

# TP2: Scheduling y cambio de contexto

## Introducción

---

**AVISO:** antes de comenzar, verificar que se tiene instalado el software necesario.


En este trabajo se implementarán el mecanismo de cambio de contexto para procesos y el *scheduler* (i.e. planificador) sobre un sistema operativo preexistente. El kernel a utilizar será una modificación de JOS, un exokernel educativo con licencia libre del grupo de [Sistemas Operativos Distribuidos](#) del MIT.

JOS está diseñado para correr en la arquitectura Intel x86, y para poder ejecutarlo utilizaremos QEMU que emula dicha arquitectura.

## Implementación

---

La implementación del TP se dividirá en tres partes.

1. Implementación del cambio de contexto 
2. Implementación de un scheduler *round robin*.
3. Implementación de un scheduler con prioridades.

### Parte 1: Cambio de contexto [↗](#)

---

JOS mantiene un arreglo en memoria como PCB (*Process Control Block*), aunque llama *environment* a los procesos. De aquí en más se usarán las palabras *proceso* y *environment* como sinónimos siempre que hablemos en el contexto de JOS.

Las funciones que se encargan de alocar espacio para un proceso nuevo, crear su espacio de direcciones virtuales y cargar el código en memoria ya se encuentran implementadas, como se puede ver en el archivo `kern/env.c`.

Entre tales funciones se encuentran:

- `env_alloc` : que reserva el espacio en el PCB para un proceso nuevo, y le inicializa algunos parámetros
- `env_setup_vm` : que inicializa el espacio de direcciones virtuales (i.e. el *page directory*) del proceso
- `load_icode` : que carga el código del proceso a partir del binario compilado
- `env_destroy` y `env_free` : para eliminar a un proceso una vez que termina

Al estar implementadas, no las modificaremos, pero es importante entender dónde y cómo son llamadas para comprender el flujo de vida de un proceso en JOS.

La definición de un *environment* puede encontrarse en `inc/env.h` y contiene, entre otras cosas, los campos necesarios para realizar el *cambio de contexto*. A continuación algunos de los campos del mismo struct.

```
struct Env {
    struct Trapframe env_tf;           // Saved registers
    struct Env *env_link;              // Next free Env
    env_id_t env_id;                   // Unique environment identifier
    env_id_t env_parent_id;            // env_id of this env's parent
    enum EnvType env_type;              // Indicates special system environments
    unsigned env_status;               // Status of the environment
    uint32_t env_runs;                 // Number of times environment has run
    int env_cpunum;                    // The CPU that the env is running on
    pde_t *env_pgdir;                 // Kernel virtual address of page dir
    [...]
}
```

Los más importantes del *struct* son: `env_id`, que identifica al environment; `env_pgdir`, que contiene su *page directory* (i.e. su espacio de direcciones virtuales a través de la tabla de paginación inicial) y `env_tf`, que mantiene el *estado de todos los registros* para ese environment.

## De modo *kernel* a modo *usuario*

A partir de esa información, el kernel podrá ejecutar cualquier proceso. La función que se encarga de tomar un proceso y ejecutarlo es `env_run`, en `kern/env.c`. Como parámetro, esta función acepta un `struct Env *` y deberá realizar lo siguiente:

1. Actualizar la variable global `curenv` del kernel, con el nuevo proceso a ser ejecutado
2. Modificar el estado del environment `env_status` para indicar que está siendo ejecutado. La lista de estados puede verse en un `enum` dentro de `inc/env.h`.
3. Realizar el *cambio de contexto*.
  1. Cargar la tabla de paginación del environment con `env_load_pgdir` (función ya implementada)
  2. Llamar a la función `context_switch` para restaurar el estado de CPU

Será la función `context_switch` la que restaure completamente el estado del environment a correr, y que realice el cambio de contexto a *modo usuario*. Es decir, esta función **no hace return jamás**, y como resultado de la misma la CPU pasará a ejecutar código de usuario en `ring 3`.

Para ello, se utilizará la ayuda del hardware, mediante la instrucción `iret` ("*interrupt return*"). Dicha instrucción permite modificar conjuntamente los registros `cs`, `eip` y `esp` de forma atómica, tomando valores desde el stack. El formato que requiere del stack para ser invocada es específico a la arquitectura x86.

Cabe notar que el resto de los registros definidos en `struct Trapframe` deben ser restaurados previamente, dado que `iret` no los modifica.

