Boletín xv6-1: Arranque y Depuración de xv6

Ampliación de Sistemas Operativos

Dpto. Ingeniería y Tecnología de Computadores (DITEC)

Universidad de Murcia

Curso 2021/2022

Índice

1. Introducción

El Proceso de Arranque de xv6 I

- El propósito de esta primera práctica será el de estudiar el arranque de xv6 y el de 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

El Sistema Operativo xv6

- Xv6 es un sistema operativo inspirado en UNIX V6 y desarrollado por el MIT en el 2006 con el objetivo de ser usado en la enseñanza
- Se trata de sistema operativo con un kernel monolítico en donde todas las llamadas al sistema se ejecutan en modo núcleo y, por tanto, todo el sistema operativo se ejecuta con acceso total al hardware
- Xv6 se compila usando el compilador de C GNU, generando binarios en formato ELF para la arquitectura x86
- Aunque xv6 puede arrancar sobre hardware real, lo usaremos sobre el emulador QEMU

De modo real a modo protegido I

- Después de que el gestor de arranque IA32 carga el kernel, debe cambiar del modo de direccionamiento real a modo protegido
  - Esto es técnicamente *código de inicio del núcleo*
- **Paso 1**: El código de inicio debe volver a habilitar la línea de dirección A20 para poder acceder >1MB de la memoria

  - Algunos gestores de arranque en sus estadios iniciales se encargan de esto (por ejemplo GRUB)
  - Algunas BIOS también se ocupan de esto
  - Se tiene que comprobar si A20 está desactivada, y si es así, volver a activarla
- **Paso 2**: El código de inicio debe configurar el modo protegido, los segmentos de memoria y el sistema de memoria virtual
  - Como mínimo, debe inicializar los descriptores de segmento del código y los datos del kernel y establecer `%cs`, `%ds`, `%es` y `%ss`
  - También puede desear establecer la tabla de páginas para la traducción de direcciones lineales a físicas

De modo real a modo protegido II

- **Paso 3**: Cambiar de modo real a modo protegido
  - Este paso es más complicado de lo que parece ...
- A grandes rasgos, el proceso es el siguiente (se han omitido algunos detalles):
  1. ¡Deshabilitar las interrupciones! Si se produce alguna interrupción durante la transición se desatará el caos
  2. Cargar el registro de la tabla global de descriptores (GDTR) con un puntero a la GDT que contiene descriptores de los segmentos del Sistema Operativo
  3. Cargar el Registro de Tareas (`%tr`) con un segmento de estado de tarea, para que el manejo de interrupciones funcione correctamente en modo protegido
  4. Activar el modo protegido (y, opcionalmente, permitir el sistema de memoria virtual paginada) escribiendo en el registro de control `%cr0`
     - Si está habilitada la paginación, también debe configurar una tabla de páginas inicial haciendo que el registro `%cr3` apunte a ella
     - En este punto, el modo protegido ya está habilitado, pero la CPU sigue ejecutando un segmento en modo real de 16 bits almacenado en la caché de la CPU

De modo real a modo protegido III

5. Forzar la CPU para cargar los selectores de segmento de 32 bits en modo protegido mediante la realización de un salto largo a la siguiente instrucción
   - El salto largo especifica un nuevo valor del selector de segmento del código del núcleo, que también carga en `%cs` el selector de segmento
6. Establecer los demás registros de segmento al valor del selector del segmento de datos del núcleo
7. Cargar la tabla de descriptores de registro de interrupción (IDTR) con un puntero a la tabla de descriptores de interrupción para el sistema operativo
8. ¡Volver a habilitar las interrupciones!

- Una vez hecho esto el núcleo del sistema operativo está listo para tomar el control

2. Depuración del arranque de xv6

Compilación e inicio de xv6 I

- 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
```

- 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`

Compilación e inicio de xv6 II

- 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
```

- 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:

Compilación e inicio de xv6 III

```
$ 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 cargador del xv6, **bootblock**, se construye compilando dos ficheros fuente:
  - El **bootasm.S** escrito en una combinación de ensamblador de x86 de 16 y 32 bits y que la BIOS carga a partir de la dirección física **0x7c00** y que es el encargado de pasar a modo protegido activando la segmentación
  - Y el **bootmain.c** escrito en C y que es el encargado de cargar en memoria el kernel del sistema operativo

Compilación e inicio de xv6 IV

- 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
```

