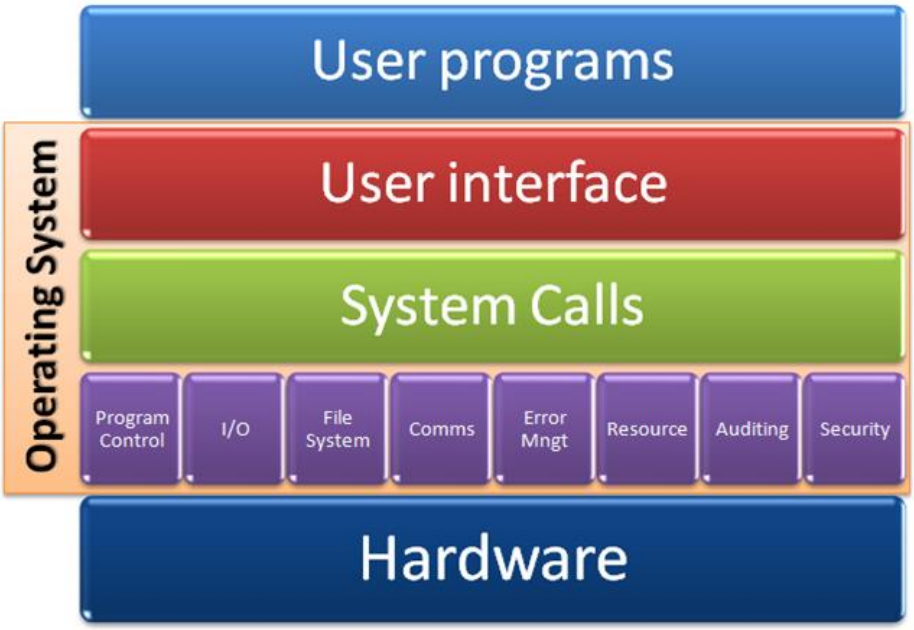
در این مقاله قصد داریم نحوه‌ی برقراری ارتباط نرم‌افزارها با سیستم‌عامل و ارسال درخواست برای ارتباط با بخش‌های مختلف آن و یا ارتباط با سخت افزارها را بیان کرده و کمی با طرز کار سیستم‌عامل آشنا شویم.

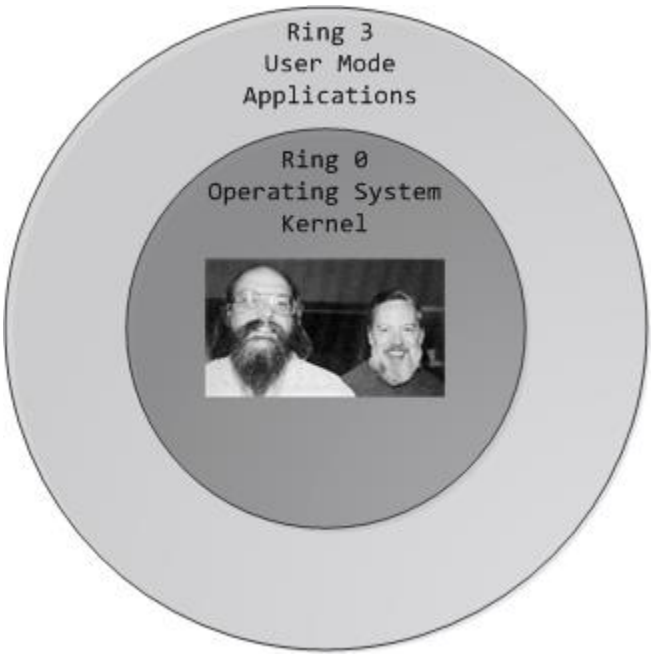
سیستم‌عامل از دو بخش کلی تشکیل می‌شود، یکی کرنل (هسته) که وظیفه‌ی ارتباط با سخت‌افزارها، فراهم کردن driver برای ارسال فرامین به آن‌ها، مدیریت حافظه، مدیریت پروسه‌ها، مدیریت ورودی/خروجی، زمان‌بندی دسترسی به پردازنده و... را داشته و بخش دوم نرم‌افزارها و کتابخانه‌های سطح کاربر می‌باشد که ظاهر سیستم‌عامل و محیط کار را فراهم می‌کنند. البته جزئیات زیادی وجود دارد ولی با یک دید خیلی بالا و حذف جزئیات می‌توانیم این دسته بندی را داشته باشیم. در شکل ۱ این ساختار مشاهده می‌شود.



