10: Dynamic memory allocation

Computer Architecture and Systems Programming

252-0061-00, Herbstsemester 2013

Timothy Roscoe

Systems@ETH Zürich

1

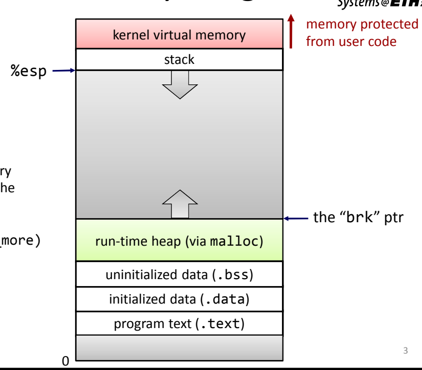
Why dynamic memory allocation?

- It's very simple:

Sizes of needed data structures may only be known at runtime

2

Process memory image



Allocators request additional heap memory from the kernel using the `sbrk()` function:

`error = sbrk(amt_more)`

3

Dynamic memory allocation

- Memory allocator?
  - VM h/w and kernel allocate pages
  - Application objects typically smaller
  - Allocator manages objects within pages
- Allocation
  - A memory allocator doles out memory blocks to application
  - "block": contiguous range of bytes
    - of any size, in this context

Application
Dynamic Memory Allocator
Heap Memory

4

Recall the malloc package

`#include <stdlib.h>`

`void *malloc(size_t size)`

Successful:

Returns a pointer to a memory block of at least `size` bytes (typically) aligned to 8-byte boundary

If `size == 0`, returns NULL

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

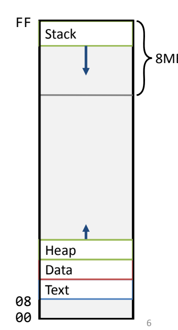
Old block has been `free()`'d (logically, if new != old)

5

Memory allocation example

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?



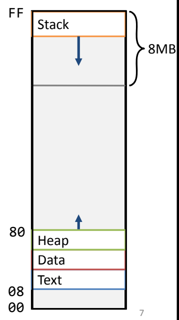
6

IA32 example addresses

address range ~2^{32}

$esp	0xffffbcd0
p3	0x65586008
p1	0x55585008
p4	0x1904a110
p2	0x1904a008
&p2	0x18049760
beyond	0x08049744
big_array	0x18049780
huge_array	0x08049760
main()	0x080483c6
useless()	0x08049744
final malloc()	0x006be166

`malloc()` is dynamically linked
address determined at runtime

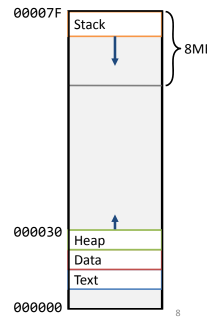


x86-64 example addresses

address range ~2^{47}

$rsp	0x7ffffff8d1f8
p3	0x2aaabaadd010
p1	0x2aaaaaadc010
p4	0x000011501120
p2	0x000011501010
&p2	0x000010500a60
beyond	0x000000500a44
big_array	0x000010500a80
huge_array	0x000000500a50
main()	0x000000400510
useless()	0x000000400500
final malloc()	0x00386ae6a170

`malloc()` is dynamically linked
address determined at runtime



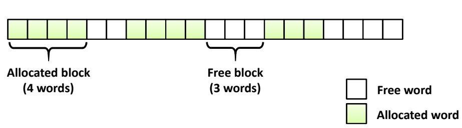
Explicit vs. implicit memory allocators

- Explicit: application allocates and frees space
  - In C: malloc() and free()
  - In C++: new() and destroy *(sort of)*
  - What we'll talk about first
- Implicit: application allocates, but does not free
  - Java, ML, Lisp, C#, etc.
  - Freeing is done by a *Garbage Collector*
  - What we talk about later…

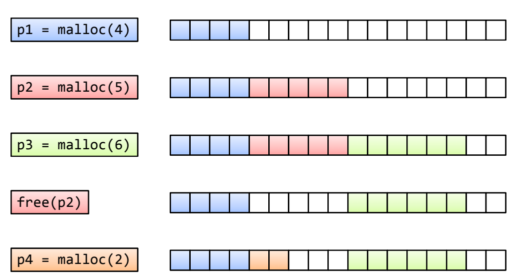
10.1: The problem

Assumptions we make (in this lecture)

