CS 61C:
Great Ideas in Computer Architecture
C Memory Management, Usage Models

Instructors:

Nicholas Weaver & Vladimir Stojanovic

http://inst.eecs.Berkeley.edu/~cs61c/sp16

Pointer Ninjitsu: Pointers to Functions

- You have a function definition
  - **`char *foo(char *a, int b){ … }`**
- Can create a pointer of that type…
  - **`char *(*f)(char *, int);`**
    - Declares f as a function taking a char * and an int and returning a char *
- Can assign to it
  - **`f = &foo`**
    - Create a reference to function foo
- And can then call it..
  - **`printf("%s\n", (*f)("cat", 3))`**

Managing the Heap

C supports functions for heap management:

- **`malloc()`** allocate a block of uninitialized memory
- **`calloc()`** allocate a block of zeroed memory
- **`free()`** free previously allocated block of memory
- **`realloc()`** change size of previously allocated block
  - careful – it might move!

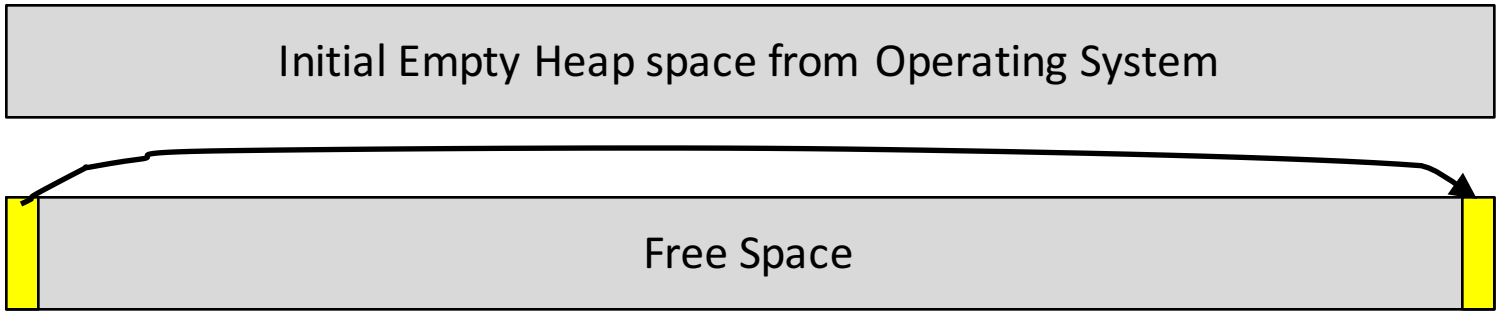
Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- *Managing the heap is tricky*: memory can be allocated / deallocated at any time

How are Malloc/Free implemented?

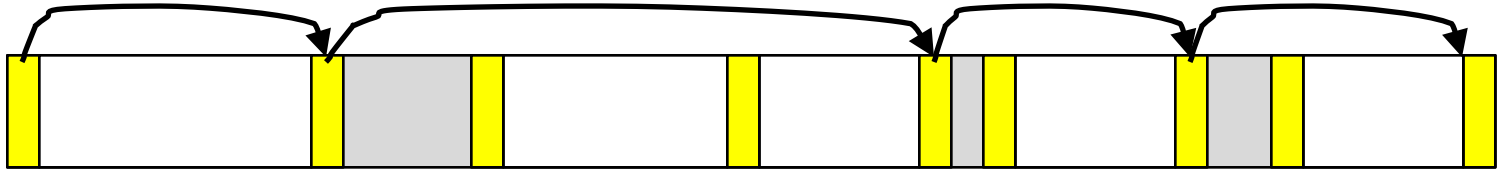
- Underlying operating system allows **`malloc`** library to ask for large blocks of memory to use in heap (e.g., using Unix **`sbrk()`** call)
- C standard **`malloc`** library creates data structure inside unused portions to track free space

Simple Slow Malloc Implementation



Malloc library creates linked list of empty blocks (one block initially)

First allocation chews up space from start of free space



After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list – might merge with neighboring free space

Clicker Question

- What will the following print:
  - ```
    int a, b, c, *d;
    a = 0;
    b = 1;
    c = 2;
    d = &a;
    (*d) += b + c;
    d = &b;
    (*d) += a + b + c;
    printf("a=%i b=%i\n", a, b);
    ```
  - A) a=0, b=3
  - B) a=3, b=3
  - C) a=3, b=4
  - D) a=3, b=7
  - E) I love pointers and am having a friend borrow my clicker
    ```

Administrivia…

- Project 1 should be out…
  - Getting 80%: Working on “correctly formatted” input should be straightforward
    - But be sure to test far more exhaustively than the provided test case
  - Getting 100% will be considerably harder...
    - A lot of corner cases you need to consider

Faster malloc implementations

- Keep separate pools of blocks for different sized objects
- “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:

Power-of-2 “Buddy Allocator”

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^3							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

Malloc Implementations

- All provide the same library interface, but can have radically different implementations
- Uses headers at start of allocated blocks and/or space in unallocated memory to hold **`malloc`**'s internal data structures
- Rely on programmer remembering to free with same pointer returned by **`malloc`**
- Rely on programmer not messing with internal data structures accidentally!
  - If you get a crash in **`malloc`**, it means that ***somewhere else*** you wrote off the end of an array

Common Memory Problems

- Using uninitialized values
  - Especially bad to use uninitialized pointers
- Using memory that you don't own
  - Deallocated stack or heap variable
  - Out-of-bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
- Memory leaks (you allocated something you forgot to later free)

Using Memory You Don't Own

- What is wrong with this code?

