

Sistemas Operativos

(75.08 / 95.03 / TA043)

INTRODUCCION	2
KERNEL	10
EL PROCESO	18
SCHEDULLING	25
THREADS	45
MEMORIA	55
CONCURRENCIA	76
FILESYSTEM	79
ANEXO I: Inicial al SO y al Kernel	95
ANEXO II: Arquitecturas x86 y risc-V	96

INTRODUCCIÓN

¿Qué es y que hace un SISTEMA OPERATIVO?

- Un **Sistema Operativo (OS)** es la capa de software que maneja los recursos de una computadora para sus **usuarios** y sus **aplicaciones**.
- En un sistema operativo de propósito general, los **usuarios** interactúan con **aplicaciones**, estas aplicaciones se **ejecutan** en un **ambiente** que es proporcionado por el **sistema operativo**. A su vez el sistema operativo **hace de mediador** para tener acceso al **hardware** del equipo.



- Software encargado de hacer que la ejecución de los programas parezca algo fácil. La forma principal para llegar a lograr esto es mediante el concepto de **virtualización**. Esto es, el sistema operativo toma un recurso físico (la memoria, el procesador, un disco) y lo transforma en algo virtual más general, poderoso, fácil de usar.
- Un sistema operativo deberá virtualizar varios recursos físicos:
 - El procesador
 - La memoria
 - La persistencia (dispositivos de almacenamiento).

Juega simultáneamente 3 roles:

Referí

GESTIONA RECURSOS:

- Gestionan los recursos compartidos entre diferentes aplicaciones en una misma máquina, decide quién va a usar qué recurso y cuándo.
- Pueden detener un programa e iniciar otro según sea necesario.
- Aíslan aplicaciones entre sí, evitando que errores en una afecten a las demás.
- Protegen al sistema y a las aplicaciones de virus informáticos.

DISTRIBUCIÓN DE RECURSOS:

- Deciden qué aplicaciones reciben cuáles recursos y cuándo, ya que todas comparten los recursos físicos disponibles.

Ilusionista

ABSTRACCIÓN DEL HARDWARE

- Proveen una abstracción del hardware físico para simplificar el diseño de aplicaciones.
- Ofrecen la ilusión de memoria casi infinita y uso exclusivo de procesadores, independientemente de la cantidad real de memoria física o número de procesadores disponibles.
- Permiten escribir aplicaciones sin preocuparse por los detalles físicos del sistema.

Pegamiento

SERVICIOS COMUNES

- Facilitan el uso compartido de información entre aplicaciones, como copiar y pegar, y acceso a archivos.
- Proveen rutinas de interfaz de usuario comunes para una apariencia y experiencia uniforme en el sistema.
- Actúan como capa de separación entre las aplicaciones y los dispositivos de entrada/salida (I/O), permitiendo que las aplicaciones funcionen independientemente del hardware específico en uso.

Virtualización

Técnica general en la que el SO logra estas funcionalidades, tomando un recurso físico como puede ser la memoria, un disco o el procesador y transformándolo en una forma virtual más general, poderosa y fácil de usar.

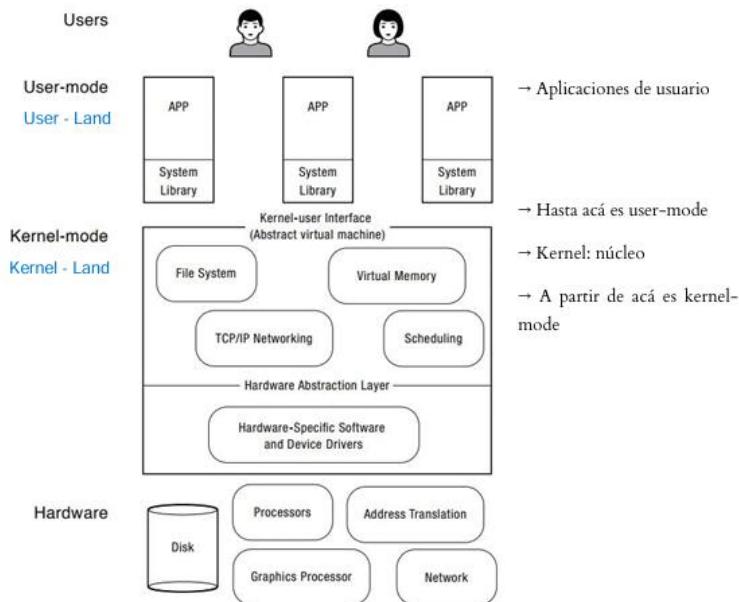
- ★ Visualización = Abstracción → Abstraer: Despojarse de complejidades q no me interesan para resolver un problema
- ★ Se visualiza la CPU, la Memoria, el Tiempo (conurrencia), los Periféricos. → Es todo abstracto para las aplicaciones.

Unix: El caso de estudio fundamental

Tiene todo lo que un SO debe tener. Su filosofía sigue los siguientes cuatro mandatos:

- Simplicidad en funcionalidades: Cada programa debe realizar una y nada más que una tarea bien, para nuevas funcionalidades se crean nuevos programas.
- Interoperabilidad y simplicidad en la salida: Cada salida de un programa pueda ser utilizada como la entrada de otros, evitando formatos complejos o interactivos.
- Pruebas tempranas y refactorización: Diseña software para ser probado temprano, idealmente en semanas.
- Uso de herramientas para eficiencia: Utiliza herramienta especializadas para simplificar tareas de programación, incluso construyendo herramientas temporales que serán descartadas luego de su uso.

Sistema Operativo Unix-like



El Sistema Operativo se ejecuta como **la capa de software de más bajo nivel** en la computadora. Este contiene:

- Una capa para la gestión de dispositivos específico
- Una serie de servicios para la gestión de dispositivos agnósticos del hardware que son utilizados por las aplicaciones. Estas dos capas suelen ser conocidas como el **kernel** del sistema operativo. Cuando código fuente de esta capa es ejecutado la computadora pasa a un estado llamado **Modo Supervisor** --> la CPU puede ejecutar instrucciones privilegiadas: por ejemplo, habilitar y deshabilitar interrupciones, leer y escribir en el registro que contiene la dirección de una tabla de páginas, etc.
- Las aplicaciones se ejecutan en un **contexto AISLADO, PROTEGIDO y RESTRINGIDO y mediante el uso de funciones que se encuentran en bibliotecas, pueden utilizar los servicios de acceso al hardware o recursos que el kernel proporciona**. El contexto de ejecución de las aplicaciones se denomina **User-land**, más restrictivo, aislado y controlado.

Requisitos clave de un sistema operativo

- **Multiplexación**: Debe permitir que varios procesos se ejecuten simultáneamente incluso cuando los mismos son más que CPUs disponibles, compartiendo tiempo y recursos entre procesos.
- **Aislamiento**: Debe asegurar que los procesos no afecten a otros en caso de errores. Si un proceso falla, no debe impactar a los procesos independientes. Cada programa de usuario está en un sandbox donde nadie puede entrar. Sin embargo, este aislamiento no debe ser absoluto para permitir la interacción controlada entre procesos.
- **Interacción**: Es fundamental que el sistema operativo permita la comunicación intencionada entre procesos. Un proceso puede interactuar con otros sin joderse. Ejemplo: Las tuberías (pipes) permiten que la salida de un proceso se convierta en la entrada de otro, facilitando la colaboración entre ellos.

Caso de Estudio: El diseño del sistema operativo Unix

- En Unix, TODO es un archivo --> Podemos hacer casi todo con una API de archivos
- Cuando un programa abre un archivo (o cualquier recurso similar), el sistema operativo le asigna un número entero llamado file descriptor, que se usa para leer, escribir o cerrar el archivo.

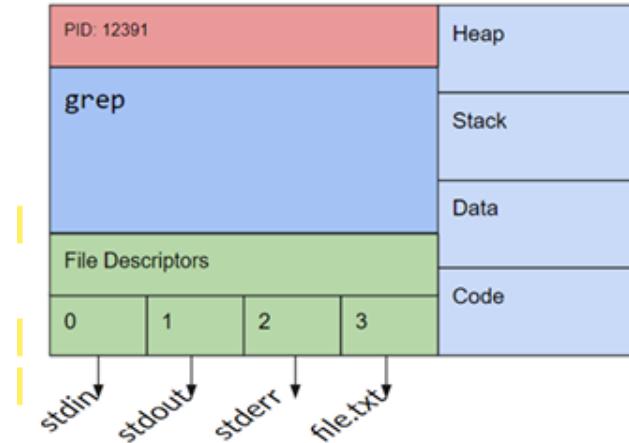
Procesos

Un proceso es un PROGRAMA en EJECUCION en modo usuario. Es DINAMICO. Tienen una estructura interna propia. Todos los procesos MENOS EL KERNEL viven en user-land. El kernel es la interfaz entre el hardware y el software.

Las partes básicas de un proceso son:

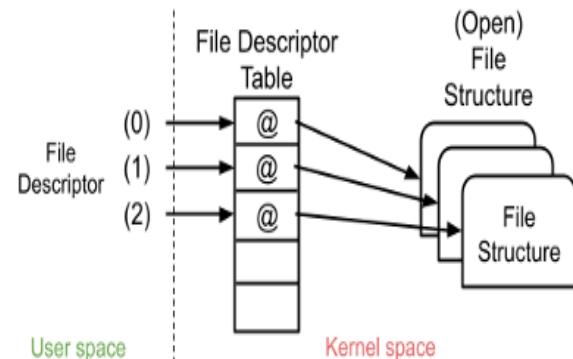
- PID: Process Id con el que lo identifica el kernel
- Nombre del Programa
- File Descriptors
- Memoria dividida en: código, datos, stack de ejecución, heap.

→ El kernel posee una tabla llamada Process Tables, donde se guarda información de cada proceso, cuya entrada es el PID del proceso.



Procesos: File Descriptors

- Los procesos tienen ASOCIADOS los archivos abiertos que están usando. Estos SE IDENTIFICAN por un NUMERO entero positivo denominado file descriptor.
- SE ALMACENAN en una tabla en el kernel llamada FILE DESCRIPTOR TABLE --> UNA POR PROCESO.
- El file descriptor está del lado del kernel, el proceso no sabe nada del archivo en sí.
- EXISTEN 3 file descriptors que son creados cuando se crea un proceso:
 - fd = 0 → llamado Stdin de solo lectura.
 - fd = 1 → llamado Stdout de solo escritura.
 - fd = 2 → llamado StdErr de solo escritura.



El API de Procesos Resumida

Argv: array de punteros a caracteres que contiene los argumentos de un programa

Argc: entero que contiene el número de argumentos que se han introducido.

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

```

char *argv[2];

argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
    close(0);
    open("input.txt", O_RDONLY);
    exec("cat", argv);
}

```

→ La cadena representa el nombre del programa que se va a ejecutar. En este caso, es el comando CAT que se utiliza para leer y concatenar archivos.

→ El fork() devuelve 0 en el proceso hijo y el PID del hijo en el proceso padre. Por lo tanto, la condición se cumple solo en el proceso hijo y el código solo se ejecutará en el hijo.

→ Cierra el descriptor de archivo 0, que es por convención la entrada estandar (stdin). Cerrar este descriptor libera el numero 0 para ser reutilizado.

→ Cierra el descriptor de archivo 0, que es por convención la entrada estandar (stdin). Cerrar este descriptor libera el número 0 para ser reutilizado.

→ Se abre el archivo en modo de solo lectura (O_RDONLY). Como el descriptor de archivo 0 ha sido cerrado previamente, open asignará el descriptor de archivo 0 al archivo input.txt.

Esto significa que cualquier lectura de la entrada estandar en este proceso hijo ahora leerá desde input.txt en lugar de la entrada estandar habitual (como el teclado).

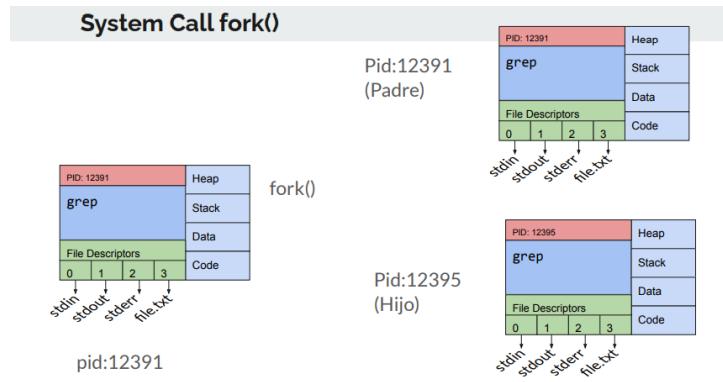
→ Reemplaza la imagen del proceso actual con una nueva, en este caso el programa cat

exec no altera la tabla de descriptores de archivo. Si tiene éxito, no regresará al código original. En lugar de eso, el proceso comenzará a ejecutar el programa cat a partir de la primera instrucción. Si exec falla, entonces la siguiente línea de código (si existiera) se ejecutaría, pero en este caso no hay ninguna línea de código después de exec.

