COSC345 2013
Software Engineering

Lectures 14 and 15: The Heap and Dynamic Memory Allocation
Outline

• Revision
• The programmer’s view of memory
• Simple array-based memory allocation
• C memory allocation routines
• Virtual memory
• Swapping
• Objective C
Revision

• From previous lectures
  – Using the von Neumann architecture
  – Linear address space
  – Every memory location exists
  – Stack is at top of memory
  – Program is at bottom of memory

• Also
  – At compile time
    • Program size is known
    • The amount of space needed for global variables is known

• So
  – The remaining unused memory is available
Computer Memory

Hardware Model
68HC11

Software Model
(PDP-11 Unix)
Program Segments

• Program
  – Low in memory (the code or text segment)

• Local Variables
  – On the stack (the stack segment)

• Global variables
  – Above the program and before the stack (the data segment)

• The Heap
  – Everything else (above the break & below the stack)
The Problems

• Don’t know how big a given structure will be until the program is running

• Don’t know how much memory is needed before the program starts

• Not enough physical memory so must re-use it

• We need some kind of re-usable space
  – Memory allocation and deallocation
Array-Based Stacking Allocation

- Hand out pieces of a large character array

```
allocbuf[10000]
```

- \( p = \text{alloc}(10) \)
Array-Based Stacking Allocation

#define ALLOCSIZE 10000   /* size of available space */
static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf;    /* next free position */

char *alloc(size_t n) { /* return pointer to n characters */
    if (allocp + n <= &allocbuf[ALLOCSIZE]) {
        allocp += n; /* it fits, so claim that space */
        return allocp - n; /* old p */
    } else { /* not enough room */
        return NULL;
    }
}
Array-Based Stacking Free

- \( p = \text{alloc}(10) \)

\[
\begin{array}{c}
\text{allocbuf[10000]} \\
\begin{array}{ccc}
\text{Used} & \text{10 bytes} & \text{Unused} \\
\hline
p & \text{Top of Used Space (allocp)}
\end{array}
\end{array}
\]

- \( \text{afree}(p) \)

\[
\begin{array}{c}
\text{allocbuf[10000]} \\
\begin{array}{ccc}
\text{Used} & \text{Unused} & \text{Unused} \\
\hline
\text{Top of Used Space (allocp)}
\end{array}
\end{array}
\]
void afree(char *p) { /* free storage pointed to by p */
    if (p >= &allocbuf[0] && p < &allocbuf[ALLOCSIZE])
        allocp = p;
}

This is nearly perfect if a newer object never needs to outlive an older one. It’s effectively the two-stack method of implementing dynamic arrays from Lecture 13.
Example

• What is the output of this program?

```c
int main(int argc, char **argv) {
    char *p, *q;

    p = alloc(1);
    q = alloc(2);
    q[0] = 'q';
    p[0] = 'p';
    p[1] = 'p';
    printf("%c %c\n", p[0], q[0]);
    return 0;
}
```
Example

• What is the output of this program?

```c
int main(int argc, char *argv[]) {
    char *p, *q, *r;

    p = alloc(1);
    p[0] = 'p';
    q = alloc(2);
    q[0] = 'q';
    q[1] = 'q';
    afree(p);
    r = alloc(3);
    r[0] = 'r';
    r[1] = 'r';
    r[2] = 'r';
    printf("%c %c\n", q[1], r[2]);
    return 0;
}
```
Example

• What is the output of this program?

```c
int main(int argc, char **argv) {
    char *p;
    int *q;

    //p = alloc(1);
    q = (int *)alloc(sizeof *q);
    *q = 123456;
    printf("%d\n", *q);
    return 0;
}
```

• What if the comment is removed?
• Is the output *always* the same?
Problems

• Must free in reverse of allocation order

• Allocated memory is not aligned correctly (this one is easy to fix)

• Memory overwrites cause data corruption

• Heap overflow causes stack corruption
  – And stack overflow causes heap corruption
Allocate and Free

• The array-based stacker is good when we
  – Want to allocate many small units and free all at once
  – Never need to free the memory
    • The O/S does it for you when your program terminates

• General purpose allocation needs
  – A container of unused (free) memory pieces
  – Details of the allocation size of a piece of memory
  – To correctly align allocated memory
  – Some sort of debugging facility
The Header

