Virtual Memory Management

Professor Hugh C. Lauer
CS-3013, Operating Systems

Caching issues

From previous topic

- When to put something in the cache
- What to throw out to create cache space for new items
- How to keep cached item and stored item in sync after one or the other is updated
- How to keep multiple caches in sync across processors or machines
- Size of cache needed to be effective
- Size of cache items for efficiency
- …

Physical memory is a cache of virtual memory, so …

- **When to swap in a page**
  - On demand? or in anticipation?
- **What to throw out**
  - Page Replacement Policy
- **Keeping dirty pages in sync with disk**
  - Flushing strategy
- **Keeping pages in sync across processors or machines**
  - Defer to another time
- **Size of physical memory to be effective**
  - See previous discussion
- **Size of pages for efficiency**
  - One size fits all, or multiple sizes?

Physical memory as cache of virtual memory

- When to swap in a page
  - On demand? or in anticipation?
- **What to throw out**
  - Page Replacement Policy
- Keeping dirty pages in sync with disk
  - Flushing strategy
- Keeping pages in sync across processors or machines
  - Defer to another time
- Size of physical memory to be effective
  - See previous discussion
- Size of pages for efficiency
  - One size fits all, or multiple sizes?

Reading assignment

- **Tanenbaum:–**
  - §3.4 — Page Replacement Algorithms
  - §3.5 — Design Issues for Paging Systems

  - §3.6 — Implementation Issues
  - §3.7 — Segmentation

VM page replacement

- **If there is an unused frame, use it.**
- **If there are no unused frames available, select a *victim* (according to policy) and**
  - Invalidate its PTE and TLB entry
  - If it contains a dirty page (M == 1)
    - write it to disk
  - Load in new page from disk (or create new page)
  - Update the PTE and TLB entry!
  - Restart the faulting instruction
- **What is cost of replacing a page?**
- **How does the OS select the page to be evicted?**

Page replacement algorithms

- **Want lowest page-fault rate.**
- **Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.**
- ***Reference string* – ordered list of pages accessed as process executes**

`Ex. Reference String is A B C A B D A D B C B`

The best page to replace

- **The best page to replace is the one that will *never* be accessed again**

- **Optimal Algorithm – *Belady's Rule***
  - Lowest fault rate for any reference string
  - Basically, replace the page *that will not be used for the longest time in the future*.
  - Belady's Rule is a yardstick
  - We want to find close approximations

Some page replacement algorithms

- Not Recently Used
- First in, First out
- Second Chance
- Clock
- LRU (least recently used)
- Not Frequently Used
- Aging
- Working Set
- WSClock
- …

Typical page table entry

1	1	1	2	20
V	R	M	prot	page frame number

- ***Valid*** **bit gives state of this** ***Page Table Entry*** **(*PTE*)**
  - says whether or not its virtual address is valid – in memory and VA range
  - If not set, page might not be in memory or *may not even exist!*
- ***Reference*** **bit says whether the page has been accessed**
  - it is set by hardware *whenever* a page has been read or written to
- ***Modify*** **bit says whether or not the page is** ***dirty***
  - it is set by hardware during *every* write to the page
- ***Protection*** **bits control which operations are allowed**
  - read, write, execute, etc.
- ***Page*** **frame number (PFN) determines the physical page**
  - physical page start address
- ***Other*** **bits dependent upon machine architecture**

Page replacement – NRU (not recently used)

- **Periodically (e.g., on a clock interrupt)**
  - Clear *R* bit from all PTE's
- **When needed, rank order pages as follows**
  1. R = 0, M = 0
  2. R = 0, M = 1
  3. R = 1, M = 0
  4. R = 1, M = 1
- **Evict a page at random from lowest non-empty class**
  - Write out if *M* = 1; clear *M* when written
- **Characteristics**
  - Easy to understand and implement
  - Not optimal, but adequate in some cases

Page replacement – FIFO (*f*irst *in*, *f*irst *out*)

- **Easy to implement**
  - When swapping a page in, place its *page id* on end of list
  - Evict page at head of list
- **Page to be evicted has been in memory the longest time, but …**
  - Maybe it is being used, very active even
  - We just don't know
- **A weird phenomenon:– Belady's Anomaly**
  - fault rate may increase when there is <u>more</u> physical memory!

