Preguntas de parcial

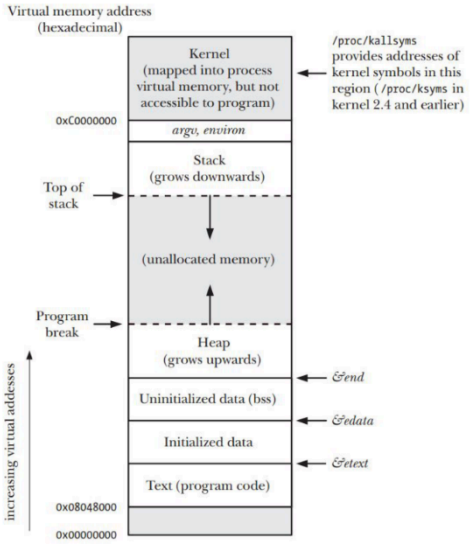
Procesos y Kernel

1) ¿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);* ¿Cómo se pasan los parámetros, dirección de retorno, etc.?
RESPUESTA:
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 y desapilar valores a esa región. Para eso se tiene un registro llamado "stack pointer" el cual siempre tiene la dirección de memoria del tope del stack, por lo que al apilar un valor se incrementa el S.P. en 4, y al desapilar se decrementa 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 pushean los registros "caller saved" (eax, ebx, ecx, edx)
   - Se pushean los parámetros en orden inverso (fd -> num -> len -> buff)
   - Se ejecuta la instrucción call que pushea 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".

2) ¿Qué es el **Address Space**? ¿Qué partes tiene? ¿Para qué sirve? Describa el/los mecanismos para crear procesos en unix, sus syscalls, ejemplifique.
RESPUESTA:
El address space es como ve a la memoria principal el proceso. El proceso ve que toda la memoria le pertenece pudiendo usar cualquier dirección, cuando en realidad esa es la "ilusión" que provee el SO.
El address space se divide en 4 partes:



- .text: Instrucciones del programa.
- .data: Variables globales.
- .heap: Memoria dinámica alocable.
- .stack: Variables locales y trace de llamadas.

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 SO 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 esta llamada continúa la ejecución desde ambos procesos desde la instrucción siguiente a fork() y para diferenciarlos le retorna al padre el processID del hijo, y al hijo le devuelve 0.
  Ejemplo:

```
int 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 setea 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");
}
```

3) Describa las system calls fork(), exec() y wait() (qué hacen, para que se usan, etc.) y su peculiar relación con la creación de procesos en UNIX. Ponga un ejemplo de su funcionamiento en el cual se describe paso a paso lo que está sucediendo.

RESPUESTA:

- **fork**: Se usa para crear un nuevo proceso (hijo) que es una copia exacta del proceso que lo llamó (padre).

  Cada proceso hijo tiene un ID único, un parentId que lo relaciona con su proceso creador y una copia de los mismos archivos abiertos de su padre. Ejemplo:

```
int main(){
int a = 0;
     int f = fork();

if (f == 0){
     a = 5; // modifico el valor de a sólo para el proceso hijo
          printf("Child Process");
     printf("a equals to %d\n", a); // imprime "a equals to 5"
} else {
     printf("Parent Process");
     printf("a equals to %d", a); // imprime "a equals to 0"
}
return 0;
}
```

  El proceso principal crea un proceso hijo que modifica el valor de la variable 5. De esta forma, para cada proceso *a* tiene un valor distinto, de modo que para el proceso hijo equivale a 5 mientras que para el proceso padre equivale a 0. Esto se debe a que cada proceso tiene su propio stack y por ende -si bien en la creación del proceso hijo, *a* equivale a 0- modificar el valor de la variable no afecta al estado del recurso en el proceso padre, que tiene una región de datos distinta a la de su hijo.

- **exec**: Se usa para reemplazar la ejecución del proceso actual por otro. El Sistema Operativo desaloja el proceso anterior y actualiza los valores de contexto, registros de CPU y archivos por los correspondientes al nuevo proceso que solicita ejecutarse.

```
int main(){
char* args[] = {"./CAT", "hola.txt", NULL}
execvp(args[0], args);
}
```

  Se ejecuta en la shell el comando *cat hola.txt*. La ejecución del proceso inicial (el que llamó a *main*) finaliza y puede o no lanzar un error según el valor resultante de execvp.

- **wait**: Se usa para esperar la finalización de un proceso hijo, suspendiendo la ejecución del proceso padre (que es el que lo llama).

```
int main(){
int a = 0;

int fds[2];

      int f = fork();

      pipe(fds);

if (f == 0){
      close(fds[0]);
            a = 5; // modifico el valor de a sólo para el proceso hijo
            printf("Child Process");
      printf("a equals to %d\n", a); // imprime "a equals to 5"
      write(fds[1], a);
            close(fds[1]);
} else {
      waitpid(f, 0, NULL);
      close(fds[1]);
      read(fds[0], a);
      printf("Parent Process");
      printf("a equals to %d", a); // imprime "a equals to 5"
      close(fds[0]);
}
return 0;
}
```

El proceso principal crea un proceso hijo y espera a que termine de ejecutarse para verificar el valor de la variable *a*, la cual tomará el peso que le venga asignado por el pipe intercomunicador entre el proceso hijo y padre. Si no se usase el método *wait* no podríamos asegurar que el valor de *a* equivalga a 5 en la ejecución de las instrucciones del proceso padre.

4) ¿Por qué en necesario el Kernel? De por lo menos 4 motivos.
RESPUESTA:
El kernel es el software que controla los recursos del hardware de una computadora y provee un ambiente bajo el cual los programas pueden ejecutarse. Gracias a la existencia del Kernel los programas son independientes del hadware subyacente.
El rol central de un sistema operativo es la protección.
Algunas de las tareas específicas del Kernel son:

- Planificar la ejecución de las aplicaciones
- Gestionar la Memoria
- Proveer un sistema de archivos
- Creación y finalización de procesos
- Acceder a los dispositivos
- Comunicaciones
- Proveer un API

5) Explicar cómo el Kernel provee aislamiento a los procesos, qué mecanismos y cómo el hardware da soporte a los mismos y en qué momento.
RESPUESTA:

El Kernel tiene distintos métodos por los cuales provee aislamiento a los procesos:

1. **Dual Mode Operation - Modo de operación dual**: Existen 2 modos operacionales utilizados de la CPU:
   I) Modo usuario o user mode: que ejecuta las instrucciones en nombre del usuario
   II) Modo supervisor o modo kernel: que ejecuta instrucciones en nombre del Kernel y estas son instrucciones privilegiadas.

   Esta diferencia entre los modos operacionales de un procesador equivale a un bit en el registro de control del procesador. Este bit permite diferenciar entre tipos e instrucciones: instrucciones privilegiadas o instrucciones normales.
   Dado que este mecanismo es proveído por el hardware, por cada instrucción a ser ejecutada el kernel chequea en qué modo de operación se encuentra.
2. **Privileged Instructions - Instrucciones Privilegiadas**: Es un set de instrucciones específicas que solo podrán ser ejecutadas cuando se esté bajo modo supervisor/Kernel. El bit de modo de operación indica al procesador si la instrucción ejecutada puede ser ejecutada o no.
3. **Memory Protection - Protección de Memoria**: Como el SO y los programas que están siendo ejecutados por el mismo deben de residir ambos en memoria al mismo tiempo, y para que la memoria sea compartida de forma segura, el SO debe poder configurar al hardware de forma tal que cada proceso pueda leer y escribir su propia porción de memoria. Esto lo hace mediante la virtualización de la memoria que le da la "ilusión" al proceso de que tiene todo la memoria física para él cuando en realidad no. Luego será la MMU la que se encargue de traducir esas direcciones virtuales (en el CPU) en direcciones físicas en la memoria.
4. **Timer Interrupts - Interrupciones por temporizador**: Para que el kernel pueda tomar el control de la computadora debe haber algún mecanismo que periódicamente le permita al kernel desalojar al proceso de usuario en ejecución y volver a tomar el control del procesador, y así de toda la máquina. En la actualidad todos los procesadores poseen un mecanismo de hardware llamado hardware counter, el cual puede ser seteado para que luego del transcurso de un determinado tiempo el procesador sea interrumpido. Cada CPU posee su propio timer.

6) ¿Qué mecanismos de hardware son necesarios para que el Kernel del Sistema Operativo proteja a las aplicaciones y los usuarios?

□ Excepciones	■ **Protección de memoria**	□ System calls
		■ **Instrucciones privilegiadas**
□ Excepciones del procesador	■ **Interrupción de tiempo**	□ Todas
□ MMU	□ Interrupciones	

RESPUESTA:

- Excepciones: NO. Son aquellas acciones que no son permitidas, por lo que el SO frena la ejecución de un proceso. Las mismas *se realizan mediante Software* por lo que no cumplen con la consigna.
- **Protección de memoria:** SI. Toda interacción con la memoria fisica se considera una acción privilegiada por lo que requiere de ayuda del hardware
- System Calls: NO. Si bien son necesarias para la protección del sistema, las mismas se encuentran *implementadas a nivel Software*.
- **Instrucciones privilegiadas**: Si. Necesario los bits para definir el nivel de privilegios y con ello el set de instrucciones que tiene permitido usar
- Excepciones del procesador: No. Levantan un flag en caso de generarse excepciones

- MMU: La MMU no es un componente *NECESARIO* ya que es una unidad dedicada a realizar las operaciones referentes a las direcciones de memoria. Sin embargo las mismas podrías ser realizadas por el propio S.O. resultando en un sistema mucho más ineficiente pero igual de funcional.
- **Interrupción de tiempo**: Si. Requieren de un timer propio de cada CPU para limitar la ejecución de los distintos procesos
- Interrupciones: NO. Dado que las interrupciones podrían ser *generadas mediante Software*, no cumplen con la consigna.

7) Explicar detalladamente 2 formas de pasar de modo usuario a modo Kernel, y 2 formas de pasar de modo Kernel a modo usuario.
RESPUESTA:

- **De modo usuario a modo Kernel**:
  - Interrupciones: Una interrupción es una señal asincrónica enviada hacia el procesador de que algún evento externo ha sucedido y pueda requerir de la atención del mismo.
  - Excepciones del procesador: un evento de hardware causado por un programa de usuario. Algunos ejemplos de excepciones son:
    - Acceder fuera de la memoria del proceso
    - Intentar ejecutar una instrucción privilegiada en modo usuario
    - Intentar escribir en memoria de solo lectura
    - Dividir por 0
  - Mediante la ejecución de system calls: Las System Calls son funciones que permiten a los procesos de usuario pedirle al kernel que realice operacione**s** en su nombre. Una System Call es cualquier función que el kernel expone que puede ser utilizada por un proceso a nivel usuario.

- **De modo Kernel a modo usuario**:
  - Un nuevo proceso: Cuando se inicia un nuevo proceso el Kernel copia el programa en la memoria, setea el contador de programa apuntando a la primera instrucción del proceso, setea el stack pointer a la base del stack de usuario y switchea a modo usuario.
  - Continuar después de una interrupción del procesador o de una system call: Una vez que el Kernel terminó de manejar el pedido, este continúa con la ejecución de proceso interrumpido mediante la restauración de todos los registros y cambiando el modo a nivel usuario.
  - Cambio entre diferentes procesos: En algunos casos puede pasar que el Kernel decida ejecutar otro proceso que no sea el que se estaba ejecutando, en este caso el Kernel carga el estado del proceso nuevo a través de la PCB y cambia a modo usuario.

