|< The main() and command line arguments | Main | C Memory Allocation and De-allocation Functions >| Site Index | Download |

COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY

My Training Period: xx hours

Note:
This Module presents quite a detail story of a process (running program). However, it is an excerpt from more complete, Tenouk's buffer overflow Tutorial. It tries to investigate how the C/C++ source codes preprocessed, compiled, linked and loaded as a running program. It is based on the GCC (GNU Compiler Collection). When you use the IDE (Integrated Development Environment) compilers such as Microsoft Visual C++, Borland C++ Builder etc. the processes discussed here quite transparent. The commands and examples of the gcc, gdb, g++, gas and friends are discussed in Linux gnu gcc, g++, gdb and gas 1 and Linux gnu gcc, g++, gdb and gas 2. Have a nice day!

The C compiler ability:

- Able to understand and appreciate the processes involved in preprocessing, compiling, linking, loading and running C/C++ programs.

W.1 COMPILERS, ASSEMBLERS and LINKERS

- Normally the C's program building process involves four stages and utilizes different 'tools' such as a preprocessor, compiler, assembler, and linker.
- At the end there should be a single executable file. Below are the stages that happen in order regardless of the operating system/compiler and graphically illustrated in Figure w.1.

    1. **Preprocessing** is the first pass of any C compilation. It processes include-files, conditional compilation instructions and macros.
    2. **Compilation** is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.
    3. **Assembly** is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.
    4. **Linking** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

- Bear in mind that if you use the IDE type compilers, these processes quite transparent.
- Now we are going to examine more details about the process that happen before and after the linking stage. For any given input file, the file name suffix (file extension) determines what kind of compilation is done and the example for GCC is listed in Table w.1.
- In UNIX/Linux, the executable or binary file doesn't have extension whereas in Windows the executables for example may have .**exe**, **.com** and **.dll**.

File extension	Description
file_name.c	C source code which must be preprocessed.
file_name.i	C source code which should not be preprocessed.
file_name.ii	C++ source code which should not be preprocessed.
file_name.h	C header file (not to be compiled or linked).
file_name.cc file_name.cp file_name.cxx file_name.cpp file_name.c++ file_name.C	C++ source code which must be preprocessed. For file_name.cxx, the xx must both be literally character x and file_name.C, is capital c.
file_name.s	Assembler code.
file_name.S	Assembler code which must be preprocessed.
file_name.o	Object file by default, the object file name for a source file is made by replacing the extension .c, .i, .s etc with .o

Table w.1

- The following Figure shows the steps involved in the process of building the C program starting from the compilation until the loading of the executable image into the memory for program running.

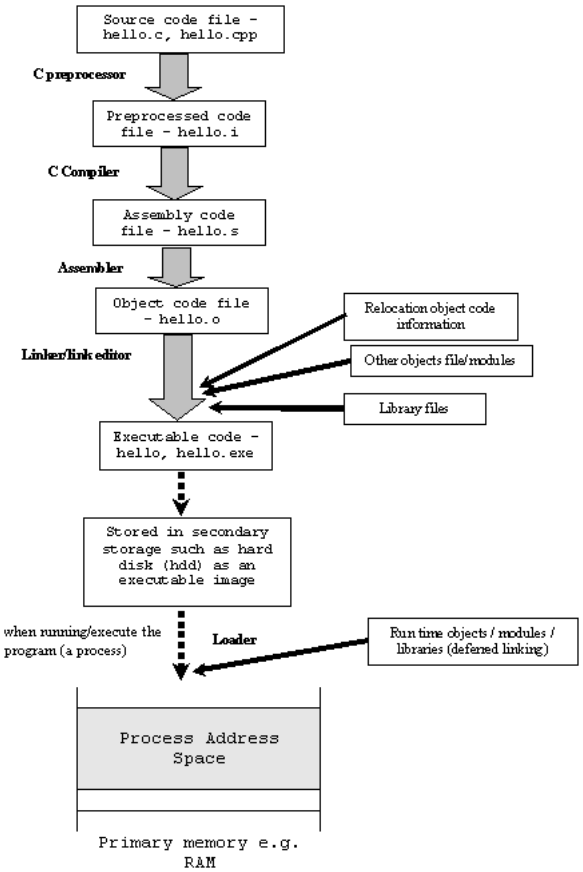

Figure w.1: Compile, link and execute stages for running program (a process)

W.2 OBJECT FILES and EXECUTABLE

- After the source code has been assembled, it will produce an **Object** files (e.g. **.o**, **.obj**) and then linked, producing an executable files.
- An object and executable come in several formats such as **ELF** (Executable and Linking Format) and **COFF** (Common Object-File Format). For example, ELF is used on Linux systems, while COFF is used on Windows systems.
- Other object file formats are listed in the following Table.

