Thrashing and
Memory Management

Thrashing

- Processes in system require more memory than is there
  - Keep throwing out page that will be referenced soon
  - So, they keep accessing memory that is not there
- Why does it occur?
  - No good reuse, past != future
  - There is reuse, but process does not fit
  - Too many processes in the system

Approach 1: Working Set

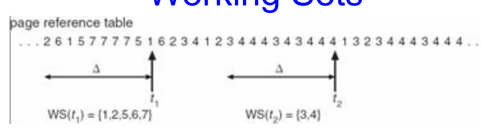
- Peter Denning, 1968
  - Defines the locality of a program

pages referenced by process in last T seconds of execution considered to comprise its working set

T: the working set parameter

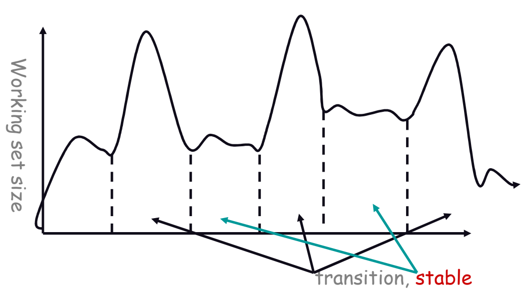
- Uses:
  - Caching: size of cache is size of WS
  - Scheduling: schedule process only if WS in memory
  - Page replacement: replace non-WS pages

Working Sets



- The <u>working set size</u> is *num* pages in the working set
  - the number of pages touched in the interval (t, t-Δ).
- 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.
- Don't run process unless working set is in memory.

Working Sets in the Real World



Working Set Approximation

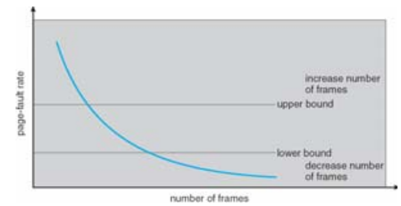
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ⇒ page in working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

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 (50)
  - If > *max* pages in memory, on page fault a page is replaced (345)
  - 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

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

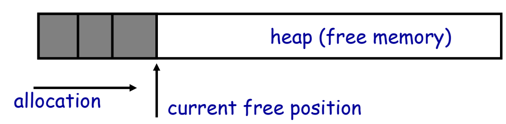


Dynamic Memory Management

- On loading a program, OS creates memory for process
  - Decides pages available for code, data, stack and heap
- Next, lets look at the 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

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



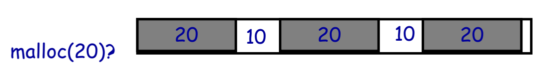
Memory allocation goals

- 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

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



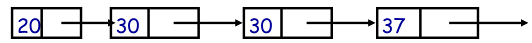
- Main challenge: avoid fragmentation

Impossibility Results

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

Best Fit Allocation

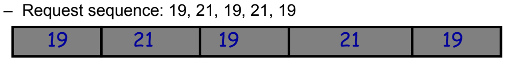
- Minimum size free block that can satisfy request
- Data structure:
  - List of free blocks
  - Each block has size, and pointer to next free block



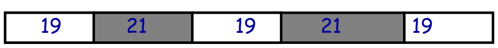
- Algorithm:
  - Scan list for the best fit

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!

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

Which chunk to allocate?

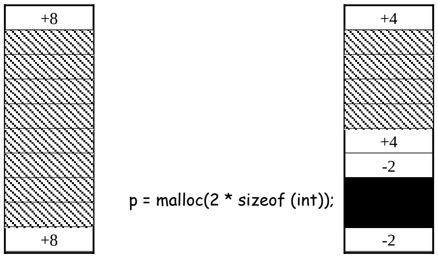
- Maintain a list of free chunks
  - Binning, doubly linked lists, etc
- Use best fit or any other strategy to determine page
  - For example: binning with best-fit
- What if allocated chunk is much bigger than request?
  - Internal fragmentation
  - Solution: split chunks
    - Will not split unless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

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

Example

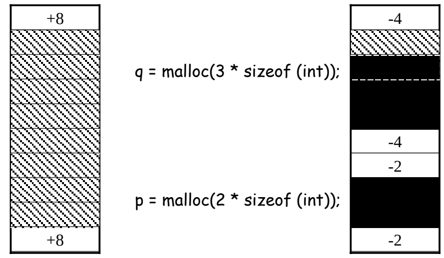
Initially one chunk, split and make signs negative on malloc



19

Example

q gets 4 words, although it requested for 3



20