



# Memory Allocation

Prof. David August  
COS 217

1

## Goals for Today's Lecture



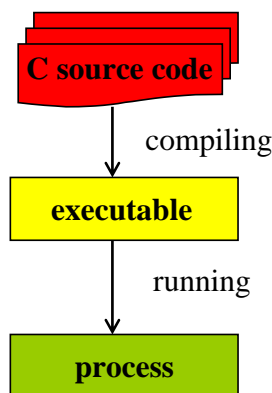
- Behind the scenes of running a program
  - Code, executable, and process
  - Main memory vs. virtual memory
- Memory layout for UNIX processes, and relationship to C
  - Text: code and constant data
  - Data: initialized global and static variables
  - BSS: uninitialized global and static variables
  - Heap: dynamic memory
  - Stack: local variables
- C functions for memory management
  - `malloc`: allocate memory from the heap
  - `free`: deallocate memory from the heap

2

## Code vs. Executable vs. Process



- C source code
  - C statements organized into functions
  - Stored as a collection of files (.c and .h)
- Executable module
  - Binary image generated by compiler
  - Stored as a file (e.g., *a.out*)
- Process
  - Instance of a program that is executing
    - With its own address space in memory
    - With its own id and execution state
  - Managed by the operating system



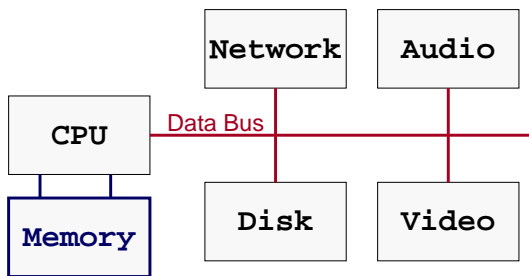
3

# Main Memory on a Computer



- What is main memory?

- Storage for variables, data, code, etc.
- May be shared among many processes



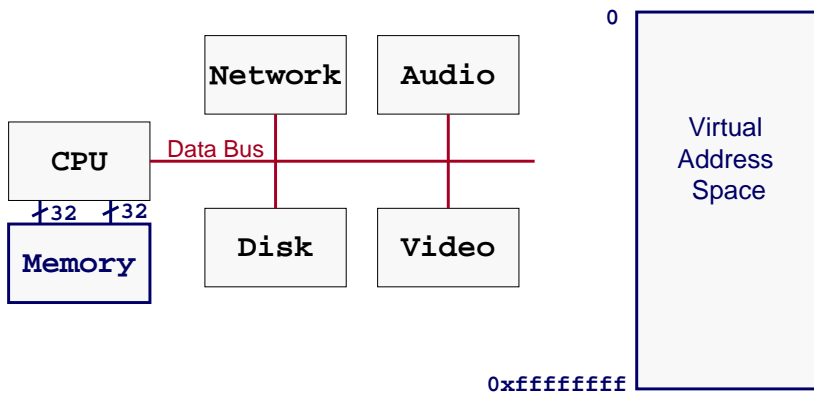
4

# Virtual Memory for a Process



- What is virtual memory?

- Contiguous addressable memory space for a single process
- May be swapped into physical memory from disk in pages
- Let's you pretend each process has its own contiguous memory



5

# What to Store: Code and Constants



- Executable code and constant data

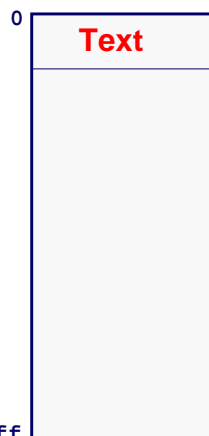
- Program binary, and any shared libraries it loads
- Necessary for OS to read the commands

- OS knows everything in advance

- Knows amount of space needed
- Knows the contents of the memory

- Known as the “text” segment

- Note: Some systems (e.g. hats) store some constants in “rodata” section



6

# What to Store: “Static” Data



- Variables that exist for the entire program
  - Global variables, and “static” local variables
  - Amount of space required is known in advance
- Data: initialized in the code
  - Initial value specified by the programmer
    - E.g., “`int x = 97;`”
  - Memory is initialized with this value
- BSS: not initialized in the code
  - Initial value not specified
    - E.g., “`int x;`”
  - All memory initialized to 0 (on most OS’s)
  - BSS stands for “Block Started by Symbol”

