# All about the Virtual Memory..

Sravanthi Sinha
Jul 29, 2017 · 11 min read

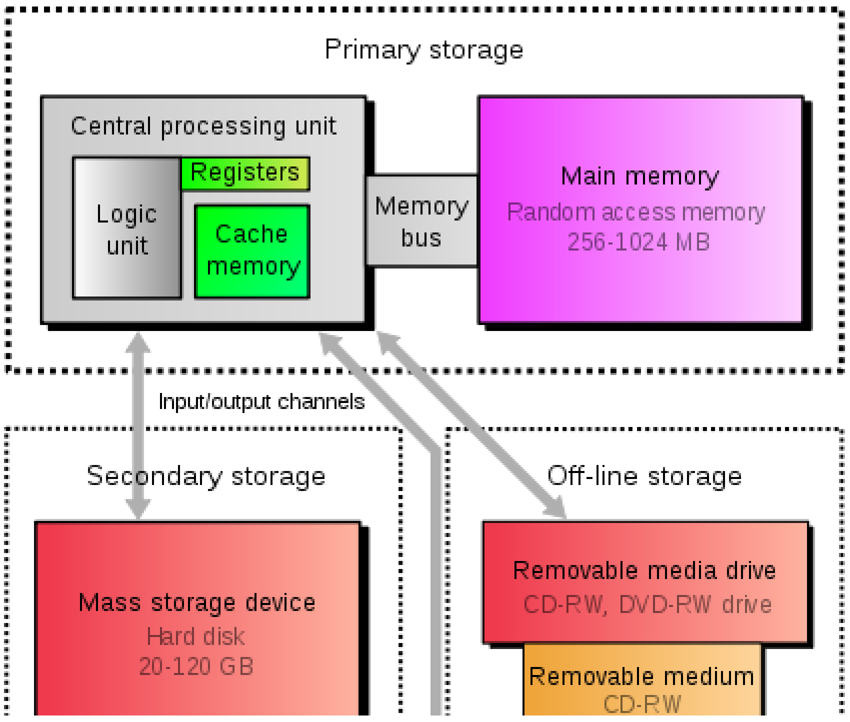
Initially, I would like to describe quick definitions/terms, then walk through on What is Virtual Memory and How does the Virtual Memory map into the RAM?

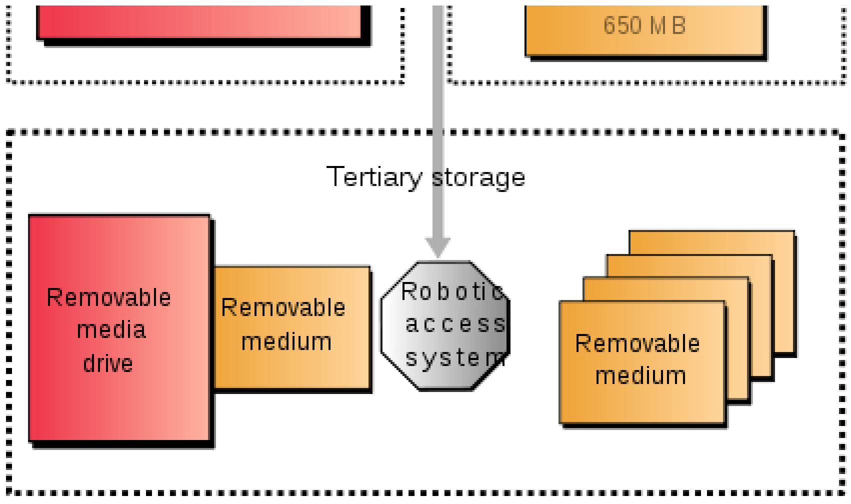
***Storage*** is where the information (such as documents, photos, videos, programs, and even operating systems) is stored.

***Memory*** (aka **system memory**, **Primary Storage, random access memory**, or **RAM**) is where information which is being processed and manipulated is stored. Data in the **system memory** is volatile, meaning that when the device is turned off, it's gone; the memory becomes blank as if nothing has been there before.

***Cache memory*** (aka **CPU memory**) is a storage unit for instructions of the programs frequently call upon during operation, for faster access.

***Secondary Storage*** (aka **auxiliary storage**) is the one which retains data until you either overwrite or delete it. So even when you turn off the device, all data is intact on this medium.





Computer storage types

Let's say you are working on a file it is said to be in the computer's memory. When you save it, a copy of it now resides on the computer's storage (say hard drive). When you close the file completely, the file now only resides on the secondary storage(hard drive) and is no longer in the memory, until you open it again.

