Working Set Algorithms Thrashing

Ken Birman

Reminders

- We've focused on *demand paging*
  - Each process is allocated Δ pages of memory
  - As a process executes it references pages
  - On a *miss* a *page fault* to the O/S occurs
  - The O/S pages in the missing page and pages out some target page if room is needed
- The CPU maintains a cache of PTEs: the TLB.
- The O/S must *flush* the TLB before looking at page reference bits, or before context switching (because this changes the page table)

Example

While "filling" memory we are likely to get a page fault on almost every reference. Usually we don't include these events when computing the hit ratio

R	3	7	9	5	3	6	3	5	6	7	9	7	9	3	8	6	3	6	8	3	5	6
S	3	7	9	5	3	6	3	5	6	7	9	7	9	3	8	6	3	6	8	3	5	6
	∅	3	7	9	5	3	6	3	5	6	7	9	7	9	3	8	6	3	6	8	3	5
	∅	∅	3	7	9	5	3	6	3	5	6	6	6	7	9	3	8	8	3	6	8	3
In	3	7	9	5	3	6	∅	∅	∅	7	9	∅	∅	3	8	6	∅	∅	∅	∅	5	6
Out	∅	∅	∅	3	7	9	∅	∅	∅	3	5	∅	∅	6	7	9	∅	∅	∅	∅	6	8

Hit ratio: 9/19 = 47%
Miss ratio: 10/19 = 53%

R(t): Page referenced at time t. S(t): Memory state when finished doing the paging at time t. In(t): Page brought in, if any. Out(t): Page sent out. ∅: None.

Thrashing

- Def: Excessive rate of paging that occurs because processes in system require more memory
  - Keep throwing out page that will be referenced soon
  - So, they keep accessing memory that is not there
- Why does it occur?
  - Poor locality, past != future
  - There is reuse, but process does not fit
  - Too many processes in the system

Approach 1: Working Set

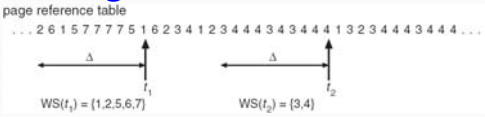
- Peter Denning, 1968
  - He uses this term to denote memory locality of a program

pages referenced by process in last Δ time-units comprise its working set

For our examples, we usually discuss WS in terms of Δ, a "window" *in the page reference string*. But while this is easier on paper it makes less sense in practice!

In real systems, the window should probably be a period of time, perhaps a second or two.

Working Sets



- The working set size is *num* pages in the working set
  - the number of pages touched in the interval [t-Δ+1..t].
- The working set size changes with program locality.
  - during periods of poor locality, you reference more pages.
  - Within that period of time, you will have a larger working set size.
- Goal: keep WS for each process in memory.

Theoretical aside

- Denning defined a policy called WSOpt
  - In this, the working set is computed over the next Δ references, not the last: R(t)..R(t+Δ-1)
- He compared WS with WSOpt.
  - WSOpt has knowledge of the future…
  - …yet even though WS is a practical algorithm with no ability to see the future, we can prove that the Hit and Miss ratio is identical for the two algorithms!

7

Key insight in proof

- Basic idea is to look at the paging decision made in WS at time t+Δ-1 and compare with the decision made by WSOpt at time t
- Both look at the same references… hence make same decision
  - Namely, WSOpt tosses out page R(t-1) if it isn't referenced "again" in time t..t+Δ-1
  - WS running at time t+Δ-1 tosses out page R(t-1) if it wasn't referenced in times t…t+Δ-1

8

Key step in proof

Page reference string

. . . 2 7 1 5 7 7 7 5 6 1 2 7 1 6 6 5 8 1 2 5 8 1 2 1 5 8 8 5 1 2 6 1 7 2 8 6 1 7 7 7 2 . . .

- At time t_1 resident page set contains { 1,2,5,7 }
  - WSOPT notices "5 will not be referenced in next 6 ime units, and pages 5 out.
  - WS will page 5 out too, but not until time t_{2m}

How do WSOpt and WS differ?

- WS maintains more pages in memory, because it needs Δ time "delay" to make a paging decision
  - In effect, it makes the same decisions, but it makes them after a time lag
  - Hence these pages hang around a bit longer

10

WSOPT and WS: same hit ratio!

