

04/05/22.

PRESENTACIÓN TP2

Procesos de usuario.

Introducción a procesos (en JOS).

- ↳ que se pueden tener procesos de usuario
- ↳ implementar syscalls
- ↳ cambios de contexto (de ring 0 a ring 3)

Procesos en JOS = environments

PID	heap
open.	stack
	Data
File Descriptors	
0 1 2 3	Cook

struct Env_t

...

env-id

env-tf (estado) (Trapframe)
del CPU

env-psdir (memoria virtual)
estado del proceso.

inst. pointer } guardan
stack pointer } esto para
el context switch

o shadow de los registros
del CPU de mi proceso.

- ▷ c/ proceso tiene
- o sus su propio
- page directory

Process Control Block (ps).

↳ aneglo de envs. (envs)

↳ mimulan a pages. (lista enlazada)

{ posición libre = struct env que no tiene info,

 posición ocupada = se guarda datos de un proceso
 independientemente si está corriendo
 o no.

Estados

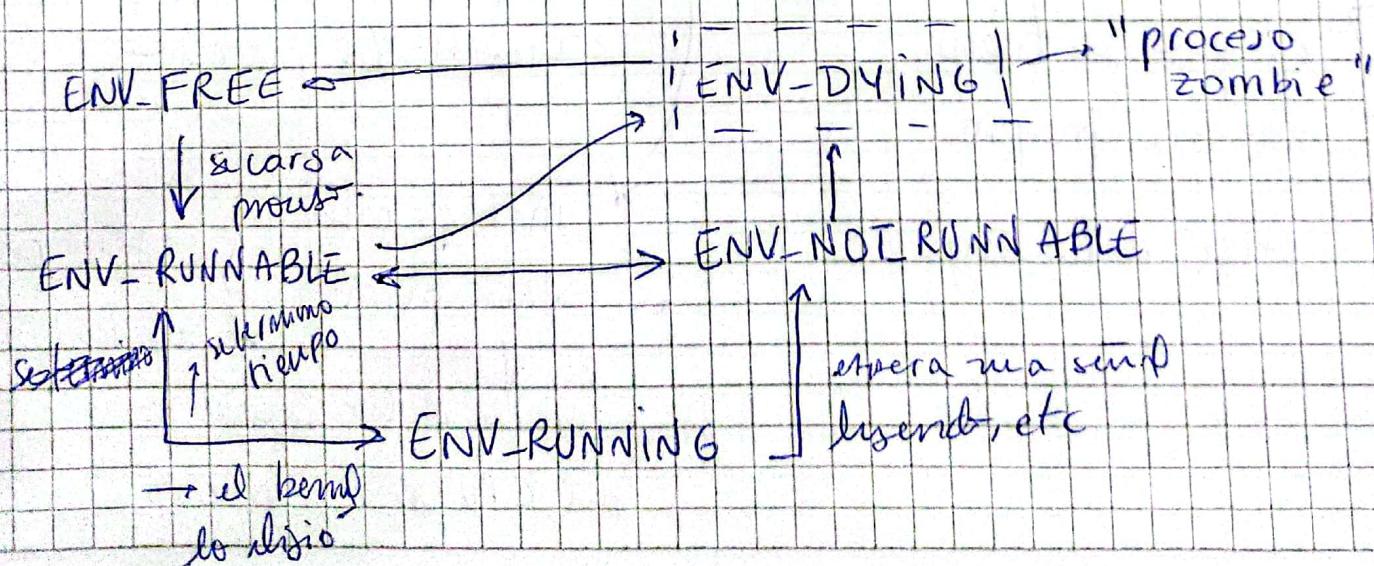
FREE → procesos libres (posiciones libres)