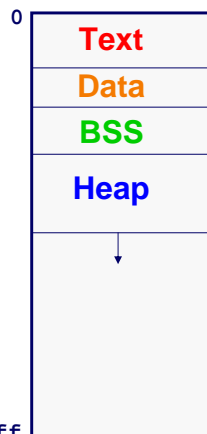


0xffffffff

# What to Store: Dynamic Memory



- Memory allocated while program is running
  - E.g., allocated using the `malloc()` function
    - And deallocated using the `free()` function
- OS knows nothing in advance
  - Doesn’t know the amount of space
  - Doesn’t know the contents
- So, need to allow room to grow
  - Known as the “heap”
  - Detailed example in a few slides
  - More in programming assignment #4

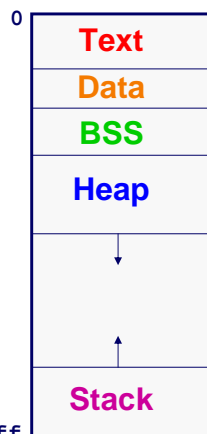


0xffffffff

# What to Store: Temporary Variables



- Temporary memory during lifetime of a function or block
  - Storage for function parameters and local variables
- Need to support nested function calls
  - One function calls another, and so on
  - Store the variables of calling function
  - Know where to return when done
- So, must allow room to grow
  - Known as the “stack”
  - Push on the stack as new function is called
  - Pop off the stack as the function ends
- Detailed example later on

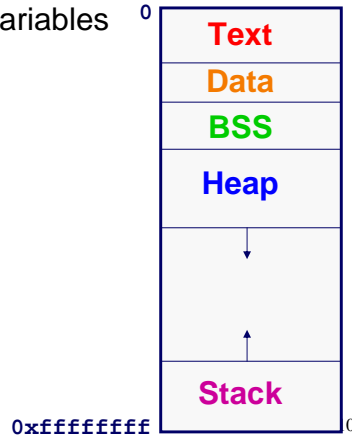


0xffffffff

# Memory Layout: Summary



- **Text**: code, constant data
- **Data**: initialized global & static variables
- **BSS**: uninitialized global & static variables
- **Heap**: dynamic memory
- **Stack**: local variables

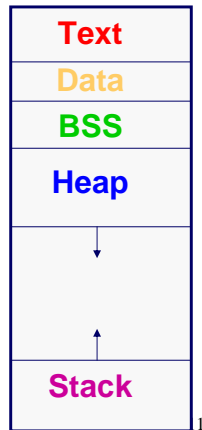


# Memory Layout Example



```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```

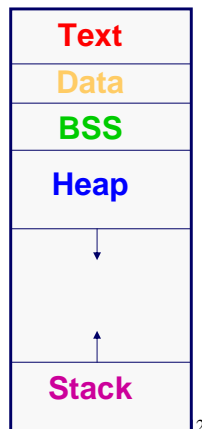


# Memory Layout Example: Text



```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```

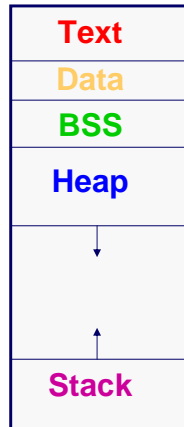


## Memory Layout Example: Data



```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



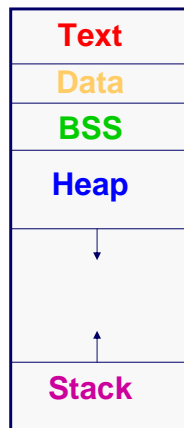
3

## Memory Layout Example: BSS



```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



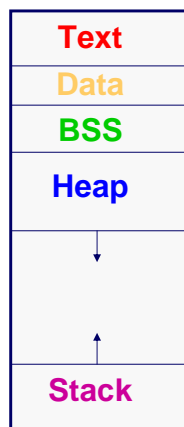
4

## Memory Layout Example: Heap



```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```



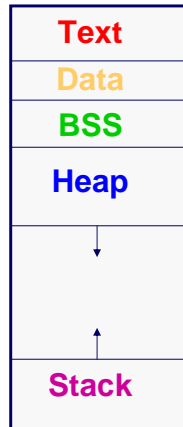
5

# Memory Layout Example: Stack



