Trabajos Prácticos con JOS

Federico del Mazo - 100029
Rodrigo Souto - 97649

Trabajos Prácticos con JOS

Respuestas teóricas de los distintos trabajos prácticos/labs de Sistemas Operativos (75.08).

TP1: Memoria virtual en JOS (26/04/2019)

Memoria física: boot_alloc_pos

1. Incluir: Un cálculo manual de la primera dirección de memoria que devolverá `boot_alloc()` tras el arranque. Se puede calcular a partir del binario compilado (obj/kern/kernel), usando los comandos `readelf` y/o `nm` y operaciones matemáticas.

Truncando la salida de ambos comandos (con `grep`), vemos las siguientes lineas:

```
sisop_2019a_delmazo_souto TP1 % readelf -s obj/kern/kernel | grep end
112: f0117950     0 NOTYPE  GLOBAL DEFAULT    6 end
sisop_2019a_delmazo_souto TP1 % nm obj/kern/kernel | grep end
f0117950 B end
```

Como podemos ver en ambos casos, la dirección de memoria que recibe `boot_alloc()` es `f0117950` (en decimal, `4027677008`). A este valor, la función (en su primera llamada) lo redondea a 4096 (`PGSIZE`) llamando a `ROUNDUP(a,n)`. Además de devolver ese valor redondeado, guarda la variable `nextfree` en una página más de lo recibido.

Por ende, el valor devuelto será el de `ROUNDUP(4027677008, 4096)`. Esta función, como indica su documentación, redondea a `a` al múltiplo más cercano de `n`. Este múltiplo será **4027678720**, que esta más cerca que el siguiente múltiplo (4027682816). Para confirmarlo desde la práctica, se traducen a Python los cálculos que utiliza la función de redondeo:

```
sisop_2019a_delmazo_souto TP1 % python3
Python 3.6.7 |Anaconda, Inc.| (default, Oct 23 2018, 19:16:44)
>>> a = 0xf0117950
>>> n = 4096
>>> def rounddown(a,n): return a - a % n
>>> def roundup(a,n): return rounddown(a + n - 1, n)
>>> res = roundup(int(a),n)
>>> res
4027678720
>>> hex(res)
'0xf0118000'
```

2. Incluir: Una sesión de GDB en la que, poniendo un breakpoint en la función `boot_alloc()`, se muestre el valor de `end` y `nextfree` al comienzo y fin de esa primera llamada a `boot_alloc()`.

```
sisop_2019a_delmazo_souto TP1 % make gdb
gdb -q -s obj/kern/kernel -ex 'target remote 127.0.0.1:26000' -n -x .gdbinit
Reading symbols from obj/kern/kernel...done.
Remote debugging using 127.0.0.1:26000
0x0000fff0 in ?? ()
```

```
(gdb) break boot_alloc
Breakpoint 1 at 0xf0100a58: file kern/pmap.c, line 89.
(gdb) continue
Continuing.
The target architecture is assumed to be i386
=> 0xf0100a58 <boot_alloc>: push   %ebp

Breakpoint 1, boot_alloc (n=4096) at kern/pmap.c:89
89  {
(gdb) print (char*) &end
$1 = 0xf0117950 ""
(gdb) watch &end
Watchpoint 2: &end
(gdb) watch nextfree
Hardware watchpoint 3: nextfree
(gdb) continue
Continuing.
=> 0xf0100aac <boot_alloc+84>:  jmp    0xf0100a68 <boot_alloc+16>

Hardware watchpoint 3: nextfree

Old value = 0x0
New value = 0xf0118000 ""
0xf0100aac in boot_alloc (n=4096) at kern/pmap.c:100
100         nextfree = ROUNDUP((char *) end, PGSIZE);
(gdb) continue
Continuing.
=> 0xf0100a81 <boot_alloc+41>:  mov    0xf0117944,%edx

Hardware watchpoint 3: nextfree

Old value = 0xf0118000 ""
New value = 0xf0119000 ""
boot_alloc (n=4096) at kern/pmap.c:111
111     if (nextfree >= (char *) (KERNBASE + npages * PGSIZE)) {
(gdb) continue
Continuing.
```

Como se puede ver, se cumple todo lo planteado. `end` comienza en `0xf0117950`, luego `nextfree` se inicializa en el número ya redondeado `0xf0118000`, y finalmente se avanza una página, y queda `0xf0119000`.

