Memory Management for System Programmers

Baris Simsek, 2005
http://www.enderunix.org/simsek/

ABOUT THIS DOCUMENT

You can find latest versions of this document at http://www.enderunix.org/simsek/

This document is a complete reference about computer memories and memory management. It is written for system programmers.

WHAT IS MEMORY

Memory is the primary data storage area for computers.

We call the basic memory unit **a bit**. A bit may contain two different values: either 0 or 1. Why do computers use binary arithmetic? It is the most reliable and efficient way to express data. Because, the digital information can be stored as voltage. We have to distinguish each different value. If we have more values (such as decimal system), it causes less separation between adjacent values. But with two values(binary system: 0 and 1) different values will be distinguished with maximum distance. Like a rule: To get the most distance we must use two points. The distance will be distance of the rule.

How can number 1453 be represented in the decimal system?

$1453 = 1x10^3 + 4x10^2 + 5x10^1 + 3x10^0$

Base 2	Base 10	Base 16
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F
10000	16	10

$(A)_{10} = K_0x10^o + K_1x10^1 + ... + K_ax10^a$

Numbers 0 through 16 are shown above. Notice that, we have 2 different numbers under base 2 (binary). And 10 numbers under base 10(decimal) and so 16 numbers under base 16(hexadecimal). The value 10 is shown in all columns. It means 2(1x2 + 0x1) in binary format and 16(1x16 + 0x1) in hexal.

The digital information stored in memory units shows its *state.* This state is not only a function of the inputs but it is also a function of a current state.

f(x) = f(inputs, f(x-1))

To create a 1-bit memory, we need a circuit that somehow "remembers" previous input values. Latches are basic memory units which can remember previous values. Following circuit illustrates S-R latch(Figure-1).

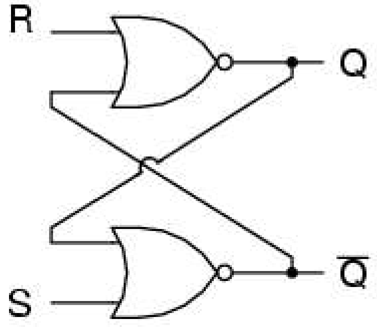
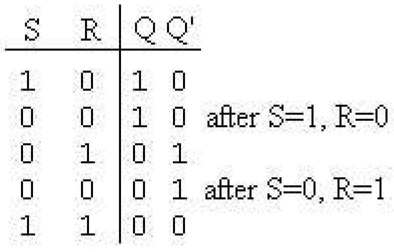


Figure-1: S-R Latch logic diagram

To create an S-R latch, we can wire two NOR gates in such a way that the output of one **feeds back** to the input of another. Remember, S comes from *set*, and R comes from *reset.* When both the set and reset inputs of the flip-flop set 1, both Q and Q' outputs go to 0. This condition violates the fact that both outputs are complements of each other. In normal operation this condition must be avoided by making sure that both inputs won't be same simultaneously.

We assume S=1 and R=0 as the initial state. Q' will be 0 and Q will be 1. When S goes to 0, Q is 1 and Q' goes to 0. As result Q will be 1. So nothing is changed. If clock sets R to 1, Q will be 0 and Q' will be 1. If R goes to 0 nothing will be changed. That means S clock sets up circuit and R clock resets.



S	R	Q	Q'	
1	0	1	0	
0	0	1	0	after S=1, R=0
0	1	0	1	
0	0	0	1	after S=0, R=1
1	1	0	0	

Figure-2: S-R Latch truth table

Between two consecutive clocks, circuit stores Q and Q'. That is the most simple memory. It is realy so difficult to control asynchronous circuits. For that we use clock cycles to trigger circuit. By this way we can make timing.

Imagine a 2-dimensions latch array with 1024x8. 8 columns and 1024 rows. Each cell has a RS latch and can store 1 bit. So that is 1 Kbytes of memory. Remember program counter(PC) which indicates address of next instruction. When the CPU fetch an instruction from memory, it increases PC to point to the next instruction. PC selects corresponding memory cell. This structure is illustrated in Figure-2.

