18ES611 Embedded System Programming

Sarath tv

Compile Time vs Run Time.

Types of allocation — static, automatic, and dynamic.

Static Allocation means, that the memory for your variables is allocated when the program starts.
- The size is fixed when the program is created.
- Applies to global variables, file scope variables, static local variable.
- Is allocated at compile time
- *allocation of static memory is handled by the compiler*

Automatic memory allocation variables defined inside functions,
- Stored on the *stack*.
- You do not have to reserve extra memory using them,
- limited control over the lifetime of this memory.

Dynamic memory allocation. Control the exact size and the lifetime of these memory locations.
- Memory leaks, which may cause your application to crash,
- When you need a lot of memory.
- ensures you allocate exactly as much memory as you really need

Heap

- Heap is the segment where **dynamic memory allocation** usually takes place.
- The Heap area is managed by **malloc**, **realloc**, and **free**.
- The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

The Concept of Heap and Its Usage in Embedded Systems

- The heap is a segment of the system memory (RAM) that provides dynamic memory allocation.
- Dynamic allocation usually requires **two basic steps**:
- *Creating the dynamic space*
- *Creating a pointer holding the address for the newly created space (new variable names can't be created while the program is running, so pointers are needed)*
- There are several functions (part of *stdlib* library) for dynamic memory allocation and management:

malloc() and calloc( ) – reserve space

realloc() – move a reserved block of memory to another allocation of different dimensions

free() – release allocated space

Heap Usage

Here are some of the most common cases when the heap is used:

- When we need a **data** that must **live** after the **function returns**. Once data is stored on the heap during **function execution** it is **not affected** when the **function ends**
- When we **need a lot of memory**
- When we don’t know exactly **how much data** we will **need to store** during the execution of the program

Heap Usage Pitfalls

Using **dynamic memory allocation** brings a certain **degree** of **complexity** and if the end application does **not explicitly requires** it, the usage of heap should be **avoided** (especially in small embedded systems).

There are a lot of specifics that a person should be familiar with before deciding on using the heap.

Below are listed some **common pitfalls**, when the heap integrity is compromised:

Overwritten heap data.

Allocation failures, when a too big buffer is requested to be allocated.

Heap is responsible for memory leaks

The C malloc() and free() function can take a long time to execute, which is in conflict with the real-time constraints of an embedded system

Dynamic Memory Allocation

A program can at any time request for additional memory.

The memory is allocated from a data structure known as a heap.

Memory allocation occurs via an **system call** (such as malloc in C).

When the program **no longer needs** access to memory that has been so allocated**, it deallocates** the memory (by calling free in C).

A garbage collector is a **task** that **runs** either periodically or when memory gets tight that analyzes the data structures that a program has **allocated and automatically frees any portions of memory that are no longer referenced within the program**.

calloc() & malloc()

Initialization: malloc() **allocates memory block** of given **size** (in **bytes**) and returns a **pointer** to the **beginning** of the **block**.

malloc() **doesn't initialize** the allocated memory.

access the content of memory block -**garbage values**.

calloc() **allocates** the **memory** and also **initializes** the allocates memory **block** to **zero**.

access the content of these blocks then we'll get 0.

Number of arguments: Unlike malloc(), calloc() takes two arguments:
1) Number of blocks to be allocated.
2) Size of each block.

Return Value: After successful allocation in malloc() and calloc(), a **pointer** to the block of memory is returned otherwise **NULL** value is returned which indicates the failure of allocation

```
void * malloc( size_t size );
```

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

```
 #include <stdio.h>

 #include <stdlib.h>

  int main() {

    int *arr;

 // malloc() allocate the memory for 5
integers containing garbage values

 arr = (int *)malloc(5 * sizeof(int)); //
 5*4bytes


 // Deallocates memory previously allocated
 by malloc() function

    free( arr );

// calloc() allocate the memory for 5
integers and

   // set 0 to all of them

   arr = (int *)calloc(5, sizeof(int));


   // Deallocates memory previously
allocated by calloc() function

   free(arr);


   return(0);

}
```

Realloc

If suppose we allocated more or less memory than required, then we can change the size of the previously allocated memory space using realloc.

void *realloc(pointer, new-size);

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
char *p1;
int m1, m2;
m1 = 10; m2 = 30;
p1 = (char*)malloc(m1);
strcpy(p1, “Embedded");
p1 = (char*)realloc(p1, m2);
strcat(p1, “System Programming");
printf("%s\n", p1);
 return 0;
}
```

When to use the Heap?

If you need to allocate a **large block of memory** (e.g. a large array, or a big struct), and you need to **keep** that **variable** around a **long time** (like a global), then you should allocate it on the **heap**. If you are dealing with relatively **small variables** that only need to **persist** as **long** as the **function** using them is **alive**, then you should use the **stack**, it's **easier** and **faster**.

If you need variables like arrays and structs that can change size dynamically (e.g. arrays that can grow or shrink as needed) then you will likely need to allocate them on the heap, and use dynamic memory allocation functions like malloc(), calloc(), realloc() and free() to manage that memory "by hand"

Stack vs Heap Pros and Cons

Stack

•very fast access
•don't have to explicitly de-allocate variables
•space is managed efficiently by CPU, memory will not become fragmented
•local variables only
•limit on stack size
•variables cannot be resized

Heap

•variables can be accessed globally
•no limit on memory size
•(relatively) slower access
•no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
•you must manage memory (you're in charge of allocating and freeing variables)
•variables can be resized using realloc()

Key Differences Between Stack and Heap Allocations

- In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
- Handling of Heap frame is costlier than handling of stack frame.
- Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
- Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
- Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
- Accessing time of heap takes is more than a stack.

Variables/automatic variables ---> stack section

Dynamically allocated variables ---> heap section

Initialized global variables -> data section

Uninitialized global variables -> data section (bss)

Static variables -> data section

String constants -> text section/code section

Text code -> text section/code section

Registers -> CPU registers

THANK YOU!!!!!