Version 2.17

UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 9

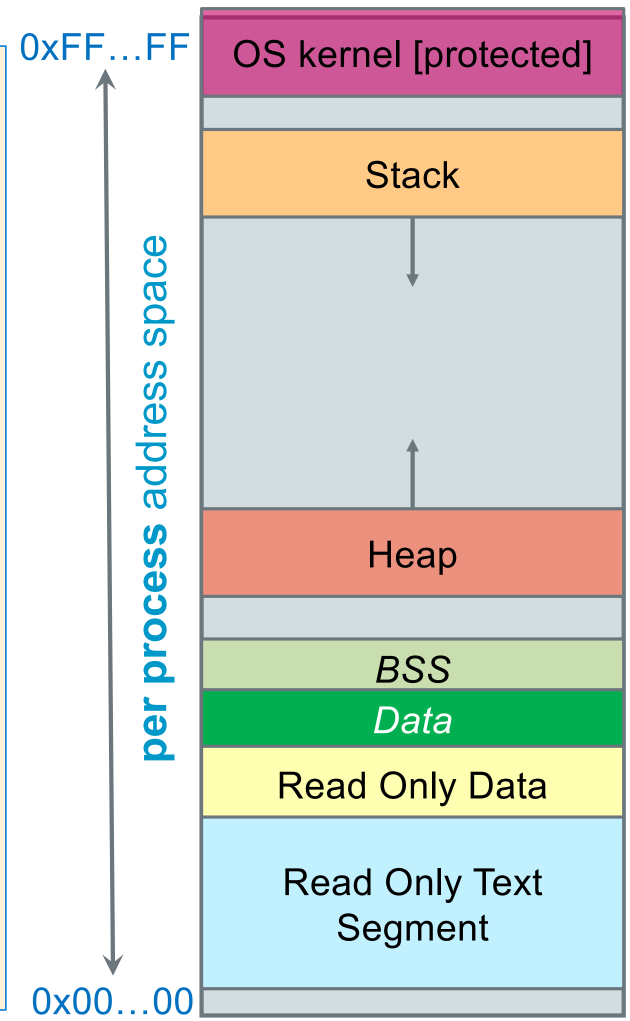
Keith Muller

Vax 11/780 1980

Thursday office hours moved to
Section B ZOOM Midterm Review
Thursday 7:30 PM – 8:30 PM
(My office hours zoom #)
(see canvas for number)

Process Memory Under Linux

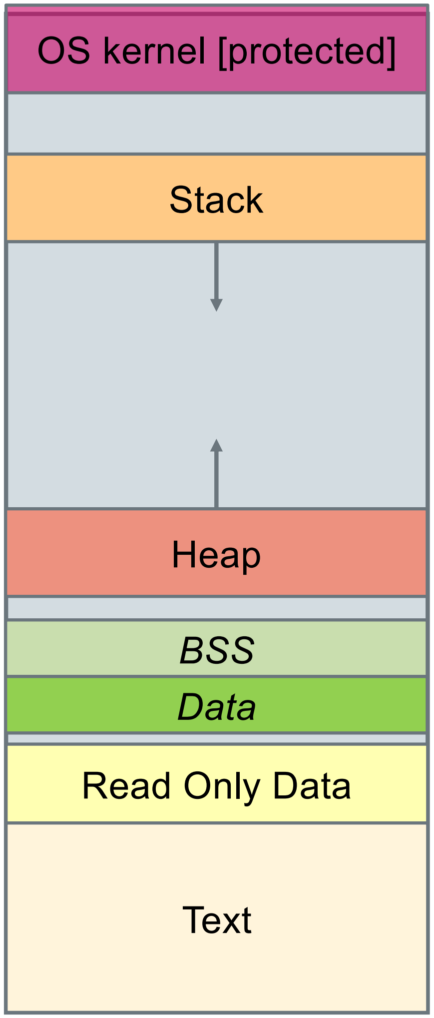
- When your program is running it has been loaded into memory and is called a process
- *Stack segment: Stores Local variables*
  - Allocated and freed at function call entry & exit
- *Data segment + BSS: Stores Global and static variables*
  - Allocated/freed when the process starts/exits
  - BSS - Static variables with an implicit initial value
  - Static Data - Initialized with an explicit initial value
- Heap segment: *Stores dynamically-allocated* variables
  - Allocated with a function call
  - Managed by the stdio library malloc() routines
- *Read Only Data: Stores immutable* Literals
- *Text*: Stores your code in machine language + libraries