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



Allocation example



Constraints

- Applications
  - Can issue arbitrary sequence of `malloc()` and `free()` requests
  - `free()` requests must be to a `malloc()`'d block
- Allocators
  - Can't control number or size of allocated blocks
  - Must respond immediately to `malloc()` 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 manipulate and modify only free memory
  - Can't move the allocated blocks once they are `malloc()`'d
    - *i.e.*, compaction is not allowed

Performance goal: throughput

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- Goals: 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
  - ***How to do `malloc()` and `free()` in O(1)? What's the problem?***

Performance goal: peak memory utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- *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
    - all `malloc()`'d stuff minus all `free()`'d stuff
- *Def: Current heap size = H_k*
  - Assume H_k is monotonically nondecreasing
    - reminder: it grows when allocator uses `sbrk()`
- *Def: Peak memory utilization after k requests*
  - $U_k = (\max_{i<k} P_i) \;/\; H_k$

Implementation Issues

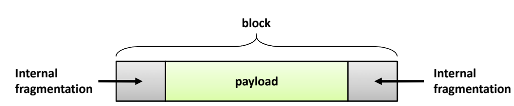
- How to know how much memory is being `free()`'d when it is given only a pointer (and no length)?
- How to keep track of the free blocks?
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?
- How to pick a block to use for allocation—many might fit?
- How to reinsert a freed block into the heap?

Challenge :fragmentation

- Poor memory utilization caused by *fragmentation*
  - ***internal*** fragmentation
  - ***external*** fragmentation

Internal fragmentation

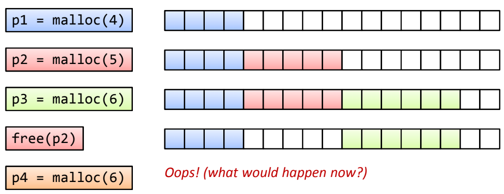
- For a given block, internal fragmentation occurs if payload < block size



- Caused by
  - overhead of maintaining heap data structures
  - padding for alignment purposes
  - explicit policy decisions (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of previous requests
  - thus, easy to measure

External fragmentation

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

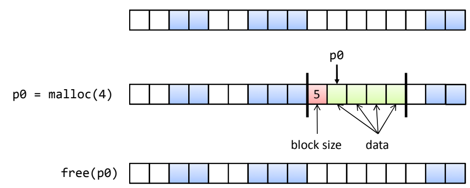


- Depends on the pattern of future requests
  - Thus, difficult to measure

19

Knowing how much to free

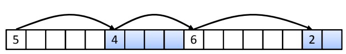
- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block



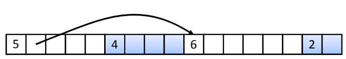
20

Keeping track of free blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers



- 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

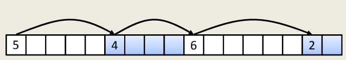
21

10.2 Implicit free lists

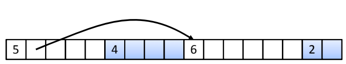
22

Keeping track of free blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers



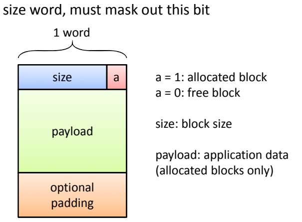
- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

23

Implicit list

- For each block we need: length, is-allocated?
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

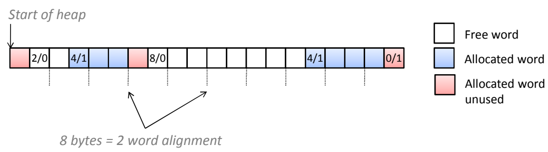
Format of allocated and free blocks



24

Example

Sequence of blocks in heap: 2/0, 4/1, 8/0, 4/1



- 8-byte alignment
  - May require initial unused word
  - Causes some internal fragmentation
- One word (0/1) to mark end of list
- Here: block size in words for simplicity

25

Implicit lists: Finding a free block

- First fit:
  - Search list from beginning, choose first free block that fits: (Cost?)

```
p = start;
while ((p < end) &&          \\ not passed end
       ((*p & 1) ||          \\ already allocated
       (*p <= len)))         \\ too small
  p = p + (*p & -2);         \\ goto next block (word addressed)
```

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

26

Implicit lists: Finding a free block

- Next fit:
  - Like first-fit, but search list starting where previous search finished
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- Best fit:
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Will typically run slower than first-fit

