SISTEMAS OPERATIVOS 2C-2023

Linux

Contenido

SO USUARIO

Perspectiva del Usuario

El usuario ve:

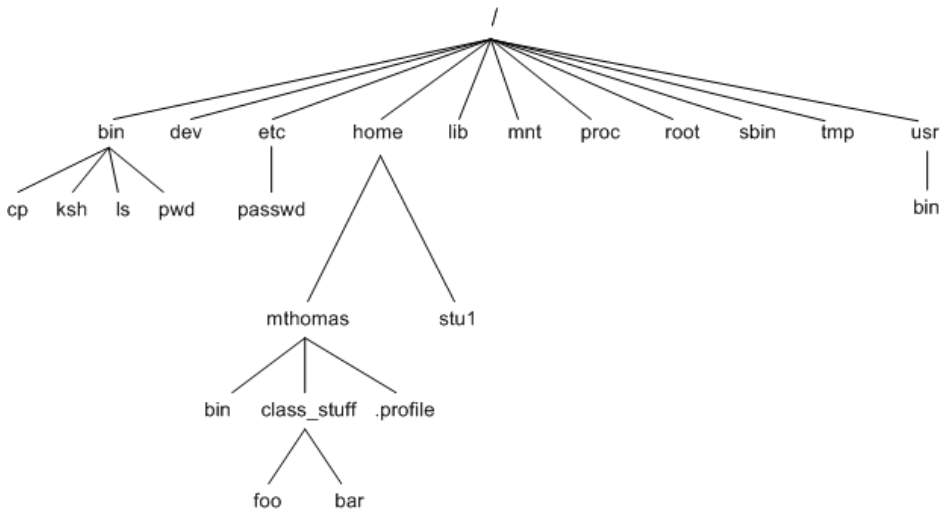
- El FileSystem.
- El Entorno de Procesamiento o Shell.
- Los Bloques primitivos de Construcción.

FileSystem

El sistema de archivos ofrece:

1. Estructura jerárquica.
2. Habilidad de crear y borrar archivos.
3. Tratamiento de dispositivos periféricos como si fueran archivos.
4. Protección de los archivos de datos.

En Linux TODO es un archivo.



Shell

Es el primer programa que se ejecuta (en un unix) en modo usuario. A su vez, ejecuta el comando de login.

Captura las teclas presionadas en el teclado y traduce a nombres de comandos.

Puede interpretar tres tipos de comandos:

- Comandos ejecutables simples: ls
- Shell scripts
- Estructuras de control: if-then-else-fi

Dado que el Shell es un programa, no forma parte del Kernel del SO.

Bloques Primitivos de Construcción

La idea de unix es la de proveer ciertos mecanismos para que los programadores construyan a partir de pequeños programas otros más complejos.

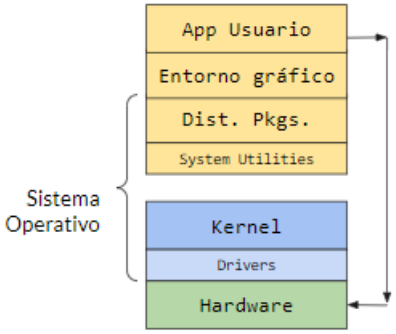
El redireccionamiento de entrada y salida de datos.:

- El operador > redirecciona el standar output a un archivo.
- El operador < redirecciona el standar input tomando los datos de un archivo.
- Las tuberías o los pipes.

KERNEL

Ejecución Directa

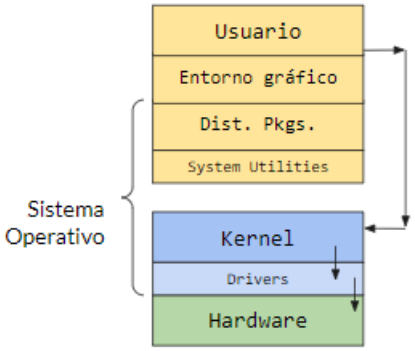
Las Apps de Usuario tienen acceso directo al hardware.



Es muy peligroso, un proceso puede adueñarse de los recursos.

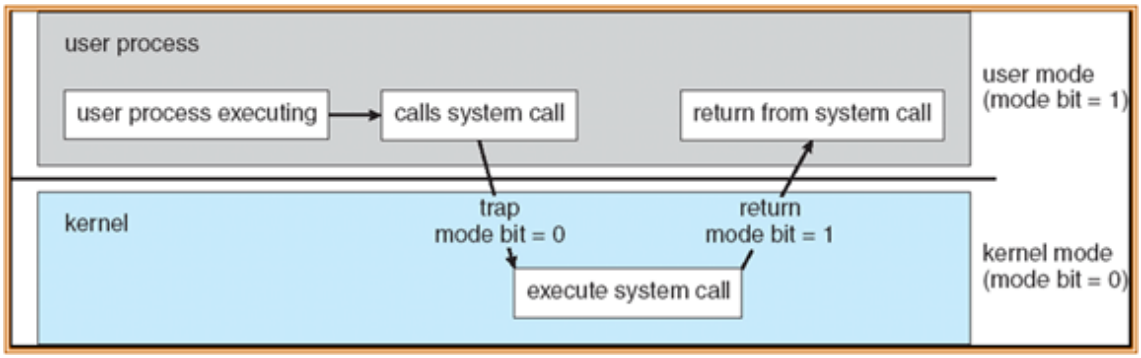
Modo dual de Operaciones

Como la ejecución directa es peligrosa, se plantea un modo dual de operaciones. El kernel interviene entre el usuario y el hardware.



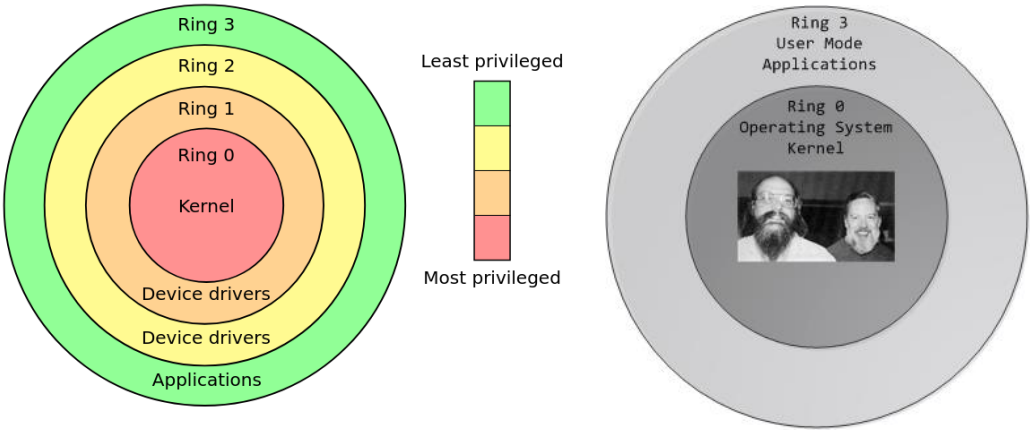
Se establecen dos modos de ejecución:

- User-Mode
- Kernel-Mode



IOPL

Para identificar el Modo de ejecución, se tiene un **I/O Privilege Level**, ocupa los bits 12 y 13 en el registro FLAGS.

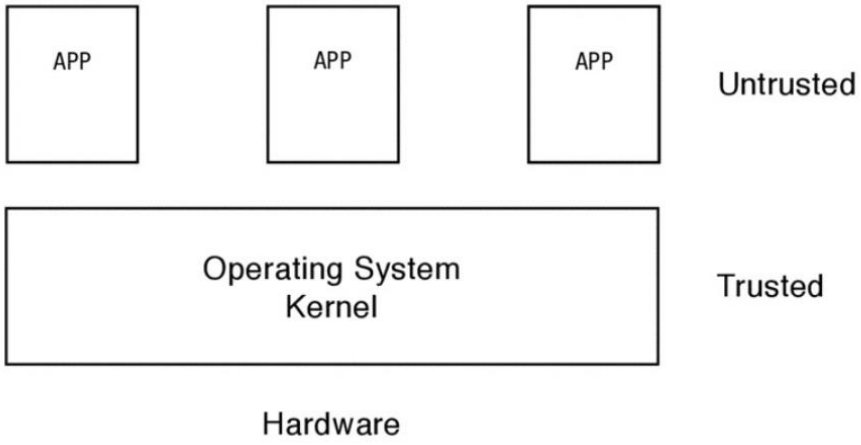


Cada ring puede ejecutar una determinada cantidad de instrucciones. Esto se detecta por hardware cada vez que una instrucción se ejecuta.

Lands

El SO se divide en:

- Kernel-Land: Capa para la gestión de dispositivos específico y una serie de servicios para la gestión de dispositivos del Hardware usados por las Apps de usuario.
- User-Land: Espacio donde viven las Apps de usuario.



Protección del SO

El Kernel protege las aplicaciones y los usuarios mediante tres bases:

- Instrucciones privilegiadas.
- Protección de Memoria.
- Timer Interrupts.

Instrucciones privilegiadas

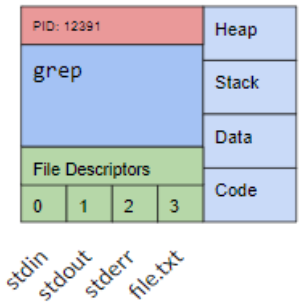
Distintos sets de instrucciones dependiendo el nivel de privilegio. Cuanto menor es el privilegio, menos instrucciones pueden ejecutarse.

Instrucciones NO privilegiadas:

- Reading the status of Processor.
- Reading the System Time.
- Generate any Trap Instruction.
- Sending the final printout of Printer.

Protección de Memoria

- Direcciones virtuales vs Direcciones físicas
- Address Space por proceso.



Timer Interrupts (Interrupciones por temporizador)

¿Cómo hace el Kernel para volver a tener lo que un proceso tiene?

Casi todos los procesadores contienen un dispositivo llamado Hardware Timer, cada timer interrumpe a un determinado procesador mediante una interrupción por hardware.

Cuando una interrupción por tiempo se dispara se transfiere el control desde el proceso de usuario al Kernel

Context Switch

¿Cómo se transiciona entre un modo y otro?

User-Mode to Kernel-Mode

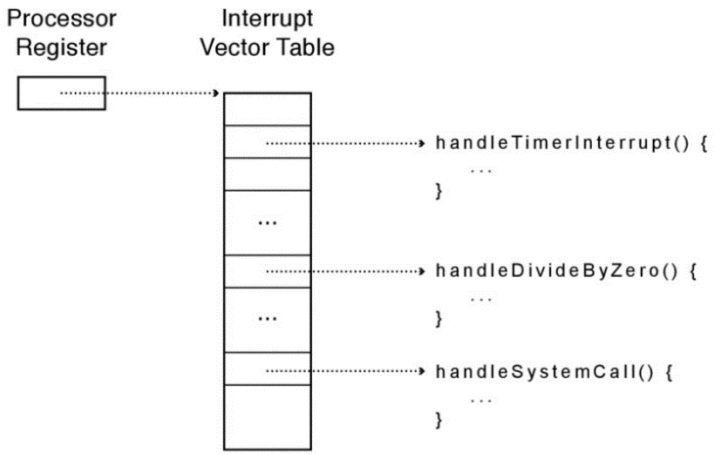
- Interrupciones
- Excepciones del Procesador
- System Calls

Interrupciones

Una interrupción es una señal asincrónica hacia el procesador avisando que algún evento externo requiere su atención

Orden de importancia:

1. Errores de la máquina.
2. Timers.
3. Discos Network devices.
4. Terminales.
5. Interrupciones de Software.



Excepciones del Procesador

Una excepción es un evento de hardware causado por una aplicación de usuario que causa la transferencia del control al Kernel. Por ej: Dividir por 0.

System Calls

Un proceso de usuario puede hacer que la transición de modo sea hecha voluntariamente.

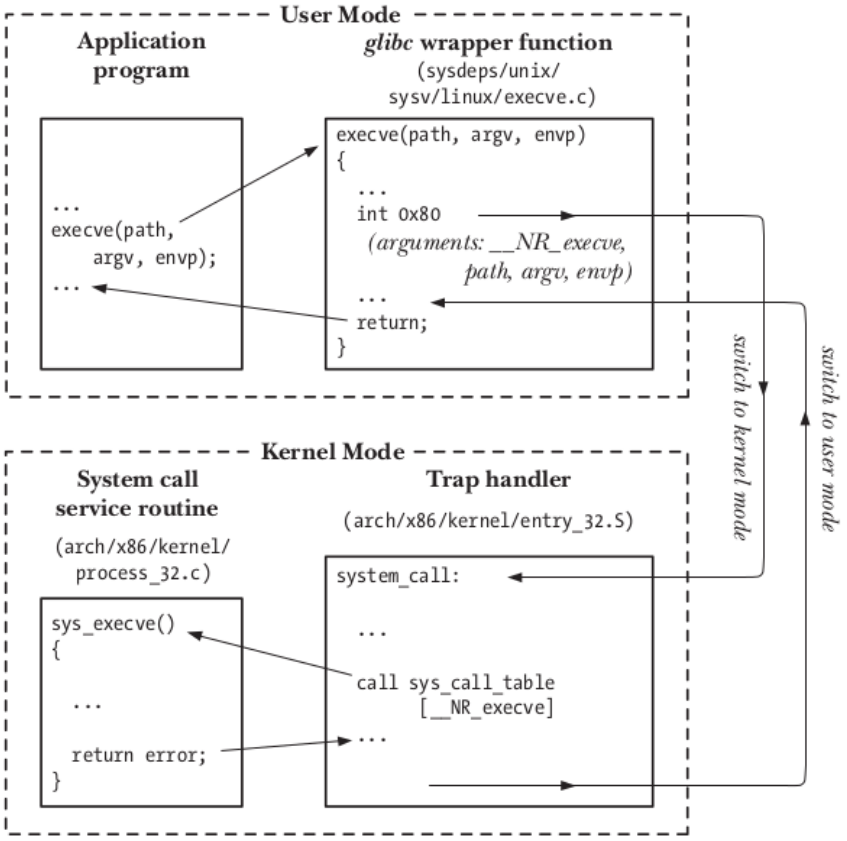
Desde el punto de vista de un programa llamar a una system call es más o menos como invocar a una función de C.

Proceso:

1. El programa realiza un llamado a una **syscall** mediante la invocación de una **función wrapper** (envoltorio) en la biblioteca de C.
2. La función wrapper tiene que proporcionar todos los argumentos al **system call trap_handling.**
    a. Estos argumentos son pasados al wrapper por el stack, pero el kernel los espera en determinados registros. La función wrapper copia estos valores a los registros.
    b. Dado que todas las system calls son accedidas de la misma forma, el kernel tiene que saber identificarlas de alguna forma. Para poder hacer esto, la función wrapper copia el número de la system call a un determinado registro de la CPU (%eax).

c. La función wrapper ejecuta una instrucción de código máquina llamada **trap machine instruction (int 0x80)**, esta causa que el procesador pase de user mode a kernel mode y ejecute el código apuntado por la dirección 0x80 (128) del vector de traps del sistema.

3. En respuesta al trap de la posición 128, el kernel invoca su propia función llamada syste_call() (arch/i386/entry.s) para manejar esa trap.
   a. Graba el valor de los registros en el stack del Kernel.
   b. Verifica la validez del número de la syscall.
   c. Invoca el servicio correspondiente a la syscall llamada a través del vector de Syscalls, el servicio realiza su tarea y finalmente le devuelve un resultado de estado a la rutina system_call().
   d. Se restauran los registros almacenados en el stack del Kernel y se agrega el valor de retorno en el stack.
   e. Se devuelve el control al wrapper y simultáneamente se pasa a user mode.
   f. Si el valor de retorno de la rutina de servicio de la syscall da error, la función wrapper setea el valor en errno.

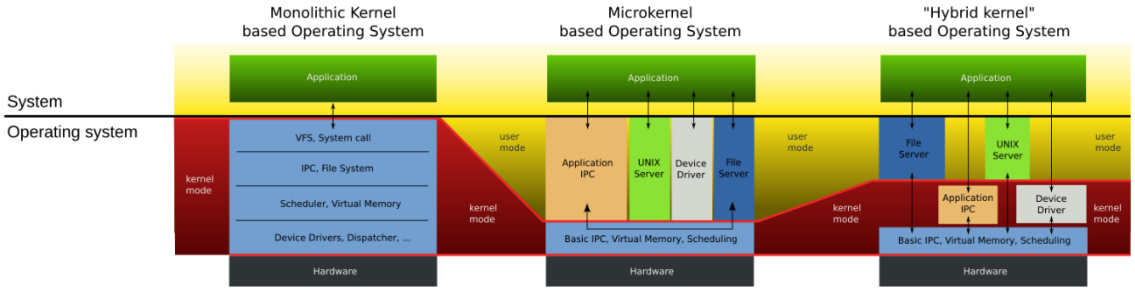


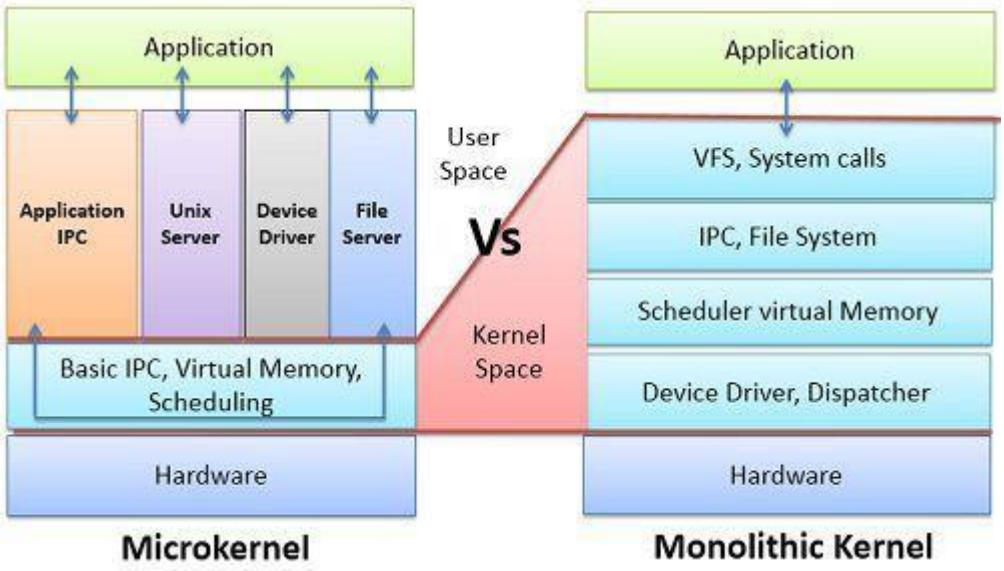
Kernel-Mode to User-Mode

- Nuevo Proceso.
- Continuar luego de una interrupción, excepción o una syscall.
- Cambiar entre diferentes procesos.

Tipos de Kernel

- Monolithic Kernel: - In a monolithic kernel, the kernel and operating system, both run in the same memory, and it is mainly used where security is not a major concern. The result of the monolithic kernel is fastly accessible. But in some situations, like if a device driver has a bug, then there may be chances of a whole system crash.
- Microkernel: - A Microkernel is the derived version of the monolithic kernel. In microkernel, the kernel itself can do different jobs, and there is no requirement of an additional GUI.
- Nano kernel: - Nano kernel is the small type of kernel which is responsible for hardware abstraction, but without system services. Nano kernel is used in those cases where most of the functions are set up outside.
- Exo kernel: - Exo kernel is responsible for resource handling and process protection. It is used where you are testing out an inhouse project, and in up-gradation to an efficient kernel type.
- Hybrid kernel: - Hybrid kernel is a mixture of microkernel and monolithic kernel. The Hybrid kernel is mostly used in Windows, Apple's macOS. Hybrid kernel moves out the driver and keeps the services of a system inside the kernel.





Inicio SO

1. Booteo.
2. Carga del Kernel.
3. Inicio de las Apps de Usuarios.

Booteo/Bootstrap

Este proceso es denominado Bootstrap, y generalmente depende del hardware de la computadora. En él se realizan los chequeos de hardware y se carga el bootloader, que es el programa encargado de cargar el Kernel del Sistema Operativo. Este proceso consta de 4 partes.

1. Cargar el BIOS (Basic Input/Output System): para eso apenas se enciende la PC, se carga CS con 0xFFFF y IP con 0x0000; por ende, la dirección de CS:IP es 0xFFFF0, justamente la dirección de memoria de la BIOS.
2. Crear la Interrupt Vector Table y cargar las rutinas de manejo de interrupciones en Modo Real: el BIOS pone la tabla de interrupciones en el inicio de la memoria 1 KB (0x00000–0x003FF), son 256 entradas de 4 bytes. El área de datos del BIOS de unos 256 B (0x00400–0x004FF), y el servicio de atención de interrupciones (8 KB), 56 KB, después de la dirección 0x0E05B.
3. La BIOS genera una interrupción 19 (INT 19) de la tabla de interrupciones la cual hace apuntar a CS:IP a 0x0E6F2.
4. Lo cual hace ejecutar el servicio de interrupciones, el handler de dicha interrupción, que es leer el primer sector de 512 bytes del disco a memoria, y ahí termina.

