

Memory Allocation

CSE 361S

Harsh Reality

Memory Matters

Memory is not unbounded

- It must be allocated and managed
- Many applications are memory dominated
 - Especially those based on complex, graph algorithms

Memory referencing bugs especially pernicious

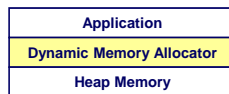
- Effects are distant in both time and space

Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

-2-

Dynamic Memory Allocation



Explicit vs. Implicit Memory Allocator

- **Explicit:** application allocates and frees space
 - E.g., `malloc` and `free` in C
- **Implicit:** application allocates, but does not free space
 - E.g. garbage collection in Java, ML or Lisp

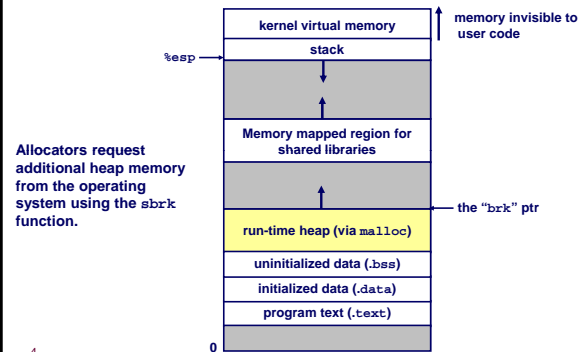
Allocation

- In both cases the memory allocator provides an abstraction of memory as a set of blocks
- Doles out free memory blocks to application

Will discuss simple explicit memory allocation today

-3-

Process Memory Image



-4-

Malloc Package

```
#include <stdlib.h>
```

```
void *malloc(size_t size)
```

- **If successful:**
 - Returns a pointer to a memory block of at least `size` bytes, (typically) aligned to 8-byte boundary.
 - If `size == 0`, returns `NULL`.
- **If unsuccessful:** returns `NULL (0)` and sets `errno`.

```
void free(void *p)
```

- Returns the block pointed at by `p` to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`.

```
void *realloc(void *p, size_t size)
```

- Changes size of block `p` and returns pointer to new block.
- Contents of new block unchanged up to min of old and new size.

-5-

Malloc Example

```
void foo(int n, int m) {
    int i, *p;

    /* allocate a block of n ints */
    if ((p = (int *) malloc(n * sizeof(int))) == NULL) {
        perror("malloc");
        exit(0);
    }
    for (i=0; i<n; i++)
        p[i] = i;

    /* add m bytes to end of p block */
    if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
        perror("realloc");
        exit(0);
    }
    for (i=n; i < n+m; i++)
        p[i] = i;

    /* print new array */
    for (i=0; i<n+m; i++)
        printf("%d\n", p[i]);

    free(p); /* return p to available memory pool */
}
```

-6-

Assumptions

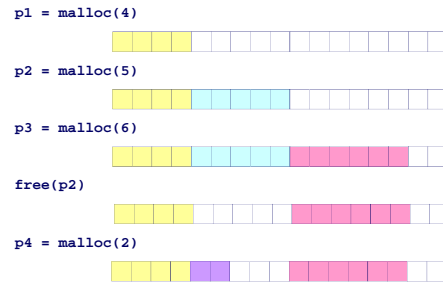
Assumptions made in this lecture

- Memory is word addressed (each word can hold a pointer)



- 7 -

Allocation Examples



- 8 -

Constraints

Applications:

- Can issue arbitrary sequence of allocation and free requests
- Free requests must correspond to an allocated block

Allocators

- Can't control number or size of allocated blocks
- Must respond immediately to all allocation requests
 - i.e.*, can't reorder or buffer requests
- Must allocate blocks from free memory
 - i.e.*, can only place allocated blocks in free memory
- Must align blocks so they satisfy all alignment requirements
 - 8 byte alignment for GNU malloc (`libc malloc`) on Linux boxes
- Can only manipulate and modify free memory
- Can't move the allocated blocks once they are allocated
 - i.e.*, compaction is not allowed

- 9 -

Goals of Good malloc/free

Primary goals

- Good time performance for `malloc` and `free`
 - Ideally should take constant time (not always possible)
 - Should certainly not take linear time in the number of blocks
- Good space utilization
 - User allocated structures should be large fraction of the heap.
 - Want to minimize "fragmentation".