27

Bit fields

- How to represent the header:
  masks and bitwise operators

```
#define SIZEMASK            (~0x7)
#define PACK(size, alloc)   ((size) | (alloc))
#define GET_SIZE(p)         ((p)->size & SIZEMASK)
```

- Bit fields

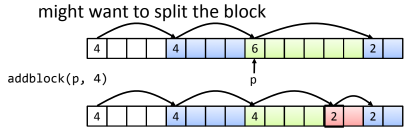
```
struct {
    unsigned allocated:1;
    unsigned size:31;
} Header;
```

Unfortunately, neither portable nor reliable ☹

28

Implicit list: allocating in a free block

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

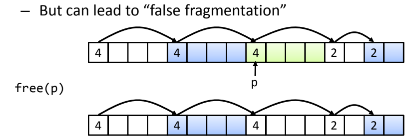



```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining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

29

Implicit list: freeing a block

- Simplest implementation:
  - Need only clear the “allocated” flag
    `void free_block(ptr p) { *p = *p & -2 }`
  - But can lead to “false fragmentation”



`malloc(5)` *Oops!*

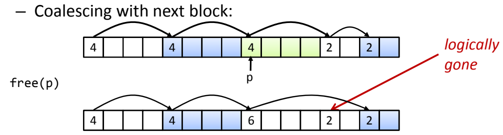
There is enough free space, but the allocator won’t be able to find it

30

10.3: Coalescing

Implicit list: coalescing

- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block:

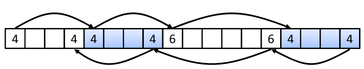


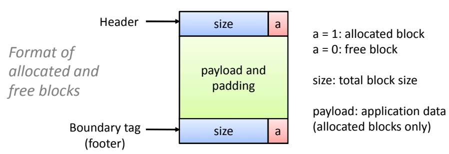

