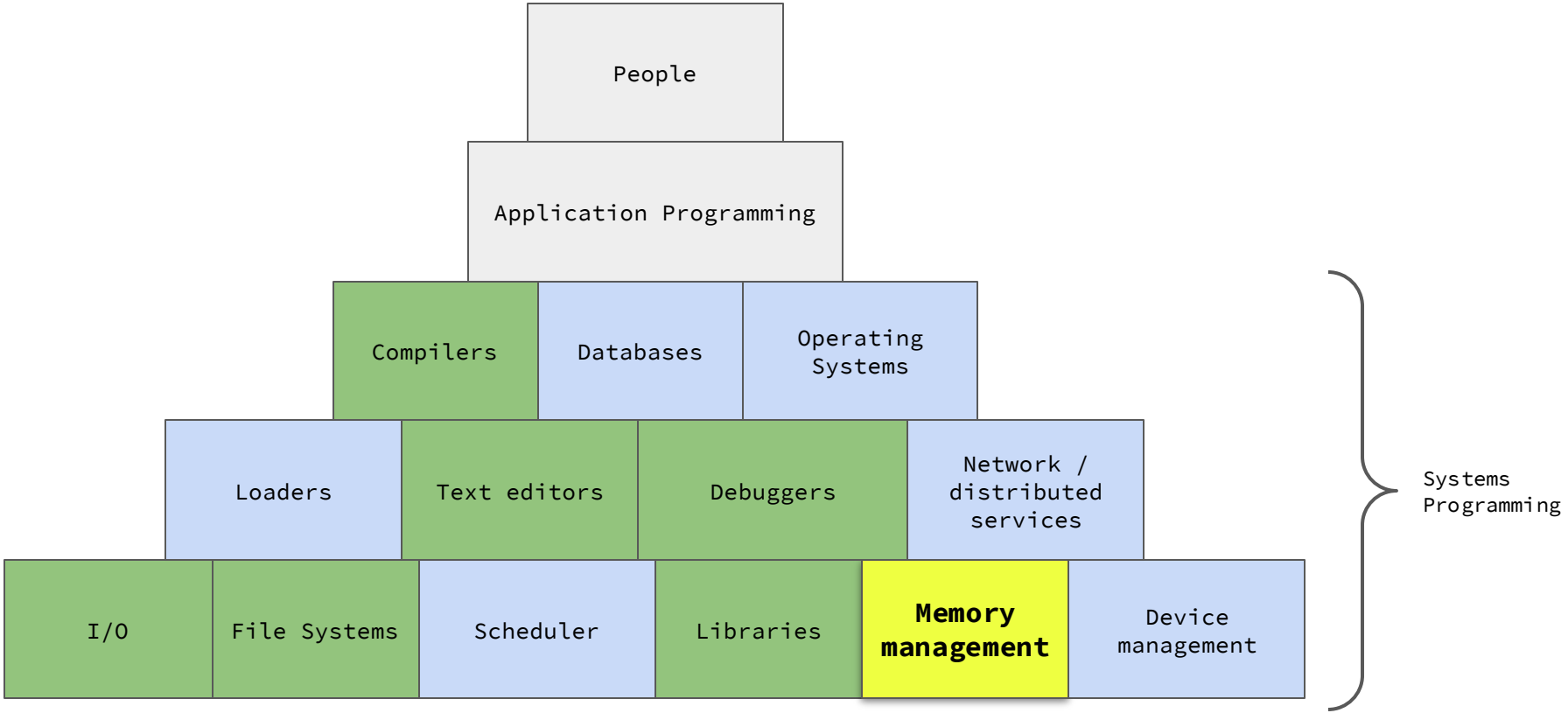
CSC / CPE 357

Systems Programming

Topics

- Memory Allocation

These contents are based on materials from Professors Aaron Keen and Andrew Migler


People
Application Programming
Compilers
Databases
Operating Systems
Loaders
Text editors
Debuggers
Network / distributed services
I/O
File Systems
Scheduler
Libraries
Memory management
Device management
Systems Programming

Virtual Memory Layout

- Virtual memory is often discussed in terms of **low memory**, closer to address `0x00000000`, or the "beginning" of memory, versus **high memory**.