شکل ۱) ساختار سیستم‌عامل

فضای حافظه‌ی اشغال شده توسط کرنل از فضای برنامه‌های سطح کاربر جدا بوده و هیچ برنامه‌ی سطح کاربری مجوز دسترسی به آنرا ندارد. اگر کمی این قضیه را بسط دهیم باید بگوییم که علاوه بر مجزا بودن فضای هسته از فضای سطح کاربر، حافظه‌ی برنامه‌ها نیز کاملا از هم جدا بوده و هیچ برنامه‌ای به صورت مستقیم و بدون استفاده از مکانیزم‌های (IPC) Inter-Process Communication حق دسترسی به فضای برنامه‌های دیگر را نداشته و در صورتیکه درخواستی برای آن ارسال کند از طرف سیستم‌عامل با یک Fault مواجه خواهد شد. در کنار این جداسازی فضای حافظه، همچنین دستورات زبان‌ماشینی وجود دارند که استفاده از آن‌ها تنها در فضای کرنل امکان‌پذیر بوده و Privilege بالاتری دارند. در مورد ارتباط مستقیم با سخت‌افزارها نیز به همین شکل است. برای کار با سخت‌افزارها می‌توان به دو صورت Memory Mapped I/O (MMIO) و یا I/O Ports ارتباط برقرار کرده و داده و دستوری برای آن‌ها ارسال کرده و یا اطلاعاتی از آن‌ها دریافت نمود. در مدل MMIO آدرس‌هایی از حافظه به صورت مستقیم برای یک سخت افزار رزرو شده و درصورتیکه

در این آدرس اطلاعاتی نوشته شود، سخت‌افزار مورد نظر آنرا برداشته و کاری انجام می‌دهد. به عنوان مثال در صورتیکه یک کرنل ابتدایی بنویسید، برای نمایش اطلاعات در صفحه‌ی نمایش 80x25 متنی باید رنگ متن و کاراکتر مورد نظر را در فضای حافظه‌ای که از آدرس 0x000B8000 شروع می‌شود درج کنید تا متن مورد نظر نمایش داده شود. (در مقاله‌های آینده به شیوه‌ی انجام اینکار خواهیم پرداخت) در مدل I/O Ports آدرس ارتباطی با سخت‌افزار مشخص شده و سپس داده‌ای برای آن ارسال شده و یا فرمانی به آن داده شده و یا اطلاعاتی از آن خوانده می‌شود. به عنوان مثال برای ارتباط با حافظه‌ی CMOS (وظیفه‌ی این حافظه نگهداری اطلاعات BIOS در زمان خاموش شدن کامپیوتر است) می‌توان از شماره‌ی پورت‌های 0x70, 0x71 استفاده نمود. در ارتباط با I/O Port از دستورات اسمبلی in, out استفاده می‌شود. (در ادامه یک مثال از کاربرد این حافظه و نحوه‌ی کار با I/O Ports ارائه خواهد شد.) پس کرنل علاوه بر چیزهایی که فراهم می‌کند، از سه چیز محافظت نیز می‌کند: حافظه، پورت‌های I/O و مجموعه‌ای از دستورات زبان ماشین (مثل دستور فعال/غیر فعال کردن وقفه). برای این جداسازی و حفاظت، باید هم سیستم‌عامل و هم پردازنده کارهایی انجام دهند. در پردازنده‌های اینتل هر دستور بر روی پردازنده می‌تواند در چهار حلقه اجرا شود که از ۰ تا ۳ شماره‌گذاری شده‌اند. حلقه‌ی صفر اولویت بالاتری را داشته و کد کرنل در این حلقه اجرا می‌شود. حلقه‌ی ۳ کمترین اولویت را داشته و کد سطح کاربر در این حلقه اجرا می‌شود. بسیاری از سیستم‌عامل‌ها از جمله ویندوز و لینوکس تنها از دوحلقه‌ی ۰و۳ استفاده کرده و از ۱و۲ استفاده نمی‌کنند. این مورد را نشان می‌دهد. شکل ۲ استفاده از حلقه‌های ۰و۳ را نشان می‌دهد.