- Podemos comprobar que se trata del punto de entrada ejecutando el comando:

```
$  readelf -h kernel
Encabezado ELF:
  Mágico:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  ...
  Dirección del punto de entrada:                0x10000c
  Inicio de encabezados de programa:          52 (bytes en el fichero)
  Inicio de encabezados de sección:          160424 (bytes en el fichero)
  ...
```

Compilación e inicio de xv6 V

- El siguiente paso será ejecutar el kernel dentro de **QEMU GDB**. Para ello:
  - 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
```

Compilación e inicio de xv6 VI

- 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*)

Compilación e inicio de xv6 VII

- Añade un *breakpoint* en `0x7c00`, justo en el comienzo del bloque de arranque (en el fichero `bootasm.S`)
- A continuación carga la tabla de símbolos del sector de arranque (escribe `file bootblock.o` en el `gdb`) y ejecuta las instrucciones paso a paso con `s`).

El * indica que es una direccion particular que le especificamos nosotros

```
(gdb) b *0x7c00
Punto de interrupción 1 at 0x7c00
(gdb) file bootblock.o
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
¿Cargar una tabla de símbolos nueva desde «bootblock.o»? (y or n) y
Leyendo símbolos desde bootblock.o...hecho.
(gdb) c
Continuando.
[   0:7c00] => 0x7c00 <start>:  cli

Thread 1 hit Breakpoint 1, start () at bootasm.S:13
13        cli                           # BIOS enabled interrupts; disable
```

Compilación e inicio de xv6 VIII

- Nos encontramos ejecutando el código que pasará la máquina a modo protegido. Para ello:
  1. Primero se deshabilitan las interrupciones y se inicializan los segmentos de datos:

```
#include "asm.h"
#include "memlayout.h"
#include "mmu.h"

# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment
```

Compilación e inicio de xv6 IX

2. A continuación de habilita el bit de direcciones **A20** usando los puertos de E/S del teclado:

```
  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

Compilación e inicio de xv6 X

3. Seguidamente pasamos a modo protegido de 32 bits:

```
# Switch from real to protected mode.  Use a bootstrap GDT that makes
# virtual addresses map directly to physical addresses so that the
# effective memory map remains unchanged during the transition
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0

# Complete the transition to 32-bit protected mode by using a long jmp
# to reload %cs and %eip.  The segment descriptors are set up with no
# translation, so that the mapping is still the identity mapping
ljmp    $(SEG_KCODE<<3), $start32
```

donde la descripción de la GDT es la siguiente (cada entrada ocupa 8 bytes):

```
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

Compilación e inicio de xv6 XI

4. Finalmente inicializamos los registros de segmento de datos en modo protegido y saltamos a la función en C **`bootmain()`**

```
.code32  # Tell assembler to generate 32-bit code now
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C
  movl    $start, %esp
  call    bootmain
```

- Continúa paso a paso hasta la llamada a **`bootmain()`**:

```
(gdb) s
=> 0x7c48 <start32+23>: call   0x7d3b <bootmain>
66        call    bootmain
(gdb) s
=> 0x7d44 <bootmain+9>: push   $0x0
bootmain () at bootmain.c:28
28        readseg((uchar*)elf, 4096, 0);
```

Compilación e inicio de xv6 XII

- La función **bootmain()** carga una imagen de kernel en formato ELF a partir del sector 1 del disco y a continuación salta a la rutina de entrada del kernel

```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  ...

  // Call the entry point from the ELF header
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
```

Compilación e inicio de xv6 XIII

- Añade otro breakpoint en `_start` y continua la ejecución del programa. Para ello carga la tabla de símbolos del kernel y a continuación añade el breakpoint

```
(gdb) file kernel
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
¿Cargar una tabla de símbolos nueva desde «kernel»? (y or n) y
Leyendo símbolos desde kernel...hecho.
(gdb) b _start
Punto de interrupción 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

Breakpoint 2, 0x0010000c in ?? ()
(gdb)
```

Compilación e inicio de xv6 XIV

- Carga la tabla de símbolos de código de entrada del kernel y lista el contenido a partir del punto de entrada al kernel:

```
(gdb) file entry.o
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
¿Cargar una tabla de símbolos nueva desde «entry.o»? (y or n) y
Leyendo símbolos desde entry.o...hecho.
(gdb) list entry
42
43      # Entering xv6 on boot processor, with paging off.
44      .globl entry
45      entry:
46        # Turn on page size extension for 4Mbyte pages
47        movl    %cr4, %eax
48        orl     $(CR4_PSE), %eax
49        movl    %eax, %cr4
50        # Set page directory
51        movl    $(V2P_WO(entrypgdir)), %eax
```

La configuración de la memoria virtual en xv6 I

- El kernel se carga a partir de la dirección física `0x100000` (justo después del primer mega de memoria) ya que la paginación todavía no está habilitada (dir. físicas = dir. virtuales)

  - El kernel residirá en todo momento en la dirección virtual a partir de KERNBASE, (`0x80000000`)
  - No podemos usar la dirección `0x80100000` (no sabemos si la máquina tendrá tanta memoria física)
  - Tampoco a partir de la dirección `0x0` ya que el rango `0xa0000:0x100000` contiene los dispositivos de E/S
- El proceso de arranque continua a partir de `_start` definido en el fichero `entry.S`
- Para ejecutar el resto del kernel, `entry` crea una tabla de páginas que mapea dir. virtuales a partir de la `0x80000000`, KERNBASE, a las dir. físicas a partir de la direción `0x0`
  - Tendremos dos rangos de direcciones virtuales apuntando al mismo rango de direcciones físicas

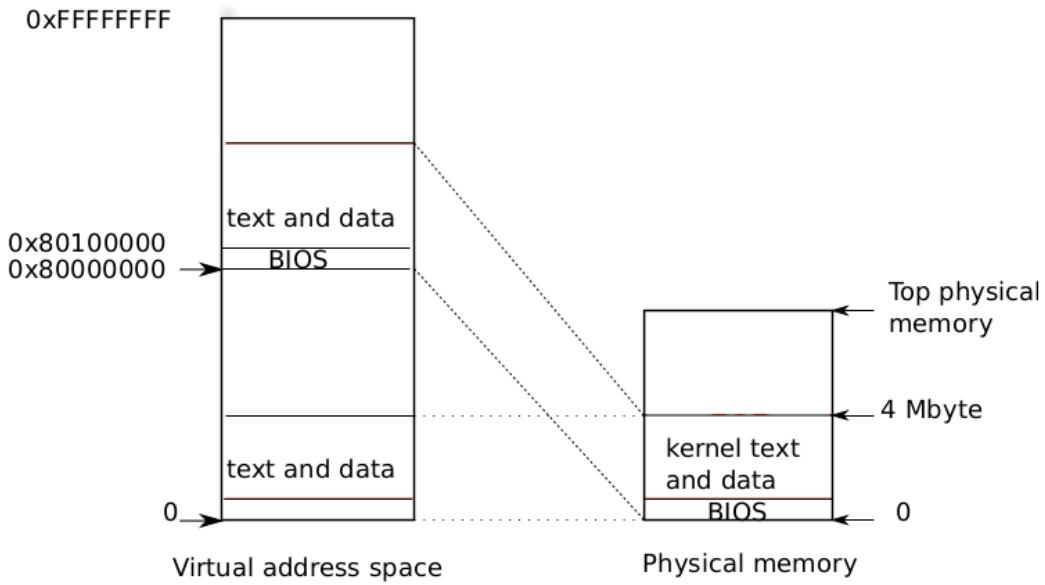
La configuración de la memoria virtual en xv6 II

```
# Entering xv6 on boot processor, with paging off
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

La configuración de la memoria virtual en xv6 III

- donde **entrypgdir** se encuentra definido en el fichero **main.c** de la siguiente forma:

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```



Proceso de inicialización del kernel I

- Carga de nuevo la tabla de símbolos del kernel y añade otro breakpoint a la función `main()`:

```
(gdb) file kernel
(gdb) b main
Punto de interrupción 3 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 3, main () at main.c:19
19      {
```

Proceso de inicialización del kernel II

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

```
(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      }
```

Inicialización de las interrupciones I

- Un aspecto importante en en el proceso de inicialización es la programación del IOAPIC (p.e., se especifica que el temporizador lance interrupciones periódicas)
- También se inicializa todos los vectores de interrupción cuando se llama a la función `tvinit()`

```
void
tvinit(void)
{
  int i;

  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");
}
```