- High memory is typically reserved for the kernel

- By default, the Linux kernel uses a 3:1 proportion. For example, 32-bit address space (4GB):
  - User space: `0x00000000` - `0xbfffffff` (3GB)
  - Kernel space: `0xc0000000` - `0xffffffff` (1GB)

Memory Layout of a C Program

- Text segment
  - Machine instructions
- Initialized data segment
  - Variables that are initialized in the program (e.g. `int months = 12;`)
- Uninitialized data segment (bss)
  - For example: `long sum[1000];` outside of any function
- Stack
  - Automatic variables, return address
- Heap
  - Dynamically allocated memory ie. `malloc()`
- The `size(1)` command reports the sizes (in bytes) of the text, data, and bss segments of an executable.

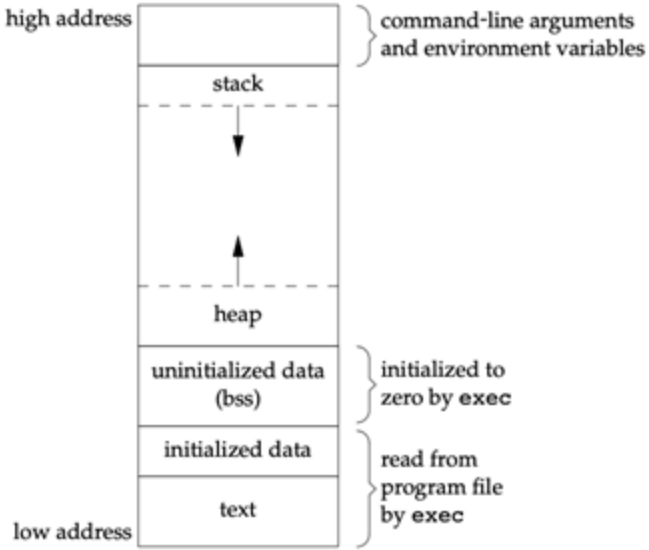


Figure 7.6 Typical memory arrangement

Stack Growth

- Assuming Linux on a 32-bit Intel x86 processor:
- Bottom of the stack: (just below) `0xC0000000`
  - Stack grows from *higher*-numbered addresses to lower-numbered addresses.
- Text segment starts at: `0x08048000`

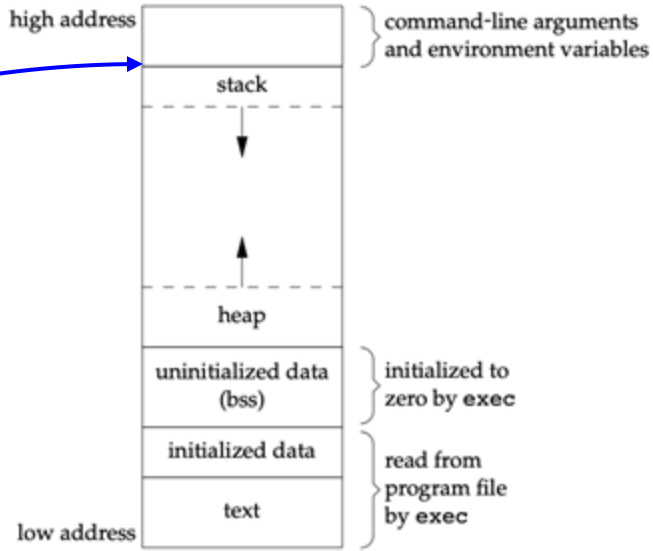


Figure 7.6 Typical memory arrangement

Function Call Stack

- The **function call stack** (often referred to as the "call stack" or just the "stack") is responsible for maintaining the local variables and parameters during function execution.

- Plays a central role in the execution of C programs

- The stack is a dynamic entity
  - Contents and size change, growing as functions are called and shrinking as functions complete and return to the calling function.
  - However, size cannot grow without bounds (hence: stack overflow)

The Stack

The stack holds return address, incoming parameters, and function-local variables.

```
int a_function(int a, int b, int c) {
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    ...
```

The Heap

Dynamic memory allocation (ie. `malloc()`) uses the **heap**

This type of allocation allows:

- Memory that persists across function calls but not for the entire runtime of the program
- Memory whose size is not known in advance