Some other goals

- Good locality properties
 - Structures allocated close in time should be close in space
 - "Similar" objects should be allocated close in space
- Robust
 - Can check that `free(p1)` is on a valid allocated object `p1`
 - Can check that memory references are to allocated space

- 10 -

Performance Goals: Throughput

Given some sequence of malloc and free requests:

- $R_0, R_1, \dots, R_k, \dots, R_{n-1}$

Want to maximize throughput and peak memory utilization.

- These goals are often conflicting

Throughput:

- Number of completed requests per unit time
- Example:
 - 5,000 malloc calls and 5,000 free calls in 10 seconds
 - Throughput is 1,000 operations/second.

- 11 -

Performance Goals: Peak Memory Utilization

Given some sequence of malloc and free requests:

- $R_0, R_1, \dots, R_k, \dots, R_{n-1}$

Def: Aggregate payload P_k :

- `malloc(p)` results in a block with a *payload* of `p` bytes..
- After request R_k has completed, the *aggregate payload* P_k is the sum of currently allocated payloads.

Def: Current heap size is denoted by H_k

- Assume that H_k is monotonically nondecreasing

Def: Peak memory utilization:

- After k requests, *peak memory utilization* is:
 - $U_k = (\max_{i \leq k} P_i) / H_k$

- 12 -

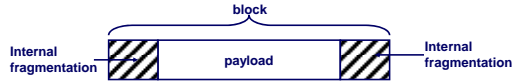
Internal Fragmentation

Poor memory utilization caused by *fragmentation*.

- Comes in two forms: internal and external fragmentation

Internal fragmentation

- For some block, internal fragmentation is the difference between the block size and the payload size.

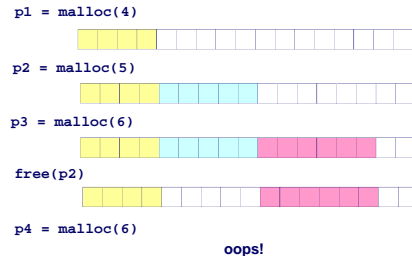


- Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).
- Depends only on the pattern of *previous* requests, and thus is easy to measure.

- 13 -

External Fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough

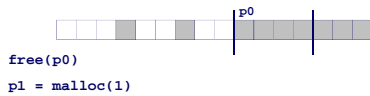


External fragmentation depends on the pattern of *future* requests, and thus is difficult to measure.

- 14 -

Implementation Issues

- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure 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?

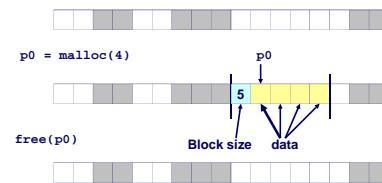


- 15 -

Knowing How Much to Free

Standard method

- Keep the length of a block in the word preceding the block.
 - This word is often called the *header field* or *header*
- Requires an extra word for every allocated block



- 16 -

Keeping Track of Free Blocks

Method 1: Implicit list using lengths -- links all blocks



Method 2: Explicit list among the free blocks using pointers within the free blocks



Method 3: Segregated free list

- Different free lists for different size classes

Method 4: Blocks sorted by size

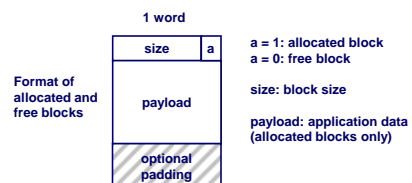
- Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

- 17 -

Method 1: Implicit List

Need to identify whether each block is free or allocated

- Can use extra bit
- Bit can be put in the same word as the size if block sizes are always multiples of two (mask out low order bit when reading size).



- 18 -

Implicit List: Finding a Free Block

First fit:

- Search list from beginning, choose first free block that fits

```
p = start;
while ((p < end) || \ \ not passed end
      (*p & 1) || \ \ already allocated
      (*p <= len)); \ \ too small
```

- Can take linear time in total number of blocks (allocated and free)
- In practice it can cause "splinters" at beginning of list

Next fit:

- Like first-fit, but search list from location of end of previous search
- Research suggests that fragmentation is worse

Best fit:

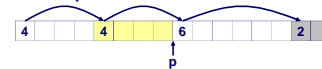
- Search the list, choose the free block with the closest size that fits
- Keeps fragments small --- usually helps fragmentation
- Will typically run slower than first-fit

