Virtual memory and address spaces

bytelab.codes/what-is-memory-part-1

David Vernet March 27, 2022

I remember when I was an undergraduate taking my first computer science course. We were learning about the `null` type in Java, and someone asked the (rather clumsy) TA, "I understand that you can compare an object to `null` to see if it was allocated, but what actually is `null`?" The TA turned on the projector, which rendered a picture of a rectangle onto a cloth screen in the front of the class. He responded, "Well…`null` is the area down here", and proceeded to circle the bottom of the rectangle in marker, thus immediately ruining the screen and destroying university property. He didn't go on to explain what he meant by "the area down here", and I was left wondering what `null` really was for several years after that.

What the confused student didn't realize at the time is that they were asking a very insightful and fundamental question about how computers work. To ask the question "What is `null`" is to ask, "what is memory"? To ask "what is memory" is to ask, "what is a cache, and how do CPUs coordinate to read from and write to memory?" The recursion goes on and on, with an engineer becoming more and more capable (and frankly, more and more lucrative to companies) as they learn the answers to these questions and gain a deeper understanding of how a computer actually works.

That's not to say that it's a hard requirement to learn all this stuff in order to be a capable and productive developer. To the contrary, most developers in the tech industry will have great careers viewing all of this as a black box, and there's nothing wrong with that. At the end of the day though, you as an engineer will always be limited in what you can accomplish and how far you can progress if you don't understand how memory works. Think about what capabilities truly understanding memory would afford you. Is your program slow? Perhaps you should check if you're accidentally thrashing your system by using more memory than is available on your machine. Does your product need to scale as your customer base grows? Leveraging a caching layer that can more quickly service requests to your customers is now a solution that you're able to suggest. These skills will allow you not only to command a higher salary, but also to take on leadership roles as an engineer that would otherwise be out of reach for someone who didn't have the necessary knowledge to make highly impactful engineering decisions.

Now that the sales pitch is out of the way, let's get started. We'll begin by developing a basic definition of and intuition for virtual memory in this post, and eventually dive into deeper (but still very important and interesting) concepts such as paging and caching in future posts. By the end of the series, you'll know what virtual memory is, what a process (i.e. an instance of a

running program) is, how CPUs actually read and write to and from memory, and much more. Note that this post makes an assumption that you're familiar with hexadecimal notation, and you know what a byte is. If you don't, leave a comment below and I'll write up another post that introduces them in an understandable way.

What is "virtual memory"?

It turns out that this is a deep question that could (and has) filled up many volumes of textbooks, research papers, and probably blog posts. Let's start with a simple (but conceptually accurate) model.

We can think of *virtual memory* as a very large array of data that is fast to read and write. The 0th index of the array can be called "address 0" (0x0), and the highest index of the array can be called "address 2^{64} - 1" (0xffffffffffffffff). Every instance of a running process in a system has a single instance of this virtual memory array, which we call its *virtual memory address space*, or more succinctly, its *address space*.

This is conceptually no different than an array in a programming language. Like any other array, a process can *store* values at entries in its address space, and then *read* those values at a later time. Unlike a regular array, however, a process's address space is *sparsely allocated*, meaning that it does not automatically have access to all entries of its address space, and will have (very large) "holes" where the addresses are unavailable for use. In order to *allocate* ranges of its address space and fill those holes, the process must ask the operating system to allocate memory on its behalf. For any address / entry in the address space that's available for use, we'll say that it's *allocated*. If an address has **not** been allocated and thus cannot be written to or read from, we'll say that it's *unallocated*.

Hold on, 2^{64}?? A computer doesn't have that much memory…

We said above that every process in a system has a virtual address space that's (usually) of size 2^{64}. You may be wondering, "How on earth is that possible? My computer only has 16 GB of RAM?" That's a very reasonable and astute question, and the answer is that we're talking about **virtual** memory, not **physical** memory. Virtual memory is a *process's view of memory*. Every process in a running system sees memory as an array of size 2^{64}, that it has exclusive access to. In reality, there is a much smaller array of *physical* memory that is managed by the operating system, and given to processes to fill the holes in a process's address space when requested.

