a. El superbloque de un sistema de archivos indica que el (3) inodo correspondiente al directorio raíz es el #43. En la siguiente secuencia de comandos, y siempre partiendo de ese directorio raíz, se pide indicar la cantidad de inodos y bloques de datos a los que se precisa acceder (leer) para resolver la ruta dada a *cat(1)* o *stat(1)*.

`# mkdir /dir /dir/s /dir/s/w`
`# touch /dir/x /dir/s/y`
`# stat /dir/s/w/x`
`# stat /dir/s/y`
	Inodos: ___ Blq. datos: ___
	Inodos: ___ Blq. datos: ___

`# ln /dir/s/x /dir/h`
`# ln -s /dir/s/y /dir/y`
`# cat /dir/h`
`# cat /dir/y`
	Inodos: ___ Blq. datos: ___
	Inodos: ___ Blq. datos: ___

Ayuda: todos los directorios ocupan un bloque. La idea es que describan como stat llega a los archivos

b. Describa la estructura de un i-nodo.

2. Scheduling , Memoria y Concurrencia

a. ¿Que es un deadlock describa por lo menos tres casos diferentes en el que puede suceder esta situación?

b. Cual es la cantidad de Kbytes que se pueden almacenar en un esquema de memoria virtual de 48 bits con 4 niveles de indirección, en la cual una dirección de memoria se describe como sigue: 9 bits page dir., 9 bits para cada page table y 12 bits para el offset. Explicar.

c. Explique cuál es la idea central de MLFQ y porque es mejor que otras políticas de scheduling, justifique su respuesta.

3. Proceso y Kernel

a. Escriba un programa en C que permita jugar a dos procesos al ping pong, la pelota es un entero, cada vez que un proceso recibe la pelota debe incrementar en 1 su valor. Se corta por overflow o cambio de signo.

b. Cuáles son los requerimientos mínimos de hardware para poder construir un kernel.

1a)
mkdir /dir /dir/s /dir/s/w
touch /dir/x /dir/s/y

```
root
└── dir
    ├── x.file
    └── s
        ├── y.file
        └── w
```

1. stat /dir/s/w/x

1. inodo root (#43) → bloque de datos con dentry (“dir”, inodo)
2. inodo dir → bloque de datos con dentry (“s”, inodo ; “x”, inodo)
3. inodo s → bloque de datos con dentry (“y”, inodo ; w, inodo)
4. inodo w → bloque de datos con dentry (vacío, el comando devuelve error)

inodos = bloque de datos = 4

2. stat /dir/s/y

1. inodo root (#43) → bloque de datos con dentry (“dir”, inodo)
2. inodo dir → bloque de datos con dentry (“s”, inodo ; “x”, inodo)
3. inodo s → bloque de datos con dentry (“y”, inodo ; w, inodo)
4. inodo y, el cual contiene los atributos para stat

inodos = 4, bloque de datos = 3

ln /dir/s/x /dir/h (asumo que existe /dir/s/x porque si no devuelve error)
ln -s /dir/s/y /dir/y (acá no es tan necesario que exista /dir/s/y pero asumo que sí)

```
root
└── dir
    ├── y.softlink
    ├── h.hardlink
    └── s
        ├── x.file
        └── y.file
```

3. cat /dir/h

1. inodo root (#43) → bloque de datos con dentry (“dir”, inodo)
2. inodo dir → bloque de datos con dentry (“s”, inodo ; “y”, inodo, “h”, inodo)
3. inodo h = inodo /dir/s/x → bloque de datos de h o /dir/s/x

inodos = bloque de datos = 3

4. cat /dir/y

1. inodo root (#43) → bloque de datos con dentry (“dir”, inodo)
2. inodo dir → bloque de datos con dentry (“s”, inodo ; “y”, inodo, “h”, inodo)

