

Segmentation and Fragmentation

Operating System Design – M1 Info

Instructor: Vincent Danjean

Class Assistant: Florent Bouchez

September 22, 2015

Outline

Segmentation

- Need for Virtual Memory

- 1st Attempt: Load Time Linking

- 2nd Attempt: Registers and MMU

- 3rd Attempt: Segmentation

Contiguous Memory Allocation: Handling Fragmentation

- Dynamic Memory Allocation. . .

- . . . A Lost Cause

- Common Strategies

- Slab Allocation

- Exploiting Patterns

- Clever Implementation Ideas

Recap

Outline

Segmentation

- Need for Virtual Memory

- 1st Attempt: Load Time Linking

- 2nd Attempt: Registers and MMU

- 3rd Attempt: Segmentation

Contiguous Memory Allocation: Handling Fragmentation

- Dynamic Memory Allocation. . .

- . . . A Lost Cause

- Common Strategies

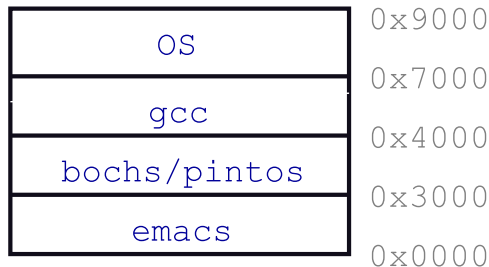
- Slab Allocation

- Exploiting Patterns

- Clever Implementation Ideas

Recap

Want processes to co-exist



- ▶ **Consider multiprogramming on physical memory**
 - ▶ What happens if pintos needs to expand?
 - ▶ If emacs needs more memory than is on the machine??
 - ▶ If pintos has an error and writes to address 0x7100?
 - ▶ When does gcc have to know it will run at 0x4000?
 - ▶ What if emacs isn't using its memory?

Issues in sharing physical memory

► Protection

- A bug in one process can corrupt memory in another
- Must somehow prevent process A from trashing B 's memory
- Also prevent A from even observing B 's memory (ssh-agent)

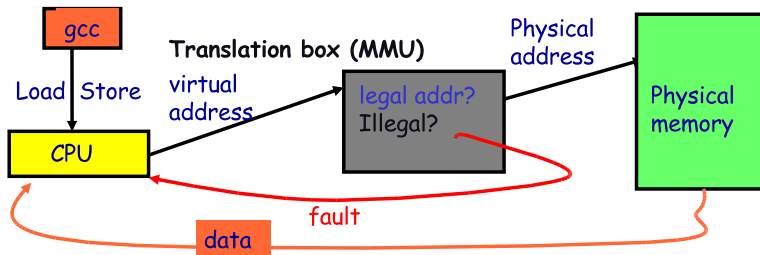
► Transparency

- A process shouldn't require particular memory locations
- Processes often require large amounts of contiguous memory (for stack, large data structures, etc.)

► Resource exhaustion

- Programmers typically assume machine has “enough” memory
- Sum of sizes of all processes often greater than physical memory

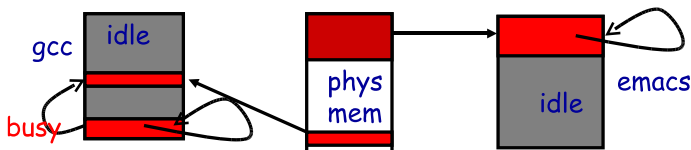
Virtual memory goals



- ▶ **Give each program its own “virtual” address space**
 - ▶ At run time, relocate each load and store to its actual memory
 - ▶ So app doesn't care what physical memory it's using
- ▶ **Also enforce protection**
 - ▶ Prevent one app from messing with another's memory
- ▶ **And allow programs to see more memory than exists**
 - ▶ Somehow relocate some memory accesses to disk

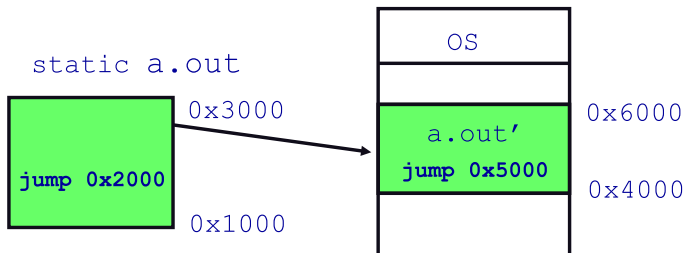
Virtual memory advantages

- ▶ **Can re-locate program while running**
 - ▶ Run partially in memory, partially on disk
- ▶ **Most of a process's memory will be idle (80/20 rule).**



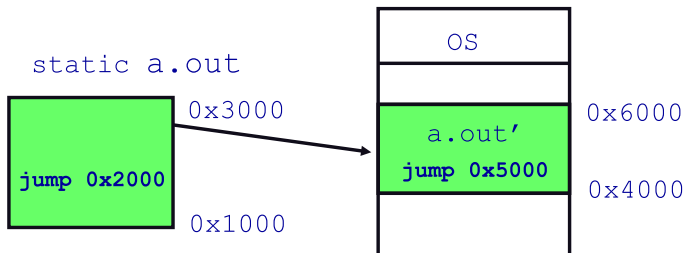
- ▶ Write idle parts to disk until needed
 - ▶ Let other processes use memory for idle part
 - ▶ Like CPU virtualization: when process not using CPU, switch. When not using a page, switch it to another process.
- ▶ **Challenge: VM = extra layer, could be slow**