But why do we need this extra layer / abstraction? Why not just have processes use physical memory directly? Well, because programming is already hard enough, and we like things to

be as simple as possible. Virtual memory gives every process the *exact* same model for memory, everywhere. If I write a program, I don't want to also have to write code to scrape through physical memory to see what's still available after other processes have already taken the memory they need.

Virtual memory is indeed a powerful concept. It provides all processes with the illusion that they have full and exclusive control over all the memory on the system. It also allows us to use a framework for running processes that *partitions* its address space into different regions, each with a specific purpose. For example (and we'll go into more detail on this in the next post), part of a process's address space consists of all of its *code*, i.e. instructions. Another part of the address space is the process's heap, i.e. space where it can dynamically allocate and use extra memory at runtime. We will go into more details on how a process is laid out in memory in the next post. For now, let's gain a bit more fundamental intuition about memory.

Everything is memory and context

We now know that a virtual memory address space is a giant, sparse array that makes up a process's view of memory. We also have some intuition for why virtual memory is critical to having a reasonable programming model. It's time for us to take our new intuition, and use it to deepen our knowledge of computers with a very important point: *Everything in software is just data and context.*

This should hopefully make some intuitive sense. When you write a program and store something in a variable, that variable has a specific purpose which ascribes meaning to what's stored there. For example, if you store the number 0 in a variable called, `num_cookies`, then you are ascribing the meaning of "number of cookies" to some value 0 that is stored somewhere in the process's address space. The data is "0", and the context is, "number of cookies". If you were storing an address in a variable (which would make it a "pointer"), the address 0 would mean, "the bottom of the address space". The data is the same, but the context gives it a completely different meaning.

Let's take a look at a real world example to apply this intuition in practice.

Compiling Code

At a high level, the job of a compiler is to translate a "human-readable" source code file into a "machine readable" binary file. The term "human readable" is a bit misleading though. Like everything else in a computer, the contents of the file are just numbers, to which we happen to ascribe the *context* of being a "human readable text file". This context ascribes a meaning of "text" to the numbers. Let me show you what I mean with an example.

Say that we had a simple program in a source file called `byte_lab.c`:

```
$ vim byte_lab.c
#include <stdio.h>

int main() {
  printf("Byte Lab is fun!\n");
  return 0;
}
```

It looks like text, but that's only because `vim` (my strongly preferred text editor) interprets the numbers (bytes) in the file as text, and presents those bytes to you as alphanumeric characters. Let's take a look at `byte_lab.c` using the `hexdump` utility, which displays the numerical contents of a file:

```
$ hexdump byte_lab.c
user@user-VirtualBox:~/experiments$ hexdump -C byte_lab.c
00000000  23 69 6e 63 6c 75 64 65  20 3c 73 74 64 69 6f 2e  |#include <stdio.|
00000010  68 3e 0a 0a 69 6e 74 20  6d 61 69 6e 28 29 20 7b  |h>..int main() {|
00000020  0a 20 20 20 20 70 72 69  6e 74 66 28 22 42 79 74  |.    printf("Byt|
00000030  65 20 4c 61 62 20 69 73  20 66 75 6e 21 5c 6e 22  |e Lab is fun!\n"|
00000040  29 3b 0a 20 20 20 20 72  65 74 75 72 6e 20 30 3b  |);.    return 0;|
00000050  0a 7d 0a                                          |.}.|
00000053
```

The right-most column should look familiar – it's the contents of `byte_lab.c` from above. The middle column shows us each the *hexadecimal representation* for each character in the file. The left-most column is the offset of each byte / character. So one-by-one, we have the following character / byte representation pair:

```
#: 0x23
i: 0x69
n: 0x6e
c: 0x63
l: 0x6c
…
0: 0x30
;: 0x3b
\n: 0x0a
}: 0x7d
\n: 0x0a
```

If you're curious as to *why* these numbers mean these specific characters, see the section below where we discuss the ASCII standard. But anyways, that's why `vim` is able to show you the textual representation of the file when you open it up. `vim` *expects* to see bytes / numbers that correspond to alphanumeric text characters, and when it does, it knows how to turn them into characters on our screen.