```
int *ipr, *ipw;
void ReadMem() {
      int i, j;
      ipr = (int *) malloc(4 * sizeof(int));
       i = *(ipr - 1000); j = *(ipr + 1000);
      free(ipr);
   }

   void WriteMem() {
      ipw = (int *) malloc(5 * sizeof(int));
      *(ipw - 1000) = 0; *(ipw + 1000) = 0;
      free(ipw);
   }
```

Using Memory You Don't Own

- Using pointers beyond the range that had been malloc'd
  - May look obvious, but what if mem refs had been result of pointer arithmetic that erroneously took them out of the allocated range?

```
int *ipr, *ipw;
void ReadMem() {
   int i, j;
   ipr = (int *) malloc(4 * sizeof(int));
   i = *(ipr - 1000); j = *(ipr + 1000);
   /* Hopefully no crash, but remember Heartbleed? */
   free(ipr);
}

void WriteMem() {
   ipw = (int *) malloc(5 * sizeof(int));
   *(ipw - 1000) = 0; *(ipw + 1000) = 0;
   /* If you are lucky… It will crash right here. */
   free(ipw);
}
```

Faulty Heap Management

- What is wrong with this code?

```
int *pi;
void foo() {
   pi = malloc(8*sizeof(int));
   …
   free(pi);
}

void main() {
   pi = malloc(4*sizeof(int));
   foo();
   …
}
```

Faulty Heap Management

- Memory leak: *more mallocs than frees*

```
int *pi;
void foo() {
   pi = malloc(8*sizeof(int));
   /* Allocate memory for pi */
   /* Oops, leaked the old memory pointed to by pi */
   …
   free(pi);
}

void main() {
   pi = malloc(4*sizeof(int));
   foo(); /* Memory leak: foo leaks it */
  …
}
```

Faulty Heap Management

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
   plk = malloc(2 * sizeof(int));
   … … …
   plk++;
}
```

Faulty Heap Management

- Potential memory leak – handle (block pointer) has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
   plk = malloc(2 * sizeof(int));
   … … …
   plk++; /* Potential leak: pointer variable
   incremented past beginning of block!
   So how can you free it later?*/
}
```

Faulty Heap Management

- What is wrong with this code?

```
void FreeMemX() {
   int fnh = 0;
   free(&fnh);
}

void FreeMemY() {
   int *fum = malloc(4 * sizeof(int));
   free(fum+1);
   free(fum);
   free(fum);
}
```

Faulty Heap Management

- Can't free non-heap memory; Can't free memory that hasn't been allocated

```
void FreeMemX() {
   int fnh = 0;
   free(&fnh); /* Oops! freeing stack memory.  If lucky… */
}

void FreeMemY() {
   int *fum = malloc(4 * sizeof(int));
   free(fum+1);
   /* fum+1 is not a proper handle; points to middle
   of a block.  If lucky… */
   free(fum);
   free(fum);
   /* Oops! Attempt to free already freed memory.  If lucky…*/
}
```

Using Memory You Haven't Allocated

- What is wrong with this code?

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  printf("%s\n", str);
}
```

Using Memory You Haven't Allocated

- Reference beyond array bounds

```
void StringManipulate() {
   const char *name = "Safety Critical";
   char *str = malloc(10);
   strncpy(str, name, 10);
   str[10] = '\0';
   /* Write Beyond Array Bounds.  If you are
   lucky…  (Nick wasn't in 60c...) */
   printf("%s\n", str);
   /* Read Beyond Array Bounds */
}
```

Using Memory You Don't Own

- What's wrong with this code?

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
    result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
    result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

Using Memory You Don't Own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
    result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
    result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

Using Memory You Don't Own

- What is wrong with this code?

```
typedef struct node {
    struct node* next;
    int val;
} Node;

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
        head = head->next;
    }
    return head->val;
}
```

Using Memory You Don't Own

- Following a NULL pointer to mem addr 0!

```
typedef struct node {
    struct node* next;
    int val;
} Node;

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
    /* What if head happens to be NULL? */
       head = head->next;
    }
    return head->val; /* What if head is NULL? */
} /* In general, assume that your functions will be
called incorrectly, so explicitly check inputs rather
than rely on the caller checking inputs */
```

Managing the Heap

- `realloc(p,size)`:
  - Resize a previously allocated block at `p` to a new `size`
  - If `p` is `NULL`, then `realloc` behaves like `malloc`
  - If `size` is `0`, then `realloc` behaves like `free`, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

E.g.: allocate an array of 10 elements, expand to 20 elements later

```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = (int *) realloc(ip,20*sizeof(int));
/* always check for ip == NULL */
/* contents of first 10 elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}

int* fill_fibonacci(int *fib, int size) {
   int i;
   init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
    fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

Using Memory You Don't Own

- Improper matched usage of mem handles

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}

int* fill_fibonacci(int *fib, int size) {
   int i;
   /* oops, forgot: fib = */ init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
    fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?

Valgrind..

- Debugging memory problems in C is a right-royal-massive-unpritable-profine-pain-in-the-rear!
  - C doesn't just let you shoot yourself in the foot, but gives you an AK-47, points it downward, and invites you to starts spraying…
  - Many of the crashes ***do not occur*** where you make your mistakes!
- Valgrind is a tool which runs your program (much much ***much*** more slowly) in a way which checks memory accesses and performs other checks
  - http://valgrind.org/docs/manual/quick-start.html
- It is ***not perfect***
  - Rare false positives
  - Some large class false negatives
  - And test input must trigger the erroneous read or write

And In Conclusion, …

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code