Idea 1: load-time linking



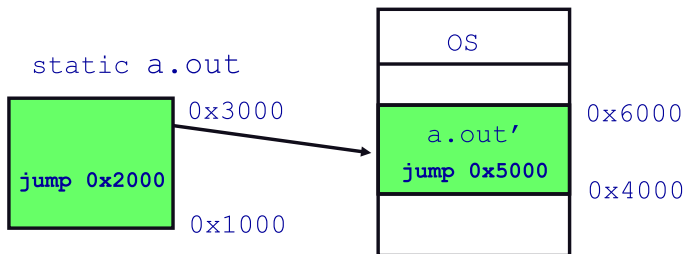
- ▶ **Link as usual, but keep the list of references**
- ▶ **Fix up process when actually executed**
 - ▶ Determine where process will reside in memory
 - ▶ Adjust all references within program (using addition)
- ▶ **Problems?**

Idea 1: load-time linking



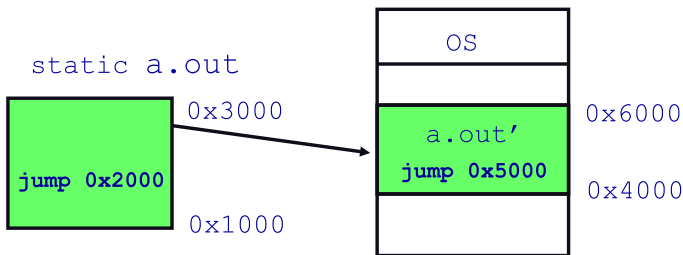
- ▶ **Link as usual, but keep the list of references**
- ▶ **Fix up process when actually executed**
 - ▶ Determine where process will reside in memory
 - ▶ Adjust all references within program (using addition)
- ▶ **Problems?**
 - ▶ How to enforce protection
 - ▶ How to move once in memory (Consider: data pointers)
 - ▶ What if no contiguous free region fits program?

Idea 2: base + bounds register



- ▶ Two special privileged registers: **base** and **bound**
- ▶ On each load/store:
 - ▶ Physical address = virtual address + **base**
 - ▶ Check $0 \leq \text{virtual address} < \text{bound}$, else trap to kernel
- ▶ How to move process in memory?
- ▶ What happens on context switch?

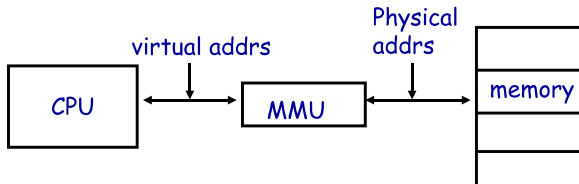
Idea 2: base + bounds register



- ▶ Two special privileged registers: **base** and **bound**
- ▶ On each load/store:
 - ▶ Physical address = virtual address + **base**
 - ▶ Check $0 \leq \text{virtual address} < \text{bound}$, else trap to kernel
- ▶ How to move process in memory?
 - ▶ Change **base** register
- ▶ What happens on context switch?
 - ▶ OS must re-load **base** and **bound** register

Definitions

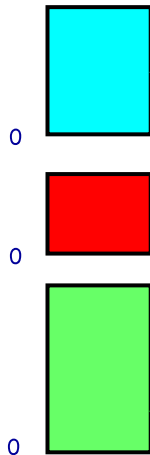
- ▶ Programs load/store to **virtual** (or **logical**) **addresses**
- ▶ Actual memory uses **physical** (or **real**) **addresses**
- ▶ Hardware has Memory Management Unit (**MMU**)



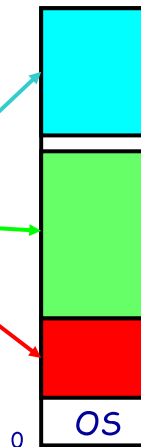
- ▶ Usually part of CPU
- ▶ Accessed w. privileged instructions (e.g., load bound reg)
- ▶ Translates from virtual to physical addresses
- ▶ Gives per-process view of memory called **address space**

Address space

Virtual Address View



Physical Address View



MMU

Base+bound trade-offs

► Advantages

- Cheap in terms of hardware: only two registers
- Cheap in terms of cycles: do add and compare in parallel
- Examples: Cray-1 used this scheme

► Disadvantages

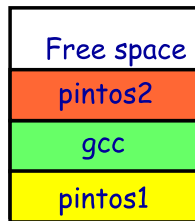
Base+bound trade-offs

► Advantages

- Cheap in terms of hardware: only two registers
- Cheap in terms of cycles: do add and compare in parallel
- Examples: Cray-1 used this scheme