An interesting question that comes to mind is, “What happens if we open vim on a binary file?” Let’s find out:

```
$ vim byte_lab # opening the byte_lab binary executable file
^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^C^@>^@^A^@^@^@`^P^@^@^@^@^@^@@^@^@^@^@^@^@^@x9^@^@^@^@^@
…
@^@^@^@^@^@H^B^@^@^@^@^@^@^A^@^@^@^@^@^@^@/lib64/ld-linux-x86-64.so.2^@^@^@^@^@
…
```

There are a lot of `^@` characters which is what vim renders when it doesn’t know what character the number is supposed to represent. There are also some `^A`’s and `^B`’s which essentially also just mean that the bytes in the file don’t correspond to something meant to be read by humans. Interestingly though, we also see some human readable text (“ELF”, and “lib64/ld-linux-x86-64.so”). Why is that? Because executable, binary ELF files actually contain some human readable text as well! The files aren’t meant to be read by humans, but that doesn’t mean that they don’t contain some values in memory that, in context, are meant to be interpreted as textual characters.

An aside: ASCII

I want to briefly touch on standards and ASCII so you’re aware of how vim knows which characters correspond to which bytes. The answer is that the characters being rendered by vim are actually *encoded* as part of a *standard*. In this case, the ASCII standard. A standard is a widely accepted agreement that specifies behavior / the expectation of something in computer systems. Sometimes standards dictate the behavior and guarantees of a programming language (e.g. the “C standard” or “C++ standard”), and sometimes they specify e.g. how characters should be *encoded* (or, how they should be represented by numbers).

ASCII is a relatively simple standard that specifies the encoding for 128 characters (which only require 7 bits). We can actually see the encoding for all of the characters with `man ascii`:

```
$ man ascii
ASCII(7)                                          Linux Programmer's Manual

NAME
ascii - ASCII character set encoded in octal, decimal, and hexadecimal

DESCRIPTION
ASCII  is  the  American  Standard  Code  for  Information Interchange.  It is a 7-bit
8859-1) contain ASCII as their lower half.  The international counterpart of ASCII is I

The following table contains the 128 ASCII characters.

C program '\X' escapes are noted.
```

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0' (null character)	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D
005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL '\a' (bell)	107	71	47	G
010	8	08	BS '\b' (backspace)	110	72	48	H
011	9	09	HT '\t' (horizontal tab)	111	73	49	I
012	10	0A	LF '\n' (new line)	112	74	4A	J
013	11	0B	VT '\v' (vertical tab)	113	75	4B	K
014	12	0C	FF '\f' (form feed)	114	76	4C	L
015	13	0D	CR '\r' (carriage ret)	115	77	4D	M
016	14	0E	SO (shift out)	116	78	4E	N
017	15	0F	SI (shift in)	117	79	4F	O
020	16	10	DLE (data link escape)	120	80	50	P
021	17	11	DC1 (device control 1)	121	81	51	Q
022	18	12	DC2 (device control 2)	122	82	52	R
023	19	13	DC3 (device control 3)	123	83	53	S
024	20	14	DC4 (device control 4)	124	84	54	T
025	21	15	NAK (negative ack.)	125	85	55	U
026	22	16	SYN (synchronous idle)	126	86	56	V
027	23	17	ETB (end of trans. blk)	127	87	57	W
030	24	18	CAN (cancel)	130	88	58	X
031	25	19	EM (end of medium)	131	89	59	Y
032	26	1A	SUB (substitute)	132	90	5A	Z
033	27	1B	ESC (escape)	133	91	5B	[
034	28	1C	FS (file separator)	134	92	5C	\ '\\'
035	29	1D	GS (group separator)	135	93	5D	]
036	30	1E	RS (record separator)	136	94	5E	^
037	31	1F	US (unit separator)	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n

057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

And we see that every character has a specific *numerical encoding*. If we go back to our example above with `hexdump`, we'll see that all of the character / byte combinations printed by `hexdump` match the encodings of the ASCII standard here.

Wrapping up

In this post we learned a few very important concepts:

1. Every process has their own view of virtual memory as a sparse array of storage going from 0, to 2^{64} - 1.
2. A process fills holes in its virtual memory address space by asking the operating system for memory, which itself is responsible for managing the physical memory on the system, and giving it to processes upon request.
3. At the end of the day, everything in computers is about data and context.
4. We sometimes use *standards* to ensure that everyone views bytes in a specific context in the same way.