***Process*** is a program which is in running, plus the data structures needed to manage it

***Address space*** is the set of valid addresses.

***Virtual Address Space(vas)*** is the set of virtual memory addresses that a process can use.

**Virtual Memory** is a memory management technique that is implemented using both hardware (**MMU**) and software (**operating system**). It abstracts from the real memory available on a system by introducing the concept of **virtual address space**, which allows each process thinking of physical memory as a *contiguous* address space (or collection of contiguous segments).

***Pages*** are blocks of contiguous virtual memory addresses of a virtual address space

***Swapping*** is the term for moving pages from/to secondary storage to/from main memory

. . .

## Virtual Memory (in detail)

Virtual memory is an address mapper. Basically, it maps virtual address space to physical address space (either on RAM or hardware device). Whenever a process is created, the kernel provides a chunk of memory which can be located anywhere at all using VM. Hence the process believes it has all the memory on the computer.

*So let's get to know, what does this Virtual Address space of a process contain*

Consider the following simple program which does nothing but return 0.

```
#include <stdio.h>
/**
 * main - Entry point of the program
 *
 * Return: 0 every time.
 */
int main(void)
{
  return (0);
}
```

> *on compiling and then linking we get an object file (main.o) and an executable (main)*

```
$gcc -Wall -Wextra -Werror main.c  -c
$gcc -o main main.o
```

> *using 'size' tool on the object file*

(we are not using (here) the executable file as it contains our code ***plus*** the content of `stdio.h` library which is pre-processed and then added to the file by the linker)

```
$size -A main.o
main.o  :
section           size   addr
.text               11      0
.data                0      0
.bss                 0      0
.comment            44      0
.note.GNU-stack      0      0
.eh_frame           56      0
Total              111
```

As we can observe the size shows the different sections like text, data, bss.

> *On inspecting the contents of text section using* objdump.

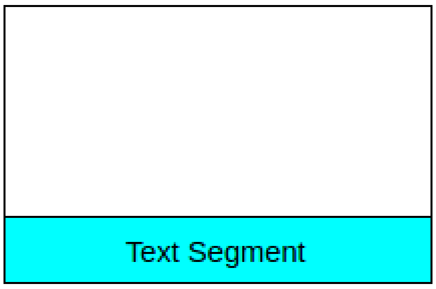
```
$objdump -M intel -j .text -d main.o

main.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0: 55                    push   rbp
   1: 48 89 e5              mov    rbp,rsp
   4: b8 00 00 00 00        mov    eax,0x0
   9: 5d                    pop    rbp
   a: c3                    ret
$
```

> *Lets add a function and check if the text segment increases in size and then inspect its contents.*

```
$cat main-2.c
#include <stdio.h>

/**
 * foo - dummy function to show how this adds to 'text'
 */
void foo(void)
{

}

/**
 * main - Entry point of the program
 *
 * Return: 0 every time.
 */
int main(void)
{
  foo();
  return (0);
}

$gcc -Wall -Wextra -Werror main-2.c -c

$size main-2.o
main-2.o  :
section           size   addr
.text               22      0
```

```
.data                     0      0
.bss                      0      0
.comment                 44      0
.note.GNU-stack           0      0
.eh_frame                88      0
Total                   154

$objdump -M intel -j .text -d main-2.o

main-2.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <foo>:
   0: 55                          push   rbp
   1: 48 89 e5                    mov    rbp,rsp
   4: 5d                          pop    rbp
   5: c3                          ret

0000000000000006 <main>:
   6: 55                          push   rbp
   7: 48 89 e5                    mov    rbp,rsp
   a: e8 00 00 00 00              call   f <main+0x9>
   f: b8 00 00 00 00              mov    eax,0x0
  14: 5d                          pop    rbp
  15: c3                          ret
$
```

*Now we can establish that the Virtual Address Space has the text segment which contains the program loaded to execute.*



Virtual Address Space

> *Lets add a static **initialized** variable to our program.*

```
#include <stdio.h>

/**
 * main - Entry point of the program
 *
 * Return: 0 every time.
 */
int main(void)
{
    static char* str = "C is fun";

    return (0);
}
```