Memoria física: page_alloc

1. Responder: ¿en qué se diferencia `page2pa()` de `page2kva()`?

Como bien indican sus nombres, `page2pa()` y `page2kva()` se diferencian en el valor de retorno. Ambas reciben una página física, pero `page2pa()` devuelve su dirección física (de tipo `physaddr_t`) mientrás que `page2kva()` devuelve la dirección virtual (kernel virtual address), de tipo `void*`.

Incluso, `page2kva()` no es más que un llamado a `page2pa()` y luego a la función del preprocesador `KADDR()` que recibe una dirección física y devuelve la respectiva dirección virtual.

Large pages: map_region_large

1. Responder: ¿cuánta memoria se ahorró de este modo? ¿Es una cantidad fija, o depende de la memoria física de la computadora?

Se ahorran 4KB, que es el tamaño de un página, ya que se deja de usar `entry_pgtable` y se mapea la misma cantidad de memoria consecutiva (4MB) directamente con una *large page*.

Debido a que JOS se compila con la arquitectura de 32 bits i386, independientemente de cuál sea la memoria física disponible de la máquina, las páginas tendrán un tamaño de 4KB, y cómo lo que sea ahorra es crear a `entry_pgtable`, que tiene el tamaño de una página, se ahorra esa cantidad de bytes.

TP2: Procesos de usuario (17/5/2019)

Inicializaciones: env_alloc

1. Responder: ¿Qué identificadores se asignan a los primeros 5 procesos creados? (Usar base hexadecimal.)

La generación de ids de entornos se logra con la siguiente porción de código:

```
generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
e->env_id = generation | (e - envs);
```

Analíticamente, sabiendo que `ENVGENSHIFT` equivale a 12, y `NENV` es `1 << 10` (1024), se puede ver que `generation` equivale al id anterior más 4096 y a eso aplicar el AND de bits con ~(1023). Luego, en la segunda linea, se le suma a este valor el número de entorno. Con un id de entorno igual a 0, `generation` será 4096 (`0x1000` en hexadecimal).

Efectivamente, gracias a GDB se puede comprobar que el valor de `generation` en la primera ejecución es el `0x1000` (esto es así ya que en la primera corrida los id de los entornos son 0; una vez que se empiecen a reciclar los entornos el valor de generation dependerá del id del entorno previo). Por ende, luego de la suma del offset, los primeros 5 entornos serán: `0x1000`, `0x1001`, `0x1002`, `0x1003` y `0x1004`.

2. Responder: Supongamos que al arrancar el kernel se lanzan `NENV` procesos a ejecución. A continuación se destruye el proceso asociado a `envs[630]` y se lanza un proceso que cada segundo muere y se vuelve a lanzar. ¿Qué identificadores tendrá este proceso en sus sus primeras cinco ejecuciones?

El primer proceso lanzado no será mas que la suma entre el primer `generation` (`0x1000`) y el offset 630 (en hexadecimal, `0x0276`). Es decir, el `0x1276`.

Luego, una vez que este proceso muera y se relance, se utilizara este id como base para el nuevo `generation`. El `generation` nuevo será entonces la suma entre `0x1276` (el previo id), la constante `0x1000`, y a eso el AND con `~(0x03FF)`. Luego, a ese número se le suma nuevamente el `0x0276`.

Para los primeros 5 procesos queda:

```
>>> def generate_id(id_prev): return hex( (id_prev + 0x1000 & ~(0x03FF)) + 0x0276 )
>>> generate_id(0x0)
'0x1276'
>>> generate_id(0x1276)
'0x2276'
>>> generate_id(0x2276)
'0x3276'
>>> generate_id(0x3276)
'0x4276'
>>> generate_id(0x4276)
'0x5276'
```

Inicializaciones: env_init_percpu

1. Responder: ¿Cuántos bytes escribe la función `lgdt`, y dónde?

`lgdt` escribe 6 bytes en la Global Descriptor Table.

2. Responder: ¿Qué representan esos bytes?

Estos bytes son el tamaño del GDT (2 bytes) y la dirección de la tabla (4 bytes).

Lanzar procesos: env_pop_tf

1. Responder: ¿Qué hay en `(%esp)` tras el primer `movl` de la función?

En `%esp` pasa a estar la dirección de memoria a la que apunta `tf` (`%0` => primer argumento de la función), notar que ya no apunta más a una dirección del *stack*.

2. Responder: ¿Qué hay en `(%esp)` justo antes de la instrucción `iret`? ¿Y en `8(%esp)`?

En `%esp` está la dirección de `tf->tf_eip`. En `8(%esp)` está la dirección de `tf->tf_eflags`.

