Introduction to Compilers and Language Design

Hardcover ISBN: 978-0-359-13804-3
Paperback ISBN: 978-0-359-14283-5
First edition.

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at `http://compilerbook.org`

Revision Date: October 9, 2018

Chapter 9 – Memory Organization

9.1 Introduction

Before digging into the translation of intermediate code to assembly language, we must discuss how the internal memory of a running program is laid out. Although a process is free to use memory in any way that it likes, a convention has developed that divides the areas of a program into logical segments, each with a different internal management strategy.

9.2 Logical Segmentation

A conventional program sees memory as a linear sequence of words, each with a numeric address starting at zero, and increasing up to some large number (e.g. 4GB on a 32-bit processor.)

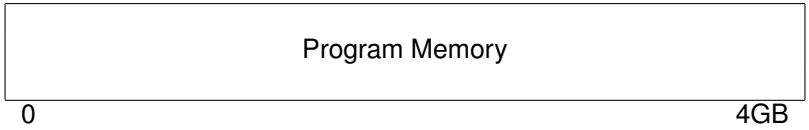


Figure 9.1: Flat Memory Model

In principle, the CPU is free to use memory in any way it sees fit. Code and data could be scattered and intermixed across memory in any order that is convenient. It is even technically possible for a CPU to modify the memory containing its code while it is running. It goes without saying that programming in this fashion would be complex, confusing, and difficult to debug.

Instead, program memory is commonly laid out by separating it into **logical segments**. Each segment is a sequential address range, dedicated to a particular purpose within the program. The segments are typically laid out in this order:

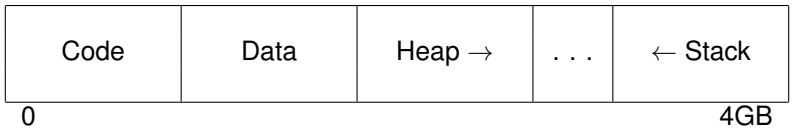


Figure 9.2: Logical Segments

- The **code segment** (also known as the **text segment**) contains the machine code of the program, corresponding to the bodies of functions in a C program.
- The **data segment** contains the global data of the program, corresponding to the global variables in a C program. The data segment may further be sub-divided into read-write data (variables) and read-only data (constants).
- The **heap segment** contains the heap, which is the area of memory that is managed dynamically at runtime by `malloc` and `free` in a C program, or `new` and `delete` in other languages. The top of the heap is historically known as the **break**.
- The **stack segment** contains the stack, which records the current execution state of the program as well as the local variables currently in use.

Typically, the heap grows "up" from lower addresses to higher addresses, while the stack grows "down" from higher to lower. In between the two segments is an invalid region of memory that is unused until overtaken by one segment or the other.

On a simple computer such as an embedded machine or microcontroller, logical segments are nothing more than an organizational convention: nothing stops the program from using memory improperly. If the heap grows too large, it can run into the stack segment (or vice versa), the program will crash (if you are lucky) or suffer silent data corruption (if you are unlucky.)

On a computer with an operating system that employs multiprogramming and memory protection, the situation is better. Each process running in the OS has its own private memory space, with the illusion of starting at address zero and extending to a high address. As a result, each process can access its own memory arbitrarily, but is prevented from accessing or modifying other processes. Within its own space, each process lays out its own code, data, heap, and stack segments.

In some operating systems, when a program is initially loaded into memory, permissions are set on each range of memory corresponding to its purpose: memory corresponding to each segment can be set appropri-

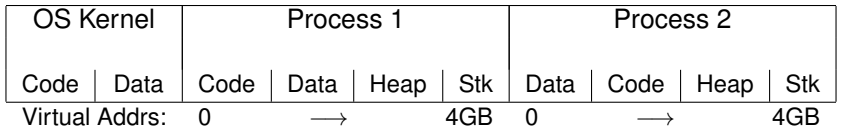


Figure 9.3: Multiprogrammed Memory Layout

ately: read-write for data, heap, and stack; read-only for constants; read-execute for code; and none for the unused area.

The permissions on the logical segments also protect a process from damaging itself in certain ways. For example, code cannot be modified at runtime because it is marked read/execute, while items on the heap cannot be executed, because they are marked read/write. (To be clear, this only prevents against accidents, not malice; a program can always ask the operating system to change the permissions on one of its own pages. For an example, look up the `mprotect` system call in Unix.)

