10장. 가상 메모리 (3)

운영체제

목차

학습목표

- 가상메모리를 정의하고 그 이점을 설명한다.
- 요구 페이징을 사용하여 페이지가 메모리에 적재되는 방법을 설명한다.
- FIFO, 최적 및 LRU 페이지 교체 알고리즘을 적용한다.
- 프로세스의 작업집합을 설명하고, 프로그램 지역성과 어떤 관련이 있는지 설명한다.
- Linux, Windows 10 및 Solaris 가상메모리를 관리하는 방법을 설명한다.
- C 프로그램 언어로 가상 메모리 관리자 시뮬레이션을 설계한다.

10.6 THRASHING

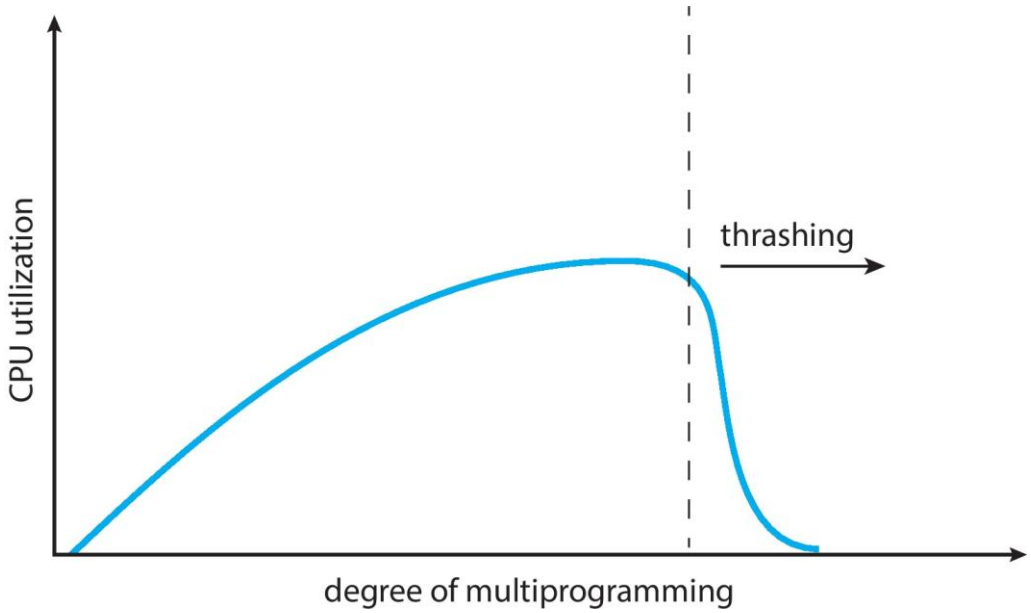
10. Virtual Memory

10.6.1 Cause of 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 the 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



Demand Paging and Thrashing

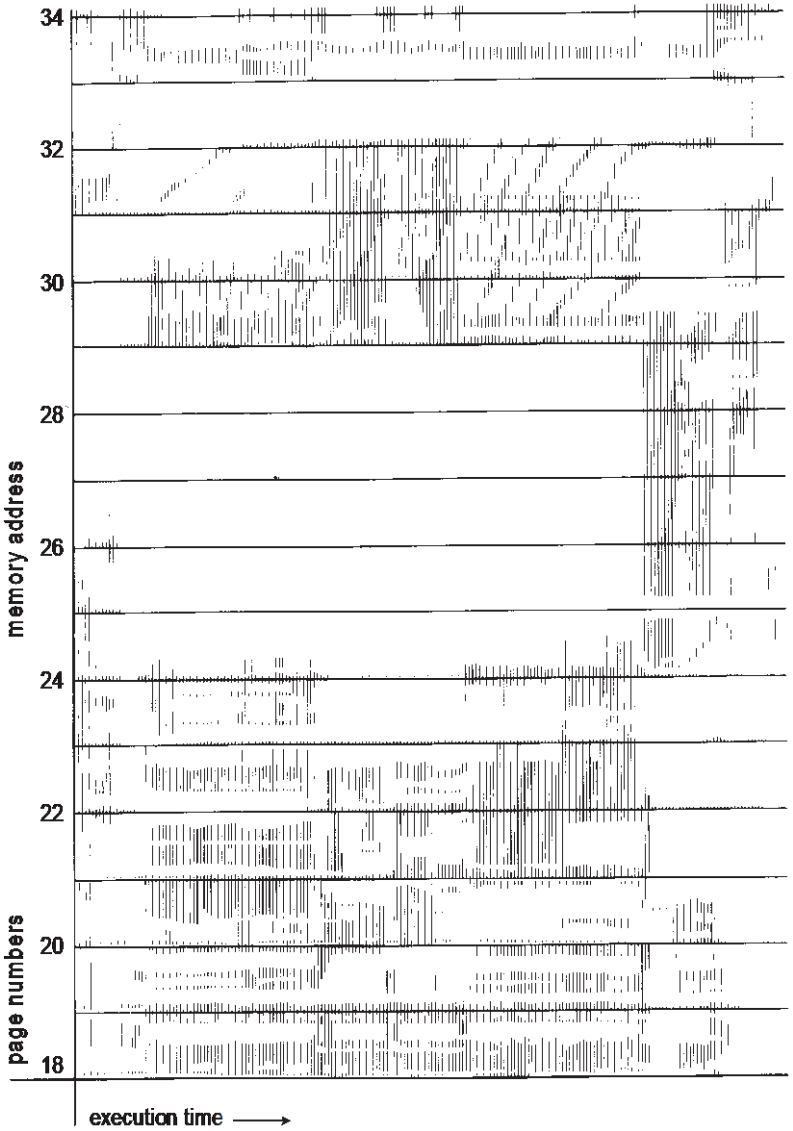
- Why does demand paging work?

