Contents

1 VIRTUALIZACIÓN

1.1 Primer tipo de ejercicio:

Ejercicio 1. Explique detalladamente como funcionan estos programas indicando cuantos procesos se crean, que hacen los hijos y padres, que archivos se cierran y abren, etc.
Suponga que en ambos casos se invocan al programa compilado `./a.out` de la siguiente forma:

```
$ ./a.out A B C D E F
```

```c
int main(int argc, char ** argv) {
  int rc = fork();
  if (rc<0)
    return -1;
  else if (0==rc)
    execvp(argv[0], argv);
  else {
    execvp(argv[0], argv);
  }
}
```

```c
int main(int argc, char ** argv) {
  char buf[L] = {'\0'};
  if (argc<=1)
    return 0;
  int rc = fork();
  if (rc<0)
    return -1;
  if (0==rc) {
    close(0);
    open(argv[0], 0);
    read(0, buf, L);
    write(1, buf, L);
  } else {
    argv[argc-1] = NULL;
    execvp(argv[0], argv);
  }
}
```

1.1.1 Cosas a tener en cuenta.

- fork(); Crea un proceso. Devuelve 0 en caso hijo, >0 en caso padre. El if es para indicar que hace el padre y el hijo en base al return
- execvp(); Esta funcion recibe como argumento un programa y un array, el array contiene el programa y argumentos.
  - **IMPORTANTE**: NO CREA UN NUEVO PROCESO, SI NO QUE REEMPLAZA AL QUE SE ESTABA EJECUTANDO
- FD: Un file descriptor es un int que sirve para identificar archivos abiertos. 0 STDIN (input) 1 STDOUT (output) 2 STDERR(error)
- close(FD); Cierra el archivo y FD que le pasamos, sin mas.
- open(archivo, flag); Abre en el FD mas chico (numero) el archivo que le pasamos con la flag que le pasemos xdd. si la flag es 0, entonces solo tiene permiso de lectura. IMPORTANTE
- read(FD, buffer, cantBytes); Lee el FD y lo mete en el buffer. La cantBytes vendria a ser el tamaño
- write(FD, buffer, CantBytes); Escribe en el FD lo que haya en buffer, la cantBytes seria el tam del buffer

2 Paginacion

Ejercicio 2. Para el sistema de paginado i386 (10, 10, 12), sabiendo que `CR3=0x00000` y que el contenido de los marcos físicos son los siguientes:

```
0x00000                 0x10000
------------------      ------------------
0x000: 0x10000 (P)      0x000: 0x00000 (P)
0x001: 0x10000 (P)      0x001: 0x00001 (P)
...                     ...
...                     ...
0x3FE: 0x10000 (P)      0x3FE: 0x003FE (P)
0x3FF: 0x00000 (P)      0x3FF: 0x003FF (P)
```

(a) Pasar de virtual a física: `0x00000EEF`, `0x00001FEE`.

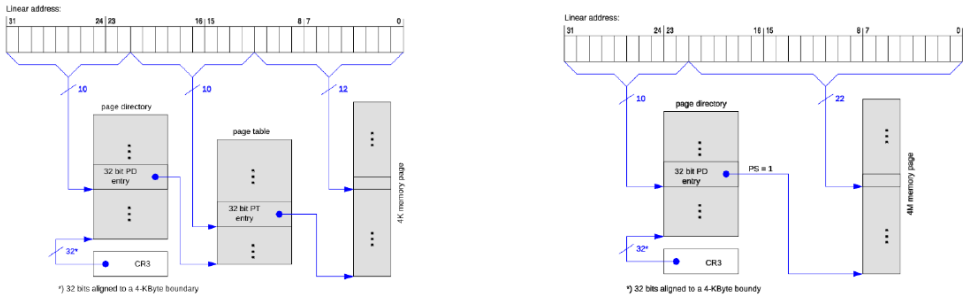
(b) Pasar de **física** a virtual: `0x00000EEF`, `0x00001FEE`.

(c) Indique qué contiene la dirección de memoria virtual `0xFFC00CFF`.