```
char* string = "hello";  
int iSize;
```

```
char* f(void)  
{  
    char* p;  
    iSize = 8;  
    p = malloc(iSize);  
    return p;  
}
```

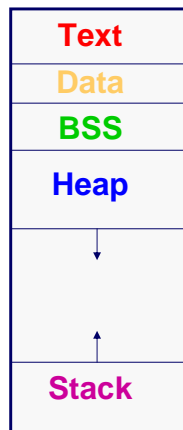


6

# Memory Allocation & Deallocation



- How, and when, is memory allocated?
  - Global and static variables = program startup
  - Local variables = function call
  - Dynamic memory = `malloc()`
- How is memory deallocated?
  - Global and static variables = program finish
  - Local variables = function return
  - Dynamic memory = `free()`
- All memory deallocated when program ends
  - It is good style to free allocated memory anyway



7

# Memory Allocation Example



```
char* string = "hello"; ← Data: "hello" at startup  
int iSize; ← BSS: 0 at startup
```

```
char* f(void)  
{  
    char* p; ← Stack: at function call  
    iSize = 8;  
    p = malloc(iSize); ← Heap: 8 bytes at malloc  
    return p;  
}
```

# Memory Deallocation Example



```
char* string = "hello"; ← Available till termination
int iSize; ← Available till termination
```

```
char* f(void)
{
    char* p; ← Deallocate on return from f
    iSize = 8;
    p = malloc(iSize); ← Deallocate on free()
    return p;
}
```

19

# Memory Initialization



- Local variables have undefined values

```
int count;
```

- Memory allocated by `malloc()` has undefined values

```
char* p = (char *) malloc(8);
```

- If you need a variable to start with a particular value, use an explicit initializer

```
int count = 0;
p[0] = '\0';
```

- Global and static variables are initialized to 0 by default

```
static int count = 0;
is the same as
static int count;
```

It is bad style to depend on this

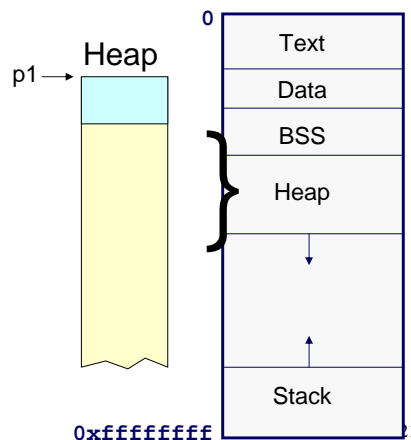
20

# Heap: Dynamic Memory



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