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. El nuevo proceso es una copia exacta de proceso padre, cuya única diferencia es su pid.

- Crea y asigna una nueva entrada en la Process Table para el nuevo proceso.
- Asigna un número de ID único al proceso hijo (pid).
- Crea una copia del espacio de memoria del proceso padre.
- El hijo hereda los File Descriptors del proceso padre
- Devuelve el número de ID del hijo al proceso padre, y un 0 al proceso hijo



```

int pid = fork();
if(pid == 0){
    printf("child: exiting\n");
} else if(pid > 0){
    printf("parent: child=%d\n", pid);
} else {
    printf("fork error\n");
}

```

Solo lo ejecuta el hijo

Solo lo ejecuta el padre

wait()

Se utiliza para esperar un cambio de estado en un proceso hijo del proceso que realiza la llamada, y obtener información sobre el proceso hijo cuyo estado ha cambiado. Retorna el pid del proceso que sufrió el cambio de estado y copia el estado de salida del proceso en cuestión en la dirección wstatus. Si no se desea info del estado se le pasa la dirección 0.

Se considera que un cambio de estado es:

- El hijo termina su ejecución
- El hijo fue parado tras recibir un signal
- El hijo continúa su ejecución tras haber recibido un signal

Si hay PROCESOS QUE ESPERAR, esta llamada es BLOQUEANTE.

```
#include <sys/wait.h>
pid_t wait(int *_Nullable wstatus);

int pid = fork();
if(pid == 0){
    printf("child: exiting\n");
    exit(0);
} elseif(pid > 0){
    printf("parent: child=%d\n", pid);
    pid = wait((int *) 0);
    printf("child %d is done\n", pid);
} else {
    printf("fork error\n");
}
```

getpid() y getppid()

```
#include <unistd.h>
```

`pid_t getpid(void);` Devuelve el process id del proceso llamador.

`pid_t getppid(void);` Devuelve el process id del padre del proceso llamador

```

exec(filename, argv)
#include <unistd.h>

int execve(const char *pathname, char *const _Nullable argv[], char *const
_Nullable envp[]);

```

Ejecuta el programa al que hace referencia el nombre de pathname, con los argumentos que se le envían por argv[]. Esto hace que el programa que está ejecutando actualmente por el proceso llamador sea reemplazado por un nuevo programa, con una pila, un heap y segmentos de datos (inicializados y no inicializados) recién inicializados. OJO que no se crea ningún proceso solo se reemplaza Un programa en ejecución por otro.

¿Qué hace y que no hace execve?:

- Carga el programa y reemplaza el espacio de memoria del proceso por el del nuevo programa
- **No** crea un nuevo proceso.
- **No** cambia el pid
- **No** modifica los File Descriptors (el nuevo programa, ve la tabla como estaba antes del execve)

```

#include<stdio.h>
#include<unistd.h>
int main (){
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello";
    argv[2] = 0;

    printf("este es un proceso cuyo pid es: %i \n",getpid());
    printf("Ahora lo vamos a pisar y hacer que el mismo proceso\n");
    printf("ejecute un programa perdiendo todo y sustituyendo \n");
    printf("todo por el nuevo programa que se iniciara a ejecutar \n");

    execve("/bin/echo", argv, NULL);

    printf("exec error\n");
    printf("esto nunca se ejecuta\n");
}

int main (){
    char *argv[3];
    int pid = fork();
    if(pid == 0){
        argv[0] = "echo";
        argv[1] = "hello yo soy el comando echo!!!";
        argv[2] = 0;

        execve("/bin/echo", argv, NULL);
        printf("esto no debe ejecutarse\n");
    } else if(pid > 0){

        printf("parent: child=%d\n", pid);
        pid = wait((int *) 0);
        printf("child %d is done\n", pid);

    } else {
        printf("fork error\n");
    }
}

```

exit()

Generalmente un proceso tiene dos formas de terminar:

- La anormal: a través de recibir una señal cuya acción por defecto es terminar el programa.
- La normal: a través de invocar a la System call exit().

kill(pid)

Envía una señal para terminar el proceso especificado por su ID (pid), permitiendo la terminación controlada de procesos. Si el hijo mata a su padre, queda huérfano y es “reparented” por el proceso INIT (El proceso con PID 1).

dup()

```

#include <unistd.h>

int dup(int fildes);

int dup2(int fildes, int fildes2);

```

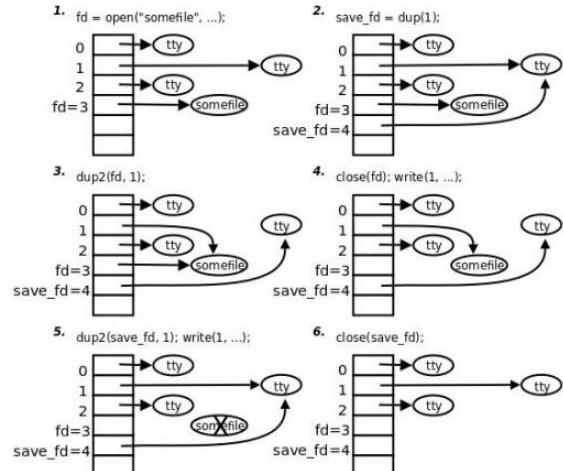
Duplican un file descriptor, el cual puede ser utilizado indiferentemente entre él y el duplicado. `dup()` retorna un nuevo file descriptor que es copia del enviado como parámetro, obteniendo el primer file descriptor libre que se encuentre en la File Descriptor Table.

```
#include<stdio.h>
#include<unistd.h>

int main (){
    int dup_fd;
    char msg1[]="Se escribe en el stdout\n";
    char msg3[]="se escribe desde el file descriptor duplicado\n";

    dup_fd=open("somefile", ...);
    write(1,msg1,sizeof(msg1)-1);
    write(dup_fd,msg3,sizeof(msg3)-1);
    close(dup_fd);

    write (1, msg1, sizeof(msg1)-1);
}
```



pipe()

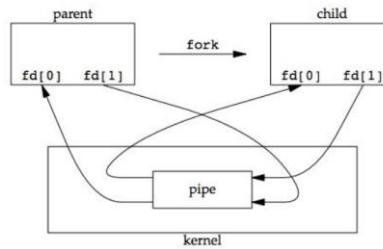
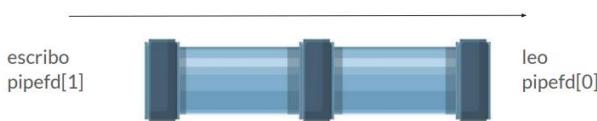
```
#include <unistd.h>

int pipe(int pipefd[2]);
```

Es un **pequeño buffer en el kernel**. Crea un CANAL DE COMUNICACION entre PROCESOS, permitiendo que uno lea datos por un extremo y el otro escriba datos por el otro extremo, es decir, un par de file descriptors conectados (un vector de 2 enteros). Por ejemplo, Si tengo que hacer un ping pong, necesito dos pipes. La lectura bloquea hasta que haya algo que leer y la escritura bloquea hasta que se pueda escribir.

!!!ES UN CANAL UNIDIRECCIONAL!!!

Int pipefd[2]

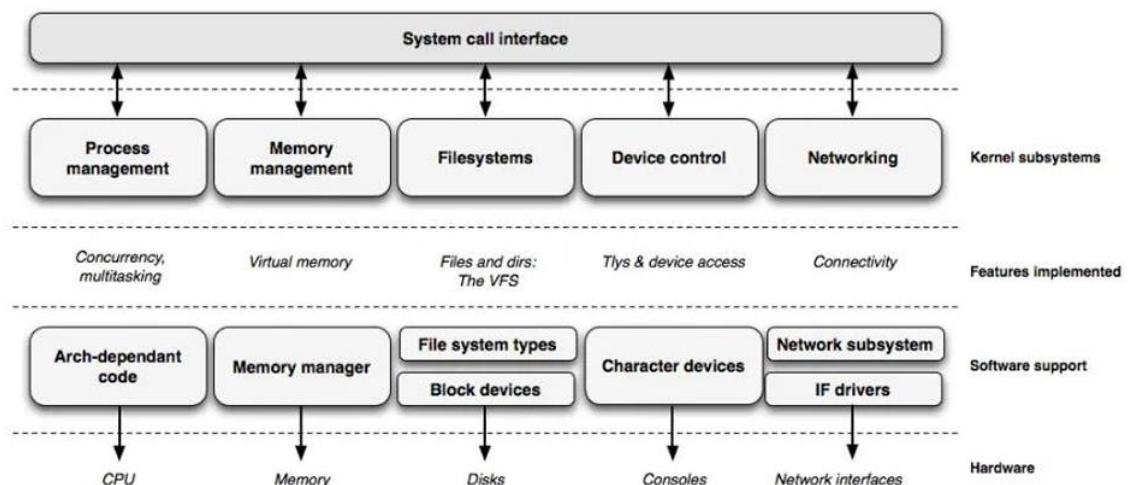
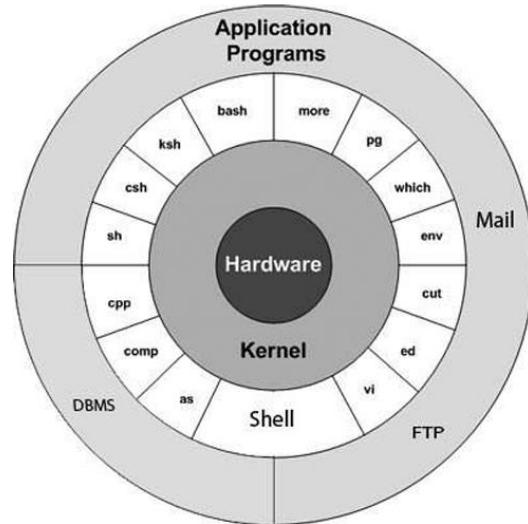


→ Acá estoy conectando el standard input de un pipe al standard output de otro pipe, entonces se mezcla hijo y padre.

KERNEL

- El kernel o núcleo es la BARRERA entre las APLICACIONES DE USER y el HARDWARE.
- Programa especial que PROPORCIONA SERVICIOS a los programas en ejecución
- La capa de kernel rodea todo el hardware, ninguna App de usuario puede atravesar directamente hacia el hardware.

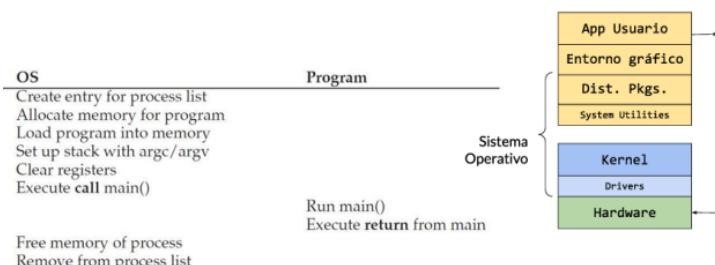
Cada programa en ejecución se llama proceso. Los servicios o las interfaces entre el procesador y el kernel y las aplicaciones de usuario que brinda el kernel se denominan llamadas al sistema (System Calls).



Ejecución Directa

- Los programas tienen ACCESO COMPLETO a todo el hardware del sistema, sin ningún tipo de control.
- El OS provee rutinas a modo de biblioteca
- Actúa principalmente como PEGAMENTO, pero NO PROVEE AISLAMIENTO.
- EJ: DOS

Se crea una entrada en la lista de procesos, se almacena la memoria para el programa, carga el programa en la memoria, se setea el stack con los parámetros del main, se vacían los registros y se ejecuta la llamada al main. Luego de parte del programa se corre el main hasta ejecutarse su return. Por último, se libera la memoria del proceso y se remueve el proceso de la lista.



