

1. (memoria virtual)

- A. ¿Qué es la memoria virtual? ¿Qué mecanismos conoce, describa los tres que a ud. le parezcan los más relevantes?
- B. Explicar el mecanismo de address translation **memoria virtual paginada** de tres niveles de indirección de 32 bits. Indique la cantidad de direcciones de memoria que provee, una virtual address :

7 bits	7 bits	6 bits	12 bits
--------	--------	--------	---------

con tablas de registros de 4 bytes.

39 pts.

1	2	3	4
5555	5555	554	5

2. (scheduling)

- A. ¿Qué es un **context switch**? En un context switch cuales de las siguientes cosas no deben/deben ser guardadas y porque: a)registros de propósito general, b) translation lookaside buffer, c) program counter, d) Page Directory Entry , e) Pcb entry, e) ninguna.
- B. Explique la política de scheduling **MLFQ** detalladamente. Sea 1 proceso, cuyo tiempo de ejecución total es de 40 ms, el time slice por cola es de 2 ms/c pero el mismo se incrementa en 5 ms por cola. Cuantas veces se interrumpe y en qué cola termina su ejecución .

3. (procesos)

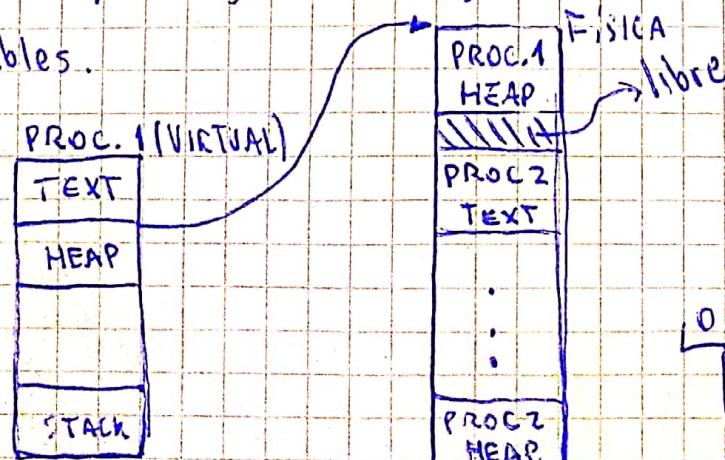
- A. ¿Qué es el stack? Explique el mecanismo de funcionamiento del stack para x86 de la siguiente función `int read(void * buff, size_t len, size_t num, int fd);`. Como se pasan los parámetros, dirección de retorno, etc.
- B. ¿Qué es el **Address Space**? ¿Qué partes tiene? ¿Para qué sirve?. Describa el/los mecanismos para crear procesos en unix, sus syscalls, ejemplifíquelo.

4. (Concurrencia)

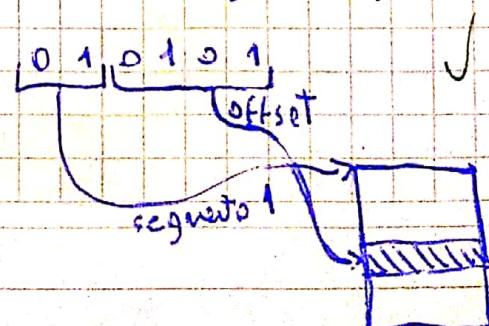
- A. Describa que es un thread y use su API para crear un programa que use 5 threads para incrementar una variable compartida por todos en 7 unidades/thread.
Hasta llegar a 1000
- B. Defina y dé ejemplos: race-condition, heisenbug, dead-lock, interleave

1) La memoria virtual es la memoria desde el punto de vista del proceso. Es la abstracción de la memoria física. Gracias a la memoria virtual se le puede dar la ilusión al proceso de que tiene toda la memoria disponible para él. Un buen mecanismo que logre eso debe ser transparente, para que el proceso no sepa lo que ocurre y no tenga que cambiar nada de su implementación debido a esto. Eficiente, para que el mecanismo influya lo menos posible en términos de tiempo de ejecución y uso de memoria. Y debe también proteger, para que un proceso no pueda acceder a datos en memoria que sean de otro proceso sin el permiso de éste.

Un mecanismo posible es el de segmentación, el cual es importante porque hoy en día se lo sigue utilizando bastante, aunque no exactamente de la misma forma en la que fue pensado en sus inicios. Consiste en que el procesador tenga registros especiales a través de los cuales el sistema operativo le indica los límites de los segmentos de memoria del proceso actual (text, heap, stack), marcando donde empiezan y donde terminan. El problema que tiene este mecanismo es la fragmentación externa, ya que pueden quedar espacios contiguos libres en la memoria que no sean suficientemente grandes como para asignarlos al segmento de algún proceso, por lo que quedan inutilizables.