- To keep a free list we need to know how big the allocated unit was

```c
struct header {
    struct header *ptr;  /* next block if on free list */
    size_t size;         /* size of this block */
};
```

- Allocate enough for user and the header

```c
    needed = sizeof(struct header) + n_bytes;
```

- When allocating return a “fake” to the user

The Free List

• Chain the blocks together when free is called
  – Chunks kept in order by increasing addresses
  – Merge adjacent blocks into one larger block

• There are alternative (faster!) solutions
  – Buckets (free lists for specific sizes)
Possible Allocation Schemes

• Best Fit
  – Scan the entire list
  – Use the smallest block large enough
  – Leaves large free areas untouched
  – Tends to leave small holes in the address space (fragmentation)
  – Slow unless you use a Cartesian tree

• First Fit / Next Fit
  – Use the first block in the list
  – Fragmentation no worse than best fit
  – Fast

• Other schemes
  – Worst fit, the buddy system, etc.
Word Alignment

- Some hardware and OSs need “word” alignment
  - Force the header to be aligned on worst case boundary

```c
typedef double Align;         /* strongest alignment likely needed */
typedef union header {         /* block header */
    struct {                   /* compare slide 16 */
        union header *ptr;     /* next block if on free list */
        size_t size;           /* size of this block */
    } s;
    Align x;                  /* force alignment of blocks */
} header;                     /* union tag & typedef should be == */
```
Allocate Chunks

• Make sure all allocation is of header sized chunks

\[
n_{\text{units}} = \frac{(n_{\text{bytes}} + \text{sizeof (header)} - 1)}{\text{sizeof (header)}};
\]

• And make room for the header itself

\[
n_{\text{units}} = \frac{(n_{\text{bytes}} + \text{sizeof (header)} - 1)}{\text{sizeof (header)}} + 1;
\]

• Now everything will be aligned in units of header

• A free list is needed

\[
\text{static header } *\text{freep} = \text{NULL}; \quad /* \text{start of free list } */
\]

• Use a sentinel

\[
\text{static header } \text{base}; \quad /* \text{empty list to get started } */
\]
void *malloc(size_t n_bytes) {
    /* malloc: general-purpose storage allocator */
    header *p, *prevp;
    size_t n_units;

    n_units = (n_bytes + sizeof (header) - 1) / sizeof (header) + 1;
    if ((prevp = freep) == NULL) { /* no free list yet */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= n_units) { /* big enough */
            if (p->s.size == n_units) { /* exactly */
                prevp->s.ptr = p->s.ptr;
            } else { /* allocate tail end */
                p->s.size -= n_units;
                p += p->s.size;
                p->s.size = n_units;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep) { /* wrapped around free list */
            p = (header *)more_core(n_units); /* ask OS to stretch data area */
            if (p == NULL) return NULL; /* not enough memory left */
        }
    }
    return NULL;
}
Morecore()

- Gets more memory from OS – calls sbrk()
  - Might, though, call alloc() (from earlier)

#define NALLOC 1024 /* minimum #units to request */
static header *more_core(size_t nu) { /* ask system for more memory */
  char *cp;
  header *up;

  if (nu < NALLOC) nu = NALLOC;
  cp = sbrk(nu * sizeof (header)); /* add to the break */
  if (cp == (char *) -1) /* no space at all */
    return NULL;

  up = (header *)cp;
  up->s.size = nu;

  free((void *)(up+1)); /* free subtracts the 1 again */
  return freep;
}
Free()

```c
void free(void *ap) { /* free: put block ap in free list */
    header *bp, *p;

    bp = (header *)ap - 1;                      /* point to block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;                               /* freed block at start or end of arena */

    if (bp + bp->s.size == p->s.ptr) {        /* join to next block */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else {
        bp->s.ptr = p->s.ptr;
    }

    if (p + p->s.size == bp) {                /* join to prev block */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else {
        p->s.ptr = bp;
    }

    freep = p;
}
```

Does the linear scan scare you? Modern malloc()s don’t do that. Come to that, the “boundary tag” method meant old ones didn’t either.
Problems