Inicio del Kernel

La función de arranque para el kernel (también llamado intercambiador o proceso 0) establece la gestión de memoria (tablas de paginación y paginación de memoria), detecta el tipo de CPU y cualquier funcionalidad adicional como capacidades de punto flotante, y después cambia a las funcionalidades del kernel para arquitectura no específicas de Linux, a través de una llamada a la función start_kernel().

Por lo tanto, el núcleo inicializa los dispositivos, monta el sistema de archivos raíz especificado por el gestor de arranque como de sólo lectura, y se ejecuta Init (/sbin/init), que es designado como el primer proceso ejecutado por el sistema (PID=1). También puede ejecutar opcionalmente initrd para permitir instalar y cargar dispositivos relacionados (disco RAM o similar), para ser manipulados antes de que el sistema de archivos raíz está montado.

En este punto, con las interrupciones habilitadas, el programador puede tomar el control de la gestión general del sistema, para proporcionar multitarea preventiva, e iniciar el proceso para continuar con la carga del entorno de usuario en el espacio de usuario.

El trabajo de Init es “conseguir que todo funcione como debe ser” una vez que el kernel está totalmente en funcionamiento. Establece y opera todo el espacio de usuario. Comprueba y monta el sistema de archivo. Pone en marcha los servicios de usuario necesarios y, en última instancia, cambia al entorno de usuario cuando el inicio del sistema se ha completado.

PROCESO

De Programa a Proceso

1. El programador edita código fuente.
2. El compilador compila el source code en una secuencia de instrucciones de máquina y datos llamada.
3. El compilador genera esa secuencia y posteriormente se guarda en disco: programa ejecutable.

Edición

El programador edita el código fuente.

Procesamiento

El preprocesador (cpp) modifica el código fuente original de un programa escrito en C de acuerdo a las directivas que comienzan con un carácter #. El resultado es otro programa en C con la extensión .i

Compilación

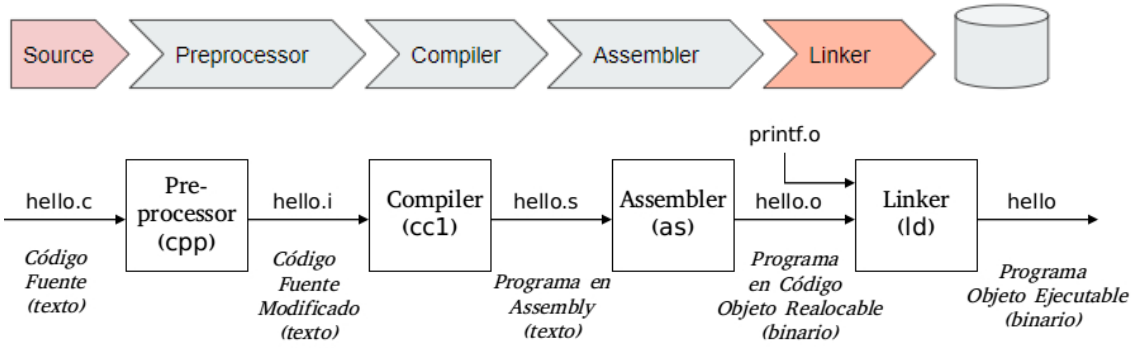
El compilador (cc) traduce el programa .i a un archivo de texto .s que contiene un programa en lenguaje assembly.

Ensamblaje

El ensamblador (as) traduce el archivo .s en instrucciones de lenguaje de máquina empaquetándolas en un formato conocido como programa objeto realocable con extensión .o

Link-Edición

La fase de link edición. Generalmente los programas escritos en lenguaje C hacen uso de funciones que forman parte de la biblioteca estándar de C que es provista por cualquier compilador de ese lenguaje. Por ejemplo la función printf(), la misma se encuentra en un archivo objeto pre compilado que tiene que ser mezclado con el programa que se está compilando, para ello el linker realiza esta tarea teniendo como resultado un archivo objeto ejecutable.



Programa

Un es un archivo que posee toda la información de cómo construir un proceso en memoria.

- **Instrucciones de Lenguaje de Máquina**: código del algoritmo del programa.
- **Dirección del Punto de Entrada del Programa**: Identifica la dirección de la instrucción con la cual la ejecución del programa debe iniciar.
- **Datos**: El programa contiene valores de los datos con los cuales se deben inicializar variables, valores de constantes y de literales utilizadas en el programa.
- **Símbolos y Tablas de Realocación**: describe la ubicación y los nombres de las funciones y variables de todo el programa.
- **Bibliotecas Compartidas**: describe los nombres de las bibliotecas compartidas que son utilizadas por el programa en tiempo de ejecución así como también la ruta del linker dinámico que debe ser utilizado para cargar dicha biblioteca.
- **Otra información**: El programa contiene además otra información necesaria para terminar de construir el proceso en memoria.

Programa en Linux: ELF

ELF: Extensible Linking Format

https://greek0.net/elf.html

comando readelf

Ver el SHT (Section Header Table)	readelf -S <archivo>
Ver el PHT (Program Header Table)	readelf -l <archivo>
Ver el ELF Header	readelf -h <archivo>
Ver la Realocation Table	readelf -r <archivo>
Ver el dump hexa	hexdump -C archivo | head -n 10

Intervención del Kernel

El Sistema Operativo más precisamente el Kernel se encarga de:

1. Cargar instrucciones y Datos de un programa ejecutable en memoria.
2. Crear el Stack y el Heap.
3. Transferir el Control al programa.
4. Proteger al SO y al Programa.

Proceso

“Un Proceso es una entidad abstracta, definida por el Kernel, en la cual los recursos del sistema son asignados” [KER]

Un proceso Incluye:

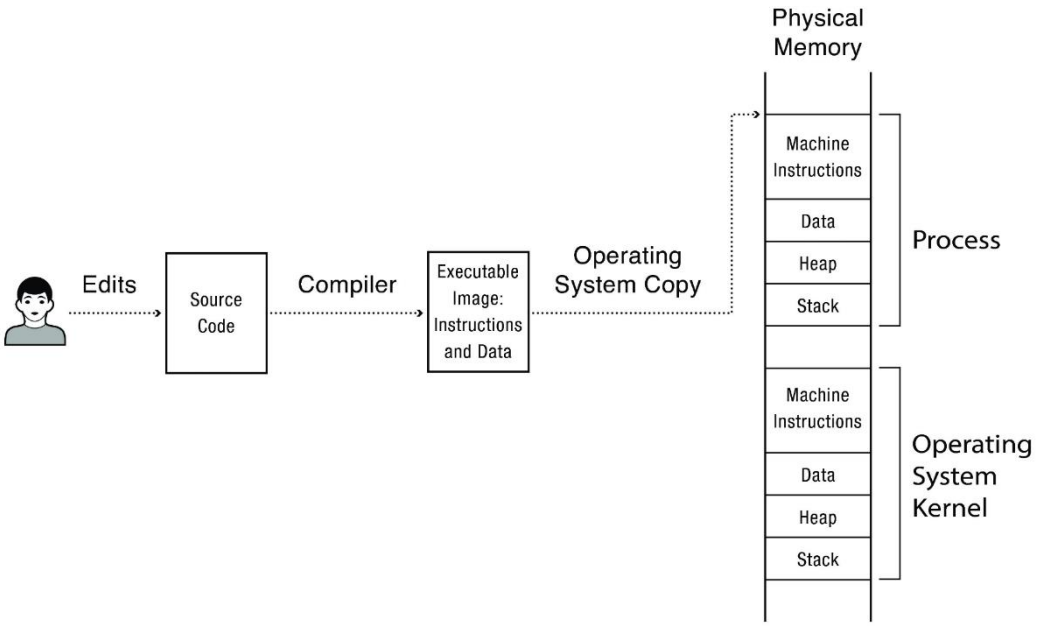
- Los Archivos abiertos.
- Las **signals** pendientes.
- Datos internos del Kernel.
- Estado completo del procesador.
- Un espacio de direcciones de memoria.
- Uno o más hilos de ejecución. Cada **thread** contiene:
    - Un único contador de programa.
    - Un Stack.
    - Un conjunto de Registros.
    - Una sección de datos globales.

Virtualización del Proceso

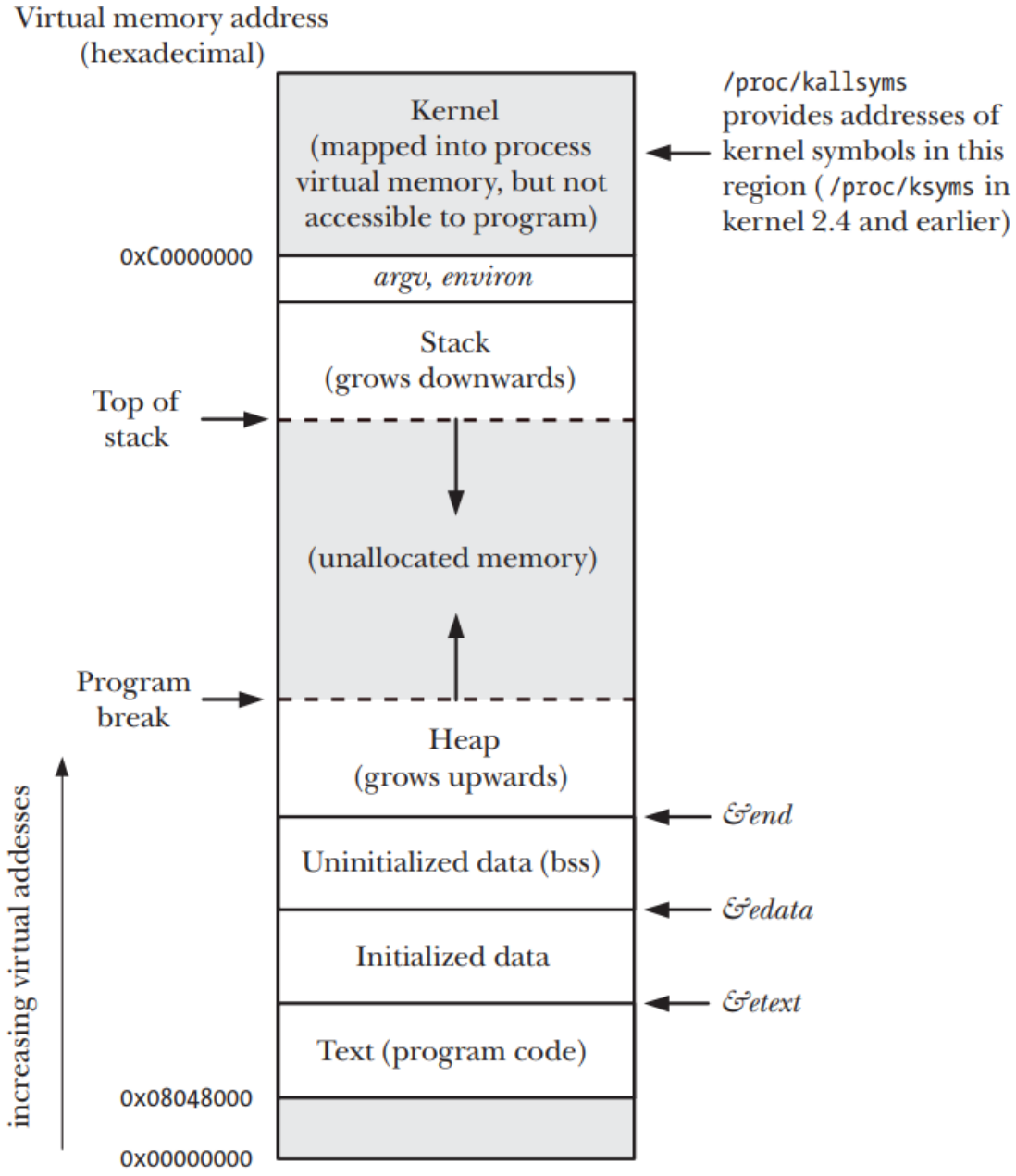
Virtualización de Memoria

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, está dividido en 4 segmentos:

- Text: Instrucciones del Programa.
- Data: Variables Globales (extern o static en C)
- Heap: Memoria Dinámica Alocable
- Stack: Variable Locales y trace de llamadas



Espacio de direcciones:



Memoria Virtual

La memoria virtual es una abstracción por la cual la memoria física puede ser compartida por diversos procesos.

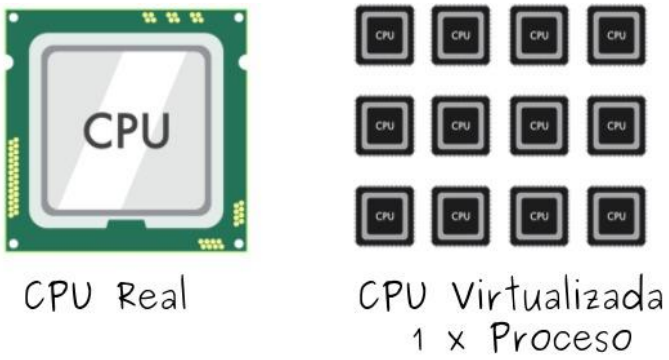
Direcciones virtuales

Un componente clave de la memoria virtual son las direcciones virtuales, con las direcciones virtuales, para cada proceso su memoria inicia en el mismo lugar, la dirección 0. El hardware (MMU) traduce la dirección virtual a una dirección física de memoria.

Virtualización de Procesador

La virtualización de procesamiento es la forma de virtualización más primitiva, consiste en dar la ilusión de la existencia de un único procesador para cualquier programa que requiera de su uso.

El SO crea esta ilusión mediante la virtualización de la CPU a través del kernel.



El Contexto de un Proceso

El contexto de un proceso es la información necesaria para describir al proceso.

Cada proceso posee un contexto.

Según Bach:" el contexto de un proceso comprende, el contenido del Address Space, el contenido de los registros de hardware y las estructuras de datos que pertenecen al kernel relacionadas con el proceso" .

El contexto de un proceso es la unión de:

- User-level context
- Register context
- System-level context

User-level Context

Consiste en las Secciones: Text, Data, Stack y Heap.

Register Context

- Program Counter Register
- Processor Status Register
- Stack Pointer Register
- General Purpose Registers

System-Level Context

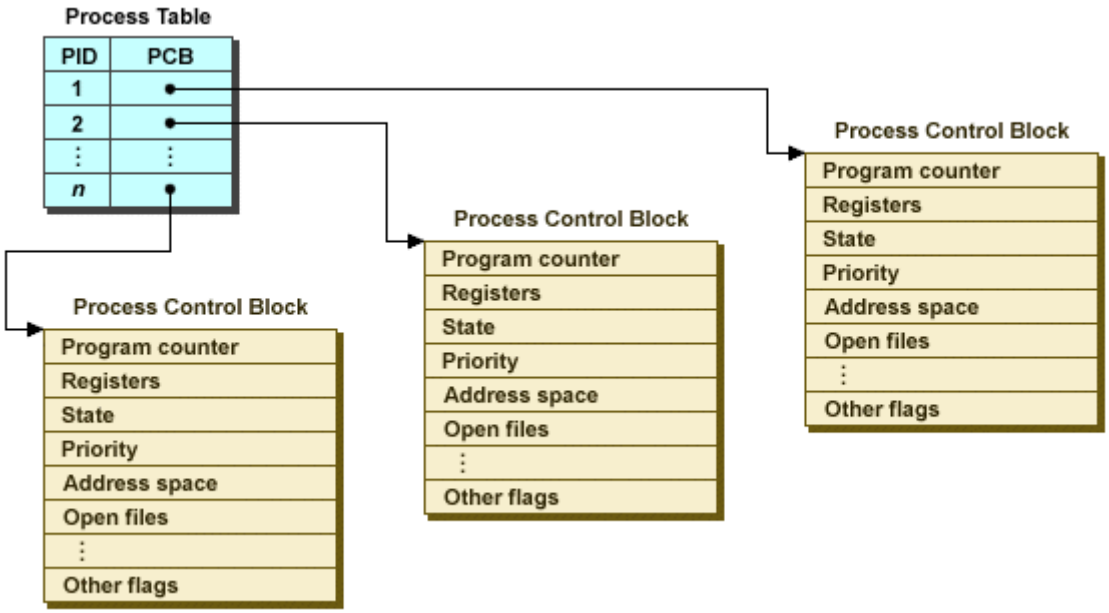
- La entrada en la **Process Table Entry**
- La **u area**
- La **Process Region Entry, Region Table y Page Table** que definen el mapeo de la memoria virtual vs memoria física del proceso.
- El Stack del Kernel, contiene los Stack frames (Marcan donde empieza la parte del Stack para cada función) de las llamadas al kernel hechas en el proceso.

La Process Table Entry y la u area son dos estructuras que pertenecen al Kernel.

Process table

Contiene información que siempre tiene que estar disponible para el kernel.

- Identificación: cada proceso tiene un identificador único o Process ID (PID) y además perteneces a un determinado grupo de procesos.
- Ubicación del mapa de direcciones del Kernel del u area del proceso.
- Estado actual del proceso.
- Un puntero hacia el siguiente proceso en el planificador y al anterior.
- Prioridad.
- Información para el manejo de señales.
- Información para la administración de memoria.



U area

Contiene campos que solo deben estar disponibles cuando el proceso está corriendo.

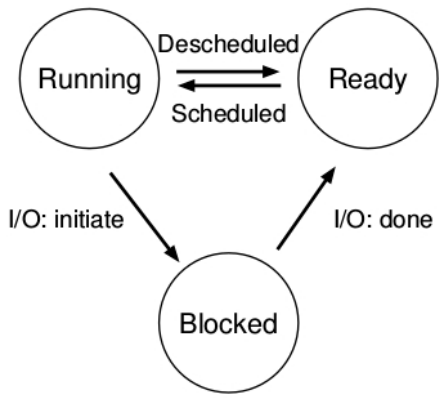
Contenido de la user area <arch/x86/include/asm/user.h>:

- Un puntero a la proc structure del proceso
- El UID y GID real
- Argumentos para, y valores de retorno o errores hacia, la system Call actual
- Manejadores de Señales
- Información sobre las áreas de memoria text, data, stack, heap y otra información.
- La tabla de descriptores de archivos abiertos (Open File descriptor Table).
- Un puntero al directorio actual.
- Datos estadísticos del uso de la CPU, información de perfilado, uso de disco y límites de recursos.