Problema de la ejecución directa: No provee aislamiento y protección

- Nada garantiza que un programa no modifique a otro (o al OS mismo)
- Acceso directo al storage permanente: Nada garantiza que un programa no dañe al filesystem.
- Nada garantiza que, al finalizar un programa, este se interrumpa

Entonces surge:

Ejecución Directa Limitada

- El hardware va a limitar ciertas operaciones
- Un programa especial y privilegiado va a arbitrar las operaciones riesgosas --> El Kernel

**SI SE QUIERE EJECUTAR ALGO O USAR UN RECURSO, SE DEBE PEDIR PERMISO AL KERNEL.
EL ROL CENTRAL DEL OS ES LA PROTECCION**

Antes de darse la ejecución, el sistema operativo en modo kernel inicializa la trap table. Antes de ejecutarse el main, se llena el stack del kernel con registros/PC. El hardware restaura estos registros y entra en modo usuario, para que el programa pueda correr el main.

User-space vs. Kernel-space

Para que pueda existir un kernel, la máquina debe tener un MODO DUAL: es la existencia en el procesador de dos conjuntos de instrucciones distintas. Ese conjunto de instrucciones se puede usar según en qué bit tenga encendido para saber si estoy en modo de usuario o modo kernel → esto es el ring. Permite chequear qué instrucción estás ejecutando.

User Space: Donde habita el proceso

- Una aplicación puede ejecutarse SOLO en modo usuario (por ejemplo, sumar números, etc.)
- NO PUEDE ejecutar instrucciones privilegiadas
- Se dice que ejecuta en user space
- Cada proceso o programa en ejecución posee memoria con las instrucciones, los datos, el stack y el heap.
- Los procesos se ejecutan en un contexto AISLADO, PROTEGIDO y RESTRINGIDO.

Kernel Space: Donde habita el Kernel

- Ejecuta en MODO SUPERVISOR.
- PUEDE ejecutar instrucciones PRIVILEGIADAS.
- se dice que se ejecuta en el espacio del kernel.

CPU: El procesador

Actúa como “policía de ejecución modes” → Es el hardware quien, en última instancia, controla que los programas (user mode) se comporten correctamente. Es el mecanismo de hardware es quien separa en dos sets de instrucciones, el privilegiado y el no privilegiado. Luego

- El hardware PROVEE el MECANISMO de PROTECCION.
- El software (kernel) DETERMINA la POLITICA de PROTECCION y configura el hardware.

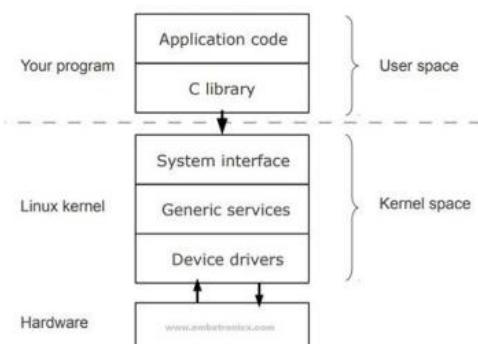
Ejemplos:

- RISC-V tiene tres modos en los que la CPU puede ejecutar instrucciones: modo máquina, modo supervisor y modo usuario.
- X86 tiene cuatro modos en los que la CPU puede ejecutar instrucciones, solo usa 2: modo supervisor y modo usuario.

En modo supervisor, la CPU puede ejecutar instrucciones privilegiadas: por ejemplo, habilitar y deshabilitar interrupciones, leer y escribir en el registro que contiene la dirección de una tabla de páginas, etc.

Comunicación Kernel/User Space

Una aplicación al querer invocar una función del Kernel, DEBE realizar una transición al mismo ya que desde ella no puede invocar directamente a la función deseada. Esto se logra gracias a una INSTRUCCION ESPECIAL que cambia de modo usuario a Kernel (RISC-V: ecall, x86: int 0x80) Ya en modo Kernel, se puede validar los argumentos de la llamada al sistema, decidir si la aplicación tiene permisos para realizar la operación solicitada y luego denegarla o ejecutarla.



ES IMPORTANTE QUE EL KERNEL CONTROLE EL PUNTO DE ENTRADA PARA LAS TRANSICIONES AL MODO SUPERVISOR; SI LA APLICACIÓN PUDIERA DECIDIR EL PUNTO DE ENTRADA AL NÚCLEO, UNA APLICACIÓN MALICIOSA PODRÍA, POR EJEMPLO, INGRESAR AL NÚCLEO EN UN PUNTO DONDE SE OMITE LA VALIDACIÓN DE ARGUMENTOS.

Modo Dual de Operaciones

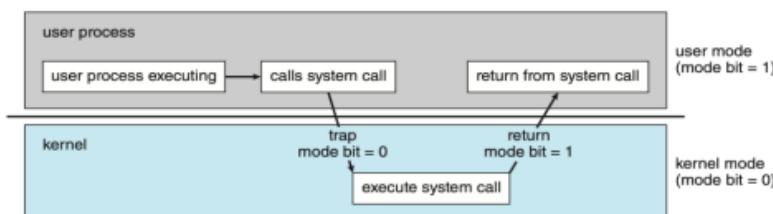


Figure 1.13 Transition from user to kernel mode.

El proceso se está ejecutando, se llama a una syscall entonces se produce una interrupción y se pasa a modo kernel. Se ejecuta la syscall y antes de cerrarse el kernel, vuelve al modo usuario. La syscall no se llama así nomás, se debe cumplir una serie de pasos, sino salta excepción. El bit de modo debe estar proporcionado por el hardware, sino no se puede hacer un kernel.

System Calls

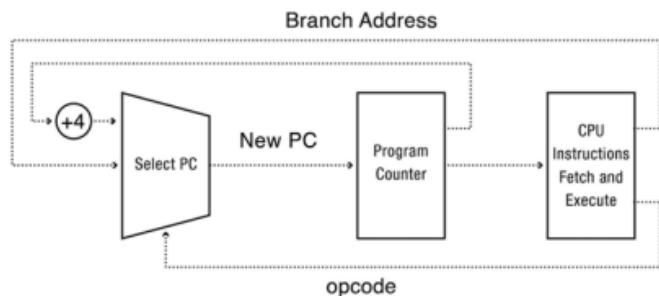
Son puntos de entrada controlados al kernel, permitiendo a un proceso solicitar que el kernel realice alguna operación en su nombre. El kernel expone una gran cantidad de servicios accesibles por un programa vía la Application Programming Interface (API) de system calls.

- Una Syscall es un servicio que provee el kernel.

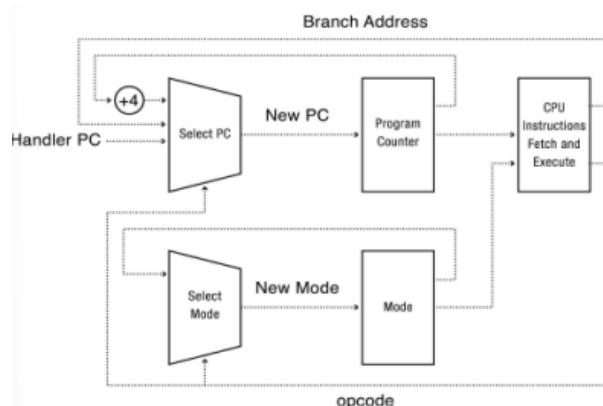
- CAMBIA el modo del procesador de user mode a kernel mode ==> La CPU podrá acceder al área protegida del kernel.
- El conjunto de system calls es FIJO. Cada system call esta identificada por un único número, invisible al programa.
- Cada system call DEBE tener un CONJUNTO de PARAMETROS que especifican información que debe ser transferida desde el user space al kernel space.

Mecanismos de protección del Hardware

En ejecución directa no hay ningún control, ya que luego de la instrucción se va a “select PC”, se lee lo que hay que hacer y se le suma al Program Counter, para luego volver a iniciar el ciclo. Así cualquier instrucción es ejecutada.



Entonces para que pueda haber ejecuciones privilegiadas, es que por hardware se controle el modo en que la instrucción puede ejecutarse y el modo en que está el hardware en ese momento (si está en modo usuario no se va a poder ejecutar en modo supervisor). Esto es el modo dual de operaciones.



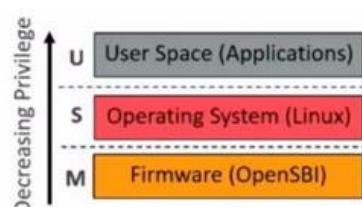
El kernel protege las aplicaciones y los usuarios del sistema operativo mediante:

Instrucciones Privilegiadas

Toda instrucción que el modo usuario no puede ejecutar. Si insiste en ejecutar, el procesador genera una excepción, o falla de alguna manera.

Modos en Risc-v

- Cada nivel de privilegio puede ejecutar distintas instrucciones. Machine: wfi, Supervisor: crrr, crrw, csrrw, User: la, li, etc.
- El encoding dice en qué modo estás. U: User Space, S: Supervisor, M: Machine mode.
- Hay distintos tipos de instrucciones que se pueden ejecutar solo en determinado modo.

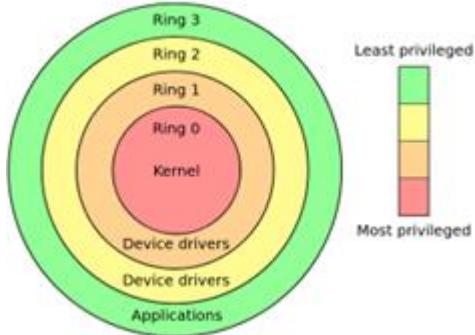


Si quiero ejecutar una instrucción que no está dentro de ese determinado modo, no puedo, me va a tirar error. Va a chequear que los bits que están en un registro específico sean los del modo que se necesita

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

Modos en x86

- Cada ring puede ejecutar una determinada cantidad de instrucciones. Esto se detecta por hardware cada vez que una instrucción se ejecuta. El ring 1 y 2 no se usan ni en Unix ni Linux.



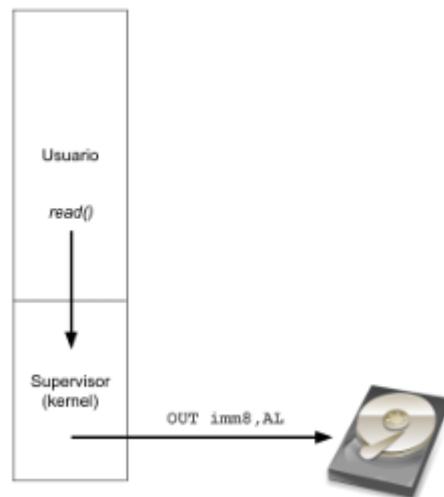
Ej: leer el disco en x86

Instrucción OUT

Output to Port: Transfiere un byte de datos o una palabra de datos desde el registro (AL, AX o EAX), dado como el segundo operando, al puerto de salida direccionado por el primer operando.

--> OUT imm8, AL

Espacio de direcciones



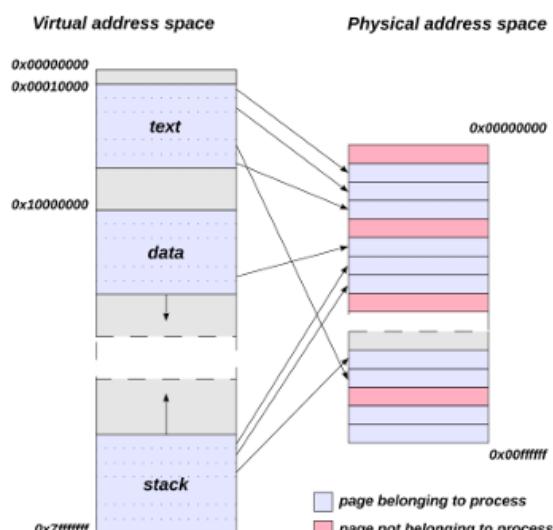
Protección de Memoria

Mecanismo que se transmite por hardware que determina que parte de la memoria pertenece a una memoria y que parte no. El mapeo se realiza mediante una tabla de páginas. La tabla contiene para cada página mapeada:

- La correspondencia Virtual → Física
- Flags y configuración de cada página

Ademas, se lanzará una excepción en caso de:

- Acceder a una página no de usuario
- Escribir a una página de solo lectura
- Acceder a una página no mapeada
- Ejecutar una página no ejecutable



Timer Interrupts

¿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.
- Esta interrupción es necesaria para que en algún momento si nadie despertó o pidió al kernel algo, el kernel se despierte y decida si continuar con lo que se está haciendo para hacer otra cosa. Si no existiese el timer, es como que si proceso nunca terminara y tomase el control.

