

Profesor: Antonio Flores Gil

Subgrupo: 3.2

Alumnos:

Cándida Barba Ruiz

candida.barba@um.es

Samuel Sánchez Álvarez

samuel.sanchez1@um

Práctica 2 - ASO

Llamadas al sistema y
reserva tardía de página en
xv6

Índices

Tabla de contenido

Boletín 7	2
EJERCICIO 1.....	2
EJERCICIO 2.....	3
EJERCICIO 3.....	4
EJERCICIO 4.....	5
Boletín 8	7
EJERCICIO 1.....	7
EJERCICIO 2.....	8

Tabla de figuras

Figura 1 Inicialización del IDT (depurador)	2
Figura 2 Captura gdb (llamada al sistema)	3

Boletín 7

Llamadas al sistema en xv6

EJERCICIO 1

¿Cuál es la rutina de servicio asociada a la interrupción 64?

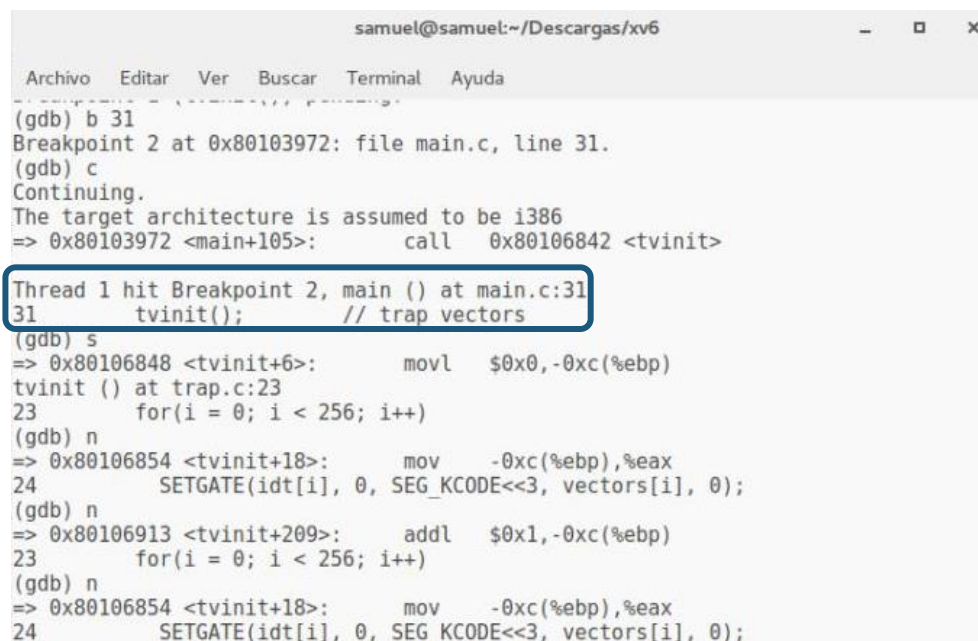
(Nota: Utiliza el depurador para ver cómo se inicializa la IDT y que manejador se asocia al trap de las llamadas al sistema)

La rutina de servicio asociada a esa instrucción es la syscall, que nos lleva a todas las llamadas al sistema. En la función tvinit() de trap.c se inicializa el IDT creando un gestor de interrupciones con la macro SETGATE y el descriptor de gate T_SYSCALL, que como podemos comprobar en el fichero mmu.h, es una constante global definida con el valor 64.

```
void
tvinit(void)
{
    inti;

    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

    initlock(&tickslock, "time");
}
```



```
samuel@samuel:~/Descargas/xv6
Archivo Editar Ver Buscar Terminal Ayuda
(gdb) b 31
Breakpoint 2 at 0x80103972: file main.c, line 31.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x80103972 <main+105>:      call    0x80106842 <tvinit>

Thread 1 hit Breakpoint 2, main () at main.c:31
31 tvinit(); // trap vectors
(gdb) s
=> 0x80106848 <tvinit+6>:      movl    $0x0, -0xc(%ebp)
tvinit () at trap.c:23
23 for(i = 0; i < 256; i++)
(gdb) n
=> 0x80106854 <tvinit+18>:      mov     -0xc(%ebp),%eax
24 SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
(gdb) n
=> 0x80106913 <tvinit+209>:      addl    $0x1, -0xc(%ebp)
23 for(i = 0; i < 256; i++)
(gdb) n
=> 0x80106854 <tvinit+18>:      mov     -0xc(%ebp),%eax
24 SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
```


Figura 1 Inicialización del IDT (depurador)

EJERCICIO 2

¿En qué punto del núcleo entra el kernel cuando se realiza una llamada al sistema?
¿Qué llamada al sistema se está ejecutando? ¿Qué se ha guardado en pila kernel del proceso desde que se ejecutó `int 64` hasta que se llama a la función `trap()` implementada en el fichero `trap.c`?