► Disadvantages

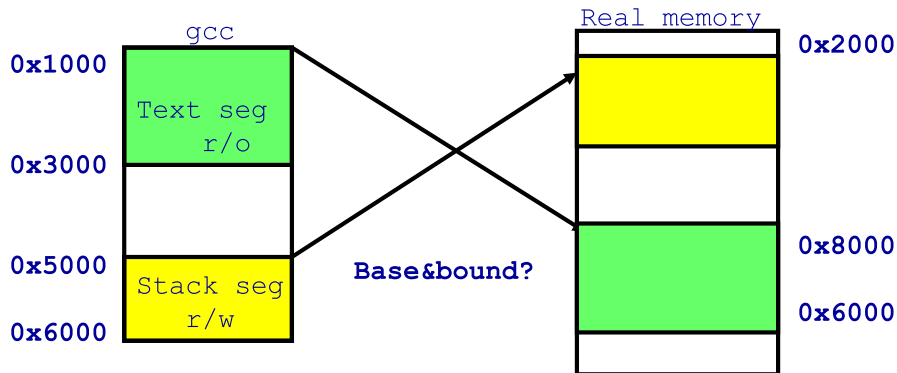
- Growing a process is expensive or impossible
- No way to share code or data (E.g., two copies of bochs, both running pintos)



► One solution: Multiple segments

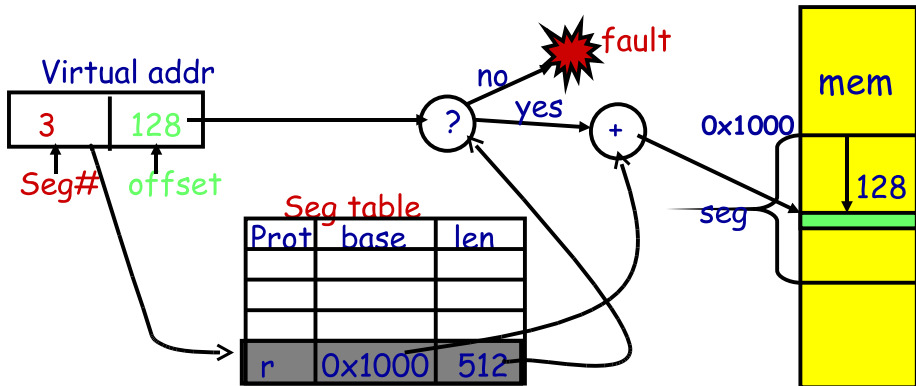
- E.g., separate code, stack, data segments
- Possibly multiple data segments

Segmentation



- ▶ **Let processes have many base/bounds regs**
 - ▶ Address space built from many segments
 - ▶ Can share/protect memory on segment granularity
- ▶ **Must specify segment as part of virtual address**

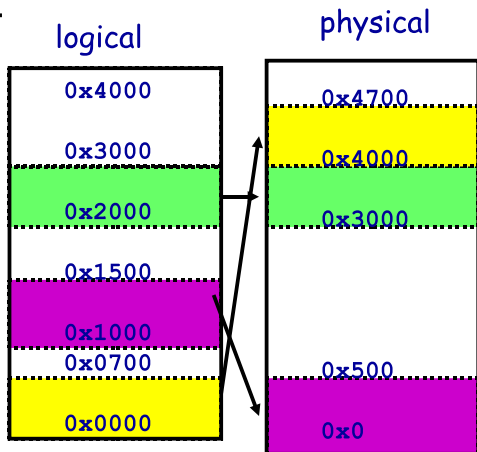
Segmentation mechanics



- ▶ Each process has a segment table
- ▶ Each VA indicates a segment and offset:
 - ▶ Top bits of addr select segment, low bits select offset (PDP-10)
 - ▶ Or segment selected by instruction or operand (means you need wider “far” pointers to specify segment)

Segmentation example

Seq	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00



- ▶ **2-bit segment number (1st digit), 12 bit offset (last 3)**
 - ▶ Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?

A process's view of the world

- ▶ **Each process has own view of machine**

- ▶ Its own address space

- ▶ `*(char *)0xc000` **different in P_1 & P_2**

- ▶ **Common memory segments:**

- ▶ **Stack:** automatically allocated / deallocated variables (r/w)

- ▶ **Heap:** manually allocated / deallocated data (r/w)

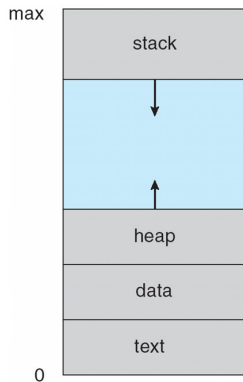
- ▶ **Data:** global variables and static variables (initialized) (r/w)

- ▶ **BSS:** statically-allocated variables (zeroed) (r/w)

- ▶ **Text:** code (r/o)

- ▶ **Greatly simplifies programming model**

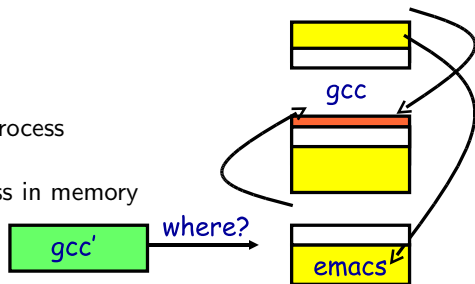
- ▶ gcc does not care that firefox is running
 - ▶ A bug in firefox does not matter to gcc



Segmentation trade-offs

► Advantages

- Multiple segments per process
- Allows sharing! (how?)
- Don't need entire process in memory