If a process attempts to access memory in a prohibited way or attempts to access the unused area, a **page fault** occurs. This forces a transfer of control to the OS, which considers the process and the faulting address. If the access indicates a violation of the logical segmentation of the program, then the process is forcibly killed with the error message **segmentation fault**.[1]

Initially, a process is given a small amount of memory for the heap segment, which it manages internally to implement `malloc` and `free`. If this area is exhausted and the program still needs more, it must explicitly request it from the operating system. On traditional Unix systems, this is done with the `brk` system call, which requests that the heap segment be extended to a new address. If the OS agrees, then it will allocate new pages at the beginning of the invalid area, effectively extending the heap segment. If the OS does not agree, then `brk` will return an error code, causing `malloc` to return an error (indicated as a null pointer) which the program must handle.

The stack has a similar problem, in that it must be able to grow down. It is not so easy for a program to determine exactly when more stack is needed, because that happens whenever a new function is invoked, or new local variables are allocated. Instead, modern operating systems maintain a **guard page** at the top of the invalid area, adjacent to the current stack. When a process attempts to extend the stack into the invalid area, a page fault occurs, and control is transferred to the OS. If the OS sees that the faulting address is in the guard page, it can simply allocate more pages for the stack, set the page permissions appropriately, and move the guard page down to the new top of the invalid area.

Of course, there are limits on how big the heap and the stack may grow;

[1] And now you understand this mysterious Unix term!

every OS implements policies controlling how much memory any process or user may consume. If any of these policies are violated, the OS may decline to extend the memory of the process.

The idea of breaking a program into segments is so powerful and useful that it was common for many decades to have the concept implemented in hardware. (If you have taken a class in computer architecture and operating systems, you have probably studied this in some detail.) The basic idea is that the CPU maintains a table of segments, recording the starting address and length, along with the permissions associated with each register. The operating system would typically set up a hardware segment to correspond to the logical organization just described.

Although hardware segmentation was widely used in operating systems through the 1980s, it has been largely replaced by paging, which was seen as simpler and more flexible. Processor vendors have responded by removing support for hardware segmentation in new designs. For example, every generation of the Intel X86 architecture from the 8086 up to the Pentium supported segmentation in 32-bit protected mode. The latest 64-bit architectures provide only paging facilities, and no segmentation. Logical segmentation continues as a useful way to organize programs in memory.

Let's continue by looking at each of the logical segments in more detail.

9.3 Heap Management

The heap contains memory that is managed dynamically at runtime. The OS does not control the internal organization of the heap, except to limit its total size. Instead, the internal structure of the heap is managed by the standard library or other runtime support software that is automatically linked into a program. In a C program, the function calls **malloc** and **free** allocate and release memory on the heap, respectively. In C++, **new** and **delete** have the same effect. Other languages manipulate the heap implicitly when objects and arrays are created and deleted.

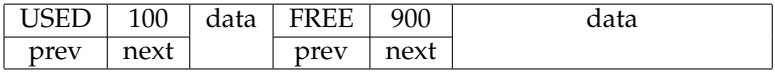
The simplest implementation of **malloc** and **free** is to treat the entire heap as one large linked list of memory regions. Each entry in the list records the state of the region (free or in use), the size of the region, and has pointers to the previous and next regions. Here's what that might look like in C:

```
struct chunk {
    enum { FREE, USED } state;
    int size;
    struct chunk *next;
    struct chunk *prev;
    char data[0];
};
```

(Note that we declared `data` as an array of length zero. This is a little trick that allows us to treat `data` as a variable length array, provided that the underlying memory is actually present.)

Under this scheme, the initial state of the heap is simply one entry in a linked list:

Suppose that the user calls `malloc(100)` to allocate 100 bytes of memory. `malloc` will see that the (single) chunk of memory is free, but much larger than the requested size. So, it will split it into one small chunk of 100 bytes and one larger chunk with the remainder. This is accomplished by simply writing a new function header into the data area after 100 bytes. This is and then connect them together into a linked list:



Once the list has been modified, `malloc` returns the address of the data element within the chunk, so that the user can access it directly. It doesn't return the linked list node itself, because the user doesn't need to know about the implementation details. If there is no chunk available that is large enough to satisfy the current request, then the process must ask the OS to extend the heap by calling `brk`.