  Locality model

  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size
- To avoid trashing:
  - Calculate the Σ size of locality
  - Policy:
    - if Σ size of locality > total memory size→ suspend or swap out one of the processes
- Issue: how to calculate "Σ size of locality"

Locality In A Memory-Reference Pattern

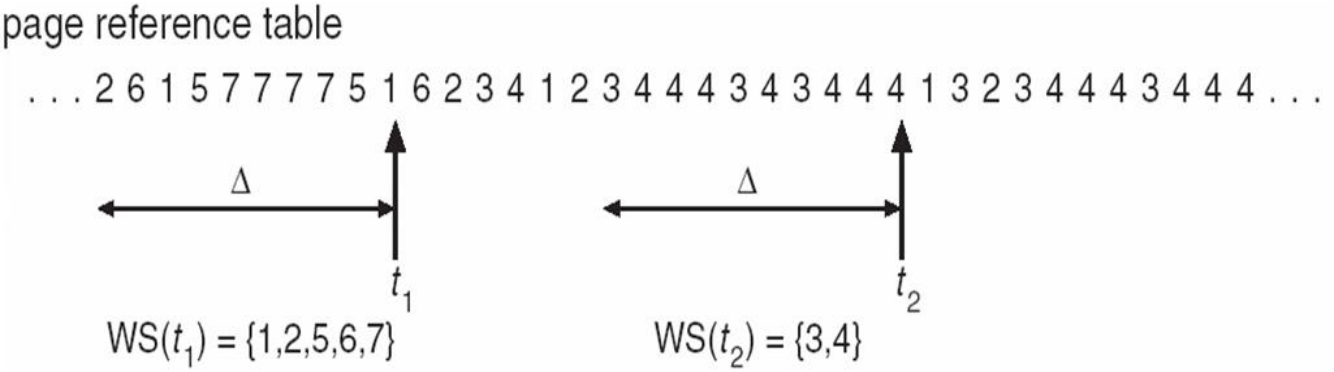


10.6.2 Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass the entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

Working-Set Model (cont.)

- Example



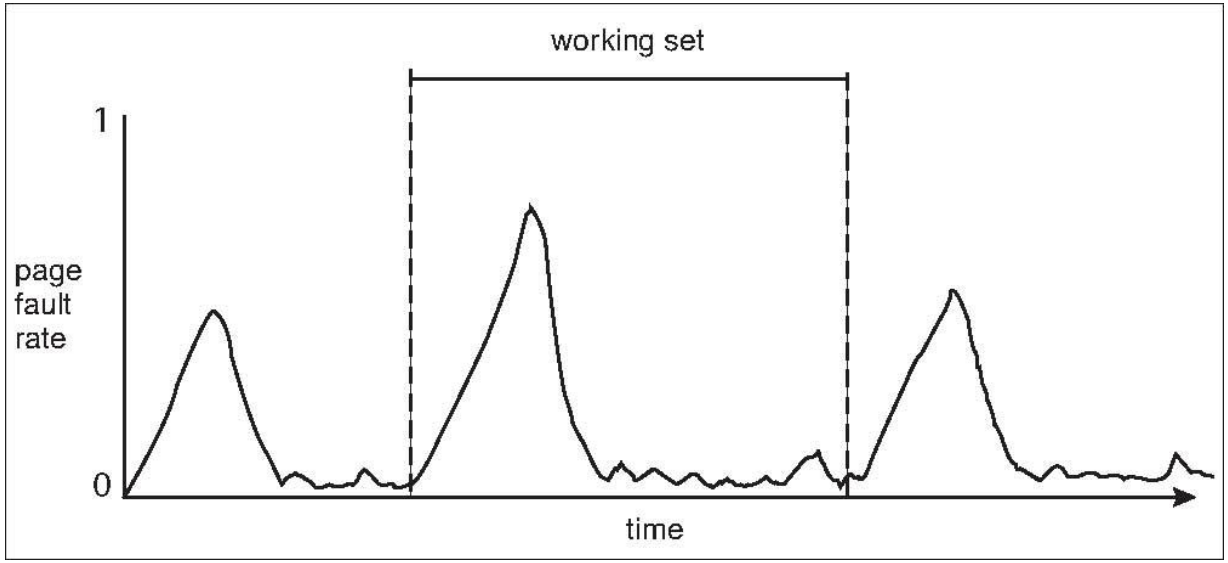
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- $m =$ total number of frames
- If $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page i – $B1_i$ and $B2_i$
  - Whenever a timer interrupts copy the reference to one of the B_j and sets the values of all reference bits to 0
  - If either $B1_i$ or $B2_i$ = 1, it implies that Page i is in the working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