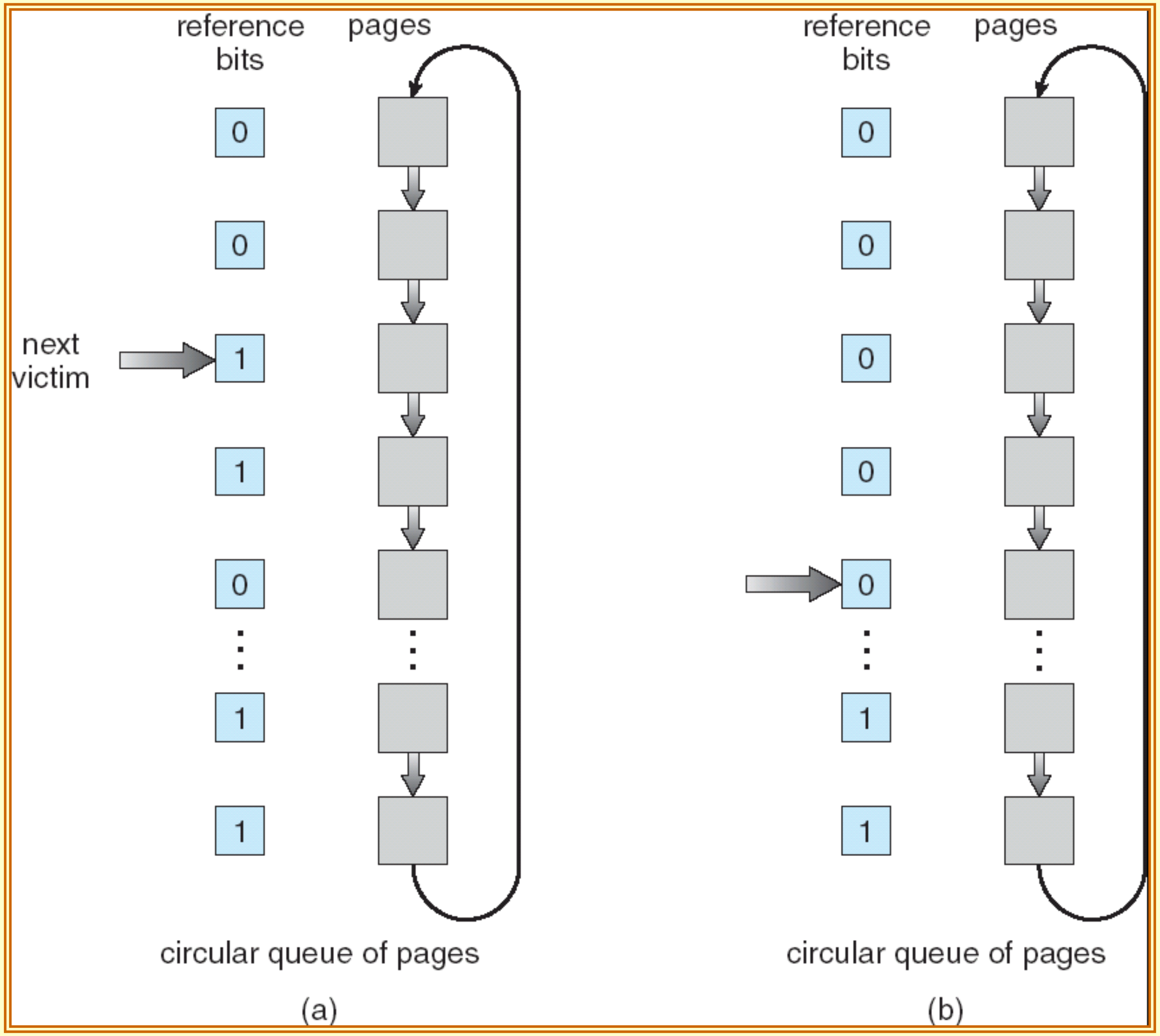
Second chance

- **Maintain FIFO page list**
- **When a page frame is needed, check reference bit of top page in list**
  - If R == 1 then move page to end of list and clear R, repeat
  - If R == 0 then evict page
- **I.e., a page has to move to top of list at least twice**
  - I.e., once *after* the last time R-bit was cleared
- **Disadvantage**
  - Moves pages around on list a lot (bookkeeping overhead)

