Sistemas Operativos
Cambios de contexto en XV6

Marcelo Arroyo
FCEFQyN
Universidad Nacional de Río Cuarto

2017

Resumen

Una de las partes mas intrincadas y llena de detalles de bajo nivel en un sistema operativo es la implementación del mecanismo de *cambios de contexto*, es decir cuando se le quita el control (se le des-asigna la CPU) a un proceso para dárselo a otro.

En éste pequeño artículo se describen los detalles de su implementación en *xv6*, un pequeño sistema operativo utilizado en educación. En particular, se describen los detalles sobre la arquitectura `x86` (*IA32* para ser precisos).

1. Introducción

En ésta sección se describen las principales estructuras de datos usadas por el *kernel* de *xv6*[1] para la gestión de procesos.

Durante la inicialización del sistema (ver `main.c`), se definen varias estructuras de datos globales que son usadas directamente por la(s) CPU(s). En particular, y para lo que nos interesa, las estructuras de datos a que haremos referencia son:

- **La tabla de procesos** `ptable`: Definida en `proc.c`, es un arreglo (estático) donde cada elemento es de tipo `struct proc` (ver `proc.h`). Cada entrada de la tabla describe el estado de un proceso. Una entrada i con el campo $ptable.proc[i].state$ igual a `UNUSED`, representa una entrada libre.
- **El vector de interrupciones** `idt`: Definido en `trap.c`, es un arreglo de 256 elementos de tipo `gatedesc`.

  Cada entrada (`gatedesc`) describe una entrada del vector de interrupciones, denominadas *call gates* en `x86`. La entrada i contiene la

dirección de la primera instrucción de la rutina *vectori*. Cada una de estas rutinas es un *interrupt handler*.

La CPU conoce la ubicación de esta tabla en la memoria por medio del registro `IDTR`, el cual es seteado por el *kernel* (en `main()`) luego de inicializar el arreglo `idt` (mediante `tvinit()`, ver `main.c`).

- La variable (por cpu) `proc`: Puntero al descriptor del proceso corriente en la tabla de procesos (ver `proc.h`). Puede ser 0 (*null*) porque puede ocurrir una interrupción cuando no se estaba ejecutando código de un proceso de usuario. En este caso la CPU estará ejecutando el ciclo externo en la función `scheduler()` (definida en `proc.c`).

  Es decir que varias partes del código del kernel pueden ejecutarse en el contexto de un proceso (`proc != 0`) o no (*scheduler context*).

Un *proceso*, es decir un programa cargado en memoria en ejecución, está representado en la memoria como se muestra en la figura 1.

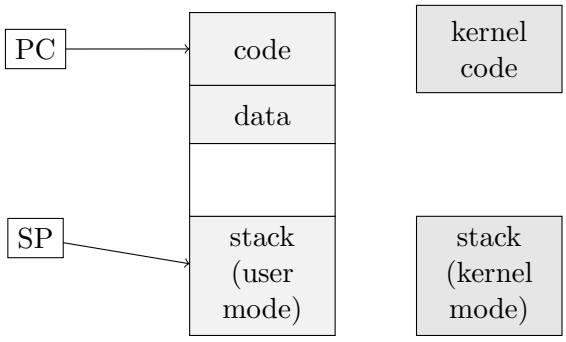

Figura 1: Esquema en memoria de un proceso xv6. CPU en *user mode*.

Suponiendo que el proceso está ejecutando su código, se puede transferir el control a código del kernel por dos posibles motivos:

1. Ocurre una interrupción de algún dispositivo.
2. El proceso ejecuta una llamada al sistema, las cuales están implementadas ejecutando una interrupción por software. `Xv6`, en la plataforma `x86`, utiliza la instrucción `int 64`).

O sea que el *kernel* siempre toma el control con la ocurrencia de interrupciones (por hardware o software).

A continuación se describen los detalles sobre qué sucede al ocurrir una interrupción, ya sea por hardware o software.

2. Interrupciones

Cuando ocurre una interrupción i (por ejemplo, el *timer*), la CPU realiza los siguientes pasos:

1. Como `idt[i].dpl == 0`, entonces $ESP \leftarrow cpu->ts.esp0$

   Esto es, como el *destination privilege level* del *handler* `vectori` es cero (*kernel mode*), la CPU debe cambiar a *ring 0* y al *stack pointer (esp)* lo hace apuntar a la dirección que tiene la estructura `cpu->ts.esp0`.

   Esta estructura (apuntada por el registro `TSR`) brinda el soporte para hacer *cambios de contexto* y tiene información sobre qué *stack* usar en cada modo (`cpu->ts.esp0`, `cpu->ts.esp1` y `cpu->ts.esp2`). Como `xv6` sólo usa los modos 3 y 0, sólo usaremos `cpu->ts.esp0`.

