




# **Virtual Memory - Remaining topics**

# Agenda

- Problem of Thrashing and possible solutions
  - Mmap(), Memory mapped files
  - Kernel Memory Management
  - Other Considerations
- 

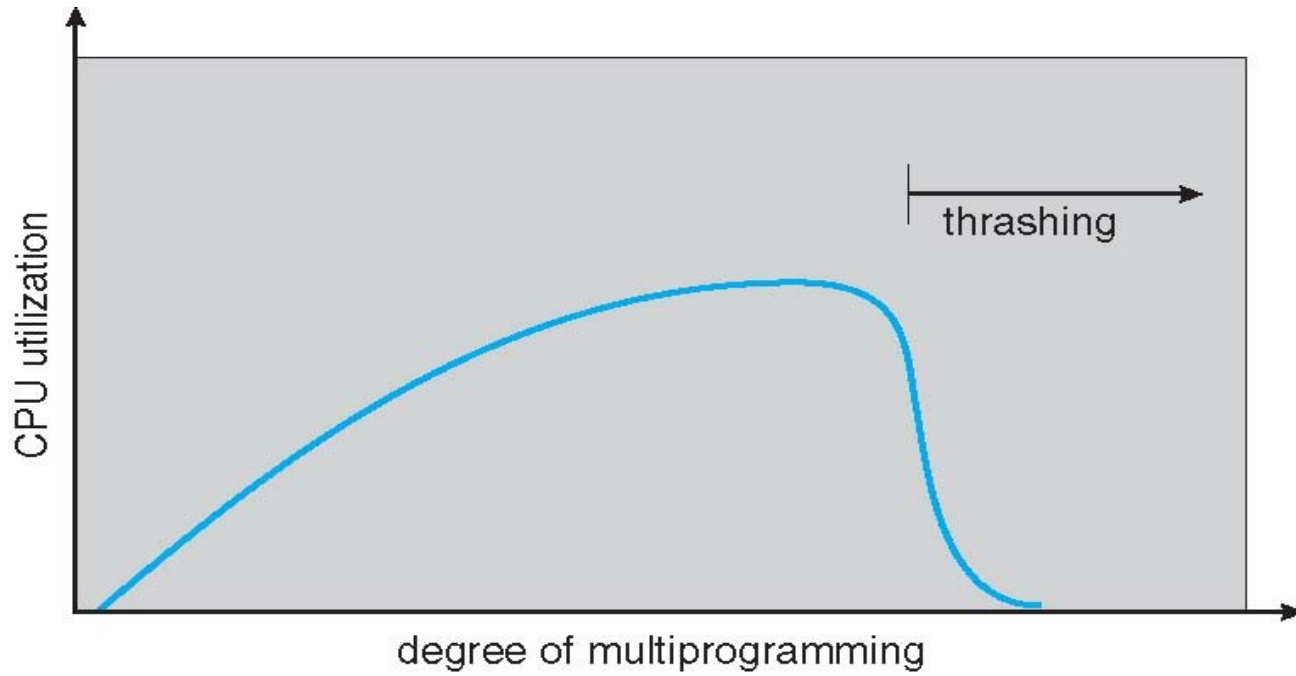


# Thrashing

# 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 : a process is busy swapping pages in and out

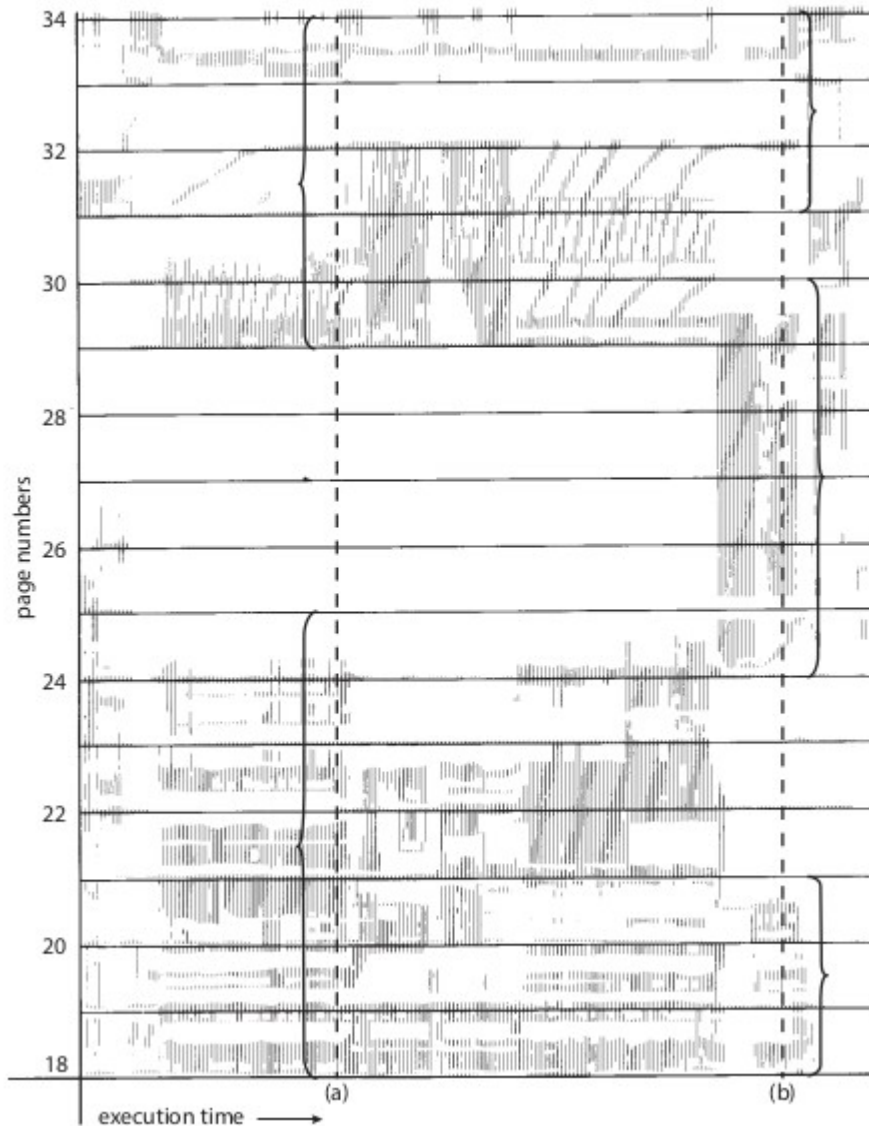
# Thrashing



# 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?
  - size of locality  $>$  total memory size
  - Limit effects by using local or priority page replacement

# Locality In A Memory- Reference Pattern



# Working set model

- $\Delta \equiv \text{working-set window} \equiv \text{a fixed number of page references}$ 
  - Example: 10,000 instructions
- Working Set Size,  $WSS_i$  (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

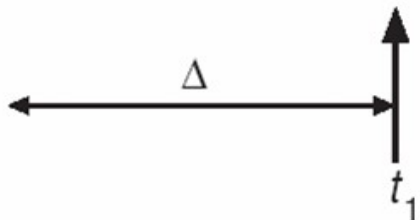


# Working set model

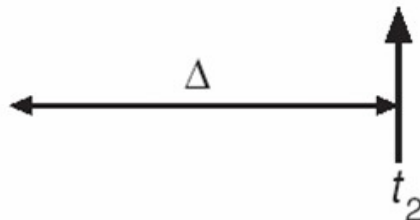
- $\mathcal{D} = \sum WSS_i \equiv$  total demand frames
  - Approximation of locality
- if  $\mathcal{D} > m$  (total available frames)  $\Rightarrow$  Thrashing
- Policy if  $\mathcal{D} > m$ , then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$WS(t_1) = \{1, 2, 5, 6, 7\}$



