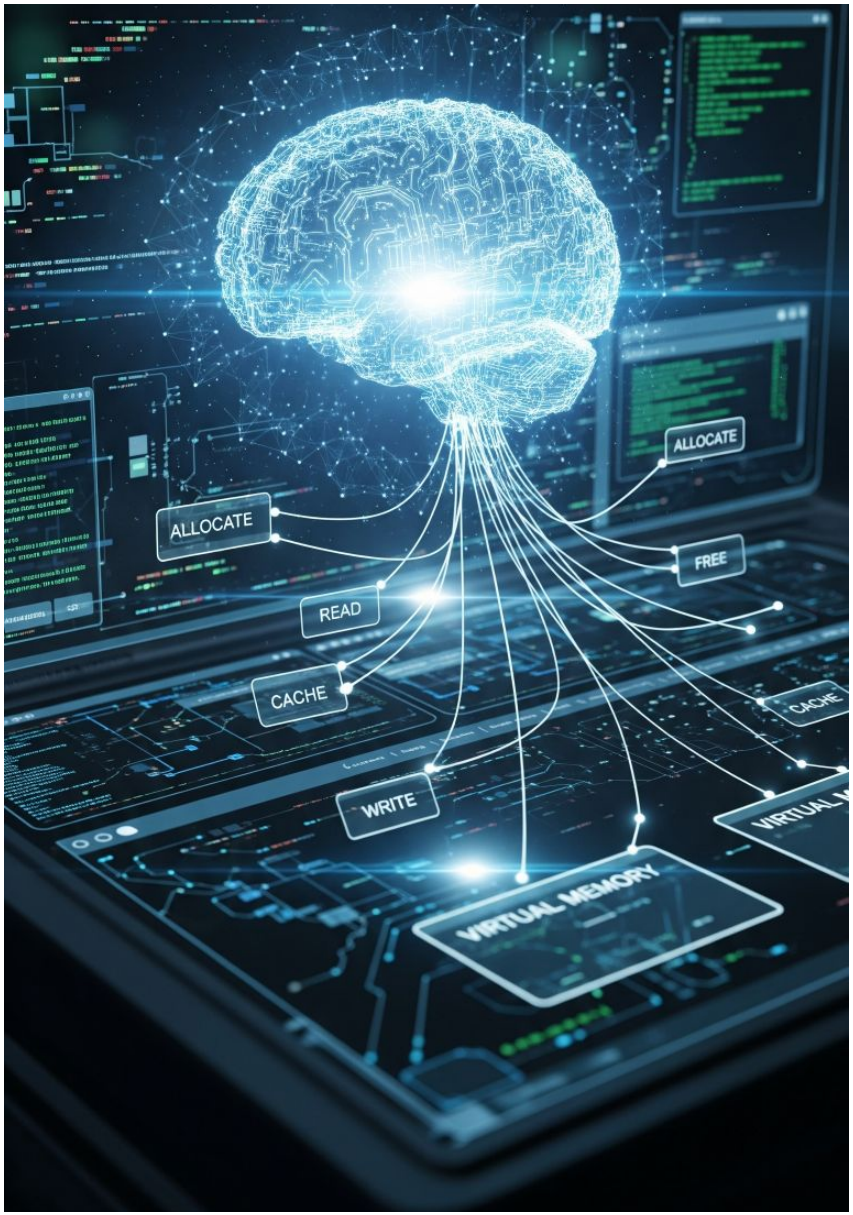


CSL 301
OPERATING SYSTEMS

Lecture 8
The Memory API

Instructor
Dr. Dhiman Saha

Image Source: Gemini Imagen

The Crux of the Problem

How to Allocate and Manage Memory?

In UNIX/C programs, understanding how to allocate and manage memory is critical for building robust and reliable software.

- What interfaces are commonly used?
- What mistakes should be avoided?