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process

struct context
{
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
                            // for this process
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    void *chan;                 // If non-zero, sleeping on chan
    int killed;                 // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;     // Switch here to run process
    struct trapframe *tf;       // Trap frame for the
                            // current interrupt
};
```

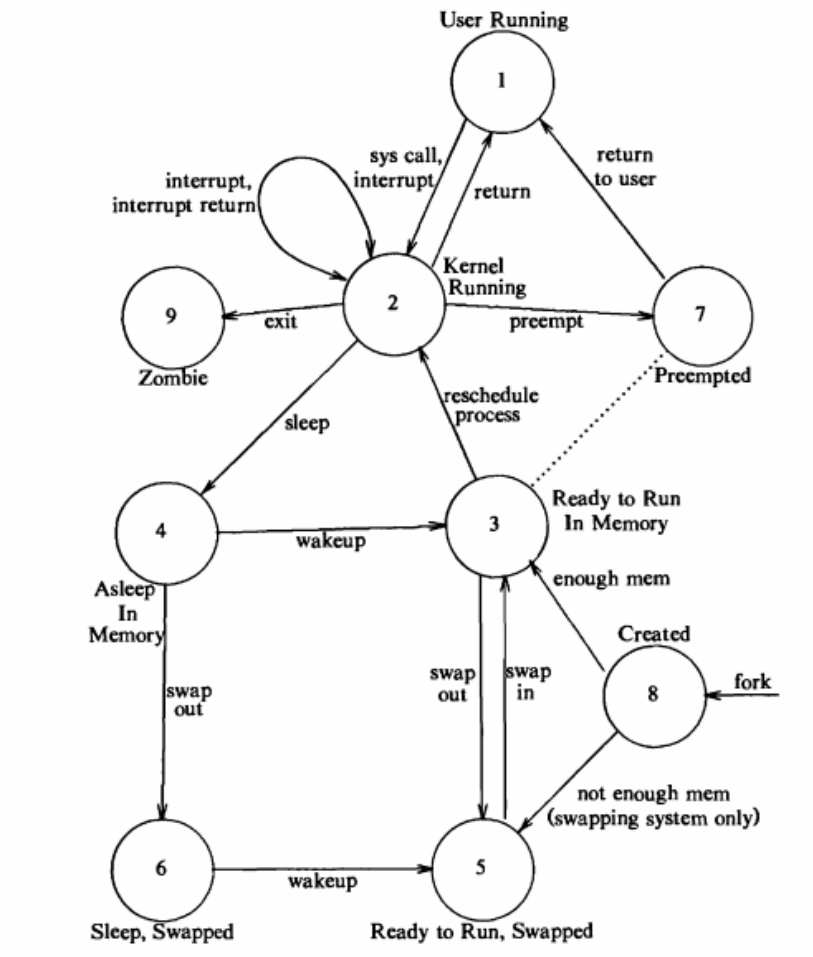
Estados de un Proceso

- Corriendo (Running): el proceso se encuentra corriendo en un procesador. Está ejecutando instrucciones.
- Listo (Ready): en este estado el proceso está listo para correr pero por algún motivo el SO ha decidido no ejecutarlo por el momento.
- Bloqueado (Blocked): en este estado el proceso ha ejecutado algún tipo de operación que hace que éste no esté listo para ejecutarse hasta que algún evento suceda.



Estados de un Proceso: System V

- Corriendo User Mode (Running User Mode): El proceso se encuentra corriendo en un procesador. Está ejecutando instrucciones.
- Corriendo kernel Mode (Running Kernel Mode).
- Listo para Correr en Memoria (Ready to Run on Memory): En este estado el proceso está listo para correr pero por algún motivo el SO ha decidido no ejecutarlo por el momento.
- Durmiendo en Memoria (Asleep In Memory) : El proceso está bloqueado en memoria.
- Listo para Correr pero Swapeado (Ready to Run but swapped): El proceso está bloqueado en memoria secundaria.
- Durmiendo en Memoria Secundaria (Asleep Swapped): El proceso se encuentra bloqueado en memoria secundaria.
- Preempt: Es igual a 1 pero un proceso que pasó antes por Kernel mode solo puede pasar a preentive.
- Creado (Created): El proceso está recién creado y en un estado de transición.
- Zombie (Zombie): El proceso ejecutó la Syscall. exit(), ya no existe más, lo único que queda es el exit state.



API Syscalls

- fork(): Crea un proceso y devuelve su id.
- exit(): Termina el proceso actual.
- wait(): Espera por un proceso hijo.
- kill(pid): Termina el proceso cuyo pid es el parámetro.
- getpid(): Devuelve el pid del proceso actual.
- exec(filename, argv): Carga un archivo y lo ejecuta.
- sbrk(n): Crece la memoria del proceso en n bytes.

Creación de un Proceso (fork)

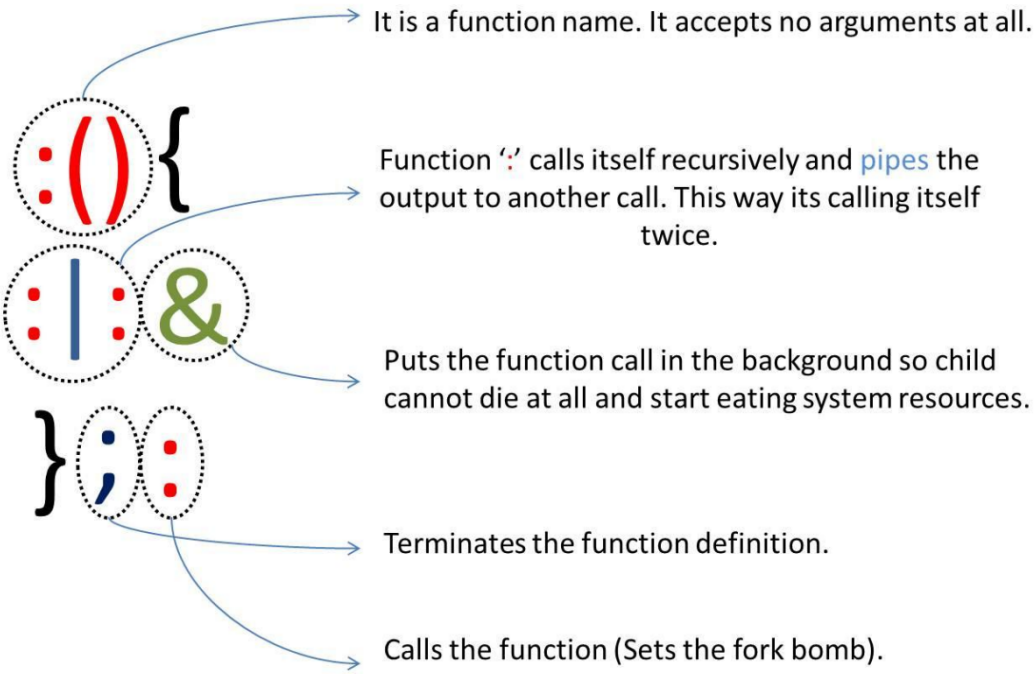
La única forma de que un usuario cree un proceso en el sistema operativo UNIX es llamando a la syscall fork.

El proceso que invoca a fork es llamado proceso padre, el nuevo proceso creado es llamado hijo.

¿Qué hace fork?:

- Crea y asigna una nueva entrada en la Process Table para el nuevo proceso.
- Asigna un número de ID único al proceso hijo.
- Crea una copia lógica del contexto del proceso padre, algunas de esas partes pueden ser compartidas como la sección text.
- Realiza ciertas operaciones de I/O.
- Devuelve el número de ID del hijo al proceso padre, y un 0 al proceso hijo

ForkBomb



MEMORIA

¿Qué es la Memoria?

La memoria física puede ser imaginada como un arreglo de direcciones de memoria una detrás de otra.

Multiprogramación

Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time on a uniprocessor. Since there is only one processor, there can be no true simultaneous execution of different programs. Instead, the operating system executes part of one program, then part of another, and so on. To the user it appears that all programs are executing at the same time.

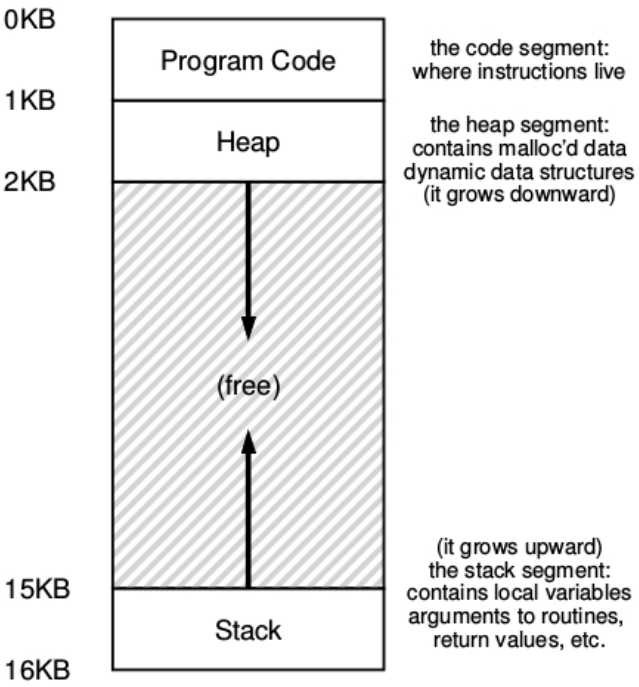
Time Sharing

Tiempo compartido se refiere a compartir de forma concurrente un recurso computacional (tiempo de ejecución en la CPU, uso de la memoria, etc.) entre muchos usuarios por medio de las tecnologías de multiprogramación y la inclusión de interrupciones de reloj por parte del sistema operativo, permitiendo a este último acotar el tiempo de respuesta del computador y limitar el uso de la CPU por parte de un proceso dado

Address Space

El espacio de direcciones es la abstracción para la memoria.

El Address Space de un proceso contiene todo el estado de la memoria de un programa en ejecución.



Virtualización de Memoria

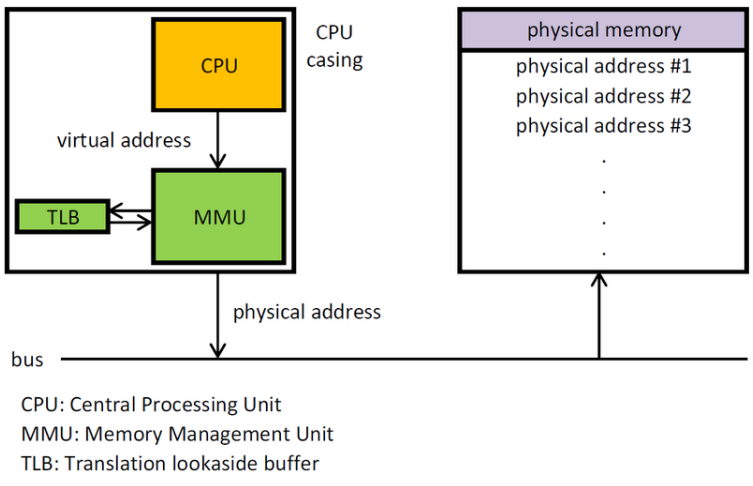
- Transparencia: invisible al programa que se está ejecutando; el programa debe comportarse como si él estuviera alojado en su propia área de memoria física.
- Eficiencia: en términos de tiempo y espacio.
- Protección: tiene que asegurarse de proteger a los procesos unos de otros como también al sistema operativo de los procesos.

Debe ser flexible y eficiente.

Address Translation

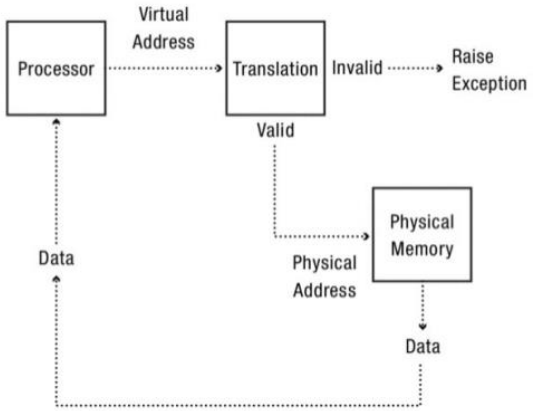
Con esta técnica el hardware transforma cada acceso a memoria, transformando la **Virtual Address** que es provista desde dentro del Address Space en una **Physical Address** en la cual la información deseada se encuentra realmente almacenada.

En todas y por cada una de las referencias a memoria, una Address Translation es realizada.



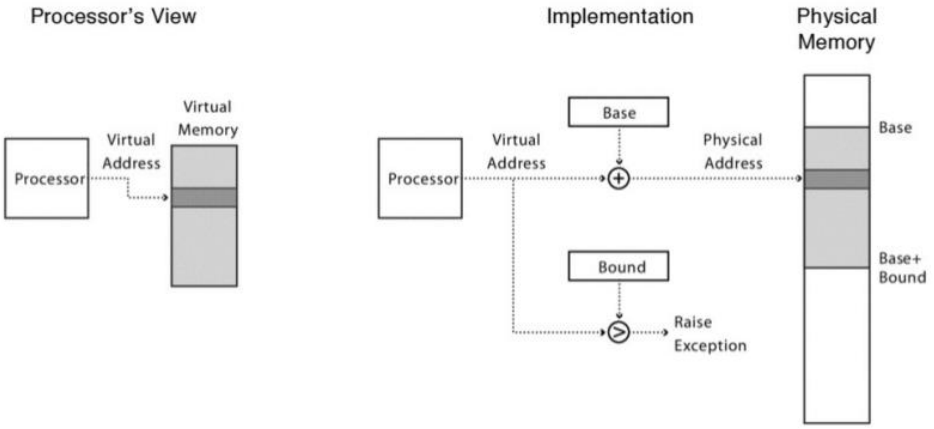
El hardware no virtualiza la memoria por si solo, éste provee un mecanismo de bajo nivel para poder hacerlo eficientemente. El SO debe involucrarse en los siguientes puntos:

- Setear el hardware de forma correcta para que se de la traducción.
- Mantener información de en qué lugar hay áreas libres y en qué lugar no.
- Intervenir criteriosamente como mantener el control sobre toda la memoria usada.



Base and Bound

Específicamente solo se necesitan dos registros de hardware dentro de cada cpu: Uno llamado registro base y el otro registro límite o Segmento.



Este par base-límite va a permitir que el Address Space pueda ser ubicado en cualquier lugar deseado de la memoria física, y se hará mientras el sistema operativo se asegura que el proceso solo puede acceder a su Address Space

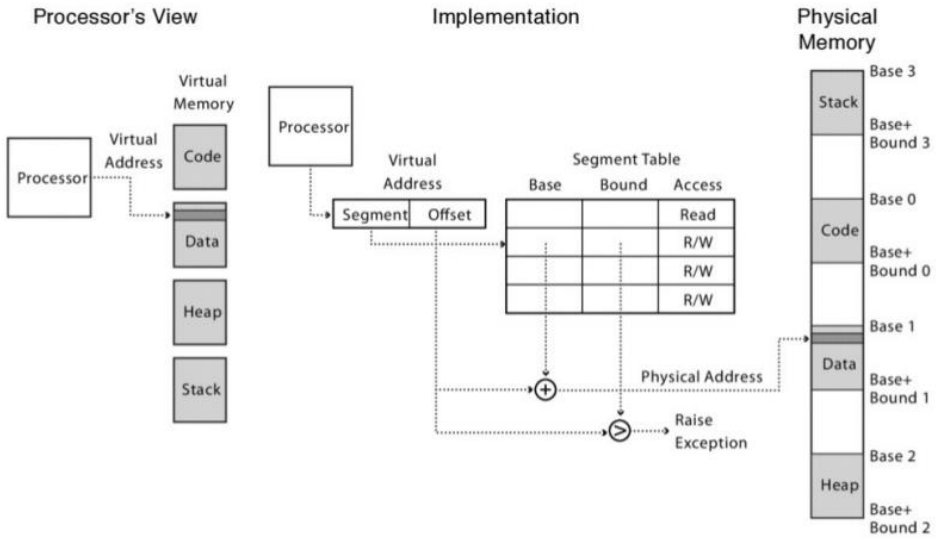
Tabla de Segmentos

El problema de la técnica anterior es que se tiene un solo registro base y solo un segmento. La mejora a este método es mediante la aplicación de un pequeño cambio: en vez de tener un solo registro límite, se tiene un arreglo de pares de (registro base, segmento) por cada proceso.

Una dirección virtual tiene dos componentes:

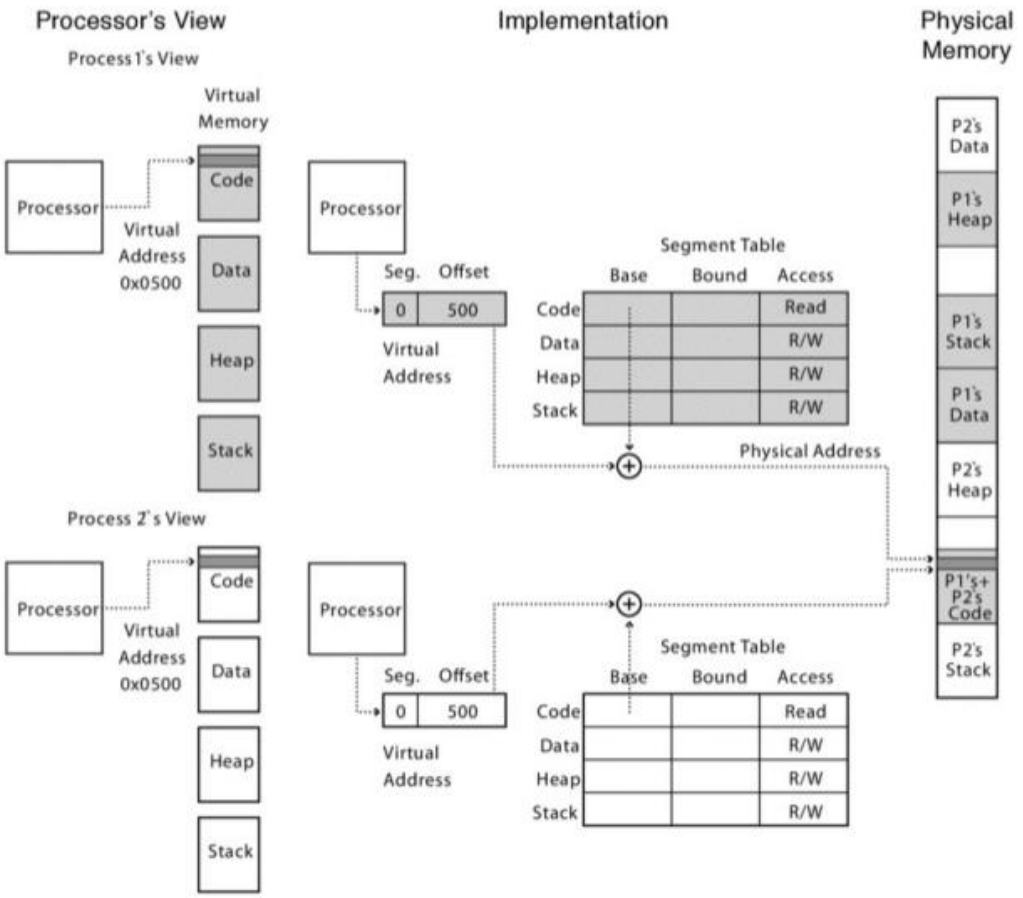
un número de segmento:un offset de segmento

El número de segmento es el índice de la tabla para ubicar el inicio del segmento en la memoria física. El registro bound es chequeado contra la suma del registro base+offset para prevenir que el proceso lea o escriba fuera de su región de memoria.



En una dirección virtual utilizando esta técnica, los bits de más alto orden son utilizados como índice en la tabla de segmentos. El resto se toma como offset y es sumado al registro base y comparado contra el registro bound.

El número de segmentos depende de la cantidad de bits que se utilizan como índice.



Memoria Paginada

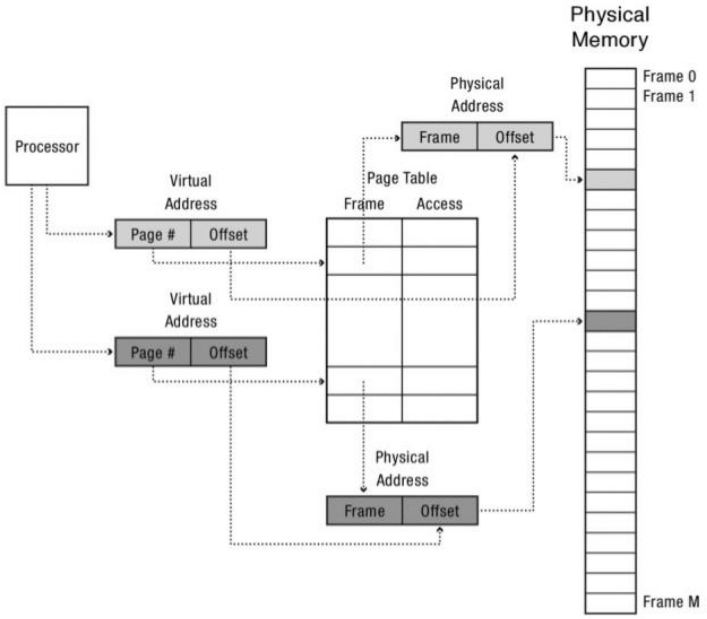
En vez de tener una página de segmentos cuyas entradas contienen punteros a segmentos, hay una tabla de páginas por cada proceso cuyas entradas contienen punteros a las page frames.

Teniendo en cuenta que los page frames tienen un tamaño fijo, y son potencia de 2, las entradas en la page tablen sólo tienen que proveer los bits superiores de la dirección de la page frame.

No es necesario tener un límite; la página entera se reserva como una unidad. Con la paginación, la memoria es reservada en pedazos de tamaño fijo llamados page frames.

El número de la página virtual es el índice en la page table para obtener el page frame en la memoria física.

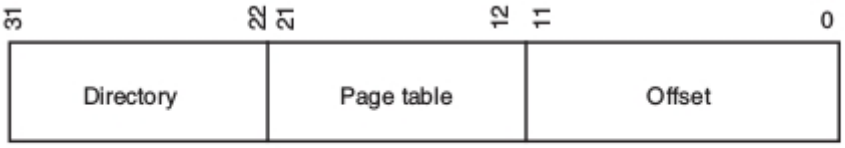
La dirección física está compuesta por la dirección física del Frame Page que se obtiene de la page table concatenada con el offset de la página que se obtiene de la virtual address. El sistema operativo maneja los accesos a la memoria.



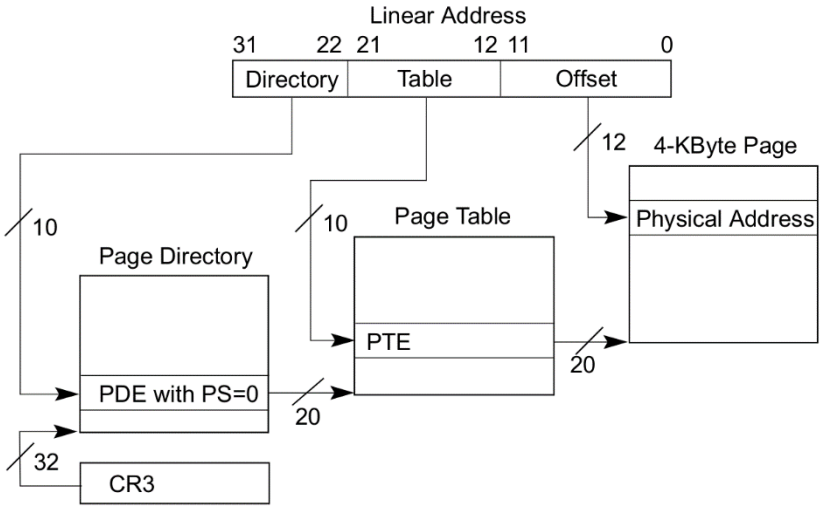
Memoria Paginada en x86

Virtual Address x86

- Page Directory: bits 31-22 (10 bits)
- Page Table Entry bits 21-12 (10 bits)
- Memory Page offset address bits 11-0 (12 bits)

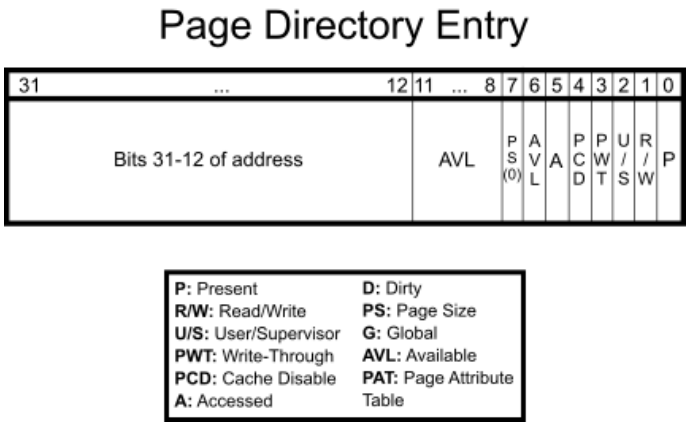


(a)



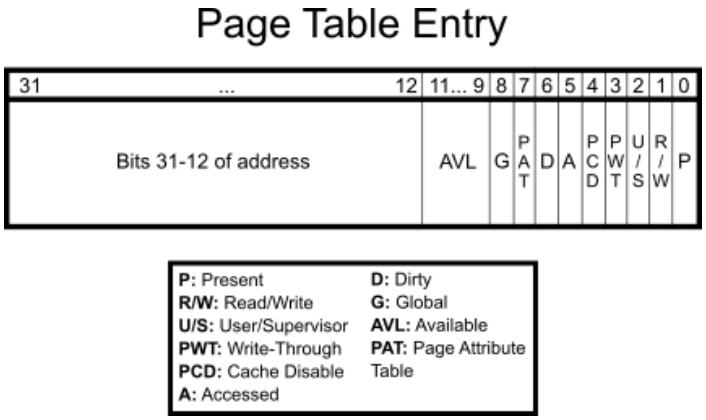
Page Directory Entry x86

Una entrada de la page directory ocupa 4 bytes. Posee 1024 entradas.



Page Table Entry x86

Una entrada de la page table, ocupa 4 bytes. Posee 1024 entradas.

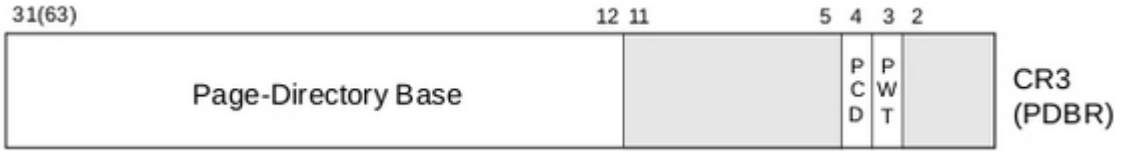


Registros importantes CR0 y CR3 x86

CR3:

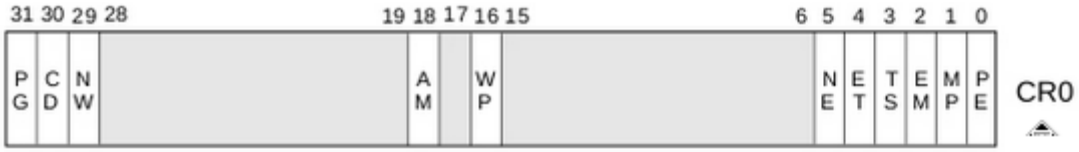
- Función Principal: El registro CR3 se utiliza para apuntar a la tabla de directorios de páginas en la memoria física. La tabla de directorios de páginas es la estructura de datos de nivel superior en el mecanismo de paginación de x86, que a su vez apunta a tablas de páginas individuales.
- Papel en la Paginación: Cuando la paginación está habilitada (es decir, el bit PG en CR0 está establecido), cada vez que el procesador necesita traducir una dirección virtual a una dirección física, consulta la estructura de paginación que comienza en la dirección física especificada en CR3.
