CSO: Recitation 10

Goktug Saatcioglu

11.21.2019

1 Manipulating Data

1.1 Array Allocation

- Basic Principle `T A[L];`
  - Array of data type `T` and length `L`
  - Contiguously allocated region of `L * sizeof(T)` bytes in memory
  - Identifier `A` used as a pointer to array element 0: Type `T*`
- Arrays are NOT gauranteed to always be allocated in successive memory blocks

1.2 Multidimensional (Nested) Arrays

- Declaration `T A[R][C];`
  - 2D array of data type `T`
  - `R` rows, `C` columns
  - Type `T` element requires `K` bytes
- Array Size
  - `R*C*K` bytes
- Arrangement in memory
  - Row-Major Ordering (i.e. the rows are stored contigiously)
- Array Elements
  - address of `A[i][j]`: `A + i * (C * K) + j * K = A + (i * C + j)* K`

1.3 Multi-Level Array

- Declaration `T *A[N]`
  - Variable `A` denotes array of 3 elements
  - Each elements is a pointer of 8 bytes
  - Each pointer points to array of `T`'s
- Item access computatation: `A[i][j]`
  - Element access `Mem[Mem[A+8*i]+T*j]`
  - Must do two memory reads: First get pointer to row array, Then access element within array

1.4 Structure Representation

- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

1.5 Alignment Principles

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent). Inefficient to load or store datum that spans quad word boundaries (i.e. 8 bytes boundaries).
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

1.6 Example

If we have the struct

```
struct s1 {
  char c;
  int i[2];
  double d;
} *p;
```

then unaliged the data coud look like

```
 --------------------
| c | i[0] | i[1] | d |
 --------------------
    ^      ^      ^   ^
   p+1    p+5    p+9 p+17
```

and aligned the data could look like

```
 ----------------------------
| c | x | i[0] | i[1] | x | d |
 ----------------------------
    ^   ^      ^      ^   ^   ^
   p+1 p+4    p+8   p+12 p+16 p+24
```

1.7 Specific Cases of Alignment (x86-64)

- 1 byte: char, ...
  - no restrictions on address
- 2 bytes: short, ...
  - address must be multiple of 2
- 4 bytes: int, float, ...
  - address must be multiple of 4
- 8 bytes: double, long, char *, ...
  - address must be multiple of 8
- 16 bytes: long double (GCC on Linux)
  - address must be multiple of 16
- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement `K`, `K` = Largest alignment of any element
  - Initial address and structure length must be multiples of `K`

1.8 Requirements and Efficiency

- For largest alignment requirement `K`
- Overall structure must be multiple of `K`
- To save space: Put large data types first

2 Dynamic Allocation

2.1 Why dynamic allocation

- Allocation size is unknown until the program runs (at runtime).
- Question: Can one dynamically allocate memory on stack? Answer: Yes, but space is freed upon function return. Use:

```
#include <stdlib.h>
void *alloca(size_t size);
```

2.2 How allocate dynamically

Ask OS for allocation on the heap via system calls using the function

```
void *sbrk(intptr_t size);
```

which increases the top of heap by size and returns a pointer to the base of new storage. The size can be a negative number. Note that `brk` stands for break value For example,

```
p = sbrk(1024) // allocate 1KB
```

will allocate 1KB on the heap. To free this memory, we can do

```
sbrk(-1024) // free p
```

There are a couple of issues with heap allocated memory. They are

1. We can only free the memory on the top of heap as this is how `sbrk` works. For example, if we did

   ```
   p1 = sbrk(1024) // allocate 1KB
   p2 = sbrk(2048) //allocate 2KB
   ```

   how can we free `p1` without freeing `p2`?

2. System calls have high performance overheads/costs. They are usually 10x greater than using the stack.

So, the question is then how do we efficiently allocate memory on the heap? Answer: request a large memory region on heap from OS once, then manage this memory region by itself. This means we need some sort of allocator and this is implemented in a user-level library.

2.3 Types of Dynamic Memory Allocators

- Explicit allocator (used by C/C++): application allocates and frees space
  - `malloc` and `free` in C
  - `new` and `delete` in C++
- Implicit allocator (used by Java,...): application allocates, but does not free space
  - Garbage collection in Java, Python etc.

2.4 Challenges facing a memory allocator

- Achieve good memory utilization
  - Apps issue arbitrary sequence of malloc/free requests of arbitrary sizes
  - Utilization = sum of malloc'd data / size of heap
- Achieve good performance
  - malloc/free calls should return quickly
  - Throughput = # ops/sec
- Constraints:
  - Cannot touch/modify malloc'd memory
  - Can't move the allocated blocks once they are malloc'd, i.e. compaction is not allowed

2.5 Fragmentation

Poor memory utilization caused by fragmentation

- Internal fragmentation:
  - Malloc allocates data from blocks of certain sizes.
  - Internal fragmentation occurs if payload is smaller than block size
  - Block size decided by allocator's designer.
  - Payload is the number of bytes you want when you call `malloc()`, ...

  - Primarily caused by limited choices of block sizes and padding for alignment purposes
  - Think of: Size given ¿ size needed, wasted = size given - size needed

- External fragmentation:
  - Occurs when there is enough aggregate heap memory, but no single free block is large enough
  - E.g.

```
p1 = malloc(100);
p2 = malloc(100);
p3 = malloc(100);
free(p1);
free(p3);
malloc(200) //?
```

  - Think of: Enough space is available but not enough contigious space can be formed. So we have enough space for allocation but since the allocation must be contigious we cannot do it.

2.6 Malloc design choices

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a space that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation – many might fit?
- How do we reinsert freed block?