Modos de Transferencia

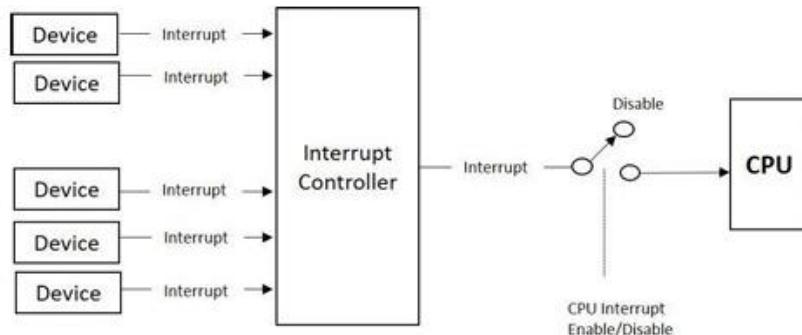
Una vez que el kernel pone a un proceso de usuario en un entorno aislado (sandbox), la próxima pregunta es: ¿cómo se transiciona entre un modo y otro?

User Mode to Kernel Mode

Las cosas que inducen un cambio de UM a KM son:

Interrupciones (dispositivos I/O y el timer)

- Señal asincrónica hacia el procesador avisando que algún evento externo requiere su atención

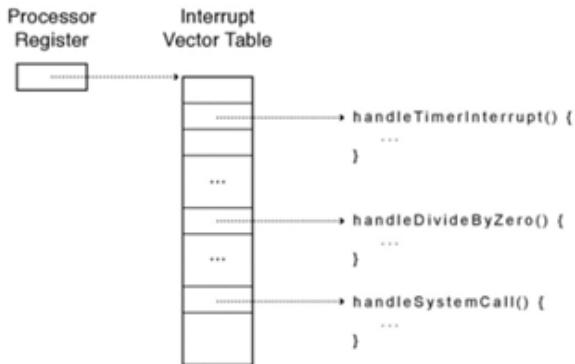


La transición a la interrupción, en general, causa un cambio de modo user → kernel.

Se puede setear al procesador de modo tal que, si hay una interrupción, no se interrumpa el CPU. Cada interrupción tiene un número asociado. El orden de importancia es:

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

El kernel mantiene una tabla de interrupciones y sus handlers específicos



→ **Vector de Interrupciones:** Se puede imaginar un array que vive en la memoria. Cada entrada en este array se asigna a un número de interrupción. Cada entrada contiene la dirección de una función que la CPU comenzará a ejecutar cuando se reciba esa interrupción junto con algunas opciones, como en qué nivel de privilegio se debe ejecutar la función del controlador de interrupciones.

Excepciones del Procesador

- Una excepción es un evento de hardware causado por una aplicación de usuario que causa la transferencia del control al Kernel.
- Se suele manejar con un mecanismo similar (o igual) que las interrupciones.

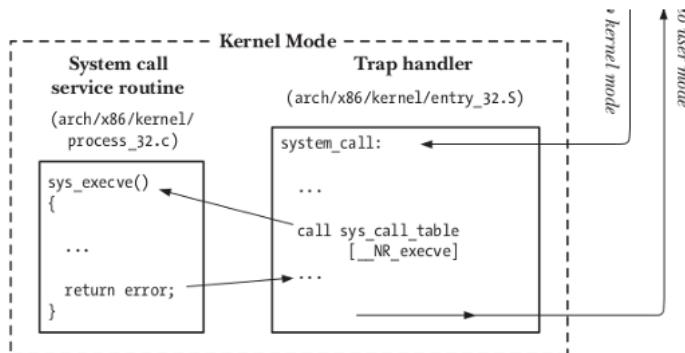
System Calls

- ¿Cómo inducimos una transición a una función (system call), pasando de modo Usuario a modo Kernel?
 - Es lo mismo que inducir un dispositivo de I/O al solicitar una interrupción.
 - Usamos el mecanismo interrupciones/excepciones/traps para ejecutar una System Call

En x86:

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

- El programa realiza un llamado a una system call mediante la invocación de una función wrapper (envoltorio) en la biblioteca de C.
- Dicha función wrapper tiene que proporcionar todos los argumentos al system call trap_handling. 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.
- 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).
- La función wrapper ejecuta una instrucción de código máquina llamada trap machine instruction (int 0x80 o SYSCALL), 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.



En respuesta al trap, el kernel invoca su propia función llamada `system_call()` (`arch/i386/entry.s`) para manejar esa trap. Este manejador:

- Graba el valor de los registros en el stack del kernel.
- Verifica la validez del número de system call.
- Invoca el servicio correspondiente a la system call llamada a través del vector de system calls, el servicio realiza su tarea y finalmente le devuelve un resultado de estado a la rutina system_call().
- Se restauran los registros almacenados en el stack del kernel y se agrega el valor de retorno en el stack. Se devuelve el control al wrapper y simultáneamente se pasa a user mode.
- Si el valor de retorno de la rutina de servicio de la system call da error, la función wrapper setea el valor en errno (error number).

Kernel Mode to User Mode

Continua luego de una interrupción, excepción o syscall. Hay un cambio de contexto (timer interrupt) entre diferentes procesos, creándose un nuevo proceso.

Clasificación de Kernels

Monolithic Kernel

- El kernel y el sistema operativo corren al mismo tiempo
- Usado mayormente cuando la seguridad NO ES PRIORIDAD

Microkernel

- Versión derivada del Kernel monolítico.
- El kernel puede hacer diferentes tareas y no requiere de un GUI adicional.
- Distintos servidores sirven para los drivers, los filesystems, las aplicaciones, etc.
- Hay ring 0, 1, 2 y 3. (En risc-v no se podría hacer)

Nano Kernel

- Pequeño tipo de kernel
- Responsable de la abstracción del hardware
- Utilizado en aquellos casos donde la mayoría de las funciones se configuran externamente.

Exo Kernel

- Pequeño tipo de kernel
- Responsable de la abstracción del hardware
- Utilizado cuando se prueba un proyecto interno y se actualiza a un tipo de kernel eficiente.

Hybrid Kernel

- Combinación entre microkernel y el monolítico
- Usado principalmente en Windows y macOS de Apple.
- Extrae el controlador y mantiene los servicios del sistema dentro del núcleo.

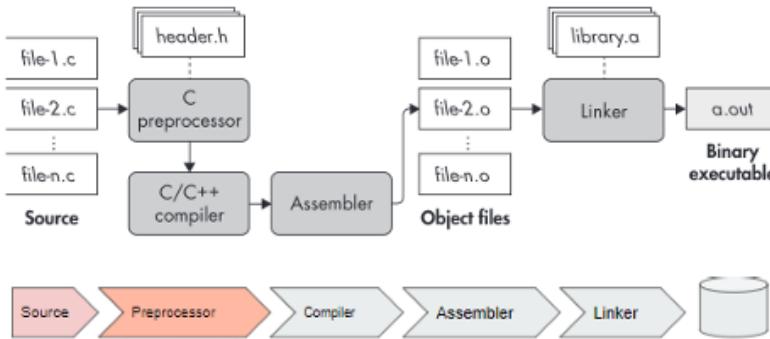
EL PROCESO

De programa a proceso

El programa:

- Programador edita código fuente
- El compilador compila el source code en una secuencia de instrucciones de máquina.
- El compilador guarda en disco esa secuencia, junto con datos y metadata(datos que describen otros datos) del programa: programa ejecutable.

- Edición → tabla de equivalencias
- Compilación



1. Fase de procesamiento (Preprocessor)

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

```
$gcc -E -P ejemplo.c
```

2. Fase de compilación (Compiler)

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

```
$gcc -S -masm=intel ejemplo.c
$cat
```

3. Fase de ensamble (Assembler)

A continuación, el ensamblador (as) traduce el archivo .s en instrucciones de lenguaje de máquina empaquetándolas en un formato conocido como programa objeto realocable. Este es almacenado en un archivo con extensión .o. En este punto es donde se genera código máquina real. Que pertenece al formato ELF (Executable and Linkable Format).

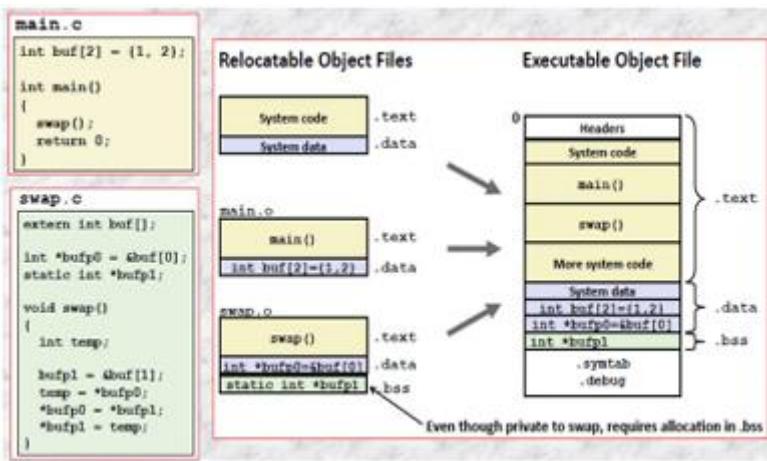
```
$gcc -c ejemplo.c
$file ejemplo.o
```

- ★ Un ARCHIVO REUBICABLE (relocatable) es un tipo de archivo que, como su nombre indica, PUEDE SER MOVIDO o reubicado en DIFERENTES AREAS de la memoria SIN AFECTAR su EJECUCION o funcionamiento.
- ★ Los ARCHIVOS OBJETO se compilan INDEPENDIENTEMENTE unos de otros, por lo que el ensamblador no tiene forma de conocer las direcciones de memoria de otros archivos objeto al ensamblar un archivo objeto.
- ★ Por eso los archivos objeto NECESITAN SER REUBICABLES; de esa manera, puedes enlazarlos juntos en cualquier orden para formar un ejecutable binario completo. Si los archivos objeto no fueran reubicables, esto no sería posible.

4. Fase de link edición (Linker)

Enlaza todos los archivos objetos en un único archivo binario ejecutable. Aquí el link editor ordena los archivos objetos en un único y coherente archivo ejecutable. Además, se enlazan las funciones de las bibliotecas estáticas (en C .a) y por último deja referencias simbólicas a las bibliotecas compartidas (shared library).

```
$gcc ejemplo.c
$file a.out
Salida:
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux
2.6.32-BuildID[sha1]=d0e23ea731bce9de65619cadd58b14ecd8c015c7,
not stripped
```

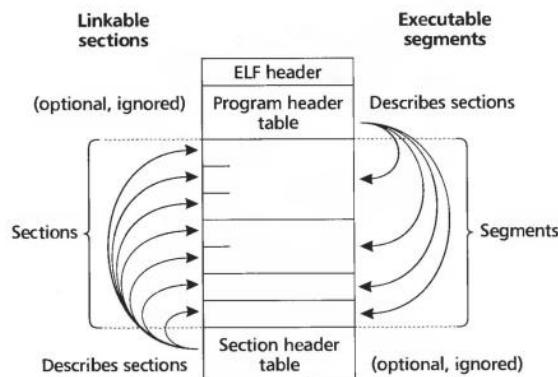


Formato ejecutable

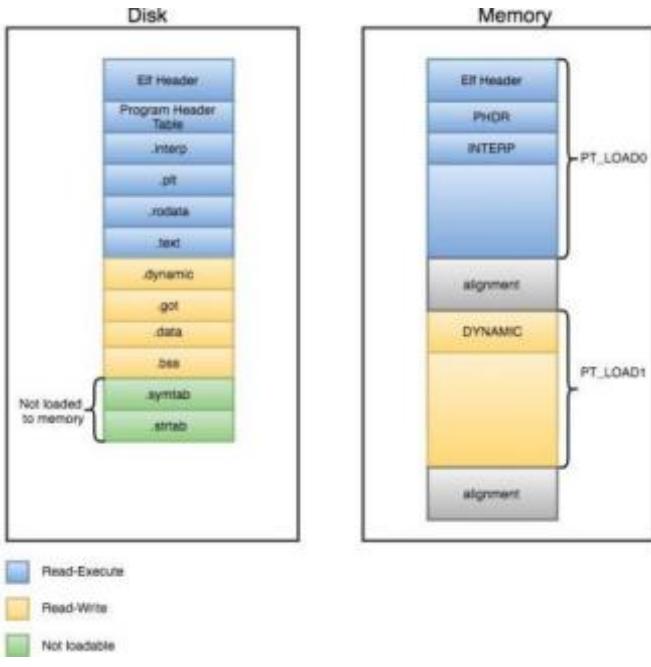
Un programa es un ARCHIVO que POSEE toda la INFORMACION de cómo CONSTRUIR un PROCESO en memoria.