x

The Heap Memory Segment

- **Heap**: "pool" of memory that is available to a program
  - Managed by C runtime library and linked to your code; **not managed by the OS**
- Heap memory is **dynamically** *"borrowed"* *or "allocated"* by calling a library function
- When heap memory is no longer needed, it is *"returned" or deallocated for* **reuse**
- Heap memory has a lifetime from allocation until it is deallocated
  - Lifetime is independent of the scope it is allocated in (it is like a static variable)
- If too much memory has already been allocated, the library will attempt to borrow additional memory from the OS and will fail, returning a NULL



x

Heap Dynamic Memory Allocation Library Functions

`#include <stdlib.h>`	args	Clears memory at runtime
`void *malloc(…)`	`size_t size`	no
`void *calloc(…)`	`size_t nmemb, size_t memsize`	yes
`void free(...)`	`void *ptr`	no

- **void *** means these library functions return a pointer to generic (untyped) memory
  - Be careful with void * pointers and **pointer math** as void * points **at untyped memory**
  - When assigned to a typed pointer, it *"converts"* it from a void * to the type of the pointer variable
- **size_t is** an unsigned integer data type, the result of a sizeof() operator

```
int *ptr = malloc(sizeof(*ptr) * 100); // allocate an array of 100 ints
```

- **please read: % man 3 malloc**

x

Use of Malloc

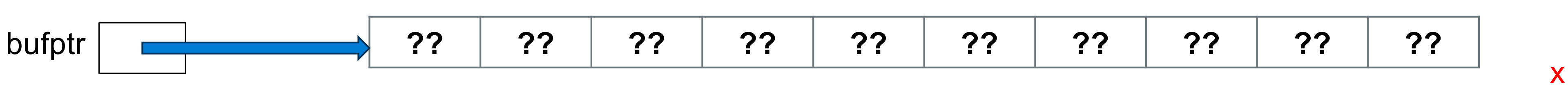
`void *malloc(size_t size)`

- Returns a pointer to a **contiguous** block of **`size`** bytes **of <u>uninitialized</u> memory** from the heap
  - The block is **aligned to an 8-byte (arm32) or 16-byte (64-bit arm/intel) boundary**
  - returns **`NULL`** if allocation failed (also sets **`errno`**) **always CHECK for NULL RETURN!**
- Blocks returned on **different calls to malloc()** are **not necessarily adjacent**
- **`void *`** is implicitly cast into **any pointer type on assignment to a pointer variable**

```
    char *bufptr;

    /* ALWAYS CHECK THE RETURN VALUE FROM MALLOC!!!! */

    if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {
        fprintf(stderr, "Unable to malloc memory");
        return NULL;
    }
// allocates a character array with 10 elements
```

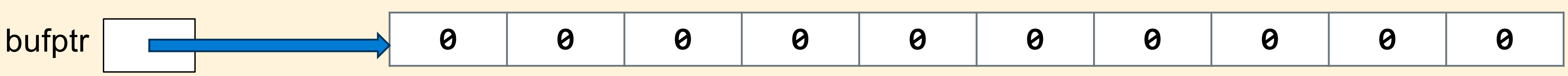


Calloc()

```
void *calloc(size_t elementCnt, size_t elementSize)
```

- `calloc()` variant of **`malloc()`** but zeros out every byte of memory during program execution before returning a pointer to it (so this has a runtime cost!)
  - First parameter is the number of elements you would like to allocate space for
  - Second parameter is the size of each element

```
// allocate 10-element array of pointers to char, zero filled
char **arr;
arr = calloc(10, sizeof(*arr));
if (arr == NULL) // handle the error
```