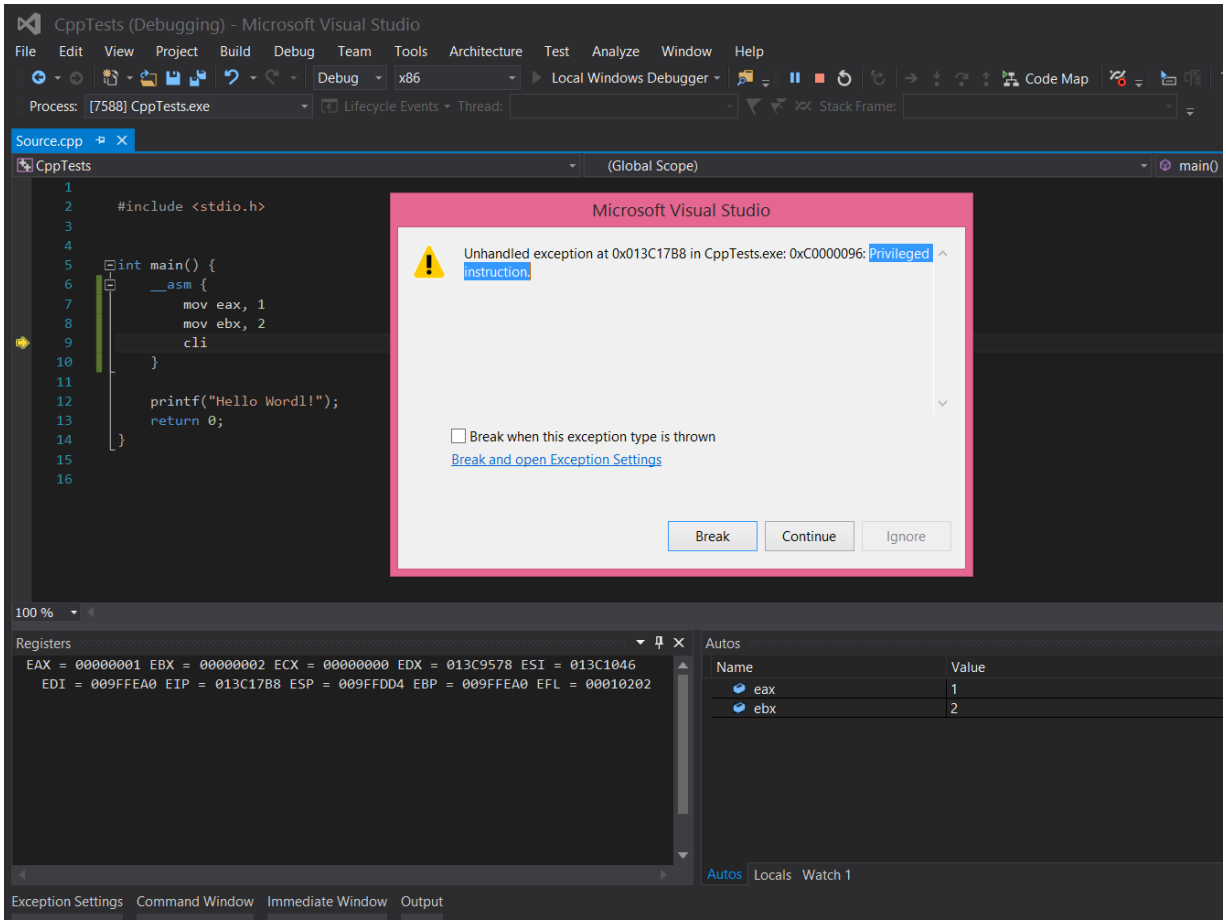


شکل ۲) نحوه‌ی استفاده از ring در سیستم‌عامل

نکته: در محصولات مجازی‌سازی مثل VirtualBox بخش هسته‌ی سیستم‌عامل میهمان در Ring1 اجرا شده و خود محصول مجازی‌سازی در Ring0 قرار داده می‌شود و به این صورت اجرای سیستم‌عامل از hypervisor کنترل می‌شود.