When the user calls `free` on a chunk of memory, the state of the chunk in the linked list is marked `FREE`, and then merged with adjacent nodes, if they are also free.

(Incidentally, now you can see why it is dangerous for a program to modify memory carelessly outside a given memory chunk. Not only could it affect other chunks, but it could damage the linked list itself, resulting in wild behavior on the next `malloc` or `free`!)

If the program always frees memory in the opposite order that it was allocated, then the heap will be nicely split into allocated and free memory. However, that isn't what happens in practice: memory can be allocated and freed in any order. Over time, the heap can degenerate into a mix of oddly sized chunks of allocated and freed memory. This is known as **memory fragmentation**.

Excessive fragmentation can result in waste: if there are many small chunks available, but none of them large enough to satisfy the current `malloc`, then the process has no choice but to extend the heap, leaving the small pieces unused. This increases pressure on total virtual memory consumption in the operating system.

In a language like C, memory chunks cannot be moved while in use, and so fragmentation cannot be fixed after it has already occurred. How-

ever, the memory allocator has some limited ability to avoid fragmentation by choosing the location of new allocations with care. Some simple strategies are easy to imagine and have been studied extensively:

- **Best Fit.** On each allocation, search the entire linked list and find the *smallest* free chunk that is larger than the request. This tends to leave large spaces available, but generates tiny leftover free fragments that are too small to be used.
- **Worst Fit.** On each allocation, search the entire linked list and find the *largest* free chunk that is larger than the request. Somewhat counterintuitively, this method tends to reduce fragmentation by avoiding the creation of tiny unusable fragments.
- **First Fit.** On each allocation, search the linked list from the beginning, and find the *first* fragment (large or small) that satisfies the request. This performs less work than Best Fit or Worst Fit, but performs an increasing amount of work as the linked list increases in size.
- **Next Fit.** On each allocation, search the linked list from the last examined location, and find the *next* fragment (large or small) that satisfies the request. This minimizes the amount of work done on each allocation, while distributing allocations throughout the heap.

For general purpose allocators where one cannot make assumptions about application behavior, the conventional wisdom is that Next Fit results in good performance with an acceptable level of fragmentation.

9.4 Stack Management

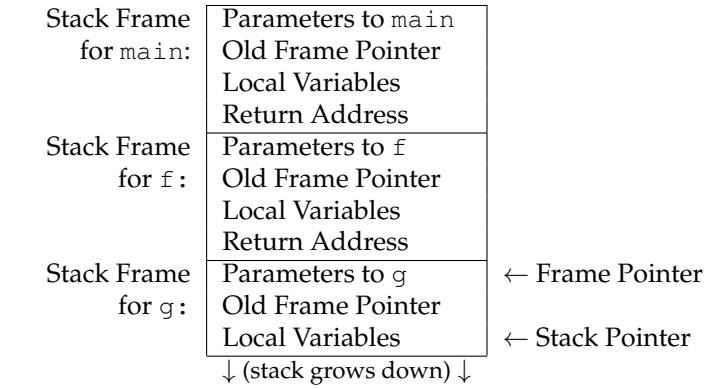
The **stack** is used to record the current state of the running program. Most CPUs have a specialized register – the **stack pointer** – which stores the address where the next item will be pushed or popped. Because the stack grows down from the top of memory, there is a confusing convention: pushing an item on the stack causes the stack pointer to move to a lower numbered address, while popping an item off the stack causes the stack pointer to move to a higher address. The "top" of the stack is actually at the lowest address!

Each invocation of a function occupies a range of memory in the stack, known as a **stack frame**. The stack frame contains the parameters and the local variables used by that function. When a function is called, a new stack frame is pushed; when the function returns, the stack frame is popped, and execution continues in the caller's stack frame.

Another specialized register known as the **frame pointer** (or sometimes **base pointer**) indicates the beginning of the current frame. Code

within a function relies upon the frame pointer to identify the location of the current parameters and local variables.

For example, suppose that the `main` function calls function `f`, and then `f` calls `g`. If we stop the program in the middle of executing `g`, the stack would look like this:



The order and details of the elements in an stack frame differ somewhat between CPU architectures and operating systems. As long as both the caller and the callee agree on what goes in the stack frame, then any function may call another, even if they were written in different languages, or built by different compilers.