## Tarea

- Implementar la función `context_switch` en `kern/switch.S`.
  - La función está en assembler, para la arquitectura x86
  - Utilizar la instrucción `iret` para finalizar el cambio de contexto
- Completar la función `env_run`, en `kern/env.c`
- Modificar `kern/init.c` de forma *temporal*, para ejecutar un único proceso `user_hello`
- Utilizar GDB para visualizar el cambio de contexto. Realizar una captura donde se muestre claramente:
  - el cambio de contexto
  - el estado del stack al inicio de la llamada de `context_switch`
  - cómo cambia el stack instrucción a instrucción
  - cómo se modifican los registros *luego* de ejecutar `iret`

## De modo *usuario* a modo *kernel*

El *cambio de contexto* descrito e implementado en la tarea anterior nos permite realizar el pasaje de modo kernel a modo usuario (es decir, de `ring 0` a `ring 3`). Sin embargo, dicho mecanismo no puede utilizarse para volver a modo kernel, dado que requiere de la instrucción privilegiada `iret`.

Para volver al modo kernel, se utilizan *interrupciones*. Las interrupciones son eventos generados por *hardware* que interrumpen al CPU en su ciclo de instrucciones y trasladan la ejecución de una forma controlada a otro contexto, permitiendo cambiar registros importantes (`eip`, `cs`, `esp`, etc.) a valores fijos definidos previamente.

El kernel configura las interrupciones en `kern/trap.c`, mediante la función `trap_init`. Ahí se genera la tabla de interrupciones (la IDT) con referencias a los *handlers* de cada tipo de interrupción.

Un tipo de interrupción común es la `syscall`. Todas las syscalls pasarán por esta única interrupción, y desembocarán en la función `syscall` del lado del kernel que se encargará de determinar qué *syscall* se necesita ejecutar y llamar a la función `sys_*` adecuada. En `kern/syscall.c`.

Así, un *handler* para la interrupción de las *syscalls* se define de la siguiente forma:

```
SETGATE(idt[T_SYSCALL], 0, GD_KT, &trap48, 3);
```

Los detalles de la macro `SETGATE` no son importantes, pero mediante los parámetros se está indicando al CPU que siempre que se genere la *interrupción número 48* (la que corresponde a las

syscalls, dado que `T_SYSCALL=48`), esperamos que se llame a la función `trap48`, que se corresponde al *handler* de la interrupción de ese número.

Mediante el resto de los parámetros, específicamente `GD_KT` (*Global Descriptor, Kernel Text*) se está indicando al CPU que siempre que se llame a ese *handler*, se deberá hacerlo en el `ring 0`. La función `trap48` está definida, de forma auto-generada vía macros, en `kern/trapentry.S`.

Como es el kernel quien define la tabla de interrupciones, y se coloca a si mismo como punto de entrada luego de cualquier interrupción, dicha entrada al kernel está controlada y el paso de `ring 3` a `ring 0` es seguro.

Observando `kern/trapentry.S` todos los *handlers* están generados usando macros, y todos desembocan en la función `_alltraps`, que está incompleta y deberán implementar.

### Tarea

- Implementar la función `_alltraps` en `kern/trapentry.S`.
  - La función está en assembler, para la arquitectura x86
  - Al momento de invocarse la función, el stack está *en el mismo estado* en el que lo dejamos al llamar a `iret`, con la excepción de los valores pusheados por las macros `TRAPHANDLER_NOEC` y `TRAPHANDLER`.
  - La función debe dejar un `struct Trapframe` en el stack, completando los registros faltantes, y terminar con una llamada a la función `trap`.
- Modificar `kern/init.c` de forma *temporal*, para ejecutar un único proceso `user_hello`
- Ejecutar el `kernel` con `qemu` y validar que las syscalls están funcionando.

Con ambas tareas implementadas, la ejecución de cualquier proceso debería poder llegar a su fin. Sin embargo, solo podemos ejecutar un proceso a la vez dado que no hay *scheduler* implementado.

## Parte 2: Scheduler *round robin* [↗](#)

Para poder hacer uso completo del arreglo `envs` (i.e. el PCB), y ejecutar más de un proceso a la vez; hace falta la implementación de un *scheduler*.

El esqueleto tiene preparado ya todo lo necesario para el mismo, en `kern/sched.c`. La función `sched_yield` es la que se invoca cada vez que se necesita ejecutar un nuevo proceso, y es aquí donde la política de scheduling deberá ser implementada.

Notar que `sched_yield` tiene dos posibles salidas: se elige y ejecuta un proceso llamando a `env_run`, o bien *no hay más procesos que ejecutar* y se desemboca en `sched_halt`, donde efectivamente el kernel queda en estado *idle*.

## Tarea

- Implementar la función `sched_yield` en `kern/sched.c`
  - La política de scheduling debe ser `round_robin`
- Ejecutar las pruebas básicas con `make grade` y validar que pasan todas

## Parte 3: Scheduler con prioridades [↗](#)

La política de scheduling *round robin* es la más sencilla y simple de implementar; y aunque es justa (le da a todos los procesos la misma proporción del CPU) puede no ser suficiente para situaciones más reales. Usualmente los procesos son distintos entre sí en cuanto a importancia y carga para el sistema.