Al haber 3 segmentos, la traducción se hace tomando los primeros 2 bits (mas altos) para indicar el segmento, y el resto como offset.

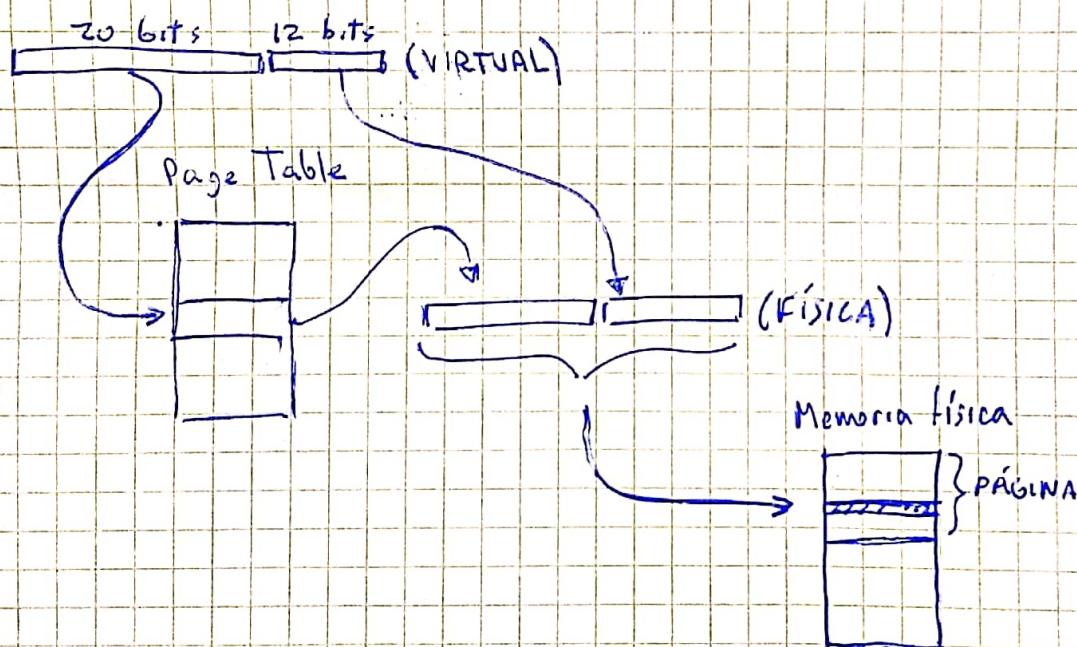


Notas:

de 1 nivel

Otro mecanismo es el de la memoria paginada. En este mecanismo se divide la memoria en bloques fijos a diferencia de bloques de tamaño variable.

Estos bloques suelen ser de 4KiB. Para este mecanismo se le debe indicar a la Memory Management Unit (MMU), en qué dirección física se encuentra una tabla conocida como Page Table, donde se encuentra la información necesaria para traducir las direcciones virtuales en físicas. La traducción se hace de la siguiente manera: Al tener páginas de 4KiB y direcciones de 32 bits, se toman los 20 bits más altos para indicar la página, y los otros 12 como offset. Los 20 bits más altos indican la entrada de la Page Table donde se encuentra la dirección física.