3. Responder: ¿Cómo puede determinar la CPU si hay un cambio de ring (nivel de privilegio)?

La CPU puede comparar los 2 bits menos significativos de `tf_cs` que contienen el nivel de privilegio con el contenido de los últimos 2 bits del registro `%cs` actualmente. Si hay una diferencia, es porque hubo un cambio de *ring*.

Lanzar procesos: gdb_hello

Incluir una sesión de GDB con diversos pasos:

1. Poner un breakpoint en `env_pop_tf()` y continuar la ejecución hasta allí.
2. En QEMU, entrar en modo monitor (`Ctrl-a c`), y mostrar las cinco primeras líneas del comando `info registers`.

```
(qemu) info registers
EAX=003bc000 EBX=f01c0000 ECX=f03bc000 EDX=0000021f
ESI=00010094 EDI=00000000 EBP=f0118fd8 ESP=f0118fbc
EIP=f0102ecd EFL=00000092 [--S-A--] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
```

3. De vuelta a GDB, imprimir el valor del argumento *tf*:

```
(gdb) p tf
$1 = (struct Trapframe *) 0xf01c0000
```

4. Imprimir, con `x/Nx tf` tantos enteros como haya en el struct *Trapframe* donde `N = sizeof(Trapframe) / sizeof(int)`.

```
(gdb) x/17x tf
0xf01c0000:     0x00000000      0x00000000      0x00000000      0x00000000
0xf01c0010:     0x00000000      0x00000000      0x00000000      0x00000000
0xf01c0020:     0x00000023      0x00000023      0x00000000      0x00000000
0xf01c0030:     0x00800020      0x0000001b      0x00000000      0xeebfe000
0xf01c0040:     0x00000023
```

5. Avanzar hasta justo después del `movl ...,%esp`, usando `si M` para ejecutar tantas instrucciones como sea necesario **en un solo paso**:

```
(gdb) disas
Dump of assembler code for function env_pop_tf:
=> 0xf0102ecd <+0>:     push   %ebp
   0xf0102ece <+1>:     mov    %esp,%ebp
   0xf0102ed0 <+3>:     sub    $0xc,%esp
   0xf0102ed3 <+6>:     mov    0x8(%ebp),%esp
   0xf0102ed6 <+9>:     popa
   0xf0102ed7 <+10>:    pop    %es
   0xf0102ed8 <+11>:    pop    %ds
   0xf0102ed9 <+12>:    add    $0x8,%esp
   0xf0102edc <+15>:    iret
   0xf0102edd <+16>:    push   $0xf0105444
   0xf0102ee2 <+21>:    push   $0x1f9
   0xf0102ee7 <+26>:    push   $0xf01053c2
   0xf0102eec <+31>:    call   0xf01000ab <_panic>
```

```
End of assembler dump.

(gdb) si 4
```

6. Comprobar, con `x/Nx $sp` que los contenidos son los mismos que *tf* (donde `N` es el tamaño de *tf*).

```
(gdb) x/17x $sp
0xf01c0000:     0x00000000      0x00000000      0x00000000      0x00000000
0xf01c0010:     0x00000000      0x00000000      0x00000000      0x00000000
0xf01c0020:     0x00000023      0x00000023      0x00000000      0x00000000
0xf01c0030:     0x00800020      0x0000001b      0x00000000      0xeebfe000
0xf01c0040:     0x00000023
```

7. Explicar con el mayor detalle posible cada uno de los valores. Para los valores no nulos, se debe indicar dónde se configuró inicialmente el valor, y qué representa.

Los primeros 8 valores que se muestran corresponden a los valores de los registros en el orden que tienen en `struct PushRegs`.

Los miembros `tf_es` (Extra Segment) y `tf_ds` (Data Segment) tienen el mismo valor: `0x00000023`. El valor corresponde al segmento de memoria.

El valor de `tf_eip`, `0x00800020`, indica a que instrucción tiene que volver el procesador al retomar el programa.

Los últimos dos bits del miembro `tf_cs`, `0x0000001b`, indican el *ring* del proceso, en este caso, al ser `0x3`, indica que está en modo usuario.

El miembro `tf_esp` indica a que posición debe volver el registro `%esp` al retomar el proceso.

Finalmente el miembro `tf_ss` (Stack Segment) contiene el mismo valor que el que tenían `tf_es` y `tf_ds`.

8. Continuar hasta la instrucción `iret`, sin llegar a ejecutarla. Mostrar en este punto, de nuevo, las cinco primeras líneas de `info registers` en el monitor de QEMU. Explicar los cambios producidos.

```
(qemu) info registers
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
ESI=00000000 EDI=00000000 EBP=00000000 ESP=f01c0030
EIP=f0102edc EFL=00000096 [--S-AP-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
```

Los registros tienen los mismos valores que los que están en el TrapFrame, el DPL de los registros `%es` y `%ds` pasó a ser 3, mientras que el de `%cs` sigue siendo 0.