Clock replacement (slight variation of second chance)

- **Create circular list of PTEs in FIFO Order**
- **One-handed *Clock* – pointer starts at oldest page**
  - Algorithm – FIFO, but check Reference bit
    - If R == 1, set R = 0 and advance hand
    - evict first page with R == 0
  - Looks like a clock hand sweeping PTE entries
  - Fast, but worst case may take a lot of time

Clock algorithm (illustrated)



Enhanced clock algorithm

- **Two-handed *clock* – add another hand that is *n* PTEs ahead**
  - Extra hand clears Reference bit
  - Allows very active pages to stay in longer
- **Also rank order the frames**
  1. R = 0, M = 0
  2. R = 0, M = 1
  3. R = 1, M = 0
  4. R = 1, M = 1

  Select first entry in lowest category
  - May require multiple passes
  - Gives preference to modified pages

Least recently used (LRU)

- **Replace the page that has not been used for the longest time**

On the assumption that it is *least likely* to be needed again soon

```
3 Page Frames   Reference String - A B C A B D A D B C
```

```
LRU – 5 faults
```

<u>**A**</u> <u>**B**</u> <u>**C**</u> **A** **B** <u>**D**</u> **A** **D** **B** <u>**C**</u>

LRU

- **Past experience may indicate future behavior**
- **Perfect LRU requires some form of timestamp to be associated with a PTE on every memory reference !!!**
- **Counter implementation**
  - Every page entry has a counter; each time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which to select
- **Stack implementation – keep a stack of page numbers in a double link form:**
  - Page referenced: move it to the top
  - No search for replacement

LRU approximations

- **Aging**
  - Keep a counter for each PTE
  - Periodically (clock interrupt) – check R-bit
    - If R = 0 increment counter (page has not been used)
    - If R = 1 clear the counter (page has been used)
    - Clear R = 0
  - Counter contains # of intervals since last access
  - Replace page having largest counter value
- **Alternatives**
  - §§3.4.6-3.4.7 in Tanenbaum

When to evict pages (cleaning policy)

I.e., a *kernel thread*

- **An OS thread called the *paging daemon***
  - wakes periodically to inspect pool of frames
  - if insufficient # of free frames
    - Mark pages for eviction according to policy, set valid bit to zero
    - Schedule disk to write dirty pages
  - on page fault
    - If desired page is *mark*ed but still in memory, use it
    - Otherwise, replace first clean marked page in pool
- **Advantage**
    - *Writing out dirty pages is not in critical path to swapping in*

Physical memory as cache of virtual memory

- **When to swap in a page**
  - On demand? or in anticipation?
- **What to throw out**
  - Page Replacement Policy
- **Keeping dirty pages in sync with disk**
  - Flushing strategy
- **Keeping pages in sync across processors or machines**
  - Defer to another time
- **Size of physical memory to be effective**
  - See previous discussion
- **Size of pages for efficiency**
  - One size fits all, or multiple sizes?