RUNNABLE → listo para ser ejecutado

NOT RUNNABLE → no estoy corriendo y ademas no
puedes ejecutado

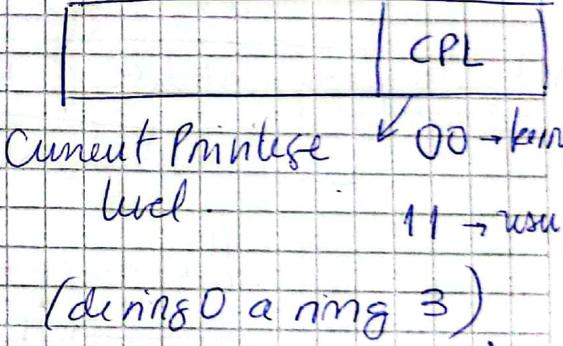
RUNNING → actualmente en CPU.

DYING → estado especial opcional si una
función es terminada mientras
está corriendo en otra CPU.



Los registros en x86.

- segment registers : controlan segmentación . protección al Protection Mode.
- propósito general .
- especiales { sp (stack pointer)
ip (instruction pointer)
flats (registro de status) .
code
- Segment registers . { cs → Code Segment →
ds: data segment últimos 2 bits
es: indican el nivel de
ss: stack segment . pin en lego actual
cs
- control registers



Segmentación en x86 .

(en modo protegido)

segmento = región de memoria .

se definen en una tabla de descripciones de los
de memoria .

Lo que indica al CPU mediante (g & tk)

→ definen qué permisos se tienen para usar el segmento (descriptor privilege level)

→ Los registros de segmento indican cuál entrada en la tabla usar y con qué perm.

{Descriptor}

RS

Base Address	segment limit
--------------	---------------

CS

segment desc	CPL
--------------	-----

DS

segment desc	RPL
--------------	-----

CPL

RPL

privilege
check

is ok to access?

*

RPL

base address	size	PPL
--------------	------	-----

*

VA

†

sp1 base address

=

LINEAR ADDRESS

Todos comienza como base address 0x0 y como límite 4GB

$$\hookrightarrow \text{desde VA} = \text{L.A.}$$

$$VA + 0x0 = L.A. = V.A.$$

$\begin{cases} GD-UT \text{ y } GD-UD \rightarrow CS \text{ y } DS \text{ en user mode} \\ GD-KT \text{ y } GD-KD \rightarrow CS \text{ y } DS \text{ en kernel mode.} \end{cases}$

~~Nota~~

Tarea 1 - Arreglo envs

- Inicializaciones.
 - mem-init-envs() \rightarrow se reserva memoria (en boot loader)
 \hookrightarrow ¿cómo reservamos y mapeamos?
 - env-init() \rightarrow similar a mem-init().
 \hookrightarrow ¿Qué hace este comando? \rightarrow kernel
 \hookrightarrow ¿Cómo mapeo el espacio del kernel? \rightarrow no se va a poner en envs.
 \rightarrow desde el arreglo de envs lo va a estar vacío.
 - env-alloc() \rightarrow ya implementada pero actualizada. Encuentran env struct libre, limpiando y duobrando.

¿Qué tiene que tener el page directory de un proceso nuevo?

→ todo lo que está en el kernel

→ Pages

→ MVS.

env-setup-vm()

↳ p = nuevo page directory.



| copia del page directory del kernel |

(↳ NO reforencia)

→ No se modifica el cr3, porque las func pgdir-walk, insut etc. las mecanismos xa reconen ese pgdir nuevo.

↳ si modifiquamos el cr3 nos vamos a ir al pgdir del ~~process~~ proceso y no del kernel.

⇒ Ahora falta mapear la "estructura" del proceso, mi código, el stack, el heap, el data.

Heap: no tenemos, hasta que se implemente algún equivalente a sbrk()

Stack: al principio está vacío xa tengo que asignar una página xa el stack.

(USTACKTOP) : ¿qué le digo donde está su stack?
↳ con el stack pointer.

→ Le especificamos con qué SP va a ser inicializado. → tenemos que marcar una pila vacía en esa direc.