8) ¿Qué es el modo dual? ¿Para qué sirve? ¿Qué mecanismos se necesitan para implementarlo? ¿Qué es una system call?
RESPUESTA:

El modo dual es un mecanismo que proveen los procesadores para poder ejecutar instrucciones en nombre del usuario o en nombre del kernel, es decir poder correr instrucciones privilegiadas a las que el usuario no tiene ni debe tener acceso, es una forma de poder aislar más el poder del usuario. Entonces sirve para proteger la memoria, los puertos de I-O y la posibilidad de ejecutar ciertas instrucciones, es decir con este bit de modo dual, cada modo o ring pasaría a tener su set de instrucciones. En principio a nivel hardware se necesita poder tener este bit extra para poder diferenciar cuando son instrucciones de usuario y cuando son instrucciones de kernel, pero esto no es suficiente dado que por ejemplo una vez que el kernel le da el control a un proceso, si no se chequea de alguna manera si el proceso está vivo, este podría tomar control de la máquina por tiempo indefinido, por eso se requiere

de interrupciones por temporizador para poder volver a tomar el control de la máquina periódicamente y que el kernel se asegure de que está todo ok, cuando esto ocurre el proceso en modo usuario le transfiere el control al kernel ejecutándose en modo kernel. A su vez se necesita proteger a la memoria, dado que el kernel mismo está en memoria, el proceso está en memoria y probablemente simultáneamente haya más procesos en ejecución. Debido a esto, para que la memoria sea compartida de forma segura, el sistema operativo debe poder configurar el hardware de forma tal que cada proceso pueda leer y escribir su propia porción de memoria.
Las system calls son funciones que permiten a los procesos de usuario pedirle al kernel que realice operaciones en su nombre. Una system call es cualquier función que el kernel expone que puede ser utilizada por un proceso a nivel usuario. Básicamente es una API de funciones que expone el kernel a los procesos de usuario, para que el proceso pueda pedirle al kernel realizar operaciones en su nombre, por ejemplo operaciones de file system, comunicaciones, calls de procesos, etc.

9) ¿Cuáles de los siguientes están contenidos en el archivo binario de un programa?

☒ **Code segment**	☒ **Stack segment**	□ Heap segment
□ Variables locales	☒ **Bibliotecas**	☒ **Data segment**

10) ¿Cuál de las siguientes opciones están contenidos en el archivo binario ELF de un programa?

Heap Segment	Si	No
Code Segment	**Si**	~~No~~
Data Segment	**Si**	~~No~~
Stack Segment	Si	No
Símbolos y tabla de realocación	**Si**	~~No~~
Entry point	**Si**	~~No~~
Bibliotecas	**Si**	~~No~~

un

El ciclo desde

11) Describir el ciclo de vida de una syscall con ejemplo.
RESPUESTA:
de vida de una syscall comienza con la llamada C a la función que actúa como wrapper de la syscall real. Este wrapper en C, se encarga de preparar todo lo necesario para realizar el llamado de forma correcta, es decir, pasa los argumentos que recibe del stack a los registros esperados, y copia al registro %eax el ID de la syscall que se quiere llamar (todas entran del mismo modo).
Una vez hecho esto, el wrapper ejecuta la interrupción 0x80, con la cual se pasa a kernel mode. A partir de esta llamada está actuando el kernel, que revisa cuál syscall se llamó y atiende lo pedido. Una vez hecho esto, prepara el resultado a devolver y pasa a user mode, terminando la parte en que se involucraba el kernel.
Llevándolo a un ejemplo, podría verse así:
Modo usuario:

1. Llamada a fork()
2. En el wrapper de fork()
3. Poner en el registro %eax la ID de la syscall fork
4. Ejecutar interrupción 0x80 (int 0x80)

Modo kernel:

5. Recibir pedido
6. Determinar en base al ID en %eax cuál syscall ejecutar
7. Realizar fork (copia de contexto, preparar retorno, etc)
8. Volver al modo usuario

Modo Usuario:

9. Actuar en base a lo que devolvió fork

12) Describa qué es un proceso: qué abstrae, cómo lo hace, cuál es su estructura. Además explique el mecanismo por el cual el proceso cree tener la memoria completa de la máquina cuando en realidad solo tiene lo necesario para su funcionamiento.
RESPUESTA:
Un proceso es la ejecución de un programa de aplicación con derechos restringidos; el proceso es la abstracción que provee el Kernel del sistema operativo para la ejecución protegida. Un proceso incluye:

- Los Archivos abiertos
- las señales (signals) pendientes
- Datos internos del kernel
- El estado completo del procesador
- Un espacio de direcciones de memoria
- **Uno o más hilos de Ejecución. Cada thread contiene:**
    - Un único contador de programa
    - Un Stack
    - Un Conjunto de Registros
    - Una sección de datos globales

El mecanismo por el cual el proceso cree tener toda la memoria es la virtualización de memoria. La virtualización de memoria le hace creer al proceso que este tiene toda la memoria disponible para ser reservada y usada como si este estuviera siendo ejecutado sólo en la computadora (*ilusión*). Todos los procesos en Linux están divididos en 4 segmentos:

- **Text**: Instrucciones del Programa.
- **Data**: Variables Globales (extern o static en C)

Espacio de direcciones del proceso

- **Heap**: Memoria Dinámica Alocable
- **Stack**: Variable Locales y trace de llamadas

El hardware se encarga de la traducción de la dirección virtual (emitida por la CPU) en una dirección física (en la memoria). Este mapeo se realiza por la MMU (Memory managment Unit).

13) En un sistema operativo, cada proceso tiene su propio:

- **Address Space y Variables globales**
- **Señales y manejadores de señales**
- **Archivos abiertos**
- **Todas las anteriores**

14) Verdadero o Falso:
Un proceso:

El contexto de un proceso tiene una componente user-level y system-level	**V**	~~F~~
Solo el Kernel puede ampliar la memoria del heap	**V**	~~F~~
Cada vez que se crea un proceso se reserva toda la memoria direccionable para ese proceso	~~V~~	**F**
La memoria de un proceso se controla con las system calls malloc() y free()	~~V~~	**F**
Se ejecuta mejor en un sistema de ejecución directa	~~V~~	**F**

15) ¿Qué sucede cuando se ejecuta este código en C?

```
int main(void) {
   printff("PID: %d\n", getpid());
   fork();
   main();
}
```

Explique detalladamente qué sucede en la salida, y además internamente

RESPUESTA:

La primera vez que se ejecute va a imprimir el PID del proceso actual que se está ejecutando, luego al hacer un fork() lo que sucede es que se crea una copia del proceso actual, a esta copia se le va a llamar proceso hijo. En este punto vamos a tener 2 procesos: el proceso padre con PID P y el proceso hijo con PID C1. Se van a seguir ejecutando ambos procesos, pero es el sistema operativo quien decide el orden. Luego, se invoca a main() nuevamente, o sea que se va a imprimir el PID actual (que puede ser P o C1) y se va a volver a hacer un fork(). Entonces, ahora cada proceso va a copiarse nuevamente, obteniendo dos nuevos procesos (y ahora habrá un total de 4 procesos), un hijo de P (con PID C2) y un hijo de C1 (con PID CC1). Al tratarse de un programa recursivo donde en cada invocación se llama a fork(), cuando se haya ejecutado 3 veces, va a quedar algo del estilo:

Posible salida:

```
PID: P
PID: P
PID: C1
PID: P
PID: C1
PID: CC1
PID: C2
```

Esquema interno:

```
P (proceso padre inicial)
+-- C1 (primer hijo - 1er fork)
|       +-- CC1 (2do fork)
|       |       +-- CCC1 (3er fork)
|       +-- CC2 (3er fork)
+-- C2 (2do fork)
|       +-- CC3 (3er fork)
+-- C3 (3er fork)
```

Es importante notar que cada proceso padre e hijo van a continuar la ejecución inmediatamente después del fork, que en este caso particular coincide con volver a llamar a main().

Esto tiene que tener un límite, ya que los recursos no son infinitos; el sistema operativo establece un límite para la cantidad de procesos, un caso así va a ser detectado y se va a cortar la ejecución del programa deteniendo todos los procesos involucrados.

Memoria virtual

1) ¿Qué es la memoria virtual? ¿Qué mecanismos conoce, describa los tres que a usted le parezcan los más relevantes?

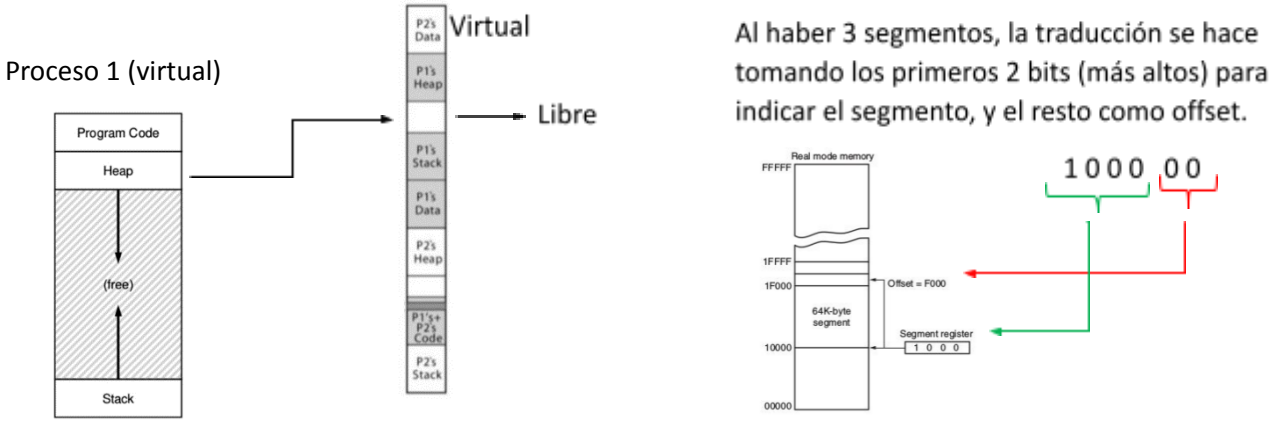
RESPUESTA:

Es la abstracción de la memoria física que el SO le provee a un programa en ejecución. Gracias a la memoria virtual se le puede dar la "ilusión" al proceso de que tiene toda la memoria disponible para él.