شیوه‌ی کار به زبان ساده (و کمی غیر دقیق ولی کافی برای بیان منظور) به این صورت است که در Descriptor های مربوط به داده و کد، دوبیت (برای ۰تا۳) اولویت نیز نگهداری شده و در زمانیکه درخواستی ارسال می‌شوند این مقدار با

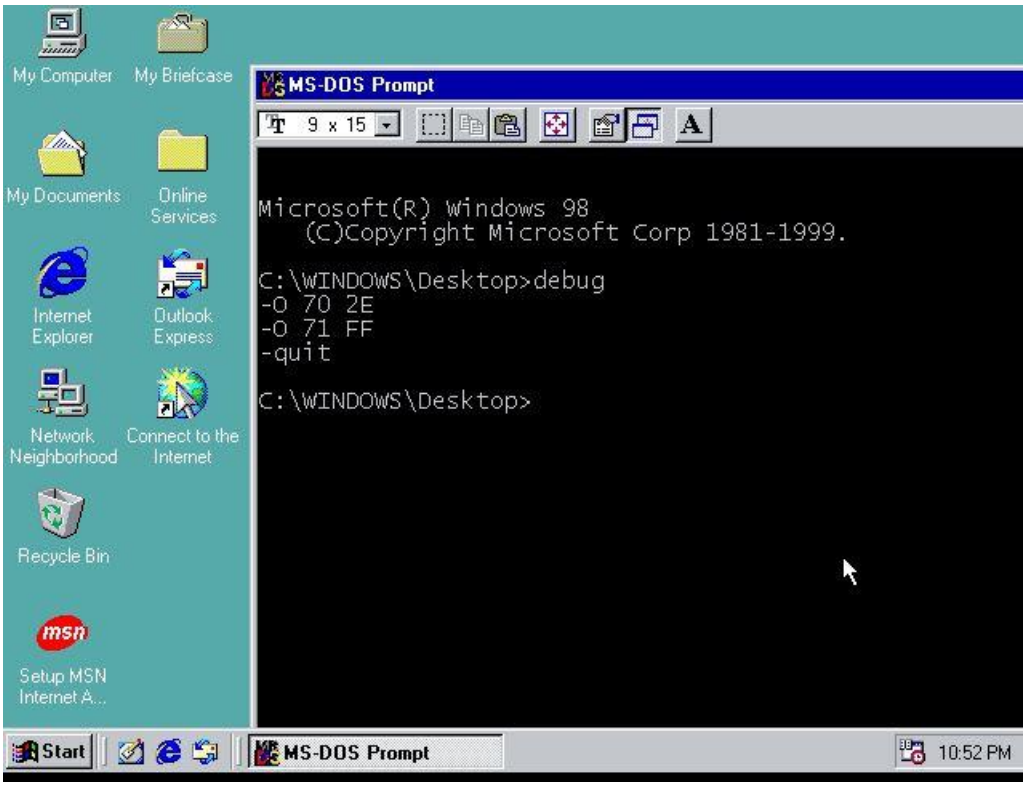
مقدار اولویت جاری CPU مقایسه شده و در صورتیکه مطابقت وجود نداشت یک General Protection Fault ارسال می‌شود. تست کردن این موضوع به سادگی قابل انجام است. برای اینکار کافی است خارج از محیط کرنل دستوری مثل cli (برای غیر فعال کردن وقفه) را اجرا کنیم تا نتیجه و ایجاد Exception را مشاهده نماییم. اما در صورتیکه به عنوان مثال در یک Linux Kernel Module این دستور اجرا شود خطایی دریافت نشده و به دلیل اجرا شدن کد در محیط کرنل و Ring0 مشکلی بوجود نخواهد آمد. در شکل ۳ اجرای دستور در محیط Visual Studio و استثنای تولید شده را مشاهده می‌کنید.