- **Instrucciones de Lenguaje de Maquina:** Almacena el 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, así como otra información que es utilizada por ejemplo para debugg.
- **Bibliotecas Compartidas o Dinámicas:** 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 usado para cargar dicha biblioteca.
- **Otra información:** El programa contiene además otra información necesaria para terminar de construir el proceso en memoria.

ELF: Archivo Ejecutable en Linux



El archivo ELF además de tener el código, es como un “plano” del proceso que el Loader (parte del sistema operativo) va a usar para “construir” el proceso durante execve()



- **readelf** es una utilidad binaria de Unix que muestra información sobre uno o más archivos ELF. GNU Binutils proporciona una implementación de software gratuita.
- **elfutils** proporciona herramientas alternativas a GNU Binutils exclusivamente para Linux.[11]
- **objdump** proporciona una amplia gama de información sobre archivos ELF y otros formatos de objetos. objdump utiliza la biblioteca Binary File Descriptor como back-end para estructurar los datos ELF.
- The Unix **file** utility can display some information about ELF files, including the instruction set architecture for which the code in a relocatable, executable, or shared object file is intended, or on which an ELF core dump was produced.

Un Proceso

- ★ Es la EJECUCION de un PROGRAMA de aplicación con derechos restringidos; la abstracción que provee el Kernel del sistema operativo para la ejecución protegida.
- ★ Es una entidad abstracta, definida por el Kernel, en la cual los recursos del sistema son asignados

Un proceso incluye:

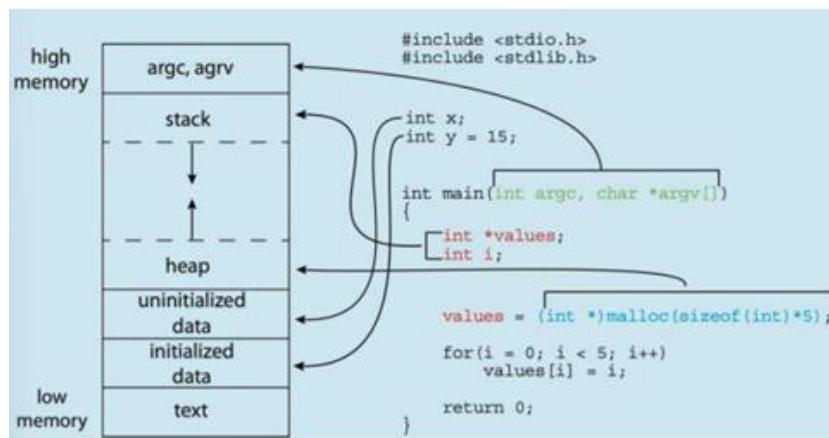
- Los Archivos abiertos
- Las señales(signals) pendientes
- Datos internos del kernel
- El estado completo del procesador
- Un espacio de direcciones de memoria

- Uno o más hilos de ejecución. Cada thread contiene:
 - Un único contador de programa
 - Un Stack
 - Un Conjunto de Registros
 - Una sección de datos globales

El 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

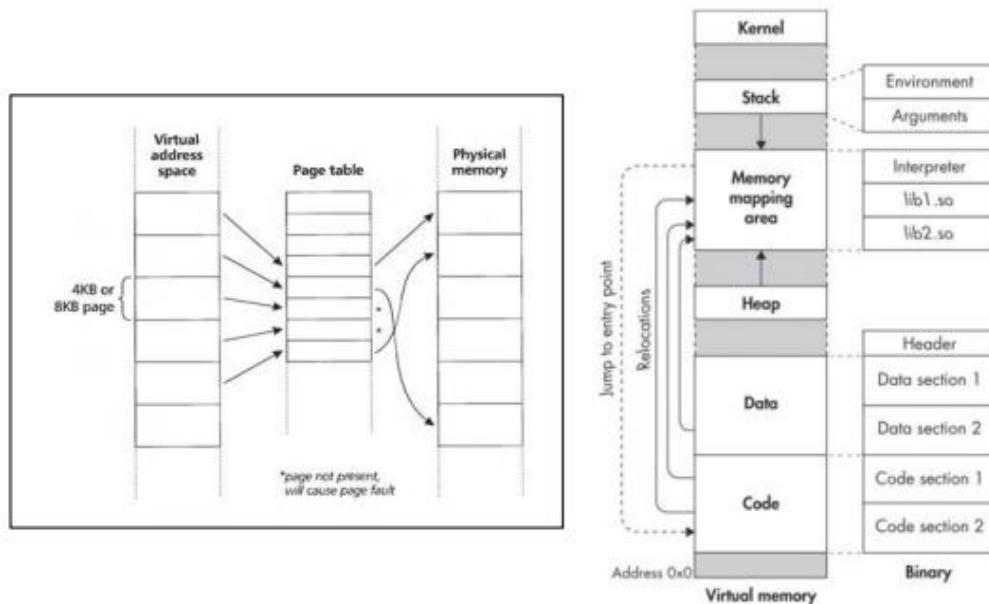
Un Proceso en memoria



El Loader usa el “plano del proceso” que es el archivo ELF y arma el espacio de direcciones.

Virtualización en memoria

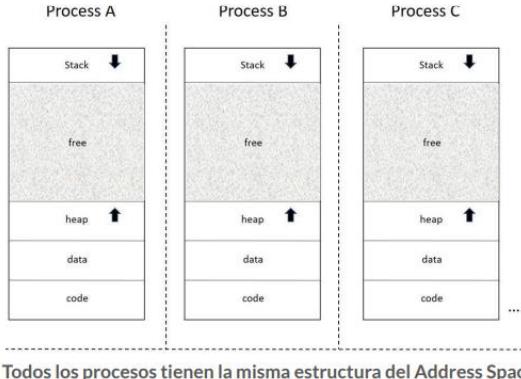
- Uno de estos mecanismos es denominado MEMORIA VIRTUAL --> abstracción por la cual la memoria física puede ser compartida por diversos procesos.
- Un componente clave de ella son las direcciones virtuales --> para cada proceso su memoria inicia en el mismo lugar, la dirección 0.
- Cada proceso piensa que tiene toda la memoria de la computadora para sí mismo, si bien obviamente esto en la realidad no sucede. El hardware traduce la dirección virtual a una dirección física de memoria.



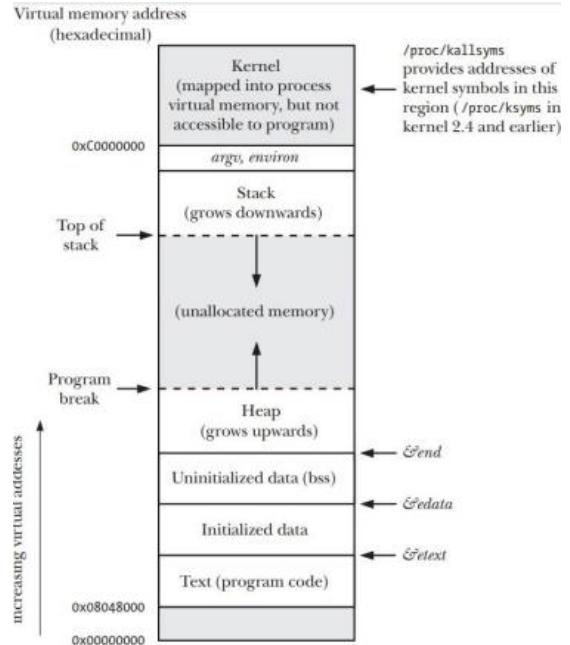
- ★ Traducción de direcciones: Se traduce una Dirección Virtual (emitida por la CPU) en una Dirección Física (la memoria). Este mapeo se realiza por hardware, más específicamente por Memory Management Unit (MMU).

La VIRTUALIZACION 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



Todos los procesos tienen la misma estructura del Address Space



brk() vs malloc()

Inicialmente el break del programa está ubicado justo en el final de los datos no inicializados. Después que brk() se ejecuta, el break es incrementado, el proceso puede acceder a cualquier memoria en la nueva área reservada, pero no accede directamente a la memoria física.

Esto se realiza automáticamente por el kernel en el primer intento del proceso en acceder al área reservada.

brk()

- Es una system call. Opera con bloques grandes (generalmente páginas completas)

malloc()

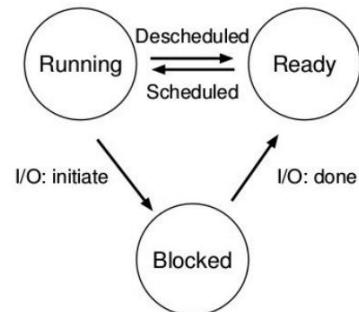
- Se implementa en espacio de usuario
- Utiliza brk().
- Maneja estructuras de datos propias (en user space) para optimizar la memoria provista por brk().
- Es el que nosotros usamos.

Virtualización del 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.

Estados del Proceso

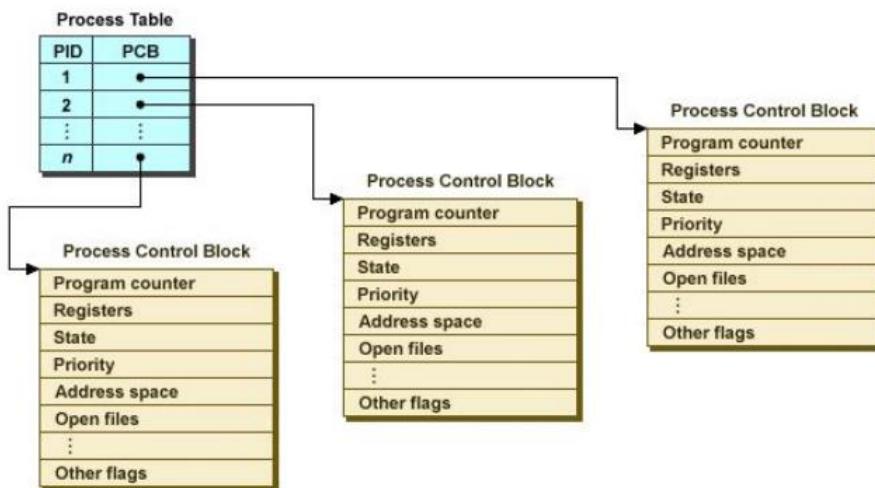
- **Corriendo** (Running): el proceso se encuentra corriendo en un procesador. Está ejecutando instrucciones.
- **Listo** (Ready): en este estado el proceso está listo para correr pero por algún motivo el SO ha decidido no ejecutarlo por el momento.
- **Bloqueado** (Blocked): en este estado el proceso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



El Contexto

- Información necesaria para describir completamente el estado de un proceso.
- Cada proceso posee su propio contexto y SOLO ve ese contexto.
- Formalmente, el contexto de un proceso es la unión de:
 - **User-level context** (memoria): Son las secciones que conforman el espacio de dirección virtual del proceso (Text, Data, Stack, Heap)
 - **Register context** (cpu): Registros del CPU
 - **System-level context** (estructuras del kernel):
 - **Process Table Entry**: La entrada en la tabla de procesos, que están representados por los **Process control block (PCB)**, el cual contiene muchos elementos de información asociados a un proceso particular
 - **Configuración de memoria**: define el mapeo de la memoria virtual vs memoria física del proceso. Eg. Page Tables
 - **Kernel Stack**: contiene los stack frames de las llamadas a funciones hechas dentro del kernel.
 - **La u area**: Estructura por proceso usada en sistemas antiguos (UNIX Version 6) para almacenar información específica de cada proceso, como descriptores de archivos abiertos, registros en modo usuario y manejo de señales. En desuso en sistemas modernos. Pero casi todo se incluye ahora en la Process Table Entry.

La tabla de PCB es generalmente un array o una lista enlazada



Depende mucho de la implementación del sistema operativo. Cosas que podría contener el PCB:

- Identificación: cada proceso tiene un identificador único o process ID (PID) y además pertenece a un determinado grupo de procesos.
- Ubicación del mapa de direcciones del Kernel del área 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.

