Lecture 6

Memory Management and more
Administrivia

- My office hours: Monday 1pm-2pm, 424 SDH.
- Rasberry PI servers online today!
Agenda

• Memory Management
• and more
C Memory Management

- How does the C compiler determine where to put all the variables in machine’s memory?
- How to create dynamically sized objects?
- To simplify discussion, we assume one program runs at a time, with access to all of memory.
- Later, we’ll discuss virtual memory, which lets multiple programs all run at same time, each thinking they own all of memory.
C Memory Management

- Program’s address space contains 4 regions:
  - **stack**: local variables inside functions, grows downward
  - **heap**: space requested for dynamic data via `malloc()` resizes dynamically, grows upward
  - **static data**: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - **code**: loaded when program starts, does not change

Memory Address (32 bits assumed here) 
\[\sim \text{FFFF FFFF}_{\text{hex}}\]
Where are Variables Allocated?

• If declared outside a function, allocated in “static” storage
• If declared inside function, allocated on the “stack” and freed when function returns
  • `main()` is treated like a function
• For both of these types of memory, the management is automatic:
  • You don't need to worry about deallocating when you are no longer using them

```c
int myGlobal;
main() {
    int myTemp;
}
```
The Stack

- Every time a function is called, a new frame is allocated on the stack
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables
- Stack frames use contiguous blocks of memory; stack pointer indicates start of stack frame
- When function ends, stack pointer moves up; frees memory for future stack frames
- We’ll cover details later for MIPS processor
Stack Animation

- Last In, First Out (LIFO) data structure

```c
main ()
{ a(0);
 }
void a (int m)
{ b(1);
 }
void b (int n)
{ c(2);
 }
void c (int o)
{ d(3);
 }
void d (int p)
{ }
```

Stack grows down

Stack Pointer
Managing the Heap

C supports functions for heap management:

- \texttt{malloc()} allocate a block of uninitialized memory
- \texttt{calloc()} allocate a block of zeroed memory
- \texttt{free()} free previously allocated block of memory
- \texttt{realloc()} change size of previously allocated block
  - careful – it might move!
  - And it \textbf{will not update other pointers pointing to the same block of memory}
Malloc()

- **void *malloc(size_t n):**
  - Allocate a block of uninitialized memory
  - NOTE: Subsequent calls probably will not yield adjacent blocks
  - **n** is an integer, indicating size of requested memory block in bytes
  - **size_t** is an unsigned integer type big enough to “count” memory bytes
  - Returns **void** pointer to block; **NULL** return indicates no more memory (check for it!)
  - Additional control information (including size) stored in the heap for each allocated block.

- **Examples:**
  - `int *ip;
    ip = (int *) malloc(sizeof(int));`
  - `typedef struct { ... } TreeNode;
    TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));`
  - **sizeof** returns size of given type in bytes, **necessary if you want portable code!**
And then free()

- **void free(void **p):**
  - p is a pointer containing the address originally returned by malloc()

- **Examples:**
  - ```
  int *ip;
  ip = (int *) malloc(sizeof(int));
  ... ...
  free((void*) ip); /* Can you free(ip) after ip++ ? */
  ```
  - ```
  typedef struct {... } TreeNode;
  TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
  ... ...
  free((void *) tp);
  ```

- When you free memory, you must be sure that you pass the original address returned from malloc() to free(); Otherwise, crash (or worse)!
Using Dynamic Memory

typedef struct node {
    int key;
    struct node *left; struct node *right;
} Node;

Node *root = NULL;

Node *create_node(int key, Node *left, Node *right){
    Node *np;
    if(!(np = (Node*) malloc(sizeof(Node))){
        printf("Memory exhausted!\n");
        exit(1);
    } else{
        np->key = key;
        np->left = left;
        np->right = right;
        return np;
    }
}

void insert(int key, Node **tree){
    if ((*tree) == NULL){
        (*tree) = create_node(key, NULL, NULL);
    } else if (key <= (*tree)->key){
        insert(key, &((*tree)->left));
    } else{
        insert(key, &((*tree)->right));
    }
}

int main(){
    insert(10, &root);
    insert(16, &root);
    insert(5, &root);
    insert(11, &root);
    return 0;
}
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
  - If you forget to deallocate memory: “Memory Leak”
    - Your program will eventually run out of memory
  - If you call free twice on the same memory: “Double Free”
    - Possible crash or exploitable vulnerability
  - If you use data after calling free: “Use after free”
    - Possible crash or exploitable vulnerability
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
Clickers/Peer Instruction!

```c
int x = 2;
int result;

int foo(int n)
{
    int y;
    if (n <= 0) { printf("End case!\n"); return 0; }
    else
    {
        y = n + foo(n-x);
        return y;
    }
}
result = foo(10);
```

Right after the `printf` executes but before the `return 0`, how many copies of `x` and `y` are there allocated in memory?

A: #x = 1, #y = 1
B: #x = 1, #y = 5
C: #x = 5, #y = 1
D: #x = 1, #y = 6
E: #x = 6, #y = 6