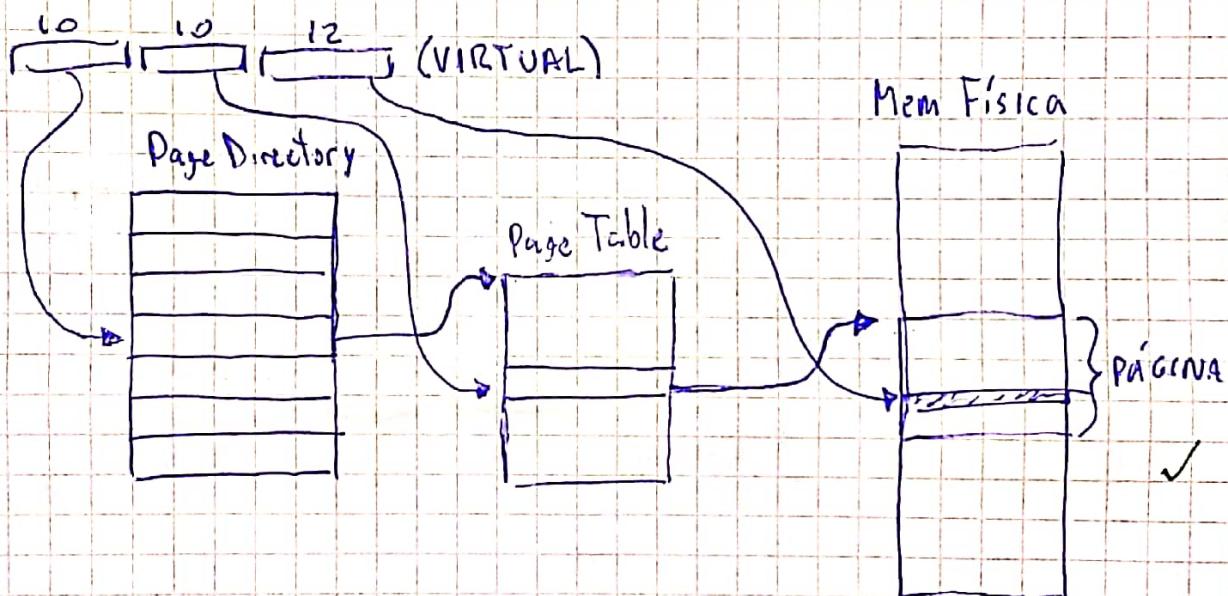
El problema que surge por este mecanismo es el tamaño de la page Table. Al estar guardada en memoria, se debe guardar un arreglo de 2^{20} entradas para cada proceso. Eso es mucho.

El tercer mecanismo que soluciona en parte ese problema (al menos para direcciones de 32 bits) es la memoria paginada de 2 niveles.

En este caso en lugar de una única Page Table, se tiene lo que se conoce como Page Directory.

Cada entrada de la Page Directory contiene la dirección física de una page Table. El beneficio es que no va a haber PageTable si aún no se está utilizando ninguna dirección que se encuentre en su rango.

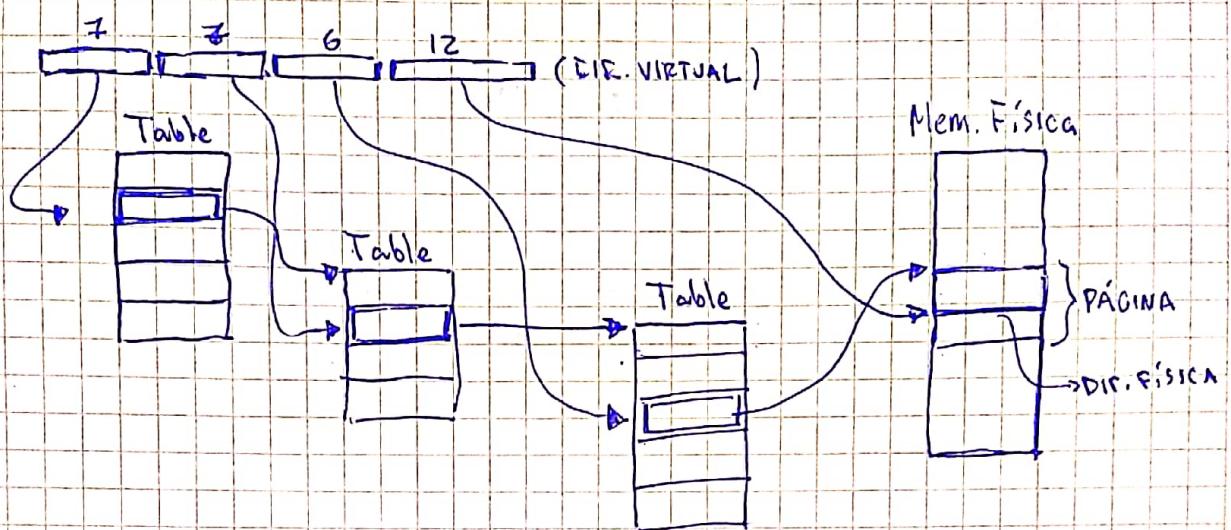
En este caso se utilizan los 12 bits más bajos para offset, y los 20 bits más altos se dividen en 10 bits para indicar la entrada en el page directory y los otros 10 para la entrada a el page Table.