- Originally designed to allocate arrays but works for any memory allocation
  - **`calloc()`** multiplies the two parameters together for the total size
- **`calloc()`** is more expensive at runtime (uses both cpu and memory bandwidth) than **`malloc()`** because it must zero out memory it allocates at runtime
- Use `calloc()` only when you need the buffer to be zero filled prior to FIRST use

Using and Freeing Heap Memory

- `void free(void *p)`
  - Deallocates the whole block pointed to by `p` to the pool of available memory
  - **Freed memory** is **used in future allocations** (expect the contents to change after freed)
  - Pointer `p` must be the same address as *originally returned* by one of the heap allocation routines **`malloc(), calloc(), realloc()`**
  - **Pointer argument to free() is not changed by the call to free()**
- Defensive programming: set the pointer to NULL after passing it to free()

```
char *bufptr;

  if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {
      fprintf(stderr, "Unable to malloc memory");
      return NULL;
  }
  // other code
  free(bufptr);                      // returns memory to the heap
  bufptr = NULL;                    // set bufptr to NULL
```


x

Mis-Use of Free() - 1

- Call **`free()`**
  - With the same address that you obtained with malloc() (or other allocators)
  - It is NOT an error to pass **`free()`** a pointer to NULL

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes)) != NULL) {
    /* some code */
    free(bytes + 5); // Program aborts free(): invalid pointer
```

- **Freeing unallocated memory:** Only call free() to free memory address that you obtain from one of the allocators (malloc(), calloc(), etc.)

```
char *ptr = "cse30";
          …
   /* some code */
   free(ptr);              // Program aborts free(): invalid pointer
```


x

Mis-Use of Free() - 2

- Continuing to write to memory after you **`free() it`** is likely to corrupt the heap or return changed values
  - Later calls to heap routines (**`malloc(), calloc(), strdup())`** may fail or seg fault

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes)) != NULL) {
    /* some code */
    free(bytes);
    strcpy(bytes, "cse30");   // INVALID! used after free
……
```

- **Double Free:** Freeing allocated memory more than once will cause your program to abort (terminate)

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes)) != NULL) {
    /* some code */
    free(bytes);
    free(bytes); // Program abort double free detected….
……
```


x

More Dangling Pointers: Continuing to use "freed" memory

- Review: **Dangling pointer** points to a memory location that is no longer "valid"
- Really hard to debug as the use of the return pointers may not generate a seg fault

```
char *dangling_freed_heap(void)
{
    char *buff = malloc(BLKSZ * sizeof(*buff));
…
    free(buff);    // memory pointed at buf may be reused
    return buff;   // but it is returned to the caller anyway - bad
}
```

- `dangling_freed_heap()` **may cause** the allocators (**`malloc()`**and friends) to **seg fault when called later to allocate memory**
  - Why? Because it corrupts data structures the heap code uses to manage the memory pool (it often stores meta-data in the freed memory)

x

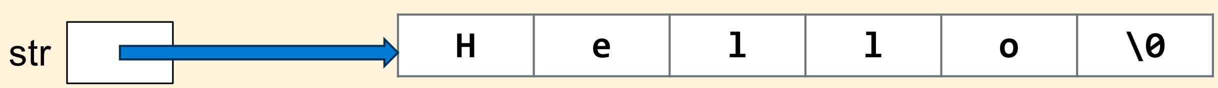
strdup(): Allocate Space and Copy a String