- ★ El kernel almacena la lista de procesos en una lista circular doblemente enlazada llamada task list.
- ★ La task_struct en sí es una estructura compleja en Linux.

Spoiler alert! CONTEXT SWITCH: ¿Cómo se cambia de contexto, de un proceso a otro?

Se CAMBIA el User-level context (memoria), el Register context (cpu) y el System-level context (estructuras del kernel) de manera muy rápida y obteniendo la **ilusión de concurrencia**.

El kernel usa funciones como cualquier programa, pero no es seguro escribir datos del kernel en el User space por lo que no utiliza su stack.

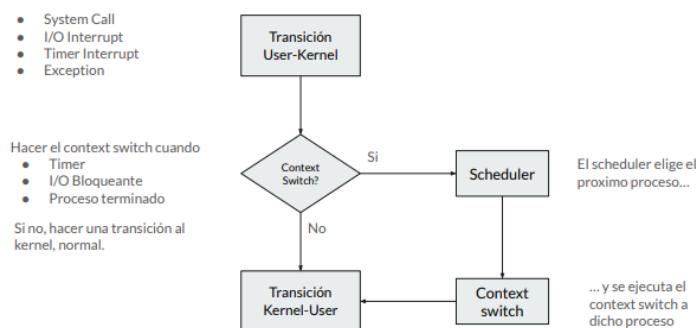
→ Solución: KERNEL STACK, un nuevo stack por proceso que está separado y protegido del usuario en la zona de memoria del kernel. Cada proceso tiene uno ya que si se encuentra “dormido” podía estar haciendo cosas completamente diferentes en el espacio Kernel antes de dormirse
Ej.: Un proceso se bloqueó ejecutando read(), Un proceso se bloqueó al hacer sleep() o Un proceso se bloqueó mientras configuraba más memoria virtual.

Al darse el context-switch, sucede lo siguiente:

- Se restauran los registros del proceso entrante (EAX, EBX, etc.)
- Se cambia toda la zona de usuario por la del proceso entrante (stack, heap, text, data)
- No se modifica la memoria del Kernel, porque el Kernel es común y compartido a todos los procesos, PERO: se apunta el stack pointer al kernel stack del proceso entrante

SCEDULLING

Diagrama de cambio de contexto



Cambio de contexto (resumido):

1. Comienza con una transición a Kernel-Space. Los procesos NO PUEDEN cambiar desde user space.
 - Esto puede ocurrir tanto por interrupciones, excepciones, como cuando el usuario llama a una system call.
 - En todos los casos se puede o no producir un cambio de contexto
2. Si SE DECIDE que hay que realizar un cambio de contexto, se invoca al SCHEDULER.
3. El scheduler ELIGE el siguiente proceso a ejecutar y llama a switch.
4. Switch le saca una foto al estado del kernel-space y del user-space (en general este último ya estaba guardado desde 1, y restaura el proceso candidato (que queda en Kernel-Space)
5. Regresa al User-Space del nuevo proceso

Si NO HAY CONTEXT SWITCH, se hace una transición al Kernel “Normal”. Las otras transiciones, no necesariamente indican que haya un cambio de contexto. Hay cambio de contexto cuando el OS interrumpe momentáneamente la ejecución de un proceso y restaura la ejecución de uno suspendido.

Transición User-Kernel en xv6

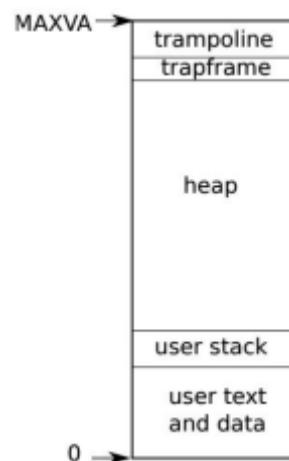
Se usa una estructura estándar de un proceso en C con dos particularidades:

Trampoline:

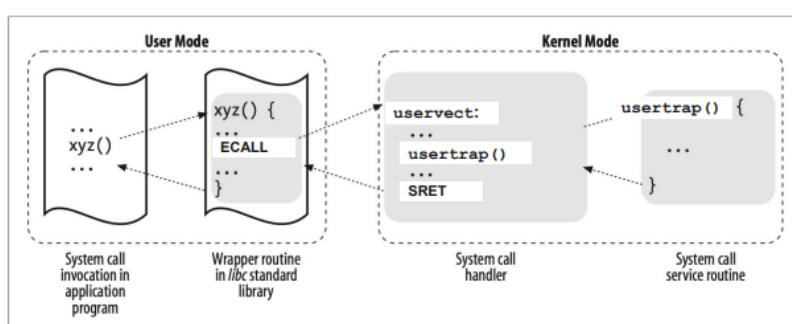
- Código assembler para transicionar de modo usuario a modo kernel.
- Única página del kernel que está siempre cargada en el user-space del proceso y única página compartida entre el address space del usuario y del kernel.
- El resto del kernel se carga solo cuando es necesario

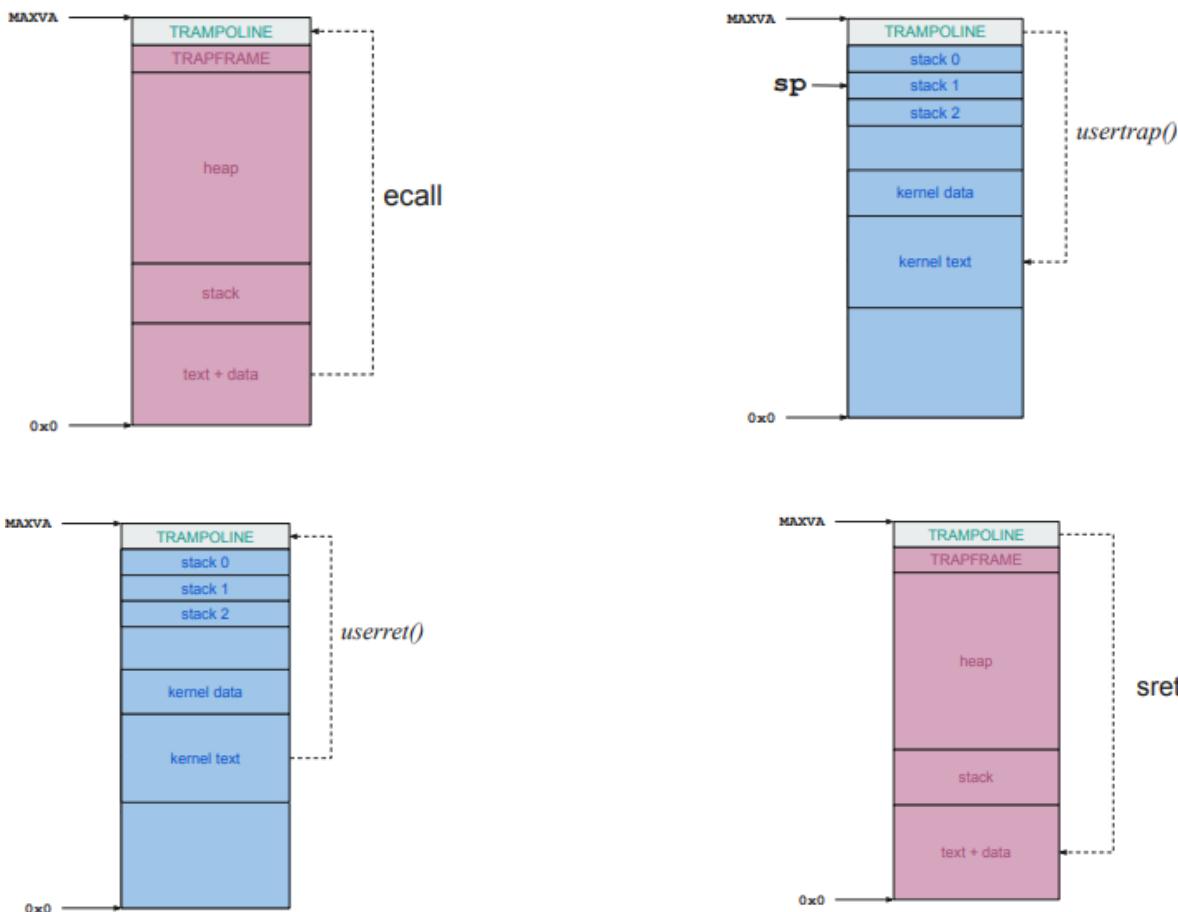
Trapframe:

Es un espacio donde el Kernel va a guardar el estado de todos los registros del procesador (la foto de los registros), previo a pasar al modo kernel, para poder restaurarlos luego leyendo directamente a este sector y colocándolos nuevamente en el procesador



Invocación de system call



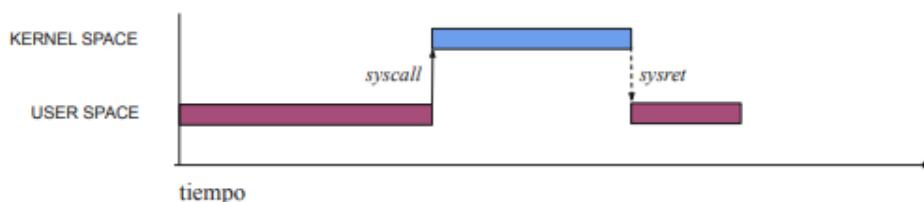


La transición inicia cuando se genera una interrupción por software mediante `ecall()`. El procesador “salta” al trampoline y cambia automáticamente a Kernel Mode. Código del Trampoline en este orden:

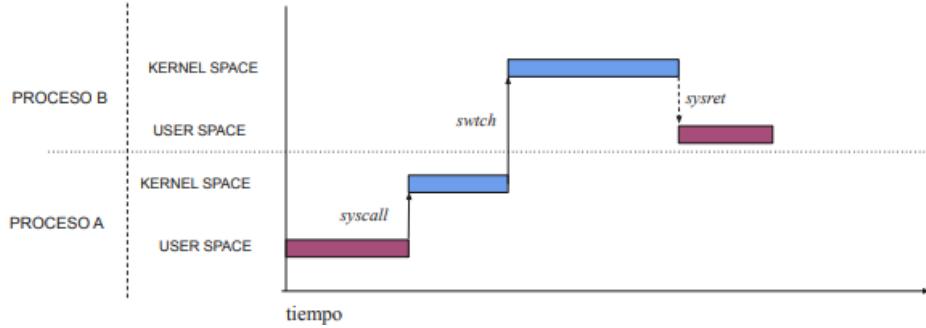
- 1) Guarda los registros del procesador en el trapframe
- 2) Cambia el registro satp, cambiando la tabla de páginas y efectivamente el address space.
- 3) El registro sp apunta al kernel stack del proceso actual.
- 4) Invoca la función `usertrap()`.
- 5) Invoca la función `userret()` en el trampoline
- 6) Cambia el registro satp, cambiando la tabla de páginas y el address space.
- 7) Restaura todos los registros guardados en el trapframe
- 8) Invoca `sret()` para volver al punto original

Cambio de contexto en xv6

Transición User-Kernel

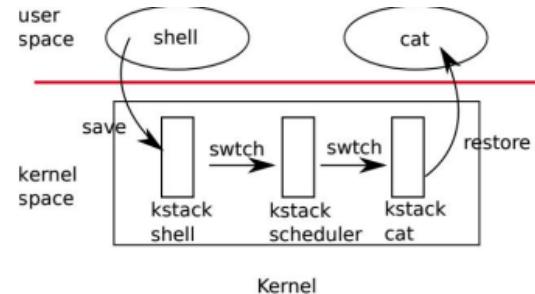


Context Switch en general



Context Switch en xv6

- Cada CPU tiene un proceso de kernel (sin user space) que es el scheduler.
- Un context switch en xv6 son 2 context switch: “Proceso A -> Scheduler” y luego “Scheduler -> Proceso”.
- Switch guarda el estado (registros) del CPU (seguimos en el kernel) en: (struct proc) p->context.
- Switch carga el estado del scheduler. Como hay un scheduler por CPU, este contexto se guarda en el struct cpu (struct cpu) c->context



Estado del CPU	Donde se guarda	
User-Space	(struct proc) p->trapframe	<ul style="list-style-type: none"> • Mapeado tambien en el user address space • Uno por proceso
Kernel en contexto del proceso	(struct proc) p->context	<ul style="list-style-type: none"> • Uno por proceso
Kernel en contexto del scheduler	(struct cpu) c->context	<ul style="list-style-type: none"> • Uno por CPU

Conceptos para entender scheduling

Multiprogramación

Más de un proceso estaba preparado para ser ejecutado en algún determinado momento, y el sistema operativo intercalaba dicha ejecución según la circunstancia. En esta era, múltiples procesos están listos para ser ejecutados un determinado tiempo según el S.O. lo decidiese en base a ciertas políticas de planificación o scheduling.

