CS354: Machine Organization and Programming

Lecture 36
Wednesday the November 25th 2015

Section 2
Instructor: Leo Arulraj

Class Announcements

Programming Assignment 4 was due today at 9 AM. You can submit it upto 48 hours after the deadline with penalties.

Programming Assignment 5 has been released. It is due by Dec 9th, Wednesday. Significantly tougher than the rest of the assignments. So start early!

Lecture Overview

What is covered in this lecture won't be important from the final exam's perspective.

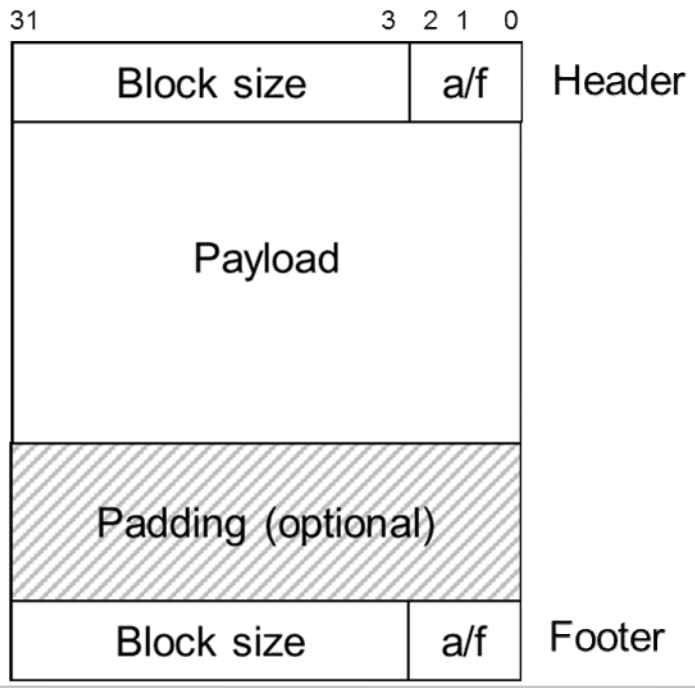
Explicit free lists
Segregated free lists

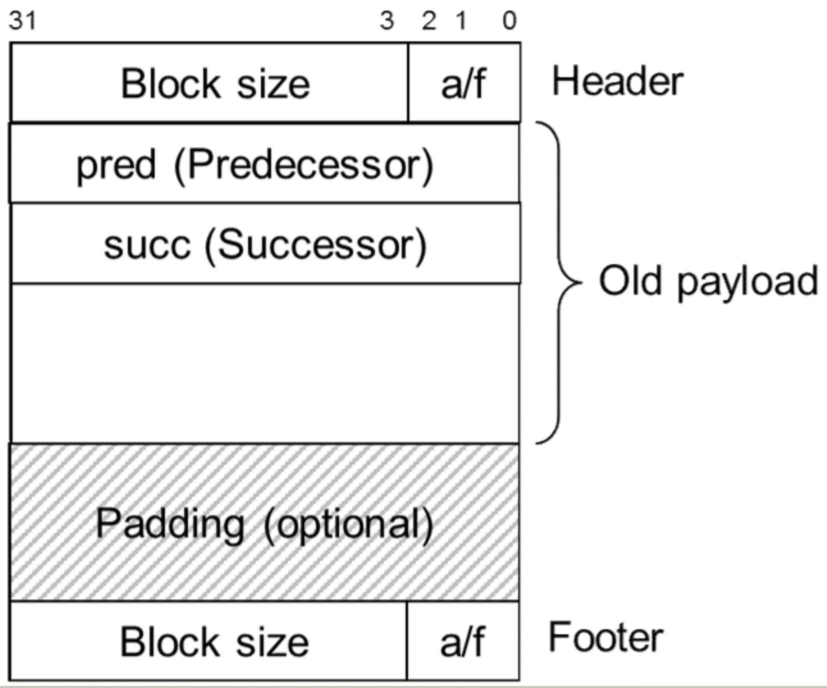
Pseudocode for Memory Allocator operations
Assignment 5 Overview : Skeleton, Test cases

Common memory related bugs in C Programs

Explicit free lists

Maintaining a doubly linked list of the free blocks can reduce the allocation time of first fit policy from linear in number of blocks to linear in number of free blocks.





