بندی قبلی انجام بدیم...

# LRU

# Least Recently Used Algorithm

- A refinement of NRU that orders how recently a page was used
  - *Keep track of when a page is used*
  - *Replace the page that has been used least recently*

# Least Recently Used Algorithm

- Replace the page that hasn't been referenced in the longest time

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Time | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Requests | | | c | a | d | b | e | b | a | b | c | d |
| Page Frames | 0 | a | a | a | a | a | a | a | a | a | a | a |
| | 1 | b | b | b | b | b | b | b | b | b | b | b |
| | 2 | c | c | c | c | c | e | e | e | e | e | d |
| | 3 | d | d | d | d | d | d | d | d | d | c | c |
| Page faults | | | | | | | x | | | | x | x |

# Least Recently Used Algorithm

- But how can we implement LRU?

- Idea #1:
  - *Keep a linked list of all pages*
  - *On every memory reference, Move that page to the front of the list*
  - *The page at the tail of the list is replaced*

# Least Recently Used Algorithm

- But how can we implement LRU?
  - *… without requiring every access to be recorded?*
- Idea #2:
  - *MMU (hardware) maintains a counter*
  - *Incremented on every clock cycle*
  - *Every time a page table entry is used*
    - MMU writes the value to the page table entry
    - This *timestamp* value is the *time-of-last-use*
  - *When a page fault occurs*
    - OS looks through the page table
    - Identifies the entry with the oldest timestamp

# Least Recently Used Algorithm

- What if we don't have hardware support for a counter?
- Idea #3:
  - *Maintain a counter in software*
  - *One every timer interrupt...*
    - Increment counter
    - Run through the page table
    - For every entry that has "ReferencedBit" = 1
      - ** Update its timestamp*
      - ** Clear the ReferencedBit*
  - *Approximates LRU*
  - *If several have oldest time, choose one arbitrarily*

# NFU

# Not Frequently Used Algorithm

- Bases decision of frequency of use rather than recency
- Associate a counter with each page
- On every clock interrupt, the OS looks at each page.
  - *If the reference bit is set increment that page's counter & clear the bit*
- The counter approximates how often the page is used
- For replacement, choose the page with lowest counter

# Not Frequently Used Algorithm

- *Problem:*
  - *Some page may be heavily used*
    - Its counter is large
  - *The program's behavior changes*
    - Now, this page is not used ever again (or only rarely)
  - *This algorithm never forgets!*
    - - *This page will never be chosen for replacement!*
  - *We may want to combine frequency and recency somehow*

# NFU With Aging

- Associate a counter with each page
- On every clock tick, the OS looks at each page.
  - *Shift the counter right 1 bit (divide its value by 2)*
  - *If the reference bit is set...*
    - Set the most-significant bit
    - Clear the Referenced Bit