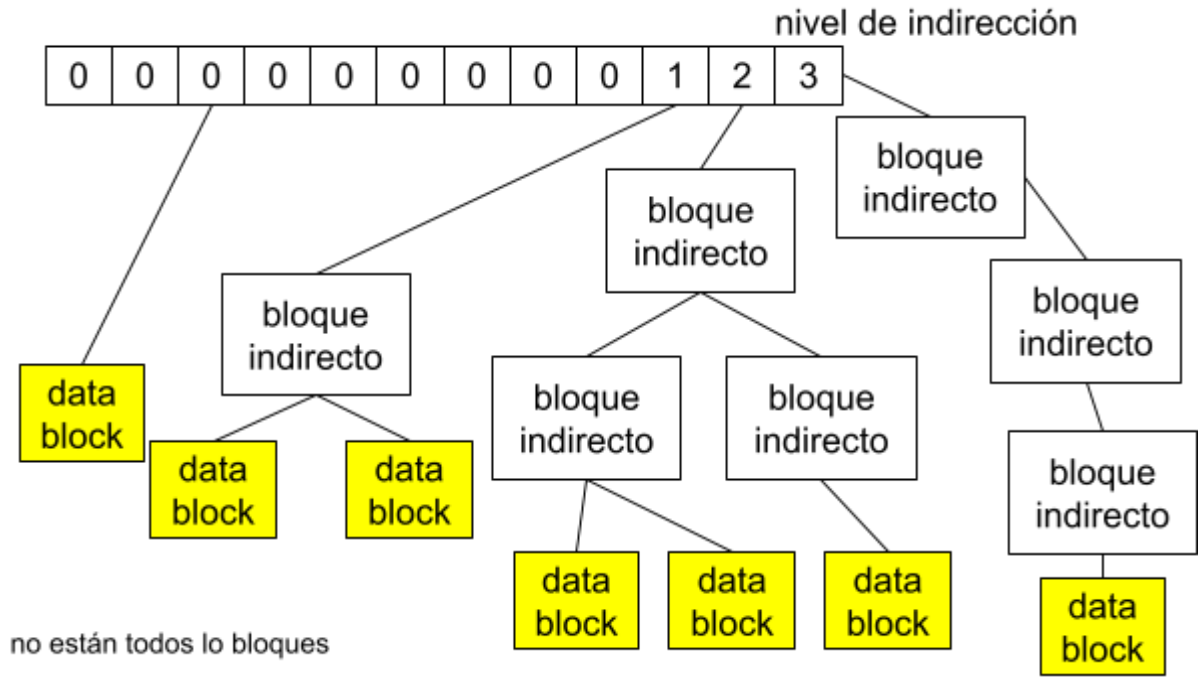
3. inodo y → bloque de datos de y (contiene path “/dir/s/y”)
4. inodo root (#43) → bloque de datos con dentry (“dir”, inodo)
5. inodo dir → bloque de datos con dentry (“s”, inodo ; “y”, inodo, “h”, inodo)
6. inodo s → bloque de datos con dentry (“x”, inodo ; “y”, inodo)
7. inodo y → bloque de datos de y

inodos = bloque de datos = 7

1b) El inodo contiene metadata y punteros (directos o indirectos) a los bloques de datos

mode	n° links	UID	GID	size	access date	mod date	create date	pointers	actual block count	generation n°	

- **mode**: permisos de lectura, escritura, ejecución
- **n° links**: número de hard links. si es cero, se elimina el archivo.
- **UID**: ID del usuario dueño; **GID**: ID del grupo dueño
- **Size**: Tamaño del archivo
- **Fecha de acceso, modificación, creación**
- **Punteros**:

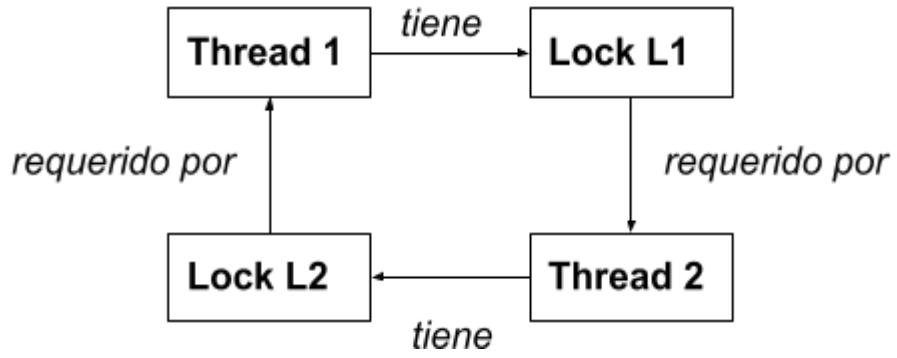


- **Actual block count**: Cantidad de data blocks utilizados
- **Generational N°**: Ni idea
- Y más (?)