9. Ejecutar la instrucción `iret`. En ese momento se ha realizado el cambio de contexto y los símbolos del kernel ya no son válidos.

- imprimir el valor del contador de programa con `p $pc` o `p $eip`
- cargar los símbolos de *hello* con `symbol-file obj/user/hello`
- volver a imprimir el valor del contador de programa
- Mostrar una última vez la salida de `info registers` en QEMU, y explicar los cambios producidos.

```
(gdb) p $pc
$2 = (void (*)()) 0x800020
(gdb) p $eip
$3 = (void (*)()) 0x800020

(gdb) symbol-file obj/user/hello
¿Cargar una tabla de símbolos nueva desde «obj/user/hello»? (y or n) y
Leyendo símbolos desde obj/user/hello...hecho.
Error in re-setting breakpoint 1: Función «env_pop_tf» no definida.

(gdb) p $eip
$4 = (void (*)()) 0x800020 <_start>
```

```
(qemu) info registers
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
ESI=00000000 EDI=00000000 EBP=00000000 ESP=eebfe000
EIP=00800020 EFL=00000002 [-------] CPL=3 II=0 A20=1 SMM=0 HLT=0
ES =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
CS =001b 00000000 ffffffff 00cffa00 DPL=3 CS32 [-R-]
```

Ahora el DPL del registro `%cs` pasó a ser 3, lo que indica que está en *user mode*.

10. Poner un breakpoint temporal (`tbreak`, se aplica una sola vez) en la función `syscall()` y explicar qué ocurre justo tras ejecutar la instrucción `int $0x30`. Usar, de ser necesario, el monitor de QEMU.

```
(qemu) info registers
EAX=00000000 EBX=00000000 ECX=0000000d EDX=eebfde88
ESI=00000000 EDI=00000000 EBP=eebfde40 ESP=eebfde18
EIP=008009f9 EFL=00000096 [--S-AP-] CPL=3 II=0 A20=1 SMM=0 HLT=0
ES =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
CS =001b 00000000 ffffffff 00cffa00 DPL=3 CS32 [-R-]

// Después del break
(qemu) info registers
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000663
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00000000
EIP=0000e05b EFL=00000002 [-------] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 00000000 0000ffff 00009300
CS =f000 000f0000 0000ffff 00009b00
```

Interrupts y syscalls: kern_idt

1. Responder: ¿Cómo decidir si usar `TRAPHANDLER` o `TRAPHANDLER_NOEC`? ¿Qué pasaría si se usara solamente la primera?

Nos guiamos con la tabla 5-1 de **Intel® 64 and IA-32 Architectures Software Developer's Manual** y con la tabla en la sección 9.10 de **Intel 80386 Reference Programmer's Manual**

Description Number	Interrupt	Error Code
Divide error	0	No
Debug exceptions	1	No
Breakpoint	3	No
Overflow	4	No
Bounds check	5	No
Invalid opcode	6	No
Coprocessor not available	7	No
System error	8	Yes (always 0)
Coprocessor Segment Overrun	9	No
Invalid TSS	10	Yes
Segment not present	11	Yes
Stack exception	12	Yes
General protection fault	13	Yes
Page fault	14	Yes
Coprocessor error	16	No
Two-byte SW interrupt	0-255	No

Si sólo usáramos `TRAPHANDLER`, en algunos casos no se *pushearía* el *error code* (ni un valor *dummy* como hace `TRAPHANDLER_NOEC`) y nuestro trapframe no tendría el formato especificado en `struct TrapFrame`, ya que no sería consistente.

2. Responder: ¿Qué cambia, en la invocación de handlers, el segundo parámetro (`istrap`) de la macro `SETGATE`? ¿Por qué se elegiría un comportamiento u otro durante un syscall?

Lo que hace es indicar si se trata de una *trap gate* (en vez de una *interrupt gate*)

Porque en una *syscall* es necesario que se prevenga que otras interrupciones intervengan con el actual *handler*.

Las *interrupt gates* modifican el valor de `IF` (*interrupt-enable flag*) y luego lo retornan al valor anterior (que se encuentra en `tf_eflags`) al ejecutar `iret`. Las *trap gates* no modifican `IF`.