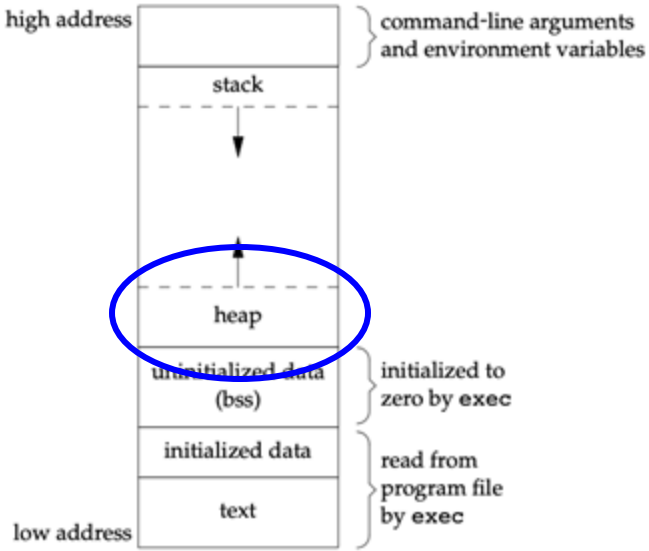


Figure 7.6 Typical memory arrangement

malloc()

- **`void *malloc(size_t size);`**
- Allocates a block of heap memory of at least the requested size
  - Returns a pointer to the first byte of allocated memory
  - Returns `NULL` if the memory allocation fails
  - Memory is uninitialized (should be treated as "mystery data")
- Example:
```
int *i = (int *) malloc(sizeof(int))
```
- With error checking:
```
// allocate space for a 10-char string (+ terminator)
char *s = (char*) malloc(11 * sizeof(char));
if (s == NULL) {
   // report error
}
```

malloc() Library Function / sbrk() System Call

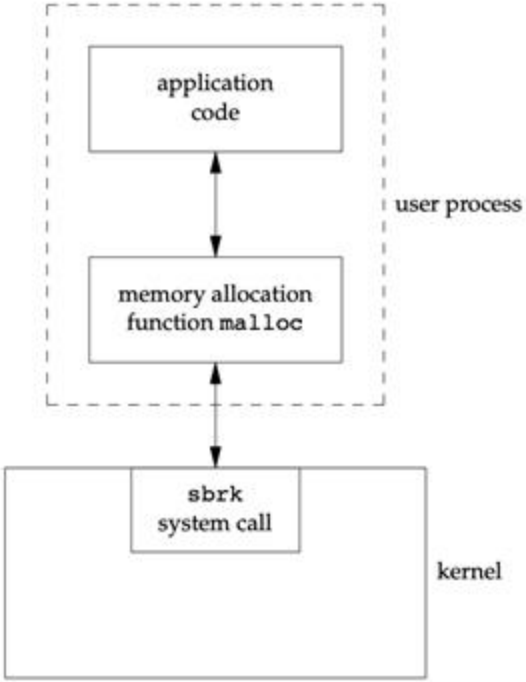


Figure 1.11 Separation of `malloc` function and `sbrk` system call

Separation of duties:

- System call in the kernel (`sbrk()`) allocates space on behalf of the process.
- The `malloc()` library function manages this space from user level.

free()

```
void free(void *ptr)
```

- Deallocates heap memory
- Pointer must point to the first byte of heap-allocated memory
  - Something previously returned by `malloc` (or related `_alloc` function)
- A memory leak occurs if malloc'ed memory is not freed

Example: arraycopy

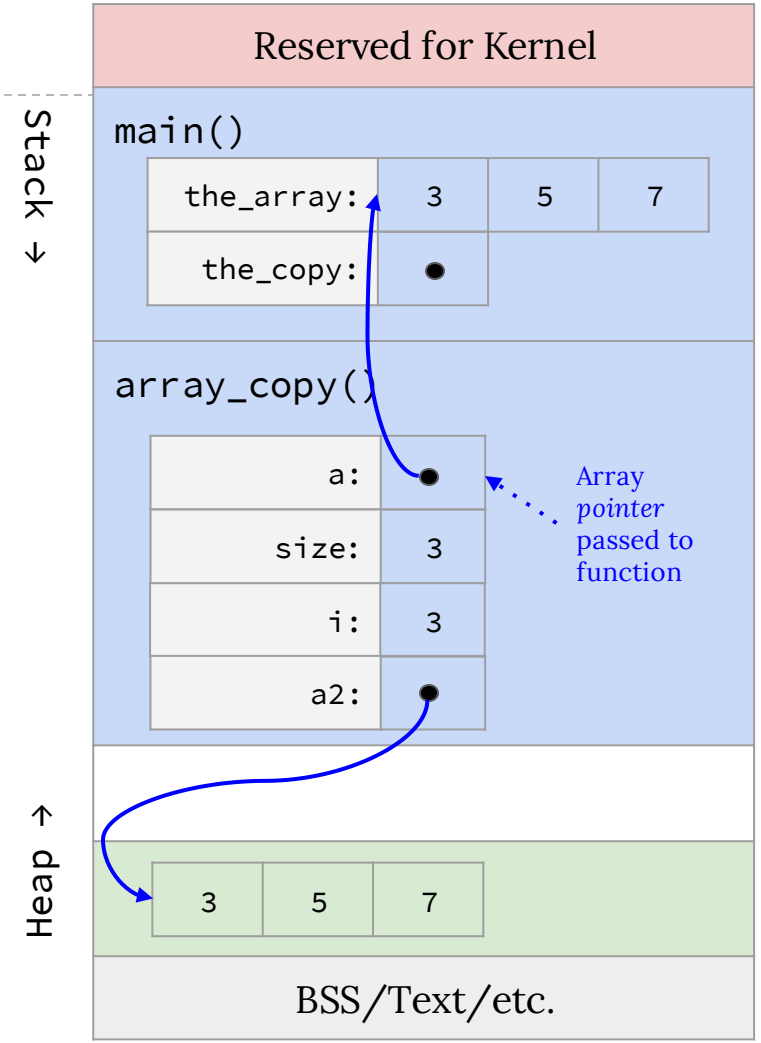
Function that copies an array of integers into a newly-allocated heap storage space.