2a)
Un deadlock ocurre cuando dos o más procesos se encuentran en un estado de espera indefinido donde ninguno puede continuar dado que esperan simultáneamente que se libere un recurso/lock poseído por otro proceso que a la vez hace lo mismo.

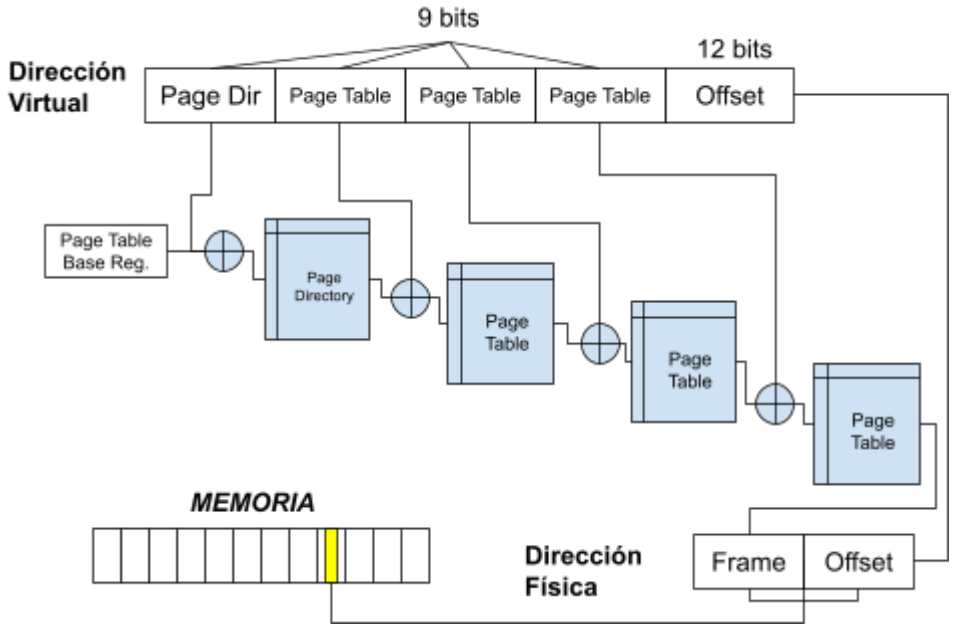
Ej 1:

se ejecuta **thread 1**: pthread_mutex_lock(**L1**); // thread 1 tiene lock L1
se ejecuta **thread 2**: pthread_mutex_lock(**L2**); // thread 2 tiene lock L2
se ejecuta **thread 1**: pthread_mutex_lock(**L2**); // thread 1 necesita lock L2
se ejecuta **thread 2**: pthread_mutex_lock(**L1**); // thread 2 necesita lock L1



2b) 48 bits → 2^{48} posiciones de memoria posibles
Si asumimos cada dirección como un byte → 2^{48} bytes
2^{48} bytes = 2^{48} kilobytes / 1024 =2^{48} kilobytes / 2^{10} = 2^{38} kilobytes
(274877906944 kilobytes = 256 terabytes)

El funcionamiento de la paginación multitabla es el siguiente:



2C) Los Multilevel Feedback Queue Schedulers mejoran sobre la idea de Round Robin, una política de scheduler que logra evitar el starvation, pero que no funciona bien con tareas de tiempo variable.
MLFQ le agrega a este múltiples colas de distintas prioridades. A cada una se le asignan procesos de una misma prioridad, y en aislamiento cada una se comporta como Round Robin. Las colas se diferencian en su Time Quantum. Este es el tiempo antes de un interrumpt que genera un context switch para darle lugar a otro proceso; a mayor prioridad, mayor Time Quantum.
La elección de prioridad sigue ciertas reglas. Por ejemplo, los procesos I/O-bound son asignados la máxima prioridad. Así mismo, nuevos son asignados máxima prioridad pero cada vez que no son completados, bajan un nivel. Esto permite solucionar el problema de performance promedio de cada tarea permitiendo que los procesos más cortos sean completados primero.
En resumen, MFQ usa colas de prioridad para lograr un compromiso entre responsiveness, low overhead de context switching y fairness.