Types of Memory: Stack vs. Heap

Stack Memory

- Managed implicitly by the compiler.
- Sometimes called "automatic" memory.
- Memory is deallocated when the function returns.

```
void func() {
    int x; // Declared on the
        stack
}
```

Listing 1: Stack Allocation

Heap Memory

- Managed explicitly by the programmer.
- For "long-lived" memory.
- This is the source of many bugs!

```
void func() {
    int *x = (int *) malloc(
        sizeof(int));
}
```

Listing 2: Heap Allocation

The ‘malloc()‘ Call

‘malloc()‘: The ”Memory Allocator”

The `malloc()` call allocates a block of memory from the heap.

- It takes a single argument: the number of bytes to allocate.
- It returns a pointer to the new block of memory.
- It returns `NULL` if the allocation fails.

```c
#include <stdlib.h> // Required header

// Allocate space for a double
double *d = (double *) malloc(sizeof(double));

// Allocate space for 10 integers
int *arr = (int *) malloc(sizeof(int) * 10);
```

Listing 3: Using malloc()

- `sizeof()` is a compile-time operator that gets the size of a type or variable.
- The cast (e.g., `(double *)`) is not strictly required but is good practice.

The `free()` Call

`free()`: Returning Memory to the Heap

Memory you allocate on the heap must be returned, otherwise you get a **memory leak**.

- `free()` takes a single argument: a pointer that was returned by `malloc()`.
- You do not need to specify the size. The memory library keeps track of it.

```
// Allocate space for 10 integers
int *x = (int *) malloc(10 * sizeof(int));

// ... do some work with x ...

// Now, free the memory
free(x);
```

Listing 4: Using `free()`

Common Memory Errors

Just because it compiles, doesn't mean it's correct!

Many memory errors will not be caught by the compiler. Your program might even seem to run correctly sometimes, and then crash unexpectedly.

We will now look at the most common memory-related bugs in C.

Error 1: Forgetting To Allocate Memory

The Problem

Using a pointer to store data, but forgetting to allocate memory for it to point to.

- This often leads to a **segmentation fault**.

Wrong

```
char *src = "hello";
char *dst; // Uninitialized!
strcpy(dst, src);
// CRASH!
```

Right

```
char *src = "hello";
char *dst = (char *)
malloc(strlen(src) + 1);
strcpy(dst, src); // OK!
```

- Note the extra '1' in the `malloc` call.
- Why is this needed?

Error 2: Not Allocating Enough Memory (Buffer Overflow)

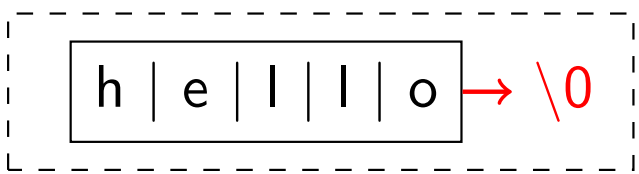
The Problem

Allocating memory, but not enough to hold what you plan to copy into it. This is a **buffer overflow** and a major source of security vulnerabilities.

```c
// "hello" is 5 chars, but strlen("hello") is 5.
// We need 6 bytes to store the string, including
// the null terminator '\0'.

char *src = "hello";
// Oops! Forgot the +1 for the null terminator
char *dst = (char *) malloc(strlen(src));
strcpy(dst, src); // Writes past the end of the buffer!
```

Listing 5: A common off-by-one error



Allocated Buffer (5 bytes)

Error 3: Forgetting to Initialize Memory

The Problem

`malloc()` allocates memory, but does not clean it. The contents are garbage. Reading from this memory before initializing it is an **uninitialized read**.

```c
// Allocate memory for 10 integers
int *data = (int *) malloc(sizeof(int) * 10);

// Whoops, forgot to put anything in data!
int sum = 0;
for (int i = 0; i < 10; i++) {
    // Reading garbage data!
    sum += data[i];
}
```

Listing 6: Uninitialized Read

- The program's behavior is now unpredictable.
- To fix, either manually initialize or use `calloc()`.

