CS9447 – 15s2 Lecture 2 part 1

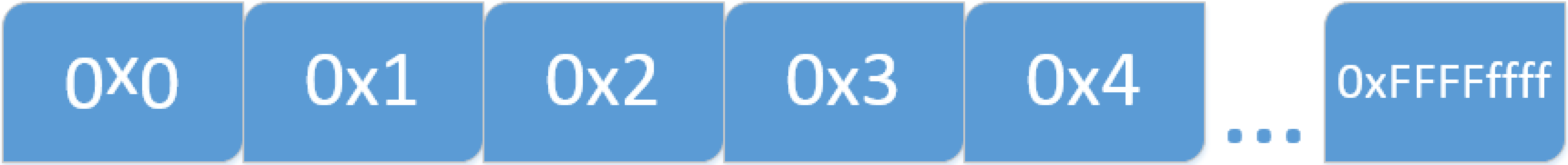
MEMORY FUNDAMENTALS

Apologies

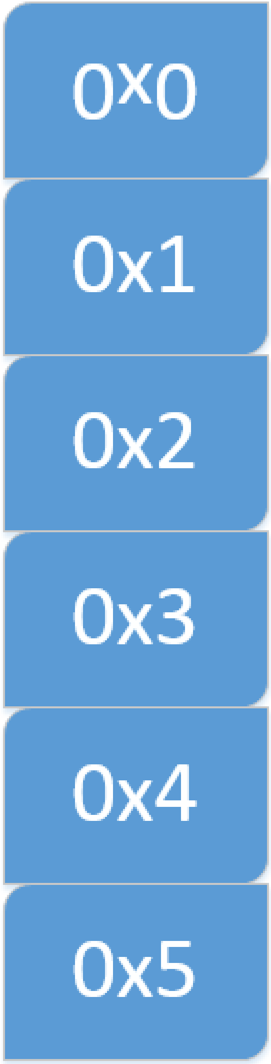
- I will rant a fair bit about which way to think about memory.
- This is because it is VERY IMPORTANT, and so many modern sources do it backwards, then have to trial and error when adding and minusing pointers in exploits etc. seriously half the diagrams are in the wrong order, and I see people after ten years of doing security still confused as to where things are mapped out, and they draw diagrams both ways and and and..
- A proper mental picture helps a lot.
- A lot of this is just raw theory, and I have no real practical component for this introductory stuff. There's some stuff you need to know before we start reversing, and this lecture is it. Next lecture will be waaaay more hands on.
- A lot of you know will know all of what I am going to say. Sorry ;-)

Thinking about memory

- CPUs, at a low level, simply see a long strip of main memory (sometimes, strange architectures or segmentation mean we have multiple strips of memory, but for all intents and purposes it makes no difference)
- CPUs also have a limited number of registers, which are small but extremely fast memory which is directly on the CPU, and is not considered main memory



Alternative strip



0x0

0x1

0x2

0x3

0x4

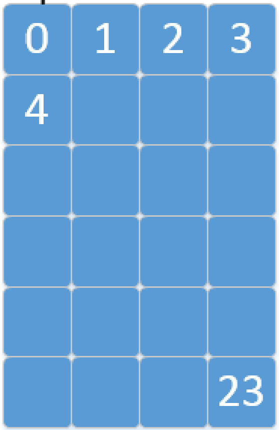
0x5

- It doesn’t really matter whether we think horiziontal or vertical, it’s not really a physical object.
- It’s just a long address range, where each byte can be addressed sequentially.
- Memory is actually more likely to be not be 1d: 2d or more likely, multiple 3d objects: Multiple sticks of ram of various sizes. But the processor just accesses it all by address.
- Registers are often 2D, for speed reasons. But we don’t care, conceptually.

Sometimes 2 dimensions help

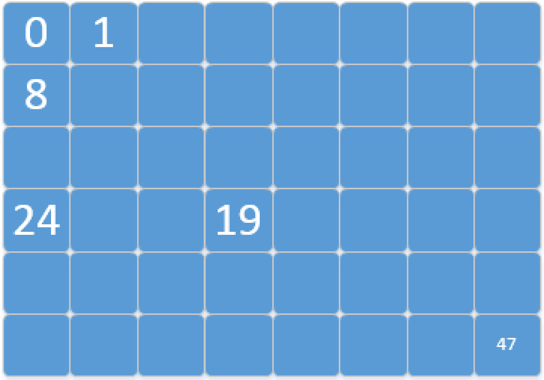
- Sometimes it’s beneficial to use 2 dimensions, where the width of the memory is the number of bytes of the basic unit we are using.

Top of 32 bit stack

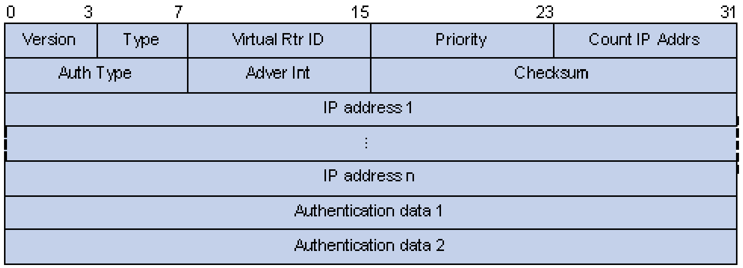


Bottom of 32 bit stack

Stacks are often visualized or thought of with a width of a pointer (ie. 32-bit stacks are 4 bytes wide, visually, 64-bit, 8)



Bottom of 64 bit stack



0 3	7	15	23	31
Version	Type	Virtual Rtr ID	Priority	Count IP Addrs
Auth Type		Adver Int	Checksum	
IP address 1				
⋮				
IP address n				
Authentication data 1				
Authentication data 2				

Network Protocols (this is VRRP) are often drawn 4 bytes wide, for convenience.

Integers in memory

- C and C++ use several core types to do the bulk of their work. These are all integers of various sizes. (Stored as whole numbers).

- Many types have signed (can be positive, zero, or negative) and unsigned (only positive and zero) values.

- **char** (signed) and unsigned **char** are usually 1 byte.
- **short** (signed) and unsigned **short** are 2 bytes or more.
- **int** (signed) and unsigned **int** are normally 4 bytes, and must be at least as long as short.
- **long** (signed) and unsigned **long**, are at least 4 bytes and at least as long as int.

Unsignedness

Unsigned integer values are encoded in pure binary form, which is a base two numbering system. Each bit is a 1 or 0, indicating whether the power of two that the bit's position represents is contributing to the number's total value. To convert a positive number from binary notation to decimal, the value of each bit position n is multiplied by 2n1.

A few examples of these are shown below:

- 0001 1011: 2^4 + 2^3 + 2^1 + 2^0 = 27
- 0000 1111: 2^3 + 2^2 + 2^1 + 2^0 = 15
- 0010 1011: 2^5 + 2^3 + 2^1 = 42

Two’s compliment

Signed values on x86 are stored using two’s complement.

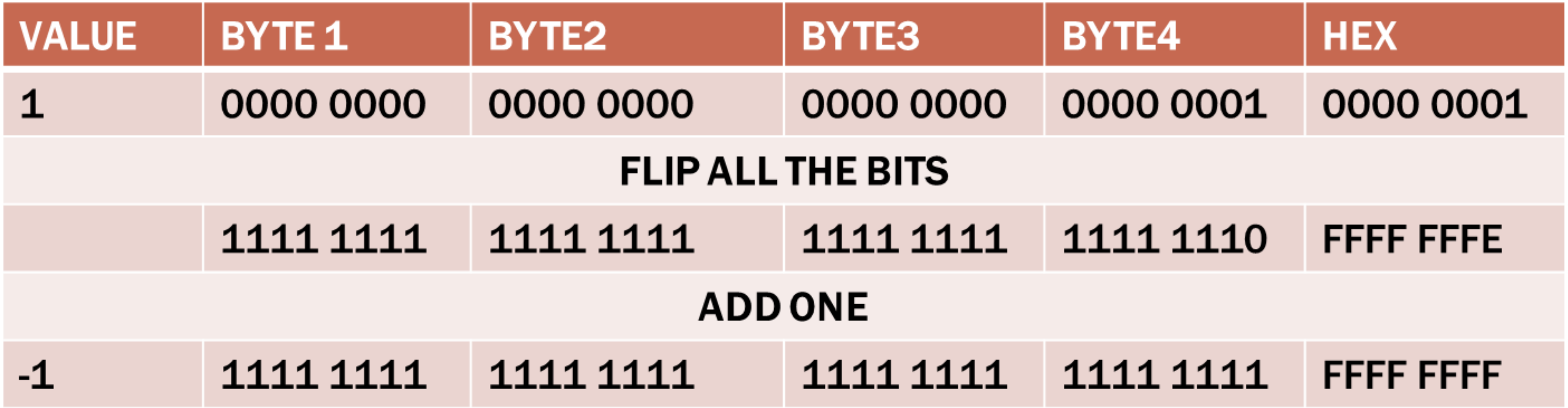
The “most significant” bit is called the sign bit, and if this bit is 1, the stored number is negative.