`arraycopy.c`

arraycopy.c Stack Diagram

Stack state as of line 16, after heap allocation and copy of array.

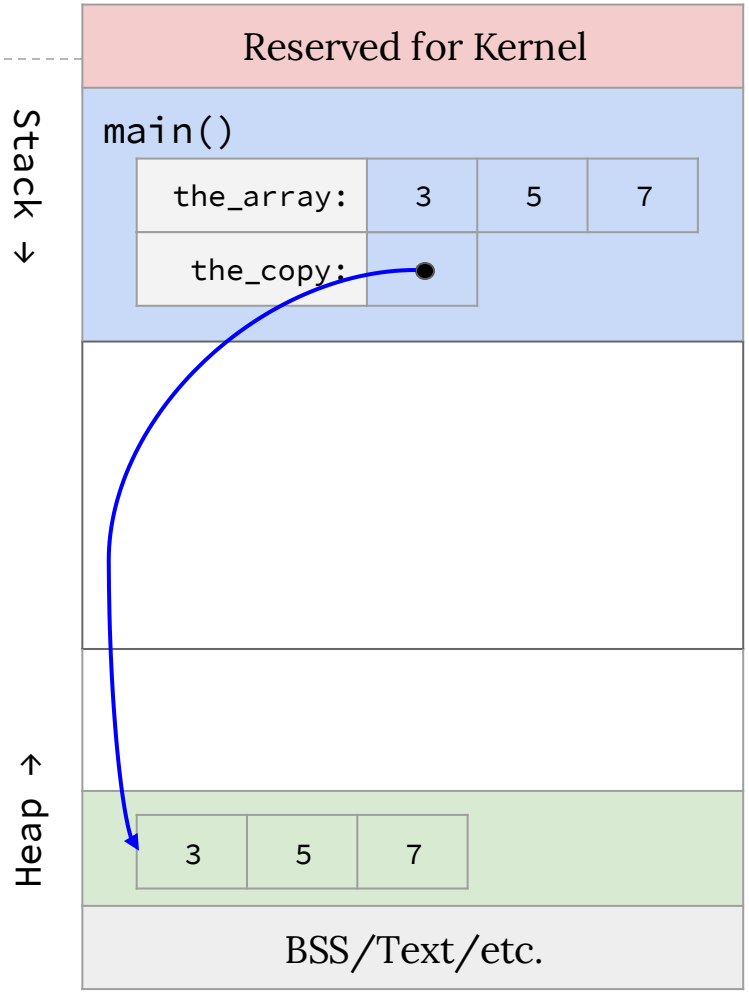
```
 3 int *array_copy(int a[], int size) {
 4     int i, *a2;
 5
 6     a2 = malloc(size * sizeof(int));
 7
 8     if (a2 == NULL) {
 9         return NULL;
10     }
11
12     for (i = 0; i < size; i++) {
13         a2[i] = a[i];
14     }
15
16     return a2;
17 }
18
19 int main(int argc, char *argv[]) {
20     int the_array[3] = { 3, 5, 7 };
21
22     int *the_copy = array_copy(the_array, 3);
23
24     // use the copied array
25
26     free(the_copy);
27
28     return EXIT_SUCCESS;
29 }
```



arraycopy.c Stack Diagram

Stack state as of line 24, after return from `array_copy()` function

```
 3 int *array_copy(int a[], int size) {
 4     int i, *a2;
 5
 6     a2 = malloc(size * sizeof(int));
 7
 8     if (a2 == NULL) {
 9         return NULL;
10     }
11
12     for (i = 0; i < size; i++) {
13         a2[i] = a[i];
14     }
15
16     return a2;
17 }
18
19 int main(int argc, char *argv[]) {
20     int the_array[3] = { 3, 5, 7 };
21
22     int *the_copy = array_copy(the_array, 3);
23
24     // use the copied array
25
26     free(the_copy);
27
28     return EXIT_SUCCESS;
29 }
```



realloc()