What to page in

- **Demand paging brings in the faulting page**
  - To bring in more pages, we need to know the future
- **Users don’t really know the future, but a few OSs have user-controlled pre-fetching**
- **In real systems,**
  - load the initial page
  - Start running
  - Some systems (e.g. Windows) will bring in additional neighboring pages (clustering)
- **Alternatively**
  - Figure out *working set* from previous activity
  - Page in entire working set of a swapped out process

Working set

- **A *working set* of a process is used to model the dynamic locality of its memory usage**
  - *Working set* = set of pages a process currently needs to execute without too many page faults
  - Denning in late 60's
- **Definition:**
  - WS(t,w) = set of pages referenced in the interval between time t-w and time t
    - t is time and w is working set window (measured in page refs)
    - Page is in working set only if it was referenced in last w references

Working set algorithm

- **$w \equiv$ working-set window $\equiv$ a fixed number of page references**
  Example: 10,000 – 2,000,000 instructions
- **WS_i (working set of Process P_i) =**
  set of pages referenced in the most recent w (varies in time)
  - if w too small will not encompass entire locality.
  - if w too large will encompass several localities.
  - as $w \Rightarrow \infty$, encompasses entire program.

Working set example

- **Assume 3 page frames**
- **Let interval be *w* = 5**
- **1 2 3 2 3 1 2 4 3 4 7 4 3 3 4 1 1 2 2 2 1**

  ***w*={1,2,3} *w*={3,4,7} *w*={1,2}**

  - if *w* too small, will not encompass locality
  - if *w* too large, will encompass several localities
  - if *w* ⇒ infinity, will encompass entire program
- **if *Total WS* > physical memory ⇒ *thrashing***
  - Need to free up some physical memory
  - E.g., suspend a process, swap all of its pages out

Working set page replacement

- **In practice, convert references into time**
  - E.g. 100ns/ref, 100,000 references ≅ 10msec
- **WS algorithm in practice**
  - On each clock tick, clear all R bits and record process virtual time *t*
  - When looking for eviction candidates, scan all pages of process in physical memory
    - If R == 1

      Store *t* in *LTU* (last time used) of PTE and clear R
    - If R == 0

      If (*t* – *LTU*) > *WS_Interval* (i.e., *w*), evict the page (because it is not in working set)
    - Else select page with the largest difference

See tanenbaum, §3.4.8

WSClock (combines Clock and WS algorithms)

- **WSClock**
  - Circular list of entries containing
    - R, M, *time of last use*
    - R and *time* are updated on each clock tick
  - Clock "hand" progresses around list
    - If R = 1, reset and update *time*
    - If R = 0, and if *age* > WS_interval, and if clean, then claim it.
    - If R = 0, and if *age* > WS_interval, and if dirty, then schedule a disk write
    - Step "hand" to next entry on list
- **Very common in practice**

See Tanenbaum, §3.4.9

Review of page replacement algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Tanenbaum, Fig 3-22

Virtual memory subsystem

- **All about managing the page cache in RAM of *virtual memory* …**
- **… which lives primarily on disk**
- **See also Chapters 15 and 16 of *Linux Kernel Development*, by Robert Love**
  - **Chapter 15:– *The Process Address Space***
  - **Chapter 15:– *The Page Cache and Page Writeback***

More on segmentation

- ***Paging* is (mostly) invisible to programmer, but *segmentation* is not**
  - Even paging with two-level page tables is invisible
- ***Segment:* an open-ended piece of VM**
  - Multics (H6000): 2^{18} segments of 64K words each
  - Pentium: 16K segments of 2^{30} bytes each
    - 8K *global* segments, plus 8K *local* segments per process
    - Each segment may be paged or not
    - Each segment assigned to one of four protection levels
- **Program consciously loads *segment descriptors* when accessing a new segment**
  - Only OS/2 used full power of Pentium segments
  - Linux concatenates 3 segments to simulate contiguous VM

***Aka* "flat" virtual memory**

VM summary

- **Memory Management – from simple multiprogramming support to efficient use of multiple system resources**
- **Models and measurement exist to determine the goodness of an implementation**
- **In real systems, must tradeoff**
  - Implementation complexity
  - Management overhead
  - Access time overhead

Virtual memory summary (continued)

- **When to swap in a page**
  - On demand? or in anticipation?
- **What to throw out**
  - Page Replacement Policy
- **Keeping dirty pages in sync with disk**
  - Flushing strategy
- **Keeping pages in sync across processors or machines**
  - Defer to another time
- **Size of physical memory to be effective**
  - See previous discussion
- **Size of pages for efficiency**
  - One size fits all, or multiple sizes?