You can read positive values directly from the value bits you can't read negative values directly; you have to negate the whole number first.

In two’s complement, a number is negated by inverting all the bits and then adding one. This works well for the machine and removes the ambiguity of having two potential values of zero.

int value = 1;

- -1 is stored as 0xFFFFffff (for 32 bit integers)



VALUE	BYTE 1	BYTE2	BYTE3	BYTE4	HEX
1	0000 0000	0000 0000	0000 0000	0000 0001	0000 0001
FLIP ALL THE BITS					
	1111 1111	1111 1111	1111 1111	1111 1110	FFFF FFFE
ADD ONE					
-1	1111 1111	1111 1111	1111 1111	1111 1111	FFFF FFFF

Pointers

- Pointers can point to many things, but they all have the same format, which is architecture dependent.
- There’s more to it than this..
- But basically, normally, they are just a variable which contains the address of where they point to.

Another borrowed diagram!

(http://www.ntu.edu.sg/home/ehchua/programming/cpp/images/MemoryAddressContent.png)

int anInt = 255;

short = aShort = 0xFFFF;

aDouble double = whatever;

Int *ptrAnInt=&anInt;

Note that here we are

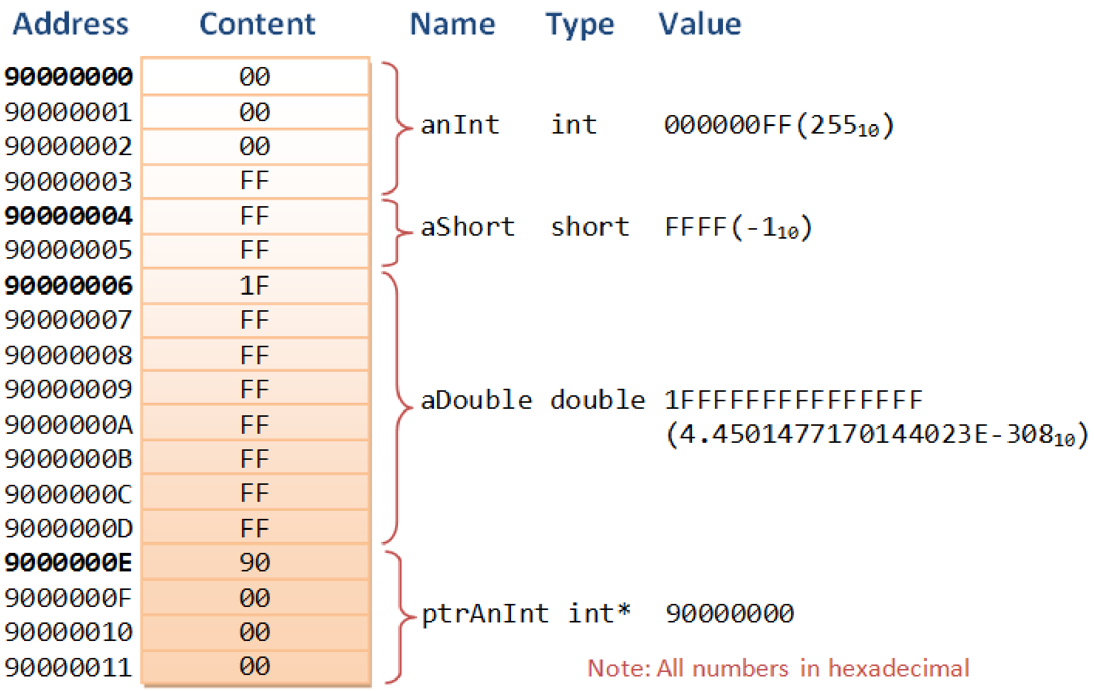
On a big-endian system,

And also we are packing

These values in much

Tighter than we normally

Would



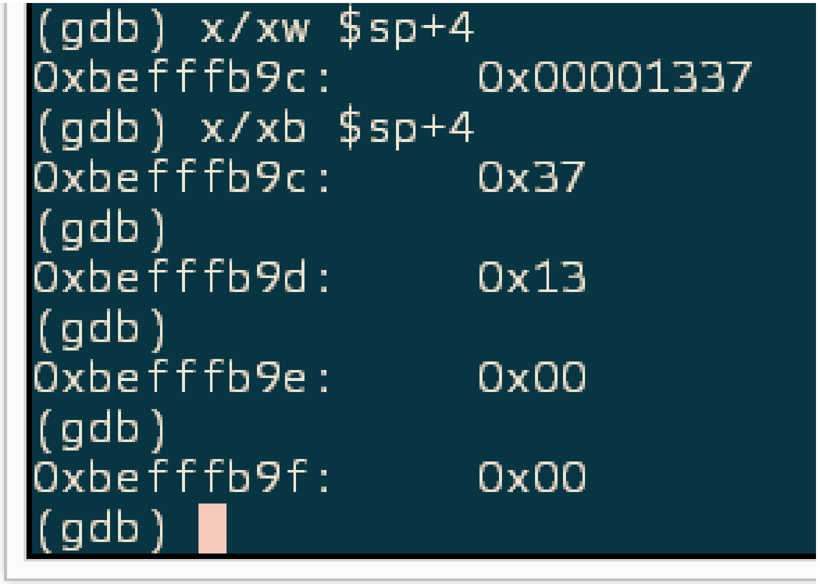
Address	Content	Name	Type	Value
90000000	00	anInt	int	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	aShort	short	FFFF(-1_{10})
90000005	FF			
90000006	1F	aDouble	double	1FFFFFFFFFFFFFFF ($4.4501477170144023E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrAnInt	int*	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

Endianess

- Each architecture has an endianess. Some are hard-set, whilst other architectures let you pick.
- Big-endian and little-endian.
- x86 and x86_64 is little endian.
- MIPS, Power, PowerPC, …
- Sparc, PowerPC, and ARM let you choose: Sparc is normally big, ARM 50-50, originally big but now, more and more, little (linux now little, iOS little..)
- Little-endian is becoming more and more prevalent (it used to be called "wrong-endian" because you can be a bit more dodgy with your types and things won't break. Basically, you need to rework values if you extend their size in big-endian. A lot of the functionality promoted as the reason why little-endian is good is dangerous/potentially exploitable. Heh ;-)
- Any performance difference is generally negated/reversed these days as.. network protocols are big-endian. Hence the htons() etc. functions. These do nothing on big-endian platforms, but on little-endian platforms they have to swap the endianess.
- Sometimes you will get stung by old fashioned file systems, which care about endianess. UFS for example is incompatible between systems of different endianess. :-O
- Endianess does not apply to registers..

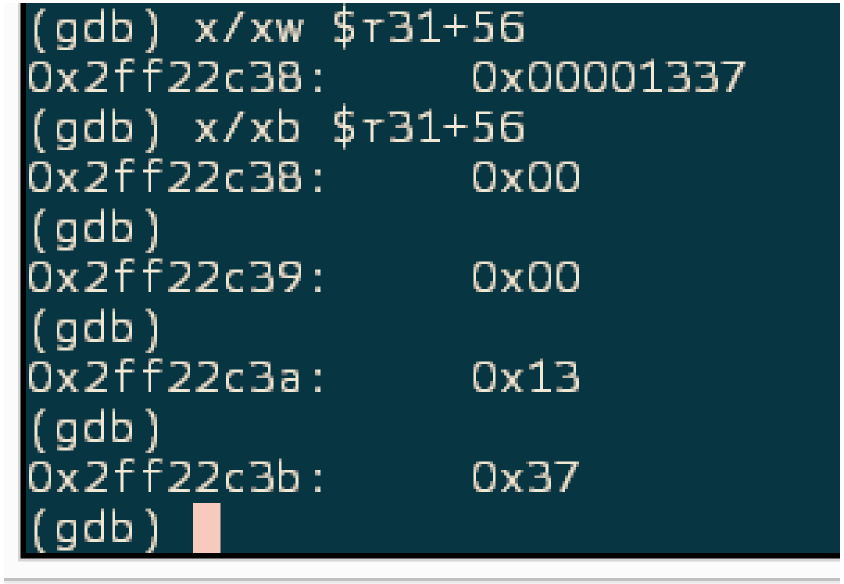
Graphically



```
(gdb) x/xw $sp+4
0xbefffb9c:     0x00001337
(gdb) x/xb $sp+4
0xbefffb9c:     0x37
(gdb)
0xbefffb9d:     0x13
(gdb)
0xbefffb9e:     0x00
(gdb)
0xbefffb9f:     0x00
(gdb)
```