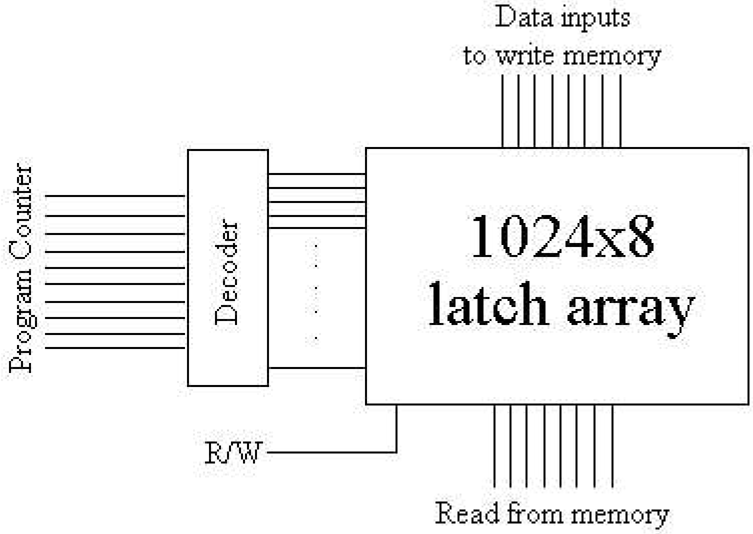


Figure-3: Simple memory design

A memory consist of a number of **cells** which can store some number of bits. The memory is just an byte array. Each cell has a number identify itself, called its **address**. Programs refers addresses to reach memory. Adjacent cells have consecutive addresses. If memory has m cells, the cells will have addresses 0 to m-1. If CPU supports n bit, it can refer addresses from 0 to 2^n – 1. For example, Intel Pentium II is a 32-bit CPU and can addresses 4 GBytes of memory. Each cell stores an integer. An integer is n-bit number. It is 32 bit(4 bytes) if you have a Intel P2.

Maximum addressable memory size = n * 2^n

In recent years, nearly all manufactures have standardized on an 8-bit cell which is called **byte**. Bytes are grouped into **words**. An 32-bit CPU has 4-bytes/word. This CPU has has 32-bit registers can holds 32-bits at a time. For that reason, registers that access to memory are also 32 bits. So it can point maximum to 11111111111111111111111111111111 in binary format(0xFFFFFFFF in hexal form). That is same with the 2^n – 1.

VIRTUAL MEMORY

In the early years, computer memories were small and more expensive. Programmers were using a total memory size of only 4096 18-bit words for the both user programs and operating system in PDP-1. So, the programmer had to fit his program in this small memory. Nowadays, computers have some gigabytes of memory but the modern programs need much more memory. To solve this problem, operating systems use secondary memories such as disk as main memory.

In the first technique, the programmer divided the program up into a number of pieces called **overlays**. At the start of the program, first overlay was loaded into memory. When it finished, loads next overlay. Programmers must manage overlays between memory and disk. He was responsible to find it from disk and load it to memory. It was diffucult for programmers.

In 1961, a group of researchers from Manchester established automatic overlay management system called **virtual memory**.

Virtual memory is organized into "**pages**". A page is the memory unit typically a few Kbytes in size. It is mostly 4-Kbytes. You can learn page size by typing `pagesize` command. When a program references to an address on a page not present in main memory, a **page fault** occurs. After a page fault, the operating system seeks for the corresponding page on the disk and loads it onto main memory by using a page replacement algorithm such as LRU. We can start a program when none of the program is in main memory. When the CPU tries to fetch the first instruction of the program, it gets a page fault, because the memory doesn't contain any piece of the program in the main memory. This method is called **demand paging**.

If a process in main memory has low priority or is sleeping, that means it won't run soon. In this case, the process can be backed up on disk by the operating system. This process is **swapped out**. The swap space is using for holding memory data.

Processes use virtual addresses for transparency. They don't know about physical memory. CPU has a unit called **Memory Management Unit** which is responsible for operating virtual memory. When a process makes a reference to a page that isn't in main memory, the MMU generates a page fault. The kernel catchs it and decides whether the reference is valid or not. If invalid, the kernel sends signal "segmentation violation" to the process. If valid, the kernel retrieves the page process referenced from the disk.

MEMORY LAYOUT FOR A PROCESS

We have discussed about the system memory above. In this section, we will talk about memory layout of an individual process.

