

نوشتن يك کرنل ساده به زبان C همراه با توابع اصلي printf و clrscr

ما در اين مقاله قصد داريم كه به شما نشون بديم نوشتن يك کرنل ساده هستش براي شروع کرنل ما در فايل Kernel_Start.asm هست.

[BITS 32]

[global start]

[extern _k_main]

start:

call _k_main ; اين تابع در يك فايل سي نوشته شده كه به برنامه لينك ;

cli ; غير فعال كردن وقفه ها ;

hlt ; متوقف كردن سي بي يو ;

اين يك كد ۳۲ بیتی هست (به این دستور توجه کنید [BITS 32]) و کرنل ما تابع k_main هست كه در فايل C قرار داره. اما در فايل اسمبلي اين تابع رو k_main صدا زدیم! دلیلش این هست كه كامپایلرهای زبان C/C++ يك زیرخط در زمان كامپایل به اول اسم هر تابع اضافه می کنند مگر قرار باشه كه بخواید این فايل رو به يك ELF لینك كنید چون فايلهای ELF نیازی به زیر خط اول موقع فراخوانی روالهای سي (لینك) ندارند و می توانید تابع رو به همون صورتی كه در سي هست فراخوانی كنند. بعد از فراخوانی تابع k_main دستور عمل CLI اجرا می شود. این دستور عمل تمام وقفه ها رو غیر فعال می كنه و دیگه وقفه ای اتفاق نمی افتد. سپس دستور HLT اجرا می شود این دستور به CPU می گوید اجرا كردن دستور عمل ها بعدی رو متوقف كن (بعد از این دستور دیگه دستور عملی اجرا نمیشه شاید با خودتون بگید خوب اگر بخوایم دوباره فعال كنیم چطوری باید این كار رو كرد اجرای دوباره برنامه با استفاده از وقفه صورت می گیرد اگر وقفه ها غیر فعال باشند سیستم عملاً هنگ می كنه) ما میتونیم به جای HLT از دستور \$ JMP نیز استفاده بكنیم اما این دستور باعث میشه كه سیكل های CPU به هدر بره و به CPU فشار بیاد و داغ كنه چون به طور مرتب فقط به مكان خودش در حال پرش هست اما دستور عمل ما (HLT) این كار رو نمیکنه و واقعا CPU رو غیر فعال می كنه و هیچ دستور عملی رو اجرا نمیکنه كه به سیستم فشار بیاد.

نکته: وقفه ها میتوانند CPU رو از حالت HLT در بیاورند اما چرا ما قبل از دستور عمل HLT وقفه ها غیر فعال کردیم برای این هست كه می خواهیم سیستم كاملاً از كار بیفته.

حالا اجازه بدهید كه ما متغیر ها و توابع اصلي رو در فايل Kernel.c تعریف كنیم. توابع ما اینها هستند حال در ذیل به توضیح هر کدامشون می پردازیم

```
#define WHITE_TXT 0x07
```

// متن سفید بر روی زمینه سیاه

```
void k_clear_screen();
```

```
unsigned int k_printf(char *message, unsigned int line);
```

```
void update_cursor(int row, int col);
```

این #define WHITE_TXT 0x07 هیچ كار خواصي انجام نمیده. ما در جلوش گفتیم كه این متغیر چه كار می كنه فقط این متغیر رو بیاد داشته باشید حالا اجازه بدهید كه ما به سمت تعریف تابع k_Kernel برویم.

```

k_main()    // این تابع در برنامه ما مثل تابع main در C هست
{
    k_clear_screen();
    k_printf("Hi!\nHow's this for a starter OS?", 0);
};

```

تابع `k_main` نقطه ورود ما به کرنل خودمون هست. ما این تابع رو در فایل اسمبلی `Kernel_Start.asm` صدا زده ایم. تابع `k_clear_screen` کارش پاک کردن صفحه نمایش هست و تابع `k_printf()` هم کارش نمایش متن بر روی صفحه نمایش هستش.

پارامتر دوم تابع `k_printf()` کارش رسم متن بر روی خط مورد نظر هست (صفر خط اول، یک خط دوم، دو خط سوم و الی آخر) و `\n` نیز همانند تابع `printf` در `C/C++` کارش رفتن به خط جدید هست.

هورا!

این چطور بود برای شروع یک سیستم عامل؟

در مود حفاظت شده شما نمی توانید از وقفه ها بایوس برای پاک کردن صفحه نمایش استفاده کنید به همین دلیل ما برای خودمون یک تابع نوشتیم که مستقیماً به صفحه نمایش دسترس داره و صفحه رو پاک می کند.

```

void k_clear_screen()    // این تابع صفحه نمایش رو پاک می کند
{
    char *vidmem = (char *) 0xb8000;
    unsigned int i=0;
    while(i < (80*25*2))
    {
        vidmem[i]=' ';
        i++;
        vidmem[i]=WHITE_TXT;
        i++;
    };
};