```
void free_block(ptr p) {
    *p = *p & -2;            // clear allocated flag
    next = p + *p;           // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;       // add to this block if
}                            //    not allocated
```

  - But how do we coalesce with *previous* block?

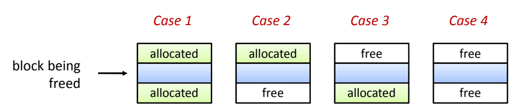
Implicit list: bidirectional coalescing

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!

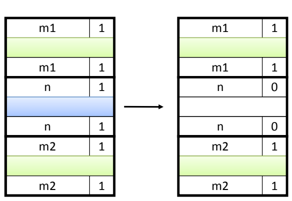




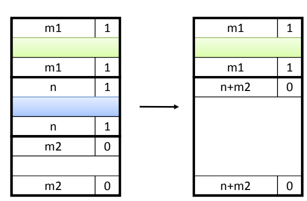
Constant time coalescing



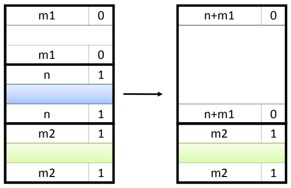
Constant time coalescing: case 1



Constant time coalescing: case 2

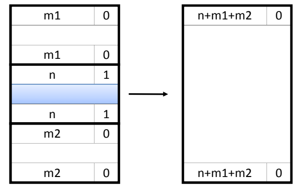


Constant time coalescing: case 3



37

Constant time coalescing: case 4



38

Disadvantages of boundary tags

- Internal fragmentation
- Can it be optimized?
  - Which blocks need the footer tag?
  - What does that mean?

39

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 to 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 each time `free()` is called
  - *Deferred coalescing:* try to improve performance of `free()` by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for `malloc()`
    - Coalesce when the amount of external fragmentation reaches some threshold

40

Implicit lists: summary

- Implementation: very simple
- Allocate cost: linear time worst case
- Free cost: 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 allocation
  - used in many special purpose applications
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators

41

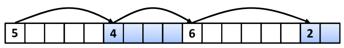
10.4: Explicit free lists

42

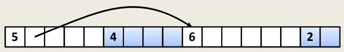
Keeping track of free blocks

Systems@ETH Zürich

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers

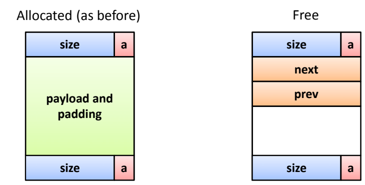


- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

43

Explicit free lists

Systems@ETH Zürich



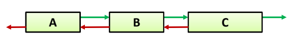
- Maintain list(s) of *free* blocks, not *all* blocks
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

44

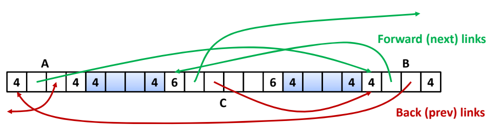
Explicit free lists

Systems@ETH Zürich

- Logically:



- Physically: blocks can be in any order

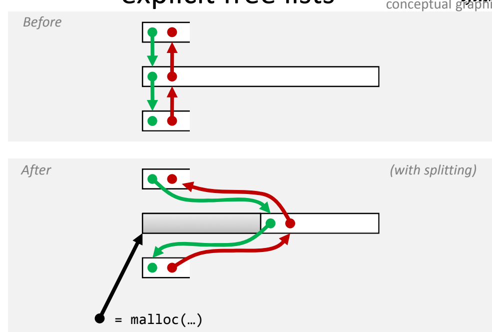


45

Allocating from explicit free lists

Systems@ETH Zürich

conceptual graphic



46

Freeing with explicit free lists

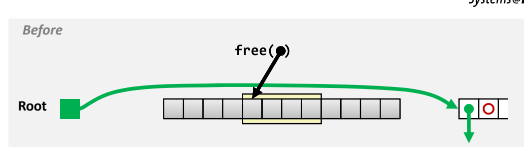
Systems@ETH Zürich

- *Insertion policy*: Where in the free list do you put a newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - ***Pro:*** simple and constant time
    - ***Con:*** studies suggest fragmentation is worse than address ordered
  - Address-ordered policy
    - Insert freed blocks so that free list blocks are always in address order: *addr(prev) < addr(curr) < addr(next)*
    - ***Con:*** requires search
    - ***Pro:*** studies suggest fragmentation is lower than LIFO

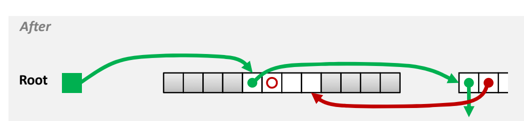
47

Freeing with LIFO policy: case 1

Systems@ETH Zürich



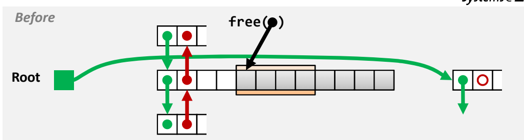
- Insert the freed block at the root of the list



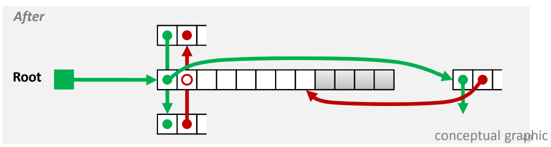
conceptual graphic

48

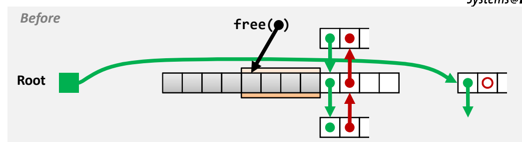
Freeing with LIFO policy: case 2



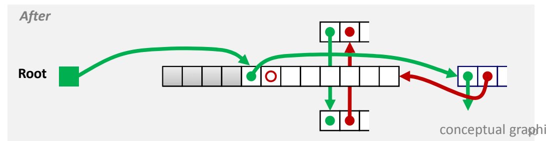
- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



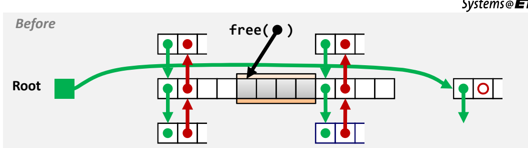
Freeing with LIFO policy: case 3



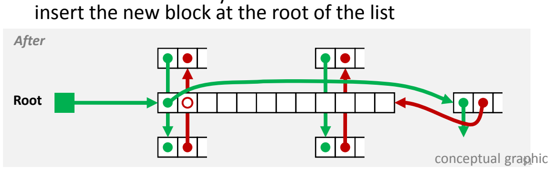
- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list



Freeing with LIFO policy: case 4



- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list



Explicit lists: summary

- Comparison to implicit list:
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?
- Most common use of linked lists is in conjunction with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

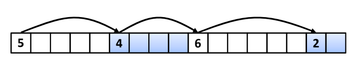
52