Memory is an array of words. But it is not functional with this simple structure. Operating systems divide it into some pieces each one has its custom behaviour. Fot example, the kernel may protect a part of process memory against write and execute. In the process memory, each memory section which has different behaviours is called **segment**.

When a program is loaded into memory, it resides in memory like following figure:

```
+-------------------+-------------------+----------+-----------+-----------+
| Code segment(r+x) | Data segment(r+w) | BSS(r+w) | Heap(r+w) | Stack(r+w)|
+--------------------------------------------------------------------------+
```

Figure-5: Runtime Memory Layout

Code(Text) Segment:

This is the area in which the executable instructions reside. In the UNIX world, it is called as "*text segment*". It has execute permission. Some old architectures allows to code change itself. For that reason, the code segment had also the 'write' permission.

Consider we have a function named `func()`. addr, points somewhere in the code segment. Because of functions reside in code segment.

```
addr = &func;
```

Data Segment:

It contains initialized global variables declared by programmer. Global variables have a fixed area in memory where they will be defined at startup.

BSS:

It contains uninitialized global variables.

Heap:

It contains variables generated dynamicly at runtime. Data segment holds variables which we created at compile time. So it is fixed in size.

Often, the programmer needs to create variables at runtime. This is called **dynamic memory allocation**. Modern C libraries provide some functions to allocate area from heap like `malloc()`. `free()` destroys variables which have been dynamically allocated from heap space. For more details please see the following "Pointers" title.

Everything is unnamed in the heap. You cannot reach any variable directly by using its name. But you can reference indirectly using a pointer.

The end of the heap is marked by a pointer called "break". When a program reference past the break, it will break. When the heap manager needs more memory, it calls `brk()` and `sbrk()` system calls. The `brk()` and `sbrk()` functions are used to change the amount of memory allocated in a process's data segment. They do this by moving the location of the `break'. The break is the first address after the end of the process's uninitialized data segment (also known as the `BSS'). See Firgure 5.

```
NAME
     brk, sbrk -- change data segment size

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/types.h>
     #include <unistd.h>

     int
     brk(const void *addr);

     void *
     sbrk(intptr_t incr);
```

The `brk()` function sets the break to addr. The `sbrk()` function raises the break by incr bytes, thus allocating at least incr bytes of new memory in the data segment. If incr is negative, the break is lowered by incr bytes. The current value of the break may be determined by calling `sbrk(0)`.

Stack:

Stack is a data structure which is accessed in last-in first-out order. There are two operations for stacks:

To insert a new item, it must be **PUSH**ed and to retrieve item, it must **POP**ped. SP(stack pointer) is a CPU register which points to the top of the stack. Data resides from higher addresses to lower addresses in stack.

A function call is the typical usage of stack. What happens after a function call?

```
10] i = 4;
11] func(i);
12] k = 2;
13] ...
```

Remember: CPU always executes instruction which IP(instruction pointer) shows. In above code, IP is 10 before calling `func()`. It will be the address of func() when `func()` is called. And then IP will be 12 after `func()` exited. Before `func()` call, we must save address of the next code line. Because we'll return back after function exited and continue to execute program from next line. To store address of the next line, the program will use stack.

After this step the program PUSHes function parameters (in above code, 'i' is the parameter) to stack. And then local variables of the function. When the function ends, it POPs local variables and function parameters orderly. Thus there is only address of the next line(in above code, it is 12) in the stack. **return** is a CPU specific instruction. After return, the address will be POPped from stack and will be assigned to IP. Hence the program will continue to execution from line 12.

Following code illustrates segments.

`mem.c`

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

int uig;
int ig = 5;

int func()
{
        return 0;
}

int main()
{
        int local;
        int *ptr;

        ptr = (int *) malloc(sizeof(int));

        printf("An address from BSS: %p\n", &uig);
        printf("An address from Data segment: %p\n", &ig);
        printf("An address from Code segment: %p\n", &func);
        printf("An address from Stack segment: %p\n", &local);
        printf("An address from Heap: %p\n", ptr);
        printf("Another address from Stack: %p\n", &ptr);

        free(ptr);

