CS 61C: Great Ideas in Computer Architecture

Memory Management and Usage

Instructor: Justin Hsia
Review of Last Lecture

• Arrays
  – Traverse using pointer or array syntax
  – If characters, call them “strings” and null-terminate

• Pointer arithmetic moves the pointer by the size of the thing it’s pointing to
  – C accommodates for pointing to structs and pointers

• From Discussion: Bit-masking
Great Idea #1: Levels of Representation/Interpretation

- Higher-Level Language Program (e.g., C)
  - Compiler
  - Assembly Language Program (e.g., MIPS)
    - Assembler
    - Machine Language Program (MIPS)
      - Machine Interpretation
        - Hardware Architecture Description (e.g., block diagrams)
        - Architecture Implementation
          - Logic Circuit Description (Circuit Schematic Diagrams)

Temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)

0000 1001 1100 0110 1010 1111 0101 1000 1010 1111 0101 1000 1010 1111 0101 1000 1010 1111
0101 1000 0000 1001 1100 0110 1010 1111

We are here
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Memory Management
• C Wrap-up: Linked List Example
• Bonus: Common Memory Problems
C Memory Layout

- Program’s address space contains 4 regions:
  - **Stack**: local variables, grows downward
  - **Heap**: space requested for pointers via malloc(); resizes dynamically, grows upward
  - **Static Data**: global and static variables, does not grow or shrink
  - **Code**: loaded when program starts, does not change

---

OS prevents accesses between stack and heap (via virtual memory)
Where Do the Variables Go?

• Declared outside a procedure:
  **Static Data**

• Declared inside a procedure:
  **Stack**
  – main() is a procedure
  – Freed when procedure returns

• Dynamically allocated:
  **Heap**
  – i.e. malloc (we will cover this shortly)

```c
#include <stdio.h>
int varGlobal;
int main() {
    int varProc;
    int *varDyn = malloc(sizeof(int));
}
```
The Stack

• Each stack frame is a contiguous block of memory holding the local variables of a single procedure

• A stack frame includes:
  – Location of caller function
  – Procedure arguments
  – Space for local variables

• Stack pointer (SP) tells where lowest (current) stack frame is

• When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames
The Stack

• Last In, First Out (LIFO) data structure

```c
int main() {
    a(0);
    return 1;
}

void a(int m) {
    b(1);
}

void b(int n) {
    c(2);
    e(4);
}

void c(int o) {
    printf("c");
}

void d(int p) {
    printf("d");
}
```

Stack Pointer grows down
Stack Misuse Example

int *getPtr() {
    int y;
    y = 3;
    return &y;
};

int main () {
    int *stackAddr, content;
    stackAddr = getPtr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    content = *stackAddr;
    printf("%d", content); /* 0 */
};

Never return pointers to local variable from functions

Your compiler will warn you about this – don’t ignore such warnings!

printf
overwrites
stack frame
Static Data

• Place for variables that persist
  – Data not subject to comings and goings like function calls
  – Examples: String literals, global variables

• Size does not change, but its data can

Code

• Copy of your code goes here
  – C code becomes data too!

• Does not change
Question: Which statement below is FALSE?
All statements assume each variable exists.

```c
void funcA() {int x; printf("A");}
void funcB() {
    int y;
    printf("B");
    funcA();
}
void main() {char *s = "s"; funcB();}
```

- x is at a lower address than y
- x and y are in adjacent frames
- x is at a lower address than *s
- y is in the 2nd frame from the top of the Stack
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Memory Management
• C Wrap-up: Linked List Example
• Bonus: Common Memory Problems
Administrivia

• Meet your fellow classmates! Form study groups and get your questions answered
  – Utilize Piazza, labs, discussions, and OHs
• End of the first week!
  – HW1 due Sunday
  – TA and Instructor response time increase drastically (slower) over the weekend
  – Check key card access to lab today
  – Justin’s OH (this week only): Sat @ 10-12 in lab
• Suggestion for weekend: finish HW1, try Lab3, then look at HW2Q1
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Memory Management
• C Wrap-up: Linked List Example
• Bonus: Common Memory Problems
Dynamic Memory Allocation

• Sometimes you don’t know how much memory you need beforehand
  – Examples: Input files, user input

• Dynamically allocated memory goes on the Heap – more permanent than Stack

• Need as much space as possible without interfering with Stack
  – Start at opposite end and grow towards Stack
Allocating Memory

• 3 functions for requesting memory:
  `malloc()`, `calloc()`, and `realloc()`

• `malloc(n)`
  – Allocates a continuous block of `n bytes` of uninitialized memory (contains garbage!)
  – Returns a pointer to the beginning of the allocated block; NULL indicates failed request (check for this!)
  – Different blocks not necessarily adjacent
Using malloc()

- Almost always used for arrays or structs
- Good practice to use `sizeof()` and typecasting
  