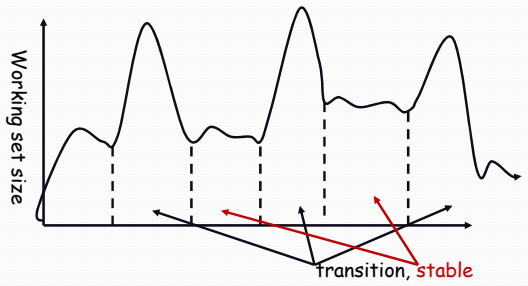
- WS is a little slower to remove pages from memory, but has the *identical pattern of paging in and paging out, just time-lagged by Δ time units*
- Thus WS and WSOPT have the identical hit and miss ratio… a rare case of a "real" algorithm that achieves seemingly optimal behavior

How do WS and LRU compare?

- Suppose we use the same value of Δ
  - WS removes pages if they aren't referenced and hence keeps less pages in memory
  - When it does page things out, it is using an LRU policy!
  - LRU will keep all Δ pages in memory, referenced or not
- Thus LRU often has a lower miss rate, but needs more memory than WS

12

Working Sets in the Real World



13

Working Set Approximation

- 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
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ⇒ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

14

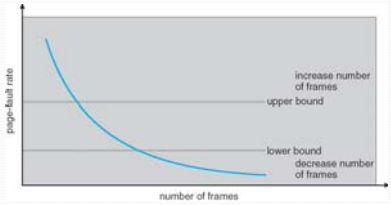
Using the Working Set

- Used mainly for prepaging
  - Pages in working set are a good approximation
- In Windows processes have a *max* and *min* WS size
  - At least *min* pages of the process are in memory
  - If > *max* pages in memory, on page fault a page is replaced
  - Else if memory is available, then WS is increased on page fault
  - The *max* WS can be specified by the application
  - The *max* is also modified then window is minimized!
    - Let's see the task manager

15

Approach 2: Page Fault Frequency

- thrashing viewed as poor ratio of fetch to work
- PFF = page faults / instructions executed
- if PFF rises above threshold, process needs more memory
  - not enough memory on the system? Swap out.
- if PFF sinks below threshold, memory can be taken away



16

OS and Paging

- Process Creation:
  - Allocate space and initialize page table for program and data
  - Allocate and initialize swap area
  - Info about PT and swap space is recorded in process table
- Process Execution
  - Reset MMU for new process
  - Flush the TLB
  - Bring processes' pages in memory
- Page Faults
- Process Termination
  - Release pages

17

Thrashing: Modern perspective

- With increasingly large physical memory sizes, some OS designers are questioning the value of demand paging
  - They agree that virtual memory is useful, because it lets us support program segmentation, memory-mapped files and sharing of dynamically linked libraries
  - But they worry that if a program starts to exceed its physical memory size, performance can collapse
  - Recall that the term for this is *thrashing*
- *Moral? Always check the page fault rate if your application gets very slow. VM size growing? Suspect a memory leak!*

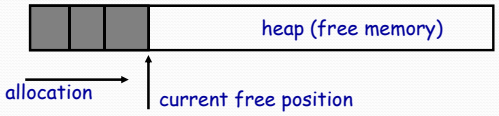
Dynamic Memory Management

- Notice that the O/S kernel can manage memory in a fairly trivial way:
  - All memory allocations are in units of "pages"
  - And pages can be anywhere in memory… so a simple free list is the only data structure needed
- But for variable-sized objects, we need a heap:
  - Used for all dynamic memory allocations
    - malloc/free in C, new/delete in C++, new/garbage collection in Java
  - Is a very large array allocated by OS, managed by program

19

Allocation and deallocation

- What happens when you call:
  - int *p = (int *)malloc(2500*sizeof(int));
    - Allocator slices a chunk of the heap and gives it to the program
  - free(p);
    - Deallocator will put back the allocated space to a free list
- Simplest implementation:
  - Allocation: increment pointer on every allocation
  - Deallocation: no-op
  - Problems: lots of fragmentation



20

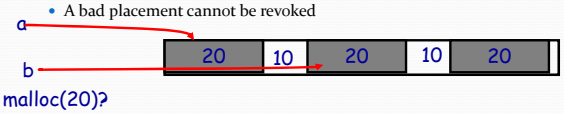
Memory allocation goals

- Minimize space
  - Should not waste space, minimize fragmentation
- Minimize time
  - As fast as possible, minimize system calls
- Maximizing locality
  - Minimize page faults cache misses
- And many more
- Proven: impossible to construct "always good" memory allocator

21

Memory Allocator

- What allocator has to do:
  - Maintain free list, and grant memory to requests
  - Ideal: no fragmentation and no wasted time
- What allocator cannot do:
  - Control order of memory requests and frees
  - A bad placement cannot be revoked



malloc(20)?