3. Responder: Leer `user/softint.c` y ejecutarlo con `make run-softint-nox`. ¿Qué excepción se genera? Si es diferente a la que invoca el programa… ¿cuál es el mecanismo por el que ocurrió esto, y por qué motivos?

Se genera la excepción *General Protection*, cuando uno esperaría que ocurra una *Page Fault*, ya que 14 corresponde a esa excepción en `idt`. Esto ocurre porque aún no se programó cómo debe responder el *kernel* en caso de una *Page Fault*.

Protección de memoria: user_evilhello

Se guarda el siguiente programa en `evilesthello.c`:

```
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    char *entry = (char *) 0xf010000c;
    char first = *entry;
    sys_cputs(&first, 1);
}
```

1. Responder: ¿En qué se diferencia el código de la versión en `evilhello.c` con `evilesthello.c`?

Como se puede observar, la diferencia esta sencillamente en que el `evilhello.c` original pasa la dirección de memoria sin modificar, mientrás que el modificado antes de eso cambia el puntero (desreferencia, y luego pasa la referencia).

Específicamente al asignar la variable `first` a `entry` (es decir, la linea `char first = *entry;`), se procede a 'engañar' al sistema operativo (con nada más que un swap) y se logra acceder a una dirección privilegiada. De ser la ejecución satisfactoria, revelaría una gran vulnerabilidad en el sistema: se puede imprimir todo lo que contenga el kernel!

2. Responder: ¿En qué cambia el comportamiento durante la ejecución? ¿Por qué? ¿Cuál es el mecanismo?

Gracias al swap de direcciones, la version modificada (aun más malvada) de `evilhello.c` sí logró imprimir el entry point del kernel como cadena (la versión original no imprimió más que simbolos basura). Esto no debería pasar y se tiene que atrapar de alguna manera (con los assertions del `user_mem_check`), ya que la dirección de memoria a la que se accede es una dirección privilegiada para el usuario y se debe prohibir el acceso a esta.

Ejecución de `evilhello.c` (versión original):

```
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffffc
f r Incoming TRAP frame at 0xeffffffc
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

Ejecución de `evilesthello.c` (versión modificada):

```
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbc
[00001000] user fault va f010000c ip 00800039
TRAP frame at 0xf01c1000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0xf010000c
  err  0x00000005 [user, read, protection]
  eip  0x00800039
  cs   0x----001b
  flag 0x00000082
  esp  0xeebfdfb0
  ss   0x----0023
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

TP3: Multitarea con desalojo (14/06/2019)

Múltiples CPUs: static_assert

1. Responder: ¿cómo y por qué funciona la macro `static_assert` que define JOS?

`static_assert` (presente en la biblioteca `assert.h`) es un macro de C introducido en 2011 por el standard C11. Lo único que hace el macro es expandir la keyword `_Static_assert`, que evalua en tiempo de compilación frente al 0. De ser la expresión evaluada igual a 0, es falsa, y se lanza un error de compilación. De ser distinto a 0, es verdadera, y todo sigue su rumbo normalmente.

Lo que hace JOS es incluir su propia versión modificada de `assert.h`, redefiniendo el macro `static_assert`. En vez de expandir a `_Static_assert`, JOS provee su propia implementación:

```
#define static_assert(x)    switch (x) case 0: case (x):
```

Esta implementación logra la misma funcionalidad, partiendo de la idea de que un switch no puede tener definido dos veces el mismo caso (se generaría `error: duplicate case value`). Entonces, de ser x igual a cero, se estaría teniendo un switch con dos veces el mismo caso (el cero) definido, resultando en un error de compilación.

Planificador y múltiples procesos: env_return

1. Responder: al terminar un proceso su función `umain()` ¿dónde retoma la ejecución el kernel? Describir la secuencia de llamadas desde que termina `umain()` hasta que el kernel dispone del proceso.

Analizando el ELF de un proceso (en este caso `/user/hello`) con el comando `readelf` y luego leyendo sus instrucciones en assembly, se puede ver (viendo la llamada que hace `libmain` pasada la llamada a `umain`) que la secuencia de instrucciones luego de llamar a `umain` consiste en llamar a `exit` (includo en `/lib/exit.c`). Siendo que `exit` la provee JOS, se puede seguir la traza en el código de JOS (y no en assembly).

Lo que hace JOS en `exit` es llamar a `sys_env_destroy` (la syscall que provee para destruir procesos, en `/lib/syscall.c`), que no es más que un wrapper a `syscall(SYS_env_destroy, 1, envid, 0, 0, 0);`. Esto llamará a `env_destroy`, quien llama a `sched_yield` y se finaliza en `sched_halt`.