  - $T_1$ *100000 = 32*
  - $T_2$ *010000 = 16*
  - $T_3$ *001000 = 8*
  - $T_4$ *000100 = 4*
  - $T_5$ *100010 = 34*

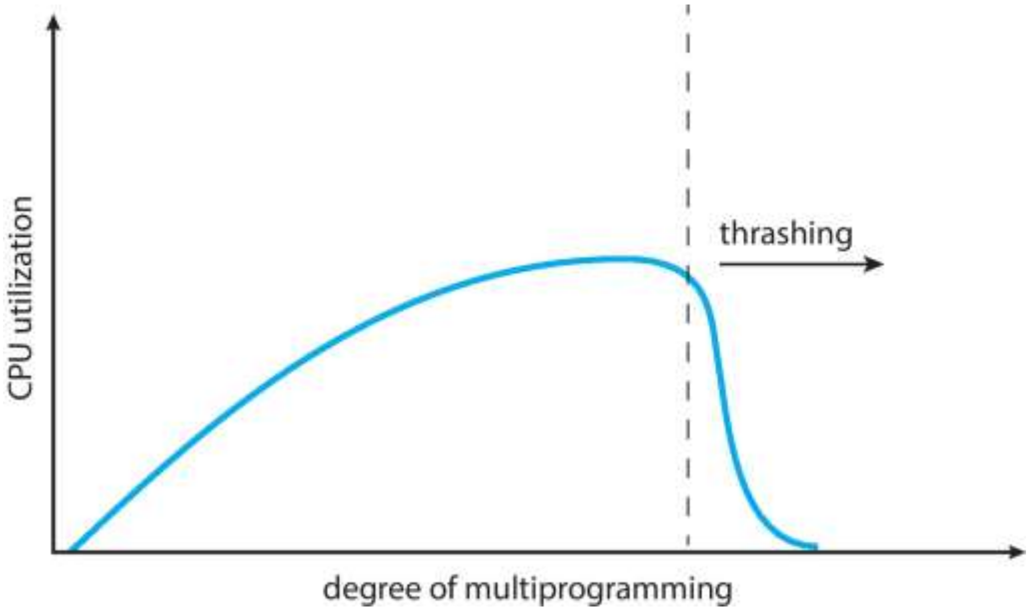
# THRASHING

# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - *Page fault to get page*
  - *Replace existing frame*
  - *But quickly need replaced frame back*
  - *This leads to:*
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

# Thrashing (Cont.)

- **Thrashing**. A process is busy swapping pages in and out



# WORKING SET

# The Working Set

- *Demand paging*
  - *Pages are only loaded when accessed*
  - *When process begins, all pages marked INVALID*

# The Working Set

- *Demand paging*
  - *Pages are only loaded when accessed*
  - *When process begins, all pages marked INVALID*
- *Locality of reference*
  - *Processes tend to use only a small fraction of their pages*
- *Working Set*
  - *The set of pages a process needs*
  - *If working set is in memory, no page faults*
  - *What if you can't get working set into memory?*

# The Working Set

- *Thrashing*
  - *If you can't get working set into memory page faults occur every few instructions*
  - *Little work gets done*
  - *Most of the CPU's time is going on overhead*

