BUCKNELL UNIVERSITY

Computer Science

# CSCI 315 Operating Systems Design

## Page Replacement
## -- Part 2 of 2

**Notice:** The slides for this lecture have been largely based on those accompanying an earlier edition of the course text *Operating Systems Concepts, 8th ed.*, by Silberschatz, Galvin, and Gagne. Many, if not all, of the illustrations contained in this presentation come from this source. Revised by X.M. Based on Professor Perrone's notes.

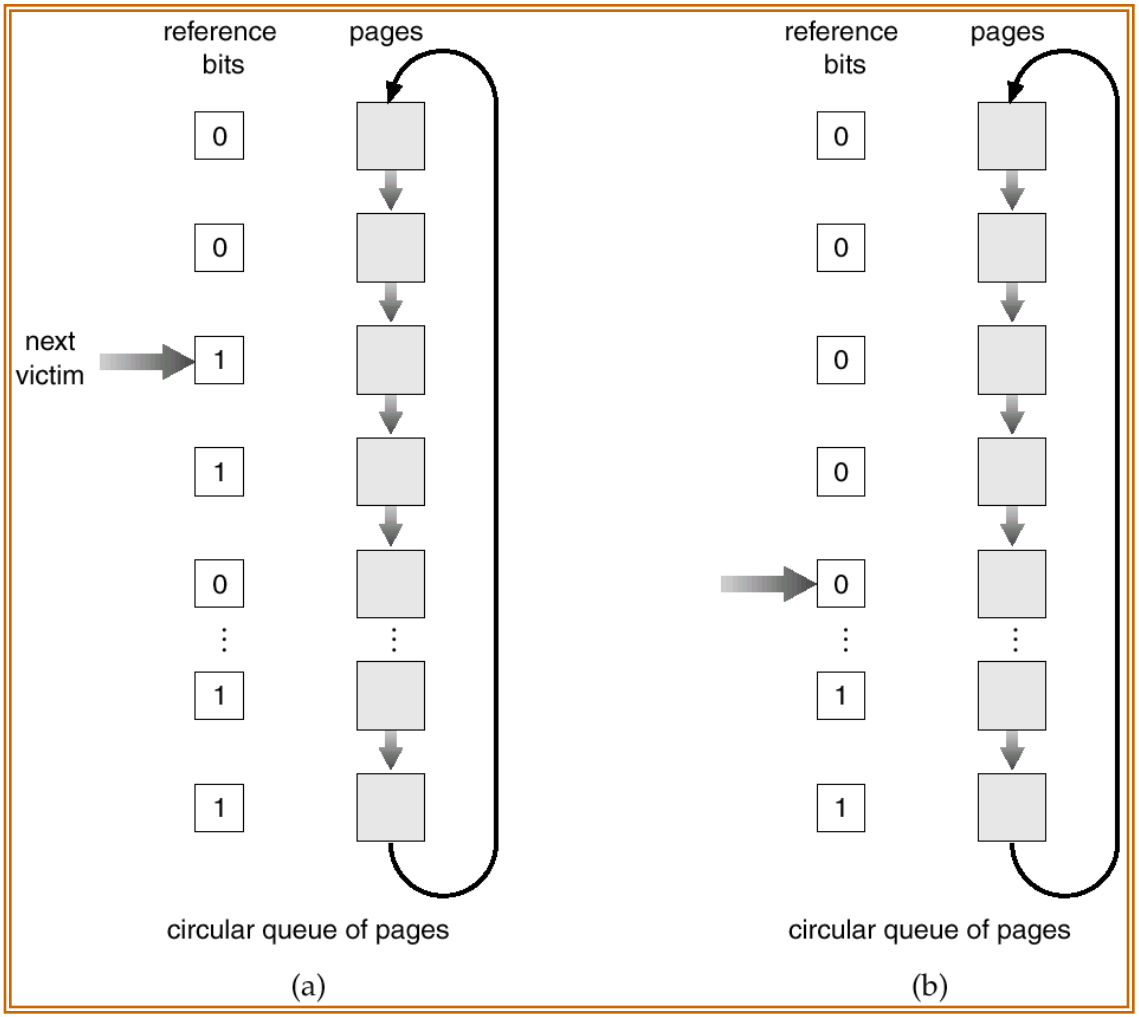
# LRU Approximation Algorithms

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1.
  - Replace the one which is 0 (if one exists). We do not know the order, however.
- **Additional reference bits (e.g., 8 bits)**
  - Every time a page is referenced
    - Shift the reference bits to the right by 1
    - Place the reference bit (1 if being visited, 0 otherwise) into the high order bit of the reference bits
    - The page with the lowest reference bits value is the one that is Least Recently Used, thus to be replaced
  - E.g., the page with ref bits 11000100 is more recently used than the page with ref bits 01110111