Segregated Free lists

Maintain multiple free lists where each list holds blocks that are roughly the same size. Two specific implementation of this idea are:

(More details in text book)

Simple Segregated Storage:

- Blocks are allocated from the appropriate free list and are never split or coalesced.
- If no new blocks are found, memory is requested from OS , split and inserted into the list without free blocks.

Segregated Fit:

- Splitting is allowed. The newly created free block after splitting is inserted into the appropriate list based on its size.
- If no new blocks are found, a block from the next list is taken.

Skeleton of Programming Assignment 5

Go over the free block structure.

Go over Mem_Init(int sizeofRegion).

Go over Mem_Dump()

Steps for Mem_Alloc() , Mem_Free()

Steps for Mem_Alloc

Mem_Alloc(int size)

- Check size is positive
- Round up size to multiple of 4 bytes
- Loop through all nodes in the implicit free list and find a best fit free block that fits the size requested. If no free block can be found return NULL.
- Calculate the leftover space in the free block and create a new free block out of it
- Mark the best fit block as allocated and return the pointer to the payload space.

Steps for Mem_Free

Mem_Free(void *ptr)

- If pointer is NULL return -1
- Loop through all the nodes in the free list in order to find the allocated block with payload at "ptr". Keep note of the previous block too.
- If no matching block can be found return -1
- If matching block is found, mark the block as free.
- Coalesce with the next block and previous block if they are free too.

Test Cases

Programming Assignment 5 comes with a lot of prewritten test cases to help you get your implementation right.

The file testlist.txt contains the list of all tests.

Common memory related bugs

Now, let us go over some of the common memory related bugs.

This list is from the textbook.

Dereferencing bad pointers

Large holes in the virtual address space that are not mapped to any meaningful data.

Attempting to write to read only virtual memory regions causes protection exception.

E.g. scanf(“%d”, val); instead of scanf(“%d”,&val);

Reading uninitialized memory

Heap memory need not be always initialized to zeros.

So, programmer must either zero it out explicitly or use calloc() variant which does the zeroing out for the programmer.

Allowing stack buffer overflows

Use those library functions that know the size of the destination buffer they are writing to.

E.g. use fgets() instead of gets()

Also, check for the return values of library functions for error scenarios and handle appropriately.

Assuming that pointers and the pointed objects are the same size

Example: Creating 2-D array of int dynamically.

```
int ** arr = (int**) malloc(n*sizeof(int))
// The above line must use sizeof(int*)
for(int i=0; i < n; i++)
      arr[i] = (int*) malloc(n*sizeof(int))
```

Wont work when int and int* are of different sizes. (E.g. 64 bit architecture)

Making off-by-one errors

For the previous example on 2D array allocation, consider this version:

```
float ** arr = (float**) malloc(n*sizeof(float))
for(int i=0; i <= n; i++)
      arr[i] = (float**) malloc(n*sizeof(float))
```

Referencing a pointer instead of the object it points to

Need to be careful about the precedence and associativity of operators in C while dealing with pointers.

Consider this:

```
int *size;
*size--;   // use (*size)-- instead.
```

Moral: Use parentheses when in doubt.

Misunderstanding pointer arithmetic

Arithmetic operations on pointers are performed in units that are the size of the objects they point to which are not necessarily bytes.

E.g.

```
float * a = (float*)malloc(10*sizeof(float))
float *p = a;
p = p + (sizeof(float)*4) // to access 5th element
//use p+=4 instead
```

Referencing non-existent variables

Naïve programmers might reference local variables in stack after they cease out of existence.

E.g.

```
struct list_node* stackref(){
        struct list_node val;
        //local variable allocated on stack
        return &val;
}
```

Referencing data in free heap blocks

Accessing heap allocated space after calling free on it leads to bugs.

E.g.

```
float ** arr1 = (float**) malloc(n*sizeof(float))
….
free(arr1);
….
….
float ** arr2 = (float**) malloc(n*sizeof(float))
for(int i=0; i <= n; i++)
        arr2[i] = arr1[i]/2; // incorrectly accessing freed arr1
```

Introducing memory leaks

Not freeing memory allocated from heap after ceasing to use it can lead to memory leaks.

Memory leaks result in chunks of memory that the program no longer uses but the memory allocator think they are in use.

E.g. void delete_node(struct list_node* head){

….

Code to remove the node from the linked list but not actually freeing it can cause memory leaks

….

}