- Cambios en CR3: Al cambiar el valor de CR3 (por ejemplo, al cambiar de contexto o al cambiar a un proceso diferente), efectivamente se está cambiando el espacio de direcciones virtual activo, ya que se está apuntando a una tabla de directorios de páginas diferente.
- Invalidación de TLB: Cada vez que se escribe en CR3, la caché de la tabla de búsqueda (TLB) se invalida automáticamente. Esto se debe a que la TLB podría contener entradas antiguas basadas en la antigua estructura de paginación, y al cambiar CR3, estas entradas ya no serían válidas.

- Formato: La dirección almacenada en CR3 debe estar alineada a una página, lo que significa que los bits inferiores de CR3 (que especificarían un desplazamiento dentro de una página) son cero y no se utilizan en la dirección. Estos bits inferiores, sin embargo, tienen otros usos en versiones más recientes de la arquitectura x86.



CR0:

- Función Principal: El registro CR0 alberga varios flags que controlan cómo opera el procesador en varios aspectos. Es especialmente importante para habilitar o deshabilitar la paginación y el modo protegido.
- Bits Principales:
    - PE (Bit 0, Modo Protegido): Cuando este bit está establecido, el procesador opera en modo protegido. De lo contrario, opera en modo real.
    - WP (Bit 16, Protección de Escritura): Cuando está establecido, determina el comportamiento de las páginas de solo lectura en modo supervisor.
    - PG (Bit 31, Paginación): Cuando este bit está establecido, la paginación está habilitada. Si está desactivado, el procesador usa una traducción de dirección lineal a física directa.



Caché

El tema es que muchos de estos métodos tienen varios niveles, hasta 4 en algunos casos, para alcanzar una dirección física, entonces eso lo hace realmente poco práctico para el procesador.

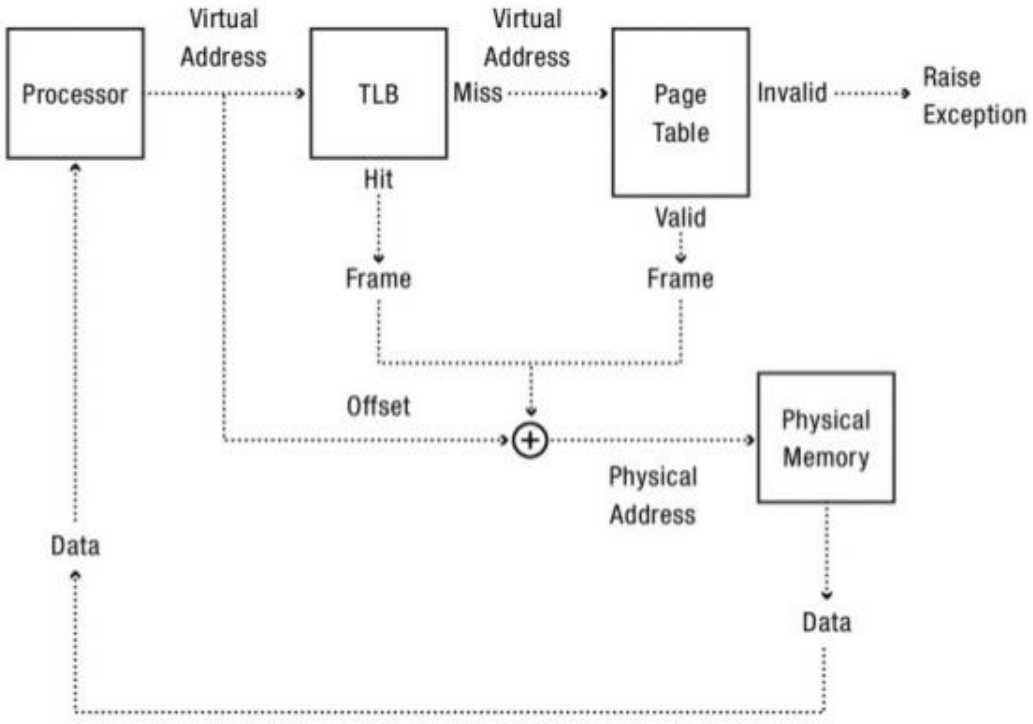
TLB

Para mejorar el Address Translation se utiliza un mecanismo de hardware llamado **Translation-Lookaside Buffer**; o también conocido como **TLB**.

La TLB es parte de la MMU y es simplemente un mecanismo de cache de las traducciones más utilizadas entre los pares virtual y physical Address. Por ende un mejor nombre para este mecanismo podría ser **Address Translation cache**.

Por cada referencia a la memoria virtual, el hardware primero chequea la TLB para ver si esa traducción esta guardada ahí; si es así la traducción se hace rápidamente sin tener que consultar a la page table (la cual tiene todas las traducciones).

Normalmente se chequean todas las entradas de la TLB contra la virtual page, si existe matcheo el procesador utiliza ese matcheo para formar la physical Address, ahorrándose todos los pasos de la traducción. Esto se llama un **TLB hit**. Cuando del proceso anterior no existe matcheo en la TLB, se dice que se tiene un **TLB miss**.



TLB flush:

En un Context Switch, las direcciones virtuales del viejo proceso ya no son más válidas, y no deben ser válidas, para el nuevo proceso. De otra forma, el nuevo proceso sería capaz de leer las direcciones del viejo proceso. Frente a un context switch, se necesita descartar el contenido de TLB en cada context switch. Este approach se denomina **flush de TLB**. Debido a que este proceso acarrearía una penalidad, los procesadores taguean la TLB de forma tal que la misma contenga el id del proceso que produce cada transacción.

TLB shutdown:

En un sistema multiprocesador cada uno puede tener cacheada una copia de una transacción en su TLB. Por ende, para seguridad y correctitud, cada vez que una entrada en la page table es modificada, la correspondiente entrada en todas las TLB de los procesadores tiene que ser descartada antes que los cambios tomen efecto.

Típicamente sólo el procesador actual puede invalidar su propia TLB, por ello, para eliminar una entrada en todos los procesadores del sistema, se requiere que el sistema

operativo mande una interrupción a cada procesador y pida que esa entrada de la TLB sea eliminada. Esta es una operación muy costosa y por ende tiene su propio nombre y se denomina **TLB shutdown**.

FILE SYSTEM

¿Qué es un File System?

Una abstracción del sistema operativo que provee datos persistentes con un nombre.

Partes Fundamentales

- Archivos: compuestos por un conjunto de datos.
- Directorios: definen nombres para los archivos.

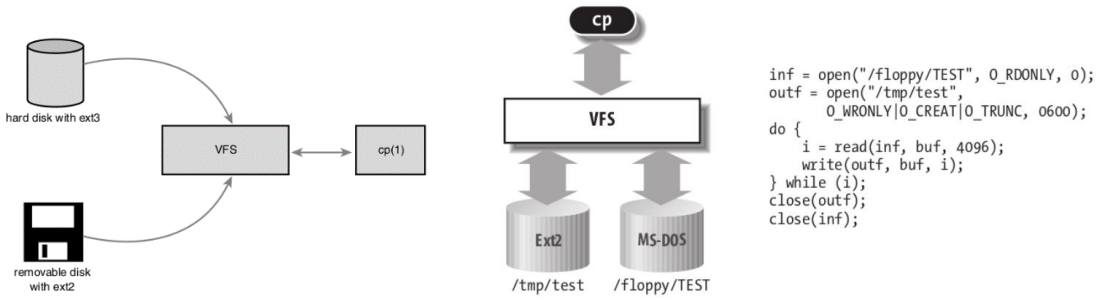
Virtual File System

El **Virtual Files System (VFS)** es el subsistema del kernel que implementa la interfaz que tiene que ver con los archivos y el sistema de archivos provistos a los programas corriendo en modo usuario.

Todos los sistemas de archivos deben basarse en VFS para:

- Coexistir
- interoperar

Esto habilita a los programas a utilizar las system calls de unix para leer y escribir en diferentes sistemas de archivos y diferentes medios.



VFS es el pegamento que habilita a las system calls como por ejemplo open(), read() y write() a funcionar sin que estas necesitan tener en cuenta el hardware subyacente

FileSystem Abstraction Layer

Es un tipo genérico de interfaz para cualquier tipo de FileSystem que es posible sólo porque el kernel implementa una capa de abstracción que rodea esta interfaz para con el sistema de archivo de bajo nivel.

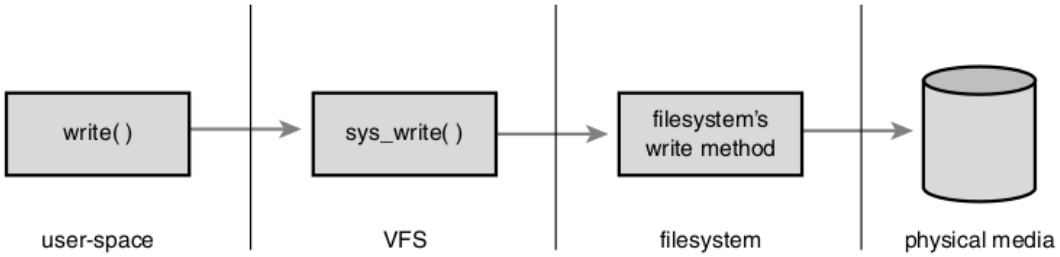
Esta capa de abstracción habilita a Linux a soportar sistemas de archivos diferentes, incluso si estos difieren en características y comportamiento.

Esto es posible porque VFS provee un modelo común de archivos que pueda representar cualquier característica y comportamiento general de cualquier sistema de archivos.

Esta capa de abstracción trabaja mediante la definición de interfaces conceptualmente básicas y de estructuras que cualquier sistema de archivos soporta.

Todos estos sistemas de archivos soportan nociones tales como archivos, directorios y además todos soportan un conjunto de operaciones básicas sobre estos.

El resultado es una capa de abstracción general que le permite al kernel manejar muchos tipos de sistemas de archivos de forma fácil y limpia.

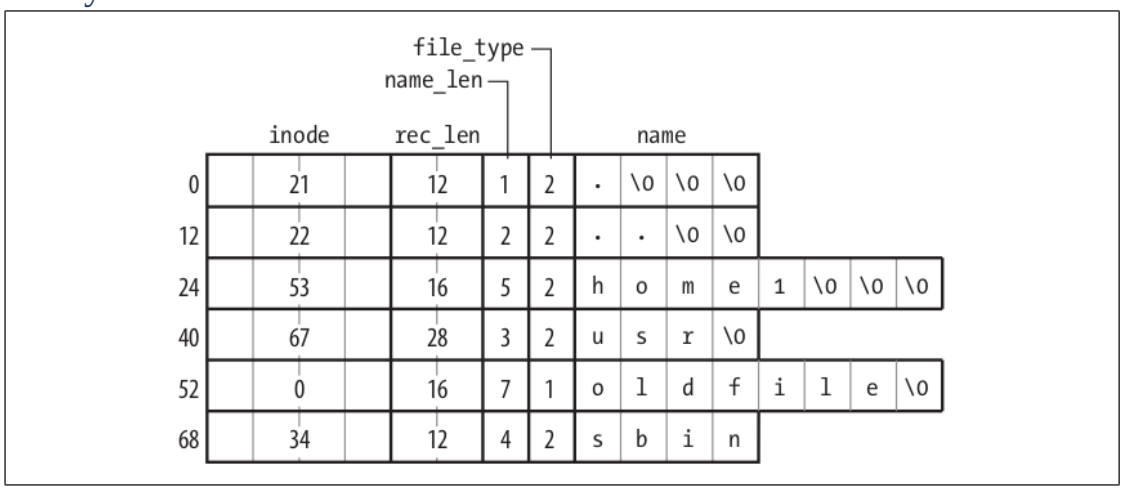


Estructuras del VFS

VFS presenta una serie de estructuras que modelan un FileSystem, estas estructuras se denominan objetos (programadas en C).

- **Super bloque**: representa a un sistema de archivos.
- **Inodo**: representa a un determinado archivo.
- **Dentry**: representa una entrada de directorio, que es un componente simple de un path.
- **File**: representa a un archivo asociado a un determinado proceso.

Dentry



Un directorio es tratado como un archivo normal, no hay un objeto específico para directorios. En unix los directorios son archivos normales que listan los archivos contenidos en ellos. :: /home/darthmendez/hola.txt

Archivo

Un archivo es una colección de datos con un nombre específico.

Metadata: información acerca del archivo que es comprendida por el Sistema Operativo, esta información es:

- tamaño
- fecha de modificación
- propietario
- información de seguridad (qué se puede hacer con el archivo).

Datos: son los datos propiamente dichos que quieren ser almacenados. Desde el punto de vista del Sistema Operativo, un archivo o file no es más que un arreglo de bytes sin tipo.

FileDescriptors

Para cada proceso, el kernel mantiene una tabla de open file-descriptors. Cada entrada de esta tabla registra la información sobre un único fd:

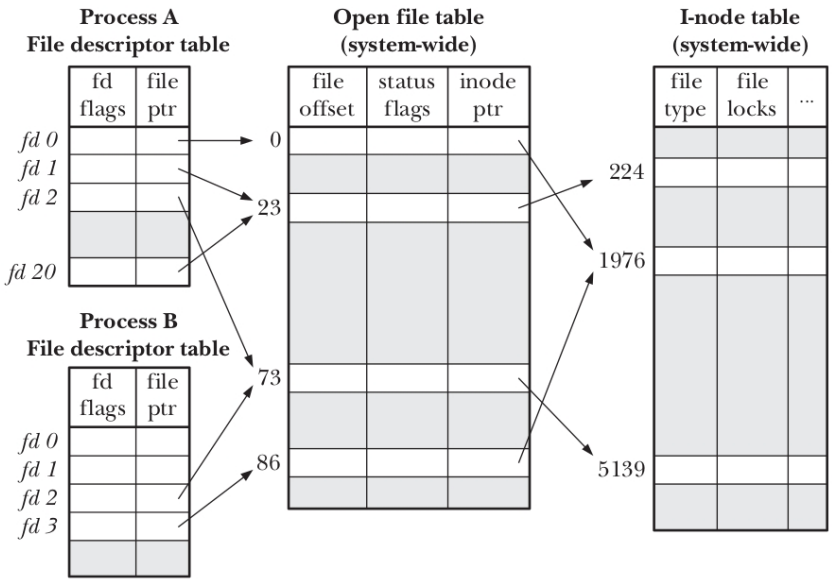
- Un conjunto de flags que controlan las operaciones del fd;
- Una referencia al open file descriptor.

Por otro lado, el kernel mantiene una tabla general para todo el systema de todos los open file descriptor (también conocida como la open files table), esta tabla almacena:

- El offset actual del archivo (que se modifica por read(), write() o por lseek();
- Los flags de estado que se especificaron en la apertura del archivo (los flags arguments de open());
- El modo de acceso (solo lectura, solo escritura, escritura-lectura);
- una referencia al objeto i-nodo para este archivo.

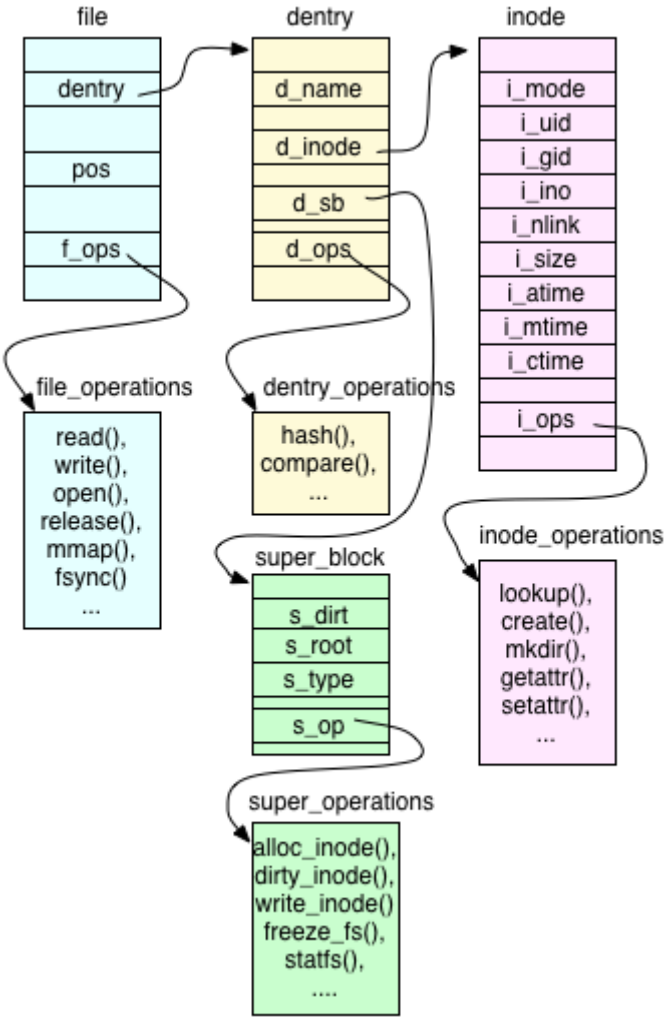
Además cada sistema de archivos posee una tabla de todos los i-nodos que se encuentran en el sistema de archivos. En esta tabla se almacenará:

- El tipo de archivo;
- Un puntero a la lista de los locks que se mantienen sobre ese archivo;
- Otras propiedades del archivo.



Operaciones del VFS

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



EL API (Unix File Systems System Calls)

Las System Calls de archivos pueden dividirse en dos clases:

- Las que operan sobre los **archivos** propiamente dichos.
- Las que operan sobre los **metadatos** de los archivos.

Syscalls sobre Archivos

open()

La System Call open() convierte el nombre de un archivo en una entrada de la tabla de descriptores de archivos, y devuelve dicho valor. Siempre devuelve el descriptor más pequeño que no está abierto.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

creat()

La System Call creat() equivale a llamar a open() con los flags O_CREAT|O_WRONLY|O_TRUNC.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int creat(const char *pathname, mode_t mode);
```