$WS(t_2) = \{3, 4\}$

# 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 (to memory) and sets the values of all reference bits to 0

If one of the bits in memory = 1  $\Rightarrow$  page in working set

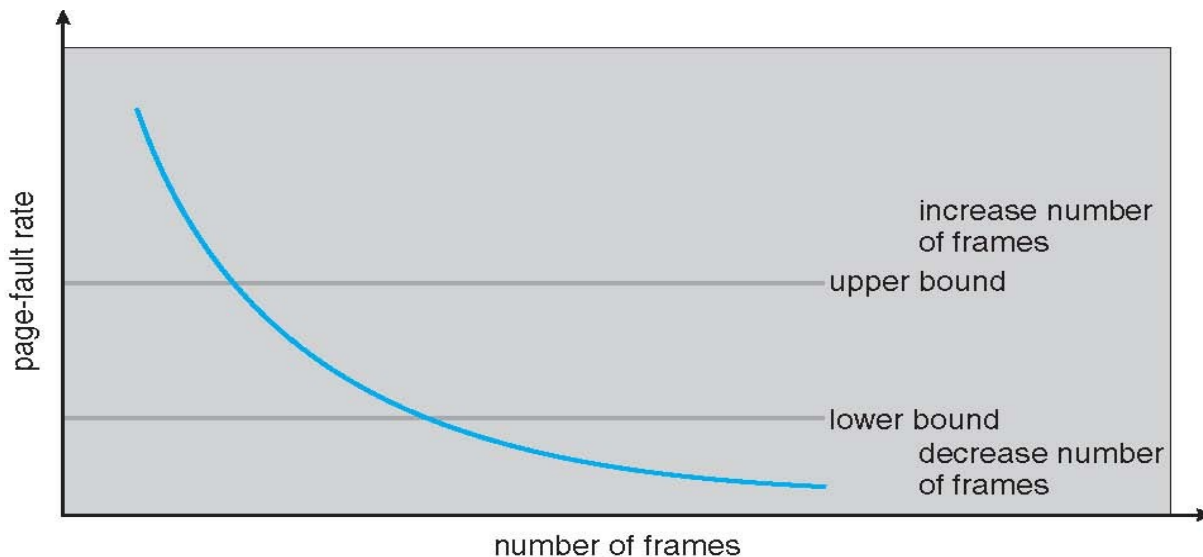
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

# Page fault frequency

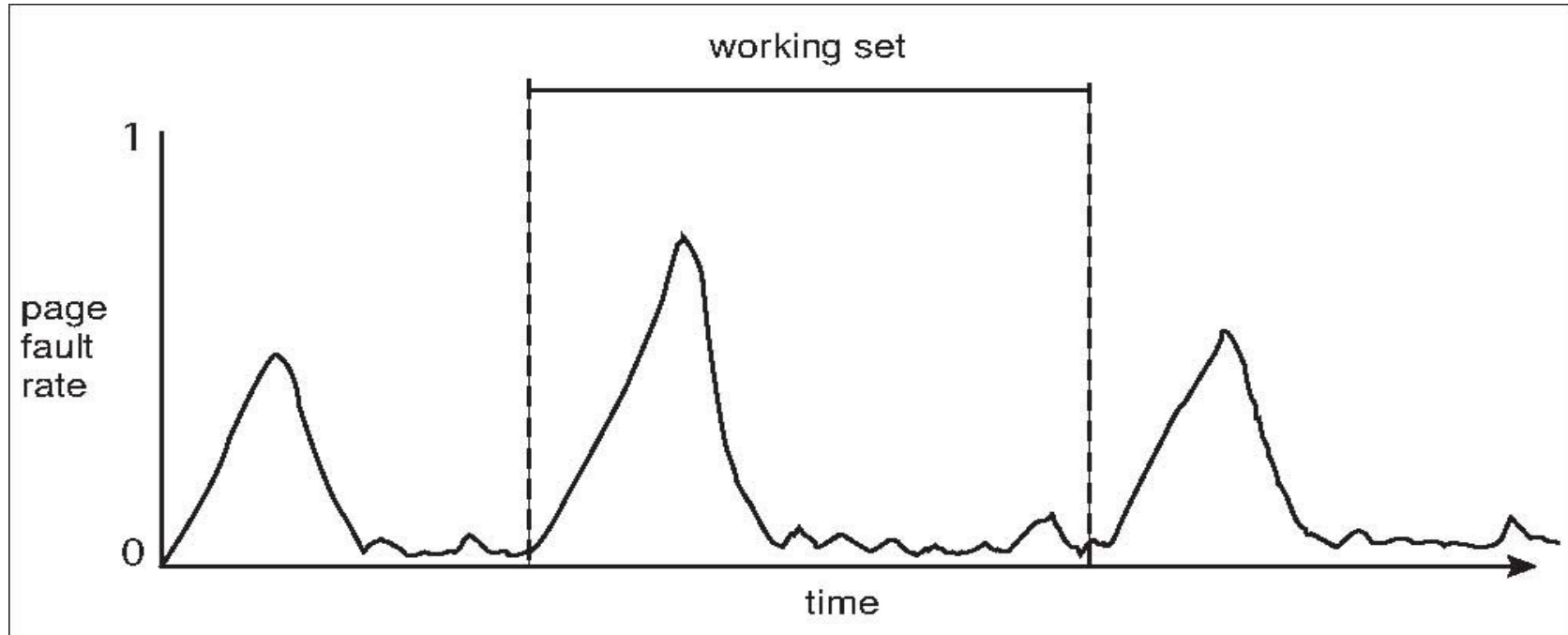
- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy

If actual rate too low, process loses frame

If actual rate too high, process gains frame



# Working Sets and Page Fault Rates





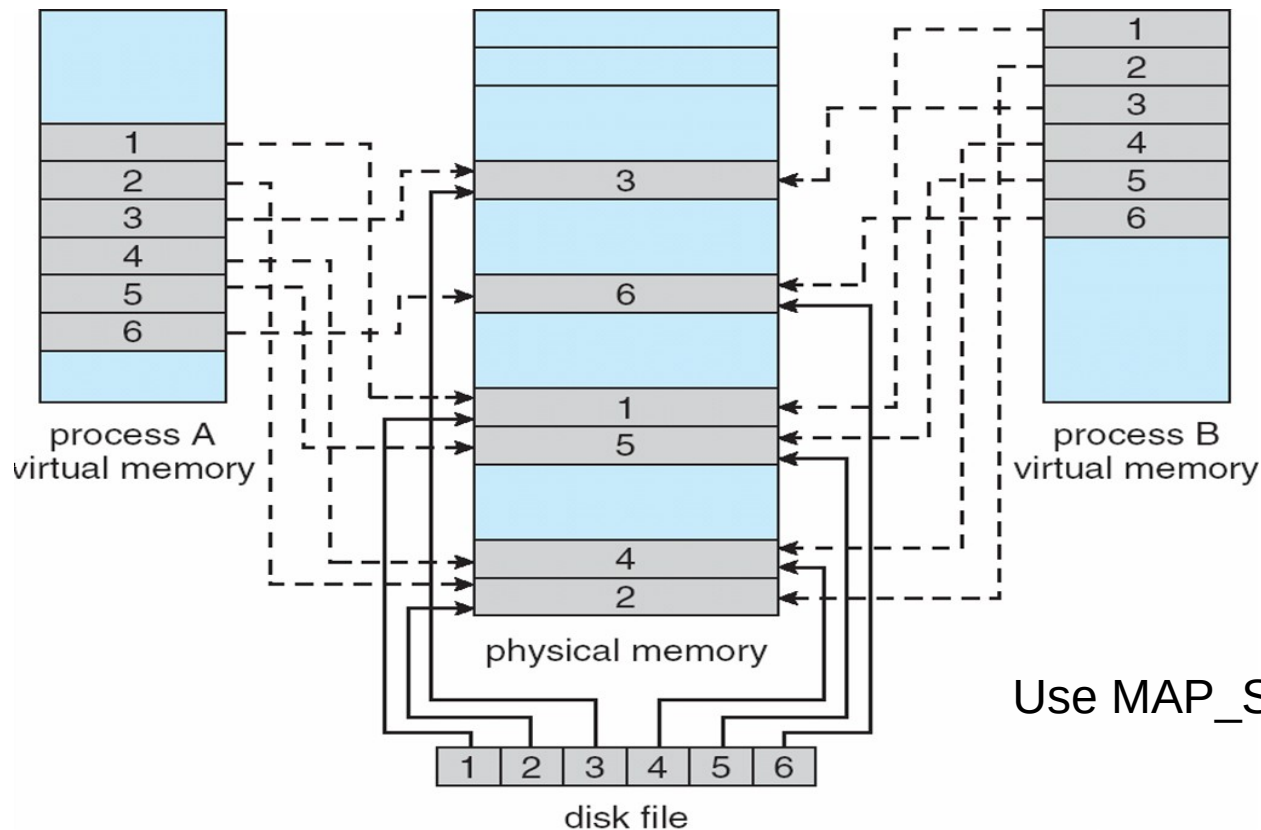
# Memory Mapped Files

# Memory-Mapped Files

- First, let's see a demo of using `mmap()`



# Memory-Mapped Files



# Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages



# Memory-Mapped Files

- Some OSes use memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call

Now file mapped into process address space

- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway

But map file into kernel address space

Process still does `read()` and `write()`

Copies data to and from kernel space and user space

Uses efficient memory management subsystem

Avoids needing separate subsystem

- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)