3C)
Por simplicidad no chequee errores de pipe, fork, read, write. Se supone que es para un examen. No olvidar que los pipes son UNIDIRECCIONALES entonces necesito dos.

```c
#define READ 0
#define WRITE 1
#define INT_MAX

int main(void) {
    int pelota = 0;

    int padre_a_hijo[2], hijo_a_padre[2];
    pipe(padre_a_hijo);
    pipe(hijo_a_padre);

    pid_t pid = fork();
    if (pid) { // PADRE
        close(padre_a_hijo[READ]);
        close(hijo_a_padre[WRITE]);
        while(pelota >= 0 && pelota <= INT_MAX) {
            pelota++;
            write(padre_a_hijo[WRITE], &pelota, sizeof(pelota));
            read(hijo_a_padre[READ], &pelota, sizeof(pelota));
        }
        close(padre_a_hijo[WRITE]);
        close(hijo_a_padre[READ]);
    }
    else { // HIJO
        close(padre_a_hijo[WRITE]);
        close(hijo_a_padre[READ]);
        while(pelota > 0 && pelota <= INT_MAX) {
            read(padre_a_hijo[READ], &pelota, sizeof(pelota));
            pelota++;
            write(hijo_a_padre[WRITE], &pelota, sizeof(pelota));
        }
        close(padre_a_hijo[READ]);
        close(hijo_a_padre[WRITE]);
    }
    return 0;
}
```

3b) CPU, RAM? Bootloader?

a. Describa Cuales son los componentes de VFS y que función cumple VFS en el kernel de linux.

b. Describir el proceso de acceso (lectura de inodos y bloques)
De /home/darthmendez/opt/tool/sisop.txt suponiendo que el archivo está vacío. Además describa para qué sirven todas las estructuras involucradas.

Scheduling , Memoria y Concurrencia

a. Concurrencia: ¿Describa detalladamente con un esquema cuál es la diferencia estructural entre un proceso y un thread? ¿Cuál es la curiosa implementación de estos en linux?

b. Memoria: Describa cómo fue variando la estructura del address space respecto la memoria física en : base y bound, tabla de registros y paginación. Explique con diagramas

c. Scheduling: Cuál es la mejora respecto a MLFQ que introduce el scheduler de linux.

roceso y Kernel

a. Escriba un programa en C que simule el funcionamiento de una shell primitiva. Que debería agregarse para poder encadenar dos comandos por las salidas y entradas estándar.

b. Que es el address space, cual es su estructura y que función cumple. Explique detalladamente y con gráficos.

1a)
El virtual file system es un sistema para

1b)

/home/darthmendez/opt/tool/sisop.txt

root
└── home
  └── darthmendez
    └── opt
      └── tool
        └── sisop.txt (vacío)

Acceso:

1. inodo root → bloque de datos con dentry (“dir”, inodo)
2. inodo dir → bloque de datos con dentry (“darthmendez”, inodo)
3. inodo darthmendez → bloque de datos con dentry (“opt”, inodo)
4. inodo opt → bloque de datos con dentry (“tool”, inodo)
5. inodo tool → bloque de datos con dentry (“sisop.txt”, inodo)
6. inodo dir, no tiene bloque de datos al estar vacío

3a)

```c
#define PROMPT "$"
#define BUF 256
#define CMD 15+1 // +1: NULL terminated

void tokenizar(char *cmd[CMD], char *buffer) {
    char *token;
    size_t i = 0;
    token = strtok(buffer, " \t\n");
    while (token != NULL && i < CMD - 1) {
        cmd[i++] = token;
        token = strtok(NULL, " \t\n");
    }
    cmd[i] = NULL;
}

int main(void) { // Shell Básica
    char buffer[BUF];
    char *cmd[CMD];
    while(1) {
        if(getcwd(buffer, BUF) == NULL) return 1;
        printf("%s%s ", buffer, PROMPT);
        if(fgets(buffer, BUF, stdin) == NULL) break; //EOF
        if (!strcmp(buffer, "exit\n")) break;

        tokenizar(cmd, buffer);

        if (!strcmp(cmd[0], "cd")) {
            if (chdir(cmd[1]) == -1) perror("Directory Not Found!");
            continue;
        }

        if (fork() == 0) // HIJO
            if(execvp(cmd[0], cmd) == -1) {
                perror("Error Comando");
                exit(EXIT_FAILURE);
            }

        // PADRE
        wait(NULL);
    }
    return 0;
}
```

Se puede agregar pipes con lógica que al encontrar el símbolo “|”, reconozca que la salida estándar del comando debe ser redirigida a la entrada estándar del siguiente comando. Se puede lograr eso usando, valga la redundancia, pipes ya que exec() y variantes no cambian los file descriptors.

3b)