close()

La System Call close cierra un file descriptor. Si este ya está cerrado devuelve un error.

```c
#include <unistd.h>

int close(int fd);
```

read()

La llamada read se utiliza para hacer intentos de lecturas hasta un número dado de bytes de un archivo. La lectura comienza en la posición señalada por el file descriptor, y tras ella se incrementa ésta en el número de bytes leídos.

```c
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

write()

La System Call write() escribe hasta una determinada cantidad (count) de bytes desde un buffer que comienza en buf al archivo referenciado por el file descriptor.

```c
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

lseek()

La System Call lseek() reposiciona el desplazamiento (offset) de un archivo abierto cuyo file descriptor es fd.

De acuerdo con el parámetro whence (de donde):

- SEEK_SET: el desplazamiento.
- SEEK_CUR: el desplazamiento es sumado a la posición actual del archivo.
- SEEK_END: el desplazamiento se suma a partir del final del archivo.

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

off_t lseek(int fd, off_t offset, int whence);
```

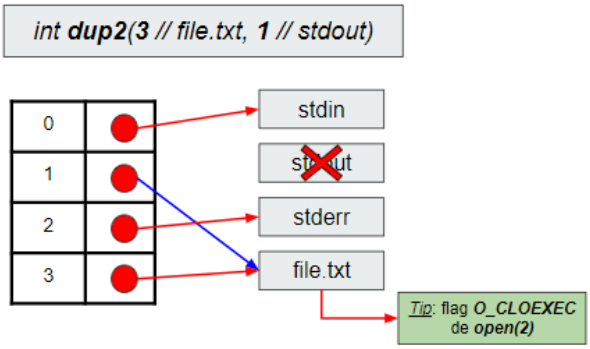
dup() y dup2()

Esta System Call crea una copia del file descriptor del archivo cuyo nombre es oldfd.

Después de que retorna en forma exitosa, el viejo y nuevo file descriptor pueden ser usados de forma intercambiable. Estos se refieren al mismo archivo abierto y por ende comparten el offset y los flags de estado.

dup2() hace lo mismo pero en vez de usar la política de seleccionar el file descriptor más pequeño utiliza a newfd como nuevo file descriptor.

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```



link()

La System Call link() crea un nuevo nombre para un archivo. Esto también se conoce como un link (hard link).

Nota: Este nuevo Nombre puede ser usado exactamente como el viejo nombre para cualquier operación, es más ambos nombres se refieren exactamente al mismo archivo y es imposible determinar cuál era el nombre original.

```c
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

unlink()

Esta System Call elimina un nombre de un archivo del sistema de archivos. Si además ese nombre era el último nombre o link del archivo y no hay nadie que tenga el archivo abierto lo borra completamente del sistema de archivos.

```c
#include <unistd.h>

int unlink(const char *pathname);
```

Syscalls sobre Directorios

mkdir()

Esta syscall crea un directorio.

```
#include <sys/stat.h>
include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
```

rmdir()

Esta syscall elimina un directorio.

```
#include <unistd.h>

int rmdir(const char *pathname);
```

opendir()

La función opendir abre y devuelve un stream que corresponde al directorio que se está leyendo en dirname. El stream es de tipo DIR *

```
#include <sys/types.h>
#include <dirent.h>
DIR * opendir (const char *dirname)
```

readdir()

Esta función lee la próxima entrada de un directorio. Normalmente devuelve un puntero a una estructura que contiene la información sobre el archivo.

```
#include <sys/types.h>
#include <dirent.h>
struct dirent * readdir (DIR *dirstream)
```

Esta es la estructura de datos provista para poder leer las entradas a los directorios.

- char d_name[]: es el componente del nombre null-terminated. Es el único campo que está garantizado en todos los sistemas posix
- ino_t d_fileno: es el número de serie del archivo.

```
struct dirent {
    ino_t d_fileno;               // i-node nr.
    char d_name[MAXNAMLEN + 1];   // file name
}
```

closedir()

Cierra el stream de tipo DIR * cuyo nombre es dirstream.

```c
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dirstream)
```

Syscalls sobre los metadatos

stat()

Esta familia de System Calls devuelven información sobre un archivo, en el buffer apuntado por statbuf. No se requiere ningún permiso sobre el archivo en cuestión, pero sí en los directorios que conforman el path hasta llegar al archivo.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

La estructura de datos apuntada por statbuf se describe de la siguiente manera:

```
struct stat {
   dev_t     st_dev;         /* ID of device containing 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 */

  /* Since Linux 2.6, the kernel supports nanosecond
     precision for the following timestamp fields.
     For the details before Linux 2.6, see NOTES. */

  struct timespec st_atim;  /* Time of last access */
  struct timespec st_mtim;  /* Time of last modification */
  struct timespec st_ctim;  /* Time of last status change */

  #define st_atime st_atim.tv_sec      /* Backward compatibility */
  #define st_mtime st_mtim.tv_sec
  #define st_ctime st_ctim.tv_sec
};
```

access()

La System Call acces chequea si un proceso tiene o no los permisos para utilizar el archivo con un determinado pathname. El argumento mode determina el tipo de permiso a ser chequeado.

El modo (mode) especifica el tipo de accesibilidad a ser chequeada, los valores pueden conjugarse como una máscara de bits con el operador |:

- F_OK: el archivo existe.
- R_OK: el archivo puede ser leído.
- W_OK: el archivo puede ser escrito.
- X_OK: el archivo puede ser ejecutado.

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

chmod()

Estas System Calls cambian los bits de modos de acceso.

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

chown()

Estas system Calls cambian el id del propietario del archivo y el grupo de un archivo.

```c
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Comando ls

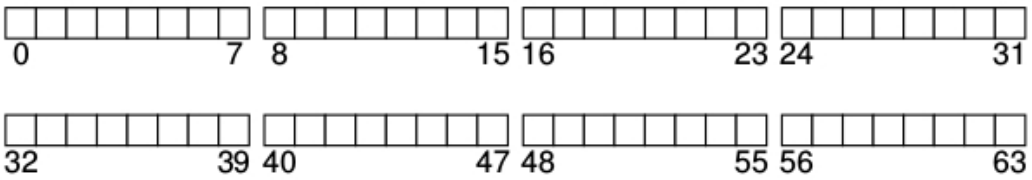
```c
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int
main (void)
{
    DIR *dp;
    struct dirent *ep;

    dp = opendir ("./");
    if (dp != NULL)
    {
        while (ep = readdir (dp))
        puts (ep->d_name);
    (void) closedir (dp);
    }
    else
        perror ("Couldn't open the directory");