```c
char *strdup(char *s);
```

- **`strdup`** is a function that has a **side effect** of returning a **null-terminated**, heap-allocated string copy of the provided text
- Alternative: **`malloc`** and copy the string with strncpy();
- The caller is responsible for freeing this memory when no longer needed

```c
char *str = strdup("Hello");
*str = 'h';  // str points at a mutable string

free(str);  // caller correctly frees up space allocated by strdup()
str = NULL;
```



x

Heap Memory "Leaks"

- A memory leak is when you **allocate memory** on the heap, **but never free it**

```
void
leaky_memory (void)
{
    char *buf = malloc(BLKSZ * sizeof(*bytes));
…
    /* code that never calls free() to deallocates the memory */
    return;  // you lose the address in buf when leaving scope
}
```

- Best practice: free up memory you allocated when you no longer need it
  - If you keep allocating memory, you may run out of memory in the heap!
- Memory leaks may cause long running programs to fault when they exhaust OS memory limits
- Valgrind is a tool for finding memory leaks (not pre-installed in all linux distributions though!)

x

Valgrind – Finding Buffer Overflows and Memory leaks

```
1 #define SZ 50
2 #include <stdlib.h>
3 int main(void)
4 {
5     char *buf;
6     if ((buf = malloc(SZ * sizeof(*buf))) == NULL)
7         return EXIT_FAILURE;
8     *(buf + SZ) = 'A';
9     // free(buf);
10     return EXIT_SUCCESS;
11 }
```

```
% valgrind -q --leak-check=full --leak-resolution=med -s ./valgexample
==651== Invalid write of size 1
==651==    at 0x10444: main (valg.c:8)
==651==  Address 0x49d305a is 0 bytes after a block of size 50 alloc'd
==651==    at 0x484A760: malloc (vg_replace_malloc.c:381)
==651==    by 0x1041B: main (valg.c:6)
==651==
==651== 50 bytes in 1 blocks are definitely lost in loss record 1 of 1
==651==    at 0x484A760: malloc (vg_replace_malloc.c:381)
==651==    by 0x1041B: main (valg.c:6)
==651==
==651== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Writing outside of allocated buffer space

Memory not freed

x

Introduction to Structs – An Aggregate Data Type

- **Structs** are a **collection (or aggregation) of values** grouped under a **single name**
  - Each variable in a struct is called a **member** (sometimes **field** is used)
  - Each member is identified with a name
  - Each member can be (and quite often are) different types, include other structs
  - Like a Java class, but no associated methods or constructors with a struct
- Structure definition **does not** define a variable instance, nor does it allocate memory:
  - It creates a new variable type uniquely identified by its tagname:

    "struct tagname" includes the keyword struct and the tagname for this type

Easy to forget semicolon!

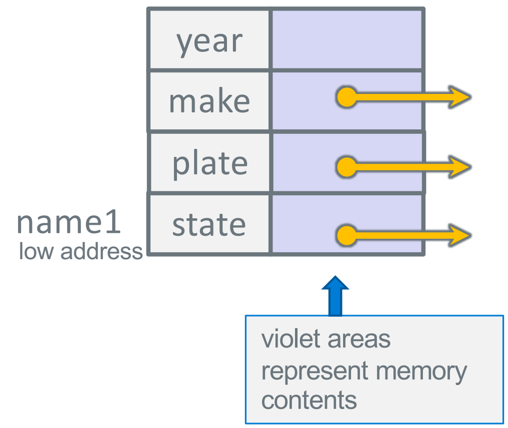
```
struct tagname {
  type1 member1;
  ...
  typeN memberN;
};
```

```
struct vehicle {
  char *state;
  char *plate;
  char *make;
  int year;
};
```


x

Struct Variable Definitions

```
struct vehicle {
  char *state;
  char *plate;
  char *make;
  int  year;
};
struct vehicle name1;
```