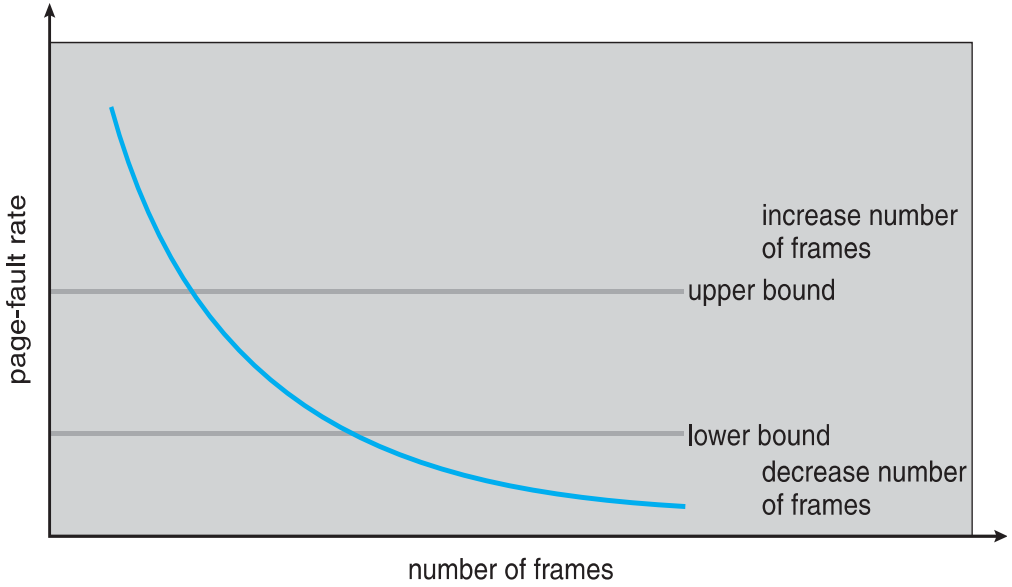
Working Sets and Page Fault Rates

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



10.6.3 Page-Fault Frequency, PFF

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



10.7 MEMORY COMPRESSION

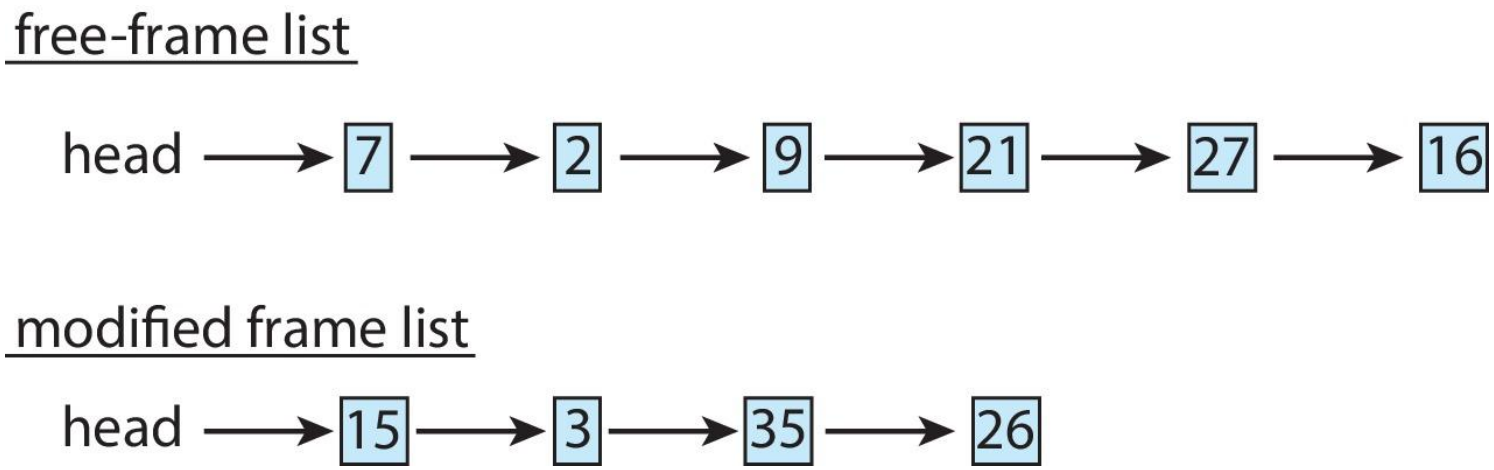
10. Virtual Memory

Memory Compression

- An alternative to paging is **memory compression**.
- Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

Memory Compression (cont.)