# **Allocating Kernel 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

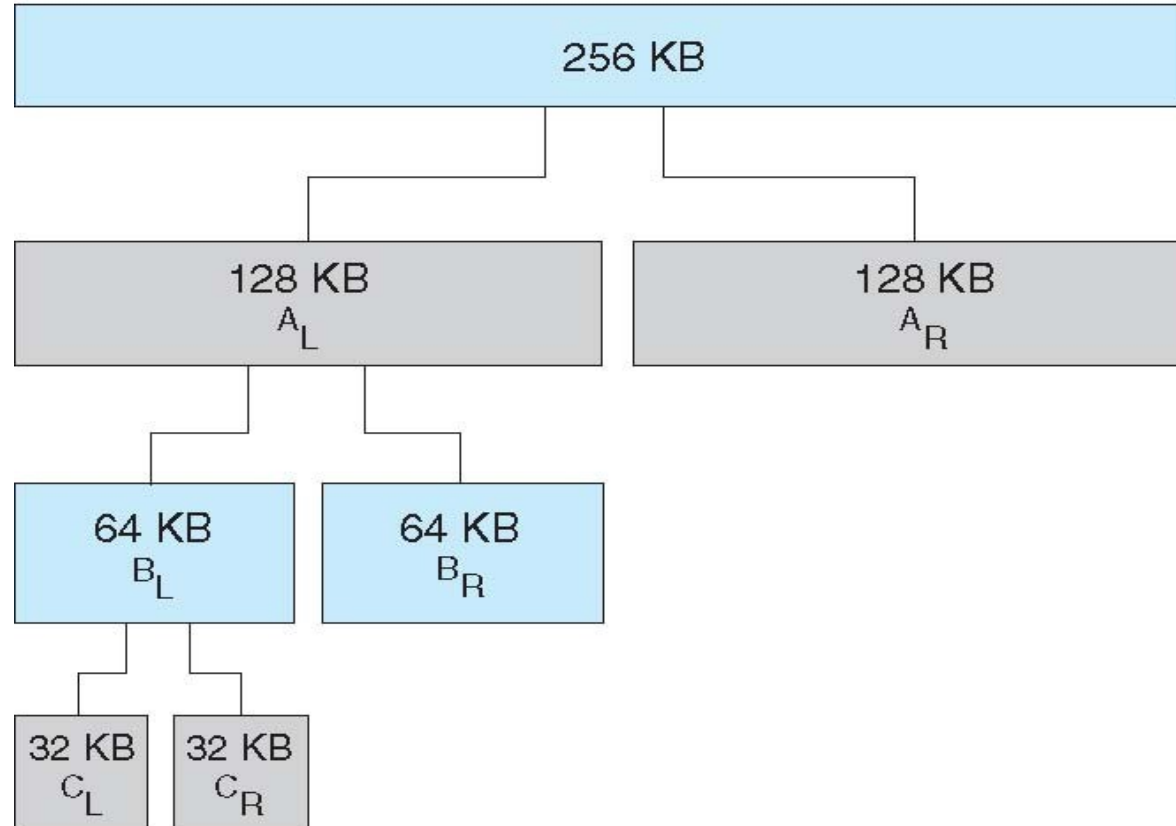
I.e. for device I/O



-

# Buddy Allocator

physically contiguous pages



# Buddy Allocator

- 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 Allocator

- Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB

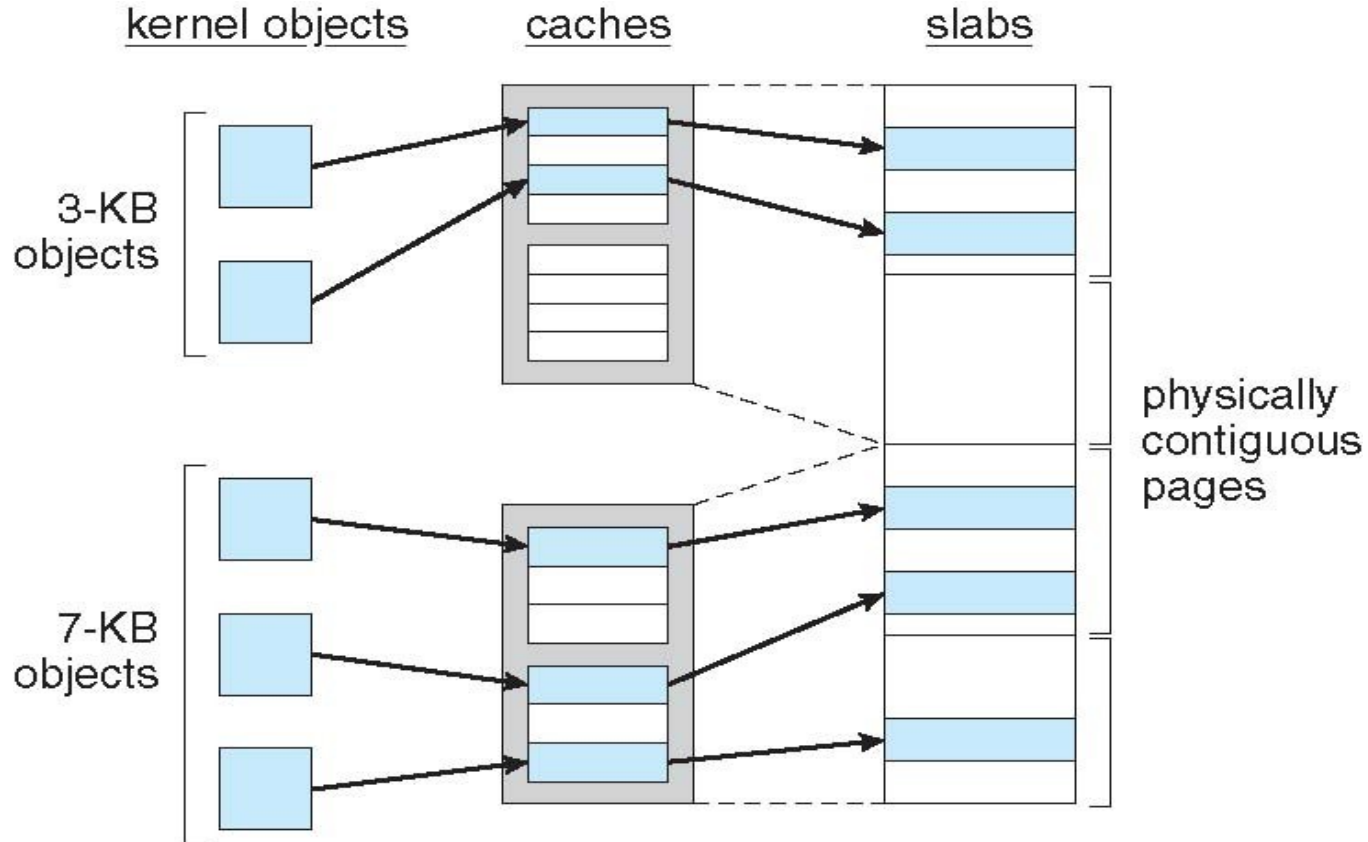
Split into AL and Ar of 128KB each

One further divided into BL and BR of 64KB

One further into CL and CR of 32KB each – one used to satisfy request

- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

# Slab Allocator



# 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





## **Other considerations**

# Other Considerations -- 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  $\alpha$  of the pages is used  
Is cost of  $s * \alpha$  save pages faults  $>$  or  $<$  than the cost of prepaging  
 $s * (1 - \alpha)$  unnecessary pages?  
 $\alpha$  near zero  $\longrightarrow$  prepaging loses

# 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

# TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- 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

# 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

# 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