        return 0;
}
```

```
# ./mem
An address from BSS: 0x804982c
An address from Data segment: 0x8049738
An address from Code segment: 0x8048520
An address from Stack segment: 0xbfbfec6c
An address from Heap: 0x804b030
Another address from Stack: 0xbfbfec68
```

Questions: Examine last line of output. It shows another address from stack.

1. Why &ptr is in the stack segment?
2. Why distance is 4 byte between local and &ptr?
3. Why &ptr is less than &local?

MEMORY LEAK

A memory leak is where allocated memory is not freed although it is never used again. There are two common types of heap problems:

1. Freeing or overwriting data that is still in use will cause "*memory corruption*".
2. Not freeing data which is no longer in use will cause "*memory leak*".

If the memory leak is in a loop, after a while the program will consume all of the memory. You will see that, your operating system is getting slower.

A good programmer always freeing allocated memory explicitly. Whenever he uses `malloc()`, puts a corresponding `free()` statement.

Garbage collection (GC), also known as automatic memory management, is the automatic recycling of dynamically allocated memory. Garbage collection is performed by a garbage collector which frees memory that will never be used again. There are many ways for automatic memory managers to determine what memory is no longer required. In the main, garbage collection relies on determining which blocks are not pointed to by any program variables.

Garbage collection was first invented by John McCarthy in 1958 as part of the implementation of Lisp. Systems and languages which use garbage collection can be described as *garbage-collected*. Java, Prolog, Smalltalk etc. are garbage collected languages. C provides more control over program to programmer. For that reason it doesn't worry about freeing unused memory.

All local variables in the stack will be freed and available for reuse after exit from its scope. But dynamic allocated variables will not be freed without a garbage collector. Since C doesn't usually perform garbage collection, the programmers must be carefull if they use `malloc()`.

Why we need dynamic memory allocation?

All variables declared statically at the compile time at stack, will be destroyed while functions are exiting (If the function is main, program will be exited). But sometimes the programmer cannot know how much space the program needs.

For example, the program reads spam words from a file and put them onto the memory. There may be 10 lines or 1,000 lines. We assume a word can be maximum 32-bytes. So in the first case we need 320-bytes of memory. And in the second case we need 32,000-bytes of memory. As result, the

programmer doesn't know about memory requirements beforehand. As a solution he can allocate 1000 word lines staticaly. But if we have 100 words, this will waste our memory. Or if the spam database is so big (for example 10,000 lines) the program won't work correctly.
char wordtable[1000][32];

The best solution is using dynamic memory. The program allocates 32-bytes for each line in a loop.

```
NAME
     malloc, calloc, realloc, free, reallocf -- general purpose memory
allocation functions

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdlib.h>

     void *
     malloc(size_t size);

     void
     free(void *ptr);
```

The `malloc()` function allocates size bytes of memory. The `free()` function causes the allocated memory referenced by ptr to be made available for future allocations.

Following code illustrates a basic memory leak programming mistake.

```
01]
02] int main()
03] {
04]     char* str;
05]     char* tmp;
06]
07]     str = malloc(sizeof(char)*32);
08]     tmp = malloc(sizeof(char)*32);
09]     fscanf(stdin, "%s", str);
10]     tmp = str;
11]
12]     do_something_with_tmp();
13]     do_something_with_str();
14]     free(str);
15]     free(tmp);
16]
17]     return 0;
18] }
19]
```

In line 10, the programmer is using tmp pointer to preserve address of str pointer. Then performing some processes with tmp. And then using str. In line 14 and 15, he is freeing pointers like a good programmer. But he forgets his assignment (tmp = str). Line 14 will success. But line 15 will fail if tmp still points at str. Because str cannot be de-allocated again. Another point is that, the programmer lost address of tmp allocated at line 8. So he never frees tmp.

POINTERS

A pointer is a group of cells (often two or four) that can hold an address. [Kernighan&Ritchie] If `ch` is a char and `p` is a pointer to this char:

```
           p           &ch
+-----+---+-----+---+---+----+-----+
| ... |...| &ch |...|...|char| ... |
+-----+---+-----+---+---+----+-----+
```

The unary operator & gives the address of an object. The *indirection* or dereference operator * gives the "contents of an object pointed by a pointer". To declare a pointer we use dereference operator.

ptr.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6         char ch1 = 'g';
 7         char ch2 = 's';
 8         char *ptr;
 9
10         printf("Char 1 is %c\n", ch1);
11         printf("Char 2 is %c\n", ch2);
12
13         printf("Address of Char 1 is %p\n", &ch1);
14         printf("Address of Char 2 is %p\n", &ch2);
15
16         ptr = &ch1;
17         ch2 = *ptr;
18
19         printf("Char 1 is %c\n", ch1);
20         printf("Char 2 is %c\n", ch2);
21
22         printf("Address of ptr is %p\n", ptr);
23
24         return 0;
25 }
```

```
# ./ptr
Char 1 is g
Char 2 is s
Address of Char 1 is 0xbfbfec83
Address of Char 2 is 0xbfbfec82
Char 1 is g
Char 2 is g
Address of ptr is 0xbfbfec83
```

We declared two char variables in line 6 and 7. Then we declared a char pointer in line 8. ptr is a variable which contains address of ch1 after line 16. ch2 is assigned to value "resides in address ptr points" in line 17. The address ptr points is address of ch1 because of line 16. So these two lines are equal the following line:

```
ch2 = ch1;
```

Examine output: After line 7, Char 1 and Char 2 both are equal and 'g' (value of Char 1). Notice that, address of ptr is equal to address of ch1.

We can make arithmetic operations on pointers.

```
ptr++; /* Points next address. */
(*ptr)++; /* Increments what ptr points to. */
```

ptr++ increments ptr to point next object. It doesn't increments ptr by 1 byte. Added value depends the size of object.

Management of dynamic created variables requires use of pointers. When the programmer need memory at runtime to store a data structure, he demands it by using `malloc()` function.

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

The malloc() function allocates size bytes of memory and returns a pointer points allocated space.

```
typedef struct rulelist rulelist;

struct rulelist {
        int             attr;
        char            ruleline[256];
        rulelist        *next;
};

rulelist *ll;

if((ll = (rulelist *) malloc(sizeof(rulelist))) == NULL) return -1;
```

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function if it is not a global variable. All local variables are accessable locally. If you pass it to function and then alter its value, it doesn't effect on variable of calling function.

```
void swap(int x, int y)  /* WRONG */
{
       int temp;

       temp = x;
       x = y;
       y = temp;
}

swap(a, b);
```

But addresses are accessable from anywhere.

```
void swap(int *px, int *py)  /* interchange *px and *py */
{
       int temp;

       temp = *px;
       *px = *py;
       *py = temp;
}
```

```
swap(&a, &b);
```

Pointer arguments enable a function to access and change objects in the function that called it. [Kernighan&Ritchie]

Pointer subtraction is also valid: if p and q point to elements of the same array, and p<q, then q-p+1 is the number of elements from p to q inclusive. This fact can be used to write yet another version of strlen:

```
int strlen(char *s)
{
       char *p = s;

       while (*p != '\0')
           p++;
       return p - s;
}
```

In its declaration, p is initialized to s, that is, to point to the first character of the string. In the while loop, each character in turn is examined until the '\0' at the end is seen. Because p points to characters, p++ advances p to the next character each time, and p-s gives the number of characters advanced over, that is, the string length.

IMPORTANT!!!

When a pointer is declared it does not point anywhere. Otherwise program will crash. This is one of the famous security bug. You must set pointer variable to point somewhere before you use it. This can be by two ways: 1. Allocating memory for this pointer 2. Assigning it to address of an existing variable.

```
int *Num;

*num = 11;
```

Above code will crash. Correct code is should be something like below:

```
int i = 5;
int *Num;

Num = &i;
*Num = 11;
```

Another way is allocating memory for the pointer from heap:

```
int *Num;

Num = (int *) malloc(sizeof(int));
*Num = 11;
```

REFERENCES

- FreeBSD System Calls Manual
- Structured Computer Organization by Andrew Tanenbaum, 4th Edition, Prentice Hall.
- Assistant Prof. Rifat Yazici, Karadeniz Tech. University, Lecture Notes of `Logic Circuits', 1996
- Dennis Ritchie and Brian Kernighan, The C Programming Language

TODO

- a.out

Nov 5, 2005

Baris Simsek
simsek ~ enderunix.org
http://www.enderunix.org/simsek/
EnderUNIX Software Development Team