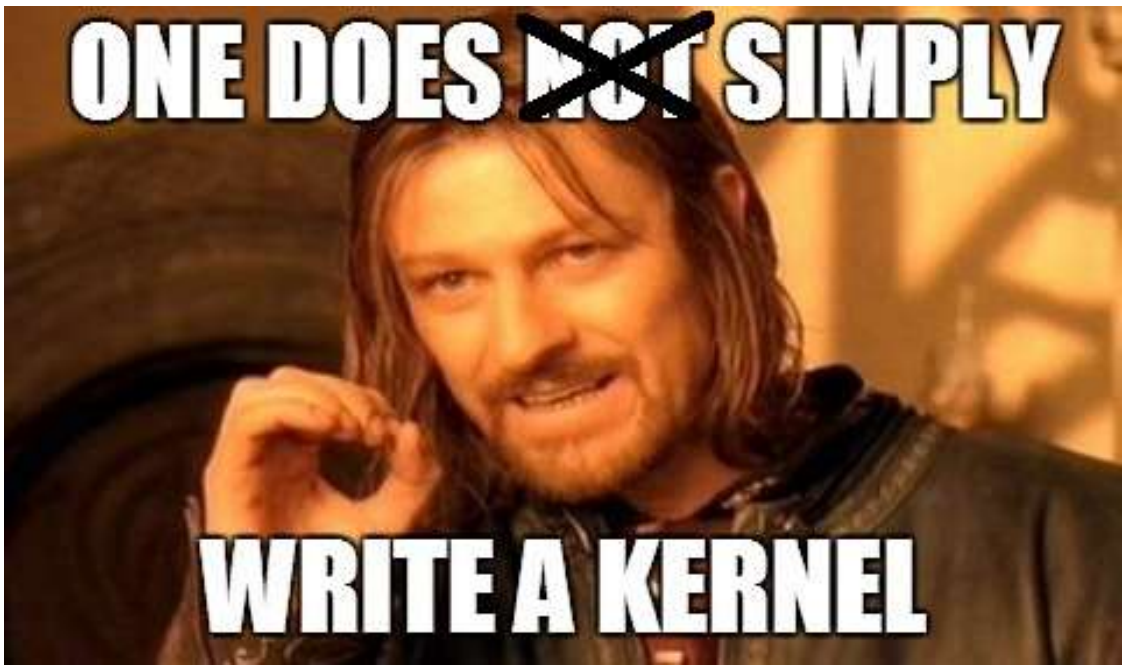
Kernels 101 – Let's write a Kernel

Hello World,

Let us write a simple kernel which could be loaded with the GRUB bootloader on an x86 system. This kernel will display a message on the screen and then hang.



How does an x86 machine boot

Before we think about writing a kernel, let's see how the machine boots up and transfers control to the kernel:

Most registers of the x86 CPU have well defined values after power-on. The Instruction Pointer (EIP) register holds the memory address for the instruction being executed by the processor. EIP is hardcoded to the value **0xFFFFFFF0**. Thus, the x86 CPU is hardwired to begin execution at the physical address 0xFFFFFFF0. It is in fact, the last 16 bytes of the 32-bit address space. This memory address is called reset vector.

Now, the chipset's memory map makes sure that 0xFFFFFFF0 is mapped to a certain part of the BIOS, not to the RAM. Meanwhile, the BIOS copies itself to the RAM for faster access. This is called **shadowing**. The address 0xFFFFFFF0 will contain just a jump instruction to the address in memory where BIOS has copied itself.

Thus, the BIOS code starts its execution. BIOS first searches for a bootable device in the configured boot device order. It checks for a certain magic number to determine if the device is bootable or not. (whether bytes 511 and 512 of first sector are **0xAA55**)

Once the BIOS has found a bootable device, it copies the contents of the device's first sector into RAM starting from physical address **0x7c00**; and then jumps into the address and executes the code just loaded. This code is called the **bootloader**.

The bootloader then loads the kernel at the physical address **0x100000**. The address 0x100000 is used as the start-address for all big kernels on x86 machines.

All x86 processors begin in a simplistic 16-bit mode called **real mode**. The GRUB bootloader makes the switch to 32-bit **protected mode** by setting the lowest bit of `CR0` register to **1**. Thus the kernel loads in 32-bit protected mode.