10.5: Segregated free lists

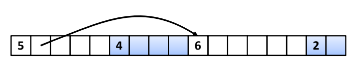
53

Keeping track of free blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers

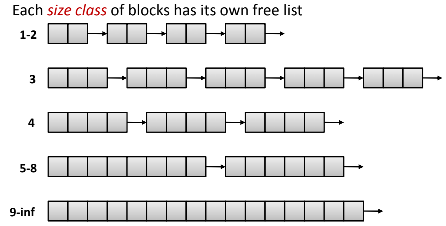


- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

54

Segregated list ("seglist") allocators

- Each *size class* of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

Seglist allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size *n*:
  - Search appropriate free list for block of size *m* > *n*
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using `sbrk()`)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.

Seglist allocator

- To free a block:
  - Coalesce and place on appropriate list (optional)
- Advantages of seglist allocators
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each block its own size class is equivalent to best-fit.

More info on allocators

- D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation
- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)

10.6: Garbage Collection

Implicit memory management: Garbage collection

- *Garbage collection:* automatic reclamation of heap-allocated storage—application never has to free

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

- Common in functional languages, scripting languages, and modern object oriented languages:
  - Lisp, ML, Java, Perl, Mathematica
- Variants ("conservative" garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

Garbage Collection

- How does the memory manager know when memory can be freed?
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them
- Must make certain assumptions about pointers
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers (e.g., by coercing them to an `int`, and then back again)

61

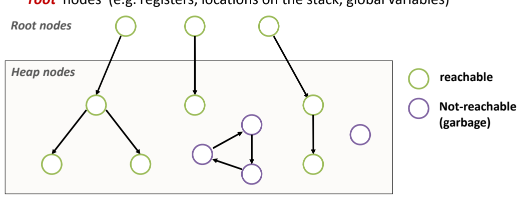
Classical GC algorithms

- Mark-and-sweep collection (McCarthy, 1960)
  - Does not move blocks (unless you also "compact")
- Reference counting (Collins, 1960)
  - Does not move blocks (not discussed)
- Copying collection (Minsky, 1963)
  - Moves blocks (not discussed)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Collection based on lifetimes
    - Most allocations become garbage very soon
    - So focus reclamation work on zones of memory recently allocated
- For more information: Jones and Lin, "*Garbage Collection: Algorithms for Automatic Dynamic Memory*", John Wiley & Sons, 1996.

62

Memory as a graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)



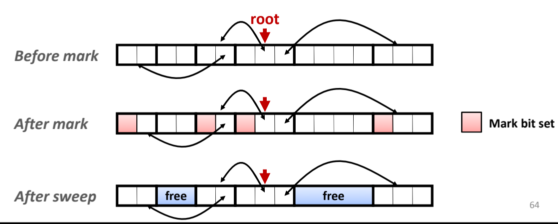
A node (block) is *reachable* if there is a path from any root to that node.

Non-reachable nodes are *garbage* (cannot be needed by the application)

63

Mark and Sweep collecting

- Can build on top of malloc/free package
  - Allocate using malloc until you "run out of space"
- When out of space:
  - Use extra ***mark bit*** in the head of each block
  - ***Mark:*** Start at roots and set mark bit on each reachable block
  - ***Sweep:*** Scan all blocks and free blocks that are not marked



64

Assumptions for a simple implementation

- Application
  - `new(n)`: returns pointer to new block with all locations cleared
  - `read(b,i)`: read location `i` of block b into register
  - `write(b,i,v)`: write v into location `i` of block b
- Each block will have a header word
  - addressed as `b[-1]`, for a block b
  - Used for different purposes in different collectors
- Instructions used by the Garbage Collector
  - `is_ptr(p)`: determines whether p is a pointer
  - `length(b)`: returns the length of block b, not including the header
  - `get_roots()`: returns all the roots

65

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // do nothing if not pointer
   if (markBitSet(p)) return;     // check if already marked
   setMarkBit(p);                 // set the mark bit
   for (i=0; i < length(p); i++)  // call mark on all words
     mark(p[i]);                  //   in the block
   return;
}
```