- Main challenge: avoid fragmentation

22

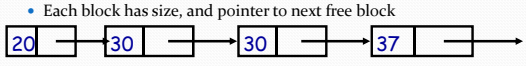
Impossibility Results

- Optimal memory allocation is NP-complete for general computation
- Given any allocation algorithm, ∃ streams of allocation and deallocation requests that defeat the allocator and cause extreme fragmentation

23

Best Fit Allocation

- Minimum size free block that can satisfy request
- Data structure:
  - List of free blocks
  - Each block has size, and pointer to next free block



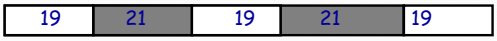
- Algorithm:
  - Scan list for the best fit

24

Best Fit gone wrong

- Simple bad case: allocate n, m (m<n) in alternating orders, free all the m's, then try to allocate an m+1.
- Example:
  - If we have 100 bytes of free memory
  - Request sequence: 19, 21, 19, 21, 19

  - Free sequence: 19, 19, 19



  - Wasted space: 57!

25

A simple scheme

- Each memory chunk has a signature before and after
  - Signature is an int
  - +ve implies the a free chunk
  - -ve implies that the chunk is currently in use
  - Magnitude of chunk is its size
- So, the smallest chunk is 3 elements:
  - One each for signature, and one for holding the data

26

Which chunk to allocate?

- Maintain a list of free chunks
  - Binning, doubly linked lists, etc
- Use best fit or any other strategy to determine page
  - For example: binning with best-fit
- What if allocated chunk is much bigger than request?
  - Internal fragmentation
  - Solution: split chunks
    - Will not split unless both chunks above a minimum size
- What if there is no big-enough free chunk?
  - sbrk or mmap
  - Possible page fault

27

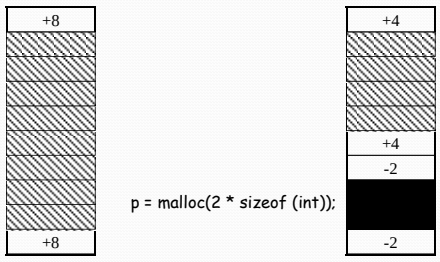
What happens on free?

- Identify size of chunk returned by user
- Change sign on both signatures (make +ve)
- Combine free adjacent chunks into bigger chunk
  - Worst case when there is one free chunk before and after
  - Recalculate size of new free chunk
  - Update the signatures
- Don't really need to erase old signatures

28

Example

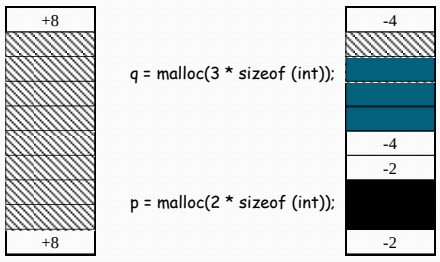
Initially one chunk, split and make signs negative on malloc



29

Example

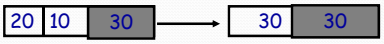
q gets 4 words, although it requested for 3



30

Design features

- Which free chunks should service request
  - Ideally avoid fragmentation… requires future knowledge
- Split free chunks to satisfy smaller requests
  - Avoids internal fragmentation
- Coalesce free blocks to form larger chunks
  - Avoids external fragmentation



31

Buddy-Block Scheme

- Invented by Donald Knuth, very simple
- Idea: Work with memory regions that are all powers of 2 times some "smallest" size
  - 2^k times b
- Round each request *up* to have form $b*2^k$

32

Buddy Block Scheme

- Keep a free list for each block size (each k)
  - When freeing an object, combine with adjacent free regions if this will result in a double-sized free object
- Basic actions on allocation request:
  - If request is a close fit to a region on the free list, allocate that region.
  - If request is less than half the size of a region on the free list, split the next larger size of region in half
  - If request is larger than *any* region, double the size of the heap (this puts a new larger object on the free list)

33

How to get more space?

- In Unix, system call sbrk()

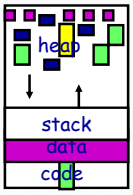
```
/* add nbytes of valid virtual address space */
void *get_free_space(unsigned nbytes) {
  void *p;
  if(!(p = sbrk(nbytes)))
       error("virtual memory exhausted");
  return p;
}
```

- Used by malloc if heap needs to be expanded
- Notice that heap only grows on "one side"

34

Malloc & OS memory management

- Relocation
  - OS allows easy relocation (change page table)
  - Placement decisions permanent at user level
- Size and distribution
  - OS: small number of large objects
  - Malloc: huge number of small objects



35