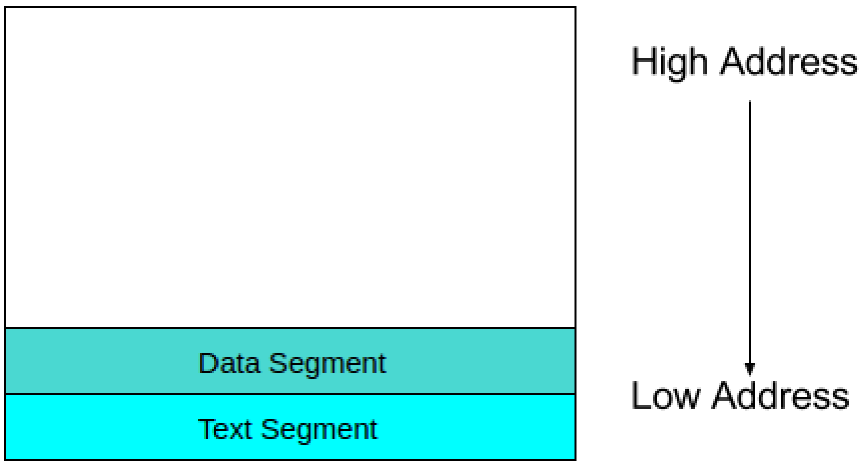
> *on size we can see the data section of this process has increased by 4*

```
$gcc main-3.c -o main-3
$size  main main-3
   text    data     bss     dec     hex filename
   1115     552       8    1675     68b main
   1124     560       8   1675     68b main-3
hybridivy@sinha:0x03-proc_filesystem$
```

*Hence we can say that the initialized static variables are stored in **data** segment. But we are not sure of the position of **data** segment.*

```
$objdump -d -j .data main-3| head -n 7 | tail -n 3
Disassembly of section .data:

0000000000601028 <__data_start>:
$objdump -d -j .text main-3| head -n 7 | tail -n 3
Disassembly of section .text:

0000000000400400 <_start>:
$
```

From the addresses used by the process for text and data, we can justify that **data** segment is after ***text*** segment.



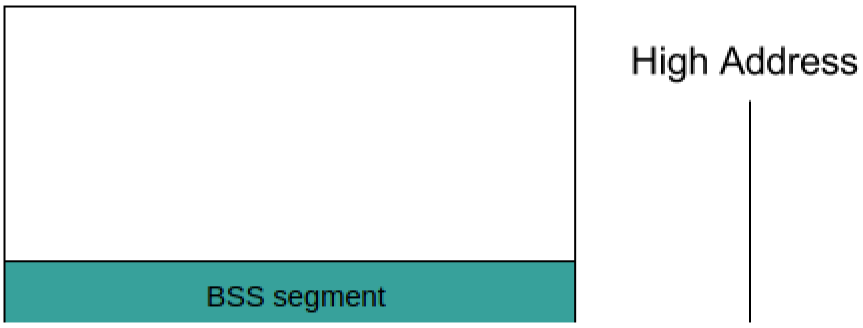
*Now, Lets make **uninitialize** the static variable .*

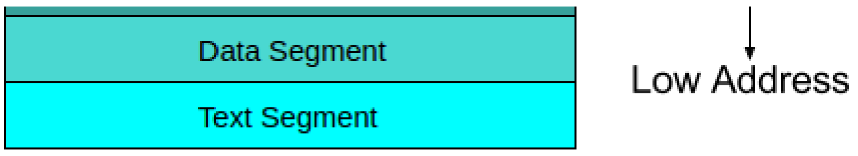
```
#include <stdio.h>

/**
 * main - Entry point of the program
 *
 * Return: 0 every time.
 */
int main(void)
{
    static char* str;

    return (0);
}
```

Similar to the process followed above of finding a size and viewing objdump we can confirm that the **bss** segment comes after the **data** segment.

```
$size main main-4
   text    data     bss     dec     hex filename
   1115     552       8    1675     68b main
   1115     552      16    1683     693 main-4

$objdump -d -j .bss main-4| head -n 7 | tail -n 3
Disassembly of section .bss:

0000000000601038 <__bss_start>:
$objdump -d -j .data main-4| head -n 7 | tail -n 3
Disassembly of section .data:

0000000000601028 <__data_start>:
```

Our updated figure would be





Virtual Address Space

> *How about performing a* dynamic memory allocation *in our program using malloc function.*

from man pages of malloc :

```
MALLOC(3)        Linux Programmer's Manual         MALLOC(3)

NAME
       malloc, free, calloc, realloc - allocate and free dynamic
memory

SYNOPSIS
       #include <stdlib.h>

void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);

DESCRIPTION
       The malloc() function allocates size bytes and returns a
pointer to the allocated memory.
...
...
 Normally, malloc() allocates memory from the heap and adjusts the
size of the heap as required
```