• Can overwrite the header (underrun)
  – Corrupts the header (size becomes corrupt)
• Can write past the end of a block of memory (overrun)
  – Will write into whatever is there
• Can overwrite the free list (heap corruption)
  – The list of free memory chunks becomes corrupt
• Can use pointers to free memory (dangling pointer)
  – Can write to a piece of memory that has been reused
• Can forget to free memory (memory leak)
  – Eventually memory will all be used and heap enters stack
  – Our malloc() leaks!
More Problems

• Can free the same block twice (double free)
• Can fragment the heap (heap fragmentation)
  – Malloc() fails even though there’s loads of memory
• Can run out of memory (out of memory)
  – Often caused by a leak
Other Allocation Routines

- **void *calloc(size_t count, size_t size)**
  - Call malloc() then set memory to all zero
    
    ```
    return memset(malloc(count * size), 0, count * size);
    ```

- **void *realloc(void *ptr, size_t size)**
  - Resize a block
    - If block needs to be larger,
      ```
      ans = memcpy(malloc(size), size, ptr);
      free(ptr);
      return ans;
      ```
    - Otherwise return ptr;
    - ptr==NULL is treated just like malloc(size)

- **char *strdup(char const *str) /* not in C std! */**
  - Allocate space for a string and copy it there
    ```
    return strcpy((char *)malloc(strlen(str) + 1), str);
    ```
Virtual Memory

• Address space divided into pages
• The MMU maps from virtual to physical addresses
• Non-existing address access causes interrupt (page fault)
• Interrupt service routine called
  – Segmentation violation
• Extend virtual address space
  – Add physical pages to virtual address space
  – UNIX: brk() (absolute), sbrk() (relative), mmap()
Swapping

- When there isn’t enough physical memory the disk is used to hold pages.

- When a page fault occurs the OS “swaps” a page in memory for one on disk.

- OS is constantly trying to free up rarely-used physical memory in favor of often-used virtual pages.
The Hard Disc Drive

- Parts of a hard disc drive
  - Platter
    - Magnetic disc on which information is stored (both sides)
  - Head
    - Physical mechanism that reads and writes to the platter

- Disk operations
  - Seek
    - Move the head to the right location on the disk
  - Read
    - Read information from the disk
  - Write
    - Write information to the disk
Swapping

• Disc access rate
  – Transfer Rate from 44.2 MB/s to 111.4 MB/s
  – Random access time: from 5 ms to 15 ms
    • That’s 512 bytes in 5ms (which is ~100KB/s)

• Physical memory access rate
  – DDR-SDRAM 200 MHz using DDR-400 chips
    • 3.200 GB/s bandwidth per channel

• Physical memory is ~30 times faster than disc
  – Include the random disk seek and it’s ~35,000 times!

• Perhaps we should write programs that don’t swap
Reference Counting

- Each dynamically allocated object has a counter
- Create a new reference => increment counter
- Remove a reference => decrement counter
- Counter reaches 0 => remove all references contained in it (compiler-generated code) release other resources like open files, then free it
- Memory reclaimed promptly; Inferno chose this approach for closing windows
- Free cascades => free can take a long time
- Simple approach doesn’t handle cycles
- ls –l shows link count (= ref count) as column 2
Objective C

- Creation takes two steps: `[[class alloc] init...]`
- Allocation allocates; initialisation may return that object or another.
- Originally used *manual reference counting*.
- `[object retain]` increments a counter.
- `[object release]` decrements and frees the object if the counter is zero.
- It is your problem to decide when to do this.
- Beware: `[x release]; x = [y retain]` does the wrong thing if `x == y`. Use `[y retain]; [x release]; x = y`.
- *Most* examples I’ve seen get that wrong!
Objective C 2

- Objective C on Mac OS X and iOS now supports *automatic reference counting*. The reference counting operations still happen, cycles are still a problem, but the compiler (usually) inserts them for you.

- Objective C on Mac OS X but *not* iOS supports *garbage collection*, almost like Java. Cycles are no problem, but a *write barrier* is used, so assignment statements are expensive. Living with C means that *conservative* collection has to be used, which can mean the occasional leak.
References

• B. Kernighan and D. Ritchie, *The C Programming Language*, Chapters 5 & 8

• D. Knuth, *The art of computer programming Volume 1 / Fundamental Algorithms* (2nd ed.), Section 2.5


Code samples adapted from: B. Kernighan and D. Ritchie, *The C Programming Language*