En esta parte, se mejorará el scheduler implementado anteriormente para agregarle un esquema de **prioridades**. Esto requerirá a su vez la adición de *syscalls* que permitan manipularlas, así como de procesos de usuario para validar el correcto funcionamiento.

## Tarea

- Agregar a JOS un scheduler basado en prioridades. Los **requisitos** son:
  - La política de scheduling debe ser *round robin* o *por prioridades* y la misma debe elegirse al llamar a `sched_yield` en tiempo de compilación (e.g. usar `#ifdef`).
  - Todo proceso debe tener asociada una prioridad, asignada al momento de su creación. Esto requiere cambios en `env_create` y/o `env_alloc`.
  - Se debe incluir una *syscall* para obtener prioridades, y otra para modificar prioridades. Ambas *syscalls* deben ser *seguras*. Esto quiere decir que, no se debe permitir a un proceso *aumentar* su prioridad pero si reducirla.
  - Se debe incluir soporte para prioridades en las *syscalls* relevantes. Por ejemplo, cuando un proceso llama a `fork`, se deberá configurar acordemente (y siguiendo algún criterio) las prioridades del proceso hijo.
- Incorporar, dentro del *scheduler*, estadísticas sobre las decisiones de scheduling. Algunas ideas/recomendaciones son:
  - Historial de procesos ejecutados/seleccionados
  - Número de llamadas al scheduler
  - Número de ejecuciones por cada proceso
  - Inicio y fin de cada proceso ejecutado
- Las estadísticas deben ser mostradas por el kernel al finalizar la ejecución de todos los procesos, durante `sched_halt`.
- Modificar `kern/init.c`, y crear procesos de usuario para mostrar el correcto funcionamiento del scheduler con prioridades. Incluir ejemplos que muestren si un

proceso puede ganar/perder prioridad.

## Esqueleto y compilación

**AVISO:** El esqueleto se encuentra disponible en **fisop/sched**.

**IMPORTANTE:** leer el archivo `README.md` que se encuentra en la raíz del proyecto. Contiene información sobre cómo realizar la compilación de los archivos, y cómo ejecutar el formateo de código.

## Compilación [↗](#)

La compilación se realiza mediante `make`. En el directorio `obj/kern` se puede encontrar:

- `kernel` — el binario ELF con el kernel
- `kernel.asm` — assembler asociado al binario

## Ejecución [↗](#)

Para correr JOS, se puede usar `make qemu` o `make qemu-nox`.

Para ejecutar *un proceso de usuario* en particular dentro del kernel, se puede usar `make run-<proceso>` o `make run-<proceso>-nox`. Como ejemplo, `make run-hello-nox` correrá el proceso de usuario `user/hello.c`.

## Depurado [↗](#)

El *Makefile* de JOS incluye dos reglas para correr QEMU junto con GDB.

En una terminal ejecutar:

```
$ make qemu-gdb
***
*** Now run 'make gdb'.
***
qemu-system-i386 ...
```

y en otra distinta:

```
$ make gdb
gdb -q -ex 'target remote ...' -n -x .gdbinit
Reading symbols from obj/kern/kernel...done.
Remote debugging using 127.0.0.1:...
```

```
0x0000fff0 in ?? ()  
(gdb)
```

## Depurado de una triple fault

En la arquitectura x86, el sistema se reinicia automáticamente cuando ocurre una “triple falla” (*triple fault*). QEMU, por omisión, obedece esta especificación.

Sin embargo, durante el desarrollo de sistemas operativos en modo protegido de x86, las *triple fault* ocurren casi exclusivamente por un bug en el kernel. Por esto, es más deseable que QEMU detenga la ejecución en lugar de reiniciarse constantemente.

QEMU no ofrece soporte *directo* para detectar fallas triples y detener la ejecución, pero existen un set de opciones que se acercan bastante a ese propósito.

Por tanto, si en una determinada versión del desarrollo ocurre que QEMU se reinicia constantemente, se recomienda probar lo siguiente:

- correr QEMU con las opciones: `-no-reboot -no-shutdown -d cpu_reset` (estas opciones pueden añadirse en la variable `QEMUOPTS` en el archivo `GNUmakefile`)
- si el error realmente fue un *Triple fault*, se mostrará ese error en la última línea del archivo `qemu.log`, y se podrá consultar el estado de los registros mediante el monitor de QEMU (`Ctrl-A C → info registers`)

## Bibliografía útil

A continuación se presentan algunos enlaces y bibliografía útiles como referencia.

- OSTEP, capítulo 7: [Scheduling: Introduction](#) (PDF)
- OSTEP, capítulo 8: [Scheduling: The Multi-Level Feedback Queue](#) (PDF)
- OSTEP, capítulo 9: [Scheduling: Proportional Share](#) (PDF)
- Manuales de Intel: [Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, Part 1](#) (PDF)