Time Sharing

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.

Planificación

Conceptualmente, es la coordinación de los procesos ejecutados en multiprogramación del OS, dada por el Planificador o Scheduler que forma parte del Kernel del OS.

Cuando hay múltiples cosas qué hacer ¿Cómo se elige cuál de ellas hacer primero?

- 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.

El Workload

- Es la CARGA de TRABAJO de un PROCESO corriendo en el sistema.
- Determinar COMO SE CALCULA el workload es FUNDAMENTAL para DETERMINAR partes de las POLITICAS de PLANIFICACION.
- Cuanto mejor es el cálculo, mejor es la política.

Podemos clasificar a los Schedulers por:

- **Interactivo:** necesita tiempos de respuesta bajos. Los procesos deben recibir atención frecuentemente. Ejemplo: Round Robin. Eg. Un procesador de texto
- **Por lotes(batch):** Los procesos pueden tardar más en completarse. Se optimiza el tiempo total de ejecución más que la respuesta inmediata. Ejemplo: FIFO, SJF. Eg. Calcular una liquidación de sueldos
- **Preemptivo:** el sistema puede interrumpir un proceso en ejecución para darle tiempo a otro. Esto permite más equidad y respuesta rápida (como en RR o STCF).
- **No Preemptivo:** una vez que un proceso empieza, no se interrumpe hasta que termina (como FIFO y SJF).

Supuestos y simplificaciones iniciales

- Las suposiciones que se harán para el cálculo del workload son más que irreales:
 - ✓ Cada proceso se ejecuta la misma cantidad de tiempo.
 - ✓ Todos los jobs llegan al mismo tiempo para ser ejecutados.
 - ✓ Una vez que empieza un job sigue hasta completarse.
 - ✓ Todos los jobs usan únicamente cpu (no realizan I/O).
 - ✓ El tiempo de ejecución (run-time) de cada job es conocido.

Métricas de Planificación

Objetivo: Comparar las distintas políticas de Scheduling. Se utilizará una única métrica llamada **turnaround time**. Que se define como el tiempo en el cual el proceso se completa menos el tiempo de arribo al sistema:

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

Debido a 2 (todos los jobs llegan al mismo tiempo) el $T_{arrival}=0$

→ El turnaround es una métrica que mide performance.

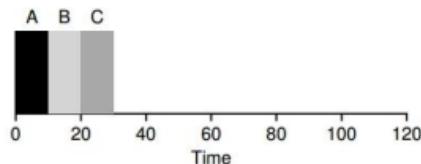
Políticas de Scheduling

Batch

First In, First Out (FIFO)

- Ventajas:
 - El más básico
 - Es simple.
 - Es fácil de implementar.
 - Funciona bárbaro para las suposiciones iniciales.

Por ejemplo, se tiene tres procesos A, B y C con $T_{arrival}=0$.



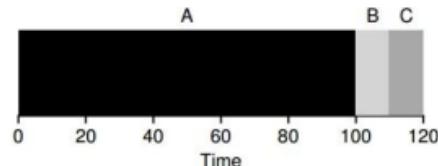
Si bien llegan todos al mismo tiempo llegaron con un insignificante retraso de forma tal que llegó A, B y C.

Si se asume que todos tardan 10 segundos en ejecutarse... ¿cuánto es el Taround?
 $(10+20+30)/3=20$

Ahora relajemos la suposición 1 y no se asume que todas las tareas duran el mismo tiempo.

Ahora A dura 100 segundos.

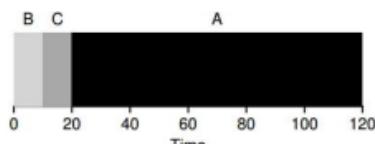
¿Cuánto es el Turnaround? $(100+110+120)/3=110$



Se termina dando algo llamado el **Convoy Effect** ya que se ejecutan los procesos todos uno después del otro, teniendo que esperar que el anterior termine.

Shortest Job First (SJF)

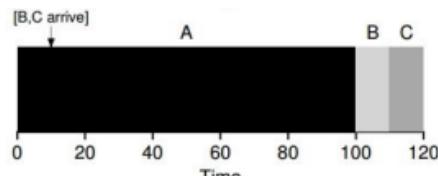
Para resolver el problema que se presenta en la política FIFO, se modifica la política para que se ejecute el proceso de duración mínima, una vez finalizado esto se ejecuta el proceso de duración mínima y así sucesivamente.



En el mismo caso de arriba, se mejora el turnaround time con el sencillo hecho de ejecutar B, C y A en ese orden: $(10+20+120)/3=50$

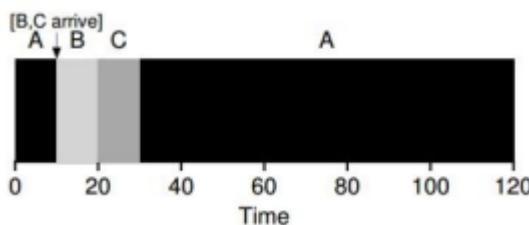
Si se relaja la suposición 2, en la cual no todos los procesos llegan al mismo tiempo, por ejemplo, llega el proceso A y a los 10 segundos llegan el proceso B y el proceso C.

¿Cómo sería el cálculo, ahora? $t=10$ seg
 $(100+110-10+120-10)/3=103.33$



Shortest Time-to-Completion (STCF)

Para poder solucionar este problema se necesita relajar la suposición 3 (los procesos se tienen que ejecutarse hasta el final). La idea es que el planificador o scheduler pueda adelantarse y determinar qué proceso debe ser ejecutado. Entonces cuando los procesos B y C llegan se puede desalojar (preemptivo) al proceso A y decidir que otro proceso se ejecute y luego retomar la ejecución del proceso A. El caso anterior, el de SFJ, es una política no-preemptiva.



Cuando un nuevo Job llega al sistema, el scheduler determina cuál de los jobs en el sistema (incluyendo al recientemente arribado) le falta menos tiempo para terminar, y elige a dicho proceso, desalojando al proceso actual. El cálculo para el turnaround time sería $(120-0+20-10+30-10)/3=50$

Interactiva

Round Robin (RR)

Utiliza una nueva métrica: **Tiempo de Respuesta**. surge con el advenimiento del time-sharing ya que los usuarios se sientan en una terminal de una computadora y pretenden una interacción con rapidez.

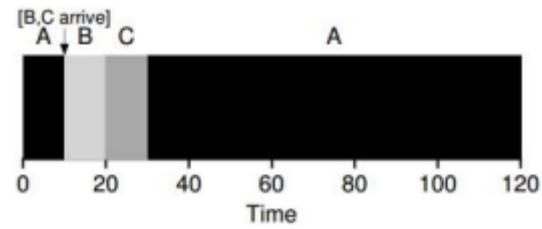
$$T_{response} = T_{firstrun} - T_{arrival}$$

El *Tresponse* del proceso A es 0.

El *Tresponse* del proceso B es... 0... llega en 10 pero tarda 10 (10-10)

El *Tresponse* del proceso C es... 10... llega en 10 pero termina en 20 (20-10)

En promedio el *Tresponse* es de 3.33 seg.

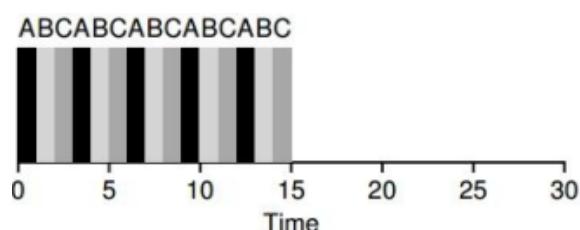


La idea del algoritmo RR es bastante simple: se ejecuta un proceso por un período determinado de tiempo (time slice) y transcurrido el período se pasa a otro proceso, y así sucesivamente cambiando de proceso en la cola de ejecución. También se llama time-slicing.

El time-slice es un múltiplo el timer interrupt.

Ejemplo: Si el timer interrupt es 10 ms, el time slice podrá ser 10ms, 20ms y cualquier n*10ms

Response time: $(0 + 1 + 2) / 3 = 1$



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.

Por ejemplo, si el tiempo de cambio de contexto está seteado en 1 ms y el time slice está seteado en 10 ms, el 10% del tiempo se estará utilizando para cambio de contexto. Sin embargo, si el time slice se setea en 100 ms, solo el 1% del tiempo será dedicado al cambio de contexto.

Acá podemos observar el código en xv6:

```

c->proc = 0;
for(;;) {
    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
        if(p->state == RUNNABLE) {
            p->state = RUNNING;
            c->proc = p;
            swtch(&c->context, &p->context);
            c->proc = 0;
            found = 1;
        }
    }
}

```

Algoritmo	Tipo de sistema	¿Preemptivo?	Tiempo de respuesta	Turnaround Time
FIFO	Batch	✗ No	✗ Malo (puede ser alto)	⚠ Variable (puede ser malo)
SJF	Batch	✗ No	✗ Malo (espera inicial larga)	✓ Excelente (mínimo promedio)
STCF	Batch	✓ Sí	✓ Bueno (procesos cortos responden rápido)	✓ Muy bueno (casi óptimo)
RR	Interactivo	✓ Sí	✓ Excelente (respuesta rápida inicial)	✗ Regular (turnaround más alto)

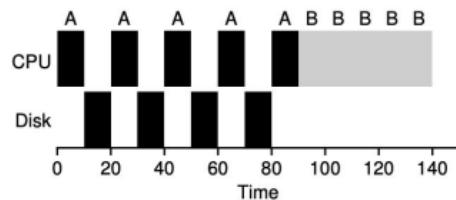
Considerando el I/O

Un proceso (un solo thread) puede estar haciendo dos cosas:

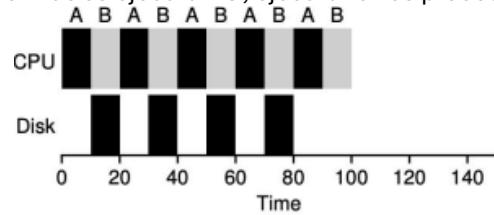
- Usando el CPU
- Esperando por un dispositivo de I/O

- ★ OBJETIVO: queremos que siempre haya alguien usando el CPU. ¡Si no, estamos desperdimando hardware!

Mal uso del CPU; ¡desperdiciamos recursos y perdemos tiempo de multitasking!



Una estrategia mejor: **Superponer (overlap)**: mientras se ejecuta I/O, ejecutar otros procesos



De esta manera se pueden pensar dos clases de programas:

I/O intensive

Usan mucho la entrada y salida, y el CPU se dedica principalmente a gestionar estos procesos. Eg. un file server, un reproductor de video que usa aceleración de video en una GPU.

- Si no tenemos cuidado, un proceso I/O intensive va a estar desperdiциando el CPU, esperando a los dispositivos de entrada y salida.
- Solución: Ejecutar muchos procesos I/O intensive simultáneamente.
- Notar que esta solución no funciona para el caso de muchos procesos CPU intensive. Si tenemos un proceso que usa el 100% del CPU, agregar más procesos simplemente hará que el CPU siga al 100%.

CPU intensive

Usan mucho las CAPACIDADES de CALCULO del CPU. Eg. cálculo de matrices, aplicaciones gráficas, entrenamiento de redes neuronales.

- No hay escapatoria, el proceso NECESITA el CPU.
- Todos estos procesos van a COMPETIR por el CPU, siendo desalojados por el timer de vez en cuando.

- En la actualidad, casi todas las aplicaciones CPU intensive tienen alguna forma de ACELERACION POR HARDWARE, esto es, mediante el uso de COPROCESADORES (eg. GPU).
- Al delegar el cálculo al coprocesador, la aplicación se transforma en I/O intensive (o más bien GPU intensive)



Los GPUs inicialmente usados para aplicaciones graficas se generalizaron para implementar algoritmos facilmente paralelizables. A esto se lo llama General-purpose computing on graphics processing units (GPGPU).

En la actualidad, las GPU son centrales en el entrenamiento de redes neuronales.

El GPU se transforma en un co-procesador del CPU. Libera al CPU de los cálculos, y transforma al proceso en un proceso I/O intensive.