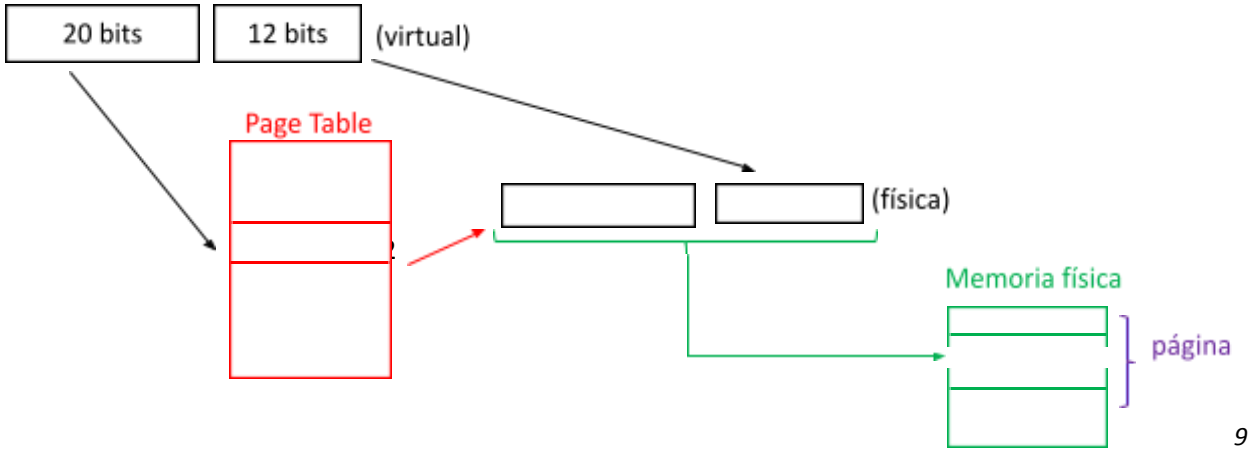
Un buen mecanismo de virtualización de memoria debe lograr:

- **Transparencia**: El SO debería implementar la virtualización de memoria de forma tal que sea *invisible* al programa que se está ejecutando; es más el programa debe comportarse como si él estuviera alojado en su propia área de memoria física privada.
- **Eficiencia**: El SO debe esforzarse para hacer que la virtualización sea lo más *eficiente* posible en términos de *tiempo* (no hacer que los programas corran más lentos) *y espacio* (no usar demasiada memoria para las estructuras necesarias para soportar la virtualización).
- **Protección**: El SO tiene que asegurarse de *proteger a los procesos unos de otros como también proteger al sistema operativo de los procesos.*

El primer mecanismo posible es el de la 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 SO le indica los límites de los segmentos de memoria del proceso actual (code, heap, stack) marcando donde empieza 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.

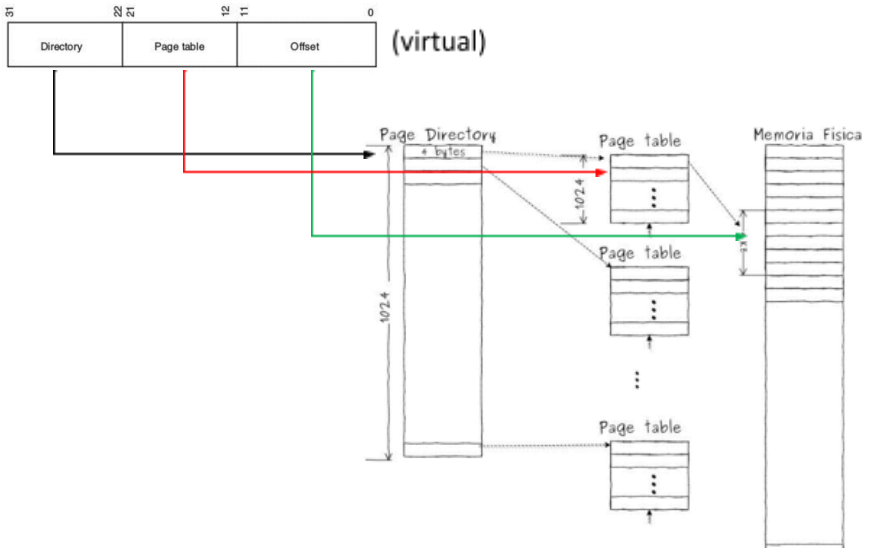


El segundo mecanismo es el de la memoria paginada de 1 nivel. En este mecanismo se divide la memoria en bloques fijos a diferencia de bloques de tamaño variable. Estos bloques suelen ser de 4 KB. Para este mecanismo se le debe de indicar a la MMU en qué dirección física se encuentra una tabla conocida 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 4 KB 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 Page Table 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 los bits para indicar la entrada en la page directory y los otros 10 para la entrada a la Page Table.



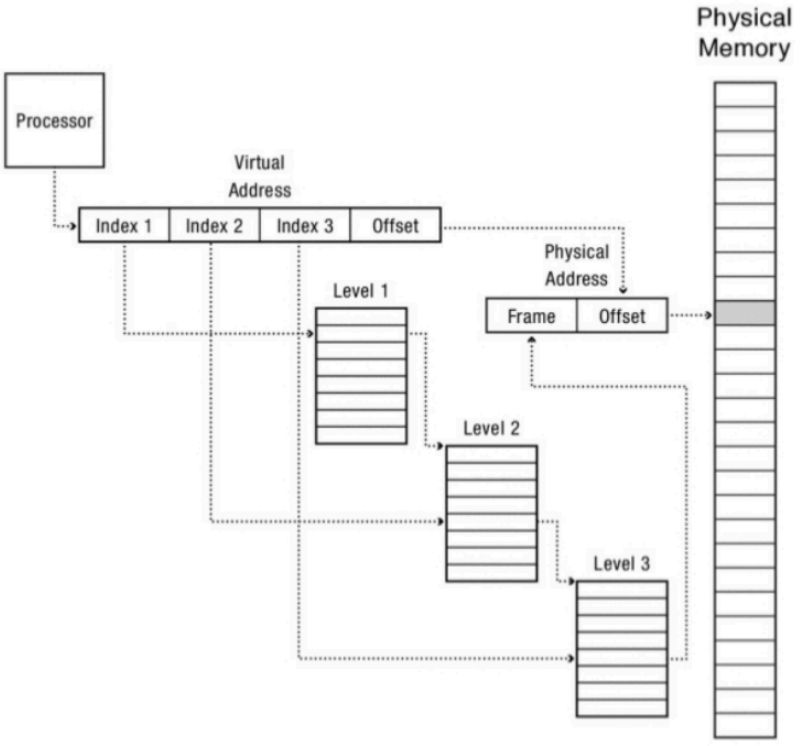
De esta manera, el Page Directory únicamente va a tener 2^{10} entradas y la Page Table también 2^{10}, con la diferencia de que este último sólo va a existir si hay una dirección mapeada que requiera su existencia.

2) 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.

RESPUESTA:



Va a haber 2^{32} direcciones de memoria posible, 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.

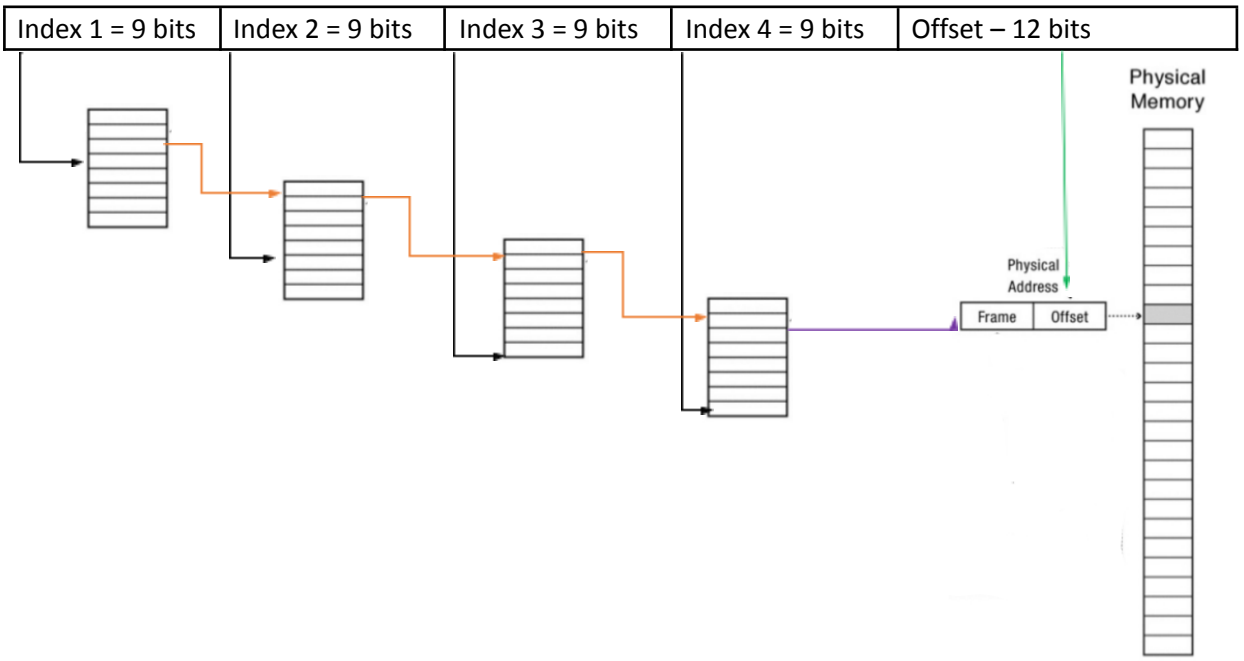
En realidad se va a poder siempre que los registros tengan tamaño suficiente para poder direccionar a las tablas del próximo nivel.

3) Dar un sistema de paginación de 4 niveles de indirección y 1 virtual address de 48 bits en los cuales 12 son el offset y los restantes grupos de 9 bits son los page directory. Decir cuál es el tamaño más grande que puede tomar el proceso (la respuesta no debe ser simplemente 9^{48}).

RESPUESTA:

Como tengo 4 niveles de indirección, la virtual address será:

(48-12=36 => 36/4 = 9)



4) Supongamos que tienes un sistema operativo de 32 bits que utiliza un esquema de direccionamiento virtual paginado con una tabla de páginas de 2 niveles. El tamaño de página es de 4 KB (kilobytes) y el tamaño de dirección virtual y física es de 32 bits.

a. ¿Cuántos bits se necesitan para representar el desplazamiento dentro de una página?

□ 10 bits ▯ **12 bits** □ 8 bits □ 20 bits

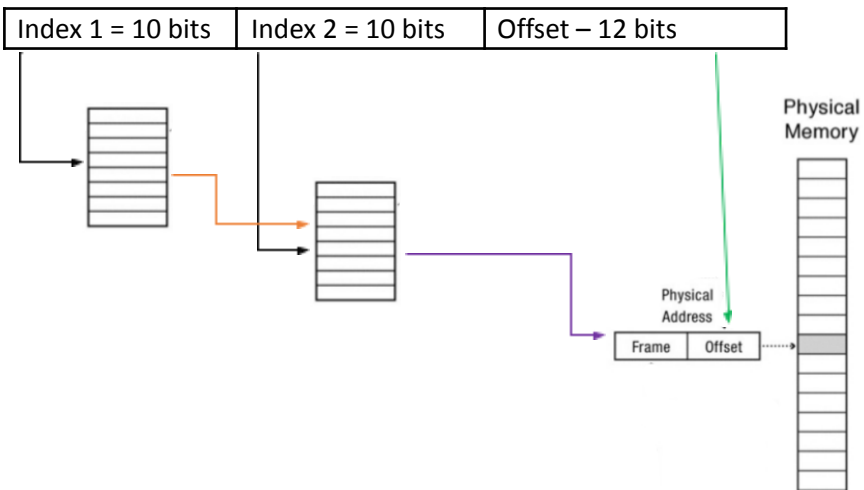
b. ¿Cuántas entradas tiene la tabla de páginas de nivel 1?