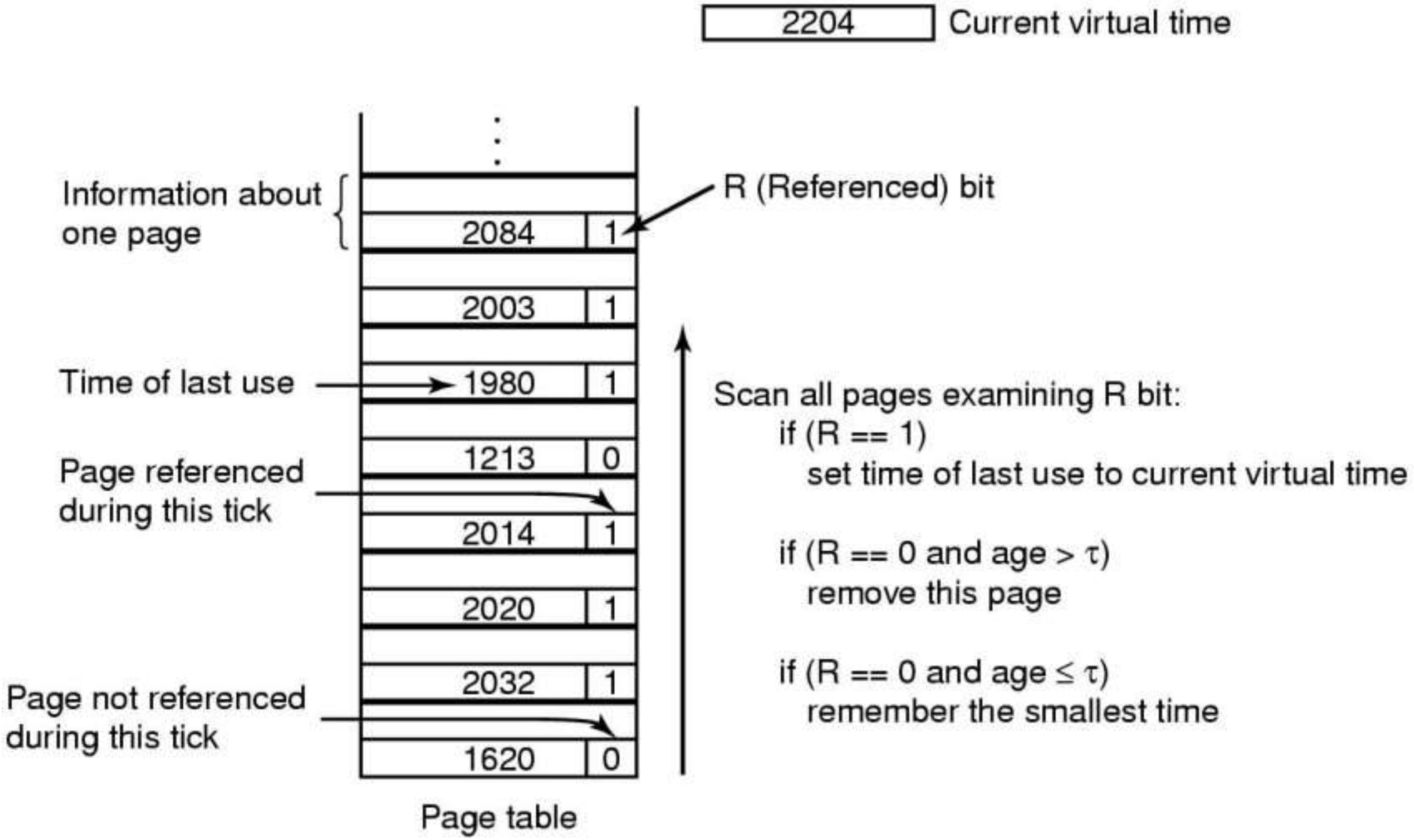
# Working Set Algorithm

- *Based on prepaging (prefetching)*
  - *Load pages before they are needed*
- *Main idea:*
  - *Try to identify the process's working set based on time*
  - *Keep track of each page's time since last access*
  - *Assume working set valid for T time units*
  - *Replace pages older than T*

# Working Set Algorithm

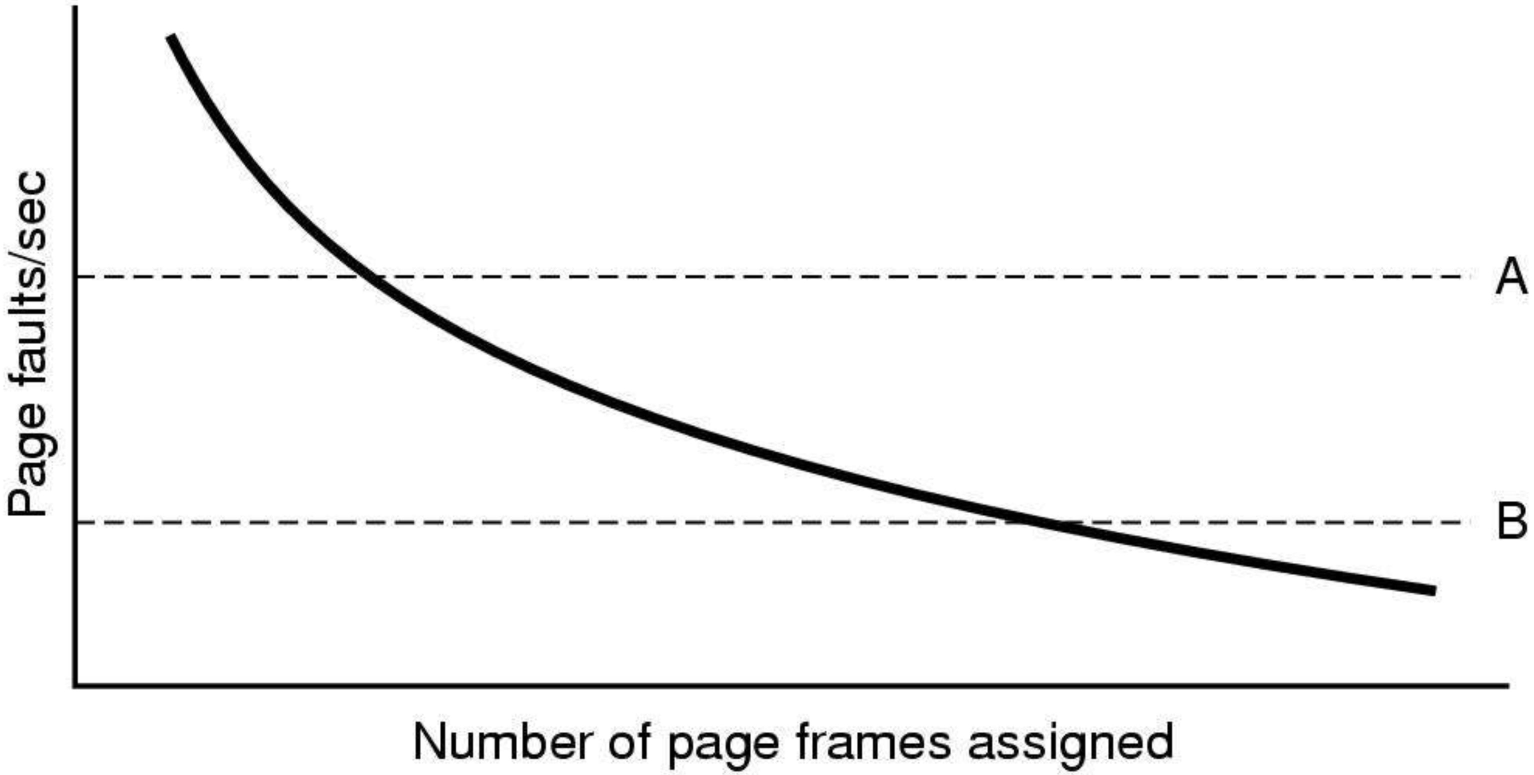
- *Current Virtual Time*
  - *Only consider how much CPU time this process has seen*
- *Implementation*
  - *On each clock tick, look at each page*
  - *Was it referenced since the last check?*
    - Yes: make a note of Current Virtual Time
  - *If a page has not been used in the last T msec,*
    - Assume it is not in the working set!
    - Evict it
    - Write it out if it is dirty