37	13	00	00

Little-endian
0x00001337
(linux, ARMLE)



```
(gdb) x/xw $r31+56
0x2ff22c38:     0x00001337
(gdb) x/xb $r31+56
0x2ff22c38:     0x00
(gdb)
0x2ff22c39:     0x00
(gdb)
0x2ff22c3a:     0x13
(gdb)
0x2ff22c3b:     0x37
(gdb)
```

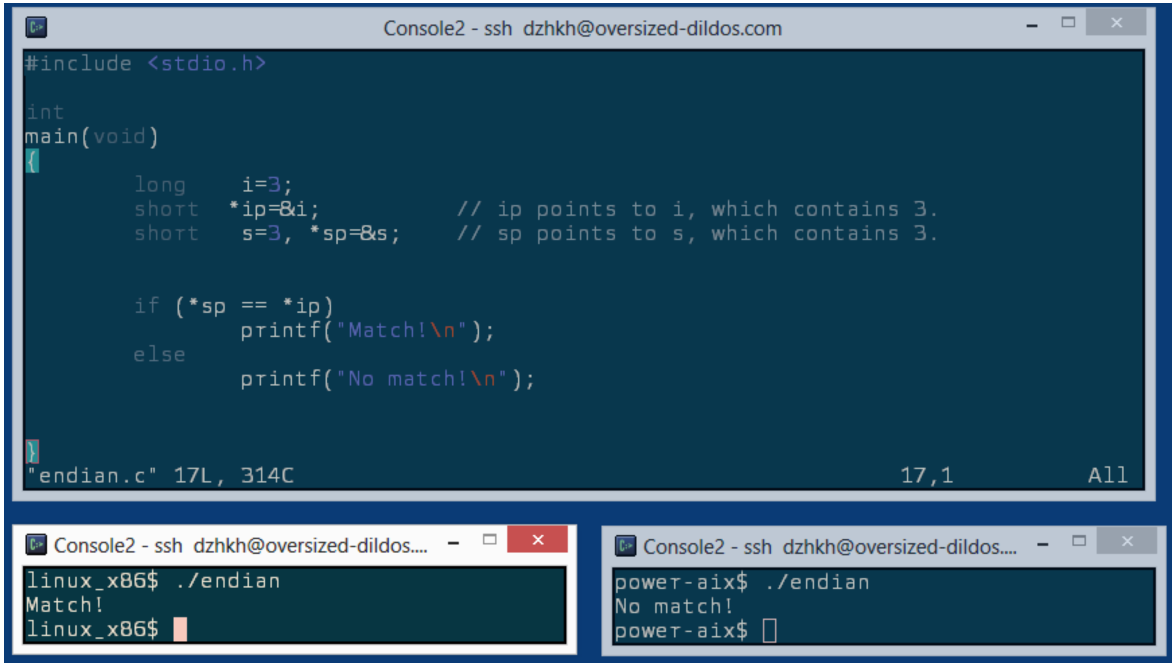
00	00	13	37

Big-endian
0x00001337
(AIX, POWER7)

- All those weird diagrams which make structures, strings, kernel concepts, stacks, etc. appear backwards in memory (lowest address in bottom right hand corner) are just trying to reconcile this one thing, and make little-endian numbers make sense.
- Little endian does make more sense for integers, but that’s no reason to fuck up every other concept and have all that stuff running backwards in diagrams.
- Just remember: Bits and bytes on integer types on little endian platforms run in the opposite direction to the rest of memory. Right to left. This is much better than having integers in the correct order but everything else in reverse order.

What was the problem with big-endian? It seems to make sense..

- It does, but it’s more difficult to extend values. And you can’t pretend integers of different width are the same, via pointer. (shouldn’t do this anyway). There are speed issues with bitshifting and increasing the width of things in general. The concept of bitshifting visually favours the little-endian model <<


Console2 - ssh dzhkh@oversized-dildos.com

```
#include <stdio.h>

int
main(void)
{
        long    i=3;
        short  *ip=&i;          // ip points to i, which contains 3.
        short   s=3, *sp=&s;    // sp points to s, which contains 3.


        if (*sp == *ip)
                printf("Match!\n");
        else
                printf("No match!\n");


}
"endian.c" 17L, 314C                                        17,1          All
```

Console2 - ssh dzhkh@oversized-dildos....

```
linux_x86$ ./endian
Match!
linux_x86$
```

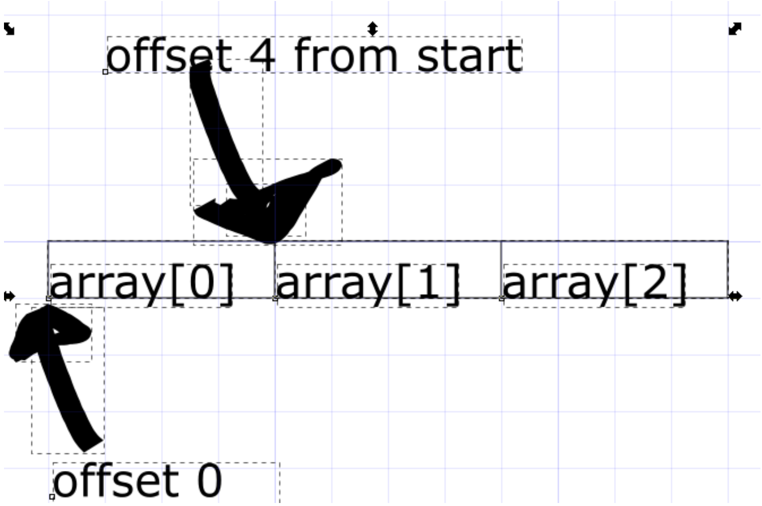
Console2 - ssh dzhkh@oversized-dildos....

```
power-aix$ ./endian
No match!
power-aix$
```

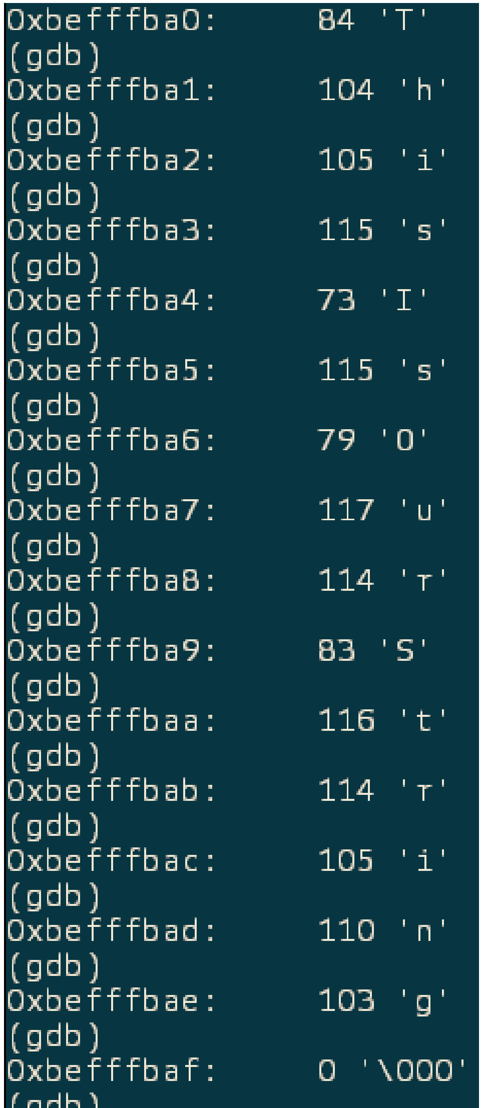
Arrays

- Arrays are just lined up lists of the original type…

int   array[3];