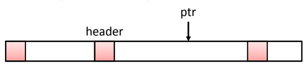
Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

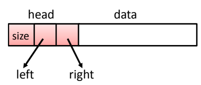
66

Conservative Mark & Sweep in C

- A "conservative garbage collector" for C programs
  - `is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But, in C pointers can point to the middle of a block



- So how to find the beginning of the block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in header (use two additional words)



Left: smaller addresses
Right: larger addresses

67

10.7: Memory pitfalls

68

Memory-related perils and pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

69

Dereferencing bad pointers

- The classic `scanf` bug

```
int val;

...

scanf("%d", val);
```

70

Reading uninitialized memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

71

Overwriting memory

- Allocating the (possibly) wrong sized object

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

72

Overwriting memory

- Off-by-one error

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```


Overwriting memory

- Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks
  - 1988 Internet worm
  - Modern attacks on Web servers
  - AOL/Microsoft IM war

Overwriting memory

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```


Referencing nonexistent variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
   int val;

   return &val;
}
```


Freeing blocks multiple times

- Nasty!

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```


Referencing freed blocks

- Evil!

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Failing to free blocks: Memory leaks

- Slow, long-term killer!

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```


Memory leaks

- Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
    ...
   free(head);
   return;
}
```


Overwriting memory

- Referencing a pointer instead of object it points to

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```


Finding memory bugs

- Conventional debugger (gdb)
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Debugging `malloc` (e.g. UToronto CSRI `malloc`)
  - Wrapper around conventional `malloc`
  - Detects memory bugs at `malloc` and `free` boundaries
    - Memory overwrites that corrupt heap structures
    - Some instances of freeing blocks multiple times
    - Memory leaks
  - Cannot detect all memory bugs
    - Overwrites into the middle of allocated blocks
    - Freeing block twice that has been reallocated in the interim
    - Referencing freed blocks

Finding memory bugs

- Some malloc implementations contain checking code
  - Linux glibc malloc: `setenv MALLOC_CHECK_ 2`
  - FreeBSD: `setenv MALLOC_OPTIONS AJR`
- Binary translator: valgrind (Linux), Purify
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Can detect all errors as debugging `malloc`
  - Can also check each individual reference at runtime
    - Bad pointers
    - Overwriting
    - Referencing outside of allocated block
- Garbage collection (Boehm-Weiser Conservative GC)
  - Let the system free blocks instead of the programmer.

10.8: Worms

Worms and viruses

- Worm: A program that
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- Virus: Code that
  - Add itself to other programs
  - Cannot run independently
- Both are (usually) designed to spread among computers and to wreak havoc

Early worms

- Term coined in 1975 by John Brunner
  - First "cyberpunk" novel: The Shockwave Rider
- Mid-1970s: research into benign worms at BBN and Xerox PARC
  - Network of Alto machines at PARC
  - Shoch, J. F. and Hupp, J. A. 1982. The "worm" programs—early experience with a distributed computation. Commun. ACM 25, 3 (Mar. 1982), 172-180.
- November 1988: Robert Morris' Worm
  - First Internet worm, attacked thousands of hosts
  - Morris now professor of Computer Science at MIT
  - Awarded the SIGOPS Mark Weiser award recently

… and the rest is history.

String library code

- Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - No way to specify limit on number of characters to read
- Similar problems with other Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

Vulnerable buffer code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

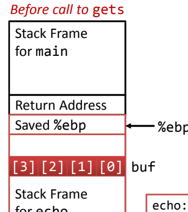

Buffer overflow disassembly

```
080484f0 <echo>:
 80484f0:   55                 push   %ebp
 80484f1:   89 e5              mov    %esp,%ebp
 80484f3:   53                 push   %ebx
 80484f4:   8d 5d f8           lea    0xfffffff8(%ebp),%ebx
 80484f7:   83 ec 14           sub    $0x14,%esp
 80484fa:   89 1c 24           mov    %ebx,(%esp)
 80484fd:   e8 ae ff ff ff     call   80484b0 <gets>
 8048502:   89 1c 24           mov    %ebx,(%esp)
 8048505:   e8 8a fe ff ff     call   8048394 <puts@plt>
 804850a:   83 c4 14           add    $0x14,%esp
 804850d:   5b                 pop    %ebx
 804850e:   c9                 leave
 804850f:   c3                 ret
```

```
 80485f2:   e8 f9 fe ff ff     call   80484f0 <echo>
 80485f7:   8b 5d fc           mov 0xfffffffc(%ebp),%ebx
 80485fa:   c9                 leave
 80485fb:   31 c0              xor    %eax,%eax
 80485fd:   c3                 ret
```