Reading assignment

- **Tanenbaum**
  - §§ 3.1–3.3 (previous topics)
    - *Memory Management*
    - *Paging*
  - §§ 3.4–3.6 (this topic)
    - *Page Replacement Algorithms*
    - *Design Issues for Paging Systems*
    - *Implementation Issues for Paging Systems*
  - § 3.7
    - *More on Segmentation*

Questions?

OS design issue — where does kernel execute?

- **In physical memory**
  - Old systems (e.g., IBM 360/67)
  - Extra effort needed to look inside of VM of any process
- **In virtual memory**
  - Most modern systems
  - Shared *segment* among all processes
- **Advantages of kernel in virtual memory**
  - Easy to access, transfer to/from VM of any process
  - No context switch needed for traps, page faults
  - No context switch needed for purely kernel interrupts

Kernel Memory Requirements

- **Interrupt handlers**
  - Must be *pinned* into physical memory
  - At locations known to hardware
- **Critical kernel code**
  - *Pinned*, never swapped out
- **I/O buffers (user and kernel)**
  - Must be *pinned* and in *contiguous* physical memory
- **Kernel data** (… [illegible] le objects, semaphores, etc.)
  - *Pi*[illegible] memory
  - [illegible] y allocated & freed
  - [illegible] multiples of page size; fragmentation is an issue

Definition: *Pinned* – not subject to being swapped out!

Reason:– I/O and other devices don't recognize paging!

Kernel memory allocation

- **E.g., Linux PCB (`struct task_struct`)**
  - > 1.7 Kbytes each, pinned
  - Created on every **`fork`** and every thread create
    - **`clone()`**
  - deleted on every **`exit`**

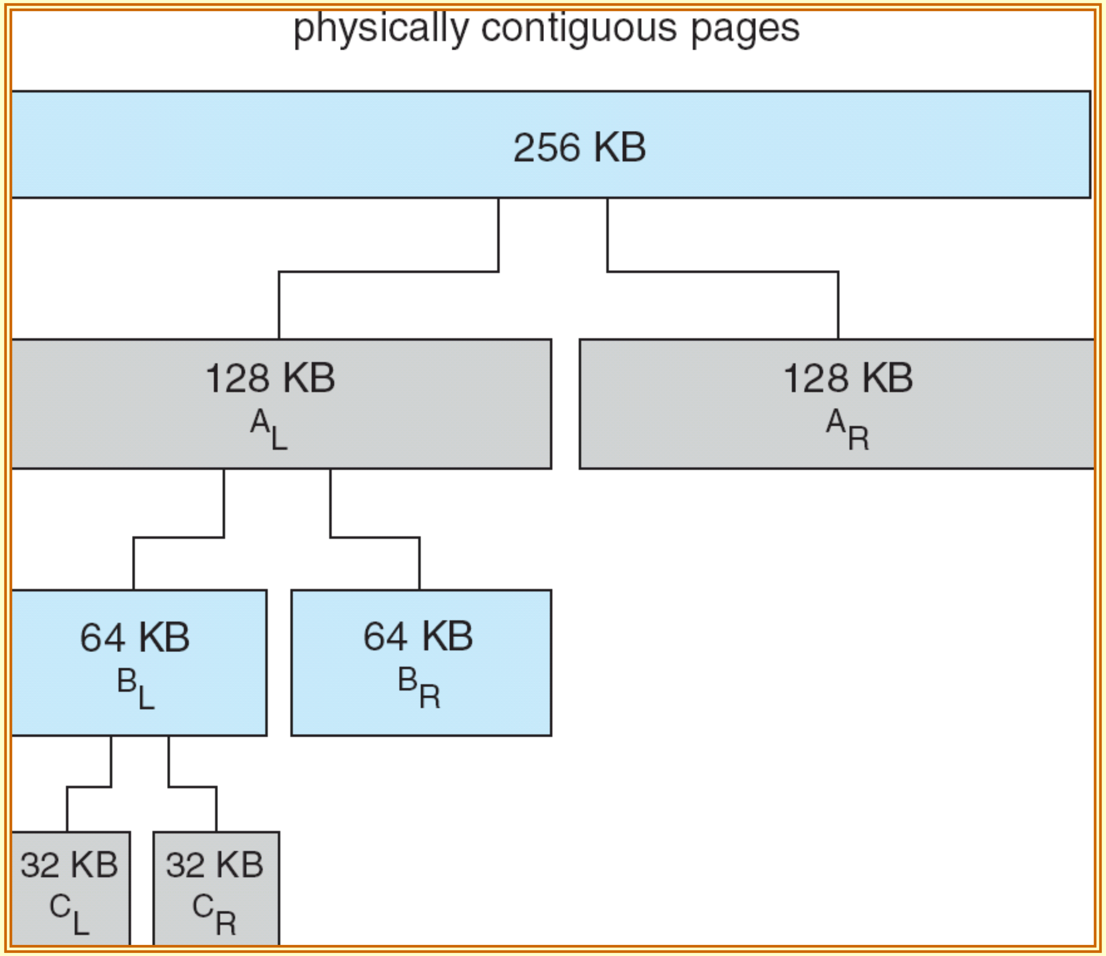
- **Kernel memory allocators**
  - **`kmalloc()`**
    - Very much like **`malloc()`**, but in kernel space
    - Subject to extreme fragmentation!
  - Buddy system
  - Slab allocation