```

در تابع بالا (`k_clear_screen`) اشاره گر (`vidmem`) اشاره می کنه به آدرس `0xB8000` که آدرس صفحه نمایش در مود محافظت شده است. ما متغیر رو از نوع `char` (یک بایت) تعریف کردیم پس ما می توانیم در هر لحظه یک بایت رو در حافظه صفحه نمایش بنویسیم مود متنی در ریزپردازنده های سری X86 به طول و عرض `80 x 25` کاراکتر هست هر کاراکتر به دو بایت نیاز داره بایت اول برای خود کاراکتر و بایت دوم نیز خاصیت کاراکتر رو کنترل می کنه مثل رنگ و چشمک زن بودن کاراکتر رو مشخص می کنه. ما با ضرب طول و عرض `80*25` می توانیم مقدار کاراکتر های که می توانیم بر روی صفحه نمایش رسم کنیم رو بدست می آوریم و ما این عدد بدست آمده رو ضرب در 2 می کنیم ما در هر لحظه به یک بایت در صفحه نمایش دسترسی داریم و ما با استفاده از دستور `'vidmem[i]=' '` می توانیم یک فاصله و یا هیچ کاراکتری چاپ نشه یا یک فضای خالی ایجاد کنیم و ما عدد 1 رو به `i` (`i++`) اضافه می کنیم و متغیر ما `I` دقیقاً به اندازه یک نقطه به جلو برده می شود. حالا بایت بعدی خاصیت هست و ما در آن عدد `0x07` رو قرار می دهیم. `0x07` تعیین می کنه یک پشت زمینه سیاه با رو زمینه (کاراکتر) سفید و بدون چشمک زن در مود متنی رسم شود.

حالا پیش به سوی تابع `k_printf` .

```
unsigned int k_printf(char *message, unsigned int line)
// the message and then the line #
{
    char *vidmem = (char *) 0xb8000;
    unsigned int i=0;
    i=(line*80*2);
    while(*message!=0)
    {
        if(*message=='\n') // check for a new line
        {
            line++;
            i=(line*80*2);
            *message++;
        } else {
            vidmem[i]=*message;
            *message++;
            i++;
            vidmem[i]=WHITE_TXT;
            i++;
        };
    };
    return 1;
};
```

تابع `k_printf` خیلی شبیه تابع `k_clear_screen` کار می‌کنه .
(`while(*message!=0)` این حلقه , کاراکترها رو می‌خونه تا به پایان رشته برسه و دوباره تابع رو ادامه بده . خط (`if(*message=='\n')` در رشته به دنبال کاراکتر `\n` می‌گردد اگر این کاراکتر رو پیدا کرد به خط جدید می‌رود و کاراکتر بعدی در ابتدای خط بعدی چاپ می‌شود . اگر کاراکتر ما `\n` نبود ما کاراکتر رو در داخل حافظه صفحه نمایش قرار می‌دهیم و خاصیت کاراکتر رو `0x07` قرار می‌دهیم (پشت زمینه سیاه و متن سفید بدون حالت چشمک زن)

کامپایل کردن کرنل

اول شما سورس کد کرنل رو دریافت کنید و همچنین شما به اسمبلر (NASM) , کامپایلر C (GCC یا DJGPP) و به یک لینکر (LD) نیاز دارید .

حالا شما در بالای فایل لینکر (`link.ld`) این خط رو می‌بینید :

```
.text 0x100000
```

شما نیاز دارید برای لود کردن کرنل به حافظه مورد نظر آدرسش رو به صورت هگزیمال بنویسید در این مورد محل مورد نظر **1MB** است که (در حالت هگزیمال می‌شود `0x100000`) در اینجا باید کرنل بار شود

اول شما فایل اسمبلی تون رو کامپایل کنید

```
nasm -f aout kernel_start.asm -o -ks.o
```

این فایل ks.o نیز مانند kernel_start.asm با فرمت aout کامپایل می شود . حالا برای C نیز این دستور رو اجرا می کنیم

```
gcc -c kernel.c -o kernel.o
```

قدم بعد و آخرین قدم هم لینک کردن دو فایل ks.o و kernel.o و تبدیل کردنشون به یک فایل است در این مورد ما نیاز داریم که فایل رو به فرمت باینری (Binary) و حالات تخت (Flat) تبدیل کنیم که ما برای این کار از فایل اسکرپتی (link.ld) که برای اینکار (Link) ساختیم استفاده می کنیم ما این دو فایل رو با این دستور به هم وصل می کنیم :

```
Ld -T link.ld -o kernel.bin ks.o kernel.o
```

این خیلی مهمه که اول ks.o لینک بشه بعد kernel.o اگر شما این دو فایل رو جا به جا لینک کنید کرنل کار نخواهد کرد ! کرنل پس از بوت شدن و اجرای بوت لودر و رفتن به مد محافظت شده و فعال شدن A20 اجرا خواهد شده . اگر شما دوست دارید که کرنل تون گراب (GRUB) داشته باشه شما می تونید گراب رو از سایتش دانلود کنید(به همون روشی که در بالا گفته شده به برنامه تان لینک کنید)

نتیجه گیری

شما الان یک کرنل ساده دارید ! شاید شما یک تابع k_printf بهتری بخواهید داشته باشید شما می تونید از این مثال استفاده کنید و امکانات(s% , d% , c% و الي اخر) بیشتری به تابع تون اضافه کنید شما می تونید در آینده مثال های بهتری بنویسید

نکته : شما قبل از اجرا کرنل باید به مود محافظت شده بروید