□ 2^11 ▯ **2^10** □ 2^12 □ 2^8

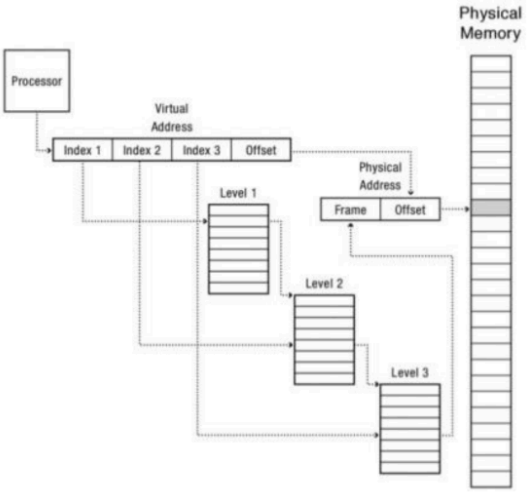
c. ¿A qué tipo de direcciones apuntan las entradas de nivel 2?

▯ **2^32** □ 2^10 □ 2^20 □ 2^12

RESPUESTA:



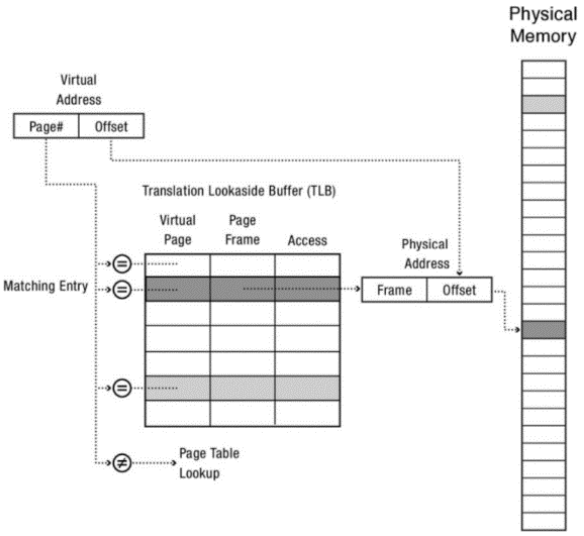
5) Explique el método de administración de memoria de paginación con 3 niveles de indirección.



RESPUESTA:

Este método consiste en dividir el mapeo de memoria en 3 niveles, para esto se parte la virtual address en 4 partes, la entrada de la primera tabla, la entrada de la segunda tabla, la entrada de la tercera tabla y por último el offset en la página de memoria física. Luego el procedimiento de traducción sería, con el index 1 buscar la tabla de paginación de segundo nivel y con el index 2 buscar la tabla de paginación del tercer nivel y utilizando el index 3 finalmente obtener la PPN para concatenarla al offset y encontrar la physical address.

6) Explicar un sistema de paginación con TLB y paginación de 3 niveles. Contar miss, hit, translaciones y acceso a memoria.



7) Seleccionar las correctas:

- □ El address translation mejora la MMU haciéndola más rapida
- ☒ **Para mejorar el address translation, se utiliza un mecanismo de hardware (TLB).**
- ☒ **La TLB es un address translatrion caché.**
- □ Contiene los registros usados para la paginación
- ☒ **Uno de los problemas del address translation reside en la velocidad de la traducción.**

8) El esquema de virtualización de memoria de x86:

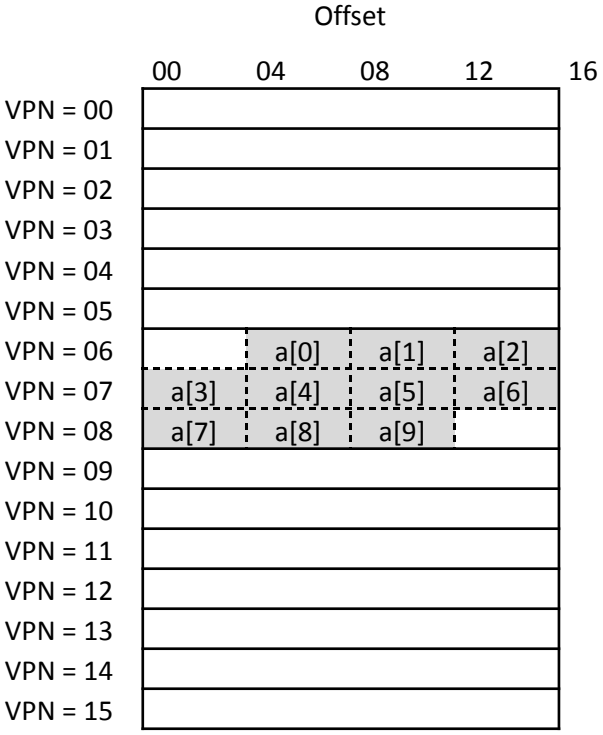
- ☒ **Admite segmentación**
- ☒ **Paginación segmentada**
- □ Paginación de 3 niveles de indirección
- ☒ **Paginación de 2 niveles de indirección**
- □ Direcciona hasta 2 Tb, con frames de 4 kb
- □ Direcciona hasta 4 Tb, con frames de 2 kb
- ☒ **Divide la memoria física en frames de 4096 bytes**
- □ Ninguna

9) En la arquitectura x64, una dirección de memoria posee 64 bits, pero la estructura de una VA es: 9 bits para el page directory, 9 bits para la page table 1, 9 bits para la page table 2, 9 bits para la page table 3 y finalmente 12 bits para el offset. La máxima cantidad de memoria direccionable es:

- □ 2^32 bytes
- □ 2^64 bytes
- □ 2^36 bytes
- ▯ **2^48 bytes**

Page directory	Page table 1	Page table 2	Page table 3	Offset
2^9	* 2^9	* 2^9	* 2^9	* 2^12 = 2^48

10) Dado el siguiente esquema explique cómo se realizan las traducciones recorriendo el arreglo en un modelo de memoria virtual con TLB y paginación de 2 niveles. En el mismo esquema decir cuántos miss, hit, accesos a memoria y traducciones hay.



Obs:
VPN = Virtual Page Number
Así que este espacio de direcciones está formado por 16 páginas y cada página posee 16 bytes.

a[0] -> miss a[1] -> hit a[2] -> hit
a[3] -> miss a[4] -> hit a[5] -> hit a[6] -> hit
a[7] -> miss a[8] -> hit a[9] -> hit

Responda:

- ~~□ En las traducciones hay 3 hits y 7 miss en la TLB~~
- □ Hay 10 accesos a memoria
- ▯ En las traducciones hay 7 hits y 3 miss en la TLB
- ▯ Hay 3 traducciones completas de VA a PA
- □ Hay 3 accesos a memoria en total
- ~~□ Hay 10 traducciones completas de VA a PA~~

11) Suponga que virtual address con las siguientes características:

- 4 bit para el segment number
- 12 bits para el page number
- 16 bits para el offset

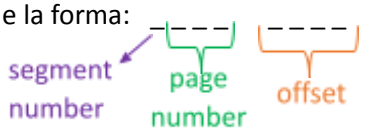
Segment table	Page Table A	Page Table B
0 Page B	0 CAFÉ	0 F000

1 Page A	1 DEAD	1 D8BF
X invalid	2 BEEF	2 3333
	3 BA11	X INVALID

Traducir las siguientes direcciones virtuales a físicas: 0000 0000, 2002 2002, 1002 2002, 0001 5555.

RESPUESTA:

Las direcciones virtuales serán de la forma:



a. *0000 0000*
Segment number: 0 -> Page B
Page number: 0 -> F000
Offset: 0000
VA: 0000 0000 ----- PA: F000 0000

b. *2002 2002*
Segment number: 2 -> invalid
VA: 2002 2002 ----- PA: Invalid

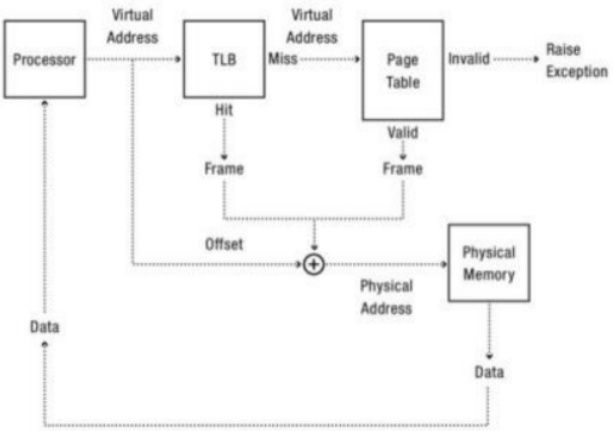
c. *1002 2002*
Segment number: 1 -> Page A
Page number: 2 -> BEEF
Offset: 2002
VA: 1002 2002 ----- PA: BEEF 2002

d. *0001 5555*
Segment number: 0 -> Page B
Page number: 1 -> D8BF
Offset: 5555
VA: 0001 5555 ----- PA: D8BF 5555

12) ¿Qué es la TLB? ¿Para qué sirve? Ponga un ejemplo de una TLB de 2 niveles

RESPUESTA:

TLB (Translation Lookaside Buffer) es un mecanismo usado por la MMU (memory management unit) para cachear las traducciones de VPN a PPN para optimizar las búsquedas de PPN dado que este suele ser un proceso costoso, por ejemplo en una memoria paginada por 3 niveles se tienen que hacer 3 accesos a memoria para determinar la PPN. La TLB es una memoria muy rápida SRAM que aprovechando el principio de localidad espacial y temporal provee eficiencia a la hora del proceso de address translation.



Si se tiene una TLB de 2 niveles, al haber un miss en el primer nivel, se va a buscar en el segundo nivel. En caso de haber un hit el dato de traducción es devuelto y agregado al primer nivel, en caso de miss se va a recurrir a la penalidad de realizar la traducción de la VPN.

13) Verdadero o Falso: ¿Qué es la TLB?

Es la responsable de realizar la traducción de memoria virtual a memoria física	~~V~~	**F**	Falso: La MMU se encarga de la traducción
La que mejora el rendimiento del sistema de traducción de memoria dentro de la cache	~~V~~	**F**	Falso: Es un chip cerca del procesador
La que proporciona un mecanismo de hardware que acelera las traducciones de las direcciones de memoria	**V**	~~F~~	

Scheduling

1) ¿Qué es un **context switch**? En un context switch cuáles de las siguientes cosas no/si deben ser guardadas y por qué:

a.	Registros de propósito general	**Si**	~~No~~
b.	Translation Lookaside Buffer (TLB)	~~Si~~	**No**
c.	Program counter	**Si**	~~No~~
d.	Page Directory entry	**Si**	~~No~~
e.	PCB entry	**Si**	~~No~~

RESPUESTA:
Un context switch es lo que hace un SO 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 al 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.
¿Debe ser guardado?

- Registros de propósito general: **Sí**, porque son utilizados internamente en el programa que el proceso está ejecutando y pueden ser modificados por otro proceso.
- Translation lookaside buffer (TLB): **No**, porque es interno de la MMU.
- Program counter: **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.
- Page Directory Entry: **Sí**, porque cada proceso tiene su espacio de direcciones y la información para la traducción está allí.
- Pcb entry: **Sí**, porque es donde se guarda todo lo anterior.