# LRU Approximation Algorithms

- **Second Chance**
  - If we consider the number of reference history bits to be zero, only using the reference bit itself, we have the *Second Chance* (a.k.a. *Clock*) algorithm
  - Need a pointer (clock handle) to point the next victim.
  - At each clock interruption, we check the reference bit for the victim.
  - If the victim page has reference bit = 1, then:
    - set reference bit 0.
    - leave this page in memory.
  - Else if the page reference bit is 0, this page can be replaced.

# Second-Chance (Clock) Page-Replacement Algorithm



# Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- **LFU Algorithm:** replaces page with smallest count.
- **MFU Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

# Allocation of Frames

- Each process needs a **minimum** number of pages.
- There are two major allocation schemes:
  - **fixed allocation**
  - **priority allocation**

# Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.
  - $s_i$ = size of process $p_i$
  - $S = \sum s_i$
  - $m$ = total number of frames
  - $a_i$ = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

# Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.
- If process $P_i$ generates a page fault,
  - select for replacement one of its frames; or
  - select for replacement a frame from a process with lower priority number.

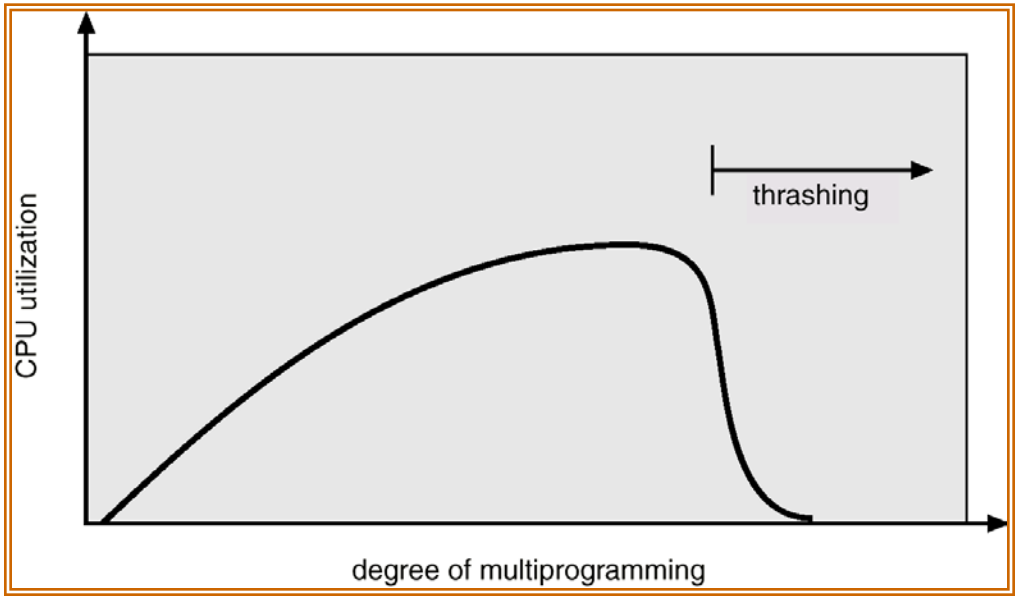
# Global vs. Local Allocation

- **Global** replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local** replacement – each process selects from only its own set of allocated frames.