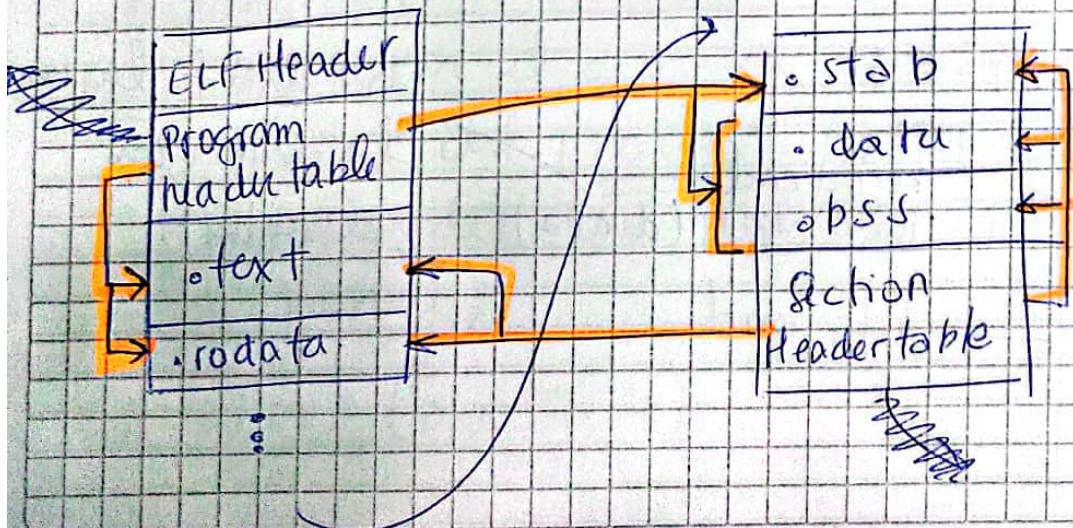
→ Data y Code.

↳ están en el binario → hay que cargarlo en memoria pero ¿cómo?

↳ ej si hay un jump, ya tiene las direc. virtuales específicas que necesita no lo puedo cargar en cualquier lado.

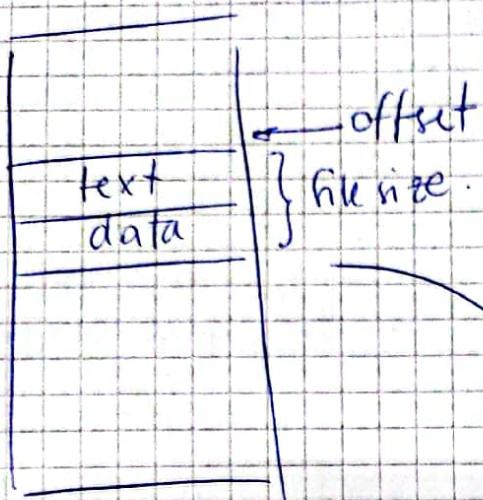
↓
Necesito una convención para traducir esa direc virtual. → ELF.

↓
Lo hace el ELF y esto nos ve a decir dónde se tiene que cargar.

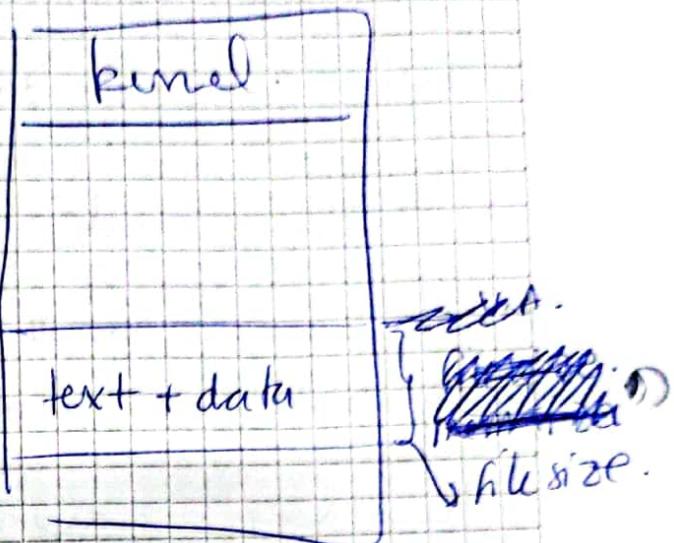


=> el ELF se lee con read-elf. -a binario.