(Nota: Repasa el código de los ficheros `usys.S`, `vectors.Sytrapasm.S`)

En la Figura 1 mostramos una captura de pantalla en la que vemos la respuesta a las 2 primeras preguntas. En el punto `$0xf` entra el kernel, recoge el valor del mismo para llevarlo al registro `%eax`. Una vez en modo núcleo, realiza la llamada al sistema `SYSCALL(open)`. El manejador de interrupciones detecta que hay una llamada al sistema (interrupción 64), carga la llama al sistema e identifica que se trata de la llamada al sistema `open`.



```
samuel@samuel:~/Descargas/xv6
Archivo Editar Ver Buscar Terminal Ayuda
14 SYSCALL(pipe)
15 SYSCALL(read)
16 SYSCALL(write)
(gdb) c
Continuing.
=> 0x2e2 <main>: lea 0x4(%esp),%ecx
Thread 1 hit Breakpoint 1, main (argc=1, argv=0x2ff4) at ls.c:75
75 {
(gdb) s
=> 0x2f6 <main+20>: cmpl $0x1,(%ebx)
78 if(argc < 2){
(gdb) s
=> 0x2fb <main+25>: sub $0xc,%esp
79 ls(".");
(gdb) s
=> 0xc4 <ls+12>: sub $0x8,%esp
ls (path=0xb22 ".") at ls.c:33
33 if((fd = open(path, 0)) < 0){
(gdb) s
=> 0x5dd <open>: mov $0xf,%eax
open () at usys.S:20
20 SYSCALL(open)
(gdb)
```

Figura 2 Captura gdb (llamada al sistema)

En el fichero `usys.S` encontramos esta fracción de código que define `SYSCALL`.

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

En la pila se han almacenado los antiguos valores de los registros `%esp` y `%ss` desde el descriptor de segmentos de la tarea. También se almacenan los registros `%eflags`, `%cs` y `%eip`. En algunos casos, se guarda también en la pila una palabra de error.

EJERCICIO 3

Añade una nueva llamada al sistema (intdate (structrtcdte* d)) a xv6. El objetivo principal del ejercicio es que veas las diferentes piezas que componen la maquinaria de una llamada al sistema

- En syscall.h hay que darle un nuevo número a la llamada

```
#defineSYS_date    22
```

- En usys.S hay que añadir la llamada date

```
SYSCALL(date)
```

- En syscall.c hay que añadir la definición de la función sys_date(). Se agrega una nueva entrada en el array de llamadas al sistema de la siguiente forma [SYS_DATE] sys_date

```
externintsys_date(void);
staticint (*syscalls[])(void) = {
    .
    .
    .
    [SYS_date]    sys_date,
}
```

- En sysproc.c es donde se implementan las llamadas al sistema que se realizan desde syscall(). Hay que añadir la función sys_date() con su implementación
- En user.h se añade la llamada date() al fichero de definición de llamadas al sistema para los programas de usuario.

```
int date(structrtcdte*);
```

A continuación, mostramos el código añadido al fichero sysproc.c:

```
int
sys_date(void)
{
    structrtcdte *date;
    if(argptr(0, (void*)&date, sizeof(structrtcdte *)) < 0)
        return -1;
    cmostime(date);
    return 0;
}
```

EJERCICIO 4

Implementa la llamada al sistema dup2 () y modifica el shell para usarla (usa como ejemplo la implementación de la llamada al sistema dup() y consulta cómo debe comportarse dup2 () según el estándar POSIX)

Para implementar cualquier llamada al sistema se modifican los ficheros syscall.h, usys.S, user.h y syscall.c, para dup2 será necesario modificar también el fichero sysfile.c.

El código de syscall.c incluye un método llamado syscall() que, dependiendo del número que esté en el registro eax del trapframe del proceso, lanzará una llamada al sistema. Esta llamada proviene de la lista de syscalls, que se encuentran en un array de funciones a las llamadas del sistema.

Para añadir una nueva llamada al sistema tenemos que:

1. Añadimos un nuevo número que represente esa llamada al sistema, agregando un nuevo número en el fichero syscall.h.

```
#define SYS_dup2    23
```

2. Añadimos a usys.S la nueva llamada al sistema. También añadimos en user.h el método de la llamada del sistema, esto permite que esa llamada se pueda realizar en modo usuario.

```
SYSCALL(dup2)
```

```
int dup2(int,int);
```

3. Añadimos la definición de la función que representa la llamada al sistema en syscall.c. Se agrega una nueva entrada en el array de llamadas al sistema de la siguiente forma [SYS_DUP2] sys_dup2.

```
extern int sys_dup2(void);

static int (*syscalls[])(void) = {
...
[SYS_dup2]    sys_dup2,
}
```

4. Realizamos la implementación de la llamada al sistema.

Para mantener un orden, se debe añadir la implementación de la llamada al sistema donde más sentido tenga. En el caso de dup2 hemos realizado la implementación de sys_dup2 en sysfile.c, ya que en este fichero se encuentran todas las implementaciones relacionadas con la gestión de descriptores de fichero del sistema operativo.