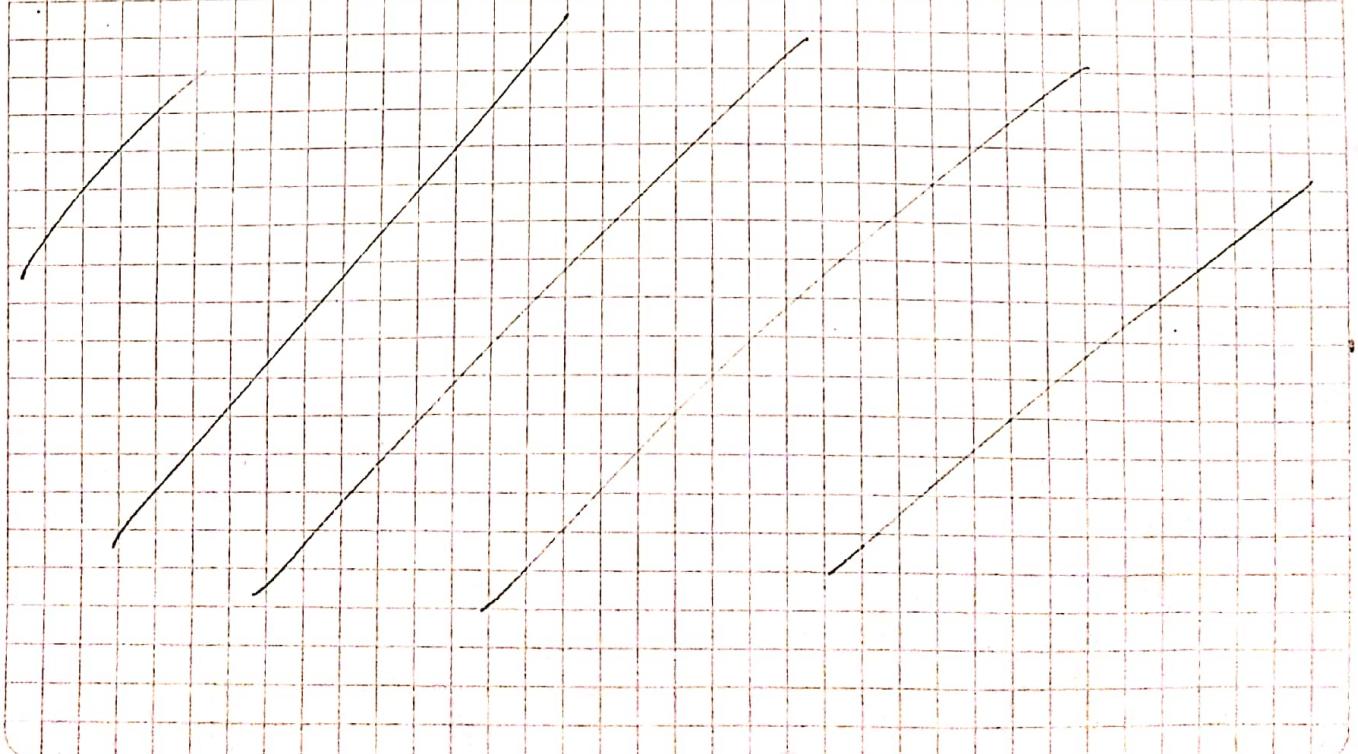
De esta manera, el Page directory únicamente va a tener 2^{10} entradas y la Page Table También 2^{10} , con la diferencia de que este último sólo va a existir si hay una dirección mapeada que requiere su existencia.

Notas:

b) Para el mecanismo de memoria paginada de tres niveles se tiene un mecanismo similar al de 2 niveles, con el siguiente esquema.



Va a haber 2^{32} direcciones de memoria posibles. ~~Porque no importa como~~
~~seas los bits. En este caso por ejemplo, tenemos 7, 6, 5, 12~~
~~estos. Porque vamos a tener 2^7 entradas en el primer nivel, 2^7 entradas~~
~~en el segundo, y 2^6 en el tercero, entonces eso nos da 2^{32} direcciones.~~ ✓
 En realidad se va a poder siempre que los registros tengan tamaño suficiente ~~para~~
~~almacenar los índices de la siguiente tabla~~ para poder direccionar a las tablas
 del próximo nivel.



2)

A) Un context switch es lo que hace un sistema operativo cuando decide dejar de correr un proceso para darle lugar a otro. Consiste en guardarse todo lo necesario para poder, en el futuro, volver a poner en marcha el proceso que fue desalojado en el mismo estado en el que estaba. La información que se guarda se conoce como contexto del proceso, y de ahí viene el nombre context switch, ya que al cambiar de proceso, además, se restauran esos valores guardados para el proceso que se está lanzando.