شکل ۳) استثنای اجرای Privileged Instruction

در خلال توضیحات به این موضوع اشاره نمودیم که برای پشتیبانی از این موضوع و عدم دسترسی به فضای حافظه، پورت‌های I/O و اجرای دستورات دارای اولویت بالا، CPU مکانیزم حلقه‌ها را داشته ولی از طرف دیگر سیستم‌عامل هم باید پشتیبانی لازم را فراهم آورد. برای تست پشتیبانی سیستم‌عامل‌های مختلف می‌توانیم از ویندوز ۹۸ استفاده نماییم!!!! علی‌رغم اینکه این نسخه از ویندوز خیلی قدیمی بوده و الان سخت می‌توان جایی را پیدا کرد که از آن استفاده می‌کنند ولی برای بررسی ما گزینه‌ی خوبی است. احتمال زیاد به این قضیه برخورده‌اید که رمز BIOS رو فراموش کرده باشید و نیاز داشته باشید که با در آوردن باتری، تنظیمات را ریست کرده و رمز را حذف کنید. این مورد کاری است که می‌توان با دسترسی به پورت‌های I/O مربوط به CMOS نیز انجام داده و با استفاده از همان دو پورت 0x70, 0x71 که معرفی شدند تنظیمات BIOS را ریست نمود. CMOS از اطلاعاتی که نگه می‌دارد یک Checksum ایجاد کرده و بخشی از آنرا

در اندیس 0x2E نگه می‌دارد، اگر این مقدار بهم بریزد، باعث می‌شود که تنظیمات ریست شده و رمز نیز حذف گردد. البته به دلیل حفاظت سیستم عامل و عدم دسترسی به پورت‌های I/O این امکان در ویندوزهای XP به بعد (البته من ویندوز ۲۰۰۰ را تست نکرده‌ام!) این امکان وجود ندارد. ولی بر روی ویندوز ۹۸ و با استفاده از دستور debug (که در واقع امکان نوشتن و trace کد اسمبلی را فراهم می‌کند و می‌توان بوسیله‌ی آن فایل COM نیز ساخت) می‌توانید از دستور out استفاده کرده و در پورت‌ای CMOS و در اندیس 0x2E مقدار نامعتبری (مثلا 0xFF) ریخته و باعث حذف رمز BIOS شوید. این مورد بر روی کامپیوتری که ویندوز ۹۸ داشته باشد (یا حتی با یک محصولی مثل Hiren's Boot CD دارای بوت ویندوز ۹۸ بوت شود) و یا محصولات مجازی‌سازی مثل VMWare قابل تست کردن است. در شکل ۴ می‌توانید اجرا بر روی VMWare را مشاهده کنید. از طریق 0x70 جایی که باید داده در آن قرار گیرد مشخص شده و از طریق 0x71 خود داده ارسال می‌شود.