ELF



Page Directories del proceso.



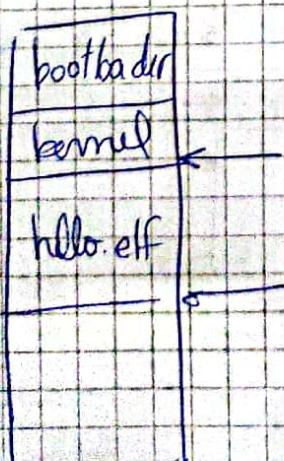
=> a veces memsize > filesize → el espacio que sobra se nullifica con ceros.

=> No más intrusa la physAddr

✓ los programas de usuario están en jos/obj y los binarios están en jos/obj.

=> Cómo accedemos al binario si no hay filesystem? 7 10

↳ x ahora están enbaldos en la imagen del kernel.



kernel end
hello_start (dirva)

hello_end

↳ el elf leyendo la dirección directa.

=> terminos que implementan env_create.

↳ casteamos el punto a Elf y lo leemos como ELF.

↓
hay struct Elf.

↳ usar VA, hizise xa cargarlo en el page directory

↓

se llama load_icode() → acá se castea.

struct Elf* my_elf = (struct Elf*) binary;
myElf
for xa cada program header

region_alloc() → aloca pag física en
dir. virtual.

→ copy de desde el binario hasta dir virtual.

=> LEER bootmain que hace nroone el ELF de la
manera que nosotras queríamos.

✓ memcpy o memset. => ANTES tengo que
cambiar el cr3 xa ~~quedas~~ cambiar
el pgdir del proceso. (ahora si que no estar
en el pgdir del proceso y no la del
kernel).

Ucr3 (dir física).

(instrucción pointer del proceso).

▷ Entry point → el ip del proceso
del ELF.

Tarea 3 - Ejecución de un proceso

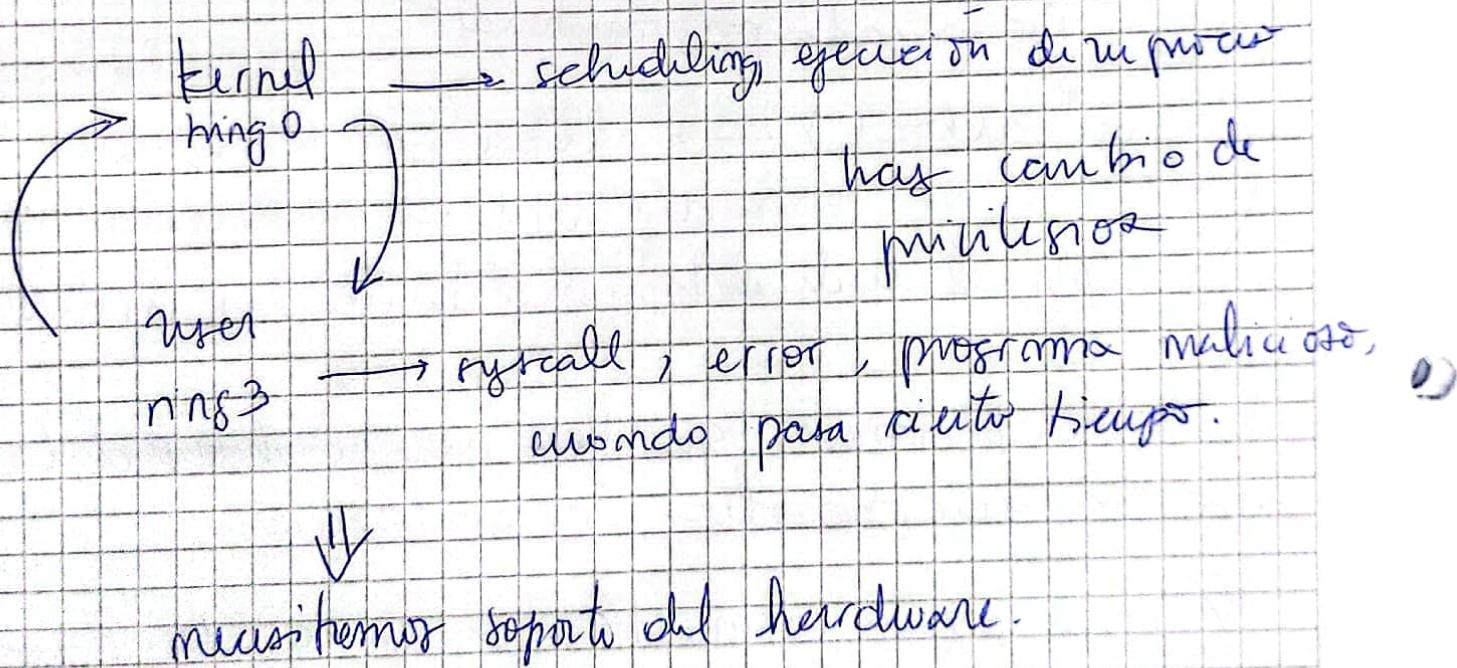
¿Cómo mandamos al env "a correr"?