2. Responder: ¿en qué cambia la función `env_destroy()` en este TP, respecto al TP anterior?

El TP previo contenía el siguiente `env_destroy()`:

```
void
env_destroy(struct Env *e)
{
    env_free(e);

    cprintf("Destroyed the only environment - nothing more to do!\n");
    while (1)
        monitor(NULL);
}
```

La diferencia con el TP actual reside en que ahora tenemos soporte para múltiples CPUs y un scheduler de procesos. Entonces, hay que preguntarse si el entorno a destruir es el que esta corriendo en esta CPU o si esta corriendo en otras. Si esta corriendo en la CPU actual, entonces luego de ser liberado habrá que llamar al scheduler para conseguir el siguiente entorno a ejecutar. De estar corriendo en otra CPU, no se llama a `env_free`, si no que solamente se lo marca como un proceso zombie (`ENV_DYING`), para que la próxima vez que aparezca en el kernel sea liberado.

Planificador y múltiples procesos: sys_yield

1. Leer y estudiar el código del programa `user/yield.c`. Cambiar la función `i386_init()` para lanzar tres instancias de dicho programa, y mostrar y explicar la salida de `make qemu-nox`.

El código de `yield.c` es:

```
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    int i;

    cprintf("Hello, I am environment %08x.\n", thisenv->env_id);
    for (i = 0; i < 5; i++) {
        sys_yield();
        cprintf("Back in environment %08x, iteration %d.\n",
            thisenv->env_id, i);
    }
    cprintf("All done in environment %08x.\n", thisenv->env_id);
}
```

Este código entra a un ciclo de 5 iteraciones donde llama a `sys_yield` (que es solamente una llamada al scheduler Round Robin, en `sched_yield`).

La modificación a `i386_init` es:

```
ENV_CREATE(user_yield, ENV_TYPE_USER);
ENV_CREATE(user_yield, ENV_TYPE_USER);
ENV_CREATE(user_yield, ENV_TYPE_USER);
```

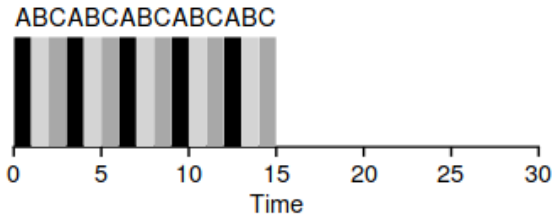
La salida de `make qemu-nox` es:

```
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
```

```
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
All done in environment 00001001.
[00001001] exiting gracefully
[00001001] free env 00001001
Back in environment 00001002, iteration 4.
All done in environment 00001002.
[00001002] exiting gracefully
[00001002] free env 00001002
```

Esta salida es un perfecto test para el scheduler. Como se puede ver, los procesos (que se van desalojando a sí mismo) le entregan el poder al scheduler, y al ser un Round Robin entre tres procesos iguales, la distribución de tiempo es enteramente justa (fair) y circular. Se corre el proceso 0, en su primera iteración, luego el proceso 1, en su primera iteración, luego el tercer proceso en su primera iteracion y así hasta completar las 5 iteraciones de los 3 procesos.

La secuencia de instrucciones es, básicamente, la siguiente imagen:



Round Robin, Operating Systems: Three Easy Pieces, Chapter 7, Arpaci-Dusseau

Creación dinámica de procesos: envid2env

1. Responder qué ocurre en JOS, si un proceso llama a `sys_env_destroy(0)`

El comentario (`// If envid is zero, return the current environment.`) dentro de la definición de `envid2env` es muy claro: Si se recibe 0, se entiende como el entorno actual. Como `sys_env_destroy` llama con ese parametro a `envid2env`, entonces con esta llamada se destruye el entorno actual.

2. Responder qué ocurre en Linux, si un proceso llama a `kill(0, 9)`

El manual de la syscall kill (con `man 2 kill`) explica que el primer parametro recibido es el `pid` y el segundo la señal a enviar. También dice: `If pid equals 0, then sig is sent to every process in the process group of the calling process.`. También (con `man 7 signal`) se nota que la señal 9 es `SIGKILL`, una señal para forzar la terminación de un proceso.

`kill(0,9)` destruye todos los procesos del process group (del proceso que llamo a `kill`).