Object File Format	Description
a.out	The a.out format is the original file format for Unix. It consists of three sections: text, data, and bss, which are for program code, initialized data, and uninitialized data, respectively. This format is so simple that it doesn't have any reserved place for debugging information. The only debugging format for a.out is stabs, which is encoded as a set of normal symbols with distinctive attributes.
COFF	The COFF (Common Object File Format) format was introduced with System V Release 3 (SVR3) Unix. COFF files may have multiple sections, each prefixed by a header. The number of sections is limited. The COFF specification includes support for debugging but the debugging information was limited. There is no file extension for this format.
ECOFF	A variant of COFF. ECOFF is an Extended COFF originally introduced for Mips and Alpha workstations.
XCOFF	The IBM RS/6000 running AIX uses an object file format called XCOFF (e**X**tended COFF). The COFF sections, symbols, and line numbers are used, but debugging symbols are dbx-style stabs whose strings are located in the .debug section (rather than the string table). The default name for an XCOFF executable file is a.out.
PE	Windows 9x and NT use the PE (Portable Executable) format for their executables. PE is basically COFF with additional headers. The extension normally .exe.
ELF	The ELF (Executable and Linking Format) format came with System V Release 4 (SVR4) Unix. ELF is similar to COFF in being organized into a number of sections, but it removes many of COFF's limitations. ELF used on most modern Unix systems, including GNU/Linux, Solaris and Irix. Also used on many embedded systems.
SOM/ESOM	SOM (System Object Module) and ESOM (Extended SOM) is HP's object file and debug format (not to be confused with IBM's SOM, which is a cross-language Application Binary Interface - ABI).

Table w.2

- When we examine the content of these object files there are areas called sections. Sections can hold executable code, data, dynamic linking information, debugging data, symbol tables, relocation information, comments, string tables, and notes.
- Some sections are loaded into the process image and some provide information needed in the building of a process image while still others are used only in linking object files.
- There are several sections that are common to all executable formats (may be named differently, depending on the compiler/linker) as listed below:

Section	Description
.text	This section contains the executable instruction codes and is shared among every process running the same binary. This section usually has READ and EXECUTE permissions only. This section is the one most affected by optimization.

.bss	BSS stands for 'Block Started by Symbol'. It holds un-initialized global and static variables. Since the BSS only holds variables that don't have any values yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data section) doesn't take up any actual space in the object file.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.
.rdata	Also known as .rodata (read-only data) section. This contains constants and string literals.
.reloc	Stores the information required for relocating the image while loading.
Symbol table	A symbol is basically a name and an address. Symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The symbol table contains an array of symbol entries.
Relocation records	Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Re-locatable files must have relocation entries' which are necessary because they contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Simply said relocation records are information used by the linker to adjust section contents.