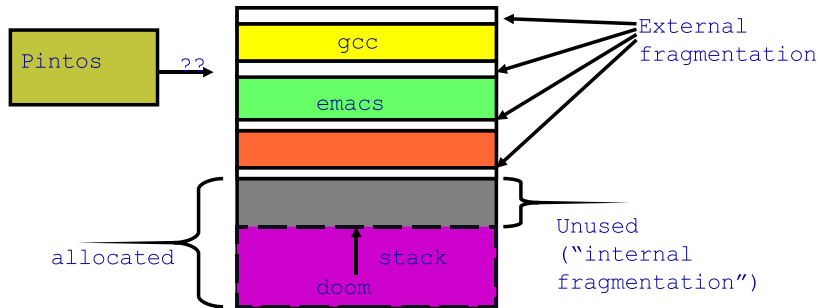


► Disadvantages

- Requires translation hardware, which could limit performance
- Segments not completely transparent to program (e.g., default segment faster or uses shorter instruction)
- n byte segment needs n contiguous bytes of physical memory
- Makes *fragmentation* a real problem.

Fragmentation

- ▶ **Fragmentation** \implies **Inability to use free memory**
- ▶ **Over time:**
 - ▶ Variable-sized pieces = many small holes (external frag.)
 - ▶ Fixed-sized pieces = no external holes, but force internal waste (internal fragmentation)



- ▶ In the next lecture, we will see a better solution to the virtual memory problem which does not suffer from fragmentation
- ▶ In the meantime, let's keep on focusing on fragmentation issues

Outline

Segmentation

- Need for Virtual Memory

- 1st Attempt: Load Time Linking

- 2nd Attempt: Registers and MMU

- 3rd Attempt: Segmentation

Contiguous Memory Allocation: Handling Fragmentation

- Dynamic Memory Allocation. . .

- . . . A Lost Cause

- Common Strategies

- Slab Allocation

- Exploiting Patterns

- Clever Implementation Ideas

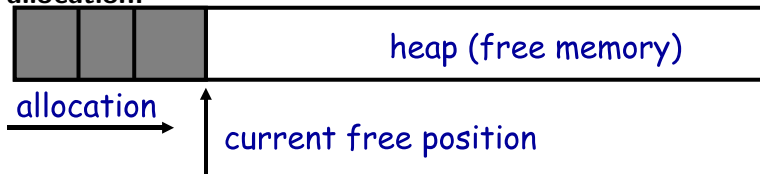
Recap

Dynamic memory allocation

- ▶ **Almost every useful program uses it**
 - ▶ Gives wonderful functionality benefits
 - ▶ Don't have to statically specify complex data structures
 - ▶ Can have data grow as a function of input size
 - ▶ Allows recursive procedures (stack growth)
 - ▶ But, can have a huge impact on performance
- ▶ **Today: how to implement it**
 - ▶ Lecture draws on [Wilson] (good survey from 1995)
- ▶ **Some interesting facts:**
 - ▶ Two or three line code change can have huge, non-obvious impact on how well allocator works (examples to come)
 - ▶ Proven: impossible to construct an "always good" allocator
 - ▶ Surprising result: after 35 years, memory management still poorly understood

Why is it hard?

- ▶ Satisfy arbitrary set of allocation and free's.
- ▶ Easy without free: set a pointer to the beginning of some big chunk of memory ("heap") and increment on each allocation:



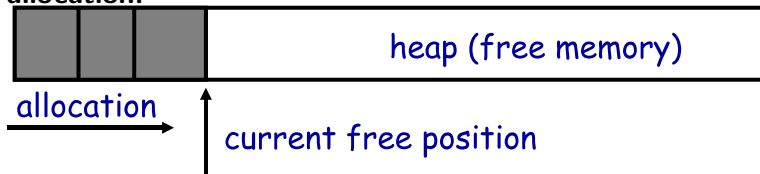
- ▶ Problem: free creates holes ("fragmentation") Result? Lots of free space but cannot satisfy request!



- ▶ Why can't we just stack everything to the left when needed ?

Why is it hard?

- ▶ Satisfy arbitrary set of allocation and free's.
- ▶ Easy without free: set a pointer to the beginning of some big chunk of memory ("heap") and increment on each allocation:



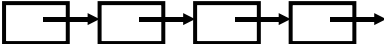
- ▶ Problem: free creates holes ("fragmentation") Result? Lots of free space but cannot satisfy request!

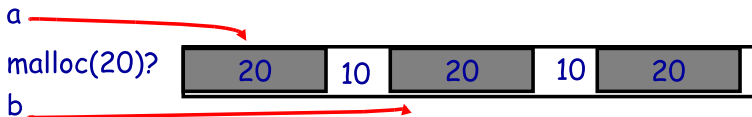


- ▶ Why can't we just stack everything to the left when needed ?
 - ▶ Requires to update memory references (and thus to know about the semantics of data)

More abstractly

freelist

- ▶ **What an allocator must do:** 
 - ▶ Track which parts of memory in use, which parts are free
 - ▶ Ideal: no wasted space, no time overhead
- ▶ **What the allocator cannot do:**
 - ▶ Control order of the number and size of requested blocks
 - ▶ Change user ptrs \Rightarrow (bad) placement decisions permanent



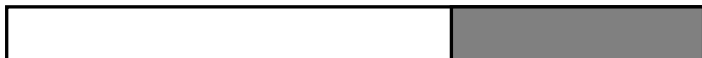
- ▶ **The core fight: minimize fragmentation**
 - ▶ App frees blocks in any order, creating holes in “heap”
 - ▶ Holes too small? cannot satisfy future requests

What is fragmentation really?