  ```c
  int *p = (int *) malloc(n*sizeof(int));
  ```
  
  - `sizeof()` makes code more portable
  
  - `malloc()` returns `void *`, typecast will help you catch coding errors when pointer types don’t match

- Can use array or pointer syntax to access
- Make sure you don’t lose the original address
  
  - `p++` is a BAD IDEA; use a separate pointer
Releasing Memory

• Release memory on the Heap using `free()`
  – Memory is limited, release when done

• `free(p)`
  – Pass it pointer `p` to beginning of allocated block; releases the whole block
  – `p` must be the address originally returned by `m/c/realloc()`, otherwise throws system exception
  – Don’t call `free()` on a block that has already been released
Dynamic Memory Example

- Need \texttt{#include <stdlib.h>}

\begin{verbatim}
typedef struct {
    int x;
    int y;
} point;

point *rect; /* opposite corners = rectangle */
...
if( !(rect=(point *) malloc(2*sizeof(point))) ) {
    printf("\nOut of memory!\n");
    exit(1);
}
...
\textbf{Do NOT change rect during this time!!!}
free(rect);
\end{verbatim}

Joint struct definition and typedef

Check for returned NULL

Joint struct definition and typedef

Do NOT change rect during this time!!!
#define N 3
int *makeArray(int n) {
    int *ar;
    ar = (int *) malloc(n);
    return ar;
}
void main() {
    int i, *a = makeArray(N);
    for (i = 0; i < N; i++)
        *a++ = i;
    printf("a[] = {%i, %i, %i}\n", a[0], a[1], a[2]);
    free(a);
}
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Memory Management
• C Wrap-up: Linked List Example
• Bonus: Common Memory Problems
Memory Management

• Many calls to `malloc()` and `free()` with many different size blocks – where are they placed?

• Want system to be fast with minimal memory overhead
  – Versus automatic garbage collection of Java

• Want to avoid *fragmentation*, the tendency of free space on the heap to get separated into small chunks
Fragmentation Example

1) Block 1: malloc(100)
2) Block 2: malloc(1)
3) Block 1: free(B1)
4) Block 3: malloc(50)
   – What if malloc(101)?
5) Block 4: malloc(60)
Basic Allocation Strategy: K&R

- Section 8.7 offers an implementation of memory management (linked list of free blocks)
  - If you can decipher the code, you’re well-versed in C!
- This is just one of many possible memory management algorithms
  - Just to give you a taste
  - No single best approach for every application
K&R Implementation

• Each block holds its own size and pointer to next block

• `free()` adds block to the list, combines with adjacent free blocks

• `malloc()` searches free list for block large enough to meet request
  - If multiple blocks fit request, which one do we use?
Choosing a Block in malloc()

• **Best-fit:** Choose smallest block that fits request
  - Tries to limit wasted fragmentation space, but takes more time and leaves lots of small blocks

• **First-fit:** Choose first block that is large enough (always starts from beginning)
  - Fast but tends to concentrate small blocks at beginning

• **Next-fit:** Like first-fit, but resume search from where we last left off
  - Fast and does not concentrate small blocks at front
Question: Which allocation system and set of requests will create a continuous block in the Heap? B3 was the last fulfilled request.

- **Best-fit:**
  \hspace{1cm} \texttt{malloc(50), malloc(50)}

- **First-fit:**
  \hspace{1cm} \texttt{malloc(50), malloc(30)}

- **Next-fit:**
  \hspace{1cm} \texttt{malloc(30), malloc(50)}

- **Next-fit:**
  \hspace{1cm} \texttt{malloc(50), malloc(30)}
Get To Know Your Instructor
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Memory Management
• C Wrap-up: Linked List Example
• Bonus: Common Memory Problems
Linked List Example

• We want to generate a linked list of strings
  – This example uses structs, pointers, `malloc()`, and `free()`

• Create a structure for nodes of the list:

```c
struct Node {
    char *value;
    struct Node *next;
};
```

Notice the recursive definition!
Simplify Code with typedef

- It gets annoying to type out `struct ListNode` over and over again
- Define new variable type for struct:

  Method 1:
  ```c
  struct Node {
      char *value;
      struct Node *next;
  };
  typedef struct Node ListNode;
  ```

  Method 2:
  ```c
  typedef struct Node {
      char *value;
      struct Node *next;
  } NodeStruct;
  ```

- Can further rename pointers:
  ```c
typedef ListNode * List; // List myLinkedList;
typedef char * String; // String value;
```
Adding a Node to the List

• Want functionality as shown:

```java
String s1 = "start", s2 = "middle";
String s3 = "end";
List theList = NULL;
theList = addNode(s3, theList);
theList = addNode(s2, theList);
theList = addNode(s1, theList);
```

In what part of memory are these stored?