# Working Set Algorithm

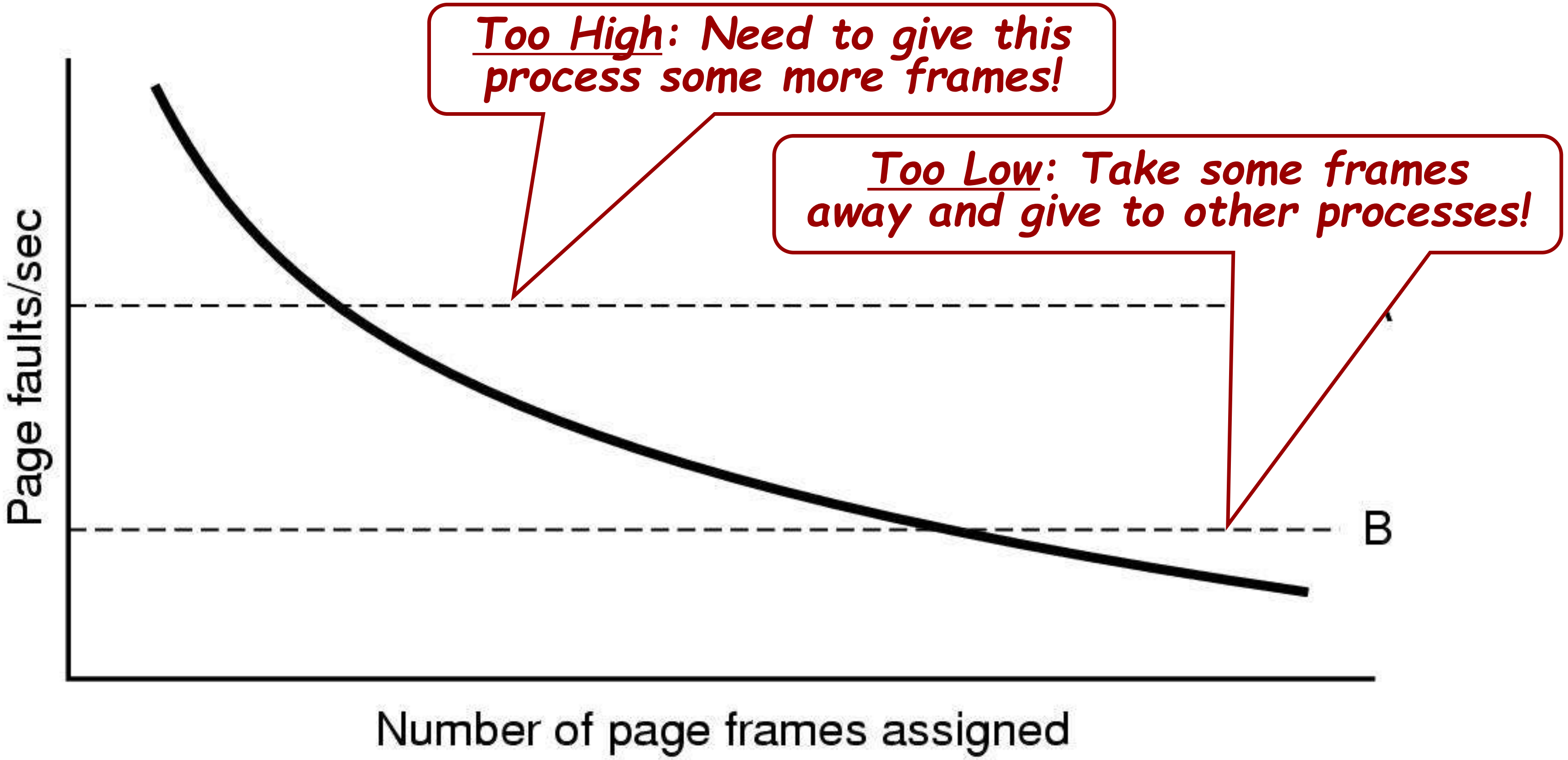


# Page Fault Frequency

- *If T is too small, page fault frequency will be high*
- *If you make it bigger page fault frequency will decline*