- Inicialmente rellena los valores de `idt` con la entrada de `vectors` correspondiente que podemos consultar en el fichero `vectors.S`
- La entrada para el `T_SYSCALL` es especial

Inicialización de las interrupciones II

- Finalmente, se carga con la instrucción `lidt`:

```
void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

3. Implementación de procesos e hilos en xv6

Implementación de procesos e hilos en xv6 I

- La unidad de aislamiento en `xv6` es el proceso
- La abstracción de proceso permite evitar que un proceso pueda acceder a los datos de otro proceso y del kernel
- Los mecanismos utilizados por el núcleo para implementar dicha abstracción incluye el flag de modo usuario/núcleo, los espacios de direcciones y la división en el tiempo de los hilos
- El núcleo de xv6 mantiene en la estructura `proc` la información relativa al estado de cada proceso. Entre los aspectos más importantes que se guardan en esta estructura está la tabla de páginas, su pila kernel y su estado de ejecución

Implementación de procesos e hilos en xv6 II

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

- Cada proceso tiene un hilo de ejecución que ejecuta las instrucciones del proceso
- La mayor parte del estado de un hilo (variables locales, dirección de retorno de la función) se almacena en las pilas del hilo

Implementación de procesos e hilos en xv6 III

- Cada proceso tiene dos pilas
  - Una pila de usuario que se usa cuando el proceso está en modo usuario
  - Una pila kernel que se usa cuando el proceso entra en el kernel

4. Depurando la creación el primer proceso en xv6

La creación del primer proceso en xv6 I

- Tras la inicialización de varios dispositivos y subsistemas el kernel pasa a crear el primer proceso llamando a **`userinit()`**
- Añade otro breakpoint en la función **`userinit()`** y continúa con la ejecución del kernel. Cuando llegues al breakpoint entra dentro de la función para estudiar cómo xv6 crea el primer proceso

```
(gdb) b userinit
Punto de interrupción 4 at 0x8010321a: file proc.c, line 122.
(gdb) c
Continuando.
=> 0x8010321a <userinit>:       push   %ebp

Thread 1 hit Breakpoint 4, userinit () at proc.c:122
122     {
(gdb)
```

La creación del primer proceso en xv6 II

- `userinit()` llama a `allocproc()`, que busca una entrada libre en la tabla de procesos e inicializa las partes del estado del proceso necesarias para que se ejecute su hilo *kernel*

```
static struct proc* allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);
```

La creación del primer proceso en xv6 III

- A continuación reserva una pila kernel para el hilo *kernel* del proceso

```
// 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;
```

La creación del primer proceso en xv6 IV

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

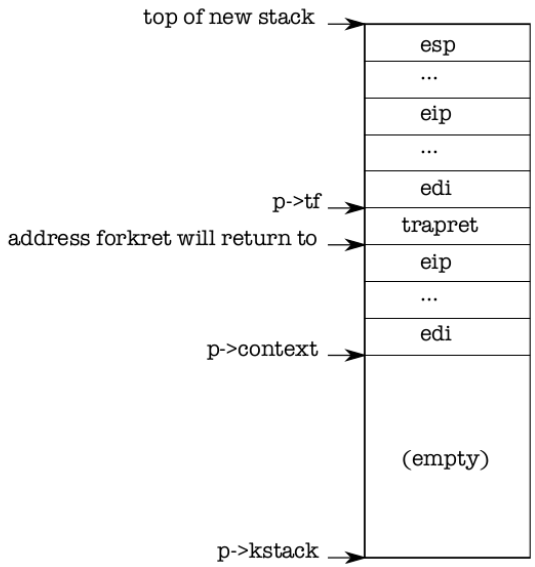
```
// Set up new context to start executing at forkret,
// which returns to trapret.
sp -= 4;
*(uint*)sp = (uint)trapret;

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

- Al terminar la ejecución de `allocproc()` tendremos en la variable `p` la estructura del proceso inicial ya parcialmente inicializada:

```
(gdb) print *p
$1 = {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>}
```

La creación del primer proceso en xv6 V

- A la derecha se muestra el estado final de la pila tras la inicialización
  - Inicializando `p->context->eip` a `forkret` hace que el hilo kernel empiece su ejecución al comienzo de `forkret`
  - Terminado `forkret` regresará a `trapret` que restaura los registros de usuario y salta al código del proceso