- Consider the following free-frame-list consisting of 6 frames

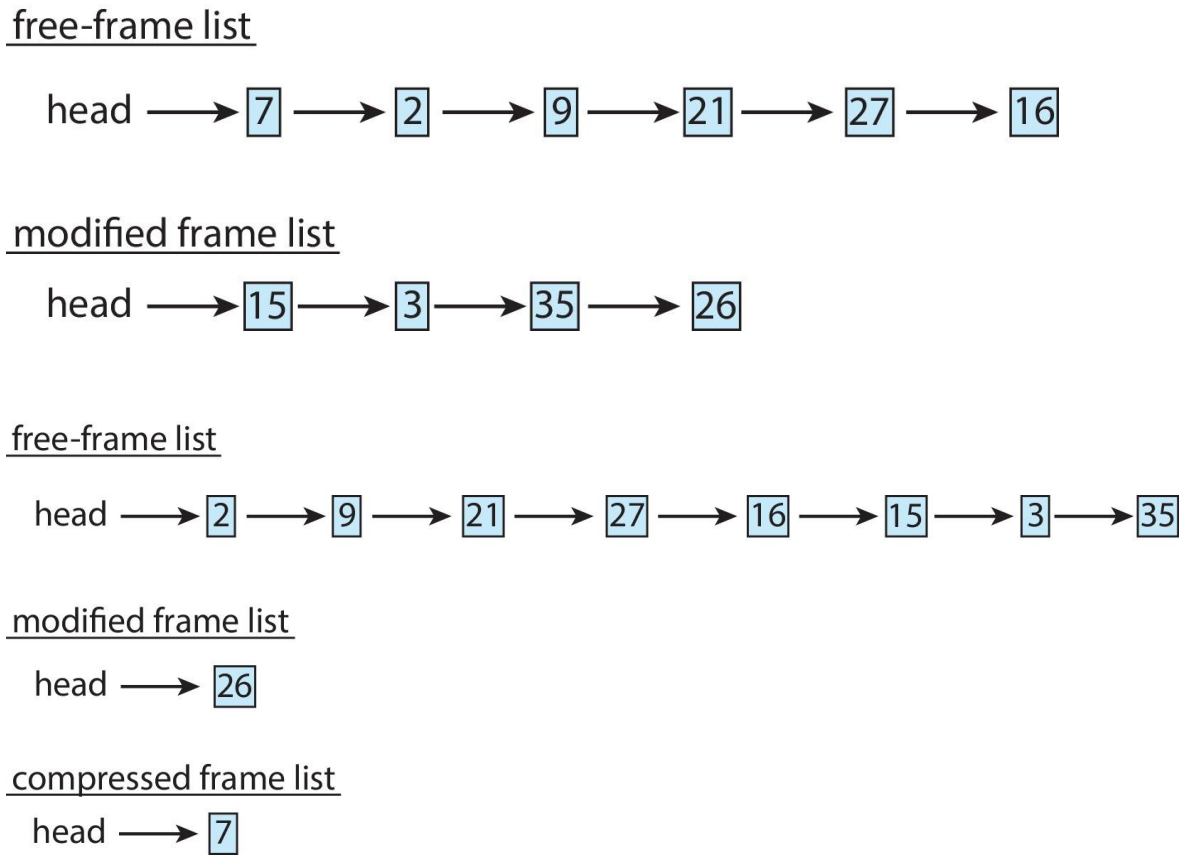


Memory Compression (cont.)

- Assume that this number of free frames falls below a certain threshold that triggers page replacement.
- The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list.
- It first places these frames on a modified-frame list.
- Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list.
- An alternative strategy is to compress a number of frames(three) and store their compressed versions in a single page frame.

Memory Compression (cont.)

- Consider the following free-frame-list consisting of 6 frames



10.8 ALLOCATING KERNEL MEMORY

10. Virtual Memory

Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - i.e., for device I/O
- Two schemes:
  - Buddy System
  - Slab Allocator