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

- The `realloc()` function changes the size of a memory block.
- The contents in the previously-allocated space will be unchanged
  - If the new size is larger than the old size, the added memory will not be initialized

```
int* arr = (int*)malloc(5 * sizeof(int));
arr = (int*)realloc(arr, 10 * sizeof(int));
if (arr == NULL) {
    // Handle reallocation failure.
}
```

realloc() Additional Notes

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

- If `ptr` is `NULL`, then the call is equivalent to `malloc(size)`
- If `size` is equal to zero, and `ptr` is not `NULL`, then the call is equivalent to `free(ptr)`
- If `realloc()` fails, the original block is left untouched; it is not freed or moved

malloc() vs realloc()

Aspect	malloc()	realloc()
Functionality	Allocates a new block of memory.	Resizes an existing block of memory.
Input	Size of memory to allocate.	Pointer to previously allocated memory and new size.
Return Value	Pointer to allocated memory or NULL if failed.	Pointer to resized memory block or NULL if failed.
Memory Content	Memory is uninitialized.	Original data is preserved up to the minimum size.
Behavior on NULL Pointer	N/A	Acts like `malloc()` if `ptr` is NULL.

calloc()

```
void *calloc(size_t nmemb, size_t size);
```

- Allocates memory for an array of `nmemb` elements of `size` bytes each
  - Returns a pointer to the start of the allocated memory
- The memory is set to zero

Malloc and Structs

Heap-based memory allocation is often paired with C structs to implement dynamic data structures (linked lists, trees, etc.)

`linked_list.c`

- Try to understand the code.
- Take note of how each part is relevant to previously discussed topics/labs.
- Compare it to how you implemented a linked list using another programming language.

strdup()

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

Convenience function that returns a null-terminated, heap-allocated string with the provided text,. No need to malloc and copy in the string yourself.

Heap Allocation Summary

```
void *malloc(size_t size);

void *calloc(size_t nmemb, size_t size);

void *realloc(void *ptr, size_t size);

char *strdup(char *s);

void free(void *ptr);
```

Undefined Behavior

- Overflow (i.e., you access beyond bytes allocated)
- Use after free, or if free is called twice on a location
- realloc/free non-heap address

`undefined.c`

Stack Vs Heap

The Stack

- Fast
  - Quick allocate/deallocate
- Convenient
  - Automatic allocation/ deallocation; declare/initialize in one step
- Reasonable type safety
  - Compiler / arrays / etc.
- Inflexible
  - Cannot add/resize at runtime
  - Scope dictated by control flow

The Heap

- Flexible.
  - Runtime decisions about how much/when to allocate
  - Resize easily with realloc
- Control over scope / lifetime
- Opportunity for error
  - Low type safety,
  - Need to explicitly allocate/free
  - Memory leaks

Malloc Implementations

- Allocation is usually implemented with the sbrk system call
- `sbrk` expands (or contracts) the heap of the process
- Most implementations of `malloc` and `free` never decrease process memory size
  - Space is kept in a "malloc pool"
- Common to allocate more space than requested
  - Additional space for record keeping:
    - size of the block
    - pointer to the next allocated block
    - etc.

Memory Allocator Implementations

- Historically, the standard malloc algorithm used either a best-fit or a first-fit memory allocation strategy.
- Most modern allocators are based on a "quick-fit" strategy
- Quick-fit is faster than best-fit or first-fit
  - Uses more memory
- Quick-fit splits memory into buffers of various sizes
  - Maintains "free lists" of unused buffers based on buffer sizes

Other Memory Allocation Implementations

- `jemalloc` (FreeBSD) - designed to scale well when used with multithreaded applications running on multiprocessor systems

- `TCMalloc` (Google) - high performance, scalability, and memory efficiency, built-in a heap checker and heap profiler

- `alloca` Function - memory is allocated from the stack frame of the current function, space freed automatically on return from function