   El `scheduler` de `xv6` anteriormente (en el previo *context switch*, cuando le cedió el control a este proceso), seteó este campo con la dirección base del *kernel mode stack* del proceso corriente (fondo del stack).

2. Deshabilita las interrupciones.

3. La CPU apila (salva) en el `kernel mode stack` del proceso el contenido del *stack pointer* (`ESP`) el registro de flags (`IFLAGS`) y el *program counter* (`EIP`), en ese orden.

4. Carga el *program counter* con la dirección de la primera instrucción de la rutina de interrupción `vectori`.

   Estas rutinas se encuentran definidas en el archivo fuente *assembly* `vectors.S`.

O sea, que ante la ocurrencia de una interrupción, se cambia de modo de ejecución, se salvan los registros y se salta a ejecutar la rutina de interrupción correspondiente.

El proceso pasa a estar ejecutando código del kernel y la CPU ejecuta en el *ring 0* (privilegiado o supervisor), tal como se muestra en la figura 2.

En este punto, no se han salvado todos los registros de la CPU. Hay que salvarlos a todos para luego poder retornar de la interrupción al proceso en el mismo estado que estaba inmediantamente antes de la interrupción.

El resto de los registros se apilan en el *kernel mode stack* del proceso por software. Lo hace el código de `trapasm.S`, al cual se salta desde todas las rutinas de interrupción.

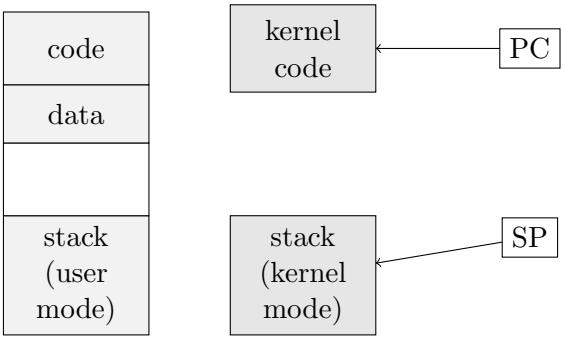

Figura 2: Ante una interrupción el proceso ejecuta código del *kernel*.

El *kernel mode stack* del proceso ahora contiene (desde su base) almacenado el estado del proceso al momento de la interrupción, tal como se muestra en la figura 3.

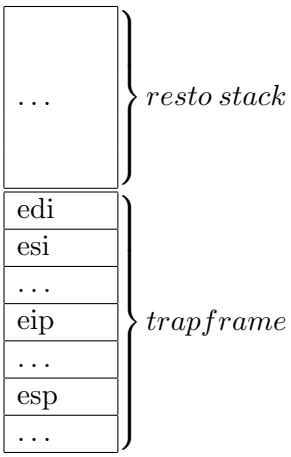

Figura 3: Estado del proceso salvado en el *kernel mode stack*.

A continuación, la rutina `alltraps` (ver `trapasm.S`), invoca a `trap()` (ver `trap.c`) pasándole como parámetro el tope del stack, es decir un puntero a una estructura de datos de tipo `trapframe`, declarada en `x86.h`.

La función `trap()` realiza los siguientes pasos:

1. Determina el tipo de interrupción ocurrió analizando el campo `tf->trapno` (el cual contiene el número de interrupción) que fue apilado por `vector`i.

   Si fue una llamada al sistema, invoca a la función `syscall()`, el cual es un *dispatcher* a la función (*sys_*()*) del kernel correspondiente. Ver

el archivo `syscall.c`.

Sino, determina qué interrupción o excepción (como por ejemplo, división por cero) ocurrió y llama al manejador de la interrupción del dispositivo correspondiente. `xv6` maneja unos pocos dispositivos, como el timer, la controladora de discos IDE, teclado y puerto serie.

Una llamada al sistema, como por ejemplo `sys_read()` puede hacer que el proceso pase a estado `SLEEPING` (ver `sleep()` en `proc.c`). En ese caso el proceso corriente abandonará la CPU y se invocará a `sched()`, la cual saltará a *scheduler()* (ya veremos en detalle cómo), la cual a su vez seleccionará otro proceso para darle el control. Esto es, habrá un *context switch*.

2. Luego de procesar la interrupción correspondiente, se retornará al final del `switch` de `trap()` y ésta chequea si el proceso quedó marcado para finalización (`proc->killed`). En tal caso, se invoca a `exit()` (definida en `proc.c`) para que efectivamente finalice el proceso corriente. Obviamente `exit()` también deberá invocar a `sched()` por lo que también ocurrirá un *context switch*.

   En el caso que la interrupción haya sido por el reloj (*timer*), `trap()` invoca a `yield()` (ver `proc.c`), la cual pasa el proceso de estado `RUNNING` a `RUNNABLE` (o *ready*) e invoca a `sched()`. Es decir que le da una oportunidad al sistema de darle la CPU a otro proceso.