- El primer proceso ejecuta un pequeño programa (`initcode.S`)
  - Este programa necesita memoria física en donde copiarse y una tabla de páginas que mapee las direcciones de usuario a dicha memoria
  - `userinit()` llama a `setupkvm()` para crear una tabla de páginas para el proceso. Posteriormente carga el binario del programa y deja todo preparado para que el proceso se pueda ejecutar

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

La creación del primer proceso en xv6 VI

- Finalmente marcamos el proceso como **RUNNABLE**. El contenido la estructura proceso en este punto es:

```
(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 ...
```

- El cometido de este primer programa es, como se puede observar, ejecutar el proceso `init` mediante la llamada al sistema `exec()` (sería equivalente a ejecutar `exec("/init", argv)`)

La creación del primer proceso en xv6 VII

```
#include "syscall.h"
#include "traps.h"

# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

La creación del primer proceso en xv6 VIII

- Cuando el planificador entre en acción, verá que existe un proceso listo para ser ejecutado. Se realizará un cambio de contexto y se la pasará el control al mismo
- Añade un breakpoint dentro del planificador y observa como se pasa el control al proceso recién creado:

```
(gdb) list scheduler
      ...
335         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336           if(p->state != RUNNABLE)
337             continue;
338
(gdb)
339           // Switch to chosen process.  It is the process's job
340           // to release ptable.lock and then reacquire it
341           // before jumping back to us.
342           c->proc = p;
343           switchuvm(p);
344           p->state = RUNNING;
345
346           swtch(&(c->scheduler), p->context);
347           switchkvm();
348
(gdb) b 342
Punto de interrupción 5 at 0x8010349e: file proc.c, line 342.
```

La creación del primer proceso en xv6 IX

```
(gdb) c
Continuando.
=> 0x8010349e <scheduler+41>:   mov    %ebx,0xac(%esi)

Thread 1 hit Breakpoint 5, scheduler () at proc.c:342
342           c->proc = p;
(gdb) n
=> 0x801034a4 <scheduler+47>:   sub    $0xc,%esp
343           switchuvm(p);
(gdb) print *p
$3 = {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 = 0x8010fa14 <icache+52>,
  name = "initcode\000\000\000\000\000\000\000"}
(gdb)
```

La creación del primer proceso en xv6 X

- El punto en donde empezará a ejecutarse será el que guardamos en la pila (`forkret`). Para comprobarlo añade un *breakpoint* en `forkret()` y continúa la ejecución del kernel:

```
(gdb) b forkret
Punto de interrupción 6 at 0x80103122: file proc.c, line 398.
(gdb) c
Continuando.
=> 0x80103122 <forkret>:        push   %ebp