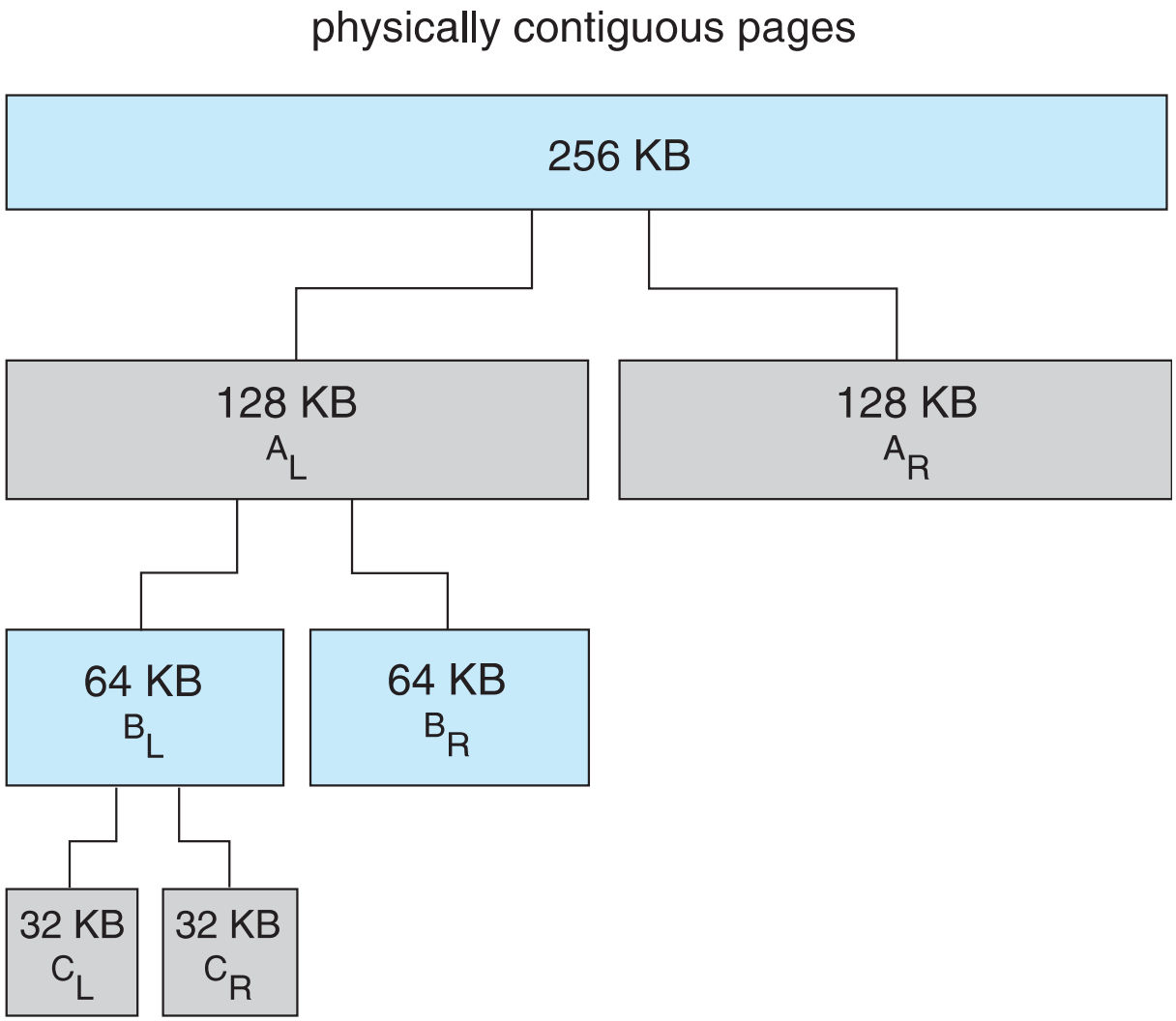
10.8.1 Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