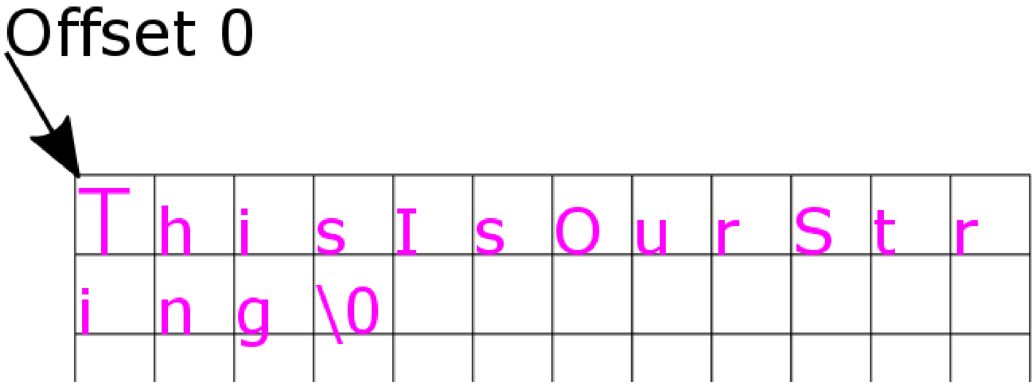


Strings are just character arrays



```
0xbefffba0:     84 'T'
(gdb)
0xbefffba1:     104 'h'
(gdb)
0xbefffba2:     105 'i'
(gdb)
0xbefffba3:     115 's'
(gdb)
0xbefffba4:     73 'I'
(gdb)
0xbefffba5:     115 's'
(gdb)
0xbefffba6:     79 'O'
(gdb)
0xbefffba7:     117 'u'
(gdb)
0xbefffba8:     114 'r'
(gdb)
0xbefffba9:     83 'S'
(gdb)
0xbefffbaa:     116 't'
(gdb)
0xbefffbab:     114 'r'
(gdb)
0xbefffbac:     105 'i'
(gdb)
0xbefffbad:     110 'n'
(gdb)
0xbefffbae:     103 'g'
(gdb)
0xbefffbaf:     0 '\000'
(gdb)
```

```
char s[] = "ThisIsOurString";
```



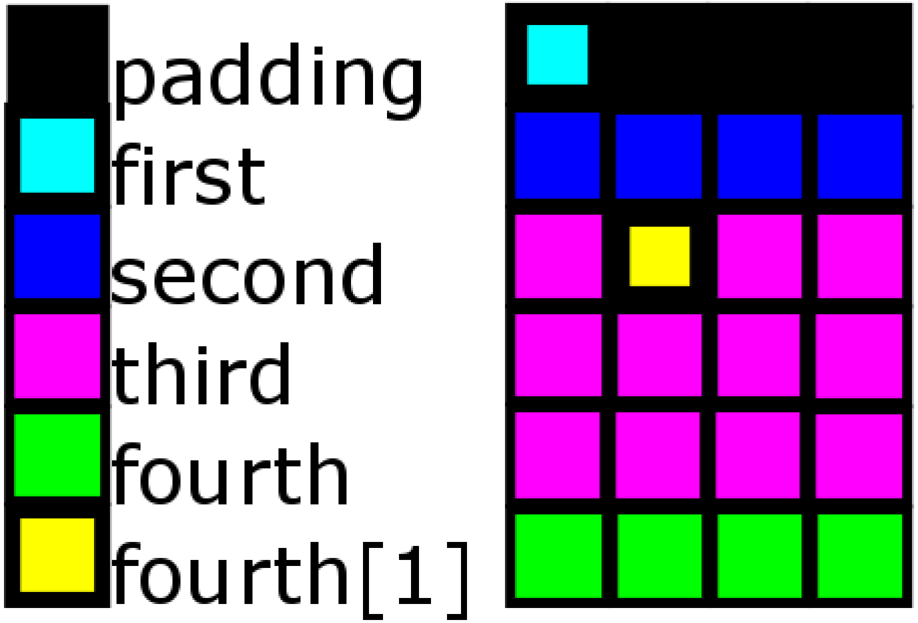
Offset 0

T	h	i	s	I	s	O	u	r	S	t	r
i	n	g	\0								

On the left I have used x/c in GDB to use each byte. As we can see, strings go from the left to the right of the page, if we start with the lowest address in the top left hand corner.

How structures look

```
struct sample {
        char  first = 1;
        int     second = 2;
        char  third[12] = “third”;
        int     fourth = 4;
} sampleOne;
```



This is just a guess. The padding is optional. The elements can be rearranged by the compiler. Etc. but this is generally how things look.

Backwards diagrams

- About half the diagrams you see are backwards; Seriously, I cannot strongly recommend enough that you do not think about memory upside down.
- All the time I see people confusing stack growth and heap growth and string and integer order etc.

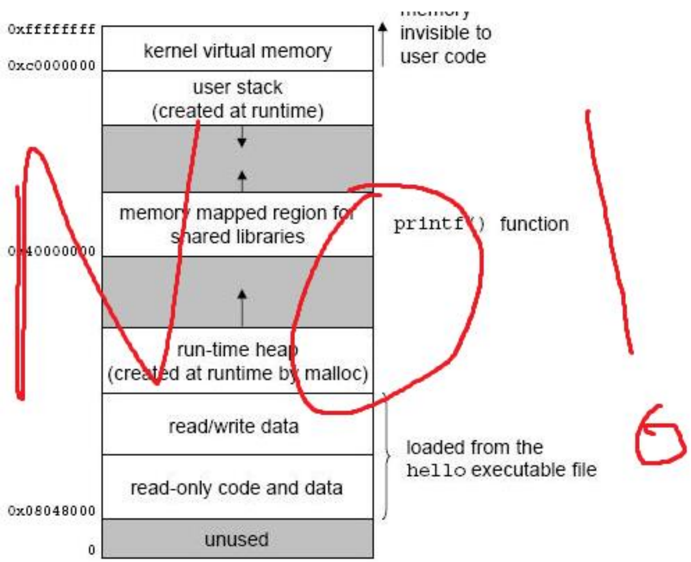
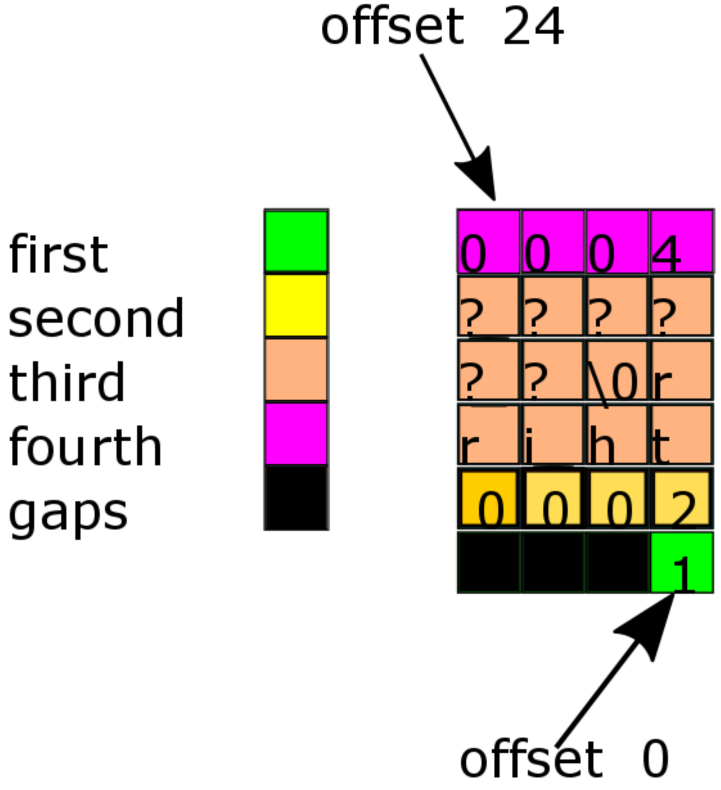


Figure 1.13: **Linux process virtual address space.**

The world in which little-endian runs the right way and we number the bottom right as the low address.

```
struct sample {
        char  first = 1;
        int   second = 2;
        char  third[12] = “third”;
        int   fourth = 4;
} sampleOne;
```



offset 24

first	🟩
second	🟨
third	🟧
fourth	🟪
gaps	⬛

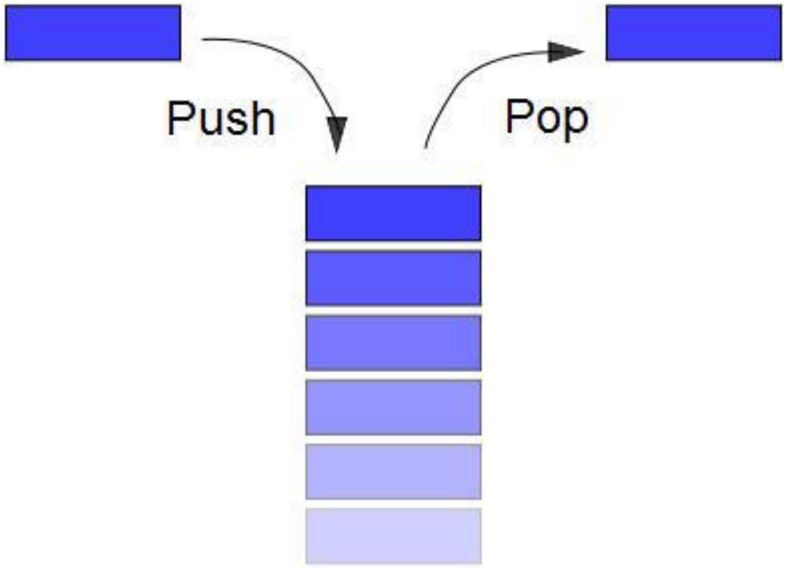
0	0	0	4
?	?	?	?
?	?	\0	r
r	i	h	t
0	0	0	2
⬛	⬛	⬛	1

offset 0

Wow that’s much better… ;-)

STACKS

- A *stack* (also called a *pushdown list*) is a data structure that is used to remember things and recall them later. A stack uses the rule "Last In, First Out.“ (LIFO) That is, when you take something out of a stack, the one you get is the one you put in most recently. The name "stack" is based on the metaphor of the spring-loaded stack of trays you find in a self-service cafeteria. You add a tray to the stack by pushing down the trays that were already there, adding the new tray at the top of the pile. When you remove a tray, you take the one at the top of the pile--the one most recently added to the stack.