```
➔ char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



0xffffffff

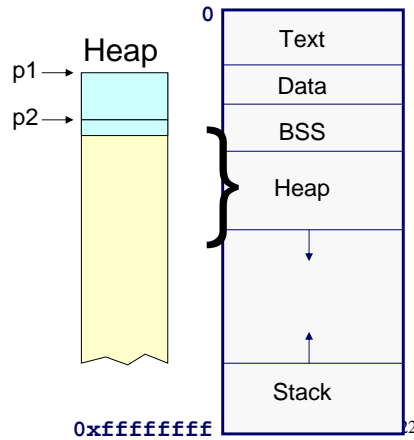
1

# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

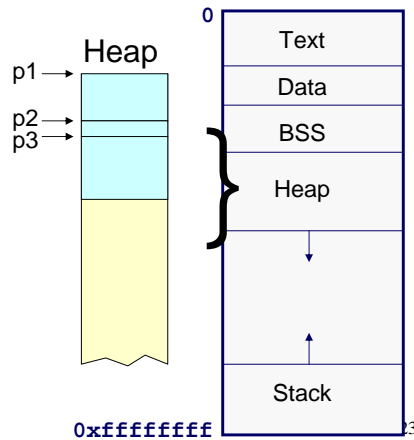


# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

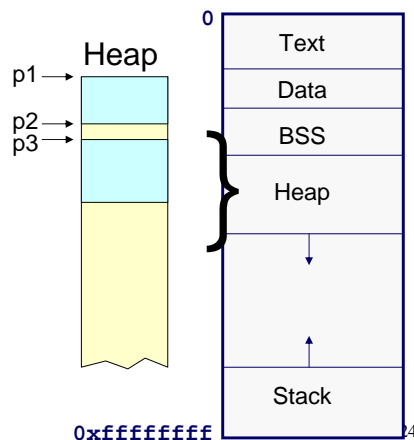


# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



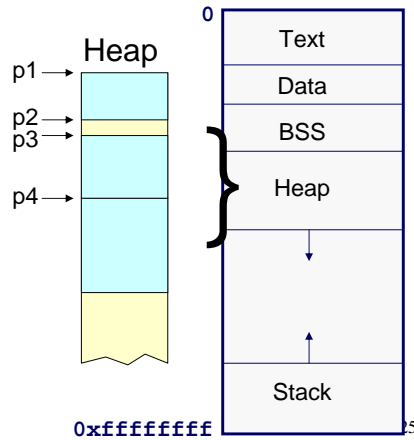


# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
➔ char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

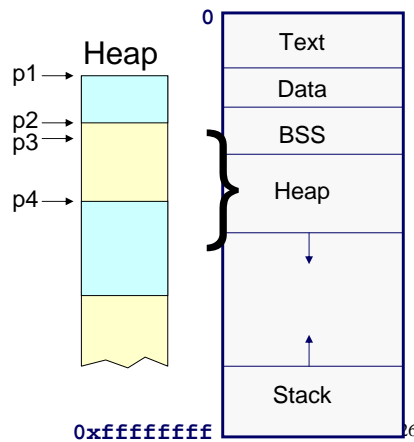


# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
➔ char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

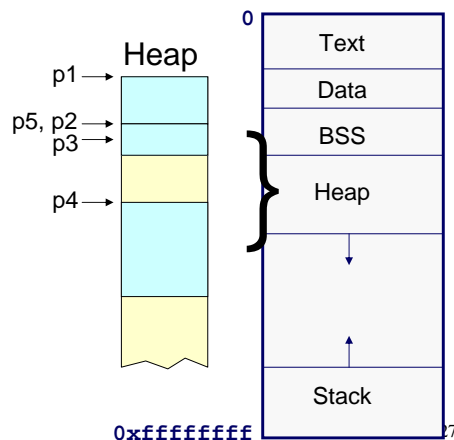


# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
➔ char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```

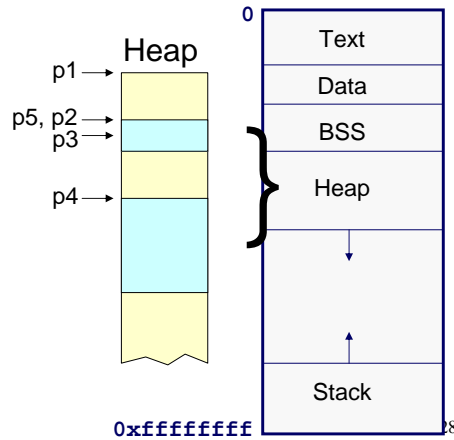


# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
➔ free(p1);
free(p4);
free(p5);
```

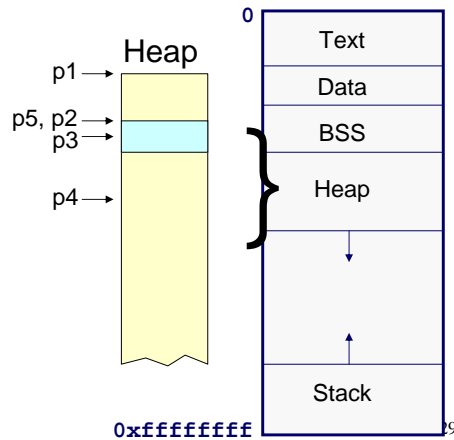


# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
➔ free(p4);
free(p5);
```