There is a lot more to unpack and learn about memory. In the next post, we'll take a closer look at how a process's address space is laid out in memory. We'll learn about what loaders are (i.e. how your program actually gets run), and that processes are composed of something called segments. We'll also learn about the heap, and the stack. If that sounds interesting, tune in!

One more thing though before we sign off – we still haven't answered the question of "What actually is `null`?" We'll answer it now: `null` is simply a term that represents address 0 in a process's address space. That's right, it's just a plain old memory address representing the

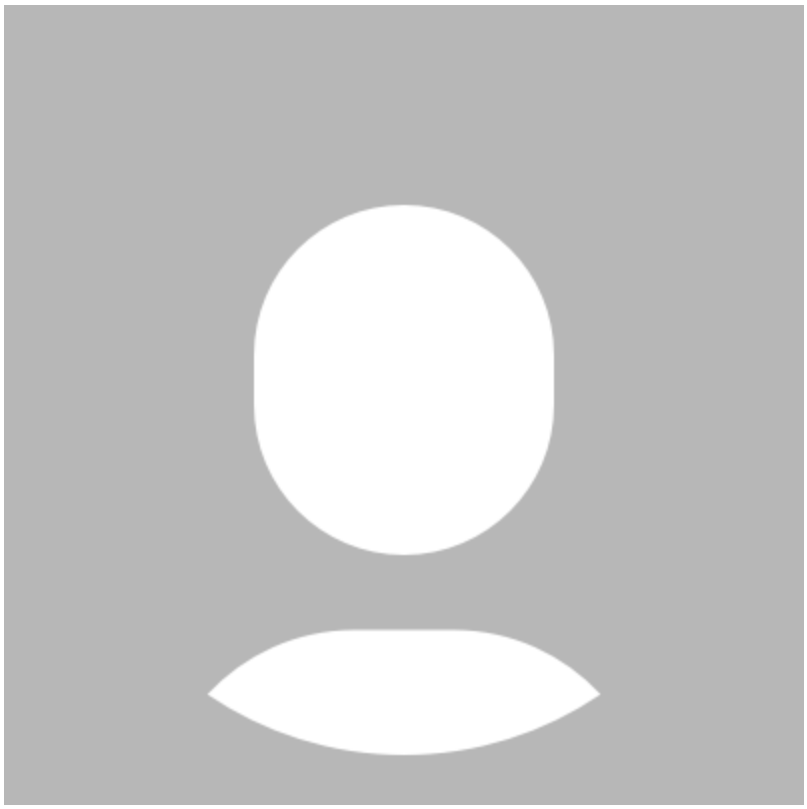
very bottom / beginning of a process's address space. This address is actually no different than any other address, but the convention in every operating system that I've ever come across is to ensure that memory at the `null` / `NULL` / `nullptr` address is **never allocated**, so accessing it will result in the process crashing.

Disclaimer: null in Java actually does have an abstract / semantic meaning which is specific to the language. The *spirit* of `null` though is as described above, and was what the TA was trying to convey!

As always, if you have any questions or suggestions, please leave them in the comments below. See you all next time!

4 Comments

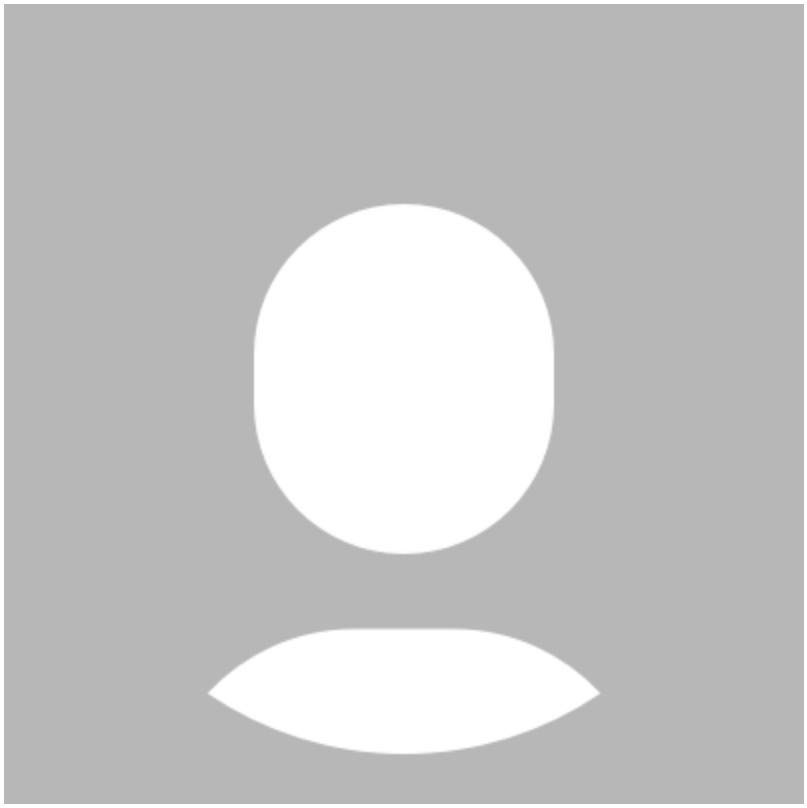
to join the conversation.