Table w.3: Segments in executable file

- The following is an example of the object file content dumping using readelf program. Other utility can be used is objdump. These utilities presented in Linux gcc, g++, gdb and gas 1 and Linux gcc, g++, gdb and gas 2.
- For Windows dumpbin utility (coming with Visual C++ compiler) or more powerful one is a free PEBrowse program that can be used for the same purpose.

```
/* testprog1.c */
#include <stdio.h>
static void display(int i, int *ptr);

int main(void)
{
    int x = 5;
    int *xptr = &x;
    printf("In main() program:\n");
    printf("x value is %d and is stored at address %p.\n", x, &x);
    printf("xptr pointer points to address %p which holds a value of %d.\n", xptr, *xptr);
    display(x, xptr);
    return 0;
}

void display(int y, int *yptr)
{
    char var[7] = "ABCDEF";
    printf("In display() function:\n");
    printf("y value is %d and is stored at address %p.\n", y, &y);
    printf("yptr pointer points to address %p which holds a value of %d.\n", yptr, *yptr);
}
```

```
[bodo@bakawali test]$ gcc -c testprog1.c
[bodo@bakawali test]$ readelf -a testprog1.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          672 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 0000de 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 00052c 000068 08      9   1  4
  [ 3] .data             PROGBIT         00000000 000114 000000 00         WA  0   0  4
  [ 4] .bss              NOBIT           00000000 000114 000000 00  WA  0   0  4
  [ 5] .rodata               PROGBITS        00000000 000114 00010a 00      A  0   0  4
  [ 6] .note.GNU-stack       PROGBITS        00000000 00021e 000000 00      0   0  1
```

```
  [ 7] .comment      PROGBITS      00000000 00021e 000031 00      0   0  1
  [ 8] .shstrtab     STRTAB 00000000 00024f 000051 00       0   0  1
  [ 9] .symtab              SYMTAB 00000000 000458 0000b0 10     10  9  4
  [10] .strtab              STRTAB 00000000 000508 000021 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no program headers in this file.

Relocation section '.rel.text' at offset 0x52c contains 13 entries:
 Offset       Info                Type   Sym.Value  Sym. Name
0000002d  00000501 R_386_32 00000000   .rodata
00000032  00000a02 R_386_PC32      00000000   printf
00000044  00000501 R_386_32 00000000   .rodata
00000049  00000a02 R_386_PC32      00000000   printf
0000005c  00000501 R_386_32 00000000   .rodata
00000061  00000a02 R_386_PC32      00000000   printf
0000008c  00000501 R_386_32 00000000   .rodata
0000009c  00000501 R_386_32 00000000   .rodata
000000a1  00000a02 R_386_PC32      00000000   printf
000000b3  00000501 R_386_32 00000000   .rodata
000000b8  00000a02 R_386_PC32      00000000   printf
000000cb  00000501 R_386_32 00000000   .rodata
000000d0  00000a02 R_386_PC32      00000000   printf

There are no unwind sections in this file.

Symbol table '.symtab' contains 11 entries:
   Num:    Value     Size Type    Bind        Vis             Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS testprog1.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000080     94 FUNC   LOCAL  DEFAULT    1 display
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000     0 SECTION LOCAL  DEFAULT    7
     9: 00000000     128 FUNC  GLOBAL DEFAULT    1 main
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf

No version information found in this file.
```

- When writing a program using the assembly language it should be compatible with the sections in the assembler directives (x86) and the partial list that is interested to us is listed below:

	Section	Description
1	Text (.section .text)	Contain code (instructions). Contain the _start label.
2	Read-Only Data (.section .rodata)	Contains pre-initialized constants.
3	Read-Write Data (.section .data)	Contains pre-initialized variables.
4	BSS (.section .bss)	Contains un-initialized data.

Table w.4

- The assembler directives in assembly programming can be used to identify code and data sections, allocate/initialize memory and making symbols externally visible or invisible.
- An example of the assembly code with some of the assembler directives (Intel) is shown below:

```
;initializing data
            .section        .data
x:          .byte           128                     ;one byte initialized to 128
y:          .long           1,1000,10000            ;3 long words

;initializing ascii data
            .ascii          "hello"                 ;ascii without null character
            asciz           "hello"                 ;ascii with \0

;allocating memory in bss
            .section        .bss
            .equ            BUFFSIZE 1024           ;define a constant
            .comm           z, 4, 4                 ;allocate 4 bytes for x with
                                                    ;4-byte alignment

;making symbols externally visible
            .section        .data
            .globl          w                       ;declare externally visible
                                                    ;e.g: int w = 10
            .text
            .globl          fool                    ;e.g: fool(void) {…}
```

```
fool:
            …
            leave
            return
```

W.3 RELOCATION RECORDS

- Because the various object files will include references to each others code and/or data, so various locations, these shall need to be combined during the link time.
- For example in Figure w.2, the object file that has main() includes calls to functions funct() and printf().
- After linking all of the object files together, the linker uses the relocation records to find all of the addresses that need to be filled in.

W.4 SYMBOL TABLE

- Since assembling to machine code removes all traces of labels from the code, the object file format has to keep these around in different places.
- It is accomplished by the symbol table that contains a list of names and their corresponding offsets in the text and data segments.
- A disassembler provides support for translating back from an object file or executable.

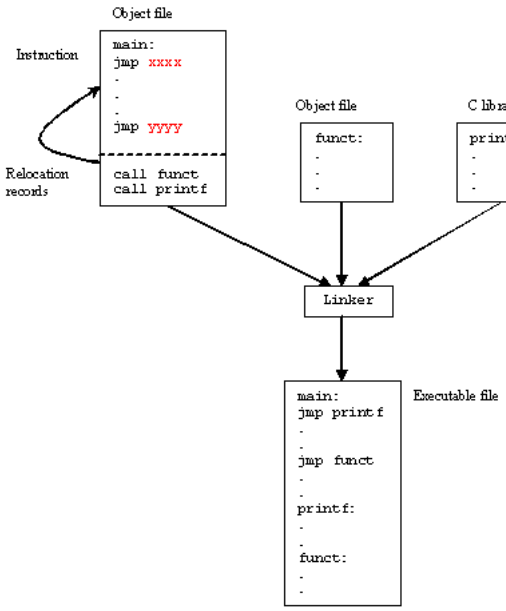

Figure w.2: The relocation record

W.5 LINKING

- The linker actually enables separate compilation. As shown in Figure w.3, an executable can be made up of a number of source files which can be compiled and assembled into their object files respectively, independently.