Esto indica que `xv6` le otorga a los procesos ráfagas de uso de cpu[1] de una duración de 1 *tick*.

3. *Context switch*: desde el proceso al scheduler

En la sección anterior vimos que desde *trap()* eventualmente se hará un *context switch* ya sea por una interrupción de reloj (vía `yield()`) o porque el proceso hizo una llamada al sistema bloqueante (vía `sleep()`).

Cualquier función del kernel que requiera hacer un *context switch* invocará a `sched()`.

En ésta sección analizaremos en detalle del mecanismo utilizado para abandonar el *contexto* del proceso corriente y darle a la CPU el estado o contexto de otro proceso.

Esto se hará pasando del contexto del proceso al contexto del scheduler, el cual posteriormente seleccionará otro proceso (eventualmente el mismo)

[1] Esto se denomina *quantum* o *time slice*.

para darle el control de la cpu.

La función `sched()` chequea que se cumplan algunos invariantes en este punto:

1. Que la tabla de procesos esté bloqueada por su correspondiente `lock`.
2. Que el proceso actual esté en estado `RUNNING`.
3. Que las interrupciones estén deshabilitadas.

Luego invoca a `swtch(&proc->context,cpu->scheduler)`.
En este punto, `proc->context` apunta a la estructura `trapframe` en el *kernel mode stack* del proceso corriente.

`swtch(&old,new)` realiza básicamente cuatro pasos:

1. salva (apila) los registros `ebp, ebx, esi, edi`[2] en la pila `old` (en éste punto en el *kernel mode stack* del proceso corriente).
2. actualiza la dirección del contexto en la pila `old` (en éste caso actualiza el puntero `proc->context` para que apunte al tope del *kernel mode stack*).
3. cambia a la pila `new`, que contiene el nuevo contexto (en este caso `cpu-> sheduler`), el cual es el contexto del *scheduler*. El *scheduler* está representado por la función *scheduler()* y tiene su propia pila[3] para el caso que no exista un proceso corriente.
4. recupera (pops) el estado de los registros `ebp, ebx, esi, edi` del nuevo contexto (pila) del scheduler.

El último paso hace que en la pila del scheduler se encuentre la dirección de retorno al final de la invocación a `swtch(&cpu->scheduler, proc->context)` en la función `scheduler()`. Por lo tanto, la ejecución de la instrucción `ret` en `swtch()`, retornará a la línea siguiente de la llamada a `swtch(&cpu->scheduler, proc->context)` en la función `scheduler()`.

Este punto de programa es inmediatemente posterior al punto donde *scheduler()* había realizado anteriormente el cambio de contexto al proceso.

A partir de ahora, en `scheduler()` no hay un proceso corriente (en esta cpu) por lo que debe cambiar al espacio de memoria del kernel (`switchkvm()`,

[2]Según la convención de invocaciones de `x86`, la rutina invocada (*callee*) debe salvar esos registros.

[3]En realidad hay una pila por cpu, ya que cada cpu inicia ejecutando `scheduler()`.

definida en `vm.c`), y pone `proc` en `null`. Luego continuará en el ciclo interior para tratar de encontrar otro proceso `RUNNING` para darle el control de la CPU.

En la figura 4 se muestra el caso en que el código del kernel (en éste caso) el código de `scheduler()` se encuentra ejecutando en contexto de *scheduler*.

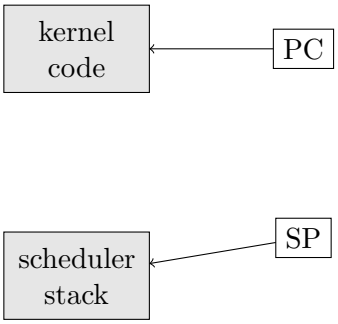

Figura 4: Código del kernel corriendo en *scheduler context*.

Cabe aclarar que en éste punto el kernel puede ser interrumpido ya que el ciclo de `scheduler()` habilita las interrupciones (`sti()`). Así, al ocurrir una interrupción en el contexto del scheduler, `trap()` se ejecutará en ese contexto (con `proc == 0`)[4].

El paso inverso, es decir desde el `scheduler()` a un proceso se describe en la sección anterior.

4. *Context switch*: desde el `scheduler()` a un proceso

En la sección anterior vimos cómo realizamos la mitad de un *context switch*, es decir, desde el proceso en ejecución hasta `scheduler()`.

En este punto, debemos asumir que la CPU se encuentra ejecutando el ciclo de `scheduler()`. Supongamos que en el ciclo interior se encuentra un proceso en estado `RUNNABLE`.

Este proceso seleccionado (`p`) previamente ha realizado el camino que describimos en la sección anterior.