    return 0;
}
```

Very Simple File System

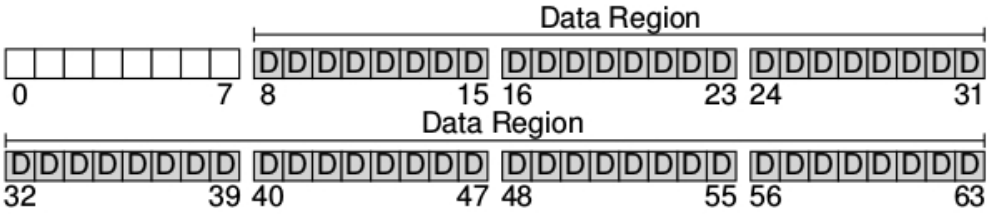
Este file system es una versión simplificada de un típico sistema de archivos unix-like. Existen diferentes sistemas de archivos y cada uno tiene ventajas y desventajas.

Organización general

1. Lo primero que se debe hacer es dividir al disco en bloques, los sistemas de archivos simples, como este suelen tener bloques de un solo tamaño. Los bloques tienen un tamaño de 4 Kbytes
   La visión del sistema de archivos debe ser la de una partición de N bloques (de 0 a N-1) de un tamaño de N * 4 KB bloques. Si suponemos en un disco muy pequeño, de unos 64 bloques, este podría verse así:

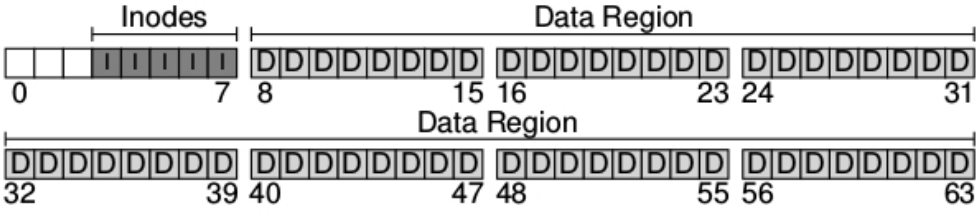


2. A la hora de armar un sistema de archivos es necesario almacenar los datos, de hecho la mayor cantidad del espacio ocupado en un file system es por los datos de usuarios. Esta región se llama por ende data-region.



3. El sistema de archivos debe mantener información sobre cada uno de estos archivos. Esta información es la Metadata y es de vital importancia ya que mantiene información como: qué bloque de datos pertenece a un determinado archivo, el tamaño del archivo, etc. Para guardar esta información, en los sistemas operativos unix-like, se almacena en una estructura llamada inodo.

   Los inodos también deben guardarse en el disco, para ello se los guarda en una tabla llamada inode table que simplemente es un array de inodos almacenados en el disco



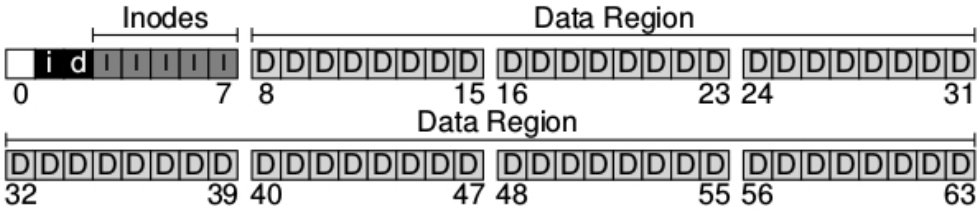
   Los inodos no son estructuras muy grandes, normalmente ocupan unos 128 o 256 bytes.

Suponiendo que los inodos ocupan 256 bytes, un bloque de 4KB puede guardar 16 inodos por ende nuestro sistema de archivo tendrá como máximo 80 inodos. Esto representa también la cantidad máxima de archivos que podrá contener nuestro sistema de archivos.

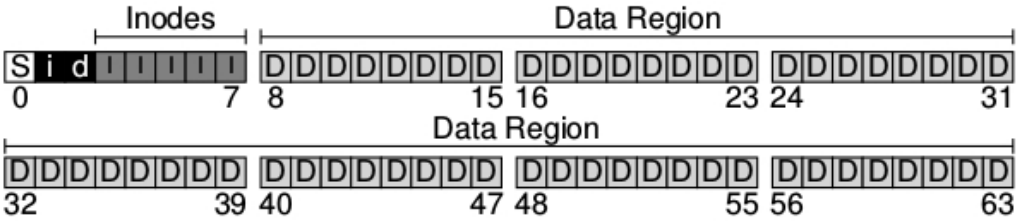
4. El sistema de archivos tiene los datos (D) y los inodos (I).

   Una de las cosas que faltan es saber qué inodos y qué bloques están siendo utilizados o están libres. Esta estructura de alocación es fundamental en cualquier sistema de archivos. Existen muchos métodos para llevar este registro pero en este caso se utilizará una estructura muy popular llamada bitmap. Una para los datos data bitmap ora para los inodos inode bitmap.

   Un bitmap es una estructura bastante sencilla en la que se mapea 0 si un objeto está libre y 1 si el objeto está ocupado. En este ejemplo, i sería el bitmap de inodos y d sería el bitmap de datos:



5. Queda un único bloque libre en todo el disco. Este bloque es llamado Super Bloque (S). El SuperBloque contiene la información de todo el file system, incluyendo:
   - cantidad inodos
   - cantidad de bloques
   - donde comienza la tabla de inodos –> bloque 3
   - donde comienzan los bitmaps



Inodos

Esta es una de las estructuras almacenadas en el disco más importantes. Casi todos los sistemas de archivos unix-like son así.

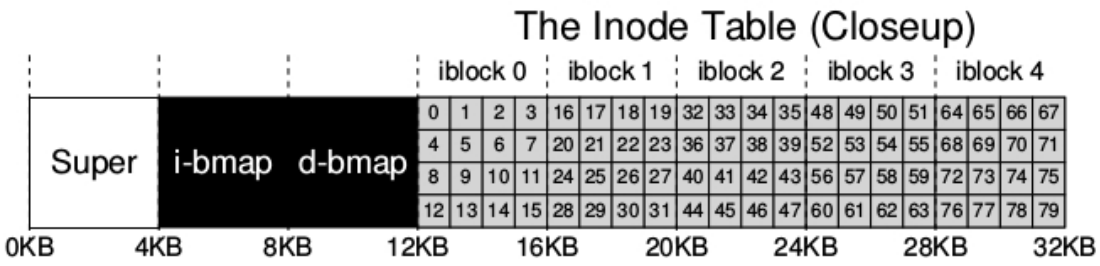
Un inodo simplemente es referido por un número llamado inumber que sería lo que hemos llamado el nombre subyacente en el disco de un archivo. En este sistema de archivos y en varios otros, dado un inumber se puede saber directamente en que parte del disco se encuentra el inodo correspondiente

Para leer el inodo número 32, el sistema de archivos debe:

1. Calcular el offset en la región de inodos 32*sizeof(inode) = 8192

2. Sumarlo a la dirección inicial de la inode table en el disco o sea 12Kb + 8192 bytes

3. llegar a la dirección en el disco, que es la 20 KB.



FAT/FAT-32

Microsoft File Allocation Table (FAT) este file system se implementó en los 70, Fue el Sistema de archivos de MS-DOS y de las versiones tempranas de Windows.

FAT fue mejorado por su versión FAT-32, el cual soporta volúmenes de 2^{32}-1 bytes.

FAT obtiene su nombre de la file allocation table, un arreglo de entradas de 32 bits, en un área reservada del volumen.

Cada archivo en el sistema corresponde a una lista enlazada de entradas en la FAT, en la que cada entrada en la FAT contiene un puntero a la siguiente entrada.

La FAT contiene una entrada por cada bloque de la unidad de disco o volumen.

Los directorios asignan a los nombres de archivo a números de archivo, y en el sistema de archivos FAT, el número de un archivo es el índice de la primera entrada del archivo en la FAT.

Así, dado el número de un archivo, podemos encontrar la primera entrada FAT y bloque de un archivo, y dada la primera entrada FAT, podemos encontrar el resto de las entradas y bloques FAT del archivo.

Seguimiento de espacio libre: La FAT también se utiliza para el seguimiento del espacio libre. Si el bloque de datos i está libre, entonces FAT[i] contiene 0. Por lo tanto, el sistema de archivos puede encontrar un bloque libre escaneando a través de la FAT para encontrar una entrada puesta a cero.

Las estrategias de asignación de las implementaciones FAT suelen ser simples. Por ejemplo, algunas implementaciones usan un algoritmo de ajuste siguiente que escanea secuencialmente a través de la FAT a partir de la última entrada que se asignó y que devuelve la siguiente entrada libre encontrada. Pueden fragmentar un archivo, esparciendo los bloques del archivo el volumen en lugar de lograr el diseño secuencial deseado.

FFS: Fixed Tree

El Fast File System de Unix (FFS) ilustra ideas importantes tanto para indexar la información de un archivo de bloques para que puedan ubicarse rápidamente y para colocar datos en el disco para obtener una buena ubicación.

En particular, la estructura del índice de FFS, llamado índice multinivel, es un árbol cuidadosamente estructurado que permite a FFS localizar cualquier bloque de un archivo y que es eficiente tanto para grandes como para pequeños archivos

Dada la flexibilidad proporcionada por el índice multinivel de FFS, FFS emplea dos localidades heurísticas (colocación de grupos de bloques y espacio de reserva) que juntas suelen proporcionar buen diseño en disco.

El inodo (raíz) de un archivo también contiene una serie de punteros para ubicar los bloques de datos del archivo (hojas). Algunos de estos punteros apuntan directamente a las hojas de datos del árbol y algunos de ellos apuntan a nodos internos en el árbol. Normalmente, un inodo contiene 15 punteros:

- Los primeros 12 son directos y apuntan directamente a los primeros 12 bloques de datos de un archivo.
- El puntero 13 es un puntero indirecto, que apunta a un nodo interno del árbol llamado bloque indirecto (bloque normal de almacenamiento que contiene una matriz de punteros directos).
  Para leer el bloque 13, primero se lee el inodo para obtener el puntero indirecto, luego el bloque indirecto para obtener el puntero directo, luego el bloque de datos.

  Con bloques de 4KB y punteros de bloque de 4 bytes, un bloque indirecto puede contener hasta 1024 punteros, lo que permite archivos de hasta un poco más de 4MB.

  Cálculo:

  *4 KB → 4 * 1024 bytes → 2^{12} bytes. (almacenamiento total de un bloque)*

  Si divido por el peso de cada puntero (4 bytes) → bloque tiene 2^{12} / 2^{2} punteros.

  *Si el bloque apuntado por el puntero indirecto tiene 1024 punteros directos a 1024 bloques de datos y cada bloque pesa 4KB en total se tiene 1024 * 4 * 1024*

  *1024 * 4 *1024 = 2^{22} = 4Mb*

- El puntero 14 es un puntero indirecto **doble**, que apunta a un nodo interno del árbol llamado doble bloque indirecto. Un bloque indirecto doble es una matriz de puntero indirectos, cada uno de los cuales apunta a un bloque indirecto. Con bloques de 4KB y punteros de bloque de 4bytes.

  El bloque indirecto puede contener hasta 1024 punteros indirectos. Así, un puntero indirecto doble puede indexar hasta 1024^2 bloques de datos.

Cálculo:

*4 KB → 4 * 1024 bytes → 2^{12} bytes. (almacenamiento total de un bloque)*

Si divido por el peso de cada puntero (4 bytes) → bloque tiene 2^{12} / 2^{2} punteros.

*Si el bloque apuntado por el puntero indirecto doble tiene 1024 punteros indirectos a 1024 bloques indirectos y cada bloque indirecto tiene 1024 punteros a bloques de datos que pesan 4KB en total se tiene 1024 * 1024 * 4 * 1024*

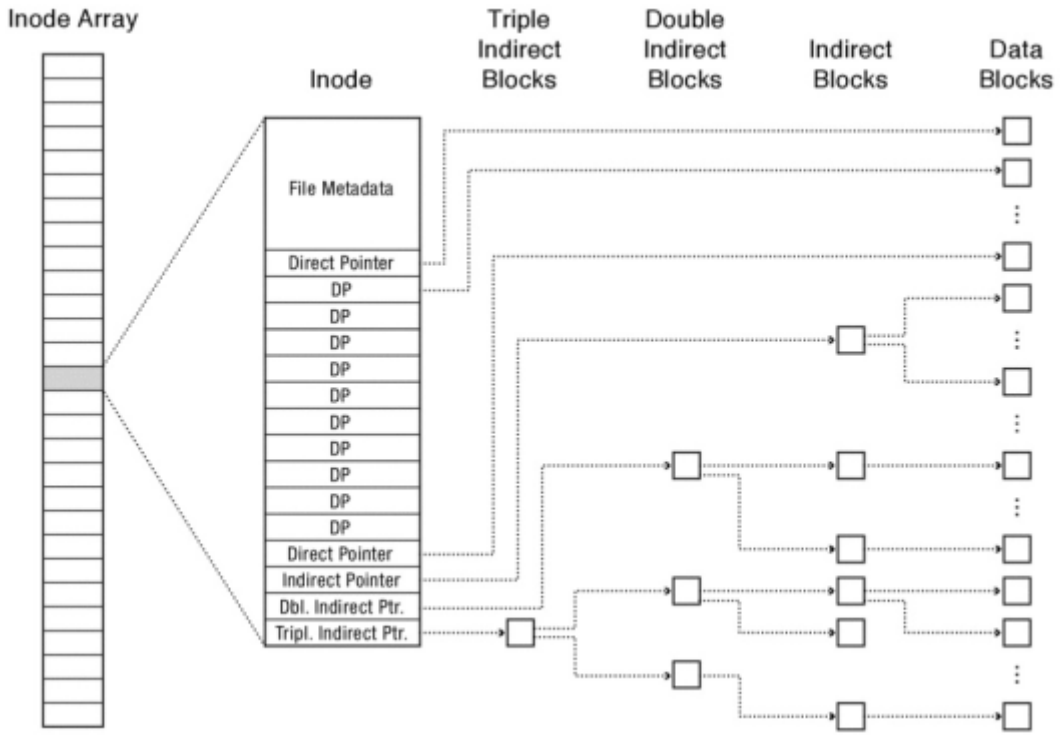
*1024 * 1024 * 4 *1024 = 2^{32} = 4Gb*

- El puntero 15 es un puntero indirecto **triple** que apunta a un bloque indirecto triple que contiene una matriz de puntero indirectos dobles.
  Con bloques de 4KB y punteros de bloque de 4bytes, un puntero indirecto triple puede indexar hasta 1024^3 bloques de datos que contienen: 4KB * 1024^3 = 212 * 230 = 242 bytes (4TB).

  Cálculo:

  *Si el bloque apuntado por el puntero indirecto triple tiene 1024 punteros indirectos dobles a 1024 bloques con 1024 punteros indirectos cada uno que apuntan a bloques indirectos que tienen 1024 punteros a bloques de datos que pesan 4KB en total se tiene 1024 * 1024 * 1024 * 4 * 1024*

  *1024 * 1024 * 1024 * 4 *1024 = 2^{42} = 4Tb*



El Fast File System de Unix (FFS) ilustra ideas importantes tanto para indexar la información de un archivo bloques para que puedan ubicarse rápidamente y para colocar datos en el disco para obtener una buena ubicación.

En particular, la estructura del índice de FFS, llamada índice multinivel, es un árbol cuidadosamente estructurado que permite a FFS localizar cualquier bloque de un archivo y que es eficiente tanto para grandes como para pequeños archivos

Dada la flexibilidad proporcionada por el índice multinivel de FFS, FFS emplea dos localidades heurísticas (colocación de grupos de bloques y espacio de reserva) que juntas suelen proporcionar buen diseño en disco.

SCHEDULING

¿Qué es el Scheduling?

Debe existir algún mecanismo que permita determinar cuánto tiempo de CPU le toca a cada proceso. Ese período de tiempo que el kernel le otorga a un proceso se denomina **time slice o time quantum**.

Multiprogramación

Más de un proceso estaba preparado para ser ejecutado en algún determinado momento, y el sistema operativo intercalaba dicha ejecución según la circunstancia.

Cuando un Sistema Operativo se dice que realiza multi-programación de varios procesos debe existir una entidad que se encargue de coordinar la forma en que estos se ejecutan, el momento en que estos se ejecutan y el momento en el cual paran de ejecutarse. En un sistema operativo esta tarea es realizada por el Planificador o **Scheduler** que forma parte del Kernel del Sistema Operativo.

Time Sharing

Tiempo compartido se refiere a compartir de forma concurrente un recurso computacional entre muchos usuarios por medio de las tecnologías de multi-programación y la inclusión de interrupciones de reloj por parte del SO.

Métricas de Planificación

Workload

Carga de trabajo de un proceso corriendo en el sistema.

Turnaround time

El tiempo en el cual el proceso se completa menos el tiempo de arribo al sistema.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Mide performance.

Response time

El tiempo de respuesta. Es el tiempo desde que arriba hasta que se ejecuta un proceso.

$$T_{response} = T_{firstrun} - T_{arrival}$$

Mide performance.

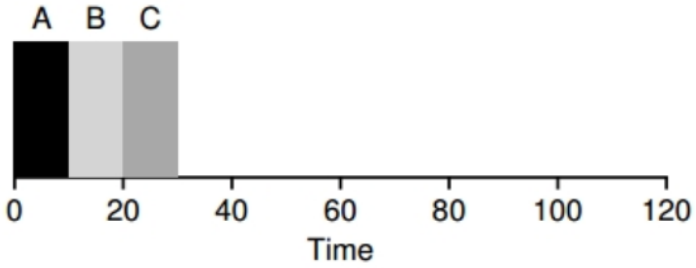
Políticas de Scheduling Mono Core

- Fist In, First Out (FIFO)
- Shortest Job First (SJF)
- Shortest Time-to-Completion (STCF)
- Round Robin (RR)

FIFO

El algoritmo más básico. El primero que llega es el primero que se ejecuta.

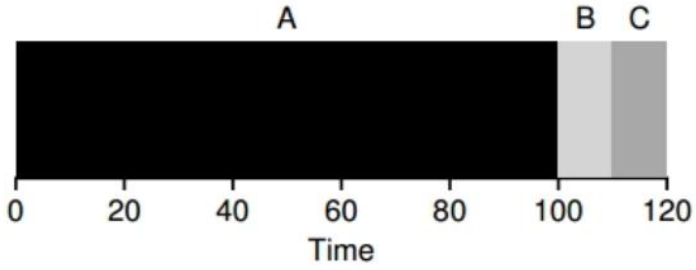
Cálculo de $T_{turnaround}$



Si todos tardan 10 segundos en ejecutarse y llegan al mismo tiempo ($T_{arrival} = 0$)

$$T_{turnaround} = \frac{(10 + 20 + 30)}{3} = 20$$

Problema: Si el primero que se ejecuta es muy largo, $T_{turnaround}$ es muy alto.



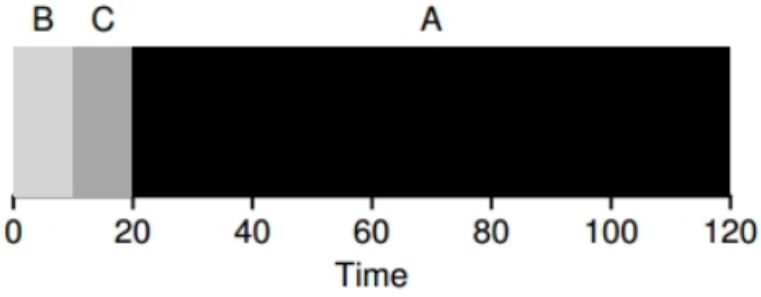
Si todos no duran lo mismo (A dura 100 segundos)

$$T_{turnaround} = \frac{(100 + 110 + 120)}{3} = 110$$

Shortest Job First (SJF)

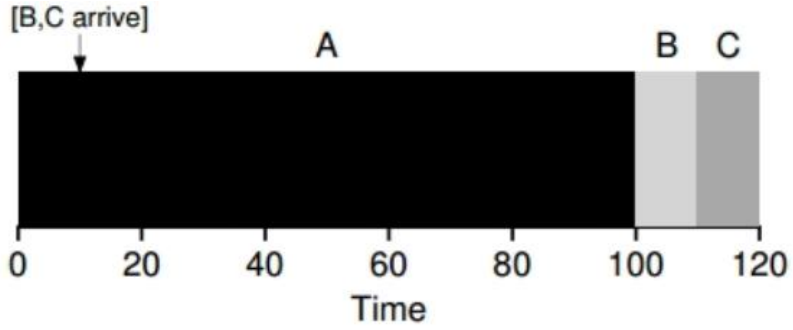
Para resolver el problema de FIFO, se ejecutan primero los procesos de duración mínima.

Cálculo de $T_{turnaround}$



$$T_{turnaround} = \frac{(10 + 20 + 120)}{3} = 50$$

Si se supone que no todos los procesos llegan al mismo tiempo ($T_{arrival} \neq 0$)



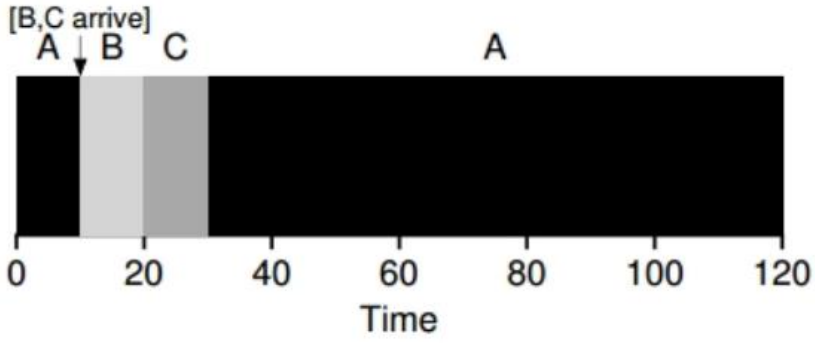
El $T_{arrival}$ de B y C es 10.

$$T_{turnaround} = \frac{[(100-0)+(110-10)+(120-10)]}{3} = 103{,}33$$

Shortest Time-To-Completion (STCF)

Para resolver SJF el planificador puede adelantarse y determinar qué proceso debe ser ejecutado. Entonces cuando B y C llegan, se puede desalojar (Preempt) al proceso A y decidir que otro proceso se ejecute y luego retomar la ejecución del proceso A.

Calculo de $T_{turnaround}$ y $T_{response}$



$$T_{turnaround} = \frac{[(120-0)+(20-10)+(30-10)]}{3} = 50$$

$T_{responseA} = 0$ (Llega y se ejecuta)

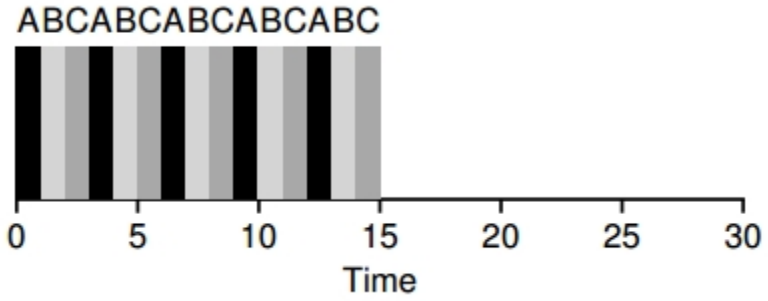
$T_{responseA} = 0$ (Llega en 10 y se ejecuta en 10)

$T_{responseA} = 10$ (Llega en 10 y se ejecuta en 20)

El promedio de $T_{response}$ es de 3,33 segundos.

Round Robin

La idea del algoritmo 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.



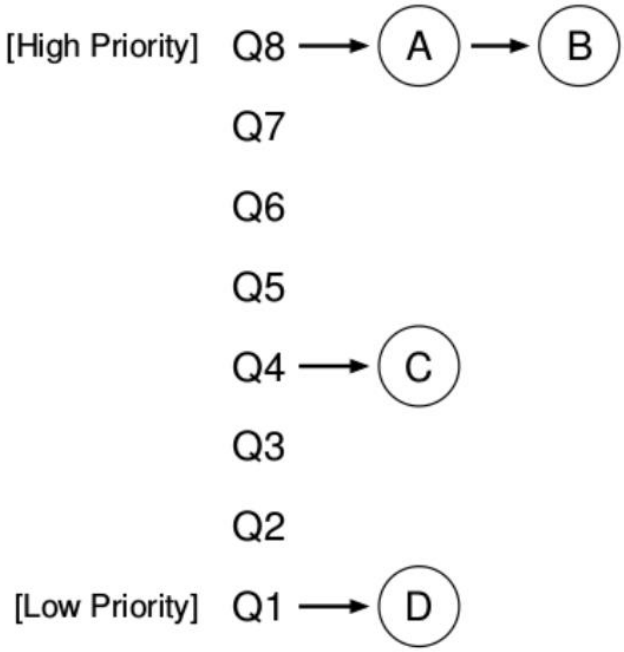
Lo importante de RR es la elección de un buen time slice, se dice que el time slice tiene que amortizar el cambio de contexto sin hacer que esto resulte en que el sistema no responda más.

Multi Level Feedback Queue

Tiene un conjunto de distintas colas, cada una tiene un nivel de prioridad.

Usa las prioridades para determinar qué proceso debe ejecutarse en un determinado tiempo.

Si hay más de un proceso en la cola de prioridad más alta ejecuta Round Robin entre ellos.



Reglas básicas:

1. Si la prioridad de A es mayor que la de B, A se ejecuta y B no.
2. Si la prioridad de A es igual a la de B, A y B se ejecutan en Round-Robin.
3. Cuando un proceso entra en el sistema se pone en la prioridad más alta.
4. Una vez que una tarea usa su asignación de tiempo (slice) de un nivel dado (independientemente de cuantas veces haya renunciado al uso de la CPU) su prioridad se reduce (baja un nivel)
5. Después de cierto tiempo S, se mueven todos los procesos a la cola con más prioridad.
    - Si el valor de S es muy alto, los procesos que requieren mucha ejecución caen en Starvation
    - Si el valor de S es muy chico, las tareas interactivas no van a poder compartir adecuadamente la CPU.

Prioridades

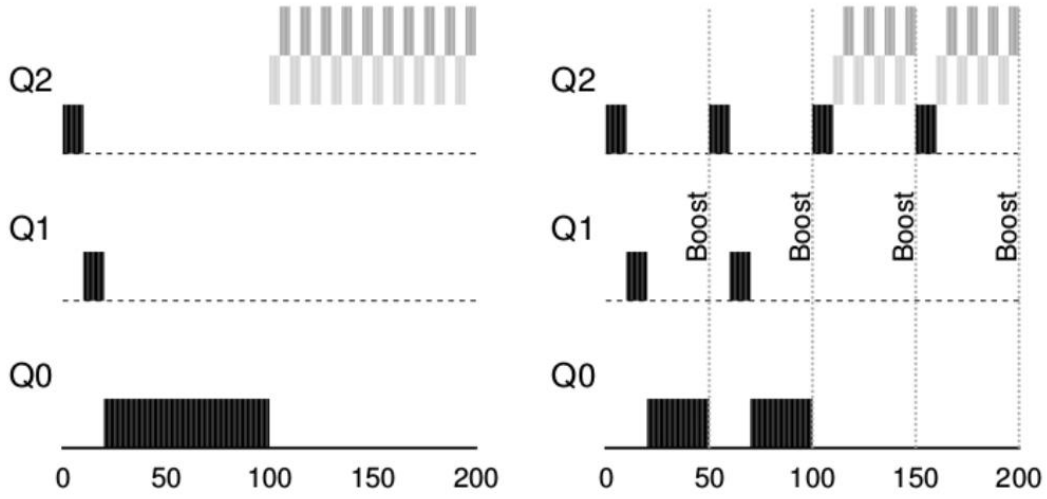
Las prioridades varían basándose en el comportamiento observado de los procesos:

- Si un proceso no utiliza la CPU mientras espera instrucciones por teclado, MLFQ va a mantener su prioridad alta.
- Si un proceso usa intensivamente por largos periodos de tiempo la CPU, MLFQ reducirá su prioridad.

Starvation

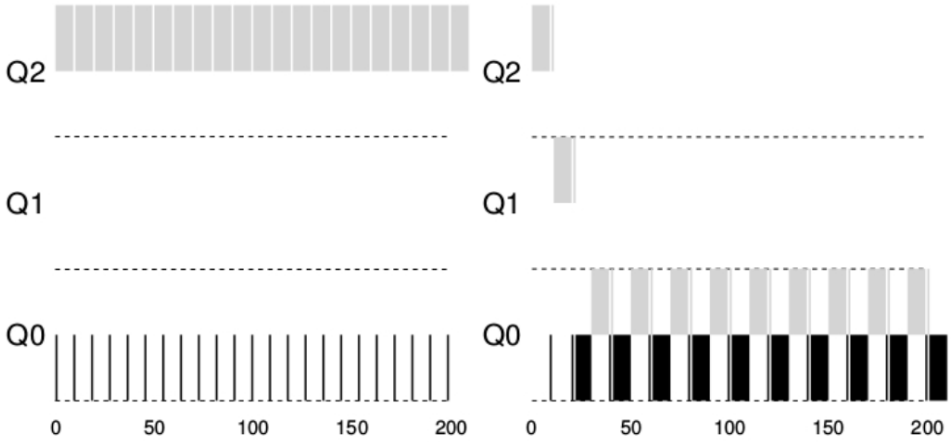
Si hay demasiadas tareas interactivas se van a combinar para consumir todo el tiempo de CPU y las tareas de larga duración nunca se van a ejecutar.

Para solucionarlo se agrega la regla 5 (Boosteo).



Gaming

Ventajear al planificador cortando los procesos antes del slice. (REGLA 4b: Si una tarea renuncia al uso de la CPU antes de un time slice completo se queda en el mismo nivel de prioridad) Para solucionarlo se modifica la regla 4.



Linux: Completely Fair Scheduler (CFS)

Fair-share-scheduling de forma altamente eficiente y escalable.

Objetivo

Mientras que los planificadores tradicionales se basan alrededor del concepto de un time-slice fijo, CFS opera de forma un poco diferente.

Su objetivo es sencillo: dividir de forma justa la CPU entre todos los procesos que están compitiendo por ella.

Vruntime

Esto lo hace mediante una simple técnica para contar llamada **virtual runtime (Vruntime).** El vruntime no es más que el runtime (es decir el tiempo que se está ejecutando el proceso) normalizado por el número de procesos runnable (se mide en nanosegundos)

A medida que un proceso se ejecuta este acumula vruntime. En el caso más básico cada vruntime de un proceso se incrementa con la misma tasa, en proporción al tiempo (real) físico. Cuando una decisión de planificación ocurre, CFS seleccionará el proceso con menos vruntime para que sea el próximo en ser ejecutado

Switches

El punto clave aquí es que hay un punto de tensión entre performance y equitatividad

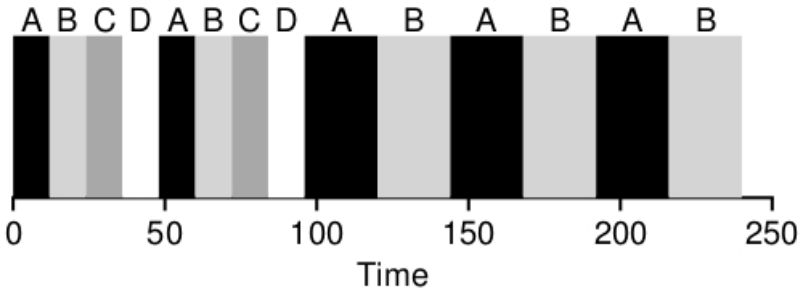
- si el CFS switchea de proceso en tiempos muy pequeños estará garantizando que todos lo proceso se ejecuten a costa de perdida de performance, demasiados context switches
- si CFS switchea pocas veces, la performance del Scheduler es buena pero el costo está puesto del lado de la equitatividad (fairness).

La forma en que CFS maneja esta tensión es mediante varios parámetros de control.

Parámetros de Control

sched_latency: este valor determina por cuánto tiempo un proceso tiene que ejecutarse antes de considerar su switcheo. (es como un time-slice pero dinámico). Un valor típico de este parámetro es de 48 ms, CFS divide este valor por el número de procesos (n) ejecutándose en la CPU para determinar el time-slice de un proceso, y entonces se asegura que por ese periodo de tiempo, CFS va a ser Completamente justo.

Por ejemplo, si n=4 procesos ejecutándose, CFS divide el valor de **sched_latency** por n asignándole a cada proceso 12 ms. CFS planifica el primer job y lo ejecuta hasta que ha utilizado sus 12 ms de (virtual) runtime, y luego chequea si hay algún otro proceso con menos vruntime, si lo hay switchea a este.



Si hay muchos procesos, esto llevaría a muchos Context Switches y por ende pequeños time slices.

Para lidiar con este problema se introduce otro parámetro llamado **min_granularity**, que normalmente se setea con el valor de 6 ms. Entonces en CFS nunca se daría el time-slice de un proceso por debajo de ese número, por ende con esto se asegura que no haya overhead por context switch.

Weighting (Niceness)

CFS tiene control sobre las prioridades de los procesos, de forma tal que pueden asignarse más CPU a un determinado proceso. Esto se hace con un mecanismo cálcico de unix llamado nivel de proceso nice , este valor va de -20 a +19, con un valor por defecto de 0. Los valores positivos de nice implican una prioridad más baja, y los valores negativos de nice implican una prioridad más alta.

CFS mapea el **nice value** con un peso como se muestra:

```
static const int prio_to_weight[40] = {
/* -20 */     88761,     71755,     56483,     46273,     36291,
/* -15 */     29154,     23254,     18705,     14949,     11916,
/* -10 */      9548,      7620,      6100,      4904,      3906,
/*  -5 */      3121,      2501,      1991,      1586,      1277,
/*   0 */      1024,       820,       655,       526,       423,
/*   5 */       335,       272,       215,       172,       137,
/*  10 */       110,        87,        70,        56,        45,
/*  15 */        36,        29,        23,        18,        15,
};
```

Estos pesos permiten calcular efectivamente el time slice para cada proceso:

$$times\_lice_k = \frac{weight_k}{\sum_{n=o}^{n-1} weight_i} * sched\_latency$$

Ejemplo:

Dado:

- $weight_A$=3121 (para el proceso A con nice = -5)
- $weight_B$=1024 (para el proceso B con nice = 0)

$$time\_slice_A = T \times (3121/(3121 + 1024))$$

$$time\_slice_B = T \times (1024/(3121 + 1024))$$

Si reemplazo T con sched_latency = 48/2 = 24

$$time\_slice_A = 24 \times (3121 + 10243121)$$

$$time\_slice_B = 24 \times (3121 + 10241024)$$

$time\_slice_A \approx 18.06$ (redondeando a dos decimales)

$time\_slice_B \approx 5.94$ (redondeando a dos decimales)

Cálculo de Vruntime

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} * runtime_i$$

Este se calcula tomando el tiempo de ejecución real que el $proceso_i$ ha acumulado ($runtime_i$) y lo escala de manera inversa según el peso del proceso.

Cálculo de Delta Vruntime

$$delta\_vruntime = time\_slice \times \frac{weight}{NICE\_ZERO\_LOAD}$$

donde:

- time_slice es la cantidad de tiempo que el proceso ha sido ejecutado.
- NICE_ZERO_LOAD es una constante que representa la carga por defecto de un proceso con un valor nice de 0. Esta constante suele tener un valor de 1024.
- Weight es el peso del proceso, que está determinado por su valor nice.

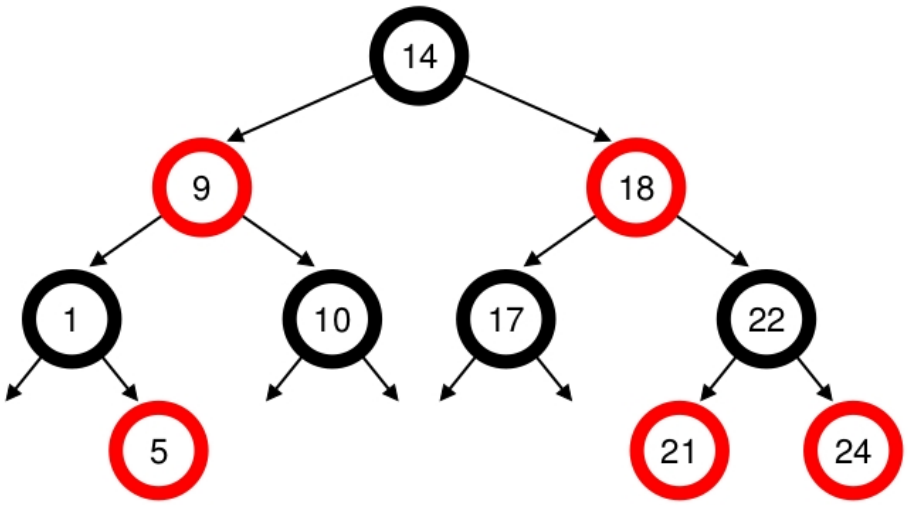
Árbol Rojo Negro

Uno de los focos de eficiencia del CFS está en la implementación de las políticas anteriores. Pero también en una buena selección del tipo de dato cuando el planificador debe encontrar el próximo Job a ser ejecutado.

- Las listas no escalan bien O(n)
- Los árboles sí, en este caso los árboles Rojo-Negro O(log(n))

Cuando el Scheduler es invocado para correr un nuevo proceso, actúa de esta forma:

1. El nodo más a la izquierda del árbol de planificación es elegido (ya que tiene el tiempo de ejecución más bajo), y es enviado a ejecutarse.
2. Si el proceso simplemente completa su ejecución, este es eliminado del sistema y del árbol de planificación.
3. Si el proceso alcanza su máximo tiempo de ejecución o de otra forma se para la ejecución voluntariamente o vía una interrupción) este es reinsertado en el árbol de planificación basado en su nuevo tiempo de ejecución (vruntime).
4. El nuevo nodo que se encuentre más a la izquierda del árbol será ahora el seleccionado, repitiéndose así la iteración.

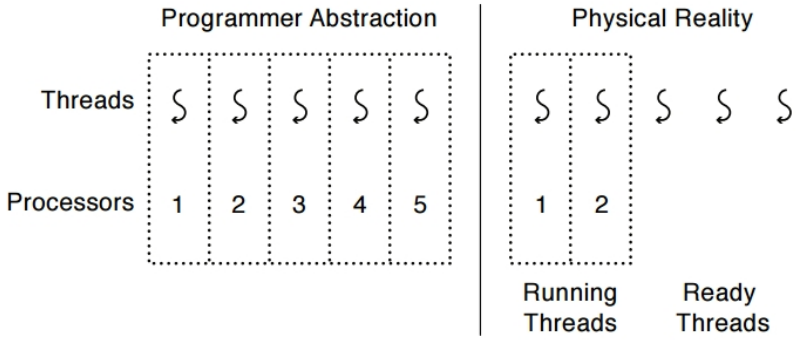


CONCURRENCIA

¿Qué es la Concurrencia?

El mundo de la concurrencia se refiere a un conjunto de actividades que pueden suceder al mismo tiempo. Anderson-Dahlin pág. 129.

El concepto clave es escribir un programa concurrente como una secuencia de streams de ejecución o threads que interactúan y comparten datos en una manera muy precisa. El concepto básico es el siguiente:



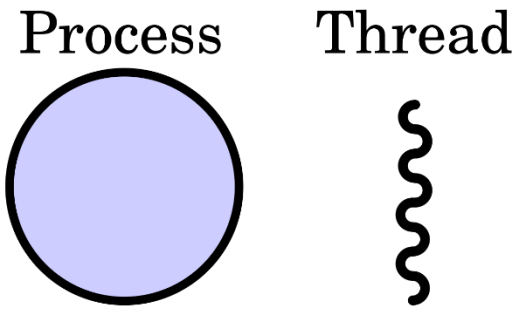
Thread

Un thread es una secuencia de ejecución atómica que representa una tarea planificable de 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 de ejecución: El sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando él desee.

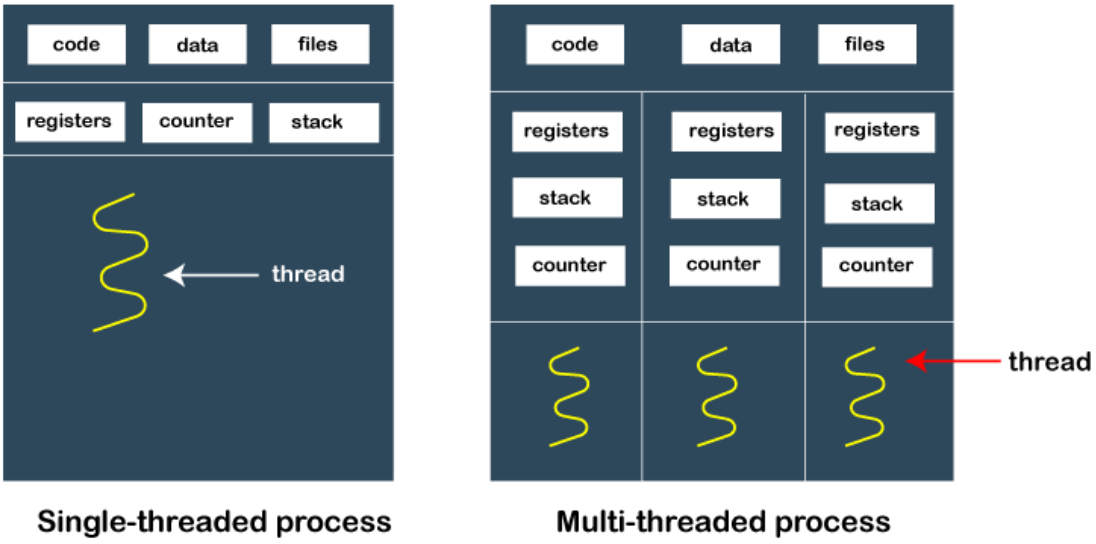
Thread vs Proceso

- Proceso: un programa en ejecución con derechos restringidos.
- Thread: una secuencia independiente de instrucciones ejecutándose dentro de un programa.



Elementos de un Thread

- Thread id.
- Conjunto de los valores de registros.
- Stack propio.
- Una política y prioridad de ejecución.
- Un propio errno.
- Datos específicos del thread.

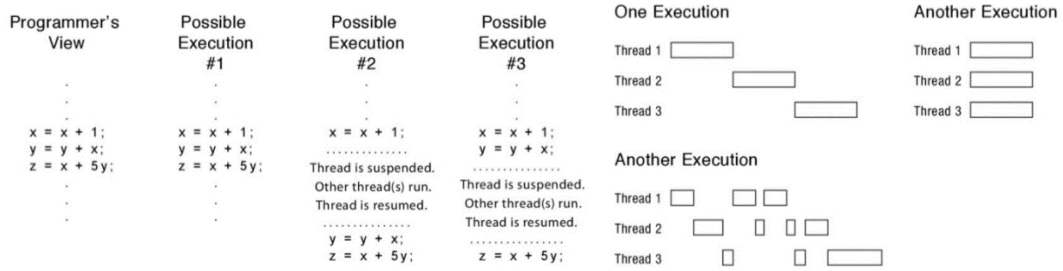


Thread Scheduler

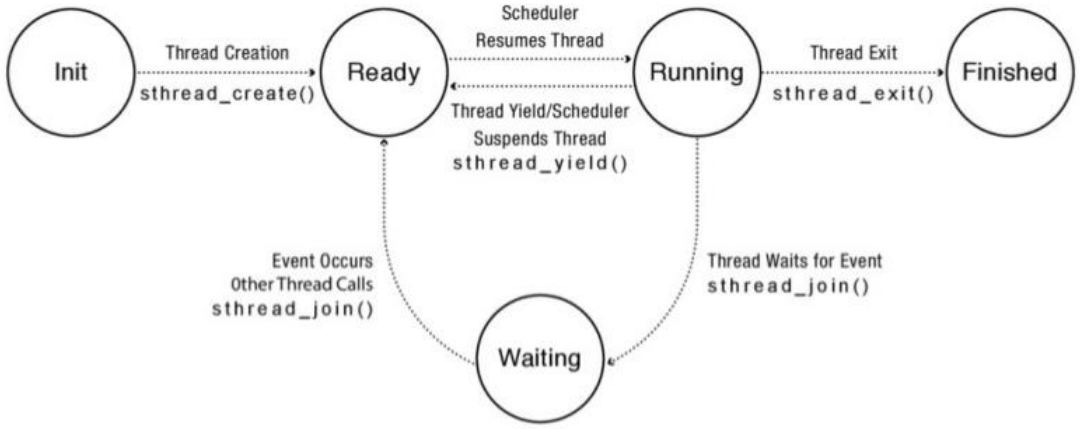
Es necesario un planificador de thread o thread-Scheduler, ya que el S.O. podría estar trabajando con un único procesador. El cambio entre threads es transparente, es decir que el programador debe preocuparse de la secuencia de instrucciones y no el cuándo éste debe ser suspendido o no.

Por ende los Threads proveen un modelo de ejecución en el cual cada thread corre en un procesador virtual dedicado (exclusivo) con una velocidad variable e impredecible Anderson-Dahlin, pág. 138.

Esto quiere decir que desde el punto de vista del thread cada instrucción se ejecuta inmediatamente una detrás de otra. Pero el que decide cuando se ejecuta es el planificador de threads o thread Scheduler



Estados de un thread



Thread vs Proceso (Linux)

Tabla de equivalencias entre procesos y threads

Process primitive	Thread primitive	Description
fork	pthread_create	crea un nuevo flujo de control
exit	pthread_exit	sale de un flujo de control existente
waitpid	pthread_join	obtiene el estado de salida de un flujo de control
atexit	pthread_cleanup	función a ser llamada en el momento de salida de un flujo de co
getpid	pthread_self	obtiene el id de un determinado flujo de control
abort	pthread_cancel	terminación anormal de un flujo de control

Diferencias

Los threads:

- Por defecto comparten memoria.
- Por defecto comparten los descriptores de archivos.
- Por defecto comparten el contexto del FileSystem.
- Por defecto comparten el manejo de señales.

Los Procesos:

- Por defecto no comparten memoria.
- Por defecto no comparten los descriptores de archivos.
- Por defecto no comparten el contexto del FileSystem.
- Por defecto no comparten el manejo de señales.

Thread en Linux

Linux utiliza un modelo 1-1 (proceso-thread), con lo cual dentro del kernel no existe distinción alguna entre thread y proceso – todo es una tarea ejecutable.

Memoria en un proceso multi-thread

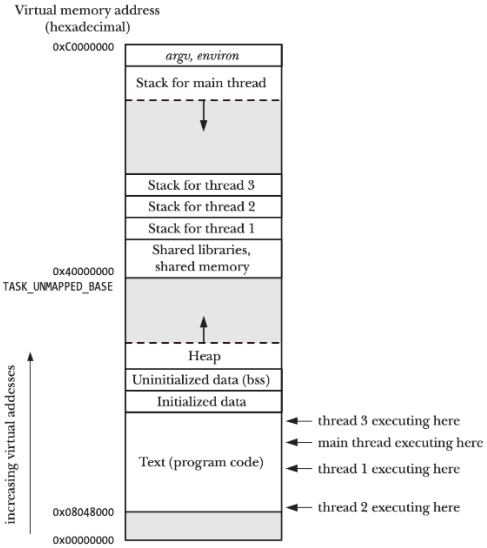


Figure 29-1: Four threads executing in a process (Linux/x86-32)

Race Conditions

Una race condition se da cuando el resultado de un programa depende en cómo se intercalaron las operaciones de los threads que se ejecutan dentro de ese proceso. De hecho los threads juegan una carrera entre sus operaciones, y el resultado del programa depende de quién gane esa carrera.

Threads ejemplo

Thread 1: saldo = saldo + 10	(saldo inicialmente vale 100)
`load r1, saldo`	saldo=100 **r1=100**
`add r1, r1, 10`	saldo=100 **r1=110**
`store saldo, r1`	**saldo=110** r1=110
Thread 2: saldo = saldo + 20	(saldo inicialmente vale 100)
`load r1, saldo`	saldo=100 **r1=100**
`add r1, r1, 20`	saldo=100 **r1=120**
`store saldo, r1`	**saldo=120** r1=120

Ejecución

`load r1, saldo`	saldo=100 r1=100
`add r1, r1, 20`	saldo=100 **r1=120**
`load r1, saldo`	saldo=100 **r1=100**
`store saldo, r1`	**saldo=100** r1=100
`add r1, r1, 10`	saldo=100 **r1=110**
`store saldo, r1`	**saldo=110** r1=110 **Se perdieron $20!!!**

Sección crítica

Secciones de código no atómicas donde se comparten recursos.

Como aquella sección del código fuente se necesita que se ejecute en forma atómica, se encierra dentro de un lock.

Locks

Un **lock** es una variable que permite la sincronización mediante la exclusión mutua, cuando un thread tiene el candado o lock ningún otro puede tenerlo.

La idea principal es que un proceso asocia un lock a determinados estados o partes de código y requiere que el thread posea el lock para entrar en ese estado. Con esto se logra que sólo un thread acceda a un recurso compartido a la vez.

Esto permite la exclusión mutua, todo lo que se ejecuta en la región de código en la cual un thread tiene un lock, garantiza la atomicidad de las operaciones.

Propiedades

Un Lock debe asegurar:

- Exclusión mutua: como mucho un solo Thread posee el lock a la vez.
- Progress: Si nadie posee el Lock, y alguien lo quiere, debe poder obtenerlo.
- Bounded waiting: Si T quiere acceder al lock y existen varios threads en la misma situación, los demás tienen una cantidad finita (un límite) de posible accesos antes que T lo haga.

PARCIALES/FINALES

2C-2023 15/11

75.08 Sistemas Operativos
Examen 15 de noviembre de 2023
Nombre y Apellido: _____
Padron:_____

1. File System
   a. El superbloque de un sistema de archivos indica que el 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`	**Inodos: ___ Blq. datos: ___**
`# stat /dir/s/y`	**Inodos: ___ Blq. datos: ___**
`# ln /dir/s/x /dir/h` `# ln -s /dir/s/y /dir/y`	
`# cat /dir/h`	**Inodos: ___ Blq. datos: ___**
`# cat /dir/y`	**Inodos: ___ Blq. datos: ___**

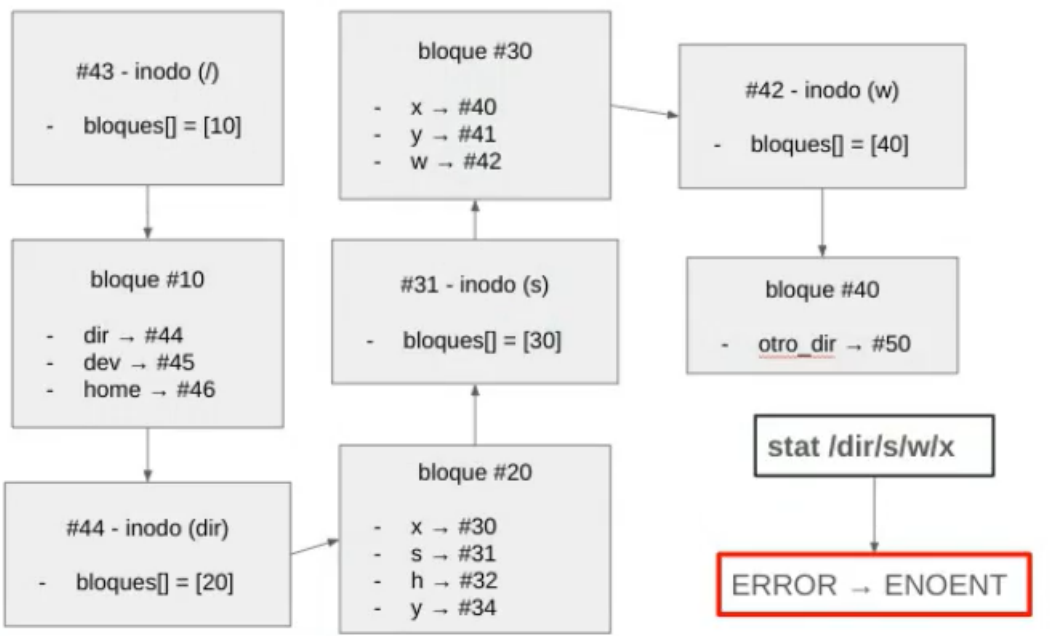
Ayuda: todos los directorios ocupan un bloque. La idea es que describan como stat llega a los archivos

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) Contar accesos

En el primer caso stat /dir/s/w/x, el archivo x no existe, por lo que tira error.



Se accede a 4 Inodos y a 4 Bloques. Si no hubiese error haría un acceso más al Inodo de x, siendo 5 en total.

En el segundo caso stat /dir/s/y llegaría al Inodo (y), que en el grafico sería como llegar al Inodo (w). Hay acceso a 4 Inodos y a 3 Bloques.

En el tercer caso cat /dir/h llegaría al bloque de h, que en el gráfico sería como llegar al bloque #30.

En el cuarto caso cat /dir/y llegaría al bloque de y, que en el gráfico sería como llegar al bloque #30.

El **stat** llega al nivel de inodo, mientras que el **cat** tiene que acceder uno más para leer el contenido del bloque.

1b) Describa la estructura de un i-nodo

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. Es una struct con metadata de un archivo.

- Tamaño
- Fecha de modificación
- Propietario
- Información de seguridad (Qué se puede hacer con el archivo)

2a) ¿Qué es un deadlock? Dar 3 casos.

Sucede cuando entre varios threads uno obtiene el lock y por alguna razón no lo libera entonces el resto de los threads se quedan esperando a que se libere el recurso cosa que nunca sucede

- Exclusión mutua: los threads reclaman control exclusivo sobre un recurso compartido que necesitan
- Hold & wait: un thread mantiene un recurso reservado para sí mismo mientras espera que se de alguna condición (lo agarra y espera).
- No preemption: no hay forma de sacarlo, ya que los recursos adquiridos no pueden ser desalojados por la fuerza.
- Circular wait: hay un conjunto de threads que de forma circular cada uno reserva uno o más recursos compartidos que son requeridos por el siguiente en la cadena. Cuando un thread se bloquea con un recurso.

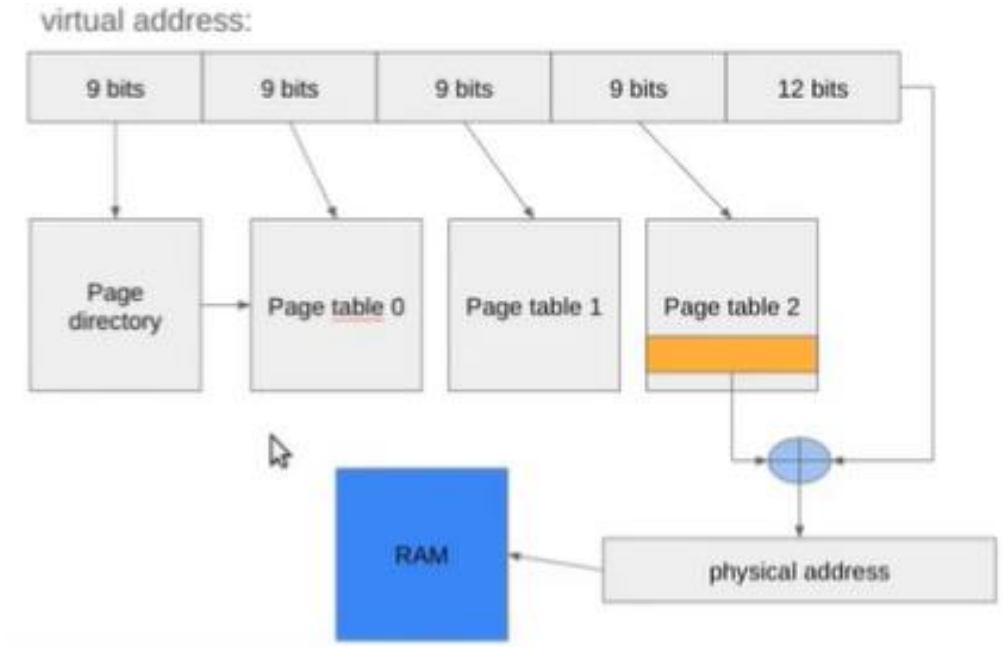
Caso de los filósofos

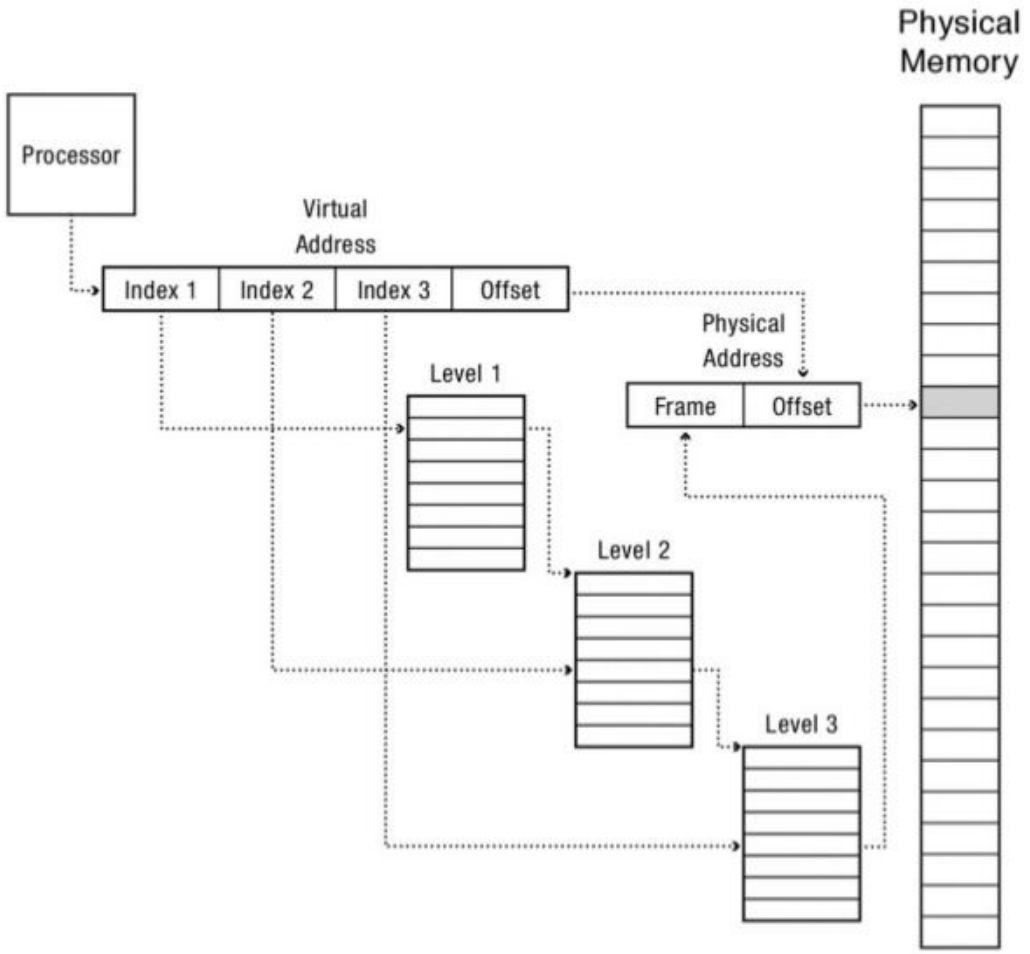
Hay, en una mesa redonda, cinco filósofos sentados esperando para comer con sus DOS palillos chinos (**Exclusión mutua**). Cada uno tiene un plato de fideos y un palillo a la izquierda de su plato. Para comer si o si necesitan ambos palillos, por lo que si tienen uno solo no podrán hacer nada. Cada uno solo puede tomar el de su izquierda y derecha, otros no. Además, cada vez que uno agarra un palito NO lo suelta (**no preemption**). Si todos toman el palito a su izquierda al mismo tiempo (**circular wait**), todos están esperando para siempre que otro suelte su palito para poder agarrar dos y comer, pero nunca pasa. Agarran un recurso que otro necesita, pero esperan la condición de tener dos para poder comer, por lo que no hacen nada y le sacan el recurso a los demás (**Hold & Wait**).

2b) Cantidad de Kbytes que se pueden almacenar en un esquema de 48 bits

En cada directory/tabla se almacena la dirección física de la próxima tabla (48 bits).

$2^9 * 2^9 * 2^9 * 2^9 * 2^{12} = 2^{48}$ bytes.





Como pide en Kb divido por 2^{10} quedando $2^{38.}$

2c) Explique MLFQ

Reglas básicas:

1. Si la prioridad de A es mayor que la de B, A se ejecuta y B no.
2. Si la prioridad de A es igual a la de B, A y B se ejecutan en Round-Robin.
3. Cuando un proceso entra en el sistema se pone en la prioridad más alta.
4. Una vez que una tarea usa su asignación de tiempo (slice) de un nivel dado (independientemente de cuantas veces haya renunciado al uso de la CPU) su prioridad se reduce (baja un nivel)
5. Después de cierto tiempo S, se mueven todos los procesos a la cola con más prioridad.
    - Si el valor de S es muy alto, los procesos que requieren mucha ejecución caen en Starvation
    - Si el valor de S es muy chico, las tareas interactivas no van a poder compartir adecuadamente la CPU.

3a) Ping-Pong

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

int main(int argc, char *argv[]) {
    int pelota = 0;

    int pipe_fds[2];
    int pipe_fds_ans[2];

    if (pipe(pipe_fds) < 0 || pipe(pipe_fds_ans) < 0) {
        printf("error in pipe\n");
        exit(-1);
    }

    pid_t child_id = fork();

    if (child_id < 0) {
        printf("error in fork\n");
        exit(-1);
    }

    if (child_id == 0) {
    // el hijo no escribe por el primer pipe
    // el hijo no lee por el segundo pipe
        close(pipe_fds[WRITE]);
        close(pipe_fds_ans[READ]);

        // mientras <pipe lectura no cerrado>:
        while (read(pipe_fds[READ], &pelota, sizeof(pelota)) > 0) {
            // incremento en 1
            pelota += 1;
            // si hay overflow o cambio de signo corto
            if (overflow o cambio de signo) {
                break;
            }
            // si está todo bien, respondo enviando la "pelota"
            if (write(pipe_fds_ans[WRITE], &pelota, sizeof(pelota)) < 0)
{
                printf("error in write in primes\n");
                close(pipe_fds[READ]);
                close(pipe_fds_ans[WRITE]);
                exit(-1);
            }
        }
        close(pipe_fds[READ]);
        close(pipe_fds_ans[WRITE]);
```

```
    } else {
        // el padre no lee por el primer pipe
        // el padre no escribe por el segundo pipe
        close(pipe_fds[READ]);
        close(pipe_fds_ans[WRITE]);

        // envio la pelota por primera vez
        if (write(pipe_fds[WRITE], &pelota, sizeof(pelota)) < 0) {
                printf("error in write in primes\n");
                close(pipe_fds[READ]);
                close(pipe_fds_ans[WRITE]);
                exit(-1);
        }

        // espero respuesta y respondo
        while (read(pipe_fds_ans[READ], &pelota, sizeof(pelota)) > 0) {
            // incremento en 1
            n += 1;
            // si hay overflow o cambio de signo corto
            if (overflow o cambio de signo) {
                break;
            }
            // si está todo bien, respondo enviando la "pelota"
            if (write(pipe_fds[WRITE], &pelota, sizeof(pelota)) < 0) {
                printf("error in write in primes\n");
                close(pipe_fds[WRITE]);
                close(pipe_fds_ans[READ]);
                exit(-1);
            }
        }

        close(pipe_fds[WRITE]);
        close(pipe_fds_ans[READ]);
        wait(NULL);
    }
    return 0;
}
```