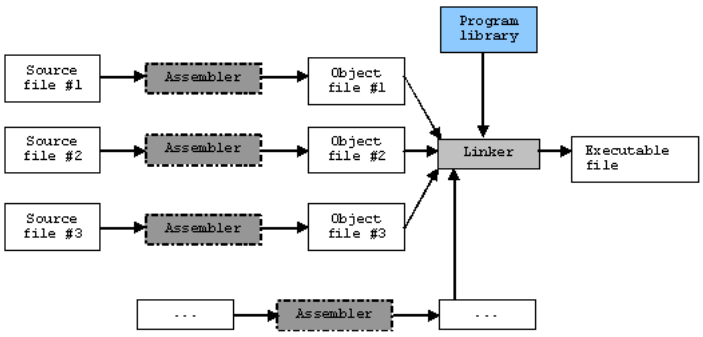

Figure w.3: The object files linking process

W.5.1 SHARED OBJECTS

- In a typical system, a number of programs will be running. Each program relies on a number of functions, some of which will be standard C library functions, like printf(), malloc(), strcpy(), etc. and some are non-standard or user defined functions.
- If every program uses the standard C library, it means that each program would normally have a unique copy of this particular library present within it. Unfortunately, this result in wasted resources, degrade the efficiency and performance.
- Since the C library is common, it is better to have each program reference the common, one instance of that library, instead of having each program contain a copy of the library.
- This is implemented during the linking process where some of the objects are linked during the link time whereas some done during the run time (deferred/dynamic linking).

W.5.2 STATICALLY LINKED

- The term 'statically linked' means that the program and the particular library that it's linked against are combined together by the linker at link time.
- This means that the binding between the program and the particular library is fixed and known at link time before the program run. It also means that we can't change this binding, unless we re-link the program with a new version of the library.

- Programs that are linked statically are linked against archives of objects (libraries) that typically have the extension of **.a**. An example of such a collection of objects is the standard C library, **libc.a**.
- You might consider linking a program statically for example, in cases where you weren't sure whether the correct version of a library will be available at runtime, or if you were testing a new version of a library that you don't yet want to install as shared.
- For gcc, the –static option can be used during the compilation/linking of the program.

```
gcc –static filename.c –o filename
```

- The drawback of this technique is that the executable is quite big in size, all the needed information need to be brought together.

W.5.3 DYNAMICALLY LINKED

- The term 'dynamically linked' means that the program and the particular library it references are not combined together by the linker at link time.
- Instead, the linker places information into the executable that tells the loader which shared object module the code is in and which runtime linker should be used to find and bind the references.
- This means that the binding between the program and the shared object is done at runtime that is before the program starts, the appropriate shared objects are found and bound.
- This type of program is called a partially bound executable, because it isn't fully resolved. The linker, at link time, didn't cause all the referenced symbols in the program to be associated with specific code from the library.
- Instead, the linker simply said something like: "This program calls some functions within a particular shared object, so I'll just make a note of which shared object these functions are in, and continue on".
- Symbols for the shared objects are only verified for their validity to ensure that they do exist somewhere and are not yet combined into the program.
- The linker stores in the executable program, the locations of the external libraries where it found the missing symbols. Effectively, this defers the binding until runtime.
- Programs that are linked dynamically are linked against shared objects that have the extension **.so**. An example of such an object is the shared object version of the standard C library, **libc.so**.
- The advantageous to defer some of the objects/modules during the static linking step until they are finally needed (during the run time) includes:

1. Program files (on disk) become much smaller because they need not hold all necessary text and data segments information. It is very useful for portability.
2. Standard libraries may be upgraded or patched without every one program need to be re-linked. This clearly requires some agreed module-naming convention that enables the dynamic linker to find the newest, installed module such as some version specification. Furthermore the distribution of the libraries is in binary form (no source), including dynamically linked libraries (DLLs) and when you change your program you only have to recompile the file that was changed.
3. Software vendors need only provide the related libraries module required. Additional runtime linking functions allow such programs to programmatically-link the required modules only.
4. In combination with virtual memory, dynamic linking permits two or more processes to share read-only executable modules such as standard C libraries. Using this technique, only one copy of a module needs be resident in memory at any time, and multiple processes, each can executes this shared code (read only). This results in a considerable memory saving, although demands an efficient swapping policy.