- ▶ **Inability to use memory that is free**
- ▶ **Two factors required for fragmentation**
 - ▶ Different lifetimes—if adjacent objects die at different times, then fragmentation:



- ▶ If they die at the same time, then no fragmentation:

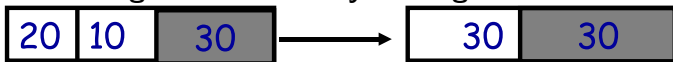


- ▶ Different sizes: If all requests the same size, then no fragmentation (as we will see later, paging relies on this to remove external fragmentation):



Important decisions

- ▶ **Placement choice: where in free memory to put a requested block?**
 - ▶ Freedom: can select any memory in the heap
 - ▶ Ideal: put block where it won't cause fragmentation later (impossible in general: requires future knowledge)
- ▶ **Split free blocks to satisfy smaller requests?**
 - ▶ Fights internal fragmentation
 - ▶ Freedom: can chose any larger block to split
 - ▶ One way: chose block with smallest remainder (best fit)
- ▶ **Coalescing free blocks to yield larger blocks**



- ▶ Freedom: when to coalesce (deferring can be good) fights external fragmentation

Impossible to “solve” fragmentation

- ▶ **If you read allocation papers to find the best allocator**
 - ▶ All discussions revolve around tradeoffs
 - ▶ The reason? There cannot be a best allocator
- ▶ **Theoretical result:**
 - ▶ For any possible allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation.
- ▶ **How much fragmentation should we tolerate?**
 - ▶ Let M = bytes of live data, n_{\min} = smallest allocation, n_{\max} = largest
 - How much gross memory required?
 - ▶ Bad allocator: $M \cdot (n_{\max}/n_{\min})$
(use maximum size for any size)
 - ▶ Good allocator: $\sim M \cdot \log(n_{\max}/n_{\min})$

Pathological examples

- ▶ **Given allocation of 7 20-byte chunks**



- ▶ What's a bad stream of frees and then allocates?

- ▶ **Given a 128-byte limit on malloced space**

- ▶ What's a really bad combination of mallocs & frees?

- ▶ **Next: two allocators (best fit, first fit) that, in practice, work pretty well**

- ▶ “pretty well” = $\sim 20\%$ fragmentation under many workloads

Pathological examples

- ▶ **Given allocation of 7 20-byte chunks**

20	20	20	20	20	20	20
----	----	----	----	----	----	----

- ▶ What's a bad stream of frees and then allocates?
 - ▶ Free every one chunk out of two, then alloc 21 bytes
 - ▶ **Given a 128-byte limit on malloced space**
 - ▶ What's a really bad combination of mallocs & frees?
-
- ▶ **Next: two allocators (best fit, first fit) that, in practice, work pretty well**
 - ▶ “pretty well” = $\sim 20\%$ fragmentation under many workloads

Pathological examples

- ▶ **Given allocation of 7 20-byte chunks**

20	20	20	20	20	20	20
----	----	----	----	----	----	----

- ▶ What's a bad stream of frees and then allocates?
- ▶ Free every one chunk out of two, then alloc 21 bytes

- ▶ **Given a 128-byte limit on malloced space**

- ▶ What's a really bad combination of mallocs & frees?
- ▶ Malloc 128 1-byte chunks, free every chunk but the first and the 64th
- ▶ Try to malloc 64 bytes.

An adversary should work backward (come up with a bad situation and figure out how to force the allocator into this situation)

- ▶ **Next: two allocators (best fit, first fit) that, in practice, work pretty well**

- ▶ “pretty well” = $\sim 20\%$ fragmentation under many workloads

Illustrating Fragmentation

Source:

<http://blog.pavlov.net/2007/11/10/memory-fragmentation/>

Point 1	Point 2	Point 3	Point 4
35MB	118MB	94MB	88MB

1. Start browser.
2. Measure memory usage (Point 1).
3. Load a URL that in turn opens many windows. Wait for them to finish loading.
4. Measure memory usage (Point 2).
5. Close them all down, and go back to the blank start page.
6. Measure memory usage again (Point 3).
7. Force the caches to clear, to eliminate them from the experiment.
8. Measure memory usage again (Point 4).

Illustrating Fragmentation

Source:

<http://blog.pavlov.net/2007/11/10/memory-fragmentation/>

1. Start browser.
2. Measure memory usage (Point 1).
3. Load a URL that in turn opens many windows. Wait for them to finish loading.
4. Measure memory usage (Point 2).
5. Close them all down, and go back to the blank start page.
6. Measure memory usage again (Point 3).
7. Force the caches to clear, to eliminate them from the experiment.
8. Measure memory usage again (Point 4).

Point 1	Point 2	Point 3	Point 4
35MB	118MB	94MB	88MB



Illustrating Fragmentation

Source:

<http://blog.pavlov.net/2007/11/10/memory-fragmentation/>

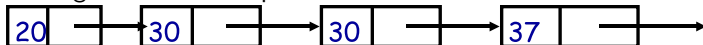
1. Start browser.
2. Measure memory usage (Point 1).
3. Load a URL that in turn opens many windows. Wait for them to finish loading.
4. Measure memory usage (Point 2).
5. Close them all down, and go back to the blank start page.
6. Measure memory usage again (Point 3).
7. Force the caches to clear, to eliminate them from the experiment.
8. Measure memory usage again (Point 4).