La implementación del dup2 consiste en:

1. Leemos los 2 descriptores de fichero que nos pasan como argumento (fd1 y fd2). Seguidamente comprobamos si ambos argumentos son el mismo descriptor de fichero, en caso afirmativo devolvemos el segundo argumento.

```
struct file *f1;
struct file *f2;
int fd1, fd2;
if(argfd(0, &fd1, &f1) < 0) return -1;
if(argint(1, &fd2) < 0) return -1;
if(fd1 == fd2) return fd2;
```

2. Comprobamos si el fichero f2 está en la lista de ficheros abiertos del proceso. Si el fichero está abierto lo cerramos.

```
if((f2 = proc->ofile[fd2]) != 0)
    fileclose(f2);
```

3. Finalmente, intercambiamos el fichero de la lista de ficheros abiertos del proceso (f2) con la posición del fichero f1, y llamamos a la función `filedup` para incrementar la referencia del fichero. Finalmente devolvemos el segundo descriptor de fichero.

```
proc->ofile[fd2] = f1;
filedup(f1);

return fd2;
```

Boletín 8

Reserva tardía de páginas en xv6

Nota: Para la realización de las pruebas de los ejercicios de este boletín se ha utilizado usertest.

EJERCICIO 1

Implementa esta característica de reserva diferida en xv6. Para ello:

1. Elimina la reserva de páginas de la llamada al sistema `sbrk(n)` (implementada a través de la función `sys_sbrk()` en `sysproc.c`)
2. Realiza las modificaciones, arranca xv6, y teclea `echo hola` en el shell:
3. El mensaje `pid 3 sh: trap...` proviene del manejador de traps definido en `trap.c`
4. Modifica el código en `trap.c` para responder a un fallo de página en el espacio de usuario mapeando una nueva página física en la dirección que generó el fallo, regresando después al espacio de usuario para que el proceso continúe

A continuación mostramos el código de la función `sys_sbrk` que hemos modificado en el fichero `sysproc.c`. El cambio se presenta en las 3 últimas líneas del código de la función que incrementa el tamaño del proceso sin reservar memoria.

```
int
sys_sbrk(void)
{
    int addr;
    int n;
    if(argint(0, &n) < 0)
        return -1;
    addr = proc->sz;
    proc->sz += n;
    return addr;
}
```

Seguidamente, mostramos el código que hemos añadido en el fichero `trap.c` para modificar el mismo y que respondiera a un fallo de página como se pedía. Aunque antes, añadimos algunos comentarios para facilitar la lectura del código:

- En la variable `address` guardamos la dirección de la página virtual.
- Se utilizan las funciones `kalloc()` y `memset()` para la reserva de memoria.
- Se realiza el mapeo con `mappages()`.

```
// In user space, we allocate a physical page
address = PGROUNDDOWN(rcr2());
char * mem;
mem = kalloc();
memset(mem, 0, sizeof(char*));
mappages(proc->pgdir, (char*)address, PGSIZE, V2P(mem), PTE_W|PTE_U);
```


EJERCICIO 2

Modifica el código del primer ejercicio para que contemple al menos **dos** de las situaciones siguientes:

- El caso de un argumento negativo al llamarse a `sbrk()`
- Verificar que `fork()` y `exit()` funciona en el caso de que haya direcciones virtuales sin memoria reservada para ellas

La modificación realizado para que se contemple el caso negativo al llamar a `sbrk()` se realiza en la función `sus_sbrk()`. Hemos añadido 2 “ifs”, en el primero comprobamos que se ha pasado un número (para filtrar un posible error si nos llega otro carácter) y en el segundo comprobamos que éste sea negativo. En caso de que el número que nos pasen sea negativo se actualiza la dirección (`addr`), se calcula un nuevo tamaño del proceso sumando el número negativo(`n`) al tamaño actual (`sz`), si este número es positivo se le asigna este nuevo tamaño al proceso (`proc`).

```
int
sys_sbrk(void) {
    int addr;
    int n;
    if(argint(0, &n) < 0){
        return -1; }
    if(n<0){
        addr = proc -> sz;
        int c=proc->sz+n;
        if(c >=0)
            proc -> sz =c;
    } else {
        addr = proc->sz;
        proc->sz+=n;
    }
    return addr;
}
```

Para asegurarnos de que `fork()` y `exit()` funcionan en el caso de que haya direcciones virtuales sin memoria reservada para ellas, se ha modificado la función `mappages()` suprimiendo la línea:

```
panic (" remap ");
```

Y sustituyéndola por la línea siguiente:

```
return 0;
```

Lo que se consigue con esto es que el proceso en lugar de entrar en panic, “ignore” la situación.

```
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm) {
    char *a, *last;
    pte_t *pte;
    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            return 0;
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0; }
```