Buffer overflow stack

Before call to **gets**

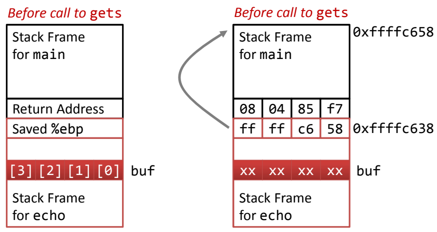



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp               # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx               # Save %ebx
    leal  -8(%ebp),%ebx      # Compute buf as %ebp-8
    subl  $20, %esp          # Allocate stack space
    movl  %ebx, (%esp)       # Push buf on stack
    call  gets               # Call gets
    . . .
```

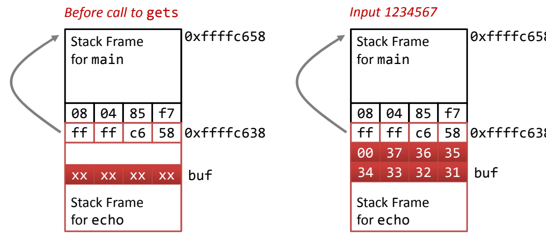
Buffer overflow stack example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffc658
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f7
```



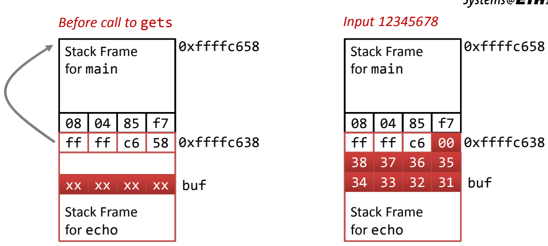

```
80485f2: call 80484f0 <echo>
80485f7: mov  0xfffffffc(%ebp),%ebx # Return Point
```


Buffer overflow example #1



Overflow buf, but no problem

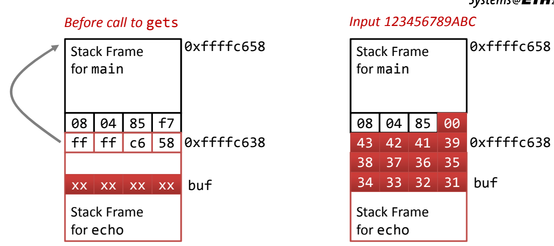
Buffer overflow example #2



Base pointer corrupted

```
. . .
804850a:  83 c4 14  add $0x14,%esp     # deallocate space
804850d:  5b        pop %ebx           # restore %ebx
804850e:  c9        leave              # movl %ebp, %esp; popl %ebp
804850f:  c3        ret                # Return
```


Buffer overflow example #3



Return address corrupted

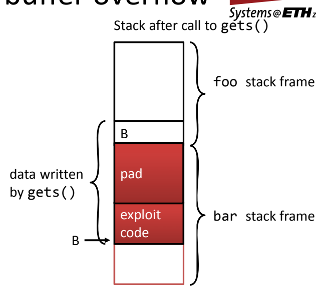
```
80485f2:  call 80484f0 <echo>
80485f7:  mov  0xfffffffc(%ebp),%ebx # Return Point
```


Malicious use of buffer overflow

```
void foo(){
  bar();
  ...
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `bar()` executes `ret`, will jump to exploit code

Exploits based on buffer overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- Internet worm (one vector)
  - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
    - `finger droh@cs.cmu.edu`
  - Worm attacked fingerd server by sending phony argument:
    - `finger "exploit-code padding new-return-address"`
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Avoiding overflow vulnerability

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- Use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don't use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` where n is a suitable integer

System-level protections

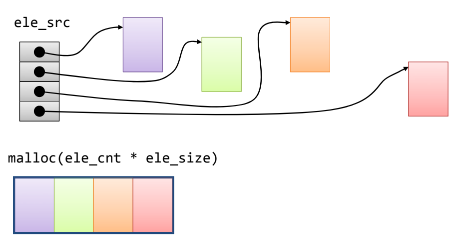
- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code
- Nonexecutable code segments
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - Add explicit "execute" permission

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```


Code security example #2

- SUN XDR library
  - Widely used library for transferring data between machines

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



XDR code

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```


XDR vulnerability

```
malloc(ele_cnt * ele_size)
```

- What if:
  - `ele_cnt` $= 2^{20} + 1$
  - `ele_size` $= 4096 = 2^{12}$
  - Allocation = ??
- How can I make this function secure?