Point 1	Point 2	Point 3	Point 4
35MB	118MB	94MB	88MB



- ▶ **Strategy: minimize fragmentation by allocating space from block that leaves smallest fragment**

- ▶ Data structure: heap is a list of free blocks, each has a header holding block size and pointers to next



- ▶ Code: Search freelist for block closest in size to the request. (Exact match is ideal)
- ▶ During free (usually) coalesce adjacent blocks

- ▶ **Problem: Sawdust**

- ▶ Remainder so small that over time left with “sawdust” everywhere
- ▶ Fortunately not a problem in practice
- ▶ Implementation (go through the whole list ? maintain sorted list ? ...)

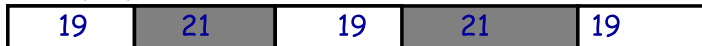
Best fit gone wrong

- ▶ **Simple bad case: allocate n , m ($n < m$) in alternating orders, free all the n s, then try to allocate an $n + 1$**
- ▶ **Example: start with 100 bytes of memory**

- ▶ alloc 19, 21, 19, 21, 19



- ▶ free 19, 19, 19:



- ▶ alloc 20? Fails! (wasted space = 57 bytes)
- ▶ **However, doesn't seem to happen in practice (though the way real programs behave suggest it easily could)**

First fit

- ▶ **Strategy: pick the first block that fits**
 - ▶ Data structure: free list, sorted lifo, fifo, or by address
 - ▶ Code: scan list, take the first one
- ▶ **LIFO: put free object on front of list.**
 - ▶ Simple, but causes higher fragmentation
 - ▶ Potentially good for cache locality
- ▶ **Address sort: order free blocks by address**
 - ▶ Makes coalescing easy (just check if next block is free)
 - ▶ Also preserves empty/idle space (locality good when paging)
- ▶ **FIFO: put free object at end of list**
 - ▶ Gives similar fragmentation as address sort, but unclear why

Subtle pathology: LIFO FF

- ▶ **Storage management example of subtle impact of simple decisions**
- ▶ **LIFO first fit seems good:**
 - ▶ Put object on front of list (cheap), hope same size used again (cheap + good locality)
- ▶ **But, has big problems for simple allocation patterns:**
 - ▶ E.g., repeatedly intermix short-lived $2n$ -byte allocations, with long-lived $(n + 1)$ -byte allocations
 - ▶ Each time large object freed, a small chunk will be quickly taken, leaving useless fragment. Pathological fragmentation

First fit: Nuances

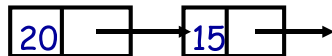
- ▶ **First fit sorted by address order, in practice:**

- ▶ Blocks at front preferentially split, ones at back only split when no larger one found before them
- ▶ Result? Seems to roughly sort free list by size
- ▶ So? Makes first fit operationally similar to best fit: a first fit of a sorted list = best fit!

- ▶ **Problem: sawdust at beginning of the list**

- ▶ Sorting of list forces a large requests to skip over many small blocks. Need to use a scalable heap organization

- ▶ **Suppose memory has free blocks:**



- ▶ If allocation ops are 10 then 20, best fit wins
- ▶ When is FF better than best fit?

First fit: Nuances

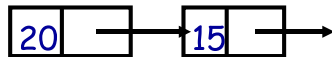
- ▶ **First fit sorted by address order, in practice:**

- ▶ Blocks at front preferentially split, ones at back only split when no larger one found before them
- ▶ Result? Seems to roughly sort free list by size
- ▶ So? Makes first fit operationally similar to best fit: a first fit of a sorted list = best fit!

- ▶ **Problem: sawdust at beginning of the list**

- ▶ Sorting of list forces a large requests to skip over many small blocks. Need to use a scalable heap organization

- ▶ **Suppose memory has free blocks:**



- ▶ If allocation ops are 10 then 20, best fit wins
- ▶ When is FF better than best fit?
- ▶ Suppose allocation ops are 8, 12, then 12 \implies first fit wins

First/best fit: weird parallels

- ▶ **Both seem to perform roughly equivalently**
- ▶ **In fact the placement decisions of both are roughly identical under both randomized and real workloads!**
 - ▶ No one knows why
 - ▶ Pretty strange since they seem pretty different
- ▶ **Possible explanations:**
 - ▶ Over time FF's free list becomes sorted by size: the beginning of the free list accumulates small objects and so fits tend to be close to best
 - ▶ Both have implicit “open space heuristic” try not to cut into large open spaces: large blocks at end only used when have to be (e.g., first fit skips over all smaller blocks)

Some worse ideas

- ▶ **Worst-fit:**

- ▶ Strategy: fight against sawdust by splitting blocks to maximize leftover size
- ▶ In real life seems to ensure that no large blocks around

- ▶ **Next fit:**

- ▶ Strategy: use first fit, but remember where we found the last thing and start searching from there
- ▶ Seems like a good idea, but tends to break down entire list

- ▶ **Buddy systems:**

- ▶ Round up allocations to power of 2 to make management faster
- ▶ Result? Heavy internal fragmentation

Slab allocation [Bonwick] (When program “bypass” malloc/free)

Remember what we told earlier:

If all requests the same size, then no fragmentation

- ▶ **Kernel allocates many instances of same structures**
 - ▶ E.g., a 1.7 KB `task_struct` for every process on system
- ▶ **Often want contiguous physical memory (for DMA)**
- ▶ **Slab allocation optimizes for this case:**
 - ▶ A **slab** is multiple pages of contiguous physical memory
 - ▶ A **cache** contains one or more slabs
 - ▶ Each cache stores only one kind of object (fixed size)
- ▶ **Each slab is full, empty, or partial**
- ▶ **E.g., need new `task_struct`?**
 - ▶ Look in the `task_struct` cache
 - ▶ If there is a partial slab, pick free `task_struct` in that
 - ▶ Else, use empty, or may need to allocate new slab for cache
- ▶ **Advantages: speed, and no internal fragmentation**

Performance Evaluation Through Randomized Workload

- ▶ **Size and lifetime distribution:**

- ▶ Exponential
 - ▶ typical programs allocate more small than big chunks and more short-lived than long-lived
 - ▶ mathematically tractable
- ▶ Spiky (out of hat)
- ▶ Modeled using statistics from real programs

Performance Evaluation Through Randomized Workload

- ▶ **Size and lifetime distribution:**
 - ▶ Exponential
 - ▶ typical programs allocate more small than big chunks and more short-lived than long-lived
 - ▶ mathematically tractable
 - ▶ Spiky (out of hat)
 - ▶ Modeled using statistics from real programs
- ▶ **Assume size and lifetime are independant ? 😞**
- ▶ **Correlation between consecutive events ? 😞**

Performance Evaluation Through Randomized Workload

- ▶ **Size and lifetime distribution:**

- ▶ Exponential
 - ▶ typical programs allocate more small than big chunks and more short-lived than long-lived
 - ▶ mathematically tractable
- ▶ Spiky (out of hat)
- ▶ Modeled using statistics from real programs

- ▶ **Assume size and lifetime are independent ? ☹️**

- ▶ **Correlation between consecutive events ? ☹️**

- ▶ **Actually, programs have patterns. Exponential behavior is often an artifact ☹️**

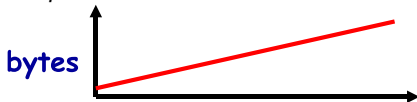
Analysis with exponentially distributed workload may lead to conclusions about allocator completely different from what happens in reality

- ▶ **Even Markov chains or other more advanced modeling are unable to accurately reflect real memory patterns**

Known patterns of real programs

- ▶ So far we've treated programs as black boxes.
- ▶ Most real programs exhibit 1 or 2 (or all 3) of the following patterns of alloc/dealloc:

- ▶ *Ramps*: accumulate data monotonically over time



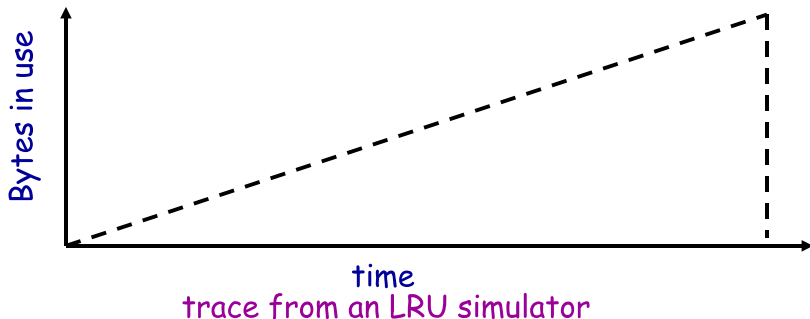
- ▶ *Peaks*: allocate many objects, use briefly, then free all



- ▶ *Plateaus*: allocate many objects, use for a long time

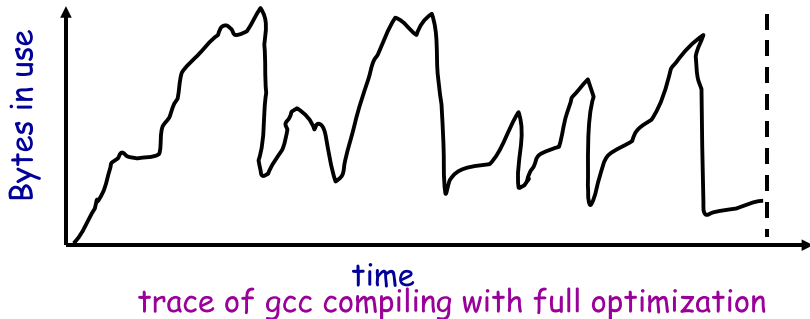


Pattern 1: ramps



- ▶ **In a practical sense: ramp = no free!**
 - ▶ Implication for fragmentation?
 - ▶ What happens if you evaluate allocator with ramp programs only?