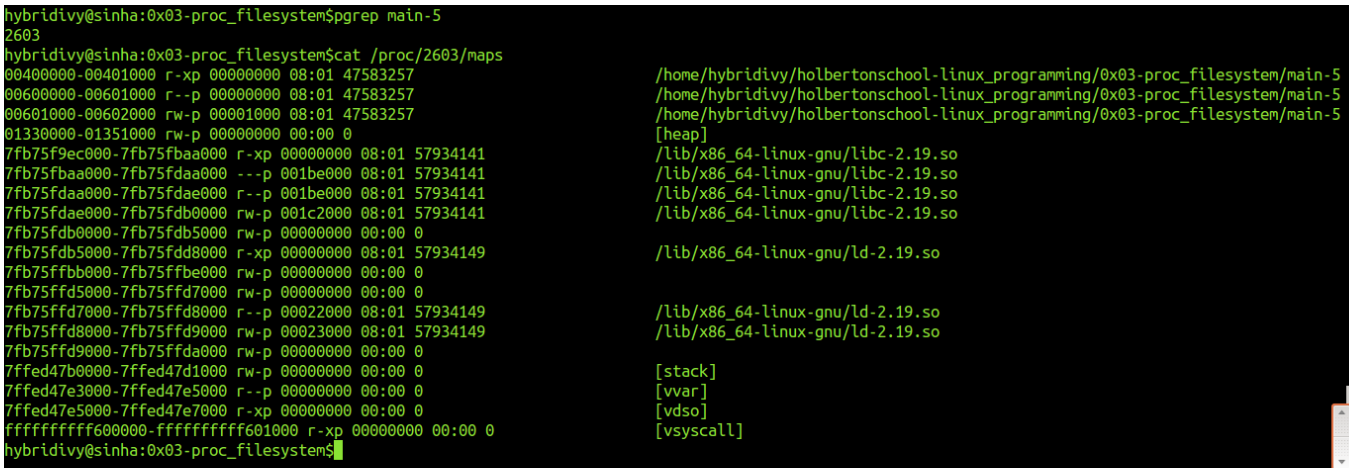
Looks like the dynamic allocation of memory uses the heap. H***eap*** and ***stack*** are created by the OS at run time; that is after the executable has been loaded into virtual memory. Therefore, they are not part of the executable file.o

The way to find the contents of heap or stack would be using a ***/proc fs*** where there are many sub directories under it, describing a lot of information for an each process instance holding a ***pid .***

In the following program we are dynamically allocating the memory using malloc and then run a loop infinitely (so that we can track its memory info)

```c
#include <stdio.h>
#include <stdlib.h>
/**
 * main - Entry point of the program
 *
 * Return: 0 every time.
 */
int main(void)
{
    char *p = (char*)malloc(sizeof(char));

    while (1)
    {
        sleep(1);
    }
    return (0);
}
```

Each time we run a process the ***pid*** changes and the memory address used by ***stack*** and ***heap*** too. On fetching the process id ***pid*** and then checking the maps file under the proc fs

