1 OPERATING SYSTEMS LABORATORY VIII - Memory Management I

1.1 Memory Management

Memory management is the art and the process of coordinating and controlling the use of memory in a computer system. Memory management can be divided into three areas:

1. *Memory management hardware (MMUs, RAM, etc.)* consists of the electronic devices and associated circuitry that store the state of a computer. These devices include RAM, MMUs (memory management units), caches, disks, and processor registers. The design of memory hardware is critical to the performance of modern computer systems. In fact, memory bandwidth is perhaps the main limiting factor on system performance.

2. *Operating system memory management (virtual memory, protection)* is concerned with using the memory management hardware to manage the resources of the storage hierarchy and allocating them to the various activities running on a computer. The most significant part of this on many systems is virtual memory, which creates the illusion that every process has more memory than is actually available. OS memory management is also concerned with memory protection and security, which help to maintain the integrity of the operating system against accidental damage or deliberate attack. It also protects user programs from errors in other programs.

3. *Application memory management (allocation, deallocation, garbage collection)* involves obtaining memory from the operating system, and managing its use by an application program. Application programs have dynamically changing storage requirements. The application memory manager must cope with this while minimizing the total CPU overhead, interactive pause times, and the total memory used.

- The **malloc()** function allocates an uninitialized memory block. It allocates a specified number of bytes of memory, as shown in the following prototype, returning a pointer to the newly allocated memory or NULL on failure:

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

- The **calloc()** function allocates and initializes a memory block. The difference is that **calloc()** initializes the allocated memory, setting each bit to 0, returning a pointer to the memory or NULL on failure. It uses the following prototype:

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

- The **realloc()** function resizes a previously allocated memory block. Use **realloc()** to resize memory previously obtained with a **malloc()** or **calloc()** call. This function uses the following prototype:

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

  - The *ptr* argument must be a pointer returned by **malloc()** or **calloc()**.
  - The *size* argument may be larger or smaller than the size of the original pointer.

- The **free()** function frees a block of memory. This function uses the following prototype:

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

  - The *ptr* argument must be a pointer returned from a previous call to **malloc()** or calloc().
  - It is an error to attempt to access memory that has been freed.

  Memory allocation functions obtain memory from a storage pool known as the *heap*.

- The **alloca()** function allocates an uninitialized block of memory. This function uses the following prototype:

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

  alloca() obtains memory from the process's *stack* rather than the *heap* and, when the function that invoked **alloca()** returns, the allocated memory is automatically freed.

1.1.1 Examples&Exercises:

1. Using Dynamic Memory Management Functions; code45.c

   - Illustrates the standard library's memory management functions.
   - Lines 15-18 illustrate **malloc()** usage. It is attempted to allocate ten bytes of memory, check **malloc()**'s return value, display the contents of the uninitialized memory, and then return the memory to the *heap*.
   - Lines 20-22 repeat this procedure for **calloc()**.
   - Rather than freeing d, however, it is attempted to extend it on lines 26-28. Whether **realloc()** succeeds or fails, it should still point to the string "foobar".
   - The pointer, e, as shown on lines 31-33, is allocated off the *stack* and, when **main()** returns (that is, when the program exits), its memory is automatically freed.

2. Dangling Pointers;

   ```
   char *str;
   str = malloc(sizeof(char) * 4)
   free(str);
   strcpy(str, "abc");
   ```

   - Write a program that containing the code segment above.
   - What kind of an error will you obtain? Why?

3. A Problem Child; code46.c.

   - C assumes you know what you are doing, most C compilers ignore uses of uninitialized memory, buffer overruns, and buffer underruns.
   - Nor do most compilers catch memory leaks or dangling pointers.
   - Bugs in the program:
     - A memory leak (line 18),
     - Overruns the end of dynamically allocated heap memory (lines 22 and 28),
     - Underruns a memory buffer (line 32),
     - Frees the same buffer twice (lines 36 and37),

  - Accesses freed memory (lines 40 and41),
  - Clobbers statically allocated stack and global memory (lines 48 and 44, respectively)

- These bugs can prevent the program from executing depending on the configuration (to allow core dumps), but leaks and clobbered memory usually show up as unpredictable behavior elsewhere in the program.
- Fix the errors (if occurs).

1.2 Memory Arrangement

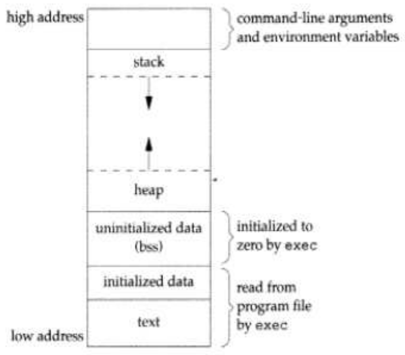
high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Figure 1: Typical Memory Arrangement

1. ***Text segment:*** This is the machine instructions that are executed by the CPU. Usually the segment is sharable so that only a single copy needs to be in memory for frequently executed programs. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

   - ***Initialized data segment:*** This is usually just called the data segment and it contains variables that are specifically initialized in the program. For example, the C declaration

     `int maxcount=99;`

     appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

- ***Uninitialized data segment:*** This segment is often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or *null* pointers before the program starts executing. The C declaration

  ```
  long sum[1000];
  ```

  appearing outside any function causes this variable to be stored in the uninitialized data segment.

2. ***Stack:*** This is where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to, and certain information about the caller's environment (such as some of the machine registers) is saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. By utilizing a stack in this fashion, C functions can be recursive.

3. ***Heap:*** Dynamic memory allocation usually takes place on the heap. Historically the heap has been located between the top of the uninitialized data and the bottom of the stack.

Summary;

- Initialized Read Only Data; This contains the data elements that are initialized by the program and they are read only during the execution of the process.

- Initialized Read Write Data; This contains the data elements that are initialized by the program and will be modified in the course of process execution.

- Uninitialized Data; This contains the elements are not initialized by the program and are set 0 before the processes executes. These can also be modified and referred as BSS(Block Started Symbol).

- Stack; This portion is used for local variables, stack frames.

- Heap; This portion contains the dynamically allocated memory.

```
int abc = 1;          ----> Initialized Read-Write Data
char *str;            ----> BSS
const int i = 10;     ----> Initialized Read-Only Data

main()
{
   int ii,a=1,b=2,c; ----> Local Variables on Stack
   char *ptr;
   ptr = malloc(4);  ----> Allocated Memory in Heap
   c= a+b;           ----> Text
}
```

1.2.1 Examples&Exercises:

1. Study the command;

   ```
   $ size /usr/bin/gcc /bin/sh
   ```

   - Try to explain what you obtained!

2. Memory layout of a C program; a program to display process memory boundary ; code47.c

   - Study the code in detail.
   - What are the followings?

     ```
     extern int _end;
     extern int _etext;
     extern int _edata;
     extern int __bss_start;
     extern char **environ;
     ```

     Make a search about them.
   - Analyze the positions and lengths of the portions.
   - Try to understand the order of the appearance of the variables (local, global, automatic, initialized, uninitialized).
   - Modify to code such that
     - able to print out the content of the variables,
     - increase the number of the arguments,
     - able to print out arguments,
     - able to print out environment.