2b) Requerimientos mínimos de hardware para implementar un Kernel.

- Manejo de Modos/Privilegios.
- Timer Interrupts. (Si hay desalojo, si es cooperativo no hace falta)
- Transición segura entre modos.

2C-2022 Parcial 4/10/22

75.08 Sistemas Operativos
Primer Parcial
Nombre y Apellido: [redacted]
Padron: [redacted]

4 de Octubre de 2022

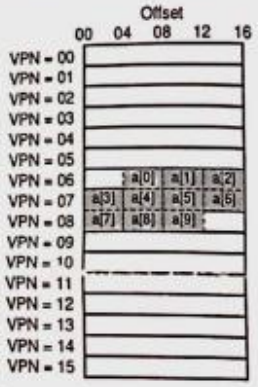
1. Kernel y Procesos (10 ptos.)

   a. Describa que 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.

   b. Cuál / cuáles mecanismos utiliza el kernel para garantizar el aislamiento entre procesos. Estos mecanismos están relacionados con el hardware, porque deben existir y donde se ve su funcionamiento.

2. Memoria (10 ptos.)

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



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.

B. Suponga que virtual address con las siguientes caracteristicas:

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 CAFE	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: 00000000, 20022002, 10022002, 00015555 .

3. Concurrencia y Scheduling (10 ptos.)

a. Explique con un ejemplo MLFQ.

b. ¿Cuáles de los siguientes mecanismos son compartidos entre threads de un mismo programa?

- ☐ Stack Segment
- ☒ File descriptors
- ☐ Registros de CPU
- ☒ Heap
- ☐ Code segment
- ☐ METADATA del Thread
- ☐ Data Segment
- ☐ Signals

1a) Describa que es un proceso

“Un Proceso es una entidad abstracta, definida por el Kernel, en la cual los recursos del sistema son asignados” [KER]

Un proceso Incluye:

- Los Archivos abiertos.
- Las **signals** pendientes.
- Datos internos del Kernel.
- Estado completo del procesador.
- Un espacio de direcciones de memoria.
- Uno o más hilos de ejecución.

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, está dividido en 4 segmentos:

- Text: Instrucciones del Programa.
- Data: Variables Globales (extern o static en C)
- Heap: Memoria Dinámica Alocable
- Stack: Variable Locales y trace de llamadas

Esto lo logra mediante **memoria virtual**: es una abstracción por la cual la memoria física puede ser compartida por diversos procesos.

Un componente clave de la memoria virtual son las direcciones virtuales, con las direcciones virtuales, para cada proceso su memoria inicia en el mismo lugar, la dirección 0. El hardware (MMU) traduce la dirección virtual a una dirección física de memoria.

1b) Aislamiento de procesos

El kernel pone a un proceso de usuario en un entorno aislado (sandbox).

Lo logra mediante:

- Privilegio de Instrucciones.
- Address Space: Virtualización de memoria.
- Timer Interrupts.

2a) TLB, miss, hit, accesos a memoria y traducciones

- Hay 10 accesos a memoria.
- Traducciones:
  - Para a[0], a[3] y a[7] hay miss. Se buscan en la TLB y no están, entonces se cargan esos frames.
  - Para el resto hay hit. Se aprovecha el miss de los anteriores y se ahorra la traducción ya que están en la TLB.
  - Por lo tanto hay 3 traducciones completas de VA a PA.

Cuando pifia, carga todo el frame entonces el resto queda en la TLB.

2b) Traducción VA a PA

- 0 000 0000:

Segment table	Page Table A	Page Table B
0 Page B 1	0 CAFE	0 F000 2
1 Page A	1 DEAD	1 D8BF
X invalid	2 BEEF	2 3333
	3 BA11	x INVALID

F000 0000 (offset)

- 2 002 2002:

Segment table	Page Table A	Page Table B
0 Page B	0 CAFE	0 F000
1 Page A	1 DEAD	1 D8BF
X invalid 1	2 BEEF	2 3333
	3 BA11	x INVALID

- 1 002 2002

Segment table	Page Table A	Page Table B
0 Page B	0 CAFE	0 F000
1 Page A 1	1 DEAD	1 D8BF
X invalid	2 BEEF 2	2 3333
	3 BA11	x INVALID

BEEF 2002 (offset)

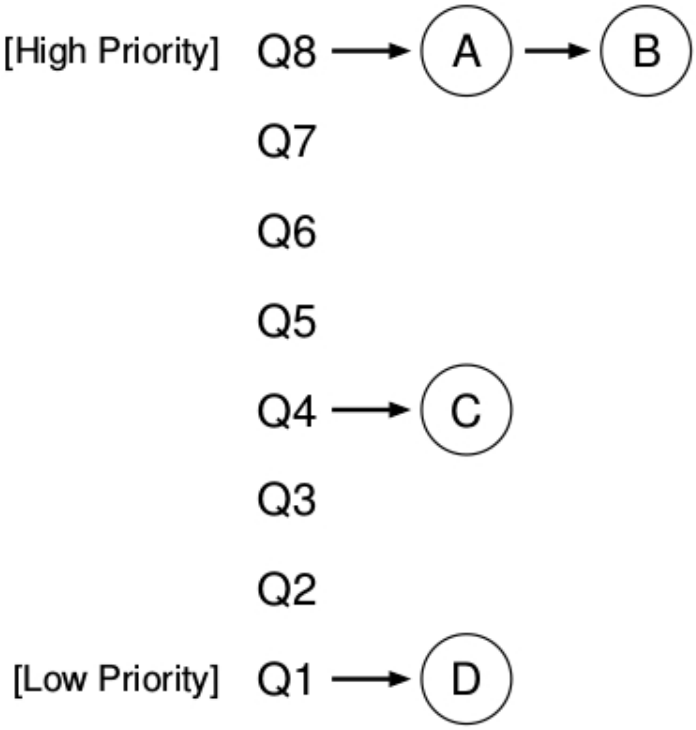
- 0 001 5555

Segment table	Page Table A	Page Table B
0 Page B 1	0 CAFE	0 F000
1 Page A	1 DEAD	1 D8BF 2
X invalid	2 BEEF	2 3333
	3 BA11	x INVALID

D8BF 5555 (offset)

3a) Ejemplo de MLFQ

Ver MLFQ

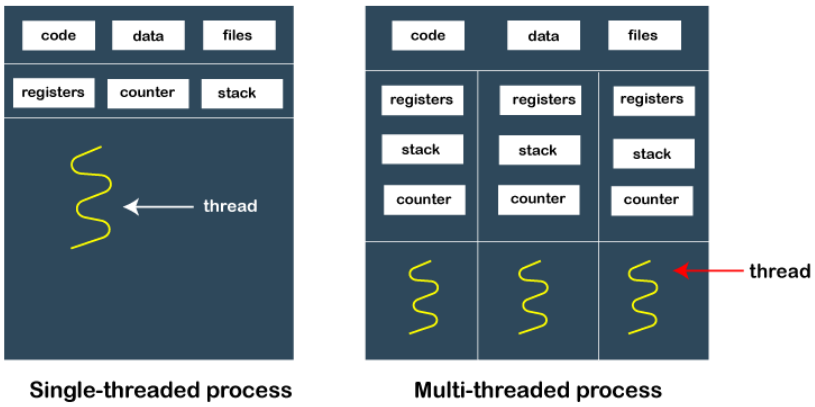


Al comienzo A y B se ejecutan con Round Robin e irán bajando luego de completar una determinada cantidad de time slices. En algún momento se boostearan todos a la primera cola para evitar starving.

3b) Mecanismos compartidos entre threads de un mismo programa

Ver Threads

- Code Segment
- Data Segment
- File descriptors
- Signals
- Heap



Ejercicios De FileSystem

¿Qué es un hardlink, softlink, un volumen y un mount point?

- **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 común 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.

Describa el API del sistema de archivos. Diferencia entre Syscalls y Library Calls, ponga un ejemplo.

Ver API

Una system call es implementada en el kernel space, en cambio una library call es implementada en el user space. Por ejemplo, en C existe la library call fopen que internamente llama a la system call open, pero devuelve un objeto FILE.

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

Ver VFS y Estructuras

“BigFS” FFS Tamaño máximo

Suponiendo que BigFS es una variante de FSS, el clásico sistema de archivos de unix, el cual posee 12 referencias a i-nodos directas, 1 indirecta, 1 doble indirecta, una triple indirecta y una cuádruple indirecta. Asumiendo bloques de 4kb, 8 bytes por puntero a bloques. Cuál es el máximo tamaño de archivo que se soporta?

Ver FFS

4 KB → 4 * 1024 bytes → 2^{12} bytes. (almacenamiento total de un bloque)

Si divido por el peso de cada puntero (8 bytes) → bloque tiene 2^{12} / 2^{3} punteros.

2^{12} / 2^{3} = 2^{9} punteros por bloque.

2^{9} * 2^{9} * 2^{9} * 2^{9} * 2^{12} = 2^{48} → Más de 1 TB

Disco posee 64 bloques de 4Kb e i-nodos de 256 bytes. Estructure FS.

Ver VSFS

- 1 inodo ↔ 1 archivo.
- 64 bloques ↔ 64 inodos como máximo.
- Un archivo puede ocupar como mínimo un bloque.
- Un archivo como máximo puede ocupar 64 bloques.

Bloque 1: Super bloque.

Bloque 2: bitmap inodos.

Bloque 3: bitmap bloques.

Si un bloque tiene 4Kb y los i-nodos 256, en un bloque entran 4096/256 = 16 i-nodos.

Si tengo 3 Bloques de i-nodos → 48 i-nodos para 58 bloques → Me faltan i-nodos.

Si tengo 4 Bloques de i-nodos → 64 i-nodos. Para 57 bloques → Bien.

Bloques 4-7: i-nodos.

Bloques 8-64: Bloques de datos.

Disco posee 256 bloques de 4Kb e i-nodos de 512 bytes. Estructure FS

Ver VSFS

Bloque 1: Super bloque.

Bloque 2: bitmap inodos.

Bloque 3: bitmap bloques.

Si un bloque tiene 4Kb y los i-nodos 512, en un bloque entran 4096/512 = 8 i-nodos.

256 bloques ↔ 256 inodos como máximo.

256 / 8 = 32 bloques de i-nodos.

Si tuviese 32 bloques de i-nodos tendría de más, podría reducir el número a 29 bloques de i-nodos → 232 i-nodos para 224 bloques de data.

Si tomara 28 bloques de i-nodos → 224 i-nodos para 225 bloques de data, me faltaría uno.

Accesos mediante tabla

Dada la siguiente información de la tabla de i-nodos y el contenido de los bloques de datos, indicar:

¿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.

I-node entry	Block ptr	data block #	content
0	1	0	mariano:4, dato:11, juan:12
		1	home:5, mnt:11, bin:3
3	5	5	ls:40, cat:41, vim:
4	11	6	cdrom:33
5	0	7	apuntes.md:101, start.sh:32
		9	jos.c.md:104, start.c.:34
10	6	11	examen.md:39, apuntes.txt:103
11	9	43	
12	7		
39	44		
40	45		
41	57		
32	111		
33	43		
34	44		
101	52		
102	53		
103	99		
104	54		

Nota: la primera columna de la tabla de i-nodos refiere al número de i-nodo, mientras que la primera columna de la tabla de bloques hace referencia a la numeración de bloques relativa a los bloques de datos. Ayuda: recordar que el i-nodo 0 es siempre el directorio raíz.

Los pasos a seguir para ver que se imprime en cualquiera de los casos pedidos, es el siguiente:

1. Se arranca por la raíz '/' que tal y como se aclara, siempre es el inodo 0.
2. A partir de acá, buscamos dicho número en la tabla, y asociado tendrá un puntero a un bloque de data.
3. Procedemos a ver dónde apunta ese puntero y vemos qué se encuentra en el bloque de datos, tendremos el número de inodo siguiente en nuestro path (para el primer caso sería 'home' por ejemplo).
4. Buscamos el inodo obtenido, su puntero a bloque asociado, etc. así hasta el final.

Ejecutar ls /home/mariano

-inodo 0

-bloque 1 → home → inodo 5

-bloque 0 → mariano → inodo 4

-bloque 11

Se imprime:

examen.md apuntes.txt

Ejecutar ls /home/juan

-inodo 0

-bloque 1 → home → inodo 5

-bloque 0 → juan → inodo 12

-bloque 7

Se imprime:

apuntes.md start.sh

Ejecutar ls /bin

inodo 0

-bloque 1 → bin → inodo 3

-bloque 5

Se imprime:

ls cat vim

Implementar el comando ls

Ver https://iq.opengenus.org/ls-command-in-c/

```c
//Used for basic input/output stream
#include <stdio.h>
//Used for handling directory files
#include <dirent.h>
//For EXIT codes and error handling
#include <errno.h>
#include <stdlib.h>

void _ls(const char *dir,int op_a,int op_l) {
    //Here we will list the directory
    struct dirent *d;
    DIR *dh = opendir(dir);
    if (!dh) {
        if (errno = ENOENT) {
            //If the directory is not found
            perror("Directory doesn't exist");
        } else {
            //If the directory is not readable then throw error and exit
            perror("Unable to read directory");
        }
        exit(EXIT_FAILURE);
    }
    //While the next entry is not readable we will print directory files
    while ((d = readdir(dh)) != NULL) {
        //If hidden files are found we continue
        if (!op_a && d->d_name[0] == '.')
            continue;
        printf("%s  ", d->d_name);
        if(op_l) printf("\n");
    }
    if(!op_l) printf("\n");
}
int main(int argc, const char *argv[]) {
    if (argc == 1) {
        _ls(".",0,0);
    }
    else if (argc == 2) {
        if (argv[1][0] == '-') {
            //Checking if option is passed
            //Options supporting: a, l
            int op_a = 0, op_l = 0;
            char *p = (char*)(argv[1] + 1);
            while(*p) {
                if(*p == 'a') op_a = 1;
                else if(*p == 'l') op_l = 1;
                else {
```

```
                    perror("Option not available");
                    exit(EXIT_FAILURE);
                }
                p++;
            }
            _ls(".",op_a,op_l);
        }
    }
    return 0;
}
```