I: Putting it all together: DLLs
II: Malloc/Free (Heap Storage Management)

Ken Birman

Dynamically Linked Libraries

What's the goal?

- Each program you build consists of
  - Code you wrote
  - Pre-existing libraries your code accesses
- In early days, the balance was "mostly your code" and libraries were small
- But by now, libraries can be immense!

3

Some libraries

- The formatted I/O library
- The windowing subsystem library
- Scientific computing libraries
- Specialized end-to-end communication libraries doing things like
  - Encoding data into XML
  - Implementing web security
  - Implementing RPC

4

Picture?

- By early 1980's, if we accounted for memory in use on a typical computer, we found that
  - Everything needed virtual memory
  - Main "use" of memory was for libraries
- Dynamically linked libraries emerged as a response to this observation

5

Basic idea

- First, let multiple processes share a single copy of each library
  - Thinking: Why have multiple copies of identical code loaded at the same time?
- Next, don't even load a library until it gets used
  - Thinking: Perhaps the application is linked to some libraries for "obscure" reasons and usually doesn't even access those routines!

6

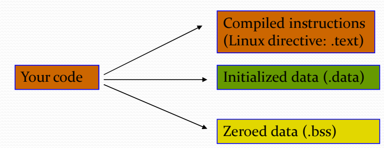
Issues?

- We need to look at:
  - How this can be implemented
  - Overheads it will incur at runtime
  - Impact this has on the size and form of page tables

7

Implementation

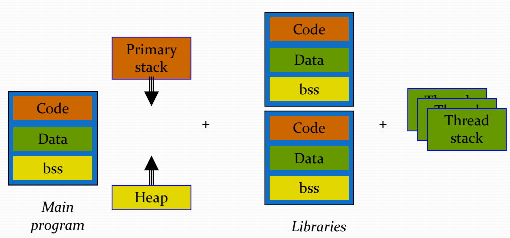
- Recall that compiler transforms each part of a program or library into several kinds of segments:



8

At runtime…

- A process will end up consisting of your main routine(s) and the segments to which it was linked



9

Memory layout for different programs will differ

- Point is that segment sizes differ…
  - E.g. you and I both use the window library
  - But your program is huge and mine is small
- Even when we share a library, the code segment won't be at the same place in memory for each of us!
- How can we compile code for this case?

10

The basic issue

- Comes down to how machine instructions are represented
- Loop:

```
        load _Xmin,r2
Loop:   load -(r0),r1
        addr1,r2,r1
        store r1,(r2)+
        add $-5,r3
        jnz Loop
```

_Xmin is in the data segment. Each copy of the library has its own data segment at a different place in memory!

Where was the constant -5 stored in memory?

Does the linker "fix" the address to which we jump at link time?

11

Position independent code (PIC)

- Idea is to compile our code so that if we have a register containing the base of the data segment for each library, it won't have any references to actual addresses in it
  - Instead of _Xmin, compiler generates something more like _XminOffset(R6)
  - Assumes that R6 is loaded with appropriate base address!

12

What about the jump?

- Here, compiler can generate PC-relative addressing
  - Instead of jnz Loop…
  - …. jnz -64(PC)
- This kind of code is a little slower hence you usually have to TELL the compiler or it won't do this

13

Managing PIC code

- Once we generate PIC code and data segments, each process needs a table
  - One entry per code segment
  - It tells where the data segment for that code segment is located
  - To call the code segment
    - Push old value of Rb (base register) to the stack
    - Load appropriate value from table
    - Call the procedure in question
    - Pop previous value of Rb back from stack

14

"Call the procedure"

- Notice that even the call needs to be done as an indirection!
- In C:
  - Suppose that code_seg_base is the base of the code segment for a function that returns an integer, and we want to call a procedure at offset 0x200 in the code segment
  - We call: res = (*(code_seg_base+0x200))(args)
  - Assumes code_seg_base is declared like this:
    int (*code_seg_base)();

15

So…

- Compile each library using PIC
- When loading the program
  - Put the code segment anywhere, but remember where you put it (code_seg_base)
  - Make a fresh, private copy of the data segment – same size as the "original" copy, but private for this process. Remember base address for use during procedure calls
  - Initialize it from the original version
  - Allocate and zero a suitable BSS region

16

In Linux…

- This results in a table, potentially visible to debugger, in Linux called _ _ DYNAMIC
  - Entries consist of
    - Name of the file containing the library code
    - Status bits: 0 if not yet loaded, 1 if loaded
    - Code base address
    - Data segment base address
  - On first access
    - "Map" the file into memory, remembering base addr
    - Malloc and initialize copy of data segment
    - Update the entry for this segment accordingly

17

Mapping a file

- Everyone is used to the normal file system interface
  - File open, seek, read, write
- But Linix and Windows also support memory mapping of files
  - `Base_addr = mmap("file-name", …)`
  - This creates a window in memory and you can directly access the file contents at that address range!
  - Moreover, different processes can share a file
  - Arguments tell mmap how to set up permissions

18

Summary

- So…
  - Compile with PIC directive to compiler
  - But also need to decide where the library will be placed in the file system
- Now, compile application program and tell it that the windowing library is a DLL
  - It needs to know the file name, e.g. /lib/xxx.so
- Resulting executable is tiny… and will link to the DLL at runtime

19

DLLs are popular!