Do note that in case of linux kernel, GRUB detects linux boot protocol and loads linux kernel in real mode. Linux kernel itself makes the switch (//github.com/torvalds/linux/blob/949163015ce6fdb76a5e846a3582d3c40c23c001/arch/x86/boot/main.c#L181) to protected mode.

What all do we need?

* An x86 computer (of course)
* Linux
* NASM assembler (//www.nasm.us/)
* gcc
* ld (GNU Linker)
* grub

Source Code

Source code is available at my Github repository - mkernel (//github.com/arjun024/mkernel)

The entry point using assembly

We like to write everything in C, but we cannot avoid a little bit of assembly. We will write a small file in x86 assembly-language that serves as the starting point for our kernel. All our assembly file will do is invoke an external function which we will write in C, and then halt the program flow.

How do we make sure that this assembly code will serve as the starting point of the kernel?

> We will use a linker script that links the object files to produce the final kernel executable. (more explained later) In this linker script, we will explicitly specify that we want our binary to be loaded at the address 0x100000. This address, as I have said earlier, is where the kernel is expected to be. Thus, the bootloader will take care of firing the kernel's entry point.

Here's the assembly code:

```
;;kernel.asm
bits 32                 ;nasm directive - 32 bit
section .text

global start
extern kmain            ;kmain is defined in the c file

start:
  cli                   ;block interrupts
  mov esp, stack_space  ;set stack pointer
  call kmain
  hlt                   ;halt the CPU

section .bss
resb 8192               ;8KB for stack
stack_space:
```

The first instruction `bits 32` is not an x86 assembly instruction. It's a directive to the NASM assembler that specifies it should generate code to run on a processor operating in 32 bit mode. It is not mandatorily required in our example, however is included here as it's good practice to be explicit.

The second line begins the text section (aka code section). This is where we put all our code.

`global` is another NASM directive to set symbols from source code as global. By doing so, the linker knows where the symbol `start` is; which happens to be our entry point.

`kmain` is our function that will be defined in our `kernel.c` file. `extern` declares that the function is declared elsewhere.

Then, we have the `start` function, which calls the `kmain` function and halts the CPU using the `hlt` instruction. Interrupts can awake the CPU from an `hlt` instruction. So we disable interrupts beforehand using `cli` instruction. `cli` is short for clear-interrupts.

We should ideally set aside some memory for the stack and point the stack pointer (`esp`) to it. However, it seems like GRUB does this for us and the stack pointer is already set at this point. But, just to be sure, we will allocate some space in the BSS section and point the stack pointer to the beginning of the allocated memory. We use the `resb` instruction which reserves memory given in bytes. After it, a label is left which will point to the edge of the reserved piece of memory. Just before the `kmain` is called, the stack pointer (`esp`) is made to point to this space using the `mov` instruction.

The kernel in C

In `kernel.asm`, we made a call to the function `kmain()`. So our C code will start executing at `kmain()`:

```
/*
*  kernel.c
*/
void kmain(void)
{
	const char *str = "my first kernel";
	char *vidptr = (char*)0xb8000;  //video mem begins here.
	unsigned int i = 0;
	unsigned int j = 0;

	/* this loops clears the screen
	* there are 25 lines each of 80 columns; each element takes 2 bytes */
	while(j < 80 * 25 * 2) {
		/* blank character */
		vidptr[j] = ' ';
		/* attribute-byte - light grey on black screen */
		vidptr[j+1] = 0x07;
		j = j + 2;
	}

	j = 0;

	/* this loop writes the string to video memory */
	while(str[j] != '\0') {
		/* the character's ascii */
		vidptr[i] = str[j];
		/* attribute-byte: give character black bg and light grey fg */
		vidptr[i+1] = 0x07;
		++j;
		i = i + 2;
	}
	return;
}
```

All our kernel will do is clear the screen and write to it the string “my first kernel”.

First we make a pointer `vidptr` that points to the address **0xb8000**. This address is the start of video memory in protected mode. The screen's text memory is simply a chunk of memory in our address space. The memory mapped input/output for the screen starts at 0xb8000 and supports 25 lines, each line contain 80 ascii characters.

Each character element in this text memory is represented by 16 bits (2 bytes), rather than 8 bits (1 byte) which we are used to. The first byte should have the representation of the character as in ASCII. The second byte is the `attribute-byte`. This describes the formatting of the character including attributes such as color.

