به نام خدا

پروژه ی سیستم عامل

حدیث غفوری ۹۸۲۵۴۱۳

Usys.S

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

به این کد یک macro میگویند.

شما می توانید در مورد ماکرو به عنوان تابعی فکر کنید که کامپایلر قبل از ایجاد کامپایل واقعی الگو را جایگزین می کند.

```
#define SYSCALL(getpid) \
  .globl getpid; \
  getpid: \
    movl $SYS_getpid, %eax; \
    int $T_SYSCALL; \
    ret
```

در این مثال سیستم کال getpid را میگیریم و میبینیم چطوری با یک ماکرو جایگزین میشه.

کد اسمبلی یک global name تعریف میکند به اسم getpid و سپس کاری که انجام میدهد را توصیف میکند.

getpid شماره ی SYS_getpid که در فایل syscall.h تعریف شده بود را به رجیستر eax% منتقل میکند و یک interrupt با شماره ی T_SYSCALL کال میکند (شماره: 64 و در فایل traps.h تعریف شده است).

vector.S

```
308  vector62:
309    pushl $0
310    pushl $62
311    jmp alltraps
312  .globl vector63
313  vector63:
314    pushl $0
315    pushl $63
316    jmp alltraps
317  .globl vector64
318  vector64:
319    pushl $0
320    pushl $64
321    jmp alltraps
322  .globl vector65
323  vector65:
324    pushl $0
325    pushl $65
326    jmp alltraps
327  .globl vector66
328  vector66:
329    pushl $0
330    pushl $66
331    jmp alltraps
```

vector.S یک vector table بزرگ را تعریف میکند. نکته مهم این است که INT $64 یک سری کار انجام میدهد

و میپره به دستور vector64 و این مقدار های صفر و 64 را توی استک push میکند قبل از اینکه alltraps را فراخوانی کند. چون هر structure در c فقط یک روشی از abstract کردن بایت های متوالی است ما هم اکنون درحال ساختن یک trapframe (see x86.h) هستیم و 64 هم ویژگی trapno است.

trapasm.S

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Alltraps فراخوانی شد و کار ساختن trapframe را تمام میکند.

از انجایی که ما داریم از یک ممووری استک استفاده میکنیم ما struct را از پایین به بالا میسازیم. بعد از errno ما gs,fs,es,dsرا داریم. و اینها دقیقا همان چیزهایی هستند که alltraps توی استک پوش میکند.

از انجایی که انها کوتاه هستند و رجیستر ها طولانی هستند trapframe.h جاهایی که خالی هستند را با padding پرمیکند.

و pushall هم رجیسترهای همه منظوره را پوش میکند. و در نهایت trap را کال میکند و ابتدای trapframe را به عنوان پارامتر پاس میدهد .

دستور trapret :

تمام رجیسترهایی که روی trapframe سیو شده اند را restore میکند و این نشان میدهد که چطوری پروگرم به اجراش بعد از trap ادامه میدهد.

trap.c

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

بعد از دستور INT رفتیم سراغ ساخت trapframe و بعد از ان سراغ trap function

Errno مقداری است که توی vector.s پوش شده است و به T_SYSCALL مربوط است. اولین if مقدار myproc()->tf را قبل از کال کردن ()syscall ست میکند.

syscall.c

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};
```

این فایل یک ارایه ای از function pointer ها را نشان میدهد که void را به عنوان پارامتر میگیرند و یک عدد int برمیگردانند.

پس داخل curly brackets تابع هایی با این نوع باید قرار بگیرد یعنی : int function (void)

و این نوع فرم هم دقیقا همانی است که سیستم کال های sys_fork, sys_exit, sys_wait و ... دارند. و کلمه هایی که سمت چپ سیستم کال ها قرار دارند به عنوان index ارایه استفاده میشوند.

اعداد این اندیس ها هم در فایل syscall.h قرار میگیرند.

syscall.h

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

تابع syscall() مقدار رجیستر eax که ما در usys.S قرارداده بودیم را به همراه شماره ی SYS_ میگیرد و sys_syscall را از ارایه فراخوانی میکند و نتیجه را در رجیستر eax میریزد.

sysproc.c

```
int
sys_kill(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return kill(pid);
}
```

میتوانیم declaration تابع int sys_kill(void) را در فایل syscall.c پیدا کنیم و همچنین پیاده سازی ان را هم در فایل sysproc.c

این تابع یک مقدار int را از process stack میگیرد و مقدارش را در متغیر pid ذخیره میکند. اگه همه چیز کار کند در نهایت تابع را کال میکند و تابع هم پروسه ای با همان ایدی را پیدا میکند و ان را kill میکند.