voidboy 4 years ago

Thanks David for your article. Looks like there is an all world yet to know behind virtual memory ! We created so much abstraction for something which on first hand looks simple. As you stated, "Everything is memory and context" and I think that as a computer engineer, knowing how a computer works is invaluable. I guess that's how you develop what people call "good taste".

♥ 1



Thanks for your comment, @voidboy!

>We created so much abstraction for something which on first hand looks simple.

Absolutely! As we continue to learn more and more about memory, we'll see that it's actually abstractions, on top of abstractions, on top of abstractions. Even in the kernel, there are many abstractions across various parts of the memory management subsystem. There is an old saying in the systems world: If you want to build something, just add another layer of indirection. That certainly applies to memory :-)

>I think that as a computer engineer, knowing how a computer works is invaluable. I guess that's how you develop what people call "good taste".

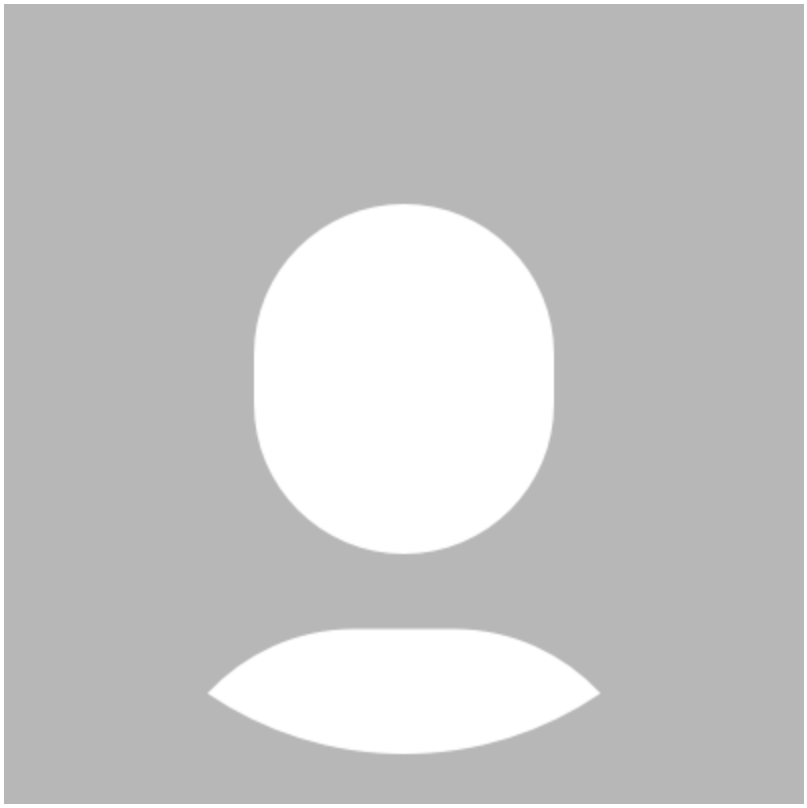
Hear, hear! I feel the same way, which is why I enjoy writing articles for Byte Lab. Please let me know if there are any topics in particular that you'd like me to cover, and thanks again for reading!

♥ 1

mohamed 2 years ago

great article

♡ 0



Astra 5 months ago

So cool, 😊

♡ 0

Powered by Cove