- a) Sí, porque son utilizados intermitentemente en el programa que el proceso está ejecutando y pueden ser modificados por otro proceso.
- b) No, porque es interno de la MMU. ✓
- c) Sí, porque indica la instrucción que se está ejecutando y es necesaria para que cuando se vuelva a correr el proceso siga desde donde estaba ✓
- d) Sí, porque cada proceso tiene su espacio de direcciones y la información para la traducción está allí. ✓
- e) Sí, porque es donde se guarda todo lo anterior (lo que puse que sí) ✓

B) La Política MLFQ consiste en tener varias colas (usualmente 8) donde cada una representa un nivel de prioridad de ejecución. Lo que va a hacer el scheduler es ejecutar utilizando Round-Robin las tareas que se encuentren en la cola de mayor prioridad, y hasta que no terminen, no se pasa a la siguiente cola, donde vuelve a ejecutar de la misma forma las tareas que se encuentren.

Las tareas no se mantienen en una misma cola sino que su prioridad puede

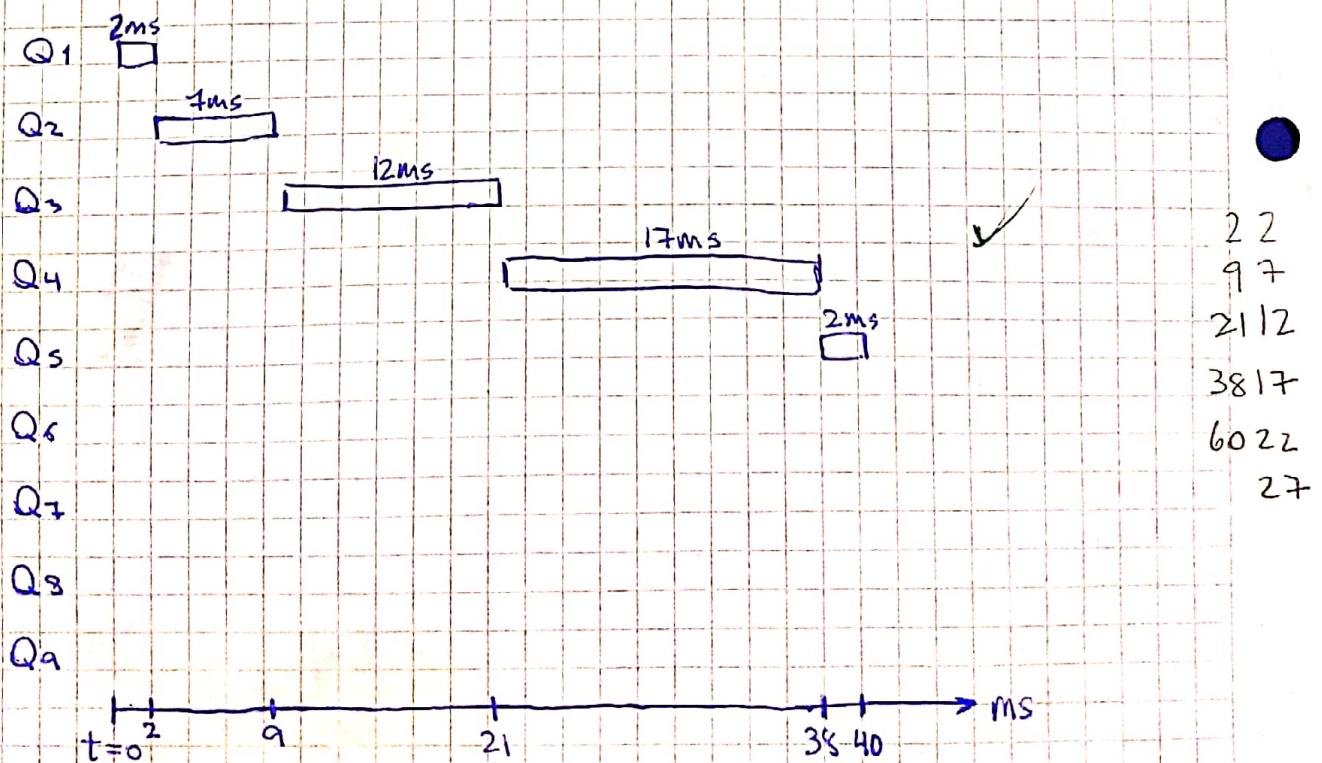
Notas:

cambiar de acuerdo a su comportamiento, y para eso MLFQ tiene reglas:

- 1) Todas las tareas están en la prioridad más alta al inicio
- 2) Si una tarea usa tiempo equivalente a un time slice de la cola donde se encuentra su prioridad (ej: en 1 cola)
- 3) Mientras no haya usado 1 time slice su prioridad se mantiene
- 4) Cada cierto tiempo se suben todas las tareas a la máxima prioridad

Lo que permiten estas reglas es que si una tarea usa poco tiempo de procesamiento quiere decir que usa mucho I/O, por lo que las tareas interactivas mantienen una alta prioridad y por lo tanto un buen response time. Para las tareas que usan mucho tiempo de procesamiento, la regla 4 evita que nunca les llegue su turno, por lo que también tienen garantizado el uso de la CPU en algún momento.