2) Supongamos que se tiene un sistema operativo con un planificador de procesos basado en MLFQ con tres niveles de prioridad. Cada nivel de prioridad tiene una cuota de tiempo asignada. Las prioridades más altas tienen cuotas de tiempo más cortas que las prioridades más bajas. Se tienen los siguientes procesos, junto con su tiempo de llegada (TL) y su tiempo de ejecución (TE):

Proceso	TL	TE
P1	0	20
P2	1	15
P3	2	8
P4	3	2
P5	4	14

Las cuotas de tiempo asignadas para cada nivel de prioridad son:
Nivel 1: 2 unidades de tiempo.
Nivel 2: 4 unidades de tiempo.
Nivel 3: 6 unidades de tiempo.
Determinar el orden de ejecución de los procesos.

RESPUESTA:
En MLFQ cuando un proceso llega, se le asigna la máxima prioridad (en este caso 1) y cuando cumple con la cantidad de unidades de tiempo de ese nivel, en caso de no hacer completado la tarea, se le baja en la cola de prioridades.

3) Se tiene el siguiente esquema de procesos:

Proceso	Duración
P1	73
P2	23
P3	61
P4	43
P5	37
P6	29

Q1 = RR (quantum=5)
Q2 = RR (quantum=10)
Q3 = FIFO

El time arrival de los procesos es 0 para todo, no hay Boost time, se pide calcular utilizando MLFC:

- Completion time: tiempo t en el que se completó el proceso
- Turnaround time (CT - AT)
- Waiting time (TAT - duración)

Muestre la evolución del sistema de colas.

RESPUESTA:

Q1	P1(73), P2(23), P3(61), P4(43), P5(37), P6(29)
Q2	
Q3	

Como Q1 se ejecuta con Round Robin, vemos cuál es el proceso de mayor prioridad para ser ejecutado, es decir, primero ejecuta P6. Ahora bien, como el mismo tiene una duración mayor al time slice definido para la cola en cuestión, según las reglas de MLFQ baja la prioridad una unidad (y por ende pasa a Q2):

Q1	P1(73), P2(23), P3(61), P4(43), P5(37)
Q2	P6(24)
Q3	

Repetimos hasta que todos los procesos estén en Q2 pues ninguno tiene un tiempo de duración menor al time slice:

Q1	
Q2	P1(68), P2(18), P3(56), P4(38), P5(32), P6(24)
Q3	

Como Q1 quedó vacía, la próxima cola que será seleccionada para ejecutar procesos será Q2. Nuevamente, ejecutamos RR aunque esta vez con un time slice de 10 y repetimos los mismos pasos:

Q1	
Q2	
Q3	P1(58), P2(8), P3(46), P4(28), P5(22), P6(14)

Como Q2 quedó vacía, la próxima cola a ejecutar es Q3. Esta tiene un método de ejecución de tipo FIFO de modo que el primer proceso en llegar es el último en ejecutar. De esta forma, los procesos se ejecutarán en el siguiente orden, completando su ejecución: P6, P5, P4, P3, P2, P1.

Finalmente, calculemos las métricas:

➔ **Completion time:**
- P6 -> 5 + (5 * 5) + 10 + (10 * 5) + 14 = 104
- P5 -> 104 + 22 = 126
- P4 -> 126 + 28 = 154
- P3 -> 154 + 46 = 200
- P2 -> 200 + 8 = 208
- P1 -> 208 + 58 = 266

➔ **Turn Around Time** (como el time arrival es 0, equivale a CT):
- P6 = 104
- P5 = 126
- P4 = 154
- P3 = 200
- P2 = 208
- P1 = 266

➔ **Waiting Time:**
- P6 = 104 - 29 = 75
- P5 = 126 - 37 = 89
- P4 = 154 - 43 = 111

◆ P3 = 200 - 61 = 139 ◆ P2 = 208 - 23 = 185 ◆ P1 = 266 - 73 = 193

4) 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. ¿Cuántas veces se interrumpe y en qué cola termina su ejecución?

RESPUESTA:

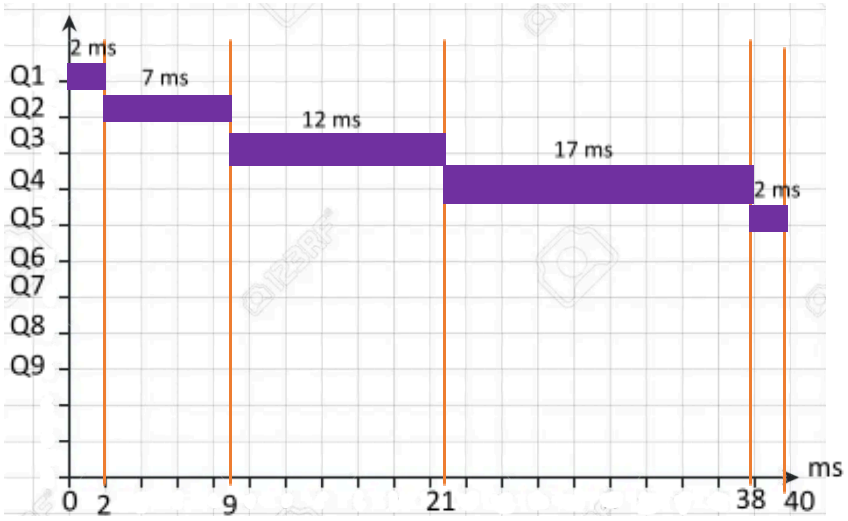
La política MLFQ consiste en tener varias colas (usualmente 8) donde cada una representa un nivel de prioridad de ejecución. Lo que va a hacer el scheduler es ejecutar utilizando Round-Robin las tareas que se encuentran en la cola de mayor prioridad, y hasta que no terminen, no se pasa a la siguiente 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 cambiar de acuerdo a su comportamiento, y para eso MLFQ tiene reglas:

I) Todas las tareas están en la prioridad más alta al inicio.
II) Si una tarea usa tiempo equivalente a un time slice de la cola donde se encuentra su prioridad baja (en 1 cola).
III) Mientras no haya usado 1 time slice su prioridad se mantiene.
IV) 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 tiene 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 4 veces.

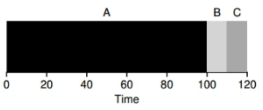
5) Describa una política de scheduling preentive y otra no.

RESPUESTA:

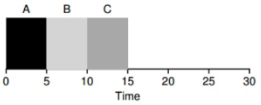
La política de Round-Robin es preentive, mientras que First In, First out no lo es.

En la política de **FIFO**, cada proceso se resuelve en el orden que llegaron. Hasta que no termine el actual, el siguiente no tendrá su turno:

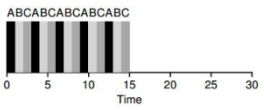
Esto causa el convoy effect.



OBS: **Shortest Job First** también es no preentive. En ese caso se modifica FIFO para que se ejecute el proceso de duración mínima primero, luego el siguiente mínimo y así sucesivamente.



Por otro lado, la política de **RR** es bastante simple, se ejecuta un proceso por un período determinado de tiempo (slice) y transcurrido el período se pasa a otro proceso, y así sucesivamente cambiando de proceso en la cola de ejecución. Lo



importante de RR es la elección de un buen time slice, se dice que el time slice tiene que amortizar el cambio de contexto sin hacer que esto resulte en que el sistema no responda más.

6) Marque como verdadero o falso:

Round Robin es una excelente política de scheduling en un entorno que necesita buen response time	**V**	~~F~~
SJF es una política de scheduling más eficiente que Shortest Job to Completion	~~V~~	**F**
Round Robin utiliza un quantum de tiempo para que todos los procesos puedan utilizar el procesador pero no mejora sustancialmente el turnabout time	**V**	~~F~~

Falso: SJTC es más eficiente que SJF

7) Marque como verdadero o falso:

a	Round Robin tiene un buen response time	**V**	~~F~~
b	El efecto convoy sucede en la política SJF	~~V~~	**F**
c	El time slice debe ser elegido teniendo en cuenta el tiempo de context switch	**V**	~~F~~
d	Los schedulers no-preemptivos son preferibles a los preemptivos	~~V~~	**F**
e	Time arrival está asociado a los procesos interactivos, mientras el time response se asocia al tiempo de llegada de los procesos	~~V~~	**F**
f	MLFQ intenta mejorar la interacción entre procesos mediante la retroalimentación de los mismos en el sistema de colas	**V**	~~F~~

b) FALSO: El efecto convoy sucede en la política *FIFO*

d) FALSO: En las políticas preemptives se puede desalojar, y justamente eso es lo que queremos de un scheduler: que pueda desalojar un proceso para que no ocupe todo el CPU

e) FALSO: Es al revés: Time arrival está asociado al tiempo de llegada de los procesos, mientras que el time response está asociado a los procesos interactivos

File System

1) Explique qué es un sistema de archivos y cuáles son sus 2 principales componentes

RESPUESTA:

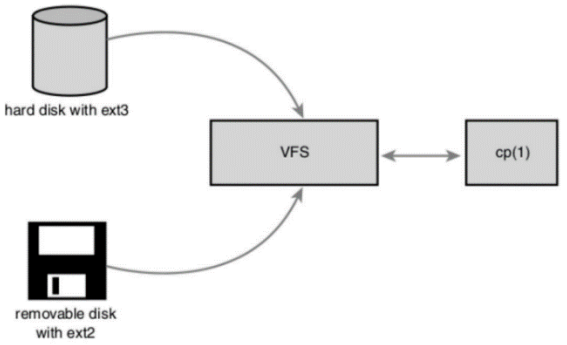
Un sistema de archivos es una abstracción para la persistencia de datos. Busca simplificar el uso de estos la escritura y lectura de estos datos. Se pueden tener varios file systems para que cada uno se especialice en una determinada área. Para que múltiples FS puedan coexistir todos tienen que implementar la API de FS del determinado sistema operativo (VFS). Esta API logra que el usuario pueda utilizar distintos file systems sin enterarse que lo está haciendo, y permite que se puedan modificar los file systems sin tener que cambiar la forma en que se usan.
Las dos principales partes de un file system son:

- Los **archivos**, que son los que tienen la data que el usuario quiere guardar, así como también algo de metadata, como la longitud, de qué manera se abrió, etc. Además de tener un identificador interno se les pone un nombre para que el usuario pueda identificarlos
- Los **directorios** son la forma en la que se organizan los archivos, esencialmente son listas que contienen los nombres de otros archivos o directorios para poder acceder a ellos

2) ¿Qué es el VFS, cuáles son sus componentes y cómo se relacionan?

RESPUESTA:

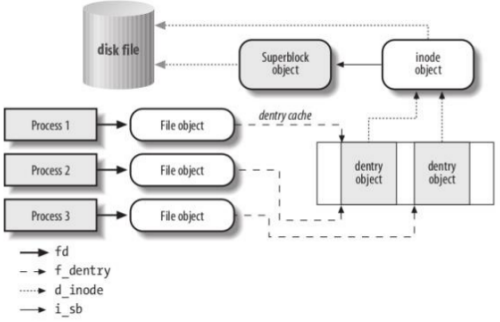
El VFS (virtual File System) es el subsistema del kernel que implementa las interfaces que tienen que ver con los archivos y sistema de archivos provistos a los programas corriendo en modo usuario. Todos los sistemas de archivos deben basarse en VFS para:



- Coexistir
- Inter-operar

Esto habilita a los programas a utilizar las system calls de UNIX para leer y escribir en diferentes sistemas de archivos y diferentes medios. Ejemplo open(), read() y write() funcionan sin que estas necesiten tener en cuenta el hardware subyacente.
(comando copy se comunica con VFS para realizar la copia del disco removible al HDD)
El VFS proporciona la capa de abstracción de archivos a los programas en user mode. Esta capa trabaja mediante la definición de interfaces básicas y de estructuras que cualquier sistema de archivos soporta.