3. Responder qué ocurre en JOS, si un proceso llama a `sys_env_destroy(-1)`

`envid2env` sabe cual es el entorno porque hace `e = &envs[ENVX(envid)];` (siendo `ENVX` la función (macro) que devuelve el offset del entorno en el arreglo `envs`, `#define ENVX(envid) ((envid) & (NENV - 1))`). Es con la definición de `ENVX` que se nota que si se recibe -1, se devolvera el offset `NENV-1`, que equivale al último elemento del arreglo. La llamada `sys_env_destroy(-1)` destruye el entorno en la última posicion de `envs`.

4. Responder qué ocurre en Linux, si un proceso llama a `kill(-1, 9)`

El manual de la syscall también dice `If pid equals -1, then sig is sent to every process for which the calling process has permission to send signals, except for process 1 (init)`. Entonces, `kill(-1,9)` destruye todos los procesos a los que el proceso que llamo a `kill` puede alcanzar (por sus permisos).

Ejecución en paralelo: multicore_init

1. ¿Qué código copia, y a dónde, la siguiente línea de la función boot_aps()? `memmove(code, mpentry_start, mpentry_end - mpentry_start);`

La linea copia el código que se encuentra en `kern/mpentry.S`, a la dirección virtual `0xf0007000`, que mapea a la dirección física `0x7000` (`MPENTRY_PADDR`).

2. ¿Para qué se usa la variable global `mpentry_kstack`? ¿Qué ocurriría si el espacio para este stack se reservara en el archivo `kern/mpentry.S`, de manera similar a `bootstack` en el archivo `kern/entry.S`?

Se utiliza porque cada CPU va a apuntar a un stack distinto. Si se reservara al igual que `bootstack`, entonces los stacks de cada CPU apuntarían a la misma memoria.

3. Cuando QEMU corre con múltiples CPUs, éstas se muestran en GDB como hilos de ejecución separados. Mostrar una sesión de GDB en la que se muestre cómo va cambiando el valor de la variable global `mpentry_kstack`:

```
(gdb) watch mpentry_kstack
Hardware watchpoint 1: mpentry_kstack
(gdb) c
Continuando.
Se asume que la arquitectura objetivo es i386
=> 0xf0100195 <boot_aps+140>:   mov    %esi,%ecx

Thread 1 hit Hardware watchpoint 1: mpentry_kstack

Old value = (void *) 0x0
New value = (void *) 0xf0247000 <percpu_kstacks+65536>
boot_aps () at kern/init.c:106
106         lapic_startap(c->cpu_id, PADDR(code));
(gdb) bt
#0  boot_aps () at kern/init.c:106
#1  0xf010021e in i386_init () at kern/init.c:55
#2  0xf0100049 in relocated () at kern/entry.S:86
(gdb) info threads
  Id   Target Id         Frame
* 1    Thread 1 (CPU#0 [running]) boot_aps () at kern/init.c:106
  2    Thread 2 (CPU#1 [halted ]) 0x000fd412 in ?? ()
  3    Thread 3 (CPU#2 [halted ]) 0x000fd412 in ?? ()
  4    Thread 4 (CPU#3 [halted ]) 0x000fd412 in ?? ()
(gdb) c
Continuando.
=> 0xf0100195 <boot_aps+140>:   mov    %esi,%ecx
```

```
Thread 1 hit Hardware watchpoint 1: mpentry_kstack

Old value = (void *) 0xf0247000 <percpu_kstacks+65536>
New value = (void *) 0xf024f000 <percpu_kstacks+98304>
boot_aps () at kern/init.c:106
106         lapic_startap(c->cpu_id, PADDR(code));
(gdb) info threads
  Id   Target Id         Frame
* 1    Thread 1 (CPU#0 [running]) boot_aps () at kern/init.c:106
  2    Thread 2 (CPU#1 [running]) 0xf01002ac in mp_main () at kern/init.c:124
  3    Thread 3 (CPU#2 [halted ]) 0x000fd412 in ?? ()
  4    Thread 4 (CPU#3 [halted ]) 0x000fd412 in ?? ()
(gdb) thread 2
[Switching to thread 2 (Thread 2)]
#0  0xf01002ac in mp_main () at kern/init.c:124
124     xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up
(gdb) bt
#0  0xf01002ac in mp_main () at kern/init.c:124
#1  0x00007062 in ?? ()
(gdb) p cpunum()
Could not fetch register "orig_eax"; remote failure reply 'E14'
(gdb) thread 1
[Switching to thread 1 (Thread 1)]
#0  boot_aps () at kern/init.c:106
106         lapic_startap(c->cpu_id, PADDR(code));
(gdb) p cpunum()
Could not fetch register "orig_eax"; remote failure reply 'E14'
(gdb) c
Continuando.
=> 0xf0100195 <boot_aps+140>:   mov    %esi,%ecx

Thread 1 hit Hardware watchpoint 1: mpentry_kstack

Old value = (void *) 0xf024f000 <percpu_kstacks+98304>
New value = (void *) 0xf0257000 <percpu_kstacks+131072>
boot_aps () at kern/init.c:106
106         lapic_startap(c->cpu_id, PADDR(code));
(gdb) info threads
  Id   Target Id         Frame
* 1    Thread 1 (CPU#0 [running]) boot_aps () at kern/init.c:106
  2    Thread 2 (CPU#1 [running]) 0xf01002ac in mp_main () at kern/init.c:124
  3    Thread 3 (CPU#2 [running]) 0xf01002ac in mp_main () at kern/init.c:124
  4    Thread 4 (CPU#3 [halted ]) 0x000fd412 in ?? ()
(gdb) bt
#0  boot_aps () at kern/init.c:106
#1  0xf010021e in i386_init () at kern/init.c:55
#2  0xf0100049 in relocated () at kern/entry.S:86
(gdb) thread 3
[Switching to thread 3 (Thread 3)]
#0  0xf01002ac in mp_main () at kern/init.c:124
124     xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up
(gdb) p cpunum()
Could not fetch register "orig_eax"; remote failure reply 'E14'
(gdb) c
Continuando.
```