# Page Fault Frequency



# Page Fault Frequency

- Measure the page fault frequency of each process
- Count the number of faults every second
- May want to consider the past few seconds as well
- Aging:
  - *Keep a running value*
  - *Every second*
    - Count number of page faults
    - Divide running value by 2
    - Add in the count for this second

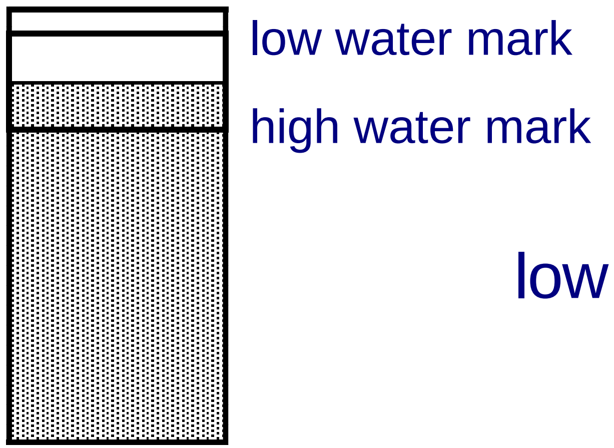
# WHICH ALGORITHM IS BEST?

# Modeling Algorithm Performance

- Run a program
  - *Look at all memory references*
  - *Don’t need all this data*
  - *Look at which pages are accessed*
  - *0000001222333300114444001123444*
  - *Eliminate duplicates*
  - *012301401234*
- This defines the *Reference String*
  - *Use this to evaluate different algorithms*
  - *Count page faults given the same reference string*

# Proactive Replacement

- Replacing victim frame on each page fault typically requires two disk accesses per page fault
- Alternative → the O.S. can keep several pages free in anticipation of upcoming page faults.
- In Unix: low and high water marks




low < # free pages < high

# UNIX Page Replacement

- Clock algorithm for page replacement
  - *If page has not been accessed move it to the free list for use as allocatable page*
  - *If modified/dirty → write to disk (still keep stuff in memory though)*
  - *If unmodified → just move to free list*

- High and low water marks for free pages
  - *Pages on the free-list can be re-allocated if they are accessed again before being overwritten*

# Local vs. Global Replacement

- Assume several processes: A, B, C, ...
- Some process gets a page fault (say, process A)
- Choose a page to replace.
- *Local page replacement*
  - *Only choose one of A's pages*
- *Global page replacement*
  - *Choose any page*

# Local vs. Global Replacement

| Original | Age | Local | Global |
|---|---|---|---|
| A0 | 10 | A0 | A0 |
| A1 | 7 | A1 | A1 |
| A2 | 5 | A2 | A2 |
| A3 | 4 | A3 | A3 |
| A4 | 6 | A4 | A4 |
| A5 | 3 | A6 | A5 |
| B0 | 9 | B0 | B0 |
| B1 | 4 | B1 | B1 |
| B2 | 6 | B2 | B2 |
| B3 | 2 | B3 | A6 |
| B4 | 5 | B4 | B4 |
| B5 | 6 | B5 | B5 |
| B6 | 12 | B6 | B6 |
| C1 | 3 | C1 | C1 |
| C2 | 5 | C2 | C2 |
| C3 | 6 | C3 | C3 |

# Local vs. Global Replacement

- Assume we have
  - *5,000 frames in memory and 10 processes*
- Idea: Give each process 500 frames
- Is this fair?
  - *Small processes do not need all those pages*
  - *Large processes may benefit from even more frames*
- Idea:
  - *Look at the size of each process (… but how?)*
  - *Give them a pro-rated number of frames with some minimum*

# Load Control

- Assume:
  - *The best page replacement algorithm*
  - *Optimal global allocation of page frames*
- *Thrashing is still possible!*
  - *Too many page faults!*
  - *No useful work is getting done!*
  - *Demand for frames is too great!*
- Solution:
  - *Get rid of some processes (temporarily swap them out)*
  - *Two-level scheduling (swapping with paging)*