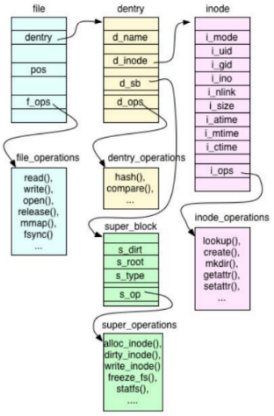
Las componentes del VFS son:



- **Super bloque**, que representa un sistema de archivos. Existe en disco y en memoria. En disco provee información al kernel de la estructura del file system. En memoria provee la información necesaria y los estados para manipular al file system montado.
- **i-node**, que representa un determinado archivo dentro del disco. Un inode puede referenciar a un archivo, un directorio o un link simbólico a otro objeto. El inode consiste de data y operaciones que describen sus contenidos y las operaciones que pueden realizarse en él (ej. open, read, write).

- **dentry**, que representa una entrada de directorio, que es un componente simple de un path. El file system tiene un dentry root, el único que no va a tener padre. Si un archivo tiene el path "/home/user/name", se van a crear cuatro dentries: "/", "/home", "/user", "/name". Vive en memoria, no en disco.
- **file**, que representa a un archivo asociado a un determinado proceso.

Los procesos abren objetos de tipo file. Para acceder al archivo realmente guardado en disco, tienen que entrar en la dentry cache, que lo que hace es darte la ruta que termina en un objeto inodo (el dentry es un archivo que relaciona el inodo con el nombre del archivo). Este objeto inodo es parte del superbloque.
Observación, un directorio es tratado como un archivo normal, no hay un objeto específico para los directorios. En UNIX, son archivos normales que listan los archivos contenidos en ellos.



Luego existe un conjunto de operaciones:

- **super_operations**, métodos que aplica el kernel sobre un determinado sistema de archivos, por ejemplo write_inode() o sync_fs().
- **inode_operations**, métodos que aplica el kernel sobre un archivo determinado, por ejemplo create() o link().
- **dentry_operations**, métodos que se aplican directamente por el kernel a un determinado directory entry, como por ejemplo, d_compare() y d_delete().
- **filte_operations()**, métodos que el kernel aplica directamente sobre un archivo abierto por un proceso, read() y write() por ejemplo.

3) ¿ Cuáles de los siguientes puntos pertenecen a VFS?

□ File descriptor	□ Bloque de datos	**Inodo**
File	**Superbloque**	□ Directorio
		dentry

4) ¿Qué es un hardlink, soflink, un volumen y un mount point?
RESPUESTA:

- **Hardlink**: Es el mapeo entre el nombre del archivo y el archivo en sí. Los hardlinks se refieren al mismo archivo a través de un mismo inodo, por lo tanto, cada enlace es una copia exacta tanto de datos, como permisos, propietario, etc.
- **Softlink**: El softlink, a diferencia del hardlink, crea un inodo completamente nuevo. Sería como un acceso directo de windows ya que apunta al mismo archivo, pero con otro inodo. A diferencia del hardlink, este enlace también se puede dar con directorios. Y además si se elimina el enlace no se eliminará el auténtico.
- **Volumen**: Es una abstracción que corresponde a un disco lógico, en el caso más es un disco correspondiente a un disco físico, podría ser un pendrive también. En síntesis, es una colección de recursos físicos de almacenamiento.
- **Mount Point**: Es un punto en el cual el root de un volumen se engancha dentro de la estructura de un file system ya existente.

5) Describa el API del sistema de archivos. Diferencia entre las system calls y library calls, ponga un ejemplo.
RESPUESTA:

El API de un FS, como ya se mencionó es el VFS, que permite que múltiples FS interactúen entre sí, y que el usuario los use todos de la misma manera.
La API contiene 4 estructuras:

- El superbloque, que contiene metadata del FS
- Los inodos que son la representación de un archivo
- Dentry que son entradas de los directorios, que representan rutas parciales a archivos u otros directorios.
- File: Es la representación de un archivo abierto (en distintos modos como lectura y escritura) para un determinado proceso.

Además de las estructuras, la API está formada por las syscalls que se implementan para usar los directorios y archivos utilizando funciones que realiza el kernel con las estructuras mencionadas antes. A nivel usuario las syscalls que se utilizan se pueden separar en dos grupos:

- *las que operan sobre los archivos:*
  - open(): convierte el nombre de un archivo en una entrada de la tabla de descriptores de archivos, y devuelve dicho valor.
  - creat(): equivale a llamar a open() con los flags O_CREAT|O_WRONLY|O_TRUNC.
  - close(): cierra un file descriptor.
  - read(): se utiliza para hacer intentos de lecturas hasta un número dado de bytes de un archivo.
  - write(): escribe hasta una determinada cantidad (count) de bytes.
  - lseek(): reposisiona el despazamiento (offset) de un archivo abierto.
  - dup(): crea una copia del file descriptor del archivo.
  - link(): crea un nuevo nombre para un archivo.
  - unlink(): elimina un nombre de un archivo del sistema de archivos.

- *las que operan sobre los **metadatos** de los archivos:*
  - stat(): devuelven información sobre un archivo.
  - access(): chequea si un proceso tiene o no los permisos para utilizar el archivo.
  - chmod(): cambian los bits de modos de acceso.
  - chown(): cambian el id del propietario del archivo y el grupo de un archivo.

Las syscalls son funciones que un proceso utiliza para pedirle al kernel que realice una operación de kernel space en su nombre. Para llamarla se deben setear los registros correspondientes en los valores deseados y se debe levantar una excepción para que el kernel venga y handelee el pedido de syscall. Luego de que el kernel ejecuta el llamado devuelve los resultados por stack o por los registros. Todo esto resulta bastante engorroso, es por eso que existen las library calls que esencialmente son wrappers de las syscalls dadas por algunas bibliotecas que se encargan de la partes de bajo nivel para llamar a una syscall, simplificando el uso de las mismas para el usuario. Por ejemplo, fork() de la biblioteca estándar de C es un library call, que envuelve a la syscall clone.

6) ¿Qué es un inodo? ¿Qué es un superbloque? ¿Qué es un bitmap?
RESPUESTA:

- **Super bloque**: representa a un sistema de archivos. Existe en disco y en memoria. En disco provee información al kernel de la estructura del file system. En memoria provee la información necesaria y los estados para manipular al file system montado.
- **Inodo**: representa a un determinado archivo dentro del disco. Un inode puede referenciar a un archivo, un directorio o un link simbólico a otro objeto. El inode consiste de data y operaciones que describen sus contenidos y las operaciones que pueden realizarse en él (ej. open, read, write). Un inodo simplemente es referido por un número llamado inumber que sería lo que hemos llamado el nombre subyacente en el disco

de un archivo. Dado un inumber se puede saber directamente en que parte del disco se encuentra el inodo correspondiente.

- **Bitmap**: es una estructura bastante sencilla en la que se mapea 0 si un objeto está libre y 1 si el objeto está ocupado.

7) Los nombres de archivos no se almacenan en los inodos, sino en: **Los directorios**
Este diseño posibilita la implementación de enlaces de tipo:

- □ syslinks
- □ ambos
- ▯ **hardlinks**
- □ ninguno

8) Describa qué es un *inodo, sus componentes internos, donde se encuentra y ponga un ejemplo de cómo almacena los datos de un archivo (chico, mediano y muy grande) en disco. A qué subsistema pertenece y cómo se relaciona con sus componentes.
RESPUESTA:
Un inodo es la abstracción asociada a un archivo en el volumen de datos. Un inodo es la estructura que almacena simultáneamente los datos crudos en cuestión y la metadata que le da orden y sentido desde la perspectiva del file system y a la vista del kernel. En un sistema de archivos que se implemente como el Very Simple File System, tendríamos a los inodos en los primeros 8 bloques del volumen de bloques. Estarían justo después del bloque del super bloque, del bloque del bitmap de datos y del bitmap de inodos. Los componentes y estructura interna se resumen en:

- ▯ permisos de escritura, lectura, ejecución para usuario y grupo
- ▯ tipo de archivo
- ▯ cantidad de bloques
- ▯ operaciones de inodo
- ▯ número de nodo
- ▯ punteros directos a bloques
- ▯ puntero indirecto

Si el archivo asociado al inodo es chico, este tendrá pocos bloques y por tanto la dirección de cada bloque podrá ser almacenada en los punteros directos (son unos 15 en VSFS). Si el archivo es de tamaño medio se sobrepasará la capacidad de almacenar por punteros directos y se deberá requerir a la indirección. El puntero de indirección apunta a un array de bloques directos y a su vez a otro puntero indirecto.
Un archivo medio se almacenaría en los arrays de punteros directos apuntados por el primer puntero de indirección. Si el archivo fuese realmente grande requeriríamos el uso de varios niveles de indirección. El máximo permitido en VSFS es de 3 (cosa que da capacidad para archivos del tamaño del orden de los teras). Solo para completar la explicación este archivo ocuparía todo el array de punteros directos. Todos los arrays de punteros directos asociados al primer puntero de indirección y varios arrays de punteros directos asociados a punteros de indirección siguientes.
Cabe aclarar que esta indirección es anidada.
El inodo es parte de la interfaz pedida por VFS. El inodo almacena la información y el dentry le da el nombre human readable. Luego tenemos el super bloque que tiene una referencia de donde esta cada componente (normalmente hay copias de este super bloque) y por último tenemos también archivos per se que estarían queriendo representar la versión abierta de un archivo en memoria (la idea del file descriptor).

9) Describa la estructura de un i-nodo
RESPUESTA:
El inodo es una estructura que almacena la metadata y es de vital importancia porque mantiene información como: tamaño del archivo, fecha de modificación, propietario, información de seguridad, qué bloque de datos pertenece a un determinado archivo, etc.
No son estructuras muy grandes, normalmente ocupan unos 128 o 256 bytes.

10) Sea un disco que posee 2049 bloques de 4 KB y un sistema operativo cuyos i-nodos son de 512 Bytes. Definir un sistema de archivos FFS. Explique las decisiones tomadas.

RESPUESTA:

Bloques de 4 KB (1024*4 = 4096), inodos de 512 Bytes => 4096/512 = 8 inodos por bloque.

Como lo que más se almacena en un sistema de archivos son datos de los usuarios, voy a dedicarle un 90% aproximadamente a la data region. Estimando eso sería alrededor de 2049 * 0.9 = 1844, así que busco una cantidad de bloques cercana. Por simplicidad, tomaré 1840 bloques de datos. Luego, me quedan 2049 - 1840 = 209 bloques.

De esos 209 bloques, ya sé que debo dedicar 1 bloque para el superbloque, otro para el bitmap de inodos y otro para el bitmap de datos. Luego, ahora tengo 206 bloques para los inodos.

Como tengo 206 bloques para los inodos, mi sistema de archivo tendrá como máximo 8*206 = 1648 inodos, que también es la cantidad máxima de archivos que podrá contener el sistema de archivos.