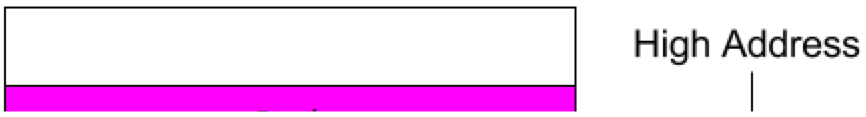


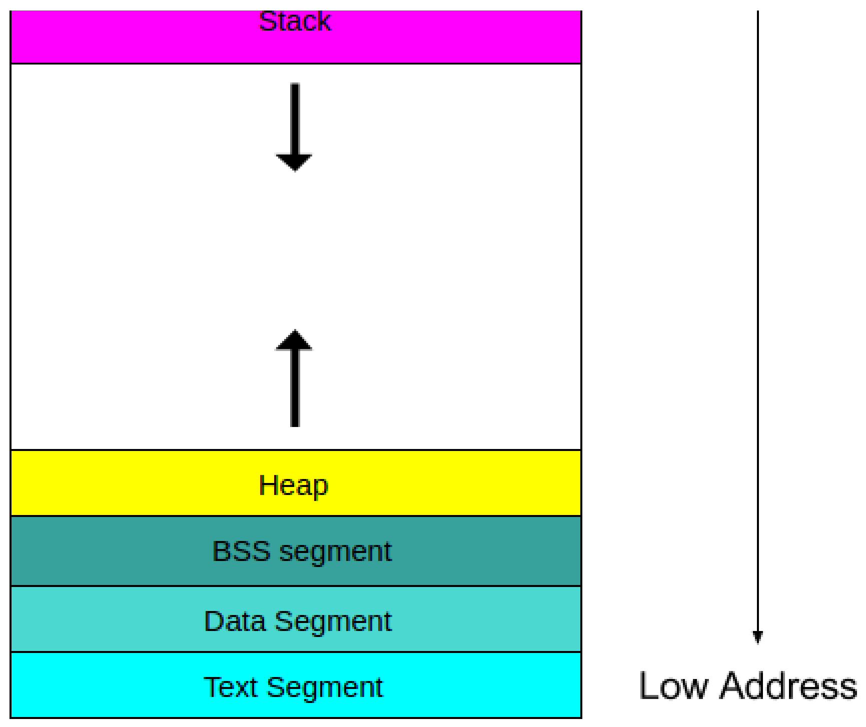
```
hybridivy@sinha:0x03-proc_filesystem$pgrep main-5
2603
hybridivy@sinha:0x03-proc_filesystem$cat /proc/2603/maps
00400000-00401000 r-xp 00000000 08:01 47583257                           /home/hybridivy/holbertonschool-linux_programming/0x03-proc_filesystem/main-5
00600000-00601000 r--p 00000000 08:01 47583257                           /home/hybridivy/holbertonschool-linux_programming/0x03-proc_filesystem/main-5
00601000-00602000 rw-p 00001000 08:01 47583257                           /home/hybridivy/holbertonschool-linux_programming/0x03-proc_filesystem/main-5
01330000-01351000 rw-p 00000000 00:00 0                                  [heap]
7fb75f9ec000-7fb75fbaa000 r-xp 00000000 08:01 57934141                   /lib/x86_64-linux-gnu/libc-2.19.so
7fb75fbaa000-7fb75fdaa000 ---p 001be000 08:01 57934141                   /lib/x86_64-linux-gnu/libc-2.19.so
7fb75fdaa000-7fb75fdae000 r--p 001be000 08:01 57934141                   /lib/x86_64-linux-gnu/libc-2.19.so
7fb75fdae000-7fb75fdb0000 rw-p 001c2000 08:01 57934141                   /lib/x86_64-linux-gnu/libc-2.19.so
7fb75fdb0000-7fb75fdb5000 rw-p 00000000 00:00 0
7fb75fdb5000-7fb75fdd8000 r-xp 00000000 08:01 57934149                   /lib/x86_64-linux-gnu/ld-2.19.so
7fb75ffbb000-7fb75ffbe000 rw-p 00000000 00:00 0
7fb75ffd5000-7fb75ffd7000 rw-p 00000000 00:00 0
7fb75ffd7000-7fb75ffd8000 r--p 00022000 08:01 57934149                   /lib/x86_64-linux-gnu/ld-2.19.so
7fb75ffd8000-7fb75ffd9000 rw-p 00023000 08:01 57934149                   /lib/x86_64-linux-gnu/ld-2.19.so
7fb75ffd9000-7fb75ffda000 rw-p 00000000 00:00 0
7ffed47b0000-7ffed47d1000 rw-p 00000000 00:00 0                          [stack]
7ffed47e3000-7ffed47e5000 r--p 00000000 00:00 0                          [vvar]
7ffed47e5000-7ffed47e7000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
hybridivy@sinha:0x03-proc_filesystem$
```

We can see that the heap is using the addresses ***`01330000`*** to ***`01351000`*** and ***stack*** is using `7fffed47`***`b0000`*** to `7fffed47`***`d1000`***. By which we can say that stack and heap are not in a ***contiguous*** memory region.

Both the stack and the heap can grow based on the way program handles the function calls and memory allocations.

Hence our VM figure would look like:





We are not done yet, where do you think the Command line arguments and Environment Variables would be? Any guesses?