- Variable definitions like any other data type:

```
struct vehicle name1, *pn, ar[3];
```

type: "struct vehicle"

single variable instance

pointer

array

x

Accessing members of a struct

- Like arrays, struct variables are aggregated contiguous objects in memory
- The `.` structure operator *"selects"* the specified field or member

```c
struct date {       // defining struct type
    int month;      // member month
    int day;        // member date
};
```

day	
month	

struct date type definition

```c
struct date bday; // define a struct instance
bday.month = 1;
bday.day = 24;

// alternative initializer syntax
struct date new_years_eve = {12, 31};
struct date final = {.day= 24, .month= 1};
```

day	24
month	1

bday definition

Accessing members of a struct with pointers

```c
struct date {       // defining struct type
    int month;      // member month
    int day;        // member date
};
```

day	
month	

- Define a *pointer* to a struct

```c
struct date *ptr = &bday;
```

- Two ways to reference a member via a struct pointer (. is higher precedence than *):

1. Use * and . operators: `(*ptr).month = 11;`
2. Use -> operator for shorthand: `ptr->month = 11;`

Operator	Description	Associativity
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left

X

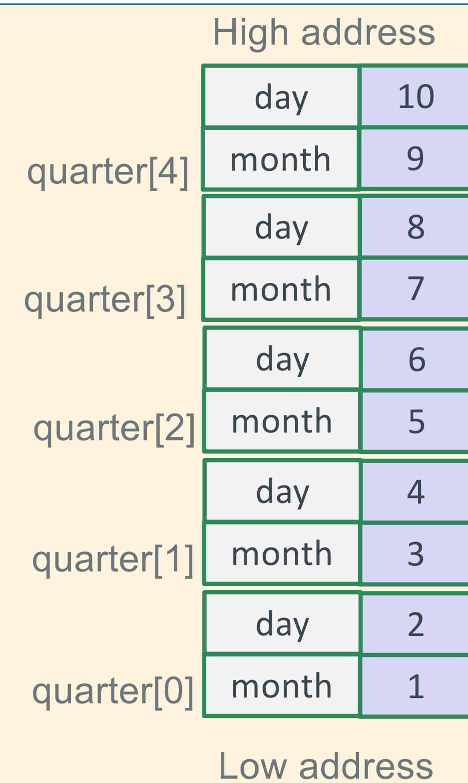
Accessing members of a struct

```c
struct date {       // defining struct type
    int month;      // member month
    int day;        // member date
};
```

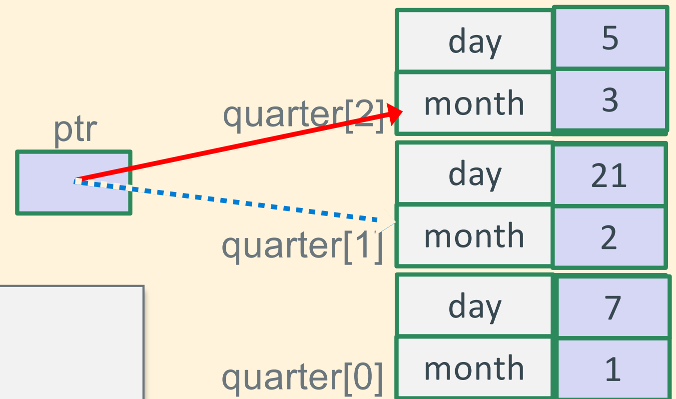
- You can create an array of structs and initialize them

```c
struct date quarter[] =
             { {1,2}, {3,4}, {5,6}, {7,8}, {9,10} };

int cnt = sizeof(quarter)/sizeof(*quarter);  // = 5
```



Accessing members of a struct




```
struct date quarter[3];
struct date *ptr;

ptr = quarter + 1;      // array name = address
ptr->month = 2;
ptr->day = 21;          // or (*ptr).day = 21;

(ptr-1)->month = 1;     // or (*(ptr-1)).month = 4;
(ptr-1)->day = 7;

(++ptr)->month = 3;
ptr->day = 5;
```


x