The agreement on the contents of the activation record is known as a **calling convention**. This is typically written out in a detailed technical document that is used by the designers of compilers, operating systems, and libraries to ensure that code is mutually interoperable.

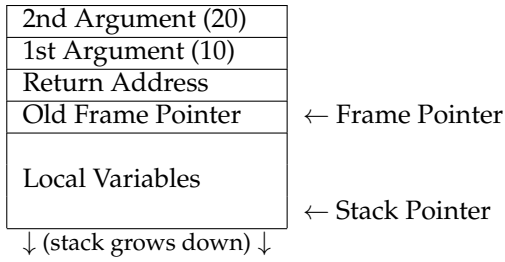
There are two broad categories of calling conventions, with many opportunities for variation in between. One is to put the arguments to a function call on the stack, and the other is to place them in registers.

9.4.1 Stack Calling Convention

The conventional approach to calling a function is to push the arguments to that function on the stack (in reverse order), and then to jump to the address of the function, leaving behind a return address on the stack. Most CPUs have a specialized `CALL` instruction for this purpose. For example, the assembly code to call `f(10,20)` could be as simple as this:

```
PUSH $20
PUSH $10
CALL f
```

When `f` begins executing, it saves the old frame pointer currently in effect and makes space for its own local variables. As a result, the stack frame for `f(10,20)` looks like this:



To access its arguments or local variables, `f` must load them from memory relative to the frame pointer. As you can see, the function arguments are found at fixed positions *above* the frame pointer, while local variables are found *below* the frame pointer. [2]

9.4.2 Register Calling Convention

An alternate approach to calling a function is to put the arguments into registers, and then call the function. For example, let us suppose that our calling convention indicates that registers %R10, %R11, etc are to be used for arguments. Under this calling convention, the assembly code to invoke `f(10,20)` might look like this:

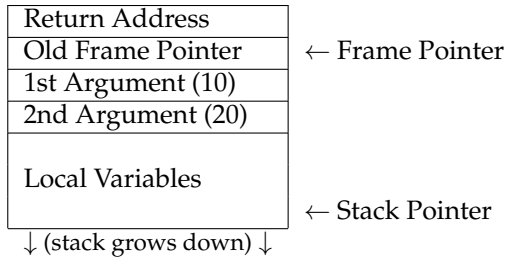
```
MOVE $10 -> %R10
MOVE $20 -> %R11
CALL f
```

When `f` begins executing, it still must save the old frame pointer and make room for local variables. It doesn't have to load arguments from the stack; it simply expects the values in %R10 and %R11 and can compute on them right away. This could confer a significant speed advantage by avoiding memory accesses.

But, what if `f` is a complex function that needs to invoke other functions? It will still need to save the current values of the argument registers, in order to free them up for its own use.

To allow for this possibility, the stack frame for `f` must leave space for the arguments, in case they must be saved. The calling convention must define the location of the arguments, and they are typically stored *below* the return address and old frame pointer, like this:

[2] The arguments are pushed in reverse order in order to allow the possibility of a variable number of arguments. Argument 1 is always two words above the frame pointer, argument 2 is always three words above, and so on.



What happens if the function has more arguments then there are registers set aside for arguments? In this case, the additional arguments are pushed on to the stack, as in the stack calling convention.

In the big picture, the choice between stack and register calling conventions doesn't matter much, except that all parties must agree on the details of the convention. The register calling convention has the slight advantage that a **leaf function** (a function that does not call other functions) can compute a simple result without accessing memory. Typically, the register calling convention is used on architectures that have a large number of registers that might otherwise go unused.

It is possible to mix the conventions in a single program, as long as both caller and callee are informed of the distinction. For example, the Microsoft X86 compilers allow keywords in function prototypes to select a convention: `cdecl` selects the stack calling convention, while `fastcall` uses registers for the first two arguments.

9.5 Locating Data

For each kind of data in a program, there must be an unambiguous method of locating that data in memory. The compiler must generate an **address computation** using the basic information available about the symbol. The computation varies with the storage class of the data:

- **Global data** has the easiest address computation. In fact, the compiler doesn't usually compute global addresses, but rather passes the *name* of each global symbol to the assembler, which then selects the address computation. In the simplest case, the assembler will generate an **absolute address** giving the exact location of the data in program memory.

