Boletín 6: Arranque y Depuración de xv6

Ampliación de Sistemas Operativos

Dpto. Ingeniería y Tecnología de Computadores (DITEC)

Universidad de Murcia

Curso 2017/2018

El Proceso de Arranque de xv6

- El propósito de esta primera práctica es familiarizarnos con las herramientas que utilizaremos en el resto de prácticas:
  - QEMU/GDB y el Sistema Operativo xv6
- Usaremos el emulador QEMU para ejecutar el sistema operativo y utilizaremos GDB para depurar remotamente el sistema mientras se ejecuta
- Comenzaremos compilando xv6 mediante la orden `make`

```
$ make
....
dd if=/dev/zero of=xv6.img count=10000
10000+0 registros leídos
10000+0 registros escritos
5120000 bytes (5,1 MB, 4,9 MiB) copied, 0,0154357 s, 332 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 registros leídos
1+0 registros escritos
512 bytes copied, 0,000105803 s, 4,8 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
350+1 registros leídos
350+1 registros escritos
179424 bytes (179 kB, 175 KiB) copied, 0,000484832 s, 370 MB/s
```

El Proceso de Arranque de xv6 (cont.)

- Como se puede observar, tras la compilación, rellenamos con ceros la imagen del xv6 con la instrucción
  - `dd if=/dev/zero of=xv6.img count=10000`
- A continuación escribimos el sector de arranque, `bootblock`, en el primer sector de la imagen (donde buscará la BIOS)
  - `dd if=bootblock of=xv6.img conv=notrunc`
- Y el resto del kernel a partir del segundo sector
  - `dd if=kernel of=xv6.img seek=1 conv=notrunc`
- Podemos desensamblar el sector de arranque con la orden `objdump -d bootblock.o`:

```
Desensamblado de la sección .text:

00007c00 <start>:
    7c00:       fa                      cli
    7c01:       31 c0                   xor    %eax,%eax
    7c03:       8e d8                   mov    %eax,%ds
    7c05:       8e c0                   mov    %eax,%es
    7c07:       8e d0                   mov    %eax,%ss
```

El Proceso de Arranque de xv6 (cont.)

- Si volcamos el contenido del sector de arranque mediante la orden `hexdump bootblock` veremos que termina con la firma `0x55 0xaa` que lo identifica como un sector de arranque:

```
$ hexdump -C bootblock
00000000  fa 31 c0 8e d8 8e c0 8e  d0 e4 64 a8 02 75 fa b0  |.1........d..u..|
00000010  d1 e6 64 e4 64 a8 02 75  fa b0 df e6 60 0f 01 16  |..d.d..u....`...|
00000020  78 7c 0f 20 c0 66 83 c8  01 0f 22 c0 ea 31 7c 08  |x|. .f...."..1|.|
00000030  00 66 b8 10 00 8e d8 8e  c0 8e d0 66 b8 00 00 8e  |.f.........f....|
...
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200
```

- El punto de entrada al kernel es `_start`. Podemos encontrar la dirección del mismo con el comando `nm`:

```
$ nm kernel | grep _start
8010b4ec D _binary_entryother_start
8010b4c0 D _binary_initcode_start
0010000c T _start
```

- El siguiente paso será ejecutar el kernel dentro de `QEMU GDB`, estableciendo un *breakpoint* en `_start`

El Proceso de Arranque de xv6 (cont.)

- Configura gdb para permitir la carga automática de ficheros:
  - Añadir al fichero `.gdbinit` la línea `set auto-load safe-path /`:

```
$ echo 'set auto-load safe-path /' >> ~/.gdbinit
```

- Desde la consola ejecuta `make qemu-gdb`

```
$ make qemu-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,
     format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -
     m 512  -S -gdb tcp::26000
```

El Proceso de Arranque de xv6 (cont.)