- 19 -

Implicit List: Allocating in Free Block

Allocating in a free block - splitting

- Since allocated space might be smaller than free space, we might want to split the block



```
void addblock(ptr p, int len) {
int newsize = ((len + 1) >> 1) << 1; // add 1 and round up
int oldsize = *p & -2; // mask out low bit
*p = newsize | 1; // set new length
if (newsize < oldsize)
*(p+newsize) = oldsize - newsize; // set length in remaining
// part of block
```

addblock(p, 2)

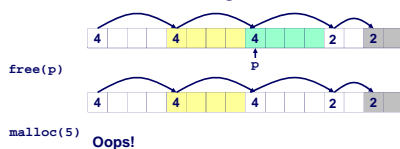


- 20 -

Implicit List: Freeing a Block

Simplest implementation:

- Only need to clear allocated flag
- ```
void free_block(ptr p) { *p = *p & -2;
```
- But can lead to "false fragmentation"



There is enough free space, but the allocator won't be able to find it

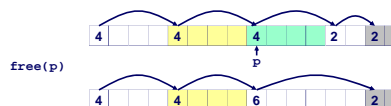
- 21 -

## Implicit List: Coalescing

### Join (*coalesce*) with next and/or previous block if they are free

- Coalescing with next block

```
void free_block(ptr p) {
*p = *p & -2; // clear allocated flag
next = p + *p; // find next block
if ((*next & 1) == 0)
*p = *p + *next; // add to this block if
// not allocated
```



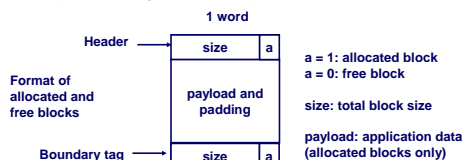
- But how do we coalesce with previous block?

- 22 -

## Implicit List: Bidirectional Coalescing

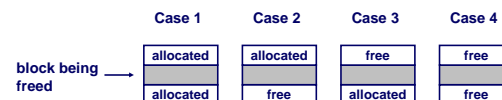
### Boundary tags [Knuth73]

- Replicate size/allocated word at bottom of free blocks
- Allows us to traverse the "list" backwards, but requires extra space
- Important and general technique!



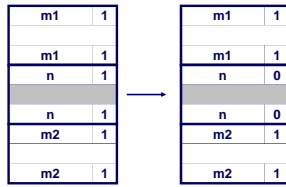
- 23 -

## Constant Time Coalescing



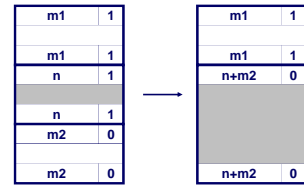
- 24 -

### Constant Time Coalescing (Case 1)



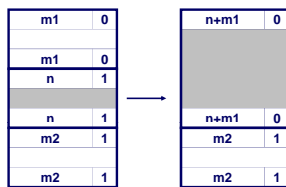
- 25 -

### Constant Time Coalescing (Case 2)



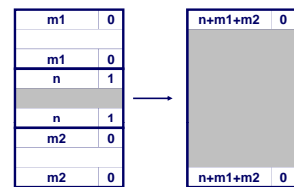
- 26 -

### Constant Time Coalescing (Case 3)



- 27 -

### Constant Time Coalescing (Case 4)



- 28 -

## Summary of Key Allocator Policies

#### Placement policy:

- First fit, next fit, best fit, etc.
- Trades off lower throughput for less fragmentation
  - Interesting observation: segregated free lists (next lecture) approximate a best fit placement policy without having the search entire free list.

#### Splitting policy:

- When do we go ahead and split free blocks?
- How much internal fragmentation are we willing to tolerate?

#### Coalescing policy:

- Immediate coalescing: coalesce adjacent blocks each time free is called
- Deferred coalescing: try to improve performance of free by deferring coalescing until needed. e.g.,
  - Coalesce as you scan the free list for malloc.
  - Coalesce when the amount of external fragmentation reaches some threshold.

- 29 -

## Implicit Lists: Summary

- **Implementation:** very simple
- **Allocate:** linear time worst case
- **Free:** constant time worst case -- even with coalescing
- **Memory usage:** will depend on placement policy
  - First fit, next fit or best fit

**Not used in practice for malloc/free because of linear time allocate. Used in many special purpose applications.**

**However, the concepts of splitting and boundary tag coalescing are general to all allocators.**

- 30 -