(d) Modificar el mapa de memoria virtual de forma tal que `0x00000EEF` apunte a `0x00000EEF` y `0x00400EEF` apunte a `0x00001EEF`.

0x0BABA	0x0BEBE	0x0B0B0
0x000: 0x10000 (P)	0x000: 0x0BEBE (P)	0x000: 0x00000 (P)
0x001: 0x10000 (P)	0x001: 0x103FF (P,PS)	0x001: 0x00001 (P)
...	...	...
...	...	...
0x3FE: 0x10000 (P)	0x3FE: 0x0B000 (P,PS)	0x3FE: 0x001FE (P)
0x3FF: 0x00000 (P)	0x3FF: 0x0BABA (P)	0x3FF: 0x001FF (P)

Ejercicio 2. Para el sistema de paginado i386 con el bit de PSE (page size extension) dentro de CR4 habilitado, el esquema de paginación puede ser $(10, 10, 12)$ o $(10, 22)$, dependiendo del bit 7 de cada entrada del *pagedir* denominado PS (page size).



Sabiendo que CR3=0x0BEBE y que el contenido de los marcos físicos son los siguientes:

2.1 Primero que hay que hacer

identificar es el sistema de paginado con el que vamos a trabajar.

2.2 caso del ejemplo 1

el sistema es i386 (10,10,12) esto quiere decir que en este esquema Los primeros 10 bits son para indexar el page directory, luego, tenemos 10 bits para indexar la page table y por último los 12 bits son el offset dentro de una page.

Sin embargo, en el ejemplo 2, como lo indica el enunciado, el esquema de paginacion va a depender del bit **PS** de cada entrada del **pagedir**.

El esquema en tanto puede ser (10,10,12) o en su defecto (10, 22).

2.3 Pasaje de direcciones

Es posible que se nos pida pasar direcciones virtuales a física o viceversa. También se nos puede pedir que pasemos direcciones físicas a TODAS las virtuales.

2.3.1 pasar direcciones virtuales a físicas

tenemos que pasar la dirección que se nos provee en hexa a binario y separar los bits según el esquema de paginación que tengamos, luego de eso, tenemos que seguir el "camino" correspondiente hasta llegar a la dirección deseada.

2.3.1.1 ejemplo 1) se nos pide pasar la dirección virtual 0x00000EEF a física.

0x00000EEF = 0x00000EEF= 0000 0000 0000 0000 0000 1110 1110 1111

PD PT OFFSET

Como CR3 = 0x00000

Luego, la dirección física es 0x00000EEF

2.3.2 Pasar de física a virtual

el tip está en deducir "todos los caminos que me lleven a esa direccion fisica".

A modo de ejemplo tenemos la direccion fisica 0x00000EEF

3 POSIBLE EJERCICIO DE ASSEMBLER EXTRA

3.1 Ejercicio

Ejercicio 2. Mostrar la secuencia de accesos a la memoria física que se produce al ejecutar este programa assembler x86_32, donde $Base_{code}$ = 0x1000, $Bounds_{code}$ = 0x100, $Base_{data}$ = 0x2000, $Bounds_{data}$ = 0x10
La versión original de este código salió en el número de abril de 1967 de Cosmopolitan.

```
 0: mov   $0x0,%edx
 6: mov   0x0(%edx),%ecx
12: mov   0x4(%edx),%eax
18: add   $0x8,%edx
21: add   (%edx),%ecx
23: add   $0x4,%edx
26: sub   $0x1,%eax
29: jne   21
```

```
0x0:  .long 0x0
0x4:  .long 0x4
0x8:  .long 0x3
0xC:  .long 0x7
0x10: .long 0x2
0x14: .long 0x1
0x18: .long 0x5
```

3.2 Cosas a tener en cuenta

- Aca nos dice que nuestro codigo arranca en 0x1000 y tiene un limite de 0x100.
- Osea que si en codigo te vas de 0x1100 da SF (Segmentation Fault).