W.6 HOW SHARED OBJECTS ARE USED

- To understand how a program makes use of shared objects, let's first examine the format of an executable and the steps that occur when the program starts.

W.6.1 SOME ELF FORMAT DETAILS

- Executable and Linking Format (ELF) is binary format, which is used in SVR4 Unix and Linux systems.
- It is a format for storing programs or fragments of programs on disk, created as a result of compiling and linking.
- ELF not only simplifies the task of making shared libraries, but also enhances dynamic loading of modules at runtime.

W.6.2 ELF SECTIONS

- The Executable and Linking Format used by GNU/Linux and other operating systems, defines a number of 'sections' in an executable program.
- These sections are used to provide instruction to the binary file and allowing inspection. Important function sections include the **Global Offset Table** (GOT), which stores **addresses of system functions**, the **Procedure Linking Table** (PLT), which stores indirect links to the GOT, .init/.fini, for internal initialization and shutdown, .ctors/.dtors, for constructors and destructors.
- The data sections are .rodata, for read only data, .data for initialized data, and .bss for uninitialized data.
- Partial list of the ELF sections are organized as follows (from low to high):

1. .init - Startup
2. .text - String
3. .fini - Shutdown
4. .rodata - Read Only
5. .data - Initialized Data
6. .tdata - Initialized Thread Data
7. .tbss - Uninitialized Thread Data
8. .ctors - Constructors
9. .dtors - Destructors
10. .got - Global Offset Table
11. .bss - Uninitialized Data

- You can use the **readelf** or **objdump** program against the object or executable files in order to view the sections.
- In the following Figure, two views of an ELF file are shown: the linking view and the execution view.

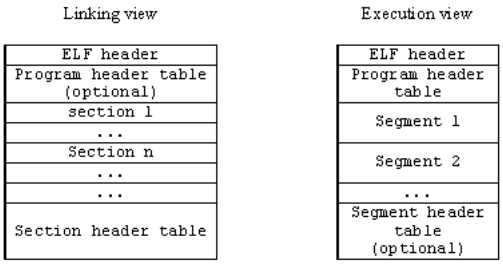

Figure w.4: Simplified object file format: linking view and execution view.

- Keep in mind that the full format of the ELF contains many more items. As explained previously, the linking view, which is used when the program or library is linked, deals with sections within an object file.
- Sections contain the bulk of the object file information: data, instructions, relocation information, symbols, debugging information, etc.
- The execution view, which is used when the program runs, deals with segments. Segments are a way of grouping related sections.
- For example, the text segment groups executable code, the data segment groups the program data, and the dynamic segment groups information relevant to dynamic loading.
- Each segment consists of one or more sections. A process image is created by loading and interpreting segments.
- The operating system logically copies a file's segment to a virtual memory segment according to the information provided in the program header table. The OS can also use segments to create a shared memory resource.
- At link time, the program or library is built by merging together sections with similar attributes into segments.
- Typically, all the executable and read-only data sections are combined into a single text segment, while the data and BSS are combined into the data segment.
- These segments are normally called load segments, because they need to be loaded in memory at process creation. Other sections such as symbol information and debugging sections are merged into other, non-load segments.

W.7 PROCESS LOADING

- In Linux processes loaded from a file system (using either the execve() or spawn() system calls) are in ELF format.
- If the file system is on a block-oriented device, the code and data are loaded into main memory.
- If the file system is memory mapped (e.g. ROM/Flash image), the code needn't be loaded into RAM, but may be executed in place.
- This approach makes all RAM available for data and stack, leaving the code in ROM or Flash. In all cases, if the same process is loaded more than once, its code will be shared.
- Before we can run an executable, firstly we have to load it into memory.
- This is done by the loader, which is generally part of the operating system. The loader does the following things (from other things):

1. Memory and access validation - Firstly, the OS system kernel reads in the program file's header information and does the validation for type, access permissions, memory requirement and its ability to run its instructions. It confirms that file is an executable image and calculates memory requirements.
2. Process setup includes:

   i. Allocates primary memory for the program's execution.
   ii. Copies address space from secondary to primary memory.
   iii. Copies the .text and .data sections from the executable into primary memory.
   iv. Copies program arguments (e.g., command line arguments) onto the stack.
   v. Initializes registers: sets the esp (stack pointer) to point to top of stack, clears the rest.
   vi. Jumps to start routine, which: copies main()'s arguments off of the stack, and jumps to main().