Thread 1 hit Breakpoint 6, 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     {
399       static int first = 1;
400       // Still holding ptable.lock from scheduler.
401       release(&ptable.lock);
402
```

La creación del primer proceso en xv6 XI

- Tras terminar **`forkret()`** regresaremos a **`trapret()`** (véase **`allocproc()`**) que está definida en **`trapasm.S`**:

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

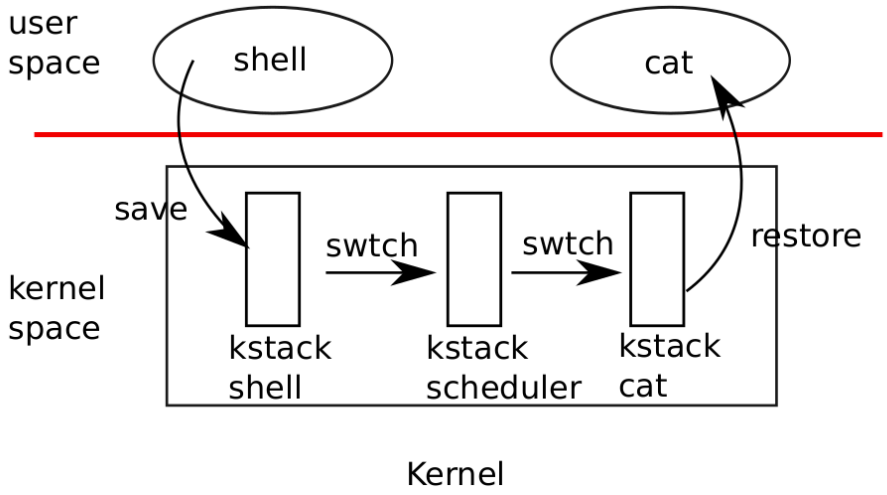
- Momento en que regresaremos a modo usuario y ejecutaremos la instrucción almacenada en la dirección 0 del espacio de memoria virtual del proceso (que es donde comienza nuestro programa)

5. Planificación de procesos en xv6

Planificación de procesos en xv6 I

- `xv6` proporciona a cada proceso la ilusión de que tiene su propio procesador virtual mediante la multiplexación de los procesos sobre los procesadores existentes
- `xv6` implementa la multiplexación cambiando de proceso en dos situaciones:
  - Los mecanismos `sleep` y `wakeup` permiten dicho cambio cuando un proceso espera a que un dispositivo o tubería finalice, está esperando a que termine un hijo o se encuentra esperando en la llamada al sistema `sleep()`
  - De forma periódica `xv6` fuerza dicho cambio cuando un proceso está ejecutando instrucciones de usuario
- La figura de la siguiente transparencia muestra, para el caso de un único procesador, los pasos necesarios para cambiar de un proceso de usuario a otro
  - Una transición de modo usuario a kernel (llamada al sistema o interrupción) al hilo del núcleo del proceso antiguo
  - Un cambio de contexto al hilo de planificación de la CPU local
  - Otro cambio de contexto al hilo del núcleo del nuevo proceso
  - Un regreso de *trap* al modo usuario del proceso

Planificación de procesos en xv6 II



- `xv6` usa dos cambios de contexto porque el planificador tiene su propia pila, lo que simplifica el proceso
- El cambio de un hilo a otro implica salvar los registros de CPU del hilo antiguo y restaurar los registros, previamente salvados, del nuevo hilo
- El hecho de guardar y restaurar los registros `%esp` y `%eip` implica que cambia el código que se está ejecutando
- `swtch()` no tiene noción acerca de los hilos, solamente guarda y restaura un conjunto de registros, denominados contextos

Planificación de procesos en xv6 III

- Cada contexto se representa por una `struct context*`, un puntero a una estructura almacenada en la pila kernel implicada

```
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

- Cuando un proceso tiene que ceder la CPU, el hilo *kernel* del proceso llama a `swtch()`, implementada en el fichero `swtch.S`, para salvar su propio contexto y regresar al contexto del planificador

Planificación de procesos en xv6 IV

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Planificación de procesos en xv6 V

- Los pasos a seguir cuando un proceso quiere liberar la CPU son los siguientes:
  - Adquirir el cerrojo que proteje la tabla de procesos (`ptable.lock`)
  - Liberar cualquier otro cerrojo que hubiera adquirido
  - Actualizar su propio estado (`myproc()->state`)
  - Llamar a `sched()`
- Tanto `yield()` como `sleep()` y `exit()` siguen esta convención

```
// Give up the CPU for one scheduling round
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

- `sched()` comprueba de nuevo dichas condiciones y además comprueba que las interrupciones están deshabilitadas (ya que se ha adquirido un cerrojo)
- Finalmente, `sched()` llama a `swtch()` para almacenar el contexto actual en `proc->context` y cambiar al contexto del planificador en `cpu->scheduler`

Planificación de procesos en xv6 VI

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

- Al producirse el cambio de contexto se regresa a `scheduler()` en el punto en que se llamó a `swtch()`
- La función `scheduler()` continua ejecutando el bucle `for` buscando un proceso que ejecutar y cambiando a dicho proceso

Planificación de procesos en xv6 VII

- La función `scheduler()` adquire el cerrojo `ptable.lock` para la mayoría de sus acciones, pero lo libera (y habilita las interrupciones) una vez por cada iteración de su bucle externo
- Esto es importante en el caso en que la CPU está desocupada para permitir que otras CPU adquieran la tabla de procesos y puedan marcar alguno de ellos como `RUNNABLE` o para permitir que se procesen las interrupciones de E/S
- Una vez encontrado un proceso, actualiza la variable `proc` y cambia a la tabla de páginas del proceso a través de `switchuvm()`, marca el proceso como `RUNNING` y llama a `swtch()` para empezar su ejecución

Planificación de procesos en xv6 VIII

```
// Per-CPU process scheduler
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor
    sti();

    // Loop over process table looking for process to run
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, p->context);
```

Planificación de procesos en xv6 IX

```
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```