To print the character `s` in green color on black background, we will store the character `s` in the first byte of the video memory address and the value 0x02 in the second byte.
`0` represents black background and `2` represents green foreground.

Have a look at table below for different colors:

```
0 - Black, 1 - Blue, 2 - Green, 3 - Cyan, 4 - Red, 5 - Magenta, 6 - Brown, 7 - Light Grey, 8 - Dark Grey,
9 - Light Blue, 10/a - Light Green, 11/b - Light Cyan, 12/c - Light Red, 13/d - Light Magenta, 14/e - Ligh
t Brown, 15/f – White.
```

In our kernel, we will use light grey character on a black background. So our attribute-byte must have the value 0x07.

In the first while loop, the program writes the blank character with 0x07 attribute all over the 80 columns of the 25 lines. This thus clears the screen.

In the second while loop, characters of the null terminated string "my first kernel" are written to the chunk of video memory with each character holding an attribute-byte of 0x07.

This should display the string on the screen.

The linking part

We will assemble `kernel.asm` with NASM to an object file; and then using GCC we will compile `kernel.c` to another object file. Now, our job is to get these objects linked to an executable bootable kernel.

For that, we use an explicit linker script, which can be passed as an argument to `ld` (our linker).

```
/*
*  link.ld
*/
OUTPUT_FORMAT(elf32-i386)
ENTRY(start)
SECTIONS
 {
   . = 0x100000;
   .text : { *(.text) }
   .data : { *(.data) }
   .bss  : { *(.bss)  }
 }
```

First, we set the **output format** of our output executable to be 32 bit Executable and Linkable Format (//elinux.org/Executable_and_Linkable_Format_(ELF)) (ELF). ELF is the standard binary file format for Unix-like systems on x86 architecture.

ENTRY takes one argument. It specifies the symbol name that should be the entry point of our executable.

SECTIONS is the most important part for us. Here, we define the layout of our executable. We could specify how the different sections are to be merged and at what location each of these is to be placed.

Within the braces that follow the SECTIONS statement, the period character (.) represents the **location counter**.
The location counter is always initialized to 0x0 at beginning of the SECTIONS block. It can be modified by assigning a new value to it.

Remember, earlier I told you that kernel's code should start at the address 0x100000. So, we set the location counter to 0x100000.

Have look at the next line **.text : { *(.text) }**

The asterisk (*) is a wildcard character that matches any file name. The expression `*(.text)` thus means all `.text` input sections from all input files.

So, the linker merges all text sections of the object files to the executable's text section, at the address stored in the location counter. Thus, the code section of our executable begins at 0x100000.

After the linker places the text output section, the value of the location counter will become
0x1000000 + the size of the text output section.

Similarly, the data and bss sections are merged and placed at the then values of location-counter.

Grub and Multiboot

Now, we have all our files ready to build the kernel. But, since we like to boot our kernel with the GRUB bootloader (//www.gnu.org/software/grub/), there is one step left.

There is a standard for loading various x86 kernels using a boot loader; called as **Multiboot specification**.

GRUB will only load our kernel if it complies with the Multiboot spec (//www.gnu.org/software/grub/manual/multiboot/multiboot.html).

According to the spec, the kernel must contain a header (known as Multiboot header) within its first 8 KiloBytes.

Further, This Multiboot header must contain 3 fields that are 4 byte aligned namely:

- a **magic** field: containing the magic number **0x1BADB002**, to identify the header.
- a **flags** field: We will not care about this field. We will simply set it to zero.
- a **checksum** field: the checksum field when added to the fields 'magic' and 'flags' must give zero.

So our `kernel.asm` will become:

```
;;kernel.asm

;nasm directive - 32 bit
bits 32
section .text
        ;multiboot spec
        align 4
        dd 0x1BADB002            ;magic
        dd 0x00                  ;flags
        dd - (0x1BADB002 + 0x00) ;checksum. m+f+c should be zero

global start
extern kmain            ;kmain is defined in the c file

start:
  cli                   ;block interrupts
  mov esp, stack_space  ;set stack pointer
  call kmain
  hlt                   ;halt the CPU

section .bss
resb 8192               ;8KB for stack
stack_space:
```

The **dd** defines a double word of size 4 bytes.