Como ya vimos, el estado de la CPU (contexto) de este proceso quedó en su *kernel mode stack*) y se cambió al contexto (pila) del `scheduler`.

En este punto, en alguna próxima iteración, la función `scheduler()` al encontrar un proceso (p) `RUNNABLE`, realiza los siguientes pasos:

[4]De hecho, esta situación es el más frecuente en el tiempo.

1. Asigna `p` a `proc`.

2. Configura la CPU para que use el mapa de memoria del proceso (`switchuvm()`). Esta función (ver `vm.c`) también reconfigura `cpu->ts.esp0` para que apunte a la base del *kernel mode stack*[5].

3. Invoca a `swtch(&cpu->scheduler, proc->context)`. Como ya vimos, ésta función salva el estado actual de la CPU en la pila del `scheduler` y carga la CPU con el estado que se encuentra en el tope del *kernel mode stack*) del proceso.

   Este estado contiene el `EIP` con la dirección de la instrucción `ret` de la función `swtch()`.

   En este punto estamos nuevamente en el contexto de un proceso (el apuntado por `proc`).

   La ejecución de `ret` en `switch()` causará que se retorne a `sched()` y luego, por ejemplo a (yield()).

Es responsabilidad de todas las funciones que invocan a `sched()`, invocarla con el *lock* a `ptable` adquirido y posteriormente liberarlo (ver `yield()` y `sleep()`.

Suponiendo que desde `sched()` se retornó a `yield()`, ahora se comienza a transitar el camino inverso (retornos) desde la ocurrencia de la interrupción de reloj, retornando a `trap()` y luego al `interrupt handler`, el cual desapilará los valores del estado de la CPU del *kernel mode stack*) del proceso y restaurándolos en los registros correspondientes de la CPU.

De esa manera el proceso retorna de la interrupción y continúa con su ejecución.

Este proceso es análogo si proceso hubiese realizado una llamada al sistema y se hubiera dormido (bloqueado) por `sleep()` (para esperar la finalización de una operación de entrada/salida, por ejemplo) o hubiese ocurrido una interrupción de un dispositivo, lo cual generalmente causa que algún proceso se despierte (desbloquee) mediante `wakeup()`.

5. El primer proceso (`init`)

El mecanismo descripto asume que ya existe al menos un proceso que ha sufrido una interrupción.

[5]Para que cuando el proceso esté ejecutándose y ocurra una interrupción, pase a ejecutar en modo kernel usando esa pila.

Inicialmente no hay ningún proceso en el sistema, por lo que `main()` crea el proceso inicial (ver (userinit())) y le crea en el *kernel mode stack*) un *estado de interrupción.*

Es decir que se configura la pila de tal forma que *simula* que el proceso comenzó su ejecución pero antes de ejecutar su primer instrucción fue interrumpido.

Esta configuración del primer proceso del sistema hace que el mecanismo descripto arriba funcione perfectamente por lo que `main()` simplemente tenga que invocar a `scheduler()` (vía `mpmain()`) para cada CPU.

6. Conclusiones

El análisis de la implementación de *context switch* nos permite concluir que la función `swtch(old,new)`, implementa una transferencia de control entre sus invocaciones entre `sched()` y `scheduler()`.

Esto hace que en realidad `sched()` y `scheduler()` se comporten como *corrutinas* que se transfieren el control mutamente, tal como se muestra en la figura 5.

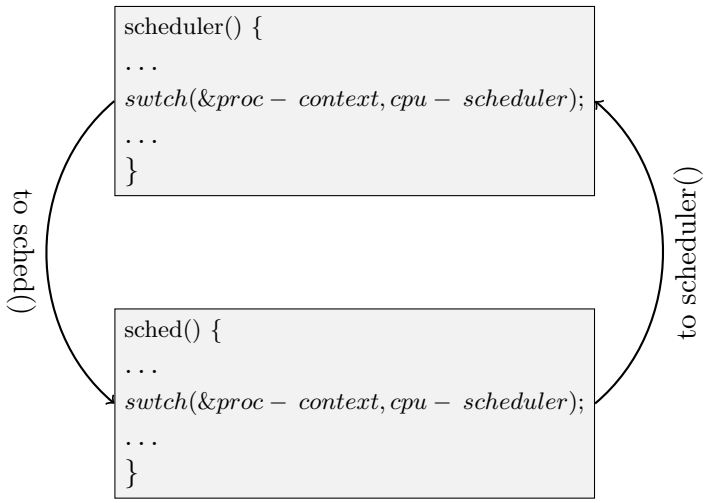

Figura 5: Las funciones `sched()` y `schedule()` como corrutinas.

Referencias

[1] Russ Cox, Frans Kaashoek, Robert Morris. *xv6: a simple, Unix like teaching operating system.* 2011.

https://pdos.csail.mit.edu/6.828/2012/xv6/xv6-rev7.pdf