Buddy System Example

- Assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request

Buddy System Allocator



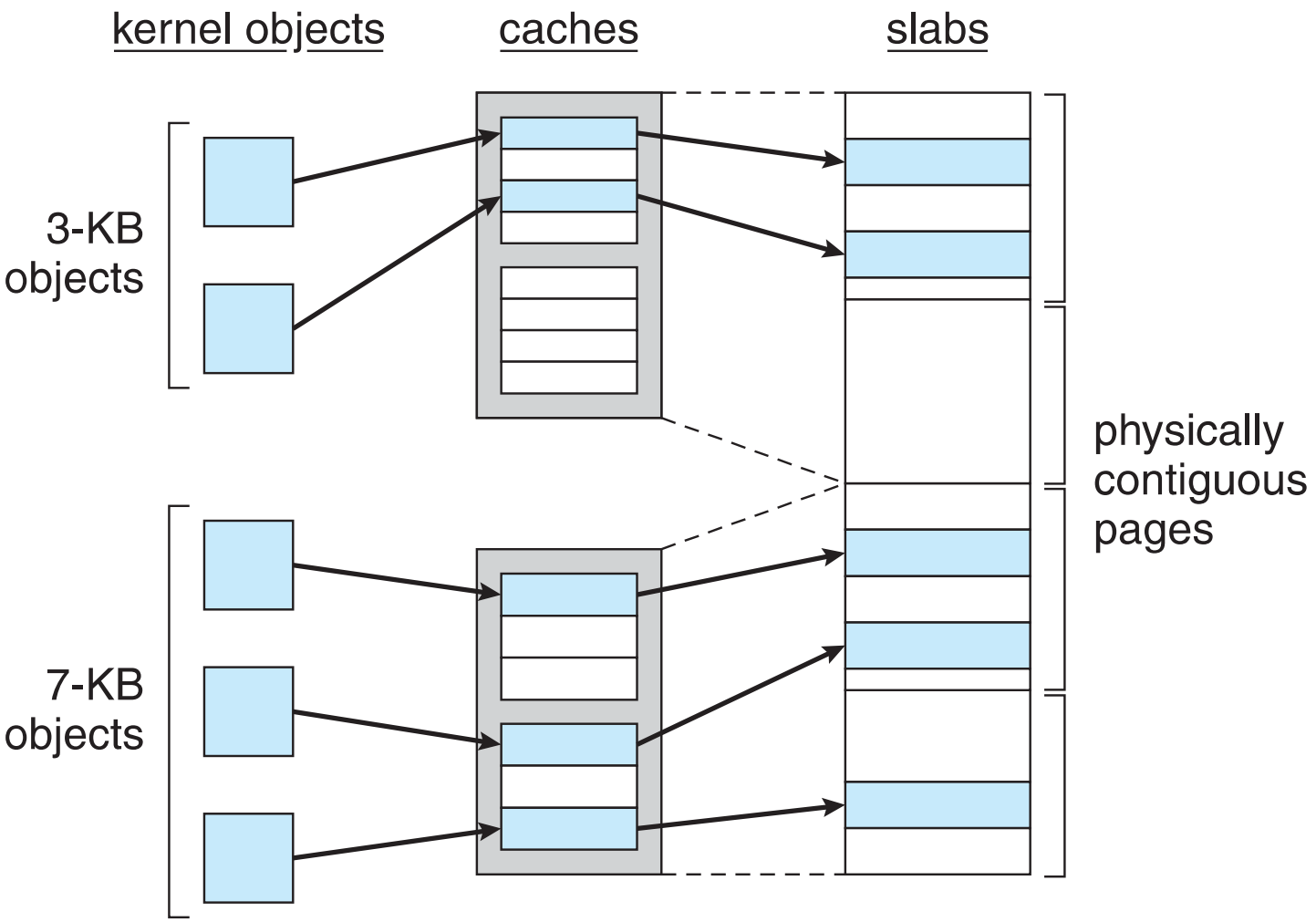
Buddy System Example

- Assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

10.8.2 Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab Allocator in Linux

- For example, process descriptor is of type `struct task_struct`
- Approximately 1.7 KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

10.9 OTHER CONSIDERATIONS

10. Virtual Memory

Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

10.9.1 Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume *s* pages are prepaged and α of the pages is used
  - Question: is the cost of $s * \alpha$ save pages faults is greater or less than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages*?*
  - If α is close to 0 $\Rightarrow$ prepaging loses
  - If α is close to 1 $\Rightarrow$ prepaging wins

10.9.2 Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

10.9.3 TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise, there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

10.9.4 Inverted Page Table

- <process-id, page-number>
- 참조된 페이지가 현재 물리 메모리 안에 없을 때 문제
  - 프로세스 하나당 확장된 페이지 테이블 유지해야
  - 결국은 프로세스당 확장된 페이지 테이블이 존재해야 하므로 메모리 절약은 무의미한 것이 아닌가?
  - 페이지 부재가 발생했을 때만 참조되므로 모든 것이 항상 있을 필요가 없고, 따라서 이 자체도 페이징할 수 있음
    - 페이지 부재가 또다른 페이지 부재로 이어짐
    - 즉, 페이지 look-up 처리 시 시간 지연 발생