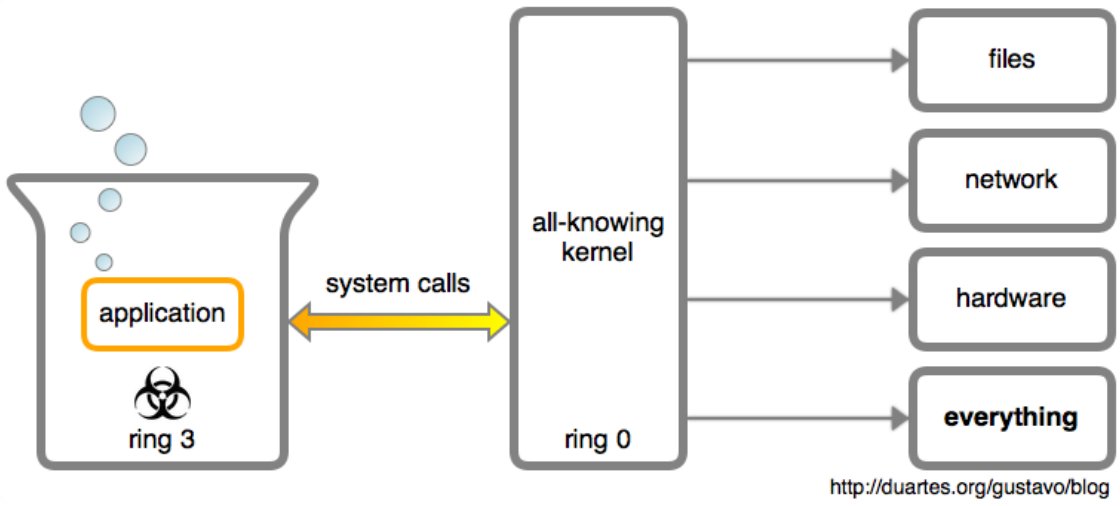


شکل ۴) ریست رمز BIOS در ویندوز ۹۸

تا اینجای کار کمی با سیستم‌عامل آشنا شده و می‌دانیم که جلوی انجام چه کارهایی را می‌گیرد! ولی سوالی که وجود دارد این است که برنامه‌ها چطور کار کرده و با حافظه، دیسک و از همه مهمتر صفحه‌ی نمایش و صفحه‌ی کلید چگونه ارتباط برقرار کرده و داده‌ای را دریافت کرده و یا نمایش می‌دهند؟! اگر جلوی ارتباط مستقیم با کلیه‌ی دستگاه‌های جانبی گرفته شده و تنها کرنل و درایورهایی که در سطح آن کار می‌کنند اجازه‌ی ارتباط مستقیم دارند، پس نرم‌افزارهای سطح کاربر چگونه ارتباط برقرار کرده و وظایف خود را به انجام می‌رسانند؟

کرنل (هسته) سیستم‌عامل واسطهایی برای ارتباط با بخش‌های مختلف فراهم می‌کند که به آن فراخوانی سیستم System Call گفته می‌شود. هر نرم‌افزاری برای انجام وظایفی که تحت کنترل هسته‌ی سیستم‌عامل می‌باشد از این واسطها استفاده

کرده و درخواست خود را به کرنل ارسال می‌کند. سپس پردازنده کد تحت کرنل را اجرا کرده و نتیجه را به بخش سطح کاربر ارسال می‌نماید. دقت کنید که در یک برنامه بسیاری از وظایف نیازی به دخالت کرنل نداشته و کد سطح کاربر به اجرای آن می‌پردازد. کلیه‌ی محاسبات، اجرای الگوریتم‌ها، پردازش داده‌ها، آماده کردن داده برای نمایش و... در سطح کاربر انجام می‌شود ولی برای وظایفی مثل ذخیره و بازیابی داده روی دیسک، ارسال برای پروسه‌ی دیگر (درون سیستم‌عامل یا تحت شبکه)، نمایش نتایج، از واسط‌های کرنل کمک گرفته می‌شود. در شکل ۵ ارتباط با کرنل از طریق فراخوانی سیستم مشاهده می‌شود.



شکل ۵) فراخوانی سیستمی

برای بررسی دقیتر روال اجرای برنامه‌ها و بررسی فراخوانی‌های سیستمی انجام شده یک برنامه‌ی بسیار ساده را تصور کنید! این برنامه‌ی سلام به دنیا! در شکل ۶ مشاهده می‌شود.

```c
#include <stdio.h>

int main() {
    printf("Hello Word!");
    return 0;
}
```

شکل ۶) برنامه سلام به دنیا!

در این برنامه اجرای تابع printf مشاهده می‌شود. این تابع با دریافت پارامترها و فرمت رشته، آنرا چاپ می‌کند. آماده‌سازی رشته‌ی نهایی از وظایف این تابع در سطح کاربر می‌باشد ولی به دلیل اینکه این تابع، در کتابخانه‌ی libc تعریف شده است، دسترسی به این کتابخانه، انجام وظایف مختلف دیگر مثل بررسی حافظه، چک کردن رجیستری (در ویندوز البته) و بسیاری کارهای دیگر از جمله موارد دیگری است که انجام گرفته و سپس رشته چاپ شده و در نهایت نیز برنامه خاتمه می‌یابد. پس فراخوانی های سیستمی مختلفی انجام می‌گیرد تا این دستور کتابخانه‌ای بارگذاری، اجرا شده و نتیجه را نمایش دهد. در ویندوز با استفاده از نرم‌افزارهای drstrace.exe و nttrace.exe (باید جداگانه دانلود کنید و پیش‌فرض با