Buddy system

- **Maintain a segment of contiguous pinned VM**
- **Round up each request to nearest power of 2**
- **Recursively divide a chunk of size 2^k into two "buddies" of size 2^{k-1} to reach desired size**
- **When freeing an object, recursively coalesce its block with adjacent free buddies**

- **Problem, still a lot of internal fragmentation**
  - E.g., 11 Kbyte page table requires 16 Kbytes

Buddy system (illustrated)

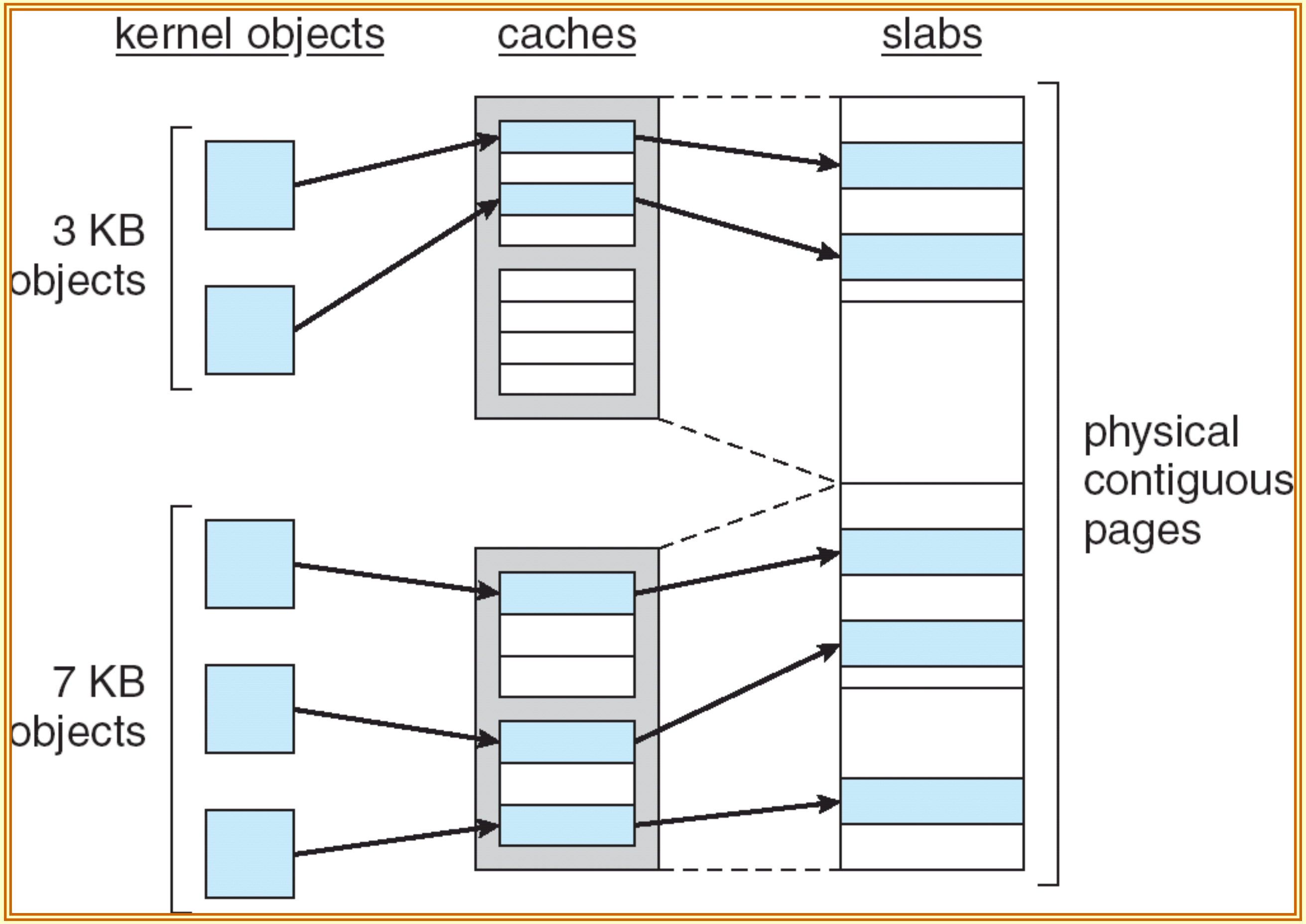


Slab allocation

Misuse of word cache

- **Maintain a separate "*cache*" for each major data type**
  - E.g., `task_struct`, `inode` in Linux
- ***Slab:* fixed number of contiguous physical pages assigned to one particular "cache"**
- **Upon kernel memory allocation request**
  - Recycle an existing object if possible
  - Allocate a new one within a slab if possible
  - Else, create an additional slab for that cache
- **When finished with an object**
  - Return it to "cache" for recycling
- **Benefits**
  - Minimize fragmentation of kernel memory
  - Most kernel memory requests can be satisfied quickly

Slab allocation (illustrated)



Note

- We use this allocation system in Project #4 (Linux kernel messaging system)

Classical Unix

- **Physical Memory**
  - Core map (pinned) – page frame info
  - Kernel (pinned) – rest of kernel
  - Frames – remainder of memory
- **Page replacement**
  - Page daemon