10.9.5 Program Structure

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

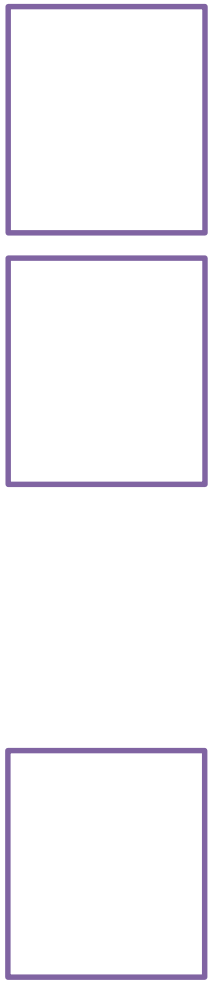
```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

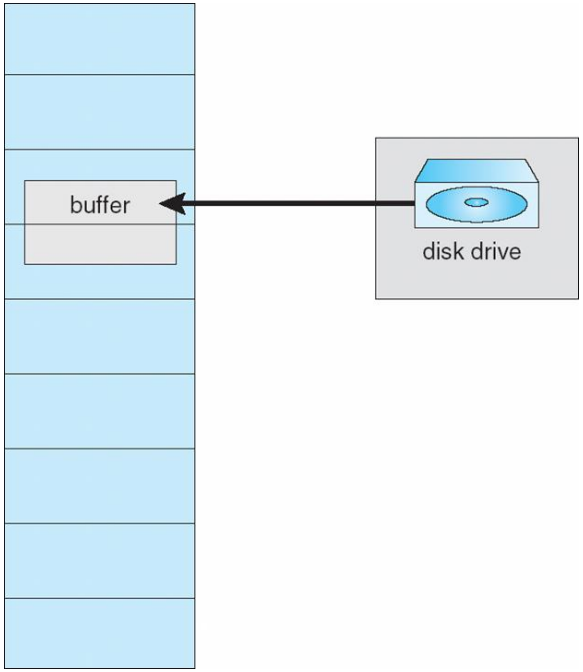
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults



10.9.6 I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



10.10 OPERATING SYSTEM EXAMPLES

10. Virtual Memory

Operating System Examples

- Linux
- Windows
- Solaris

10.10.1 Linux

- 요구 페이징을 이용하여 가용 프레임 리스트에서 페이지 할당
- LRU 근사 알고리즘과 유사한 전역 페이지 교체 정책 사용
- 각 페이지에 참조할 때마다 설정하는 accessed 비트 존재
- active_list
- inactive_list
- kswapd
  - active_list와 inactive_list 상대적 균형 유지

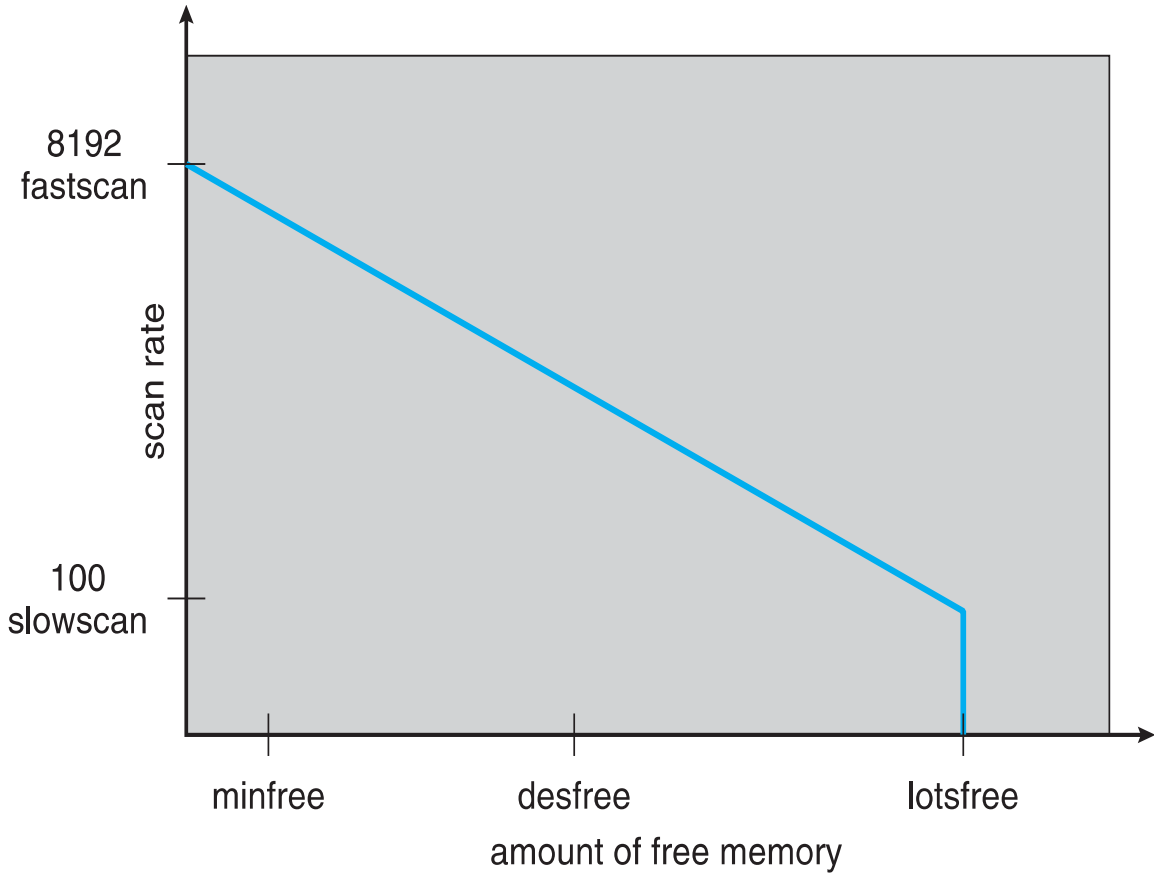
10.10.2 Windows

- Uses demand paging with **clustering**.
  - Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
  - Working set minimum is the minimum number of pages the process is guaranteed to have in memory
  - A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold**, automatic working set trimming** is performed to restore the amount of free memory
  - Working set trimming removes pages from processes that have pages in excess of their working set minimum

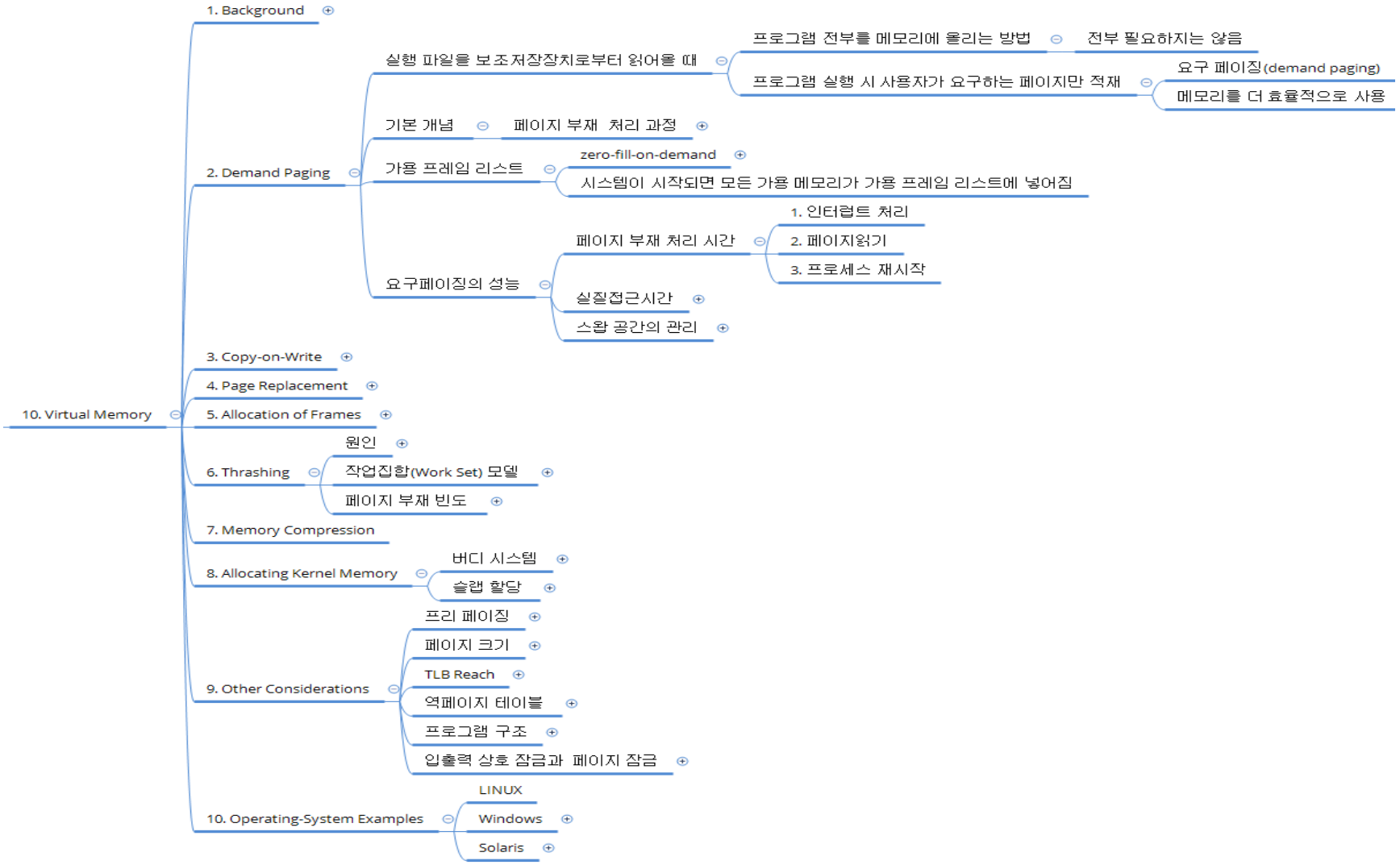
10.10.3 Solaris

- Maintains a list of free pages to assign faulting processes