Must be able to handle a NULL input

If you’re more familiar with Lisp/Scheme, you could name this function cons instead.
Adding a Node to the List

- Let's examine the 3rd call ("start"): 

  ```
  List addNode(String s, List list) {
      List node = (List) malloc(sizeof(NodeStruct));
      node->value = (String) malloc (strlen(s) + 1);
      strcpy(node->value, s);
      node->next = list;
      return node;
  }
  ```

Don’t forget this for the null terminator!
Removing a Node from the List

• Delete/free the first node ("start"): 

```c
List deleteNode(List list) {
    List temp = list->next;
    free(list->value);
    free(list);
    return temp;
}
```

What happens if you do these in the wrong order?
Additional Functionality

- How might you implement the following:
  - Append node to end of a list
  - Delete/free an entire list
  - Join two lists together
  - Reorder a list alphabetically (sort)
Summary

• C Memory Layout
  – **Static Data:** globals and string literals
  – **Code:** copy of machine code
  – **Stack:** local variables (grows & shrinks in LIFO manner)
  – **Heap:** dynamic storage using `malloc` and `free`
    The source of most memory bugs!

• Memory Management
  – Want fast with little *fragmentation*
You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture. They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.
Agenda

• C Memory Layout
  – Stack, Static Data, and Code
• Administrivia
• Dynamic Memory Allocation
  – Heap
• Memory Management
• C Wrap-up: Linked List Example
• Bonus: Common Memory Problems
Know Your Memory Errors
(Definitions taken from http://www.hyperdictionary.com)

• Segmentation Fault More common in 61C
  “An error in which a running Unix program attempts to access memory not allocated to it and terminates with a segmentation violation error and usually a core dump.”

• Bus Error Less common in 61C
  “A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error.”
Common Memory Problems

1) Using uninitialized values
2) Using memory that you don’t own
   – Using NULL or garbage data as a pointer
   – De-allocated stack or heap variable
   – Out of bounds reference to stack or heap array
3) Using memory you haven’t allocated
4) Freeing invalid memory
5) Memory leaks
Using Uninitialized Values

• What is wrong with this code?

    void foo(int *p) {
        int j;
        *p = j;
    }

    void bar() {
        int i=10;
        foo(&i);
        printf("i = %d\n", i);
    }

    j is uninitialized (garbage), copied into *p

    Using i which now contains garbage
Using Memory You Don’t Own (1)

• What is wrong with this code?

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

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

What if `head` is NULL?

No warnings! Just Seg Fault that needs finding!
What’s wrong with this code?

```c
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[MAXSIZE];
    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;
}
```

Local array appears on Stack

Pointer to Stack (array) no longer valid once function returns
Using Memory You Don’t Own (3)

- What is wrong with this code?

typedef struct {
    char *name;
    int age;
} Profile;

Profile *person =(Profile *)malloc(sizeof(Profile));
char *name = getName();
person.name = malloc(sizeof(char)*strlen(name));
strcpy(person.name,name);
...
  // Do stuff (that isn’t buggy)
free(person);
free(person.name);

Did not allocate space for the null terminator! Want `(strlen(name)+1)` here.

Accessing memory after you’ve freed it. These statements should be switched.
Using Memory You Haven’t Allocated

• What is wrong with this code?

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

Write beyond array bounds
Read beyond array bounds
Using Memory You Haven’t Allocated

• What is wrong with this code?

```c
char buffer[1024]; /* global */
int main(int argc, char *argv[]) {
    strcpy(buffer, argv[1]);
    ...
}
```

What if more than a kibi characters?

This is called BUFFER OVERRUN or BUFFER OVERFLOW and is a major security flaw!!!
Freeing Invalid Memory

• What is wrong with this code?

```c
void FreeMemX() {
    int fnh = 0;
    free(&fnh);  // 1) Free of a Stack variable
}

void FreeMemY() {
    int *fum = malloc(4*sizeof(int));
    free(fum+1);  // 2) Free of middle of block
    free(fum);
    free(fum);   // 3) Free of already freed block
}
```
Memory Leaks

• What is wrong with this code?

```c
int *pi;
void foo() {
    pi = (int*)malloc(8*sizeof(int));
    ... 
    free(pi);
}
void main() {
    pi = (int*)malloc(4*sizeof(int));
    foo();
}
```

Overrode old pointer!
No way to free those 4*sizeof(int) bytes now

foo() leaks memory
Memory Leaks

• **Rule of Thumb:** More `mallocs` than `frees` probably indicates a memory leak

• Potential memory leak: Changing pointer – do you still have copy to use with `free` later?

```c
plk = (int *)malloc(2*sizeof(int));
...
plk++;
```

Typically happens through incrementation or reassignment
Debugging Tools

• Runtime analysis tools for finding memory errors
  – Dynamic analysis tool: Collects information on memory management while program runs
  – No tool is guaranteed to find ALL memory bugs; this is a very challenging programming language research problem

• You will be introduced to Valgrind in Lab 3

http://valgrind.org