- Address space is memory space that contains program code, stack, and data segments or in other word, all data the program uses as it runs.
- The memory layout, consists of three segments (**text**, **data**, and **stack**), in simplified form is shown in Figure w.5.
- The dynamic data segment is also referred to as the **heap**, the place dynamically allocated memory (such as from malloc() and new) comes from. Dynamically allocated memory is memory allocated at run time instead of compile/link time.
- This organization enables any division of the dynamically allocated memory between the heap (explicitly) and the stack (implicitly). This explains why the stack grows downward and heap grows upward.

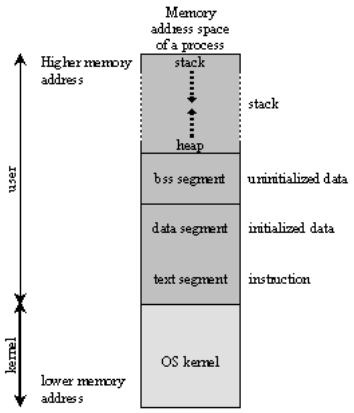

Figure w.4: Process memory layout

W.8 RUNTIME DATA STRUCTURE – From Sections to Segments

- A process is a running program. This means that the operating system has loaded the executable file for the program into memory, has arranged it to have access to its command-line arguments and environment variables, and has started it running.
- Typically a process has **5 different areas of memory allocated** to it as listed in Table w.5 (refer to Figure w.4):

Segment	Description
Code - **text** segment	Often referred to as the **text segment**, this is the area in which the executable instructions reside. For example, Linux/Unix arranges things so that multiple running instances of the same program share their code if possible. Only one copy of the instructions for the same program resides in memory at any time. The portion of the executable file containing the text segment is the **text section**.
Initialized data – **data** segment	Statically allocated and global data that are **initialized with nonzero values** live in the **data segment**. Each process running the same program has its own data segment. The portion of the executable file containing the data segment is the data section.
Uninitialized data – **bss** segment	**BSS** stands for '**Block Started by Symbol**'. Global and statically allocated data that **initialized to zero by default** are kept in what is called the BSS area of the process. Each process running the same program has its own BSS area. When running, the BSS data are placed in the data segment. In the executable file, they are stored in the **BSS section**. For Linux/Unix the format of an executable, only variables that are initialized to a nonzero value occupy space in the executable's disk file.
Heap	The heap is where dynamic memory (obtained by malloc(), calloc(), realloc() and *new* for C++) comes from. Everything on a heap is anonymous, thus you can only access parts of it through a pointer. As memory is allocated on the heap, the process's address space grows. Although it is possible to give memory back to the system and shrink a process's address space, this is almost never done because it will be allocated to other process again. Freed memory (free() and *delete*) goes back to the heap, creating what is called holes. It is typical for the heap to **grow upward**. This means that successive items that are added to the heap are added at addresses that are numerically greater than previous items. It is also typical for the heap to start immediately after the BSS area of the data segment. The end of the heap is marked by a pointer known as the **break**. You cannot reference past the break. You can, however, move the break pointer (via brk() and sbrk() system calls) to a new position to increase the amount of heap memory available.
Stack	The stack segment is where **local (automatic) variables are allocated**. In C program, local variables are all variables declared inside the opening left curly brace of a function body including the main() or other left curly brace that aren't defined as static. The data is popped up or pushed into the stack following the **Last In First Out** (LIFO) rule. The stack holds local variables, temporary information, function parameters, return address and the like. When a function is called, a **stack frame** (or a procedure activation record) is created and PUSHed onto the top of the stack. This stack frame contains information such as the address from which the function was called and where to jump back to when the function is finished (**return address**), parameters, local variables, and any other information needed by the invoked function. The order of the information may vary by system and compiler. When a function returns, the stack frame is POPped from the stack. Typically the **stack grows downward**, meaning that items deeper in the call chain are at numerically lower addresses and toward the heap.

Table w.5

- When a program is running, the initialized data, BSS and heap areas are usually placed into a single contiguous area called a data segment.
- The stack segment and code segment are separate from the data segment and from each other as illustrated in Figure w.4.
- Although it is theoretically possible for the stack and heap to grow into each other, the operating system prevents that event.
- The relationship among the different sections/segments is summarized in Table w.6, executable program segments and their locations.