Keep talking about stacks

- So, they’re general purpose data storage devices, last in, first out. Where do we use them?
- On a lot of architectures, we use them for keeping track of where to return to, after the current function ends. That is, if main calls function 1 which then causes function 2, when each function is called, the address of the instruction the call occurs from is stored on the stack.
- We use them for passing arguments to functions: Sometimes all the arguments are put on the stack, sometimes none, and sometimes just a subset, depending on the calling method our OS/program has chosen.
- Stacks are also used for local storage. The variables which are just declared, in a function (ie. “int i;”, inside a function block), are stored locally on the stack.
- Objects with a lifetime which must survive beyond the current function returning cannot be stored on the stack.

Stack Frames

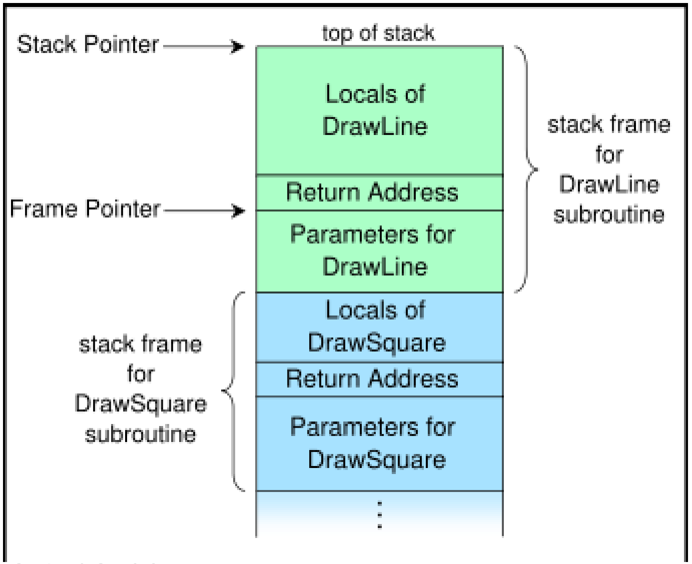
- A stack ‘frame’ refers to the component of the stack used by a particular function. In the diagram below, DrawSquare has called
- The stack pointer points to the top of the stack.
- Sometimes, a “frame pointer” or “base pointer” is used, which points to the bottom of the current stack frame. Note that the bottom green box

Which includes the

Params has actually

Been set up by the

Calling function



Actually quite a shit diagram

Heaps

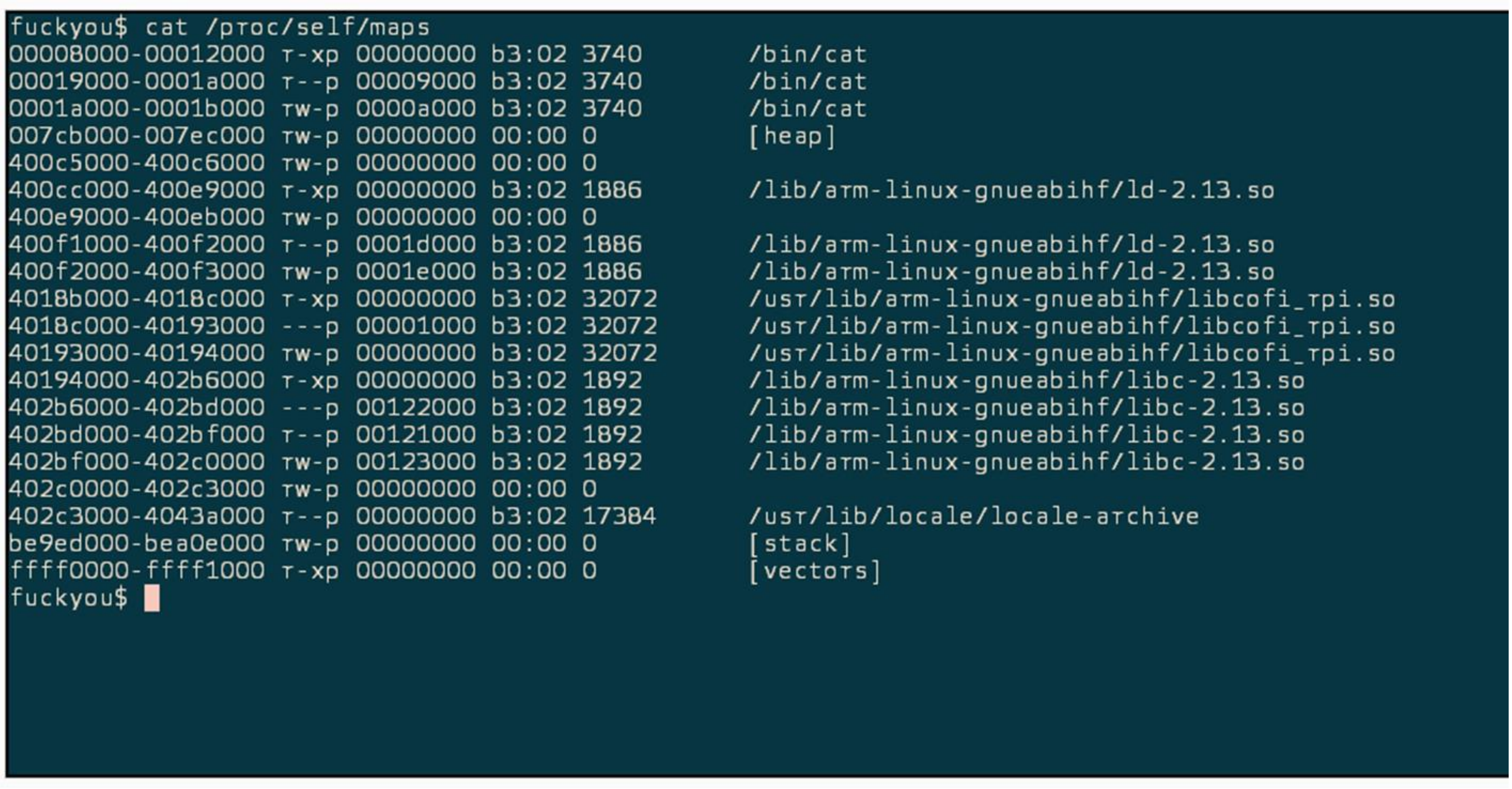
- We will be talking more about heaps later on
- Separate from stack
- Used for large allocations, and for allocations which must survive after the function returns (the stack is cleaned up)
- malloc() free() etc. provide memory on the heap for nixes.
- We can resize chunks, return them, ask for more, etc. run-time memory allocation. We don’t need to know our memory requirements at compile time.
- Process can have many heaps.
- Each thread has a heap on windahs.

Other pieces

- Permissions are generally applied to sections (ie. One set of permissions for stack (rw-), one for code (r-x).
- Text/code <!– this is where I program is. Generally (r-x)
- Data: contains the global variables and static variables that are initialized by the programmer. Generally (rw-)
- Rodata (constant, read-only (r--) section.
- Each lib is mapped into a separate text or lib section etc.
- Libs have exports symbols, which list the entry points for calling them.
- **.bss** or **bss** is used by many compilers and linkers for a part of the data segment containing statically-allocated variables represented solely by zero-valued bits initially (i.e., when execution begins). It is often referred to as the "bss section" or "bss segment".