Para una tarea que dura 40 ms, su planificación va a ser de la siguiente manera:



Termina su ejecución en la cola 5 y se la interrumpe 11 veces.

3) El stack es una región del espacio de direcciones que funciona como una pila. Mediante las instrucciones push y pop (en x86) se pueden apilar valores a esa región y desapilarlos. Para eso se tiene un registro llamado "stack pointer" el cual siempre tiene la dirección de memoria del topo del stack, por lo que al apilar un valor se incrementa el S.P. en 4, y al desapilar, se descuenta en 4 también.

Para la llamada a una función en x86 se tiene una convención llamada "call convention". Por lo que siguiendo esa convención, la llamada a "read" se hará de la siguiente manera.

- Se puslean los registros "caller saved" (eax, ebx, ecx, edx)
- Se puslean los parámetros en orden inverso (fd → num → len → buff)
- Se ejecuta la instrucción call que pusea la dirección de retorno y salta a la función.

Luego, la función read tiene que asegurarse de que al momento de retornar, el stack se encuentre en el mismo estado en el que se recibió, para eso tiene que:

- Guardar los registros "callee-saved" y restaurarlos antes del ret
- Hacer la misma cantidad de push y pop para los valores que necesite tener en el stack internamente.

En el ret se hace pop de la dirección de retorno y se vuelve a la rutina que hizo la llamada con el stack "intacto".

B) El Address space es conocido como la memoria principal del proceso. El proceso ve que toda la memoria le pertenece pudiendo usar cualquier dirección, cuando en realidad esa es la ilusión que provee el sistema operativo.

El address space se divide en 4 partes:

- .text : región donde se encuentra el código -
- .data : constantes globales -
- .heap : memoria dinámica que el proceso pide al sistema operativo -
- .stack : explícito en punto A -

Sirve para proteger a los procesos de otros, y que no puedan acceder a la memoria del otro.
Para crear un proceso en UNIX el sistema operativo se encarga de fabricar el espacio de direcciones con esas 4 regiones. Para eso copia el código del programa desde donde está almacenado y lo mapea en .text, copia las constantes globales y las mapea en .data, reserva memoria para el heap y el stack y setea el stack pointer para que apunte al tope de la pila.

Las syscalls involucradas en la creación de un proceso son fork() y exec().

La syscall fork realiza una copia del proceso actual y todo su espacio de direcciones, resultando en 2 procesos, el proceso padre es el que llama a fork() y el proceso hijo es el creado. A partir de ésta llamada continúa la ejecución desde ambos procesos desde la instrucción siguiente a fork() y para diferenciarlos le retorna al padre el process ID (PID) del hijo, y al hijo le devuelve cero.

Ejemplo:

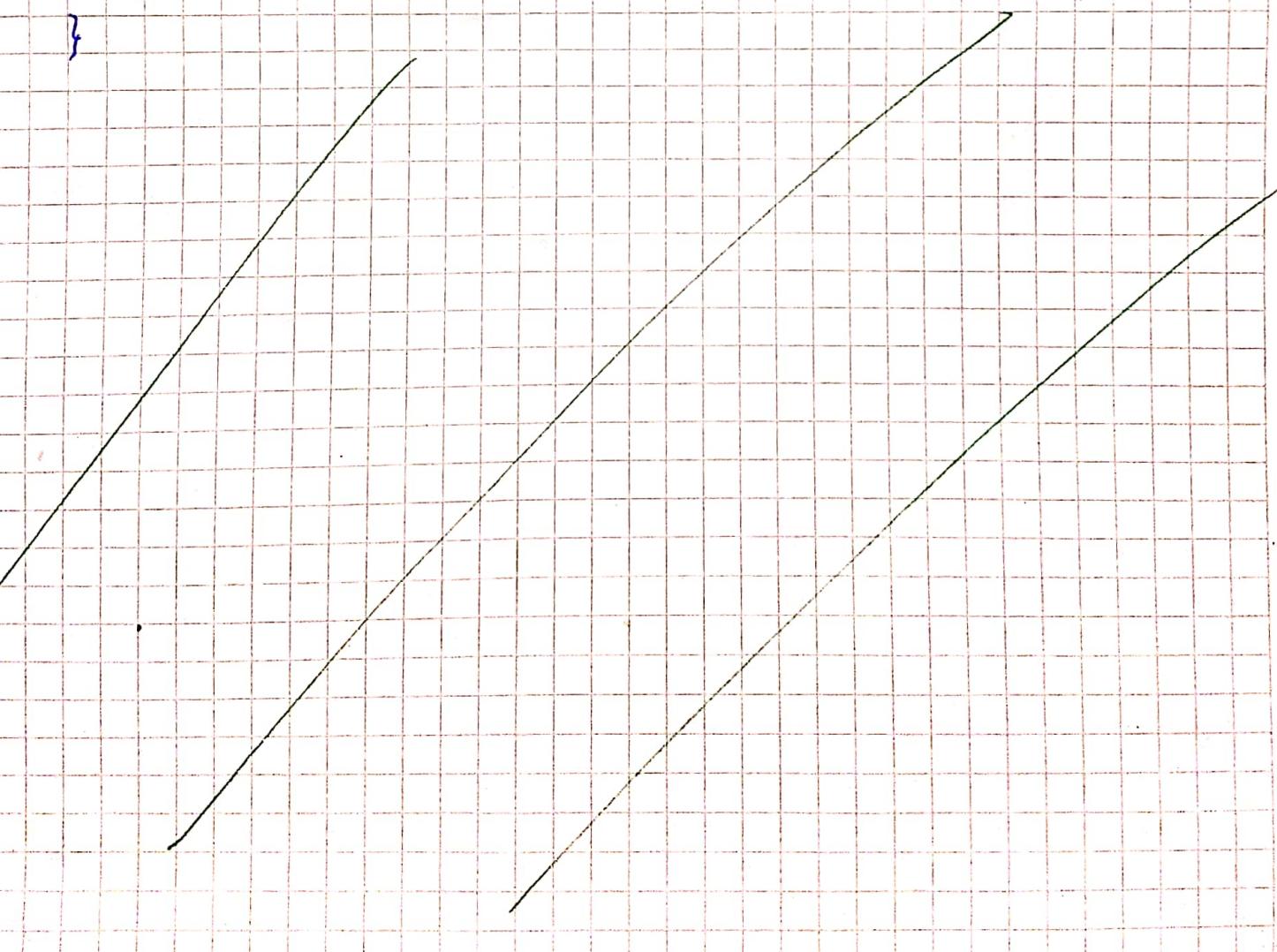
```
pid = fork();  
if (pid == 0)  
    printf ("Soy el hijo");  
else  
    printf ("Soy el padre");
```



La syscall exec() reemplaza el programa que se está ejecutando en el proceso actual por otro, para eso crea un nuevo heap y un nuevo stack y reemplaza el código en .text por el del nuevo programa. Por la naturaleza de esta syscall, la llamada a exec no debería volver.

Ejemplo:

```
int pid = fork()
if (pid == 0) {
    exec("hello")
    printf("Esto no debería ocurrir")
} else {
    printf("Soy el padre")
```



4) B)

No 1 MISMA
SÍNCRONO
MISMO NÚMERO
CÓDIGO

Race-condition: Es cuando más de un thread ejecuta una misma porción de código y el resultado depende de en qué orden se ejecutan las instrucciones.

Ejemplo

Thread 1:

$$x = 2 * x$$

Thread 2:

$$x = x + 1$$



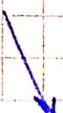
Si al principio $x = 2$ entonces hay dos resultados posibles:

$$x = 5 \text{ ó } x = 6$$

Heisenbug: Es el nombre que se le da al bug que ocurre en un programa debido a la indeterminación que surge a partir de tener threads corriendo de forma concurrente. Es un bug especialmente complejo ya que no está determinado cuándo ocurre porque depende de cómo se planifica la ejecución, por lo que puede generar un error en la ejecución una vez de muchas, y no ocurrir en la mayoría de las ejecuciones del programa.

Ej: Cuando se tiene una race condition que da un resultado esperado en la mayoría de las combinaciones posibles pero no en todas.

Dead-lock: Es un bug que puede ocurrir al utilizar locks para sincronizar la ejecución entre threads. Y ocurre cuando se llega a un estado del que no se puede salir porque hay threads esperando a que se liberen algún lock que nunca se va a liberar.