Error 4: Forgetting To Free Memory

The Problem

Forgetting to call `free()` on memory that is no longer needed. This is a **memory leak**.

```
void leaky_function() {
    // Allocate some memory
    int *p = malloc(1000 * sizeof(int));

    // ... use p ...

    // Oh no, we returned without freeing p!
    // The memory is now leaked.
}
```

Listing 7: A simple memory leak

- For short-lived programs, the OS reclaims memory on exit.
- For long-running programs (like servers or the OS itself), leaks are a huge problem, eventually causing the program to run out of memory and crash.

Error 5: Freeing Memory Before You Are Done

The Problem

Freeing a block of memory and then trying to use it later. This is called using a **dangling pointer**.

```c
int *p = (int *) malloc(sizeof(int));
*p = 123;

free(p); // Memory is now free

// ... some time later ...

printf("Value is %d\n", *p); // Using a dangling pointer!
// This can crash, or print garbage, or seem to work...
```

Listing 8: Dangling Pointer

- The memory that p points to could have been re-allocated to something else! Writing to it can corrupt other data.

More Errors: Double and Invalid Frees

Double Free

- Calling `free()` on the same pointer twice.
- The result is undefined, but it often corrupts the memory allocator's internal data structures, leading to a crash later.

```
int *p = malloc(sizeof(int));
free(p);
// ...
free(p); // Double free!
```

Invalid Free

- Calling `free()` on a pointer that was not returned from `malloc()`.
- Examples: pointers to the stack, or pointers to the middle of a heap allocation.

```
int stack_var;
free(&stack_var); // Invalid!

int *arr = malloc(10*sizeof(
    int));
free(arr + 5); // Invalid!
```

Underlying OS Support

System Calls vs. Library Calls

`malloc()` and `free()` are **not** system calls. They are library functions in the C standard library.

- The malloc library manages your program's heap internally.
- When it needs more memory from the OS, it uses system calls like `brk()` or `mmap()` to grow the heap.
- You should almost never call `brk()` or `mmap()` directly for general purpose allocation. Stick to `malloc()` and `free()`.

Other Useful Functions

`calloc()`

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

- Allocates memory for an array of `num` elements of `size` bytes each.
- **Crucially, it zeroes out the memory before returning.** This prevents uninitialized reads.

```
// Allocates and zeroes 10 integers
int *p = (int *) calloc(10, sizeof(int));
```

`realloc()`

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

- Resizes a previously allocated block of memory pointed to by `ptr`.
- Can be used to make a buffer larger or smaller.

Finding Memory Bugs: Tools

You are not alone!

Because these bugs are so common and so hard to find, powerful tools have been developed to help.

Valgrind

An instrumentation framework that can detect memory management and threading bugs. Its `memcheck` tool is essential for C programmers.

- Detects: Memory leaks, use-after-free, double frees, buffer overflows, use of uninitialized memory.

```
# Compile with -g to get line numbers in error reports
gcc -g -o my_program my_program.c

# Run with memcheck to find memory errors
valgrind --leak-check=full ./my_program
```

Listing 9: Running a program with Valgrind

Finding Memory Bugs: Tools

You are not alone!

Because these bugs are so common and so hard to find, powerful tools have been developed to help.

GDB (The GNU Debugger)

- Helps you see what is going on 'inside' another program while it executes.
- Essential for finding the cause of crashes
  - Like segmentation faults

Summary

- Memory is managed either on the **stack** (automatic) or the **heap** (manual).
- Use `malloc()` to request memory from the heap.
- Use `free()` to return it.
- Be vigilant about common errors:
  - Buffer Overflows
  - Memory Leaks
  - Dangling Pointers
  - Uninitialized Reads
  - Double / Invalid Frees
- **Use tools like Valgrind!** They will save you countless hours of debugging.