Building the kernel

We will now create object files from `kernel.asm` and `kernel.c` and then link it using our linker script.

```
nasm -f elf32 kernel.asm -o kasm.o
```

will run the assembler to create the object file kasm.o in ELF-32 bit format.

```
gcc -m32 -c kernel.c -o kc.o
```

The '-c ' option makes sure that after compiling, linking doesn't implicitly happen.

```
ld -m elf_i386 -T link.ld -o kernel kasm.o kc.o
```

will run the linker with our linker script and generate the executable named **kernel**.

Configure your grub and run your kernel

GRUB requires your kernel to be of the name pattern `kernel-<version>`. So, rename the kernel. I renamed my kernel executable to kernel-701.

Now place it in the **/boot** directory. You will require superuser privileges to do so.

In your GRUB configuration file `grub.cfg` you should add an entry, something like:

```
title myKernel
        root (hd0,0)
        kernel /boot/kernel-701 ro
```

Don't forget to remove the directive `hiddenmenu` if it exists.

Reboot your computer, and you'll get a list selection with the name of your kernel listed.

Select it and you should see:

That's your kernel!!

PS:

* It's always advisable to get yourself a virtual machine for all kinds of kernel hacking. * To run this on **grub2** which is the default bootloader for newer distros, your config should look like this:

```
menuentry 'kernel 701' {
        set root='hd0,msdos1'
        multiboot /boot/kernel-701 ro
}
```

* Also, if you want to run the kernel on the `qemu` emulator instead of booting with GRUB, you can do so by:

```
qemu-system-i386 -kernel kernel
```

Also, see the next article in the Kernel series:
Kernels 201 - Let's write a Kernel with keyboard and screen support (http://arjunsreedharan.org/post/99370248137/kernels-201-lets-write-a-kernel-with-keyboard-and)

References and Thanks

1. wiki.osdev.org (//wiki.osdev.org/)
2. osdever.net (//www.osdever.net/)
3. Multiboot spec (http://www.gnu.org/software/grub/manual/multiboot/multiboot.html)
4. Thanks to Rubén Laguna (//rubenlaguna.com/) from comments for grub2 config

3:12 PM 14th avril 2014 (https://arjunsreedharan.org/post/82710718100/kernels-101-lets-write-a-kernel) 65 notes (https://arjunsreedharan.org/post/82710718100/kernels-101-lets-write-a-kernel#notes)

kernel	operating system	c	asm	assembly	x86	cpu	booting
programming	bootloader	grub	nasm	ld	gcc	technology	

Arjun Sreedharan Mod ➚ Livid • 9 years ago

Hey, i am actually planning to write another post with addition of a keyboard driver among others. :)

39 ^ | v • Reply • Share ›

boris ➚ Arjun Sreedharan • 9 years ago

I would be extremely enthusiastic about such a post! :]

9 ^ | v • Reply • Share ›

guybrush ➚ Arjun Sreedharan • 8 years ago

Please do write about keyboard input, or share a tip on where to look next.

^ | v • Reply • Share ›

Arjun Sreedharan Mod ➚ guybrush • 8 years ago

here's it - at last:

http://arjunsreedharan.org/...

6 ^ | v • Reply • Share ›

capcase • 9 years ago

I am interested in a follow-up post about how to write your own bootloader. thanks!

11 ^ | v • Reply • Share ›

Terrence Andrew Davis • 9 years ago • edited

I wrote a kernel and compiler.
http://www.templeos.org/Tem...
And other tools, and bootloaders, an editor, and a graphic's library.

24 ^ | v 4 • Reply • Share ›

Anon ➚ Terrence Andrew Davis • 9 years ago

Good work

4 ^ | v • Reply • Share ›

Anon ➚ Terrence Andrew Davis • 9 years ago

Please leave troll.

8 ^ | v 4 • Reply • Share ›

Vidar Hokstad ➚ Anon • 9 years ago

He is known to have mental problems, but why do you feel the need to make a comment like that when he for once makes an on-topic, relevant comment without anything offensive?

21 ^ | v • Reply • Share ›

Miles Rout ➚ Vidar Hokstad • 6 years ago

Because he's still a despicable person. He's not welcome in the OS development community.