> *Let's bring Command line arguments and Environment Variables.*

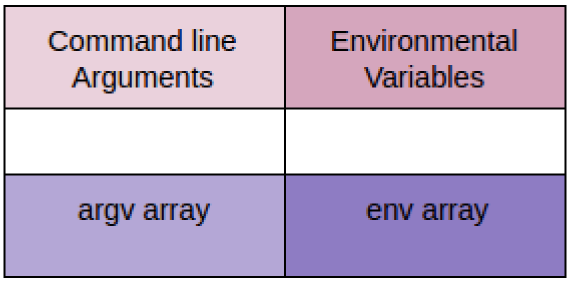
```
#include <stdio.h>

/**
 * main - print locations of Command line arguments and environment
 * variables.
 *
 * Return: 0 every time.
 */
int main(int argc, char *argv[], char **env)
{
  int i;

  printf("Address of the array of arguments: %p\n", (void *)argv);
  printf("Addresses of the arguments:\n");
  for (i = 0; i < argc; i++)
  {
    printf("\t[%s]:%p\n", argv[i], argv[i]);
  }
  printf("Address of the array of environment variables: %p\n",
(void *)env);
  printf("Address of the first environment variable: %p\n", (void *)
(env[0]));
  while (1)
  {
    sleep(1);
  }
  return (0);
}
```

*On Compiling and executing.*

```
$./main-6 "C is fun!" "C is awesome!"
Address of the array of arguments: 0x7ffe78aa82c8
Addresses of the arguments:
 [./main-6]:0x7ffe78aaa090
 [C is fun!]:0x7ffe78aaa099
 [C is awesome!]:0x7ffe78aaa0a3
Address of the array of environment variables: 0x7ffe78aa82e8
Address of the first environment variable: 0x7ffe78aaa0b1
```

We can observe that the command line arguments and the environmental variables are allocated in contagious memory allocations, Similarly the ***argc*** array and the ***env*** array are allocated in contagious memory allocations. Let's make sure of it.

```
$printf "Distance btw last command line arg and first env variable:
%d\n" $((0x7ffe78aaa0b1-0x7ffe78aaa0a3))
Distance btw command line args and env variables: 14
$echo "C is awesome!"| wc -c
14
$printf "Distance btw argv array  and env array: %d\n"
$((0x7ffe78aa82e8-0x7ffe78aa82c8))
Distance btw argv array  and env array: 32
$echo "./main-6 C is fun C is awesome!"| wc -c
32
$
```



Now, we are only left with fitting this block in the VM space.

Let's look at the ***address space*** located for ***stack*** vs the address of an ***argv*** array.

*Fetching the* **pid** *and inspecting the address of the* **stack** *using /proc/pid/maps file.*

```
$pgrep main-6
5902
$cat /proc/5902/maps | grep “stack”
7ffe78a8a000-7ffe78aab000 rw-p 00000000 00:00 0 [stack]
```

Looks like the argv array which is at `0x7ffe78aa82c` is in the space allocated for the stack. Does that mean, Command Line arguments and the Environmental variables are in a stack?

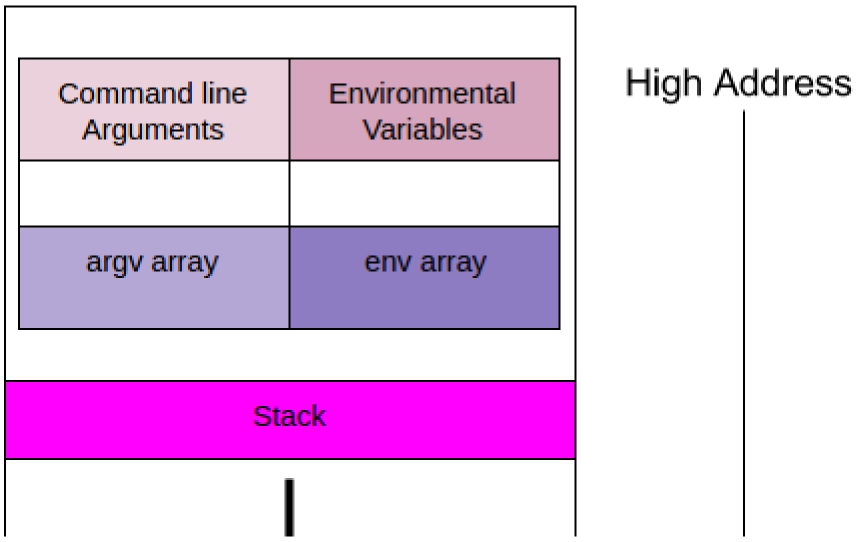
*But wait! Is the process really using the space allocated for a stack?*