La estructura queda tomando la partición de N = 2049 bloques, es decir de 0 a 2048:

[0] – Superbloque

[1] – Bitmap de inodos

[2] – Bitmap de datos

[3 - 208] – Inodos

[209 - 2048] – Data region

11) Tenes 1124 bloques de 4kB cada uno y cada inodo pesa 256 bytes. Implementar un FFS (file system simple).

RESPUESTA:

Bloques de 4 KB (1024*4 = 4096), inodos de 256 Bytes => 4096/256 = 16 inodos por bloque.

Como lo que más se almacena en un sistema de archivos son datos de los usuarios, voy a dedicarle un 90% aproximadamente a la data region. Estimando eso sería alrededor de 1124 * 0.9 = 1011, así que busco una cantidad de bloques cercana. Por simplicidad, tomaré 1024 bloques de datos. Luego, me quedan 1124 - 1024 = 100 bloques.

De esos 100 bloques, ya sé que debo dedicar 1 bloque para el superbloque, otro para el bitmap de inodos y otro para el bitmap de datos. Luego, ahora tengo 97 bloques para los inodos.

Como tengo 97 bloques para los inodos, mi sistema de archivo tendrá como máximo 16*97 = 1552 inodos, que también es la cantidad máxima de archivos que podrá contener el sistema de archivos.

La estructura queda tomando la partición de N = 1124 bloques, es decir de 0 a 1123:

[0] – Superbloque

[1] – Bitmap de inodos

[2] – Bitmap de datos

[3 - 99] – Inodos

[100 - 1123] – Data region

12) Sea un disco que posee 128 bloques de 4kb y un sistema operativo cuyos i-nodos son de 256 bytes. Defina la estructura completa del sistema de archivos unix-like. Justificar cada elección.

RESPUESTA:

Utilizo VSFS como vimos en el apunte.

Al contar con inodos de 256 bytes (2^8) y bloques de 4kb (2^12), entonces cada bloque de i-nodos puede guardar 16 i-nodos.

Como tenemos 128 bloques, y cada bloque de inodos referencia a 16 de estos, necesitaremos 8 bloques de i-nodos.

El bloque 0 va a ser el Super Bloque.

El bloque 1 será el bitmap de i-nodos, donde sabemos que 1 alcanza ya que tiene 2^15 bits en un bloque de 4kb, y esto sobra.

El bloque 2 va a ser el bitmap de data, y de forma análoga, tiene espacio más que suficiente.

Los bloques 3 hasta el 10, seran la tabla de inodos, con los 8 bloques de 16 inodos cada uno.

Los bloques restantes desde el 11 hasta el 127 son bloques de datos.

No utilizamos boot block en esta implementación.

13) Sea un disco que posee 256 bloques de 4kb y un sistema operativo cuyos i-nodos son de 512 bytes. Defina la estructura completa del sistema de archivos unix-like. Justificar cada elección.

RESPUESTA:

Los inodos ocupan 512 bytes, con lo cual en cada bloque entran 4096/512 = 8 inodos. Necesitamos tener, al menos, 256 inodos para cubrir todos los bloques del disco. Por lo tanto, necesitariamos 256/8 = 32 bloques para inodos.
El primer bloque se utiliza para el superbloque. El segundo y tercero para el inode-bitmap y data-bitmap respectivamente. Estos bitmaps indican cuales bloques e inodos estan utilizados o libres.
Luego vienen los 32 bloques de inodos y el resto de data.
En resumen, tenemos que:
Bloque 0: Superbloque.
Bloque 1: inode-bitmap.
Bloque 2: data-bitmap.
Bloques 3-34: inode blocks.
Bloques 35-255: data blocks.

14) Sea un disco que posee 512 bloques de 8kb y un sistema operativo cuyos i-nodos son de 256 bytes. Defina la estructura completa del sistema de archivos unix-like. Justificar cada elección.

RESPUESTA:

Para definir una estructura completa del sistema de archivos unix-like necesitamos asignar espacio a un super bloque, un bitmap de datos, un bitmap de inodos, una región de datos y una tabla de inodos.
Mínimamente se puede asignar un bloque a cada estructura, por lo que si bien los bitmaps van a ocupar mucho menos espacio necesariamente le tenemos que asignar 1 bloque.
Estructura:
- Bloque 0 -> Super bloque
- Bloque 1 -> Bitmap de inodos
- Bloque 2 -> Bitmap de datos

Quedan asignar 512-3=509 bloques a la región de datos y a la tabla de inodos.
Cada bloque ocupa 8 KB (8192 bytes) por lo que si los inodos ocupan 256 bytes, se disponen de 8192/256=32 inodos por bloque.
Por otro lado, la cantidad máxima de inodos que se pueden tener en esta estructura de 512 bloques es de 512 inodos. Pues cada archivo ocupa mínimamente 1 bloque y está asociado a un inodo. La cantidad de inodos disponibles puede ser inferior a 512 (cantidad total de bloques) en el caso de que existan archivos grandes en el sistema que ocupen más de un bloque.
Si asignamos arbitrariamente 16 bloques a la tabla de inodos, podemos almacenar 16*32=512 inodos y nos quedarían disponibles 509-16=493 bloques para asignar a la región de datos por lo que necesitamos mínimamente 493 inodos. 512-493=19 < 32 por lo que no estamos desperdiciando bloques.
Finalmente, podemos asignar 16 bloques a la tabla de inodos y el resto de los 512-3-16=493 bloques a la región de datos.
- Bloques 3,4,..,18 -> tabla de inodos
- Bloques 19,12,..,511 -> región de datos

15) Se tiene un file system basado en i-nodos con la siguientes características:

- Los bloques son de 1 kiB (1024 bytes) [2^10]
- El tamaño de un i-nodo es de 64 bytes [2^6]
- La distribución de los bloques es:
- El bloque 0 es el boot_block
- El bloque 1 es el superblock
- El bloque 2 es el i-node bitmap
- El bloque 3 es el block-bitmap

I-node entry	Block ptr				data block #	content
0	1				0	mariano:10, dato:11, juan:12
					1	home5, mnt:4, bin:3
3	5				5	ls:40, cat:41
4	6				6	cdrom:33
5	0				7	apuntes.md:101, start.sh:32
					9	jos.c.md:104, start.c.:34
10	9				11	
11	9				43	
12	11					
40	45					
41	57					
32	111					
33	43					
34	44					
101	52					
102	53					
104	54					

- Hay 126 bloques dedicados a la i-node table
- Hay 128 bloques dedicados a datos

Dada la siguiente información de la tabla de i-nodos y el contenido de los bloques de datos, indicar:
a) ¿Qué se mostraría en pantalla o que equivale ejecutar ls /bin, ls /home/juan, ls /home/mariano? Indicar la secuencia de operaciones (lecturas de bloque blkrd indicando la numeración relativa a la sección de i-nodos o datos; y la numeración dentro del sistema entero), que se realizan para acceder al archivo /home/dato/start.sh. Indicar para cada bloque leído qué información contiene y qué parte resulta relevante.
b) ¿Hay algún archivo que tenga más de una referencia (hard link)? ¿Qué syscall o comando unix usaría para borrar este tipo de archivos?

RESPUESTA:

a) Para poder ejecutar los siguientes comandos:
ls/bin
ls/home/juan
ls/home/mariano
home/dato/start.sh
Vamos a empezar desde el root y ver a que block pointer nos manda, una vez que tenemos ese block pointer vamos a buscar el contenido de ese bloque al que apunte y acceder únicamente a los que nos solicita cada ejemplo.
ls/bin
inode entry / block ptr ----- data block # / content

0	1	1	home:5, mnt:4, bin:3 (me quedo con bin)
3	5	5	ls:40, cat:41

Entonces al ejecutar ls/bin obtenemos un listado de cat y ls
ls/home/*juan*
inode entry / block ptr ----- data block # / content

0	1	1	home:5, mnt:4, bin:3 (me quedo con home)
5	0	0	mariano: 10, dato: 11, *juan*: 12 (me quedo con juan)
12	11	11	- no hay nada

Entonces al ejecutar ls/home/juan no obtenemos nada

ls/home/*mariano*
inode entry / block ptr ----- data block # / content

0	1	1	home:5, mnt:4, bin:3 (me quedo con home)
5	0	0	*mariano*: 10, dato: 11, juan: 12 (me quedo con mariano)
10	9	9	jos.c.md: 104, start.c: 34

Entonces al ejecutar ls/home/mariano obtenemos jos.c.md y start.c

home/dato/start.sh
inode entry / block ptr ----- data block # / content

0	1	1	home:5, mnt:4, bin:3 (me quedo con home)
5	0	0	mariano: 10, *dato*: 11, juan: 12 (me quedo con dato)
11	9	9	jos.c.md: 104, start.c: 34

Entonces al querer acceder al archivo home/dato/start.sh nos va a lanzar un error ya que no existe el archivo en ese directorio

b) El bloque de datos 11 es el único que cuenta con dos referencias. Este bloque corresponde a los directorios /home/mariano y /home/dato. Debido a como se organizan los inodos, no hay forma de saber cual es un link al otro.
Para borrar un archivo se puede utilizar la syscall unlink, la cual le resta una unidad a la cantidad de links que tiene ese archivo. Cuando el contador llega a 0 es eliminado del sistema de archivos.

16) A partir de las siguientes syscalls, escriba un programa que utilice el API de archivos para:

a) simular un ls -i recursivo.

b) simular un find, que recibe un nombre y lo busca en el directorio actual.

c) simular un copy.

```c
#include <sys/types.h>
#include <dirent.h>

struct dirent {
    ino_t          d_fileno;                /* inode number */
    off_t          d_off;                 /* offset to the next dirent */
    unsigned short d_reclen;              /* length of this record */
    unsigned char  d_type;                /* type of file; not supported by all file system types */
    char           d_name[MAXNAMLEN + 1]; /* filename */
};

struct stat {
    dev_t st_dev;           /* ID of device cointaining file */
    ino_t st_ino;           /* Inode number */
    mode_t st_mode;         /* File type and mode */
    nlink_t st_nlink;       /* Number of hard links */
    uid_t st_uid;           /* User ID of owner */
    gid_t st_gid;           /* Group ID of owner */
    dev_t st_rdev;          /* Device ID (if special file) */
    off_t st_size;          /* Total size, in bytes */
    blksize_t st_blksize;   /* Block size for filesystem I/O */
    blkcnt_t st_blocks;     /* Number of 512B blocks allocated */
};

DIR* opendir(const char *dirname);
struct dirent* readdir(DIR* dirstream);
int closedir(DIR* dirstream);
int stat(const char *pathname, struct stat *statbuf);

int open (const char *pathname, flags);
int creat(const char *pathname, mode);
size_t read(fd, ...);
size_t write(fd, ...);
int unlink(path);
int fstat(int fd, statbuf);
```

RESPUESTA:

```c
void print_ls_i(char *pathname) {
    DIR *dp; // declaro el puntero al directorio
    struct dirent *ep; // declaro el puntero a la lista de elementos del directorio

    dp = opendir(pathname); // intento abrir el dir
    if (dp != NULL) {
        while (ep = readdir(dp)) { // mientras pueda leer del stream del dir,
                                   // imprimo el inode y el nombre
            puts("%u %s", ep->d_fileno, ep->d_name);
                }
        closedir(dp); // cierro el directorio
    }
    else { // si no abrió, lanzo error
        perror("No such directory");
    }
    return 0;
}
```

```c
void find(char *file) {
    DIR *dp; // declaro el puntero al directorio
    struct dirent *ep; // declaro el puntero a la lista de elementos del directorio

    dp = opendir("./"); // intento abrir el dir actual
    if (dp != NULL) {
        while (ep = readdir(dp)) { // mientras pueda leer del stream del dir,
                        // Me fijo que el string pasado se contenga en el nombre
                        if (strstr(ep->d_name, file) != NULL) {
                        // imprimo el nombre
                puts("%s",ep->d_name);
                        }
                }
        closedir(dp); // cierro el directorio
    }
    else { // si no abrió, lanzo error
        perror("No such directory");
    }
    return 0;
}
```

```c

void copy(string path1, string path2) {
    int fd_old; // file descriptor para path1
    int fd_new; // file descriptor para path2

    fd_old = open(path1, O_RDONLY); // intento abrir el path1.
    if (fd_old == -1) {
        perror("No such file or dir");
    }

    fd_new = create(path2, 0666); // intento crear el path2.
    if (fd_new == -1) {
        perror("Creation Error");
    }

    int count_read; // declaro la cantidad de chars leídos
    char buffer[2048]; // declaro un buffer

        // mientras que pueda seguir leyendo del fd del archivo original
    while (count_read = read(fd_old, buffer, sizeof(buffer)) > 0) {
                // escribo lo leído en el nuevo archivo
        write(fd_new, buffer, count_read);
    }
    exit(0);
}
```

Concurrencia

1) ¿Qué es un thread, cuál es su estructura y qué estados posee?