- Even Microsoft DOS had them
  - In DOS, they sometimes put multiple DLLs at the same base address
  - Requires them to swap A out if B gets used, and vice versa, but makes memory footprint of programs smaller
- Very widely used now… almost universal

20

Consequence for page table?

- A single process may be linked to a LOT of segments
  - Suppose: 10 libraries and 30 threads
    - You'll have 2 segments per library
    - Plus approximately five for the main process
    - Plus 30 for lightweight thread stacks
    - … a total of 55 segments!
  - And these are spread "all over the place" with big gaps between them (why?)

21

Issue

- Page table might be huge
  - Covers an enormous range of addresses
  - And has big holes in it
- One approach: page the page table
  - Costs get high
- Better approach: an *inverted page table*

22

Inverted page table

- Must be implemented by hardware
- Idea is this:
  - Instead of having one page table entry per virtual memory page, have one PTE per physical memory page
  - It tells which process and which virtual page is currently resident in this physical page
  - The O/S keeps "remaining" PTE's for non-resident virtual pages in some other table

23

Benefits?

- With an inverted page table, we have an overhead of precisely one PTE per physical page of memory
- CPU needs to be able to search this quickly
  - Turns out to be identical to what TLB already was doing (an associative lookup)
  - So can leverage existing hardware to solve this problem

24

Summary

- All modern systems use DLLs heavily
  - You can build your own
  - Just need to understand how to tell the compiler what you are doing (for PIC, and to agree on the name for the DLL files)
- Only some computers support inverted page tables
- Others typically have a two-level paging scheme that pages the page table

25

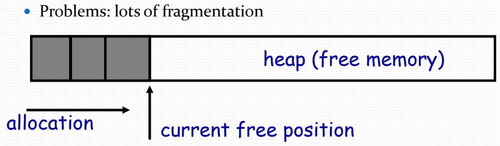
Dynamic Memory Management

- 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

26

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



27

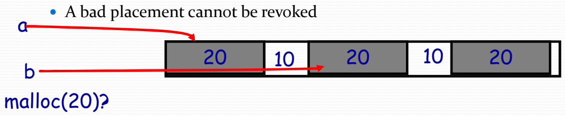
Memory allocation goals

- 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

28

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

29

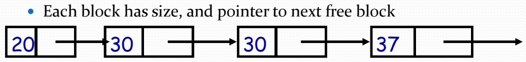
Impossibility Results

- 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

30

Best Fit Allocation

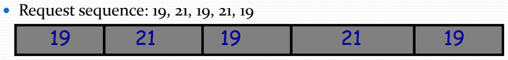
- Minimum size free block that can satisfy request
- Data structure:
  - List of free blocks
  - Each block has size, and pointer to next free block



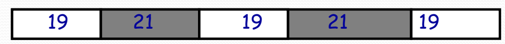
- Algorithm:
  - Scan list for the best fit

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?

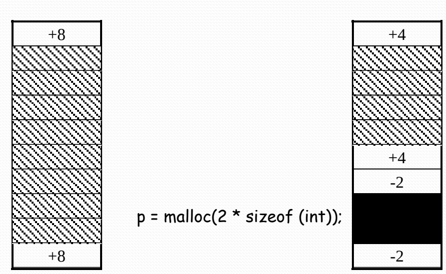
- Maintain a list of free chunks
  - Binning, doubly linked lists, etc
- Use best fit or any other strategy to determine page
  - For example: binning with best-fit
- What if allocated chunk is much bigger than request?
  - Internal fragmentation
  - Solution: split chunks
    - Will not split unless 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

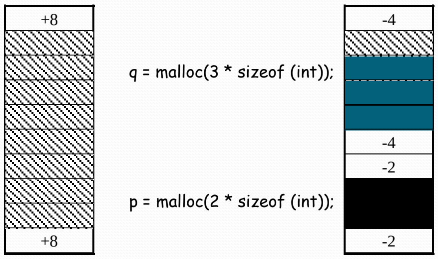
Example

Initially one chunk, split and make signs negative on malloc



Example

q gets 4 words, although it requested for 3



37

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

38

Buddy-Block Scheme

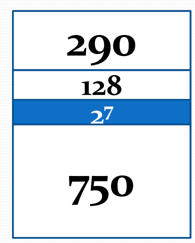
- Invented by Donald Knuth, very simple
- Idea: Work with memory regions that are all powers of 2 times some "smallest" size
  - 2^k times *b*
- Round each request *up* to have form $b*2^k$

39

Buddy-Block Scheme

Buddy Block in Action

- P = malloc(290)
  - … round up to $2^8 = 512$
- Q = malloc (111)
  - …. Round up to $2^7 = 128$
- Z = malloc(750)
  - …. Round up to $2^9 = 1024$
- Free(Q)
  - … consolidate



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)

41

How to get more space?

- In Unix, system call sbrk()

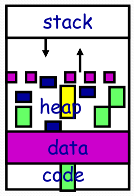
```
/* add nbytes of valid virtual address space */
void *get_free_space(unsigned nbytes) {
  void *p;
  if(!(p = sbrk(nbytes)))
      error("virtual memory exhausted");
  return p;
}
```

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

42

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



43

Summary

- Modern OS includes a variety of memory allocators
  - Best of all is the one used for paging but this is because pages are “interchangable”
  - For other purposes, we use fancier structures, like Knuth’s buddy-block scheme
  - Some applications also include their own free-list managers, to avoid cost of malloc/free for typical objects