  However, the simple approach isn't necessarily efficient, because an absolute address is a full word (e.g. 64 bits), the same size as an instruction in memory. This means that the assembler must use several instructions (RISC) or multi-word instructions (CISC) to load the address into a register. Assuming that most programs don't use the entire address space, it isn't usually necessary to use the entire word.

  An alternative is to use a **base-relative address** that consists of a base address given by a register plus a fixed offset given by the assembler.

For example, global data addresses could be given by a register indicating the beginning of the data segment, plus a fixed offset, while a function address could be given by a register indicating the beginning of the code segment plus a fixed offset. Such an approach can be used in dynamically loaded libraries, when the location of the library is not fixed in advance, but the location of a function within the library is known.

Yet another approach is to use a **PC-relative address** in which the exact distance in bytes between the referring instruction and the target data is computed, and then encoded into the instruction. This works as long as the relative distance is small enough (e.g. 16 bits) to fit into the address field of the instruction. This task is performed by the assembler is usually invisible to the programmer.

- **Local data** works differently. Because local variables are stored on the stack, a given local variable does not necessarily occupy the same absolute address each time it is used. If a function is called recursively, there may be multiple instances of a given local variable in use simultaneously! For this reason, local variables are always specified as an offset relative to the current frame pointer. (The offset may be positive or negative, depending on the function calling convention.) Function parameters are just a special case of local variables: a parameter's position on the stack is given precisely by its ordinal position in the parameters.
- **Heap data** can only be accessed by way of pointers that are stored as global or local variables. To access data on the heap, the compiler must generate an address computation for the pointer itself, then dereference the pointer to reach the item on the heap.

So far, we have only considered atomic data types that are easily stored in a single word of memory: booleans, integers, floats, and so forth. However, any of the more complex data types can be placed in any of the three storage classes, and require some additional handling.

An array can be stored in global, local, or heap memory, and the beginning of the array is found by one of the methods above. An element in the array is found by multiplying the index by the size of the items in the array, and adding that to the address of the array itself:

```
address(a[i]) = address(a) + sizeof(type) * i
```

The more interesting question is how to deal with the length of the array itself. In an unsafe language like C, the simple approach is to simply do nothing: if the program happens to run off the end of the array, the compiler will happily compute an address outside the array bounds, and chaos results. For some applications where performance is paramount, the simplicity of this approach trumps any increase in safety.

A safer approach is to store the length of the array at the base address of the array itself. Then, the compiler may generate code that checks the actual index against the array bounds before generating the address. This prevents any sort of runtime accident by the programmer. However, the downside is performance. Every time the programmer mentions `a[i]`, the resulting code must contain this:

1. Compute address of array `a`.
2. Load length of `a` into a register.
3. Compare array index `i` to register.
4. If `i` is outside of array bounds, raise an exception.
5. Otherwise, compute address of `a[i]` and continue.

This pattern is so common that some computer architectures provide dedicated support for array bounds checking. The Intel X86 architecture, (which we will examine in detail in the next chapter) provides a unique `BOUND` instruction, whose only purpose is to compare a value against two array bound values, and then raise a unique "Array Bounds Exception" it it falls outside.

Structures have similar considerations. In memory, a structure is very much like an array, except that it can contain items of irregular size. To access an item within a structure, the compiler must generate an address computation of the beginning of the structure, and then add an offset corresponding to the name of item (known as the **structure tag**) within the structure. Of course, it is not necessary to perform bounds checking since the offsets are fixed at compile time.

For complex nested data structures, the address computation necessary to find an individual element can become quite complicated. For example, consider this bit of code to represent a deck of cards:

```
struct card {
    int suit;
    int rank;
};

struct deck {
    int is_shuffled;
    struct card cards[52];
};

struct deck d;

d.card[10].rank = 10;
```

To compute `d.card[10].rank`, the compiler must first generate an address computation for `d`, depending on whether it is a local or global variable. From there, the offset of `card` is added, then the offset of the tenth item, then the offset of `rank` within the card. The complete address computation is:

```
address(d.card[10].rank) =
    address(d) + offset(card) + sizeof(card)*10
                                    + offset(rank)
```

9.6 Program Loading

Before a program begins executing in memory, it first exists as a file on disk, and there must be a convention for loading it into memory. There are a variety of ways of **executable formats** for organizing a program on disk, ranging from very simple to very complex. Here are a few examples to give you the idea.