RESPUESTA:

Un thread es una secuencia de ejecución atómica que representa una tarea planificable en ejecución.
- Secuencia de ejecución atómica, cada thread ejecuta una secuencia de instrucciones como lo hace un bloque de código en el modelo de programación secuencial.
- Tarea planificable en ejecución: El sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando el desee.

En criollo, un thread es una secuencia de instrucciones independiente el cual el SO controla y planifica su ejecución, ejecutándose dentro de un proceso. El sistema operativo provee la ilusión de que cada uno de estos threads se ejecuta en su propio procesador de manera inmediata. Un thread se caracteriza por:

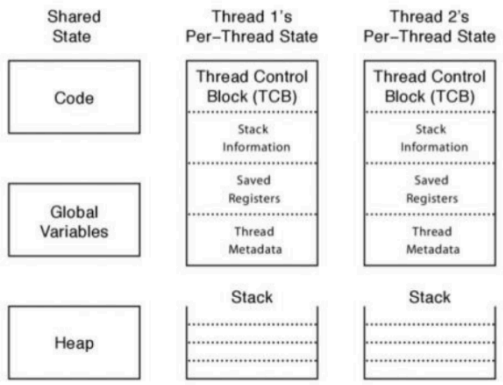
- Thread ID
- Un conjunto de los valores de registros
- Stack propio
- Política y prioridad de ejecución
- Propio errno
- Datos específicos del thread

Cada thread tiene una estructura que representa su estado, esta es llamada Thread Control Block (TCB), se crea una entrada por cada thread. La TCB almacena el estado per-thread de un thread.
Para poder crear múltiples Threads, pararlos y re-arrancarlos, el sistema operativo debe poder guardar el estado actual del bloque de ejecución en la TCB:

- Puntero al stack del thread.
- Una copia de sus registros del procesador.

Además se guarda metadata sobre el thread en esta estructura:

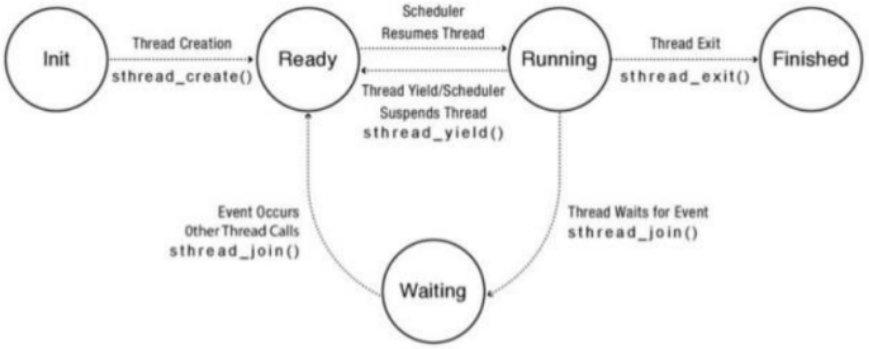
- ID
- Prioridad de Scheduling
- Status



El sistema operativo para permitir esta abstracción también se debe guardar la información de estado que es compartida por varios threads (estado compartido).

- Código.
- Variables globales.
- Heap.

Los estados que puede tomar un thread son los siguientes:



- **Init**: Un thread se encuentra en este estado cuando se está inicializando el estado per-thread y se está reservando el espacio necesario para las estructuras. Una vez que esto finaliza se lo pasa a estado READY y se lo pone en una lista llamada ready-list en la cual están esperando todos los threads listos para ejecutar.
- **Ready**: Thread está listo para correr pero no está siendo ejecutado en ese instante. La TCB está en la ready-list, en cualquier momento el thread scheduler puede transicionar al estado RUNNING
- **Running**: Thread está siendo ejecutado por el procesador en ese instante y los valores de los registros de la TCB están cargados en el procesador, en ese momento el thread puede pasar a estado READY de dos formas:
  - El scheduler lo pasa de estado mediante el desalojo o preemption del mismo mediante el guardado de los valores de registro y cambiando el thread que se está ejecutando por el próximo en la lista.
  - Voluntariamente un thread puede solicitar abandonar la ejecución mediante la utilización de thread_yield, por ejemplo.
- **Waiting**: El thread está esperando que algún evento suceda. Un thread no puede pasar de Waiting a Running directamente, por lo tanto estos threads se almacenan en la lista de waiting-list. Una vez que el evento ocurre el scheduler se encarga de pasar el thread de Waiting a Running, moviendo la TCB desde el waiting-list a la ready-list.
- **Finished**: Un thread que se encuentra en este estado jamás podrá volver a ser ejecutado. Existe una lista llamada finished-list en la que se encuentran los TCB que han terminado.

2) Defina la abstracción que mejor maneja la **granularidad más fina de cómputo** (thread). Cómo se denomina, sus características principales. Y en un ejemplo que muestre cómo funciona y cuáles son las precauciones que un programador debe tener en cuenta a la hora de utilizar dicha abstracción. Sin el ejemplo es inválido el ejercicio.

RESPUESTA:

Un thread es una secuencia de ejecución atómica que representa una tarea planificable de ejecución y está compuesto por: metadata, stack, registros de CPU, código y datos.

3) Explique detalladamente qué es un thread. Cuáles son sus componentes. Qué metadata se debe almacenar. Cuál es su relación con el paralelismo.

RESPUESTA:

4) ¿Cuáles de los siguientes mecanismos son compartidos entre threads de un mismo programa?

□ Stack segment	▯ **Heap**	▯ **Data segment**
▯ **File descriptors**	▯ **Code segment**	□ Signals
□ Registros de CPU	□ METADATA del thread	

5) Describa qué 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

RESPUESTA:

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.

```
void main() {
      int* x = malloc();
      t1 = pthread_create(sumar, x);
                  …
      t5 = pthread_create(sumar, x);
```

```
        pthread_join(t1, x);
              …
        pthread_join(t5, x);
}

int sumar (int x) {
        lock (l1);
        if (*x < 1000) {
              *x = *x + 7;
        }
        unlock(l1);
        return(x);
}
```

6) Defina y dé ejemplos: race-condition, heisenbug, dead-lock, interleave.

RESPUESTA:

- **Race-condition**: Es cuando más de un thread ejecuta sobre una misma región del 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 2 resultados posibles:

  Ejecutando primero el thread 1 y después el 2, tenemos:
  ```
  x = 2
  x = 2*x = 2*2 = 4
  x = x + 1 = 4 + 1 = 5
  x = 5
  ```
  En cambio, ejecutando primero el thread 2 y después el 1, tenemos:
  ```
  x = 2
  x = x + 1 = 2 + 1 = 3
  x = 2*x = 2*3 = 6
  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.
  Ejemplo: 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.
  Ejemplo:

  Thread 1:
  ```
  lock (L1);
  lock (L2);
  ```
  Thread 2:
  ```
  lock (L2);
  lock (L1);
  ```
  Si el orden de ejecución se da como indica 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.

- **Interleave**: Concepto que representa el intercalado de la ejecución de los threads en el CPU y depende del scheduler.
  Ejemplo:

7) Defina: Intercalado de instrucciones atómicas, race-condition, exclusión mutua, dead-lock.

RESPUESTA:

- **Intercalado de instrucciones atómicas**:
- **Race-condition**: Es cuando más de un thread ejecuta sobre una misma región del código y el resultado depende de en qué orden se ejecutan las instrucciones.
- **Exclusión mutua**:
- **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.

8) Muestre una implementación de un tipo de dato en el cual una variable que es compartida por varios threads es modificada y accedida de forma sincronizada.

RESPUESTA:

9) Dado la siguiente estructura de datos, implemente la función insertar y la función crear, de forma tal que sea thread safe:

```c
int List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));

    if (new == NULL) {
        perror("malloc");
        return -1; // fail
    }

    new->key = key;
    new->next = L->head;
    L->head= new;
    return 0; // success
}

// nodo
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

// lista
typedef struct __list_t {
    node_t *head;
} list_t;
```

RESPUESTA:

10) Dado el siguiente tipo de dato, identificar la sección crítica:

```c
struct QNode {
    int key;
    struct QNode* next;
};

struct Queue {
    struct QNode *front, *rear;
};

void deQueue(struct Queue* q){
    if (q->front == NULL) {
        return;
    }

    struct QNode* temp = q->front;
    q->front = q->front->next;

    if (q->front == NULL) {
        q->rear = NULL;
    }

    free(temp);
}
```

RESPUESTA:

Veamos que al comenzar la función, se accede al atributo *front* del objeto *q*, de modo que el estado del mismo afecta al resto de la implementación y si -durante otro proceso- fuera modificado, el resto de la implementación se vería afectada. Veamos un ejemplo:

a. Un proceso A llama a la función *deQueue(q0)*, donde *q0 != NULL*
b. El proceso A llega hasta la línea 12 y comienza la ejecución de un proceso B que llama a la misma función *deQueue(q0)*
c. El proceso B llega hasta la línea 12 y continúa la ejecución del proceso A
d. El proceso A llega a la línea 13 y modifica el valor de *q->front* por *NULL*, luego continúa hasta la línea 16 y continúa la ejecución del proceso B
e. El proceso B llega a la línea 13 y se lanza una excepción de tipo segmentation fault, pues *q->front == NULL* y no puede asignársele el valor de *q->front->next*.

Esto podría haberse evitado si el objeto *q0* se hubiera lockeado durante la ejecución de ambos procesos. Así, la sección crítica del código en cuestión es toda la implementación de la función *deQueue* y podría corregirse de la siguiente manera:

```
struct QNode {
    int key;
    struct QNode* next;
};

struct Queue {
    struct QNode *front, *rear;
};

void deQueue(struct Queue* q){
    lock(q);
    if (q->front == NULL) {
        unlock(q);
        return;
    }

    struct QNode* temp = q->front;
    q->front = q->front->next;

    if (q->front == NULL) {
        q->rear = NULL;
    }

    free(temp);
    unlock(q);
}

```