Executable file section (disk file)	Address space segment	Program memory segment
`.text`	Text	Code
`.data`	Data	Initialized data
`.bss`	Data	BSS
-	Data	Heap
-	Stack	Stack

Table w.6

W.9 THE PROCESS (IMAGE)

- The diagram below shows the memory layout of a typical C's process. The process load segments (corresponding to "text" and "data" in the diagram) at the process's base address.
- The main stack is located just below and grows downwards. Any additional threads or function calls that are created will have their own stacks, located below the main stack.
- Each of the stack frames is separated by a guard page to detect stack overflows among stacks frame. The heap is located above the process and grows upwards.
- In the middle of the process's address space, there is a region is reserved for shared objects. When a new process is created, the process manager first maps the two segments from the executable into memory.
- It then decodes the program's ELF header. If the program header indicates that the executable was linked against a shared library, the process manager will extract the name of the dynamic interpreter from the program header.
- The dynamic interpreter points to a shared library that contains the runtime linker code. The process manager will load this shared library in memory and will then pass control to the runtime linker code in this library.

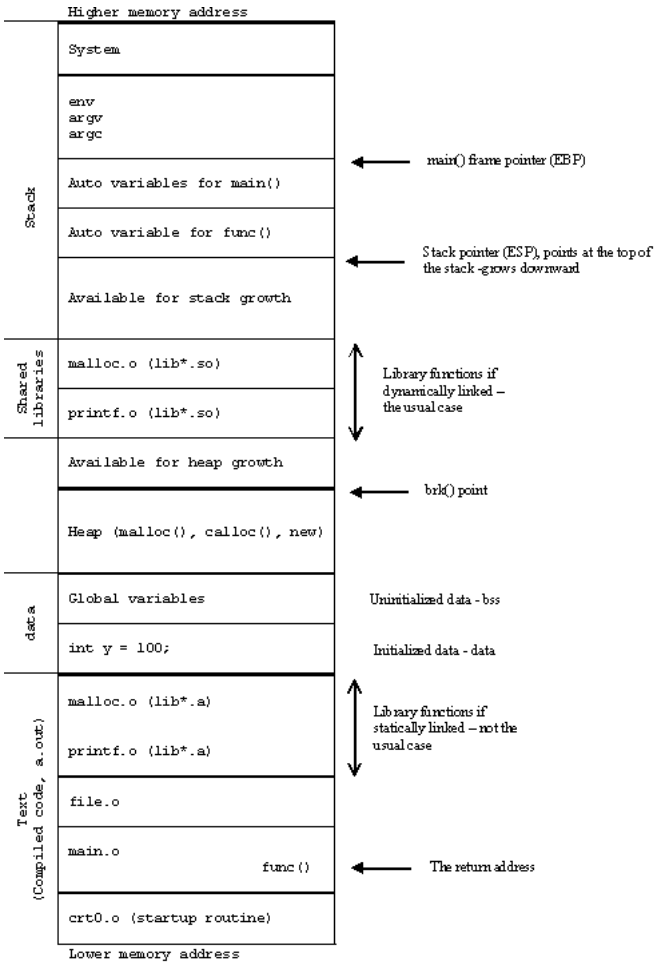

Figure w.5: C's process memory layout on an x86.

W.10 RUNTIME LINKER AND SHARED LIBRARY LOADING

- The runtime linker is invoked when a program that was linked against a shared object is started or when a program requests that a shared object be dynamically loaded.
- So the resolution of the symbols can be done at one of the following time:

  1. **Load-time dynamic linking** – the application program is read from the disk (disk file) into memory and unresolved references are located. The load time loader finds all necessary external symbols and alters all references to each symbol (all previously zeroed) to memory references relative to the beginning of the program.
  2. **Run-time dynamic linking** – the application program is read from disk (disk file) into memory and unresolved references are left as invalid (typically zero). The first access of an invalid, unresolved, reference results in a software trap. The run-time dynamic linker determines why this trap occurred and seeks the necessary external symbol. Only this symbol is loaded into memory and linked into the calling program.

- The runtime linker is contained within the C runtime library. The runtime linker performs several tasks when loading a shared library (.so file).
- The dynamic section provides information to the linker about other libraries that this library was linked against.
- It also gives information about the relocations that need to be applied and the external symbols that need to be resolved. The runtime linker will first load any other required shared libraries (which may themselves reference other shared libraries).
- It will then process the relocations for each library. Some of these relocations are local to the library, while others require the runtime linker to resolve a global symbol.
- In the latter case, the runtime linker will search through the list of libraries for this symbol. In ELF files, hash tables are used for the symbol lookup, so they're very fast.
- Once all relocations have been applied, any initialization functions that have been registered in the shared library's init section are called. This is used in some implementations of C++ to call global constructors.