2.7 Knowing how much to free

The standard method is to keep the length of a block in the header field preceding the block. This means that there will be some required overhead for every allocated block.

2.8 Keeping track of free blocks

There are three methods

- Method 1: Implicit list using length — links all blocks
- Method 2: Explicit list among the free blocks using pointers
- Method 3: Segregated free list. This means we keep different free lists for different size classes.

2.9 Free List Details

- Malloc grows a contiguous region of heap by calling `sbrk()`
- Heap is divided into variable-sized blocks
- For each block, we need both size and allocation status
- So we store in a block a header, the paylod and some padding if necessary
- The header is 4 bytes in length as stores at the first 31 bits the length of the block and then the last bit is whether the block has been allocated or not. If the allocated bit is 1 then that block has been allocated and otherwise it has not been allocated.
- How do we find a free block?

  - First fit: Search from beginning, choose first free block that fits. Can lead to a lot of fragmentation.
  - Next fit: Like first fit, except search starts where previous search finished. Somewhat more efficient compared to first fit but can still cause fragmentation.
  - Best fit: Search the list, choose the best free block: fits, with fewest bytes left over (i.e. pick the smallest block that is big enough for the payload). Lowers the amount of fragmentation but will typically run slower compared to first fit.

- How do we allocate a free block?
  - We use splitting. When we find a free block, the space we need may be smaller than the available free space. So we split the block and then allocate the memory we need.

- How do we free a block?
  - The simplest implementation is to simply set the allocated flag to false. However, this can lead to false fragmentation. Consider two consecutive blocks. The first is in use and the second is not in use. Suppose then we free the first block. This will lead to two different blocks when in fact we have one larger free block. This leads to fragmentation.
  - Better idea: join (coalesce) with next/previous blocks, if they are free. This solves the issue described in the simplest implementation.
  - Bidirectional coalescing: We use what our called boundary tags [Knuth73]. We replicate the block header at the end of blocks which allows us to traverse the in both directions: forwards and backwards. We pay a small penalty as we need to use extra space. However, we can now better coalesce as we can not only join blocks ahead of a certain block but also blocks preceding a certain block.

- When should we coalesce?
  - Immediate coalescing: coalesce each time `free()` is called
  - Deferred coalescing: try to improve performance of free by deferring coalescing until needed. For example, we can coalesce as we scan the free list for a `malloc()` call or we can coalesce when the amount of external fragmentation reaches some threshold.

- Summary:
  - Implementation: very simple
  - Allocate cost: linear time worst case
  - Free cost: constant time worst case, even with coalescing
  - Memory usage: will depend on first-fit, next-fit or best-fit
  - Not used in practice for `malloc()/free()` because of linear-time allocation costs but used in many special purpose applications since it is simple to implement and works well.

2.10 Explicit Free List Details

- Maintain list(s) of free blocks instead of all blocks
- Need to store forward/back pointers in each free block, not just sizes. Why? A: Because free blocks may not be contiguous in heap. This naturally will increase the space overhead for free blocks but not for allocated blocks. We also still use the header+footer for the blocks.
- How to insert a new freed block into the list?
  - Insert freed block at the beginning of the free list (LIFO). This is simple to use and takes constant time.

- Insert freed blocks to maintain address order: addr(prev) ¡ addr(curr) ¡ addr(next). This may lead to less fragmentation compared to LIFO but takes longer.

- We see that for explicit free lists allocation is linear time in # of free blocks instead of all blocks since we just search the blocks that we know are free. However, it can still take a while to find a free block that fits. This leads to the segmented list implementation where we keep multiple linked lists of different size classes.

2.11 Segregated List (Seglist) Details

- Multiple free lists each linking free blocks of similar sizes (i.e. one list that holds blocks of size 4, another that holds blocks of size 3 and etc.)

- The free list works as follows. Given an array of free lists, each one for some size class
  - To allocate a block of size `n`:
    - Search in appropriate free list containing size `n`
    - Split found block and place fragment on appropriate list
    - try next larger class if no blocks found
  - If no block is found:
    - Request additional heap memory from OS
    - Allocate block of `n` bytes from this new memory
    - Place remainder as a single free block in largest size class.
  - To free a block:
    - Coalesce and place on appropriate list

- Seglist allocators have the advantages of fast allocation (we simply find the required block by searching the list and only ask for more memory from the OS as a last resort), better memory utilization (we use the list to find the minimally larger than required block we need and if we do have a segment that can be put back into the list we put it back so it can be used again) and the first-fit search of segregated free list approximates a best-fit search of entire heap (not always the best but most of the time is very good).

2.12 Garbage Collection

- In C, it is the programmer's responsibility to free any memory allocated on the heap.

- A garbage collecter is a dynamic storage allocator that automatically frees allocated blocks that are no longer needed by the program.

- Allocated blocks that are no longer needed are called garbage.

- It is used in systems that support garbage collection (e.g. Java, Perl, Mathematica, ...).

- Applications will explicitly allocate heap blocks (e.g. in Java we do `new String("abc");`) but never free them when they are no longer needed.

- The garbage collector periodically identifies garbage and make appropriate calls to free. It does this by scanning what is called a reachability graph. The unreachable blocks of this graph can be freed.

2.13 Conclusions

- Dynamic memory allocator manages the heap.
- Dynamic memory allocator is part of the user-space.
- The allocator has two main goals: 1. reaching higher throughput (operations per second) and 2. better memory utilization (i.e. reduces fragmentation).
- Explicit allocator: Works in terms of blocks and keeps track of free blocks (Implicit list, Explicit list, Segregated list)
- Implicit allocator (a.k.a garbage collector)