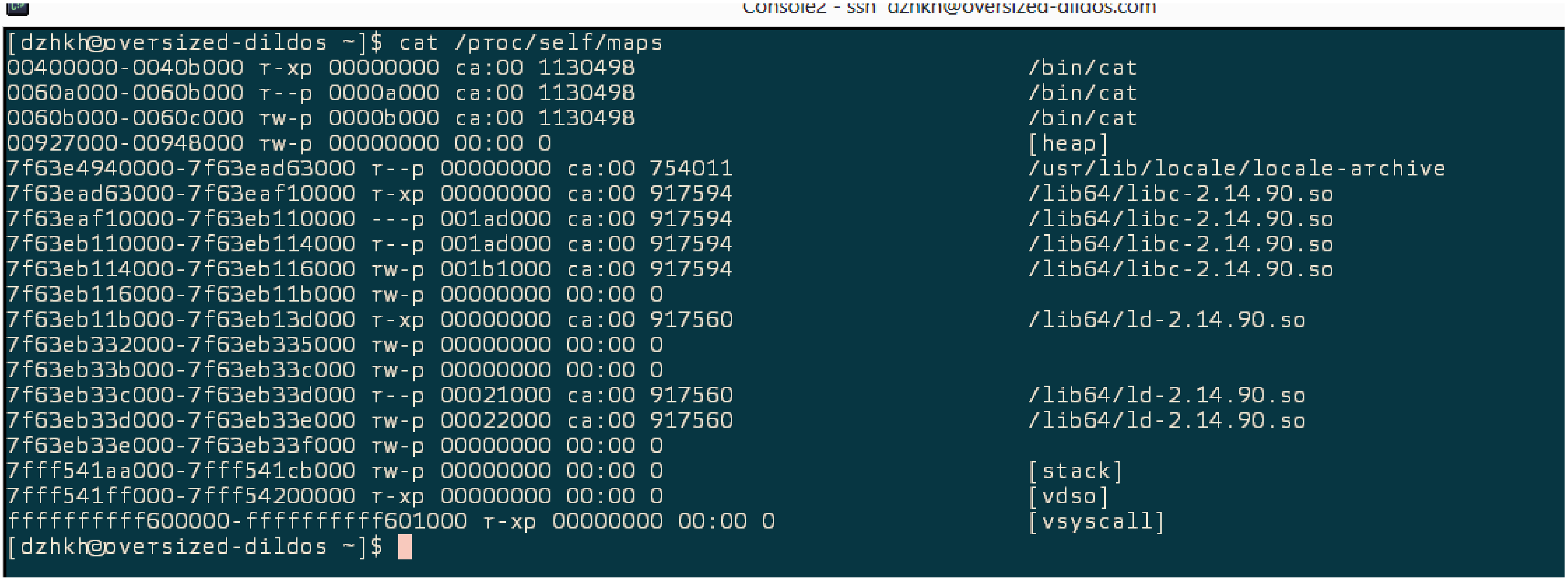
32 bit linux arm process - cat /proc/self/maps

- First cat: the actual code. Second: rodata. Third: got, data, bss. Use “maint info sect” inside GDB for more detailed info..



```
fuckyou$ cat /proc/self/maps
00008000-00012000 r-xp 00000000 b3:02 3740       /bin/cat
00019000-0001a000 r--p 00009000 b3:02 3740       /bin/cat
0001a000-0001b000 rw-p 0000a000 b3:02 3740       /bin/cat
007cb000-007ec000 rw-p 00000000 00:00 0          [heap]
400c5000-400c6000 rw-p 00000000 00:00 0
400cc000-400e9000 r-xp 00000000 b3:02 1886       /lib/arm-linux-gnueabihf/ld-2.13.so
400e9000-400eb000 rw-p 00000000 00:00 0
400f1000-400f2000 r--p 0001d000 b3:02 1886       /lib/arm-linux-gnueabihf/ld-2.13.so
400f2000-400f3000 rw-p 0001e000 b3:02 1886       /lib/arm-linux-gnueabihf/ld-2.13.so
4018b000-4018c000 r-xp 00000000 b3:02 32072      /usr/lib/arm-linux-gnueabihf/libcofi_rpi.so
4018c000-40193000 ---p 00001000 b3:02 32072      /usr/lib/arm-linux-gnueabihf/libcofi_rpi.so
40193000-40194000 rw-p 00000000 b3:02 32072      /usr/lib/arm-linux-gnueabihf/libcofi_rpi.so
40194000-402b6000 r-xp 00000000 b3:02 1892       /lib/arm-linux-gnueabihf/libc-2.13.so
402b6000-402bd000 ---p 00122000 b3:02 1892       /lib/arm-linux-gnueabihf/libc-2.13.so
402bd000-402bf000 r--p 00121000 b3:02 1892       /lib/arm-linux-gnueabihf/libc-2.13.so
402bf000-402c0000 rw-p 00123000 b3:02 1892       /lib/arm-linux-gnueabihf/libc-2.13.so
402c0000-402c3000 rw-p 00000000 00:00 0
402c3000-4043a000 r--p 00000000 b3:02 17384      /usr/lib/locale/locale-archive
be9ed000-bea0e000 rw-p 00000000 00:00 0          [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
fuckyou$ 
```

64 bit linux x86_64 example.

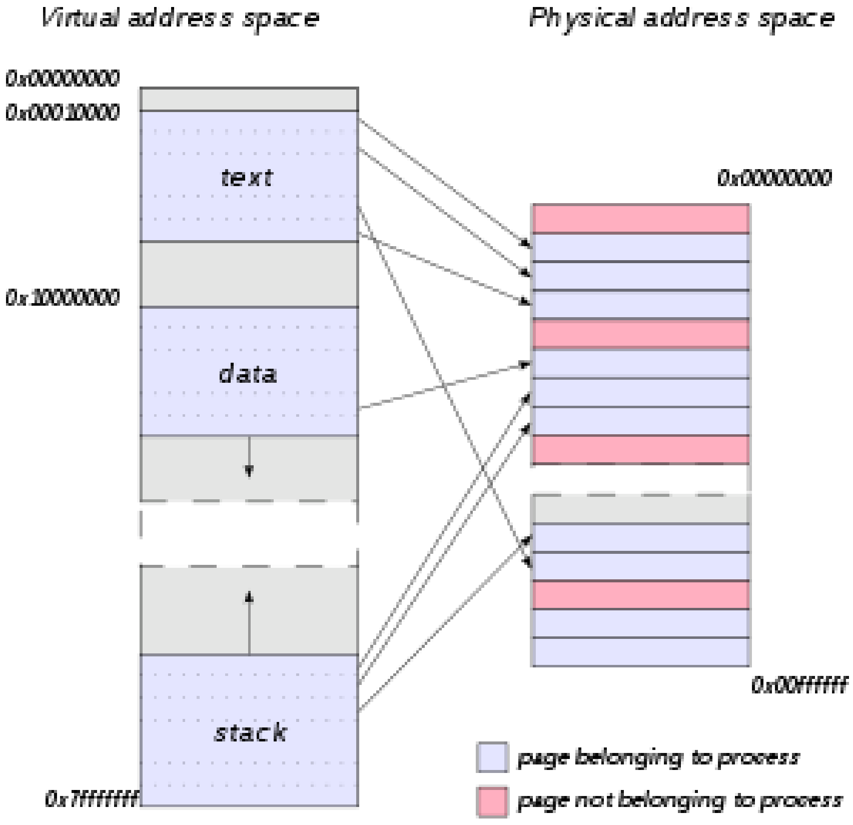


Console2 - ssh dzhkh@oversized-dildos.com

```
[dzhkh@oversized-dildos ~]$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 ca:00 1130498                            /bin/cat
0060a000-0060b000 r--p 0000a000 ca:00 1130498                            /bin/cat
0060b000-0060c000 rw-p 0000b000 ca:00 1130498                            /bin/cat
00927000-00948000 rw-p 00000000 00:00 0                                  [heap]
7f63e4940000-7f63ead63000 r--p 00000000 ca:00 754011                     /usr/lib/locale/locale-archive
7f63ead63000-7f63eaf10000 r-xp 00000000 ca:00 917594                     /lib64/libc-2.14.90.so
7f63eaf10000-7f63eb110000 ---p 001ad000 ca:00 917594                     /lib64/libc-2.14.90.so
7f63eb110000-7f63eb114000 r--p 001ad000 ca:00 917594                     /lib64/libc-2.14.90.so
7f63eb114000-7f63eb116000 rw-p 001b1000 ca:00 917594                     /lib64/libc-2.14.90.so
7f63eb116000-7f63eb11b000 rw-p 00000000 00:00 0
7f63eb11b000-7f63eb13d000 r-xp 00000000 ca:00 917560                     /lib64/ld-2.14.90.so
7f63eb332000-7f63eb335000 rw-p 00000000 00:00 0
7f63eb33b000-7f63eb33c000 rw-p 00000000 00:00 0
7f63eb33c000-7f63eb33d000 r--p 00021000 ca:00 917560                     /lib64/ld-2.14.90.so
7f63eb33d000-7f63eb33e000 rw-p 00022000 ca:00 917560                     /lib64/ld-2.14.90.so
7f63eb33e000-7f63eb33f000 rw-p 00000000 00:00 0
7fff541aa000-7fff541cb000 rw-p 00000000 00:00 0                          [stack]
7fff541ff000-7fff54200000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[dzhkh@oversized-dildos ~]$ 
```

A processes address space

- CPUs fetch both code and data from main memory.
- The following shows a sample process and the memory areas. We're looking at the virtual address space, here.