# Thrashing

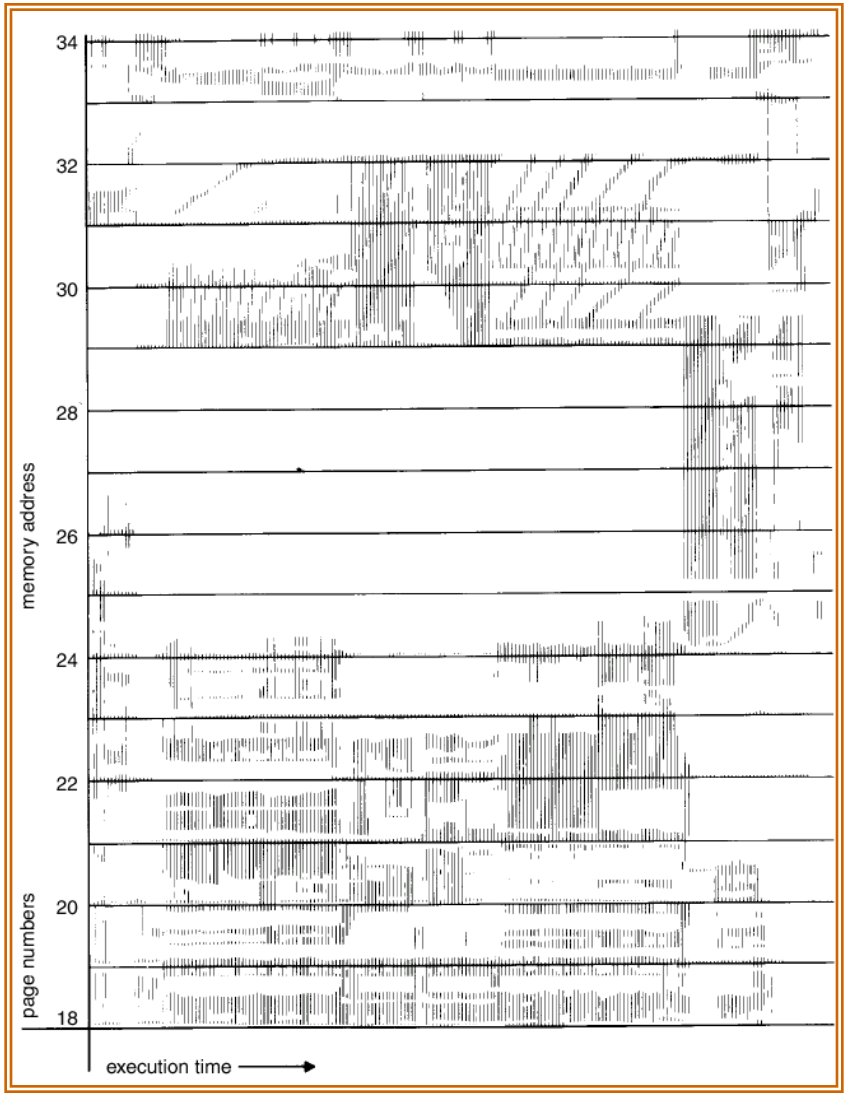
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - **Low CPU utilization**.
  - Operating system thinks that it needs to increase the degree of multiprogramming.
  - Another process added to the system.
- **Thrashing** ≡ a process is busy swapping pages in and out to a degree that no effective computing is accomplished.

# Thrashing




- Why does paging work?
  Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  $\Sigma$ size of locality > total memory size