Despues tenemos que tener en cuenta que la data arranca en 0x2000 y tiene un limite de 0x10. Osea que si nos pasamos para 0x2010 da SF de nuevo. **Observemos tambien que como tenemos una direccion a la izquierda, la asignacion es de derecha a izquierda.**

Bien, cada vez que accedamos a una linea, se hace el calculo:

Base$_{\text{code}}$ + linea.**

Ejemplo, supongamos que ejecutamos la linea 18:

0x1000 + 0x18 = 0x1018.

Luego, en tema Base$_{\text{data}}$ hay que revisar que tenemos algo a la izquierda en la linea que puede sumar.

Por ejemplo, en la linea 12 0x4(%edx),%eax, si bien %edx = 0x0, como tiene esa cosa a la izquierda se convierte en 0x0 + 0x4.

Bien, luego fijemonos que no pasa nada si queremos acceder a %edx que seria %0x0. Puesto que nuestra base data se la banca. Pero fijemonos que en la linea 29 va a loopear de nuevo. Cuando nuestro programa quiera acceder a 0x10 o mas, va a dar SF.

Bien, en este tipo de ejercicios no hay mucha complicacion. En el a y b deberiamos de intentar lograr que se llegue a ese escenario usando los programas enteros o por lineas. En general son faciles. **Puede ser que no exista caso de ejecu-**

Ejercicio 3. Para el programa de la Figura 1, donde la **atomicidad es línea-a-línea**:

(a) Muestre un (1) escenario de ejecución con $N = 5$ de forma tal que la salida del multiprograma cumpla con $a = [0, 0, 0, 0, 1]$.

(b) Muestre un (1) escenario de ejecución con $N = 4$ de forma tal que la salida del multiprograma cumpla con $a = [0, 0, 2, 2]$.

(c) Exprese **todos los valores posibles** de salida de arreglo `a` para la Figura 2. Explique.

(d) Sincronice el multiprograma de la Figura 1 **usando semáforos** para que el resultado **siempre** sea $a = [0, 0, ..., 0, 1]$ para cualquier valor de $0 < N$. Puede colocar condicionales (`if`) sobre el valor de `i,j` para hacer wait/post de los semáforos. Ninguna variable del programa se puede modificar.

Pre: 0<N ∧ i,j=0 ∧ (∀k : 0≤k<N : a[k]= 2)	

1 P0: while (i<N) {
2 a[i] = 0;
3 ++i;
}
	a P1: while (j<N) {
	b a[j] = 1;
	c ++j;
	}

a=?	

Figura 1: Concurrent Vector Writing (CVW)

Pre: 0<N ∧ i,j=0 ∧ s,t=0, 2 ∧ (∀k : 0≤k<N : a[k]= 2)	

P0: while (i<N) {
sem_wait(s);
a[i] = 0;
++i;
sem_post(t);
}
	P1: while (j<N) {
	sem_wait(t);
	a[j] = 1;
	++j;
	sem_post(s);
	}

a=?	

Figura 2: CVW sincronizado

cion

c)En este caso prestarle atencion a el valor del semaforo, si es 1, entonces seria todo [0,0,0,0,..,0]. En este caso tenemos 2, por lo que hay que ver los casos donde gane el de la derecha o el de la izquierda, pero siempre arrancaria por 0.

d) En este tipo de ejercicio tenes que rebuscartela un poco, podes usar un if para postear despues de cierto I o J ponele.

En este ejercicio es facil, fijate en a) que la mayoria de veces no se cumple, digamos en cualquier ejercicio porque hay muchas formas de salir del bucle. y en el b y c deberias ver si podes llegar dividiendolos por alguno que multiplique

Ejercicio 4. Considere los procesos `P0` y `P1` a continuación, donde las sentencias son atómicas.

Pre: n= 0	

P0 : while (n<100) {
n = n+1;
}
	P1 : while (n<100) {
	n = n*3;
	}

Post: n= 66	

(a) ¿Se cumple la postcondición? Demostración rigurosa o contraejemplo.

(b) Sincronizar con semáforos para que siempre dé como resultado `n=666`. Puede colocar condicionales (`if`) sobre el valor de `n` para hacer wait/post de los semáforos.