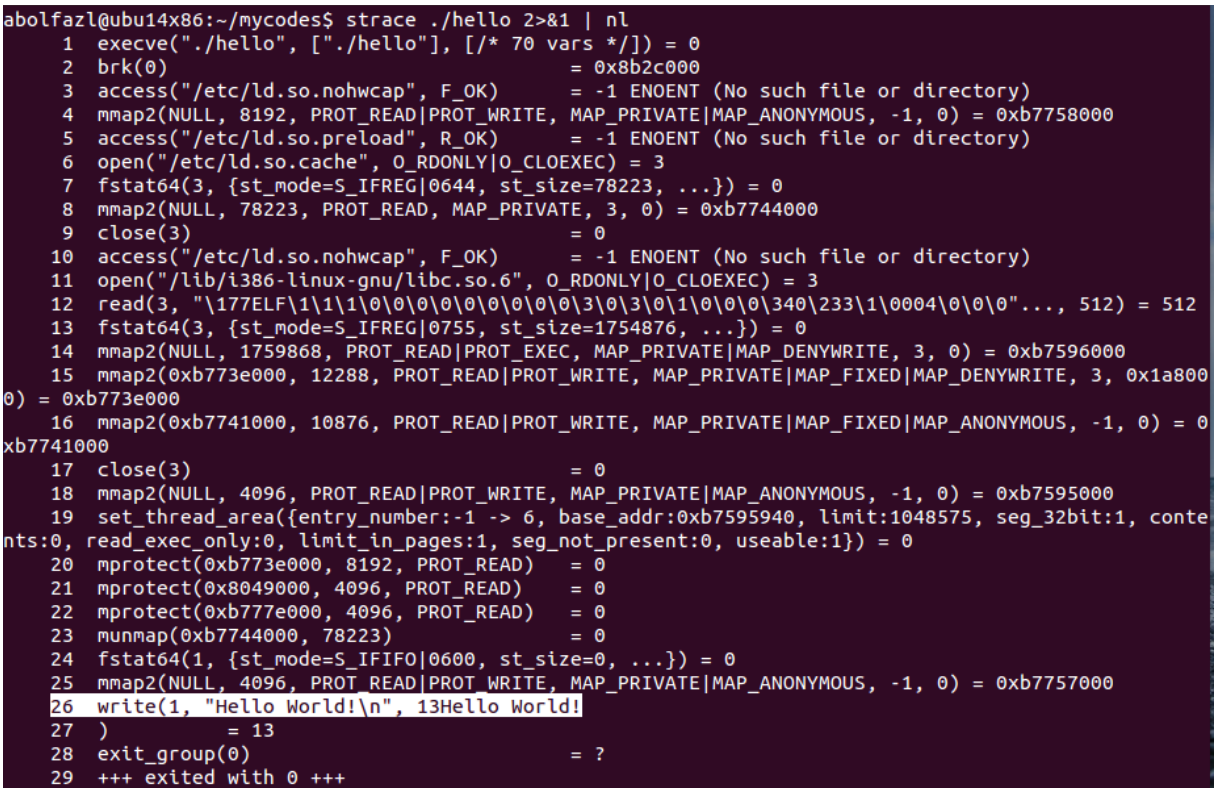
ویندوز وجود ندارند) و در لینوکس با استفاده از strace می‌توانید فراخوانی‌های سیستمی انجام گرفته توسط یک نرم‌افزار را مشاهده نمایید. در شکل ۷ بخشی از فراخوانی‌های سیستمی انجام گرفته تحت ویندوز در اجرای برنامه‌ی شکل ۶ مشاهده می‌شود.

```
1  Process 6628 starting at 0000000000A61046
2  F:\NtTrace\CppTests.exe
3  Loaded DLL at 00007FF819620000 C:\Windows\SYSTEM32\ntdll.dll
4  Loaded DLL at 00000000779A0000 C:\Windows\SysWOW64\ntdll.dll
5  NtQueryPerformanceCounter( Counter=0xc4f100 [2.58075e+010], Freq=null ) => 0
6  NtProtectVirtualMemory( ProcessHandle=-1, BaseAddress=0xc4f148 [0x00007ff819764000],
   Size=0xc4f140 [0x1000], NewProtect=4, OldProtect=0xc4f180 [8] ) => 0
7  NtProtectVirtualMemory( ProcessHandle=-1, BaseAddress=0xc4f148 [0x00007ff819764000],
   Size=0xc4f140 [0x1000], NewProtect=8, OldProtect=0xc4f180 [4] ) => 0
8  NtQuerySystemInformation( SystemInformationClass=0 [SystemBasicInformation],
   SystemInformation=0xc4f110, Length=0x40, ReturnLength=null ) => 0
9  NtQueryInformationProcess( ProcessHandle=-1, ProcessInformationClass=0x24 [ProcessCookie],
   ProcessInformation=0xc4ee58, Length=4, ReturnLength=null ) => 0
10 NtProtectVirtualMemory( ProcessHandle=-1, BaseAddress=0xc4ee30 [0x00007ff819761000],
   Size=0xc4ee28 [0x4000], NewProtect=2, OldProtect=0xc4ee20 [4] ) => 0
11 NtOpenKey( KeyHandle=0xc4eaf8 [8], DesiredAccess=0x9,
   ObjectAttributes="\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion\Image File
   Execution Options" ) => 0
12 NtOpenKey( KeyHandle=0xc4eba8, DesiredAccess=0x9, ObjectAttributes=8:"CppTests.exe" ) =>
   0xc0000034 [2 'The system cannot find the file specified.']
13 NtQueryInformationProcess( ProcessHandle=-1, ProcessInformationClass=0x1a
   [ProcessWow64Information], ProcessInformation=0xc4eff8, Length=8, ReturnLength=null ) => 0
14 NtQueryInformationProcess( ProcessHandle=-1, ProcessInformationClass=0x24 [ProcessCookie],
   ProcessInformation=0xc4ee20, Length=4, ReturnLength=null ) => 0
```

