Announcement

- Assignment 6
  - Due November 11
  - Sorting: Combining C and IA-32 assembly

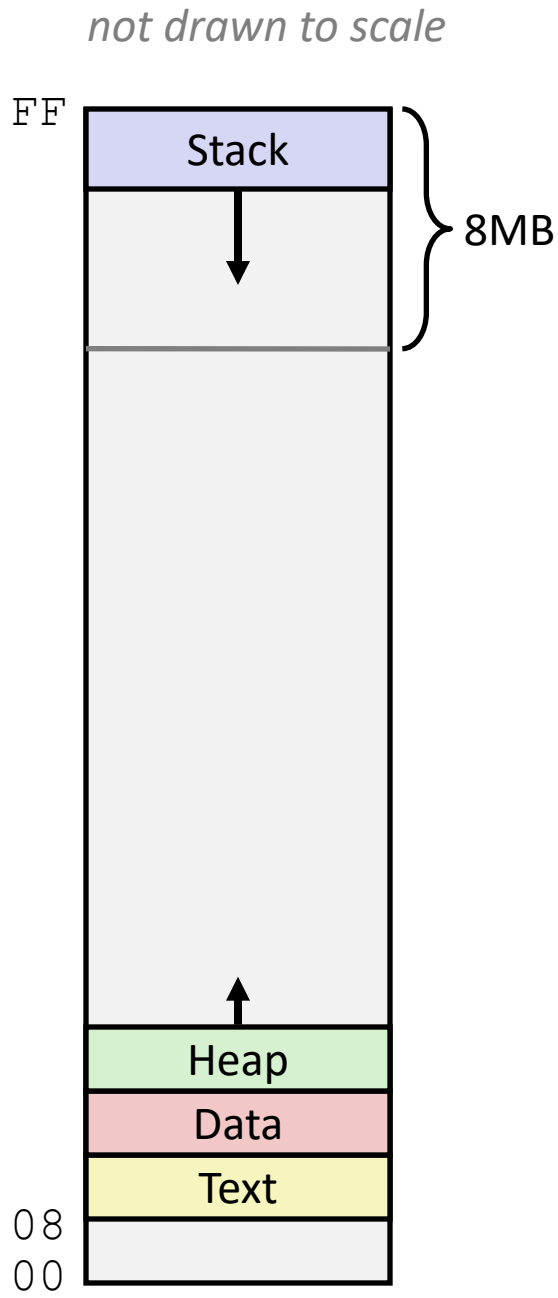
Lecture 24

Dynamic Memory

CPSC 275
Introduction to Computer Systems

Recall IA-32 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
  - e. g., local variables
- Heap
  - Dynamically allocated storage
  - When call `malloc`, `calloc`, `new`
- Data
  - Statically allocated data
  - e.g., globals & strings declared in code
- Text
  - Executable machine instructions
  - Read-only (*re-entrant*)



Upper 2 hex digits
= 8 bits of address

Types of memory allocation

- *Automatic* memory – stack frames created & destroyed per function call
- *Static* memory – variables with fixed size and lifetime (`global`, `static`)
- *Dynamic* memory – explicitly requested at runtime from the heap.
- Why dynamic?
  - Arrays or data structures whose size isn't known at compile time.
  - Lifetime independent of function scope.
  - Examples: text editors, compilers, operating systems.

Static memory allocation

```
void foo() {
    int x = 0;  // automatic variable
    x++;
    printf("x = %d\n", x);
}

void bar() {
    static int x = 0;  // static variable
    x++;
    printf("x = %d\n", x);
}

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

Output:

```
x = 1
x = 1
x = 1
x = 2
x = 1
x = 3
```

Dynamic memory in C

```
#include <stdlib.h>
void *malloc(size);
```

The malloc function allocates `size` number of bytes in the **heap** memory and returns a pointer to that memory.

Note: The return value is a `void` pointer, so we must cast it to a pointer of the type that we want.

Dynamic memory allocation

- To allocate an `int`:

```
int *ip;
ip = (int *) malloc(sizeof(int));
```

- To allocate a `char`:

```
char *cp = (char *) malloc(sizeof(char));
```

- To allocate a `struct`:

```
struct person *sp;
sp = (struct person *) malloc(sizeof(struct person));
```

Dynamic memory deallocation

When we've finished using dynamically allocated memory, we must ***free*** it:

```
void free(void *);
```

For example,

```
free(sp);
```

Failing to do so creates ***memory leakage.***

Dynamic memory allocation

```
#include <stdlib.h>
void *calloc(size);
void *realloc(ptr,newsize)
```

The `calloc` function allocates `size` number of bytes in the heap memory, **zero-fills**, and returns a pointer to that memory.

The `realloc` function resizes an existing memory (`ptr`) block to `newsize` and returns a pointer to that memory.

Allocating dynamic arrays

To allocate an array of *n* doubles:

```
double *arr;
. . .
arr = (double *) malloc(sizeof(double)*n);
```

To reference an array element:

- by index:

  ```
  arr[4]
  ```

- by address:

  ```
  *(arr + 4)
  ```

Allocating two-dimensional dynamic arrays

- Approach: allocate a block of memory as "1-dimensional" that's large enough to store our 2D array and treat it as stored in row-major order.
- If we need *n* rows and *m* columns and our data is of size *s*, then allocate *n* * *m** *s* bytes.

Example:

```
#define N 100
...
int *a;
...
a = (int *) malloc(N * N * sizeof(int));
...
free(a);
```

Using two-dimensional dynamic arrays

- To access 2D array elements created in this way, we ***can't*** use `a[j][k]` notation.
- However, we can use 2D subscript calculation, or use the pointer notation.

Example:

```
#define N 100
...
int *a;
int j, k;
...
a = (int *) malloc(N * N * sizeof(int));
...
a[j * N + k] = 17;  // or *(a + N*j + k)
free(a);
```

Common mistakes

- Forget to free – memory leak
- Double free – undefined behavior
- Use after free – segmentation fault
- Uninitialized read – random data
- Wrong size – segmentation fault
- Good practice:
  - Always set the pointer to NULL after free.
  - Use tools like `valgrind`.

Stack vs. Heap

Feature	**Stack**	**Heap**
Lifetime	automatic	manual
Location	grows down from `%esp`	grows up via `brk`/`sbrk`
Allocation time	very fast (pointer move)	slower (system calls)
Typical size	8 MB (default)	virtually unlimited
Errors	overflow	leak, fragmentation