^ | v 5 • Reply • Share ›

Meemah ➚ Miles Rout • 5 years ago

Is that a joke? You don't speak for the OSdev community in the slightest, meanwhile terry has numerous very real accomplishments and TempleOS is an incredibly good OS especially for trying out various things without 20 layers of abstraction getting in the way.

Not to mention how you're insulting someone and calling them despicable for having (provably, and obviously) severe mental issues even though they have managed to contribute to society despite them.

4 ^ | v 1 • Reply • Share ›

AnonReply ➚ Anon • 9 years ago

He is not a troll but a religious fanatic with really good programing skills. I know him from other sites.

9 ^ | v • Reply • Share ›

Anon ➚ Anon • 8 years ago

you are a little cock sucker, you fucking trolling nigger

1 ^ | v 4 • Reply • Share ›

krzysiunet ➚ Anon • 5 years ago

I didn't know he's a racist. I'm far from defending such person, but err, it actually was you, Anon, who promotes his ideas. Even in negative context, I don't see how posting his slurs helps anybody.

^ | v • Reply • Share ›

Nathan Masi ➚ Terrence Andrew Davis • 6 years ago

amazing!!!!

2 ^ | v • Reply • Share ›

Anon the Second ➚ Terrence Andrew Davis • 2 years ago

HOLY FUCK
I cannot believe that I came across this comment.
Rest In Peace. TempleOS lives on.

^ | v • Reply • Share ›

Dario Ivan Martinez Faudoa ➚ Terrence Andrew Davis • 3 years ago

I can not believe I ran into this comment. I laugh at all the comments who had no idea who you were. RIP Terry, You actually inspired me to look into writing my own kernel. For those who do not know Terry was the best fucking programmer in the world. His whole story is interesting. Sure he was mentally Ill and a schizo but he made his mark in the IT world. TempleOS is a gods gift translated to us by Terry Davis. If anyone sees this I encourage you to watch DowntheRabbitshole video on Terry Davis almost 2 Million Views. This post is History and I cant believe I found it.

^ | v • Reply • Share ›

Arjun Sreedharan Mod ➚ Dario Ivan Martinez Faudoa • 3 years ago

Yeah, poor guy. Rest in peace.

^ | v • Reply • Share ›

Arjun Menon • 9 years ago • edited

How does the CPU begin execution at physical address 0xFFFFFFF0 if you have less than 4 GiB of main memory? Suppose you had a system with 256 MiB of main memory – what physical address range will these 2^28 bytes of memory correspond to? (Are physical addresses mapped in reverse order?) In addition, what subsystem is responsible for copying BIOS and placing the jump instruction at 0xFFFFFFF0 before BIOS starts executing? By the way, thanks for writing this piece!

4 ^ | v • Reply • Share ›

James Cooper ➚ Arjun Menon • 9 years ago

Every address is just a number that is broadcast on the address bus. What is listening to this address depends on what is connected. In a 32-bit memory space, the upper memory addresses are not mapped to RAM, but rather to memory mapped hardware. The BIOS ROM will also be mapped in here. Likewise for older 16-bit processors, where the video RAM and BIOS were mapped into parts of the memory between 640KB and 1MB.

For more info:
http://en.wikipedia.org/wik...

http://en.wikipedia.org/wik...

http://wiki.osdev.org/Memor...

4 ^ | v • Reply • Share ›

James Cooper ➚ James Cooper • 9 years ago

I forgot to elaborate that this is only for physical linear memory addresses. The x86 processor has many different addressing modes and those that are used by your typical 32-bit or 64-bit operating system (e.g. protected mode) will allow for things like virtual addresses, where the logical memory addresses referred to in a program are mapped through descriptor tables in the processor to some other physical address. But during initial boot, the processor will start in real mode. There are many layers of compatibility in place so that code from the original 8086 can still function on a modern x86 processor. It makes writing a bootloader and OS kernel quite an experience!

1 ^ | v • Reply • Share ›

Arjun Sreedharan Mod ➚ Arjun Menon • 9 years ago

Since RAM is faster than ROM, BIOS copies itself to the RAM. This is called `shadowing`. So on translation, the address [0xFFFFFFF0] could map to a BIOS memory block.

1 ^ | v • Reply • Share ›

Rubén Laguna ➚ Arjun Menon • 9 years ago

I found an excellent article that explains it. http://duartes.org/gustavo/...