4. ¿Qué valor tendrá el registro `%eip` cuando se ejecute la línea `movl $(RELOC(entry_pgdir)), %eax` de `kern/mpentry.S`? ¿Se detiene en algún momento la ejecución si se pone un breakpoint en mpen-

try_start? ¿Por qué?

Redondeada a 12 bits, el %eip apuntará a la región de memoria 0x7000 (MPENTRY_PADDR), ya que todo el bloque de código (mucho menor a una página) de mpentry.S se mapeó allí.

La ejecución no se detiene al poner in breakpoint en mpentry_start porque el registro %eip nunca llega a pasar por esa dirección, ya que el código ese se mapeó a 0.

5. Con GDB, mostrar el valor exacto de %eip y mpentry_kstack cuando se ejecuta la instrucción anterior en el último AP.

```
(gdb) b *0x7000 thread 4
Punto de interrupción 1 at 0x7000
(gdb) c
Continuando.

Thread 2 received signal SIGTRAP, Trace/breakpoint trap.
[Cambiando a Thread 2]
Se asume que la arquitectura objetivo es i8086
[ 700:   0]    0x7000:  cli
0x00000000 in ?? ()
(gdb) disable 1
(gdb) si 10
Se asume que la arquitectura objetivo es i386
=> 0x7020:  mov    $0x10,%ax
0x00007020 in ?? ()
(gdb) x10i $eip
orden indefinida: «x10i». Intente con «help»
(gdb) x/10i $eip
=> 0x7020:  mov    $0x10,%ax
   0x7024:  mov    %eax,%ds
   0x7026:  mov    %eax,%es
   0x7028:  mov    %eax,%ss
   0x702a:  mov    $0x0,%ax
   0x702e:  mov    %eax,%fs
   0x7030:  mov    %eax,%gs
   0x7032:  mov    $0x11f000,%eax
   0x7037:  mov    %eax,%cr3
   0x703a:  mov    %cr4,%eax
(gdb) watch $eax == 0x11f000
Watchpoint 2: $eax == 0x11f000
(gdb) c
Continuando.
=> 0x7037:  mov    %eax,%cr3

Thread 2 hit Watchpoint 2: $eax == 0x11f000

Old value = 0
New value = 1
0x00007037 in ?? ()
(gdb) p $eip
$1 = (void (*)()) 0x7037
(gdb)  p mpentry_kstack
$2 = (void *) 0x0
```

Comunicación entre procesos: ipc_recv

1. Un proceso podría intentar enviar el valor númerico -E_INVAL vía ipc_send(). Se completan las condiciones de los if de ambas versiones para detectar correctamente los errores

```c
// Versión A
envid_t src = -1;
int r = ipc_recv(&src, 0, NULL);

if (r < 0)
  if (src == 0) // Comentario en ipc_recv: If the system call fails, then store 0 in *fromenv
    puts("Hubo error.");
  else
    puts("Valor negativo correcto.")

// Versión B
int r = ipc_recv(NULL, 0, NULL);

if (r < 0)
  if (/* ??? */) // No hay manera de detectar el error.
                 // La función recibe NULL tanto en fromenv como perm_store,
                 //     haciendo que no haya puntero contra el cual verificar
    puts("Hubo error.");
  else
    puts("Valor negativo correcto.")
```