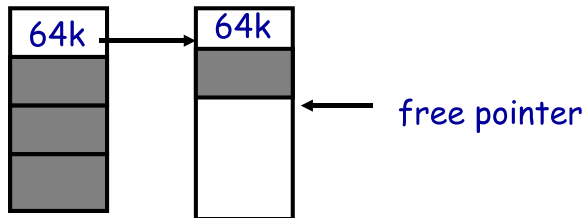
Pattern 2: peaks



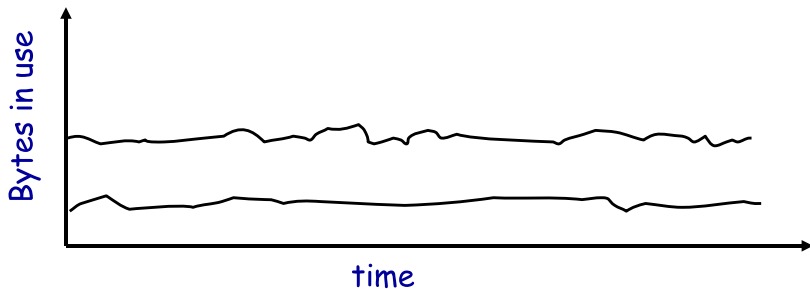
- ▶ **Peaks: allocate many objects, use briefly, then free all**
 - ▶ Surviving data are likely to be of different types
 - ▶ Fragmentation a real danger
 - ▶ What happens if peak allocated from contiguous memory?
 - ▶ Interleave peak & ramp? Interleave two different peaks?

Exploiting peaks

- ▶ **Peak phases: alloc a lot, then free everything**
 - ▶ So have new allocation interface: alloc as before, but only support free of everything
 - ▶ Called “arena allocation”, “obstack” (object stack), or `alloca/procedure` call (by compiler people)
- ▶ **Arena = a linked list of large chunks of memory**
 - ▶ Advantages: alloc is a pointer increment, free is “free”
No wasted space for tags or list pointers



Pattern 3: Plateaus



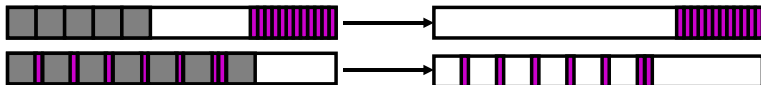
trace of perl running a string processing script

- ▶ **Plateaus:** allocate many objects, use for a long time
 - ▶ What happens if overlap with peak or different plateau?

Exploit ordering and size dependencies

► Segregation = reduced fragmentation:

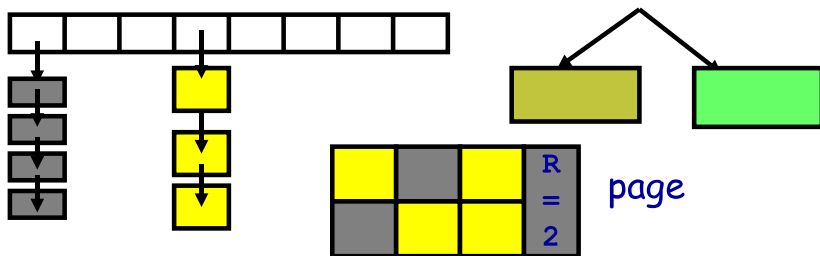
- Allocated at same time ~ freed at same time
- Different type ~ freed at different time



► Implementation observations:

- Programs allocate small number of different sizes
- Fragmentation at peak use more important than at low
- Most allocations small (< 10 words)
- Work done with allocated memory increases with size
- Implications?

Simple, fast segregated free lists

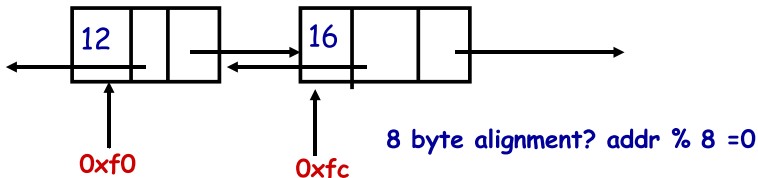


- ▶ **Array of free lists for small sizes, tree for larger**
 - ▶ Place blocks of same size on same “page”
 - ▶ Have count of allocated blocks: if goes to zero, can return page
- ▶ **Pro: segregate sizes, no size tag, fast small alloc**
- ▶ **Con: worst case waste: 1 page per size even w/o free, after pessimal free waste 1 page per object**

Typical space overheads

- ▶ **Free list bookkeeping + alignment determine minimum allocatable size:**

- ▶ Store size of block
- ▶ Pointers to next and previous freelist element



- ▶ Machine enforced overhead: alignment. Allocator doesn't know type. Must align memory to conservative boundary
- ▶ Minimum allocation unit? Space overhead when allocated?

Outline

Segmentation

- Need for Virtual Memory

- 1st Attempt: Load Time Linking

- 2nd Attempt: Registers and MMU

- 3rd Attempt: Segmentation

Contiguous Memory Allocation: Handling Fragmentation

- Dynamic Memory Allocation. . .

- . . . A Lost Cause

- Common Strategies

- Slab Allocation

- Exploiting Patterns

- Clever Implementation Ideas

Recap

Recap

- ▶ **Fragmentation is caused by**
 - ▶ size heterogeneity;
 - ▶ isolated deaths;
 - ▶ time-varying behavior;
- ▶ **Allocator should try to:**
 - ▶ exploit memory patterns
 - ▶ be evaluated under real workload
 - ▶ have smart and cheap implementation

Your first assignment will be about implementing such an allocator.

- ▶ **User should be aware of it try to ease the allocator's task**
 - ▶ by paying attention to when doing the allocation
 - ▶ by bypassing the allocator
- ▶ **Fighting Fragmentation is a lost cause and is inherent when using segmentation only**
 - ~ We need an other mechanism for Virtual Memory