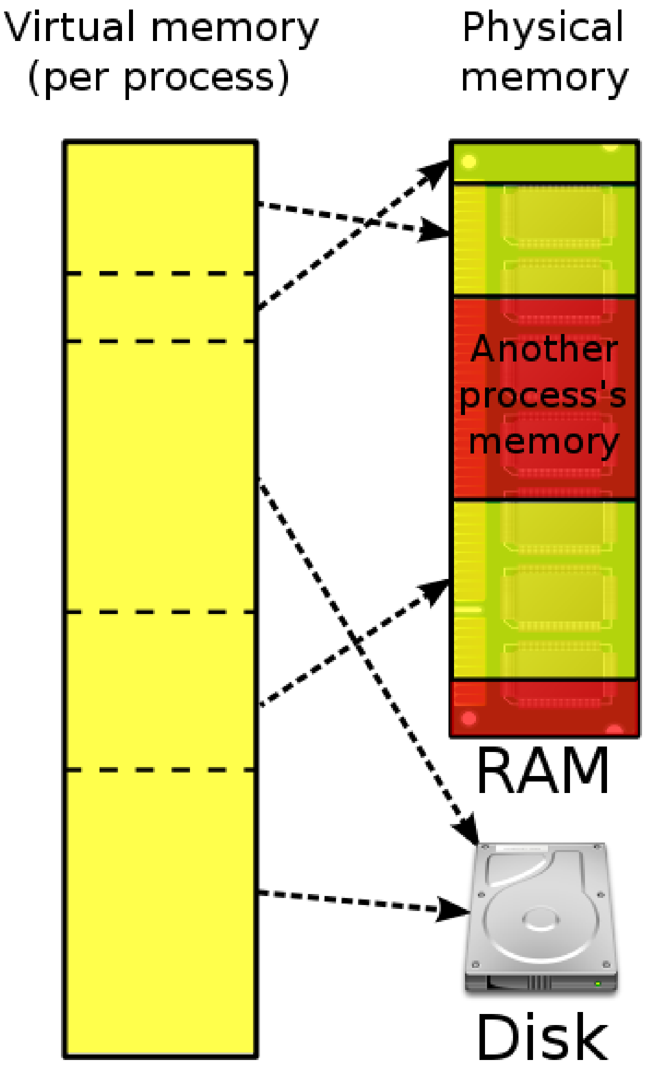
Virtual Memory

- Presents to each process a view where it has the full address space.

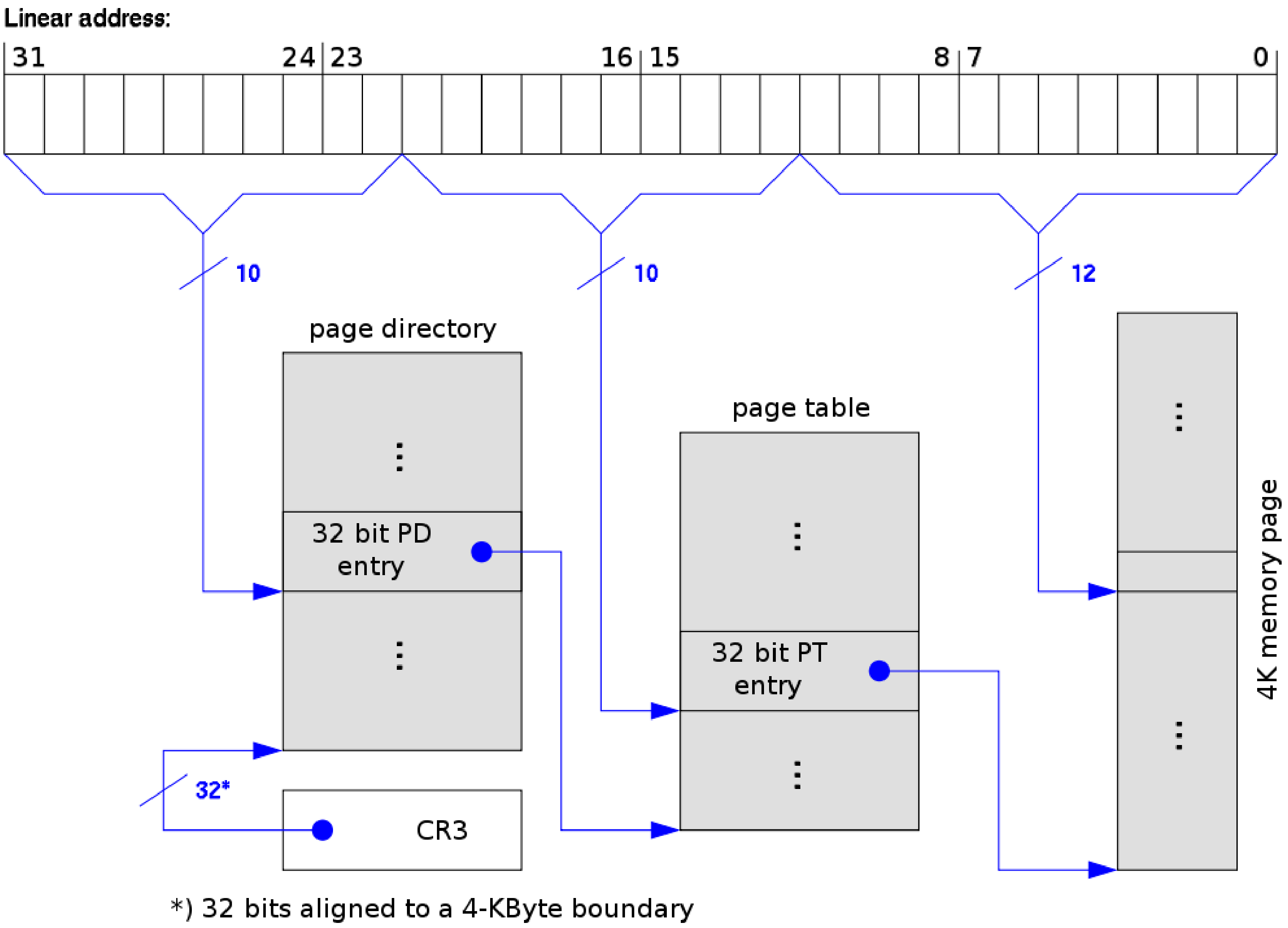
Ie. On x86, processes can address from 0 to 0xFFFFffff.

- Allows us to have bigger address range than we actually
- Many processes can have the same addreses, ie. Firefox and utorrent both have pages mapped, from their own perspective, at 0x71720000 (just an example)
- These are different physical memory pages
- Access controlled via the kernel and via hardware support for restricting access
- Virtual Memory is mapped to Physical Memory by the kernel, using page tables
- Memory is not mapped byte for byte to different physical addresses; this would be crazy, and require more memory than we actually need to manage the mapping (by far)
- Memory is mapped physical <!-> virtual via break each memory into pages.
- Pages are a fixed size per system (ie. 4KB (Grrr… KiB) or 4MB..)