1. Cargar los res de prop. general

2. Cargar los res de tipo cs.

3. Modificar el cr3 xa para que apunte al pdidiral memoria.

Cambio de contexto.



User a kernel.

→ se utilizan INTERRUPCIONES.

↓
es la única forma xq tienen privilegios.

- intento de memoria ~~sea~~ prohibida
- intento de usar una syscall.

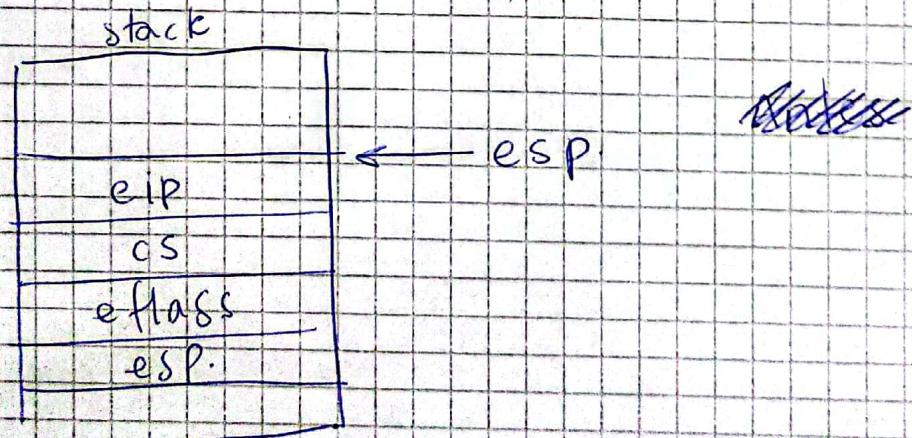
kernel a user.

↳ instrucción especial iret. (interrupt return)

Básicamente se
recupera la info del
estado del proceso
cuando se generó
una interrupción
antes.

guarda dirección del stack una
dirección de memoria para saltar
a esa posición. (xq volver).

↳ esa dirección se guarda cuando
se que llenan a otra instrucción
pero en algún momento tiene
que volver.



Trapframe (env-tf) → representación de los registros del CPU.

tf-eip
tf-CS
tf-paddins
tf-eflags.

} lo que el iret va a leer.

↳ están en el orden indicado para hacer pop y push del stack a los registros reales.

env-pop-tf() → leerla

env-run() → implementar.

gdb-hello → seguimiento de la ejecución de un programa.