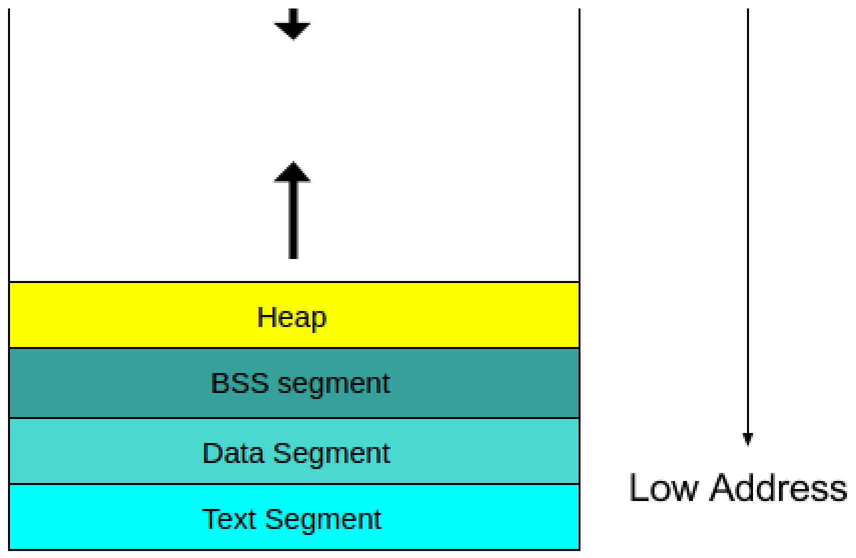
Let’s make use of the *proc/pid/stat* file, which provides various status information about a process like the state (sleeping/waiting/etc), parent pid (ppid), ***startstack (28)*** — The address of the start (i.e., bottom) of the stack, ***kstkesp*** **(29)** — The current value of ESP (stack pointer), as found in the kernel stack page for the process.

```
$(printf 'Stack Allocated from %d to %d\n' 0x7ffe78aab000
0x7ffe78a8a000) && echo 'Stack Actually Utilized:' $(awk -F' '
'{print $28"  "$29}' /proc/5902/stat)  &&  (printf 'Address of the
command line arguments array %d \n' 0x7ffe78aa82c8)

Stack Allocated from 140730922872832 to 140730922737664
Stack Actually Utilized: 140730922861248 140730922860488
Address of the command line arguments array 140730922861256
```

It clearly shows that the Command Line Arguments & Environmental variables block is above stack**.**

Hence our final VM Address Space looks like





Virtual Address Space

## Concept of Virtual Memory

- Each process is given physical memory called the process's *virtual memory space and a* process is unaware of the details of its physical memory (i.e. where it physically resides). All the process knows is how big the chunk is and that its chunk begins at address 0.
- Each process is unaware of any other chunks of VM belonging to other processes. Even if the process *did* know about other chunks of VM, it's prevented by the kernel from accessing that memory.
- Each time a process wants to read or write to memory, its request must be translated from a VM address to a physical memory address. Conversely, when the kernel needs to access the VM of a process, it must translate a physical memory address into a VM address

## Benefits of Virtual Memory

1. Each process can have a different memory mapping

- One process's RAM is inaccessible (and invisible) to other processes. (Built-in memory protection)
- Kernel RAM is invisible to userspace processes.