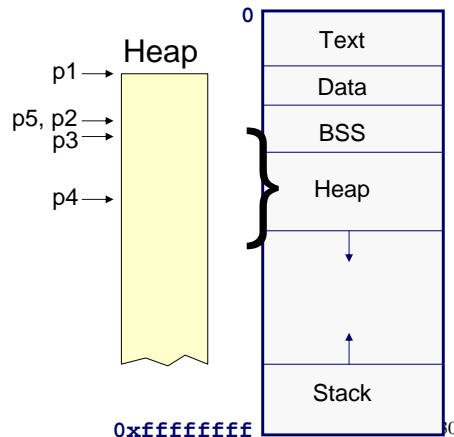


# Heap: Dynamic Memory



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

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
➔ free(p5);
```



# How Do Malloc and Free Work?



- Simple answer

- Doesn't matter
- Good modularity means you can use it without understanding it

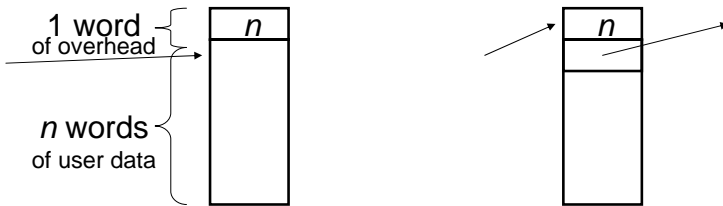
- Real answer

malloc(s)

$n = \lceil s / \text{sizeof}(\text{int}) \rceil$

free(p)

put p into linked list of free objects



31

# Using Malloc and Free



- Types

- `void*`: generic pointer to any type (can be converted to other types)
- `size_t`: unsigned integer type returned by `sizeof()`

- `void* malloc(size_t size)`

- Returns a pointer to space of size `size`
- ... or `NULL` if the request cannot be satisfied
- E.g., `int* x = (int *) malloc(sizeof(int));`

- `void* calloc(size_t nobj, size_t size)`

- Returns a pointer to space for array of `nobj` objects of size `size`
- ... or `NULL` if the request cannot be satisfied
- Bytes are initialized to 0

- `void free(void* p)`

- Deallocate the space pointed to by the pointer `p`
- Pointer `p` must be pointer to space previously allocated
- Do nothing if `p` is `NULL`

32

# Using realloc and (never) alloca



- `void* realloc(void* ptr, size_t size)`

- "Grows" the allocated buffer
- Moves/copies the data if old space insufficient
- ... or `NULL` if the request cannot be satisfied

- `void* alloca(size_t size)`

- **Not guaranteed to exist (not in any official standard)**
- Allocates space on local stack frame
- Space automatically freed when function exits
- Particularly useful for following:

```
int calc(int numItems) {  
    int items[numItems];  
    int *items = alloca(numItems * sizeof(int));  
}
```

33

## Sorting w/o Linked Lists



```
int alloc = 4, used = 0;
Item *temp, *buf = NULL;

while ((temp = NextItem()) != NULL) {
    if (used >= alloc) {
        alloc *= 2;
        buf = realloc(buf,
                      alloc*sizeof(Item));
    }
    buf[used++] = temp;
}

qsort(...);
```

34

## Avoid Leaking Memory



- Memory leaks “lose” references to dynamic memory

```
int f(void)
{
    char* p;
    p = (char *) malloc(8 * sizeof(char));
    ...
    return 0;
}

int main(void) {
    f();
    ...
}
```

35

## Avoid Dangling Pointers



- Dangling pointers point to data that's not there anymore

```
char *f(void)
{
    char p[8];

    ...
    return p;
}

int main(void) {
    char *res = f();
    ...
}
```

36

# Debugging Malloc Problems



- Symptom: “random” failures, especially on call return
  - Corrupted the stack frame return info
- Symptom: calls to malloc/free fail
  - Corrupted the malloc bookkeeping data
- Symptom: program magically works if printf inserted
  - Corrupted storage space in stack frame
- “Debugging” mallocs exist
  - Doing “man malloc” on Linux reveals MALLOC\_CHECK\_
  - Searching “debug malloc” yields dmalloc, other libraries
  - Larger problems: valgrind, electric fence, etc.

37

# Summary



- Five types of memory for variables
  - **Text**: code, constant data (constant data in rodata on hats)
  - **Data**: initialized global & static variables
  - **BSS**: uninitialized global & static variables
  - **Heap**: dynamic memory
  - **Stack**: local variables
- Important to understand differences between
  - Allocation: space allocated
  - Initialization: initial value, if any
  - Deallocation: space reclaimed
- Understanding memory allocation is important
  - Make efficient use of memory
  - Avoid “memory leaks” from dangling pointers

38