Basically the north bridge chip (part of the chipset of the motherboard) routes memory requests, most of the time it routes to RAM modules but certain address are used for communication with PCI card, video memory, bios flash, etc. It's all on the link above.

^ | v • Reply • Share ›

Budi Mulyawan ➚ Arjun Menon • 9 years ago

I believe this is virtual address space http://en.wikipedia.org/wik... which get mapped into physical ram

^ | v • Reply • Share ›

James Cooper ➚ Budi Mulyawan • 9 years ago

No, virtual memory only exists in protected mode. The processor starts in real mode for backwards compatibility.

1 ^ | v • Reply • Share ›

Yatao Li ➚ James Cooper • 9 years ago

I think that will depend on the processor and the motherboard. The memory controller is linked to the memory banks, but there might be a multiplexer to wire the memory bus somewhere else, so that all accessible things appear on a same address line. (Now perhaps all these are integrated on the chip)

The name of this is not "virtual memory" but "unified addressing" imo.

Of course X86 has independent addresses for I/O ports, unlike some ARM chips.

^ | v • Reply • Share ›

vikrantc14 • 8 years ago

Awesome! First step to my life's aim... You are my Guru from now on.

3 ^ | v • Reply • Share ›

anees • 9 years ago

super duper post! hardly few people explain such stuff these days. Thanks a ton. Anxiously waiting for the next post.

3 ^ | v • Reply • Share ›

John Youi • 5 years ago

Cool, I start wondering about how the characters being painted without kernel code? who decides their fontface and size?

2 ^ | v • Reply • Share ›

Ciro Santilli • 5 years ago

I've also uploaded several x86 bare metal examples at: https://github.com/cirosant...

2 ^ | v • Reply • Share ›

joe • 8 years ago

u are the best man :)

2 ^ | v • Reply • Share ›

amr • 3 years ago • edited

sorry the code was not readable , can you make all text white on background black

edit: never mind , it was my damn browser that was acting funny

1 ^ | v • Reply • Share ›

Nathan Masi • 5 years ago

how do i turn this into 64 bit?

1 ^ | v • Reply • Share ›

Nathan Masi • 6 years ago

just comes up with a blank screen
I've got all the exact code from the tutorial
and the boot code i have is:
submenu '[CONFIDENTIAL]' {
menuentry 'version [CONFIDENTIAL]' {
recordfail
set root=(hd0,gpt7)
multiboot /boot/[CONFIDENTIAL]-899 ro
boot
}
}
and ofcourse all the "[CONFIDENTIAL]"'s are replace with what it actually is
I've tried:
removing the boot command
change which root i put it in
check for any errors but none come up
run this through the grub command line
but once I've gotten so far. it just shows... nothing
please help

1 ^ | v • Reply • Share ›

Guest1 ➚ Nathan Masi • 6 years ago

See this
https://github.com/arjun024...

1 ^ | v • Reply • Share ›

Nathan Masi ➚ Guest1 • 6 years ago

still doesnt work
cannot find disk '(hd0,msdos1)'
I've tried changing that afterwards to all the other options but it just shows a blank screen

1 ^ | v • Reply • Share ›

Zafer Balkan • 6 years ago

I really appreciate it. After wiorking as an industrial engineer I began going deep into CS. When I had taken some courses on Operating Systems, I thougth it would be a great thing to write a kernel on my

own but it was a big project for me. I didn't know it was just this easy!

Now I think about the KNOX environment of Samsung devices. It has a real-time kernel protection system. But I think i might try a protection-first kernel, with the help of these architectures. It might have some tradeoffs with performance but it's a necessary thing I believe.

Thanks for the enlightening post.

1 ^ | v • Reply • Share ›

saurabh kumar gupta • 7 years ago

I am hitting following linking error, tried couple of thing but couldn't succeed.
1>ld -m elf_i386 -T link_target.ld -o kernel kasm.o kc.o
error:
ld: unrecognised emulation mode: elf_i386
Supported emulations: i386pe

2>ld -m i386pe -T link_target.ld -o kernel kasm.o kc.o
error: C:\MinGW\bin\ld.exe: cannot perform PE operations on non PE output file 'kernel'.

3>ld --oformat elf32-i386 -T link_target.ld -o kernel kasm.o kc.o
error:C:\MinGW\bin\ld.exe: cannot perform PE operations on non PE output file 'kernel'.

Please help me out.