W.11 SYMBOL NAME RESOLUTION

- When the runtime linker loads a shared library, the symbols within that library have to be resolved. Here, the order and the scope of the symbol resolution are important.
- If a shared library calls a function that happens to exist by the same name in several libraries that the program has loaded, the order in which these libraries are searched for this symbol is critical. This is why the OS defines several options that can be used when loading libraries.
- All the objects (executables and libraries) that have global scope are stored on an internal list (the global list).
- Any global-scope object, by default, makes available all of its symbols to any shared library that gets loaded.
- The global list initially contains the executable and any libraries that are loaded at the program's startup.

W.12 DYNAMIC ADDRESS TRANSLATION

- In the view of the memory management, modern OS with multitasking, normally implement dynamic relocation instead of static.
- All the program layout in the address space is virtually same. This dynamic relocation (in processor term it is called dynamic address translation) provides the illusion that:

    1. Each process can use addresses starting at 0, even if other processes are running, or even if the same program is running more than one time.
    2. Address spaces are protected.
    3. Can fool process further into thinking it has memory that's much larger than available physical memory (virtual memory).

- In dynamic relocation the address changed dynamically during every reference. **Virtual address** is generated by a process (also called **logical address**) and the **physical address** is the actual address in physical memory at the run-time.
- The address translation normally done by **Memory Management Unit** (MMU) that incorporated in the processor itself.
- Virtual addresses are relative to the process. Each process believes that its virtual addresses start from 0. The process does not even know where it is located in physical memory; the code executes entirely in terms of virtual addresses.
- MMU can refuse to translate virtual addresses that are outside the range of memory for the process for example by generating the segmentation faults. This provides the protection for each process.
- During translation, one can even move parts of the address space of a process between disk and memory as needed (normally called swapping or paging).
- This allows the virtual address space of the process to be much larger than the physical memory available to it.
- Graphically, this dynamic relocation for a process is shown in Figure w.6.

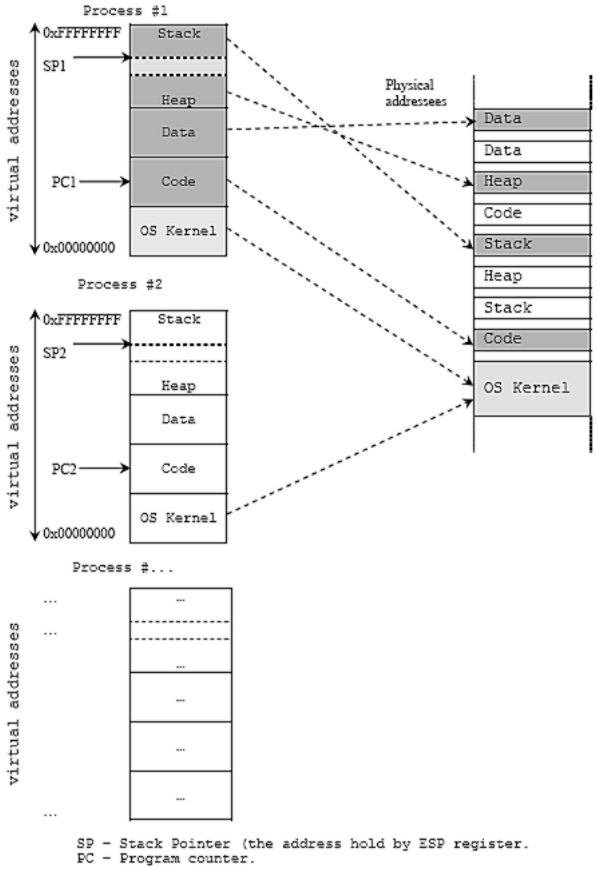

Figure w.6: Physical and virtual address: Address translation

- More complete related information can be found at Tenouk's buffer overflow Tutorial that include the stack construction and destruction for function call.
- You may also want to explore the Windows .NET Framework from the system perspective where the executable is called **assembly** with all new terms and features.

Further related reading:

1. Check the best selling C / C++, Linux and Open Source books at Amazon.com.
2. To view Windows the executable file content, you can use **dumpbin** tool that comes with Microsoft Visual Studio or more powerful one is a free PEBrowse utility.

3. For Linux/Unix/Fedora you can use **readelf** or other tools that can be found Linux gnu gcc, g++, gdb and gas 1 or Linux gnu gcc, g++, gdb and gas 2.
4. Windows implementation of processes, threads and synchronization using C can be found Win32 processes and threads tutorials.
5. Windows Dynamic-Link Library, DLL story and program examples can be found Win32 DLL tutorials.
6. Windows Services story can be found Windows Services tutorials.

|< The main() and command line arguments | Main | C Memory Allocation and De-allocation Functions >| Site Index | Download |