The simplest computer systems simply store an executable as a **binary blob** on disk. The program code, data, and initial state of the heap and stack are simply dumped into one file without distinction. To run the program, the OS must simply load the contents of the file into memory, and then jump to the first location of the program to begin execution.

This approach is about as simple as one can imagine. It does work, but it has several limitations. One is that the format wastes space on uninitialized data. For example, if the program declares a large global array where each element has the value zero, then every single zero in that array will be stored in the file. Another is that the OS has no insight into how the program intends to use memory, so it is unable to set permissions on each logical segment, as discussed above. Yet another is that the binary blob has no identifying information to show that it is an executable.

However, the binary blob approach is still occasionally used in places where programs are small and simplicity is paramount. For example, the very first boot stage of PC operating system reads in a single sector from the boot hard disk containing a binary blob, which then carries out the second stage of booting. Embedded systems often have very small programs measured in a few kilobytes, and rely on binary blobs.

An improved approach used in classic Unix systems for many years is the **a.out** executable format. [3] There are many slight variations on the format, but they all share the same basic structure. The executable file consists of a short header structure, followed by the text, initialized data, and symbol table:

[3] The first Unix assembler sent its output to a file named `a.out` by default. In the absence of any other name for the format, the name stuck.

The header structure itself is just a few bytes that allow the operating system to interpret the rest of the file:

Magic Number
Text Section Size
Data Section Size
BSS Size
Symbol Table Size
Entry Point

The **magic number** is a unique integer that clearly defines the file as an executable: if the file does not begin with this magic number, the OS will not even attempt to execute it. Different magic numbers are defined for executables, unlinked object files, and shared libraries. The **text size** field indicates the number of bytes in the text section that follows the header. The **data size** field indicates the amount of *initialized* data that appears in the file, while the **BSS size** field indicates the amount of *uninitialized data*. [4]

The uninitialized data need not be stored in the file. Instead it is simply allocated in memory as part of the data segment when the program is loaded. The **symbol table** in the executable lists each of the variable and function names used in the program along with their locations in the code and data segment; this permits a debugger to interpret the meaning of addresses. Finally, the **entry point** gives the address of the starting point of the program (typically `main`) in the text segment. This allows the starting point to be something other than the first address in the program.

The `a.out` format is a big improvement over a binary blob, and is still supported and usable today in many operating systems. However, it isn't quite powerful enough to support many of the features needed by modern languages, particularly dynamically loaded libraries.

The **Extensible Linking Format (ELF)** is widely used today across many operating systems to represent executables, object files, and shared libraries. Like `a.out`, an ELF file has multiple sections representing code, data, and bss, but it can have an arbitrary number of additional sections for debugging data, initialization and finalization code, metadata about the tools used, and so forth. The number of *sections* in the file outnumbers the *segments* in memory, and so a **section table** in the ELF file indicates how multiple sections are to be mapped into a single segment.

[4] BSS stands for "Block Started by Symbol" and first appeared in an assembler for IBM 704 in the 1950s.

File Header
Program Header
Code Section
Data Section
Read-Only Section
. . .
Section Header

9.7 Further Reading

1. Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau, "Operating Systems: Three Easy Pieces", Arpaci-Dusseau Books, 2015.
http://www.ostep.org
Operating systems is usually the course in which memory management is covered in great detail. If you need a refresher on memory allocators (or anything else in operating systems), check out this online textbook.

2. John R. Levine, "Linkers and Loaders", Morgan Kaufmann, 1999.
This book provides a detailed look at linkers and loaders, which is an often-overlooked topic that falls in cracks between compilers and operating systems. A solid understanding of linking is necessarily to create and use libraries effectively.

3. Paul R. Wilson, "Uniprocessor Garbage Collection Techniques", Lecture Notes in Computer Science, volume 637, 1992.
https://link.springer.com/chapter/10.1007/BFb0017182
This widely-read article gives an accessible overview of the key techniques of garbage collection, which are an essential component of the runtime of modern dynamic languages.