- En una nueva terminal entra al directorio del xv6 y ejecuta gdb

```
$ gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
...
+ target remote localhost:26000
aviso: A handler for the OS ABI "GNU/Linux" is not built into this
     configuration
of GDB.  Attempting to continue with the default i8086 settings.

Se asume que la arquitectura objetivo es i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb)
```

- El *stub* de depuración remota de QEMU para la máquina virtual antes de ejecutar la primera instrucción
  - Antes incluso de que se empiece a ejecutar cualquier código de la BIOS
- El procesador se encuentra en modo real en la dirección `0xfff0` (direccción de la primera instrucción a ejecutar en la arquitectura x86 tras un *reset*)

El Proceso de Arranque de xv6 (cont.)

- Añade un breakpoint en `_start` y continua la ejecución del programa

```
(gdb) b _start
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

Breakpoint 1, 0x0010000c in ?? ()
(gdb)
```

- **EJERCICIO 1**: ¿Cual es el contenido de la pila en este punto?

```
(gdb) info reg
...
(gdb) x/24x $esp
...
(gdb)
```

  - Para responder a la pregunta consulta los ficheros `bootasm.S`, `bootmain.c` y `bootblock.asm` (que contiene la salida del compilador/ensamblador)

El Proceso de Arranque de xv6 (cont.)

- Observa dónde se inicializa la pila en el arranque y sigue los cambios que sufre hasta que llega al *breakpoint*. Para ello:
  1. Reinicia `qemu` y `gdb`, y pon un *breakpoint* en `0x7c00`, el comienzo del bloque de arranque (`bootasm.S`). Ejecuta paso a paso las instrucciones (escribe `si` en `gdb`, o bien `file bootblock.o` y se ejecutan instrucciones con `n`). ¿Dónde se inicializa el puntero de pila?
  2. Continúa paso a paso hasta la llamada a `bootmain()` para determinar dónde se inicializa la pila; ¿qué hay en la pila después de realizar la llamada?
  3. ¿Qué hace con la pila las primeras instrucciones de `bootmain()`? Mira en `bootblock.asm`
  4. Continúa trazando el arranque del xv6 y busca la llamada que cambia el valor de `eip` a `0x10000c`. ¿Cómo afecta la llamada a la pila? (Piensa en lo que esta llamada está intentando hacer en la secuencia de arranque e intenta identificar este punto en `bootmain.c`, y la instrucción correspondiente en `bootblock.asm`)

El Proceso de Arranque de xv6 (cont.)

- Continúa con la ejecución del proceso de inicialización del *kernel* hasta llegar al punto en donde se crea el primer proceso (`userinit()`). Para ello pon un *breakpoint* en la función `main()` y continúa la ejecuación del *kernel*:

```
(gdb) file kernel
(gdb) b main
Punto de interrupción 1 at 0x8010385d: file main.c, line 19.
(gdb) c
Continuando.
Se asume que la arquitectura objetivo es i386
=> 0x8010385d <main>:   lea    0x4(%esp),%ecx

Breakpoint 1, main () at main.c:19
19  {
```

- Observa como se van llamando a las distintas funciones que inicializan las diversas estructuras que conforman el sistema operativo:

El Proceso de Arranque de xv6 (cont.)

```
(gdb) list main
18  main(void)
19  {
20    kinit1(end, P2V(4*1024*1024)); // phys page allocator
21    kvmalloc();      // kernel page table
22    mpinit();        // detect other processors
23    lapicinit();     // interrupt controller
24    seginit();       // segment descriptors
25    picinit();       // disable pic
26    ioapicinit();    // another interrupt controller
27    consoleinit();   // console hardware
28    uartinit();      // serial port
29    pinit();         // process table
30    tvinit();        // trap vectors
31    binit();         // buffer cache
32    fileinit();      // file table
33    ideinit();       // disk
34    startothers();   // start other processors
35    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
36    userinit();      // first user process
37    mpmain();        // finish this processor's setup
38  }
```

- Añade otro breakpoint en la función `userinit()` y continua con la depuración del kernel

El Proceso de Arranque de xv6 (cont.)

```
(gdb) b userinit
Punto de interrupción 2 at 0x801043b7: file proc.c, line 122.
(gdb) c
Continuando.
=> 0x801043b7 <userinit>:   push   %ebp

Breakpoint 2, userinit () at proc.c:122
122 {
```