(c) Sincronizar con semáforos para que siempre dé como resultado `n=243`. Puede colocar condicionales (`if`) sobre el valor de `n` para hacer wait/post de los semáforos.

ponele.

a) Este ejercicio es igualisimo al practico. 1/CantRPM, y vas pasandolo primero a segundos (*60) y despues a Ms (*1000)

Ejercicio 5. El disco *Pepito Digital Green* de 2 TiB e intefaz SATA-3 tiene una velocidad d
RPM, 6.5 ms de latencia de búsqueda y 80 MiB/s de tasa de transferencia máxima (un dis
olvido).

(a) Indicar cuantos *ms* tarda en dar una vuelta completa.

(b) Indicar la tasa de transferencia de lectura **al azar** de bloques de 8 MiB.

(c) Si la tasa de transferencia máxima está dada por la velocidad rotacional que no requ
(no sufre del *seek time*), deducir cuantos MiB almacena cada pista.

b) En este ejercicio hay 3 cosas a calcular, primero la transferencia.

Transferencia = TamBloque/TransfMaxima

TiempoIO = Latencia + (VueltaCompletaMs/2) + Transferencia

LecturaIO = TamBloque/TiempoIO

a) En este ejercicio lo que calculamos es LectCabezal =TransfMaxima/Cabezales. Luego LectCabezal/RPS donde RPS = RPM *

60.

b) para el inode-bitmap debemos pasar los inodos a bits y pasarlo a KiB. Notar que como estan en bits primero habria que dividir por 8 para obtenerlos en Bytes. Y para el data-bitmap, sabemos que cada bloque ocupa un bit en el data-bitmap, y que tenemos 512MiB repartidos en 4KiB. Por lo que si los dividimos obtendremos el tamaño. (Capaz conviene tenerlos en bytes o no se, cosa suya ea).

Ejercicio 6. En un sistema de archivos de tipo UNIX, tenemos los bloques de disco dispuestos dentro del *i-nodo* con 12 bloques directos, 1 bloque indirecto, 1 bloque doble indirecto y 1 bloque triple indirecto. Cada bloque es de 4 KiB y los números de bloque ocupan 32 bits.
La partición está formateada con 262144 i-nodos y una región de datos de 512 MiB.
(a) Calcule el tamaño en KiB del *inode-bitmap* y del *data-bitmap*.
(b) Suponiendo un tamaño de inodo de 128 bytes, calcule el tamaño de la tabla de inodos.
(c) ¿Es posible agotar la capacidad de almacenamiento de la partición sin almacenar un solo byte de datos?
(d) ¿A partir de que tamaño en KiB el archivo empieza a utilizar los bloques doble indirectos?

c) size tabla de i-nodes = cantidad de i-nodes * size de struct inode

d) Es posible, si usamos todos los inodes disponibles con archivos vacios

e) Para este tipo de pregunta, calculamos (bloques directos + indireccion^1+...+indireccion^n)*TamBloque

Ejercicio 6. En un sistema de archivos de tipo UNIX, tenemos los bloques de disco dispuestos dentro del *i-nodo* con 8 bloques directos, 1 bloque indirecto y 1 bloque doble indirecto. Cada bloque es de 1 KiB y los números de bloque ocupan 16 bits.
(a) Calcule la capacidad máxima de un archivo.
(b) Calcule la capacidad máxima del *block pool* del sistema de archivos.
¿Hay algo raro entre la capacidad máxima de un archivo y del disco?
(c) ¿A partir de que tamaño en KiB el archivo empieza a utilizar los bloques doble indirectos?

Este es el otro tipo de ejercicio que te podes topar.

a) Para esto tenes que calcular

CantidadPunteros = TamBloque / numerosDBloque (Esto siempre son los bits que te indican)

Luego, tenemos que realizar el calculo (Bloques directos + cantblockInd(CantPunteros)^1 + … + cantBlockInd(cantPunteros)^n)*tambloc

a) Aca te lo pueden decir como data region o como block Pool. La formula es $2^{\text{BitsNumBloque}}$ * TamBloque

Lo que puede pasar de raro es que el archivo sea mas grande que la data region

a) Este lo explique antes, es hacer el calculo del a) pero hasta la indireccion anterior a la que te piden