- **`Lotsfree`** – threshold parameter (amount of free memory) to begin paging
- **`Desfree`** – threshold parameter to increasing paging
- **`Minfree`** – threshold parameter to being swapping
- Paging is performed by **`pageout`** process
  - **`Pageout`** scans pages using modified clock algorithm
- **`Scanrate`** is the rate at which pages are scanned.
  - This ranges from **`slowscan`** to **`fastscan`**
- **`Pageout`** is called more frequently depending upon the amount of free memory available
- ***Priority paging*** gives priority to process code pages

Solaris 2 Page Scanner



학습한 내용


10. Virtual Memory
1. Background
프로그램 전부를 메모리에 올리는 방법
전부 필요하지는 않음
실행 파일을 보조저장장치로부터 읽어올 때
프로그램 실행 시 사용자가 요구하는 페이지만 적재
요구 페이징(demand paging)
메모리를 더 효율적으로 사용
2. Demand Paging
기본 개념
페이지 부재 처리 과정
가용 프레임 리스트
zero-fill-on-demand
시스템이 시작되면 모든 가용 메모리가 가용 프레임 리스트에 넣어짐
요구페이징의 성능
페이지 부재 처리 시간
1. 인터럽트 처리
2. 페이지읽기
3. 프로세스 재시작
실질접근시간
스왑 공간의 관리
3. Copy-on-Write
4. Page Replacement
5. Allocation of Frames
6. Thrashing
원인
작업집합(Work Set) 모델
페이지 부재 빈도
7. Memory Compression
8. Allocating Kernel Memory
버디 시스템
슬랩 할당
9. Other Considerations
프리 페이징
페이지 크기
TLB Reach
역페이지 테이블
프로그램 구조
입출력 상호 잠금과 페이지 잠금
10. Operating-System Examples
LINUX
Windows
Solaris