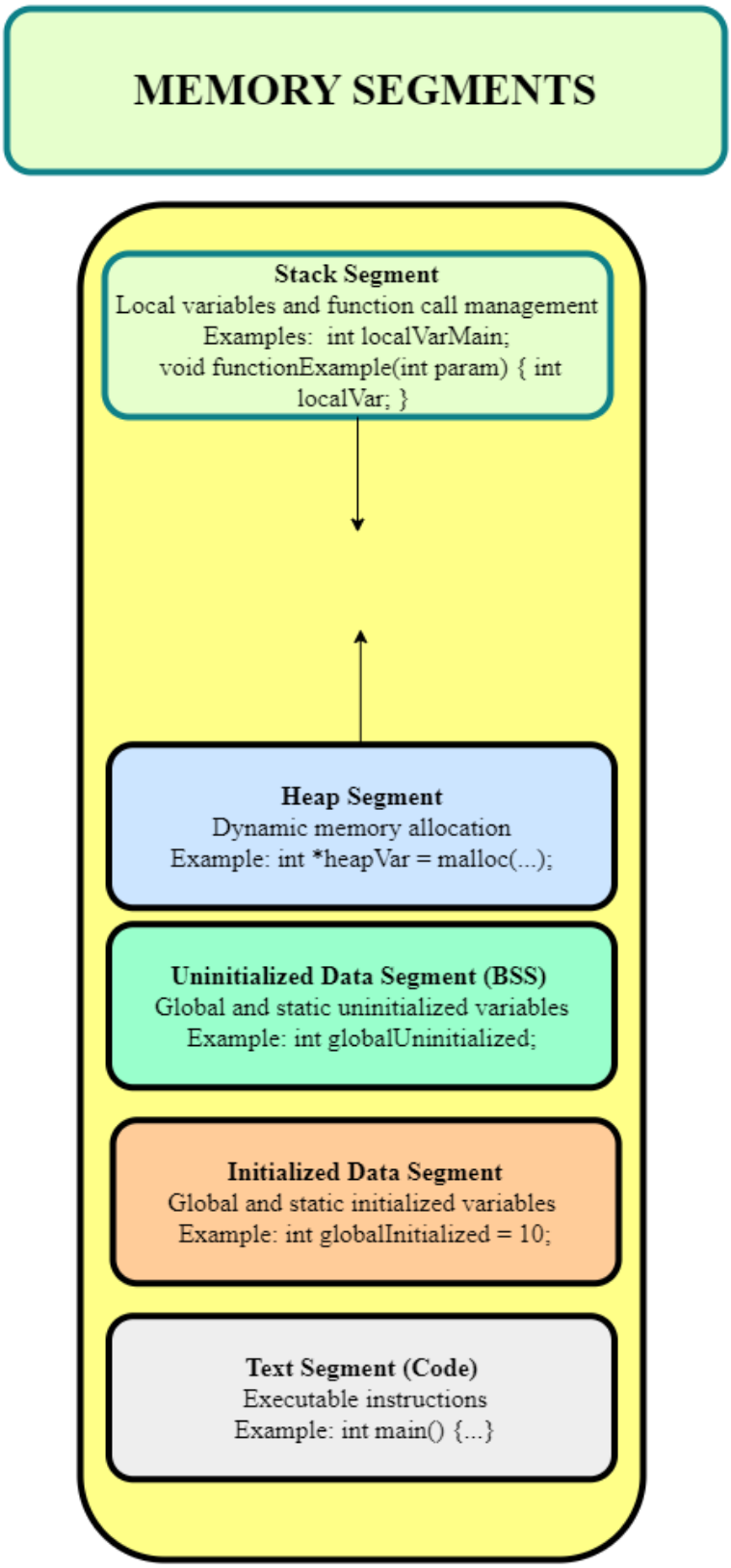
A VISIT TO MEMORY SEGMENTS



In general, in computers, memory is divided into segments or sections: Stack, Heap, uninitialized data, initialized data, and text segments.

Yashwanth Naidu Tikkisetty

Computer Memory Segments

1. **Text Segment**:
    - **Description**: Also known as the code segment, this section contains the compiled program's code. It is usually read-only to prevent accidental modification of instructions.
    - **Usage**: Stores the executable instructions of a program.
    - **Protection**: Typically, this segment is marked as read-only to prevent modification. Attempts to write to this segment can cause a segmentation fault.

```
// The following line of code would cause a segmentation fault
// if we try to modify the text segment:
const char *code = "text segment";
code[0] = 'T';  // Segmentation fault
```

2. **Data Segment**:
    - **Initialized Data Segment**:
        - **Description**: This segment contains global and static variables that are initialized by the programmer. The values are set before the program starts running.
        - **Usage**: Stores global and static variables with defined initial values.

```
int initializedVar = 10;  // Stored in initialized data segment
```

    - **Uninitialized Data Segment (BSS)**:
        - **Description**: The BSS (Block Started by Symbol) segment contains global and static variables that are not initialized by the programmer.
        - The BSS directive would define a label (symbol) and allocate space for the uninitialized data block starting at that label. The size of the block was specified, but its contents were not initialized, meaning the memory was simply reserved.
        - **Usage**: Stores global and static variables that default to zero.
        - **Example**: `int globalVar;` without an initial value is stored here.

```
int globalVar;  // Stored in BSS segment
```