1 ^ | v • Reply • Share ›

saurabh kumar gupta ➚ saurabh kumar gupta • 7 years ago

waiting...

^ | v • Reply • Share ›

Arjun Sreedharan Mod ➚ saurabh kumar gupta • 7 years ago

Please get yourself a linux distro or at least a VM.
PE is the default executable format on windows NT, and therefore the target format for mingw. PE is too much trouble.

^ | v • Reply • Share ›

Manoj • 9 years ago

Great post! I had already tried this but I am eager to know how to write keyboard driver.

1 ^ | v • Reply • Share ›

Shi Wei • 9 years ago

interesting,like to learn more

1 ^ | v • Reply • Share ›

Kieron • 9 years ago

In the screen-clearing section, I'm having trouble understanding how the full screen is cleared, and not just half. If each character is represented by 2 bytes, shouldn't the 'while' loop be 'while (j < 80 * 25 * 2)'?

1 ^ | v • Reply • Share ›

Arjun Sreedharan Mod ➚ Kieron • 9 years ago • edited

You are right. Will Fix.Thanks

 (https://opheliarand-blog.tumblr.com/)opheliarand-blog (https://opheliarand-blog.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://siri-how-is-the-weather.tumblr.com/)siri-how-is-the-weather (https://siri-how-is-the-weather.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://hacktheblog.tumblr.com/)hacktheblog (https://hacktheblog.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://feelzonwheelz.tumblr.com/)feelzonwheelz (https://feelzonwheelz.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://sputnik-schlaeft.tumblr.com/)sputnik-schlaeft (https://sputnik-schlaeft.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://the-friend-with-little-benefits.tumblr.com/)the-friend-with-little-benefits (https://the-friend-with-little-benefits.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://katrona-blog1.tumblr.com/)katrona-blog1 (https://katrona-blog1.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://saken-mx.tumblr.com/)saken-mx (https://saken-mx.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://blackmesasecurity.tumblr.com/)blackmesasecurity (https://blackmesasecurity.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://bahsana-blog.tumblr.com/)bahsana-blog (https://bahsana-blog.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://yoanribeiro.tumblr.com/)yoanribeiro (https://yoanribeiro.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://goodstoriesneverend.tumblr.com/)goodstoriesneverend (https://goodstoriesneverend.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://green-hanuta.tumblr.com/)green-hanuta (https://green-hanuta.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://aikalie.tumblr.com/)aikalie (https://aikalie.tumblr.com/post/83000405824) a reblogué ce billet depuis computerpile (https://computerpile.tumblr.com/)

 (https://memiux.tumblr.com/)memiux (https://memiux.tumblr.com/post/82995446033) a reblogué ce billet depuis arjunsreedharan (https://arjunsreedharan.org/)

 (https://splenty.tumblr.com/)splenty (https://splenty.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://computerpile.tumblr.com/)computerpile (https://computerpile.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://silverlinethings.tumblr.com/)silverlinethings (https://silverlinethings.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://goshpenny.tumblr.com/)goshpenny (https://goshpenny.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://inputmismatchexception.tumblr.com/)inputmismatchexception (https://inputmismatchexception.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://comp.errs.io/)compiler-errors (https://comp.errs.io/post/82907518361) a reblogué ce billet depuis intercal (https://intercal.xyz/)

 (https://rossn.tumblr.com/)rossn (https://rossn.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://sinyeol-blog.tumblr.com/)sinyeol-blog (https://sinyeol-blog.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://ilikebadmusic.tumblr.com/)ilikebadmusic (https://ilikebadmusic.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://hauleth.tumblr.com/)hauleth (https://hauleth.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://benknis.tumblr.com/)benknis (https://benknis.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://armish-blog.tumblr.com/)armish-blog (https://armish-blog.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://intercal.xyz/)intercal (https://intercal.xyz/) a ajouté ce billet à ses coups de cœur

 (https://intercal.xyz/)intercal (https://intercal.xyz/post/82788619797) a reblogué ce billet depuis arjunsreedharan (https://arjunsreedharan.org/)

 (https://thefox21.tumblr.com/)thefox21 (https://thefox21.tumblr.com/) a ajouté ce billet à ses coups de cœur

 (https://arjunsreedharan.org/)arjunsreedharan (https://arjunsreedharan.org/) a publié ce billet

Afficher plus de notes