- **EJERCICIO 2**: Describe a grandes rasgos cuál es el proceso de creación de este primer proceso
  - La función encargada de crear la pila de núcleo del proceso es `allocproc()` definida en en fichero `proc.c` que nos devolverá una estructura proceso parcialmente inicializada. La parte del código en donde se crea dicha pila es:

El Proceso de Arranque de xv6 (cont.)

```
// Allocate kernel stack.
if((p->kstack = kalloc()) == 0){
  p->state = UNUSED;
  return 0;
}
sp = p->kstack + KSTACKSIZE;
```

- En dicha función también se deja espacio para el *trap frame*, y se hace coincidir con la estructura `trapframe`:

```
// Leave room for trap frame.
sp -= sizeof *p->tf;
p->tf = (struct trapframe*)sp;
```

- Asi como el punto en donde comenzará a ejecutarse dicho hilo y el espacio para el contexto

```
// Set up new context to start executing at forkret,
// which returns to trapret.
sp -= 4;
*(uint*)sp = (uint)trapret;

sp -= sizeof *p->context;
p->context = (struct context*)sp;
memset(p->context, 0, sizeof *p->context);
p->context->eip = (uint)forkret;
```

El Proceso de Arranque de xv6 (cont.)

- ¿Dónde comienza a ejecutarse dicho hilo? Indica la dirección de la primera instrucción que se ejecuta
- Al terminar la ejecución de `allocproc()` tendremos en la variable `p` la estructura del proceso inicial ya parcialmente inicializada:

```
(gdb) print *p
$3 = {sz = 0, pgdir = 0x0, kstack = 0x8dfff000 "", state = EMBRYO, pid =
    1,
  parent = 0x0, tf = 0x8dffffb4, context = 0x8dffff9c, chan = 0x0, killed
      = 0,
  ofile = {0x0 <repeats 16 times>}, cwd = 0x0,
  name = '\000' <repetidos 15 veces>}
```

- ¿En qué fichero se encuentra el código a ejecutar? Observa que en `userinit()` se inicializa la zona de usuario del proceso de la siguiente forma:

```
inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
```

- ¿Cuál es el contenido de la pila kernel en el momento previo al paso a modo usuario? Observa como se termina de inicializar el proceso en `userinit()` e inspecciona el contenido de la pila de kernel con gdb

El Proceso de Arranque de xv6 (cont.)

```
(gdb) print *p
$4 = {sz = 4096, pgdir = 0x8dffe000, kstack = 0x8dfff000 "", state =
      RUNNABLE,
  pid = 1, parent = 0x0, tf = 0x8dffffb4, context = 0x8dffff9c, chan = 0x0
        ,
  killed = 0, ofile = {0x0 <repeats 16 times>}, cwd = 0x80111a94 <icache
       +52>,
  name = "initcode\000\000\000\000\000\000\000"}
(gdb) print ...
```

- ¿Cómo se regresa al modo usuario? Para ello añade un *breakpoint* en `forkret()` y continua la ejecución del kernel:

```
(gdb) b forkret
Punto de interrupción 3 at 0x80104af7: file proc.c, line 398.
(gdb) c
Continuando.
=> 0x80104af7 <forkret>:    push   %ebp

Breakpoint 3, forkret () at proc.c:398
398 {
(gdb) list
393
394 // A fork child's very first scheduling by scheduler()
395 // will swtch here.  "Return" to user space.
396 void
397 forkret(void)
398 {
```

El Proceso de Arranque de xv6 (cont.)

```
399   static int first = 1;
400   // Still holding ptable.lock from scheduler.
401   release(&ptable.lock);
402
```

- ¿Qué código se ejecuta en dicho proceso? ¿Qué llamada al sistema realiza?
- ¿Cuál es el contenido de la pila de usuario en el momento previo a realizar la llamada al sistema comentada anteriormente? (Nota: mira el contenido del fichero `initcode.asm`)