3. **Heap Segment**:
    - **Description**: The heap is used for dynamic memory allocation. It grows and shrinks as memory is allocated (`malloc`, `new`) and deallocated (`free`, `delete`).
    - **Usage**: Used for variables that need to persist across function calls or have a size that cannot be determined at compile time.
    - **Management**: Managed manually by the programmer. Improper management can lead to memory leaks or corruption.

```
int *heapVar = (int*)malloc(sizeof(int) * 10);  // Allocate memory on the heap
if (heapVar == NULL) {
    // Handle memory allocation failure
}
free(heapVar);  // Free the allocated memory
```

4. **Stack Segment**:
    - **Description**: The stack is used for local variables and function call management. It follows a Last In, First Out (LIFO) structure.
    - **Usage**: Stores function parameters, return addresses, and local variables.
    - **Growth**: The stack grows and shrinks as functions are called and return.
    - **Limitations**: Stack overflow can occur if too much memory is used (e.g., with deep recursion or large local arrays).

```
void recursiveFunction(int count) {
    int stackVar = 10;  // Local variable stored on the stack
    if (count > 0) {
        recursiveFunction(count - 1);  // Recursive call
    }
}
```

Memory Access and Faults

When a program attempts to access a memory segment in a way that it is not allowed, a fault occurs. If it goes beyond a section/segment, we call it a segmentation fault.

- **Segmentation Fault**:
    - **Description**: Occurs when a program tries to access a memory segment it is not allowed to, or attempts to access out-of-bound sections.

- **Cause**: This happens within the virtual address space of the process. Common causes include dereferencing null or uninitialized pointers, buffer overflows, and accessing memory that the program doesn't have permissions for.

```
int *ptr = NULL;
*ptr = 10;  // Segmentation fault: dereferencing a null pointer
```

- **Page Fault**:
  - **Description**: Occurs when a program tries to access a portion of memory (a page) that is not currently mapped to RAM.
  - **Handling**: The operating system handles page faults by either loading the required page from disk into RAM or terminating the program if the access is

Segmentation and Faults in Memory Management

In general, in computers, memory is divided into segments or sections: Stack, Heap, uninitialized data, initialized data, and text segments.

When a program attempts to access a memory segment in a way that it is not allowed, a fault occurs. Since it goes beyond a section/segment, we call it a segmentation fault.

In early days, for old computers, we could directly access the entire memory space. As the complexity of programming increased with advancing computing technologies, more sophisticated memory management techniques were introduced.

Segmentation refers to a memory management technique where memory is divided into different segments. Each segment has its own base address and length. This 'segmenting' of memory into various sections was introduced to isolate different types of memory usage and to provide memory protection.

Now, segmentation faults and page faults are different.

Segmentation Fault occurs when a program tries to access a memory segment that it is not allowed to access or when it attempts to access out-of-bound sections. It occurs within the virtual address space of the process.

Page Fault occurs when a program tries to access a portion of memory (a page) that is not currently mapped to RAM.

Summarized:

Memory Segments Explained:

1. **Text Segment**: Contains the compiled program's code and is typically read-only.
2. **Data Segment**:
    - **Initialized Data**: Stores global and static variables initialized by the programmer.
    - **Uninitialized Data (BSS)**: Stores global and static variables that are not initialized by the programmer, defaulting to zero.
3. **Heap Segment**: Used for dynamic memory allocation and grows/shrinks as needed.
4. **Stack Segment**: Manages function calls and local variables, growing and shrinking with each call and return.

```c
#include <stdio.h>
#include <stdlib.h>

// Global variable, uninitialized (BSS segment)
int globalUninitialized;

// Global variable, initialized (Data segment)
int globalInitialized = 10;

void functionExample(int param) {  // param is in the Stack segment
    // Local variable (Stack segment)
    int localVar = 20;

    // Static local variable (Data segment)
    static int staticLocalVar = 30;

    printf("Parameter: %d\n", param);
    printf("Local variable: %d\n", localVar);
    printf("Static local variable: %d\n", staticLocalVar);
}

int main() {
    // Local variable (Stack segment)
    int localVarMain = 40;

    // Dynamic memory allocation (Heap segment)
    int *heapVar = (int*)malloc(sizeof(int) * 5);
    if (heapVar == NULL) {
        // Handle memory allocation failure
        printf("Memory allocation failed\n");
        return 1;
    }
```

```
    // Initialize the allocated memory
    for (int i = 0; i < 5; i++) {
        heapVar[i] = i * 10;
    }

    // Function call (uses Stack segment)
    functionExample(localVarMain);

    // Printing global variables
    printf("Global uninitialized variable: %d\n", globalUninitialized);
    printf("Global initialized variable: %d\n", globalInitialized);

    // Printing dynamic memory content
    for (int i = 0; i < 5; i++) {
        printf("Heap variable [%d]: %d\n", i, heapVar[i]);
    }

    // Free allocated memory (Heap segment)
    free(heapVar);

    return 0;
}

// Code segment starts here (Text segment)
// The actual instructions for main() and functionExample() are stored here
```

Article Written By: Yashwanth Naidu Tikkisetty