    - runs periodically to free up page frames
    - Global replacement – multiple parameters
    - Current BSD system uses 2-handed clock
  - Swapper – helps paging daemon
    - Look for processes idle 20 sec. or more and swap out longest idle
    - Next, swap out one of 4 largest – one in memory the longest
    - Check for processes to swap in

Linux VM

- **Kernel is pinned**
- **Rest of frames used**
  - Processes
  - Buffer cache
  - Page Cache
- **Multilevel paging**
  - 3 levels
  - Contiguous slab memory allocation using Buddy Algorithm
- **Replacement – goal keep a certain number of pages free**
  - Daemon (*kswapd*) runs once per second
    - Clock algorithm on page and buffer caches
    - Clock on unused shared pages
    - Modified clock (by VA order) on user processes (by # of frames)

From Robert Love, for Linux aficionados:–

- **Chapter 11:– Kernel memory mgmt.**
- **Chapters 12-13:– (about file systems)**
- **Chapter 14:– Process address space**
- **Chapter 15:– Page Cache and writeback**

Tanenbaum, §11.5, 11.6

Windows NT and successors

- Uses demand paging with *clustering*. Clustering brings in pages surrounding the faulting page.
- Processes are assigned *working set minimum* (20-50) and *working set maximum* (45-345)
- 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. (Balance set manager)
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Questions?