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. Inlcuir: 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	Interrupt Number	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 0xefffffbc
[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!
```