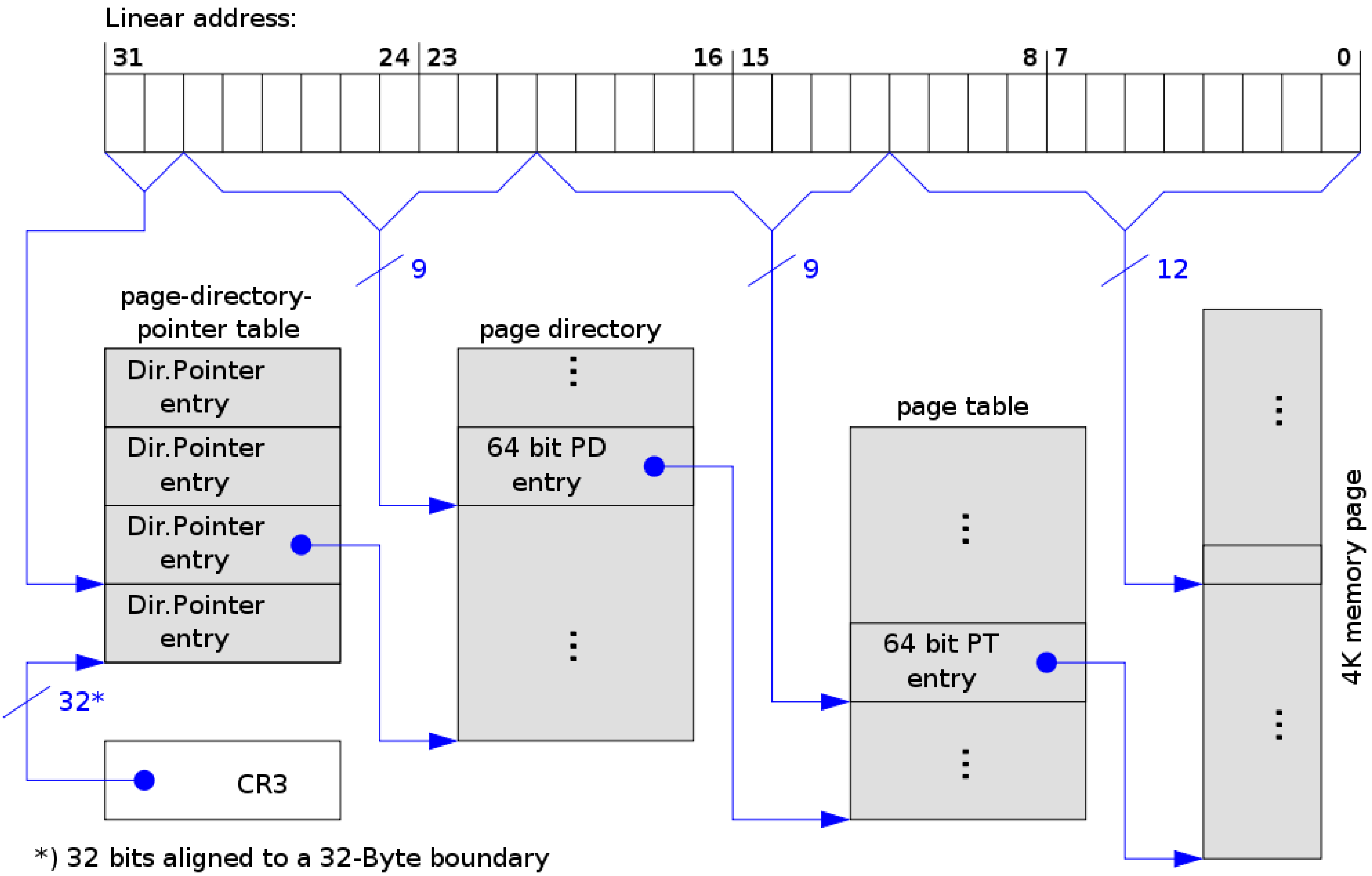
This diagram shows this nicely



Two-level page table structure in x86 architecture (without PAE or PSE).

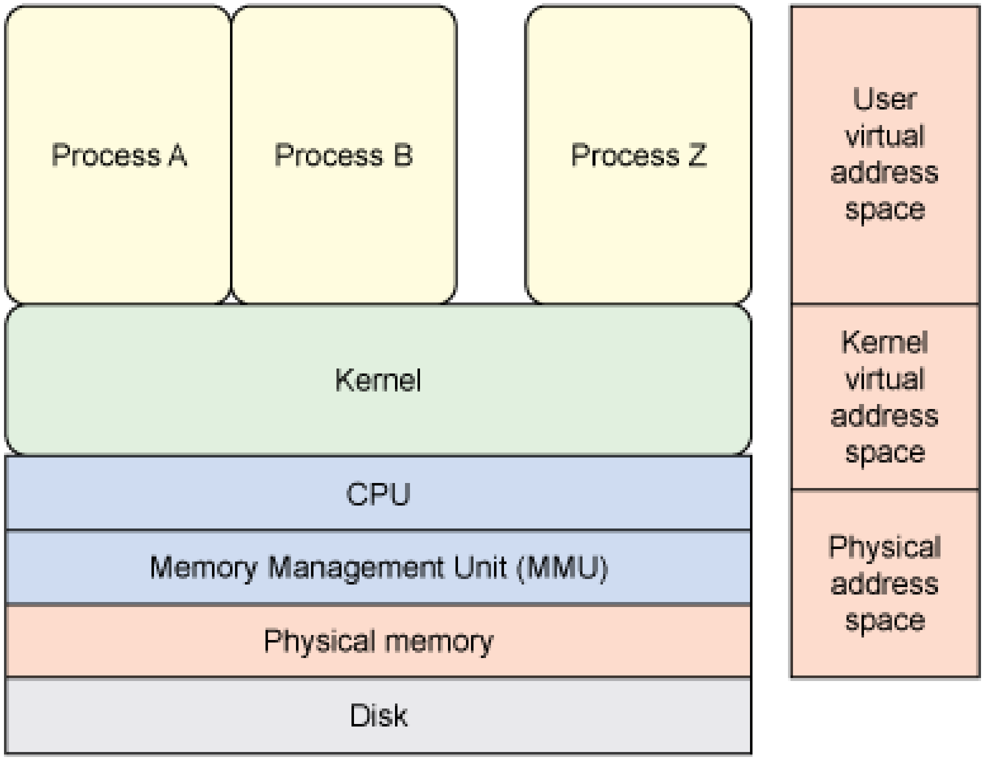


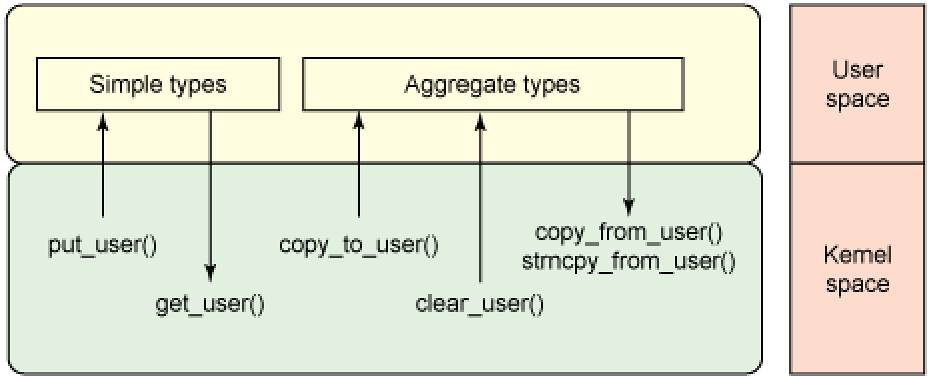
Three-level page table structure in x86 architecture (with PAE, without PSE



Kernelspace vs userspace (combine mode)

- All pastel diagrams from: http://www.ibm.com/developerworks/linux/library/l-kernel-memory-access/index.html?ca=dgr-lnxw06LXUserSpacedth-LX


Process A
Process B
Process Z
Kernel
CPU
Memory Management Unit (MMU)
Physical memory
Disk
User virtual address space
Kernel virtual address space
Physical address space



Real splits

The linux diagram here shows a

2G/2G split: the lowest 2 gig are for user processes to use, and the upper 2g Is for the kernel. Note that whilst all the different processes each can address from 0 to 2GB (0x7FFF,FFFF) being the

top address, the data is different for each address. The kernel component however is the same.

Only kernel code (when the

processor is running in supervisor

/privileged/ring 0) mode can read

The kernel data.

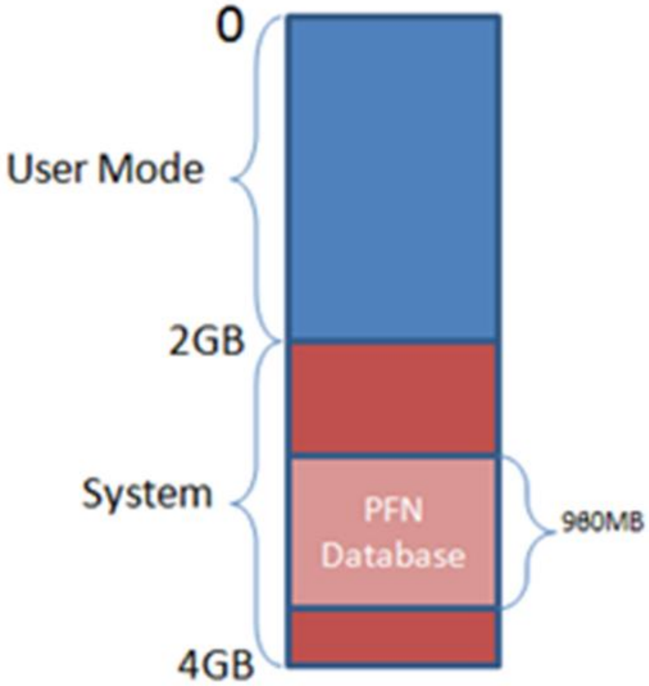
32-bit windows and modern x86 Linux now has

uses a 3/1 split: kernel address start at 3GB

(0xC0000000)

i.e. 0 to 0xBFFFffff is user space

0xC0000000 to 0xFFFFffff is kernel space



Split/Seperated address space

- Some OS/arch combos have separate address space.
- Linux 4g/4g patch
- This means that the kernel memory is not mapped into each processes address space. Separate space for kernel and user.