EJ:

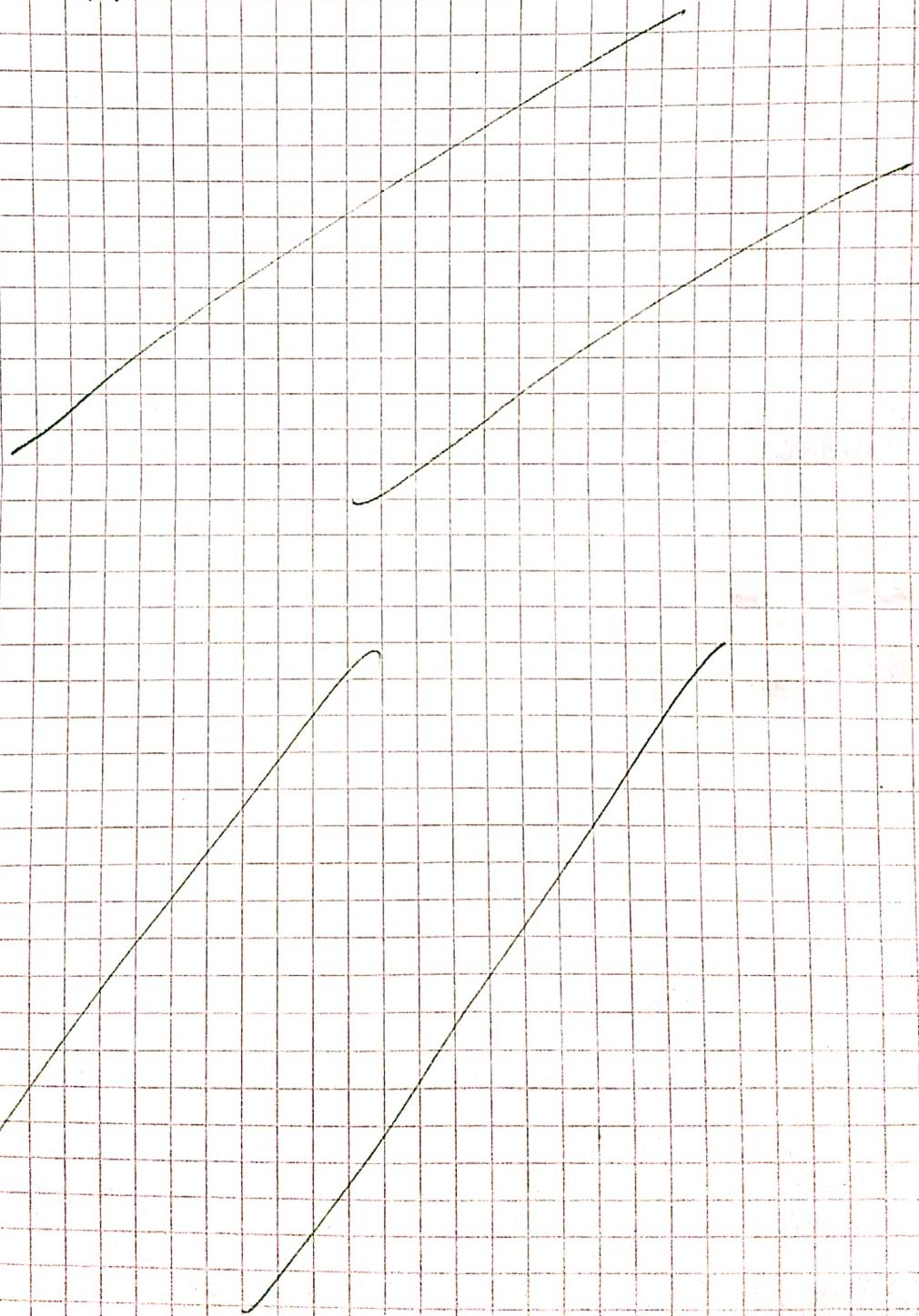
Thread 1 :

lock (L1); --- --- --- \Rightarrow lock (L2);

lock (L2); <--- --- --- \Rightarrow lock (L1);

Thread 2 :

Si el orden de ejecución se da como indican la flecha, estamos en un deadlock ya que ambos se quedan esperando a un Lock que tiene el otro, por lo que ninguno se va a liberar.



4) A)

Un thread es un hilo de ejecución atómico dentro de un proceso.

El proceso lo utiliza para decirle al sistema operativo que ejecute de manera concurrente distintas partes de su código. Es más útil cuando se tiene más de una CPU, ya que en ese caso posiblemente se puedan ejecutar los diferentes threads de forma paralela, acelerando la ejecución en muchos casos.

4) A)

```
void main()
{
    int* x = malloc(1);
    *x = 0;
    t1 = pthread_create(&sumar, x)
    :
    ts = pthread_create(&sumar, x) } 5 veces
    pthread_join(t1, x);
    :
    pthread_join(ts, x);
}

int sumar(int x)
{
    lock(L1);
    if (*x < 1000)
        *x = *x + 7;
    unlock(L1);
    return x;
}
```