2. Physical RAM can be mapped into multiple processes at once.

- Shared memory

3. Memory regions can have access permissions

- Read, write, execute

4. Being able to conceptually use more memory than might be physically available, using the technique of **paging**.

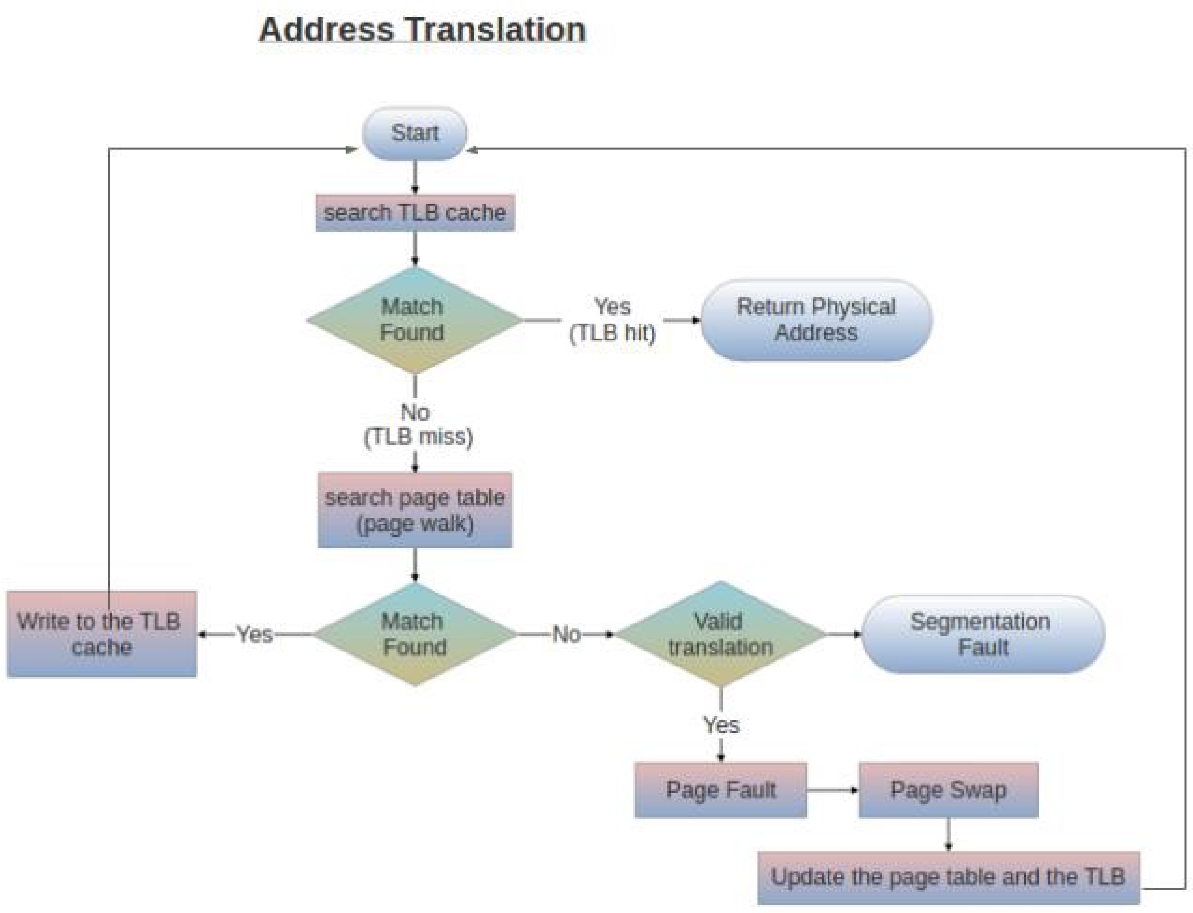
. . .

## How does Virtual Memory work?

The OS stores the mappings between virtual and physical addresses in a data structure called as **pagetable.**

The most recently used mappings are cached by the MMU in **Translation Lookaside Buffer (TLB).**

Address Translation is done by on the CPU chip by a specific hardware element called **Memory Management Unit** or **MMU,** where as the *virtual address spaces management is done by the OS.*



Address Translation as done by MMU

. . .

It is good to know that there are three kinds of virtual addresses in Linux

1. Kernel Logical Addresses which have a fixed mapping between physical and virtual address space implying the virtually-contiguous regions are by nature also physically contiguous.

2. Kernel Virtual Addresses which are used for non-contiguous memory mappings.

3. User Virtual Addresses which represent memory used by user space programs, where each process has its own mapping. Unlike kernel logical addresses, which use a fixed mapping between virtual and physical addresses, user space processes make full use of the MMU.

- Only the used portions of RAM are mapped
- Memory is not contiguous

## Thank you for reading!

Any questions or feedback? Comments are always welcome! :)

*Many thanks to Alex for proof-reading!*

Read More:

**Memory Map in C**

A typical memory representation of C program consists of following sections. 1. Text segment 2.…

automotivetechis.wordpress.com

**Anatomy of a Program in Memory**

Memory management is the heart of operating systems; it is crucial for both programming and…

duartes.org

**Virtual Memory**

© 2003 by Charles C. Lin. All rights reserved. A cache stores a subset of the addresss space of…

www.cs.umd.edu

**text, data and bss: Code and Data Size Explained**

In "Code Size Information with gcc for ARM/Kinetis" I use an option in the ARM gcc tool…

mcuoneclipse.com

**Virtual Memory, Paging, and Swapping**

Virtual Memory is a memory management technique that is implemented using both…

gabrieletolomei.wordpress.comhttp://events.linuxfoundation.org/sites/events/files/slides/elc_2016_mem…

**Peter's gdb Tutorial: Memory Layout And The Stack**

To effectively learn how to use GDB, you must understand frames, which are also called stack frames because they're the…

www.dirac.org

Virtual Memory C Memory Map Programming Address Translation