شکل ۷) فراخوانی‌های سیستمی در ویندوز

دقت کنید که توابع از فایل ntdll.dll بارگذاری شده و نام تمامی آن‌ها با Nt شروع شده است. در این شکل تنها ۴خط ابتدایی یک فایل ۴۱۳ خطی نمایش داده شده است! در ویندوز به دلیل تغییرات ایجاد شده در نسخه‌های مختلف، استفاده‌ی مستقیم از فراخوانی‌های سیستمی توصیه نشده و بجای آن باید از APIهای تعریف شده استفاده شود. (یک لایه‌ی Abstraction ایجاد شده بر روی فراخوانی‌های سیستمی) این توابع حتی به صورت رسمی مستند نیز نشده‌اند و تمامی مستندات به APIهای رسمی اشاره کرده‌اند.

در لینوکس قضیه کمی متفاوت بوده و استفاده‌ی مستقیم از فراخوانی‌های سیستم متداول است. حتی امکان نوشتن فراخوانی سیستمی جدید و اضافه کردن آن به کرنل لینوکس وجود دارد. (در مقاله‌ای نحوه‌ی انجام اینکار را توضیح خواهیم داد) در شکل ۸ فراخوانی‌های سیستمی انجام شده در اجرای برنامه‌ی شکل ۶ نمایش داده شده است. درک نحوه‌ی کار و پیدا کردن بخشی که واقعا عملیات چاپ رشته در آن انجام می‌پذیر در این خروجی کوتاه بسیار ساده‌تر از خروجی ویندوز است. در این خروجی نیز مشخص است که دسترسی به فایل‌های کتابخانه‌ای برای دسترسی به توابع انجام گرفته است. همچنین در خط اول استفاده از تابع execve برای ایجاد پروسه و اجرا کردن کد hello (توسط دستور strace) مشاهده می‌شود. بخش اصلی که نمایش رشته در آن انجام می‌پذیرد نیز در خط ۲۶ مشاهده می‌شود. در این خط از write برای نوشتن در فایلی با شماره‌ی ۱ استفاده شده است. (در لینوکس STDOUT==1 و STDERROR==2 می‌باشند)



شکل ۸) فراخوانی‌های سیستمی در لینوکس

در مقاله‌ی دیگری استفاده از اسمبلی برای فراخوانی سیستمی و عدم نیاز به کتابخانه‌ها و درنتیجه استفاده از تنها ۲ فراخوانی سیستمی برای نمایش رشته و خروج از برنامه را توضیح خواهیم داد.

امیدوارم مفید بوده باشه، موفق باشید.