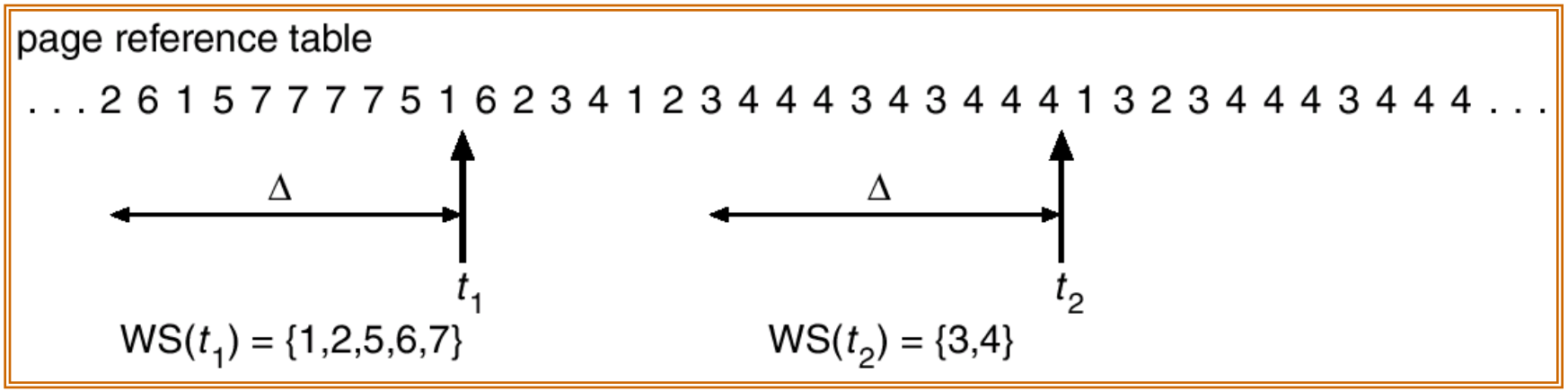
# Locality in Memory-Reference Pattern



# Working-Set Model

- $\Delta \equiv$ **working-set window** $\equiv$ a fixed number of page references.
- $WSS_i$ (working set of Process $P_i$) = total number of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality.
  - if $\Delta$ too large will encompass several localities.
  - if $\Delta = \infty \Rightarrow$ will encompass entire program.
- $D = \Sigma\ WSS_i \equiv$ total demand frames from all active processes
- if $D > m \Rightarrow$ **Thrashing**
- Policy if $D >$ m, then suspend one of the processes.

# Working-set Model Illustration

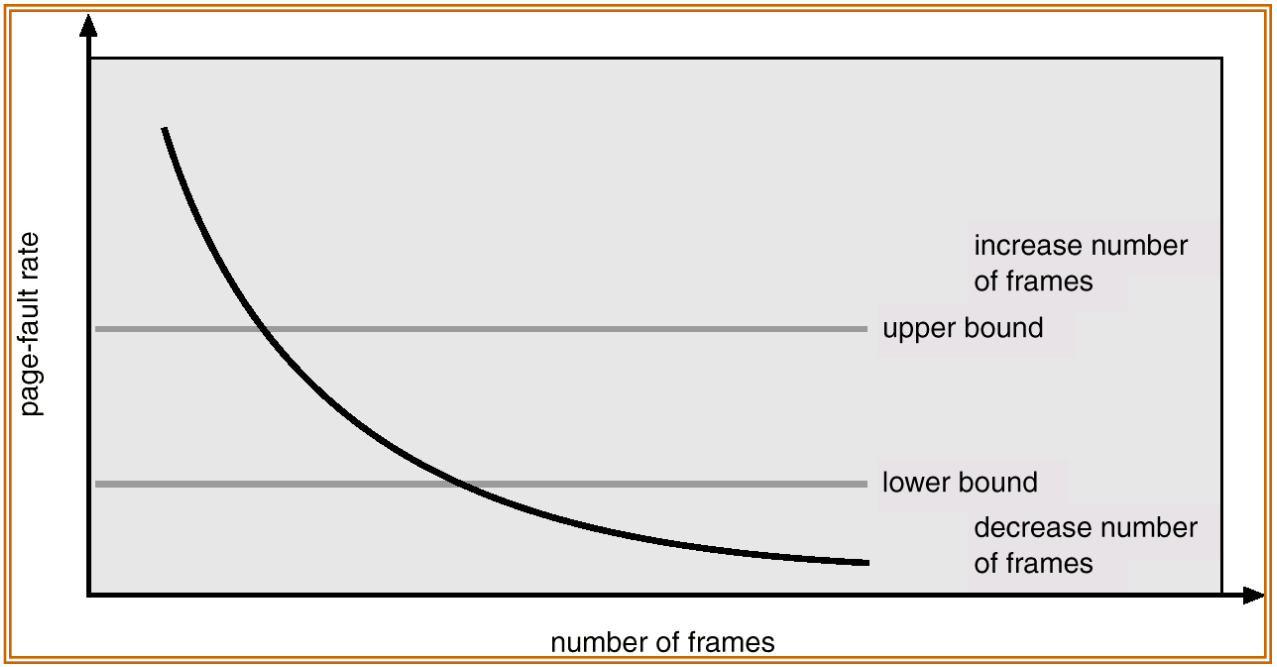



Δ = 10 pages (frames)

# Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta$ = 10,000
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts copy and set the values of all reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.

# Page-Fault Frequency Scheme




**Establish "acceptable" page-fault rate.**

- If actual rate too low, process loses frame.
- If actual rate too high, process gains frame.

# Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time

# Prepaging

- **Prepaging**: In order to avoid the initial number of page faults, the system can bring into memory all the pages that will be needed all at once.
- This can also be applied when a swapped-out process is restarted. The smart thing to do is to remember the working set of the process.
- One question that arises is whether all the pages brought in will actually be used…
- Is the cost of prepaging less than the cost of servicing each individual page fault?

# The Effect of Program Structure

- Program structure
  - `int[128][128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

128 page faults