Estrategias para procesos I/O Intensive

- ★ Asumiendo una calendarización perfecta: si el 20% del tiempo de ejecución de un programa es sólo cómputo y el 80% son operaciones de entrada y salida, con tener 5 procesos en memoria se pueden acomodar los procesos para utilizar el 100% de la CPU.
- ★ **El I/O es mucho más lento que las instrucciones.** Siendo un poco más realista se supone que las operaciones de E/S son bloqueantes (una operación de lectura a disco tarda 10 miliseg y una instrucción registro registro tarda 1-2 nanoseg), es decir, paran el procesamiento hasta que se haya realizado la operación de E/S.

Calculo probabilístico de utilización del CPU

Si se supone que un proceso permanece una fracción p bloqueado a la espera de E/S entonces la probabilidad de que en un instante determinado ese proceso este bloqueado es p . Ejemplo. Si el proceso se la pasa el 80% del tiempo bloqueado por E/S, la probabilidad de que en un instante determinado el proceso está bloqueado será de 0.8.

Si tenemos n procesos idénticos, la probabilidad de que todos ellos estén bloqueados simultáneamente es p^n .

Recíprocamente, la probabilidad de que exista al menos un proceso no bloqueado, y por lo tanto ejecutable es $1-p^n$

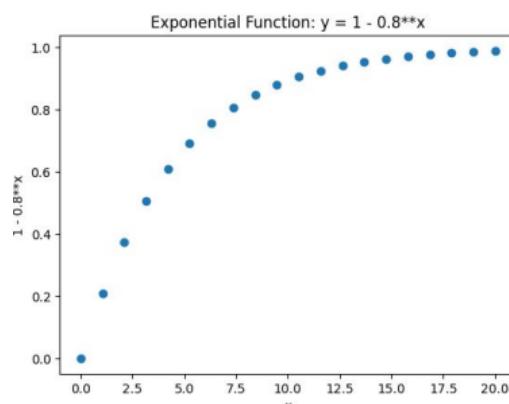
Esta fórmula es conocida como utilización de CPU:

$$\text{Utilización del CPU: } 1-p^n$$

Por ejemplo: si se tiene un solo proceso en memoria y este invierte un 80% del tiempo en operaciones de E/S el tiempo de utilización de CPU es $1 - 0.8 = 0.2$ que es el 20%.

Ahora bien, si se tienen 3 procesos con la misma propiedad, el grado de utilización de la cpu es $1 - 0.8^3 = 0.488$ es decir el 49% de ocupación de la cpu.

Si se supone que se tienen 10 procesos, entonces la fórmula cambia a $1 - 0.8^{10} = 0.89$ el 89% de utilización, aquí es donde se ve la IMPORTANCIA de la Multiprogramación.



Multi Level Feedback Queue

TECNICA de PLANIFICACION descripta inicialmente en los años 60 en un sistema conocido como Compatible Time Sharing System CTSS. Este trabajo en conjunto con el realizado sobre MULTICS llevó a que su creador ganara el Turing Award.

Intenta atacar dos problemas:

- Optimizar el turnaround time, lo cual se logra ejecutando primero los trabajos más cortos; desafortunadamente, el sistema operativo generalmente no sabe cuánto tiempo va a durar un trabajo, precisamente el conocimiento que requieren algoritmos como SJF (Primero el Trabajo Más Corto) o STCF (Primero el Trabajo Más Corto con Expiración).
- que el sistema sea responsive a los usuarios interactivos, minimizando el response time; sin embargo, algoritmos como Round Robin reducen el response time, pero son terribles para el turnaround time.

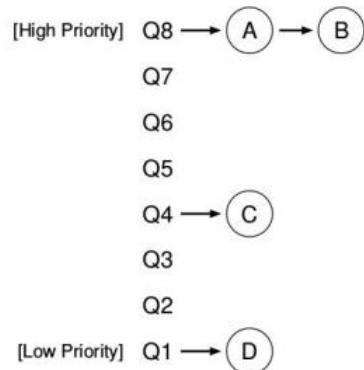
Las reglas básicas:

MLFQ tiene un conjunto de distintas colas, cada una de estas colas tiene asignado un nivel de prioridad.

REGLA 1: si la prioridad (A) es mayor que la prioridad de (B), (A) se ejecuta y (B) no.

REGLA 2: si la prioridad de (A) es igual a la prioridad de (B), (A) y (B) se ejecutan en Round-Robin.

La clave para la planificación MLFQ subyace entonces en cómo el planificador setea las prioridades. En vez de dar una prioridad fija a cada tarea, MLFQ varía la prioridad de la tarea basándose en su comportamiento observado.



Por ejemplo, si una determinada tarea repetidamente no utiliza la CPU mientras espera que un dato sea ingresado por el teclado, MLFQ va a mantener su prioridad alta, así es como un proceso interactivo debería comportarse.

Si, por lo contrario, una tarea usa intensivamente por largos periodos de tiempo la CPU, MLFQ reducirá su prioridad. De esta forma MLFQ va a aprender mientras los procesos se ejecutan y entonces va a usar la historia de esa tarea para predecir su futuro comportamiento.

Primer intento: ¿Como cambiar la prioridad?

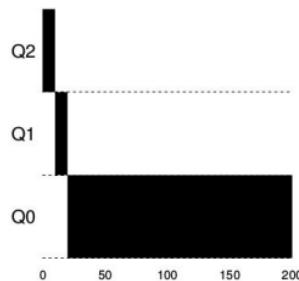
Para hacer esto hay que tener en cuenta nuestra carga de trabajo (workload): una mezcla de tareas interactivas que tienen un corto tiempo de ejecución y que pueden renunciar a la utilización de la CPU y algunas tareas de larga ejecución basadas en la CPU que necesitan tiempo de CPU, pero poco tiempo de respuesta.

REGLA 3: Cuando llega una tarea al sistema se la coloca en la cola de más alta prioridad.

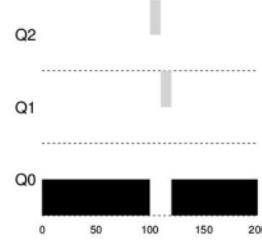
REGLA 4a: Si durante una ejecución una tarea usa su time slice completo entonces su prioridad se reduce en una unidad (baja a la cola directamente inferior)

REGLA 4b: Si una tarea renuncia al uso de la CPU antes de completar un time slice se queda en el mismo nivel de prioridad.

Ejemplo 1: Una única tarea con ejecución larga.



Ejemplo 1: Llega una tarea corta.

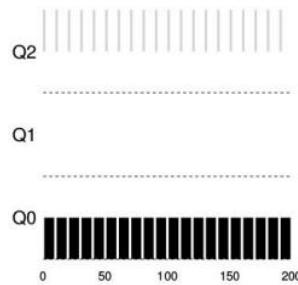


Existen 2 tareas, una de larga ejecución de CPU, A y B con una ejecución corta e interactiva. B tarda 20 milisegundos en ejecutarse.

De este ejemplo se puede ver una de las metas del algoritmo dado que no sabe si la tarea va a ser de corta o larga duración de ejecución, inicialmente asume que va a ser corta, entonces le da la mayor prioridad.

Si realmente es una tarea corta se va a ejecutar rápidamente y va a terminar, si no lo es se moverá lentamente hacia abajo en las colas de prioridad haciendo que se parezca más a un proceso BATCH.

Ejemplo 1: Qué pasa con la entrada y salida.



Como se considera en la regla 4 si la tarea renuncia al uso del procesador antes de un time slice se mantiene en el mismo nivel de prioridad. El objetivo de esta regla es simple: si una tarea es interactiva, por ejemplo, entrada de datos por teclado o movimiento del mouse, esta no va a requerir uso de CPU antes de que su time slice se complete en ese caso no será penalizada y mantendrá su mismo nivel de prioridad.

PROBLEMA con este Approach de MLFQ

- **Starvation (inhanición):** Si hay demasiadas tareas interactivas en el sistema se van a combinar para consumir todo el tiempo del CPU y las tareas de larga duración nunca se van a ejecutar.
- Un usuario inteligente podría “hackear” el scheduler; reescribir sus programas para obtener más tiempo de CPU. Por ejemplo: Antes de que termine el time slice se realiza una operación de entrada y salida entonces se va a relegar el uso de CPU haciendo esto se va a mantener la tarea en la misma cola de prioridad. Entonces la tarea puede monopolizar todo el tiempo de CPU.

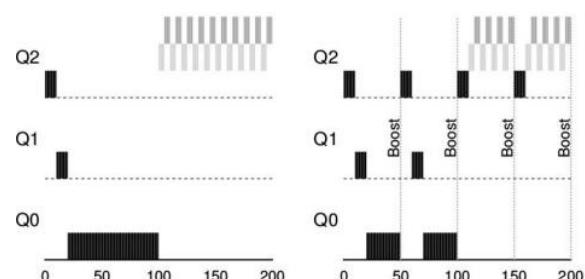
Segundo Approach

Para cambiar el problema del starvation y permitir que las tareas con larga utilización de CPU puedan progresar lo que se hace es aplicar una idea simple, se mejora la prioridad de todas las tareas en el sistema. Se agrega una nueva regla:

Regla 5: Despues de cierto periodo de tiempo S, se mueven las tareas a la cola con más prioridad.

Haciendo esto se matan 2 pájaros de 1 tiro:

- Se garantiza que los procesos no se van a starve: Al ubicarse en la cola tope con las otras tareas de alta prioridad estos se van a



ejecutar utilizando round-robin y por ende en algún momento recibirá atención.

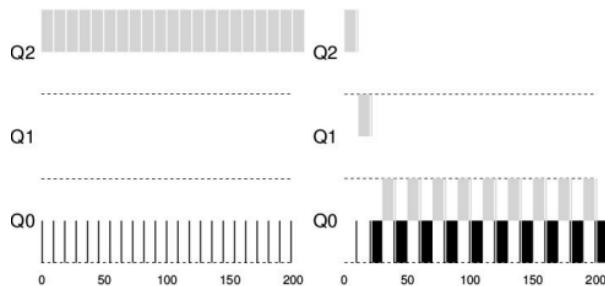
- Si un proceso que consume CPU se transforma en interactivo el planificador lo tratará como tal una vez que haya recibido el boost de prioridad.

¿Cuánto debería ser el valor del tiempo S? => si el valor es demasiado alto, los procesos que requieren mucha ejecución van a caer en starvation; si se setea a S con valores muy pequeños las tareas interactivas no van a poder compartir adecuadamente la CPU.

Se debe solucionar otro problema: COMO PREVENIR QUE VENTAJEEN (gaming) AL PLANIFICADOR. La solución es llevar una mejor contabilidad del tiempo de uso de la CPU en todos los niveles del MLFQ. El planificador debe hacer un seguimiento desde que un proceso ha sido asignado a una cola hasta que es trasladado a una cola de distinta prioridad. Cada proceso tiene un contador, que trackea cuando time slice van consumiendo en total, independientemente de las interrupciones.

Ya sea si usa su time slice de una o en pequeños trocitos, esto no importa por ende se reescriben las reglas 4a y 4b en una única regla:

Regla 4: Una vez que una tarea consume su time-slice en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU) su prioridad se reduce; baja un nivel en la cola de prioridad.



Recapitulando

- ★ **REGLA 1:** si la prioridad(A) > prioridad(B), (A) se ejecuta y (B) no.
- ★ **REGLA 2:** si la prioridad(A) = prioridad(B), (A) y (B) se ejecutan en Round-Robin.
- ★ **REGLA 3:** Cuando una tarea llega al sistema se la coloca en la más alta prioridad.
- ★ **REGLA 4:** Una vez que una tarea consume su time-slice en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU) su prioridad se reduce; baja un nivel en la cola de prioridad.
- ★ **REGLA 5:** Después de un cierto periodo de tiempo S, se mueven todas las tareas a la cola con más prioridad.

Ejercicio de parcial

Se tiene un scheduler MLFQ, de 3 niveles de prioridad y un timeslice de 20 ms. Asumir que no usa boost (regla 5). Existe un programa A en el sistema que hace un uso constante del CPU (nada de I/O) y que inició hace mucho tiempo.

Se quiere ejecutar un programa B que realiza el siguiente ciclo de ejecución:

1. Utiliza el CPU para un cálculo que lleva 15 ms.
2. Hace una llamada de I/O que tarda 1 ms.
3. Repite en loop 1 y 2 hasta completar un uso total del CPU de 65ms

Ejemplo de ejecución (sin el programa A):



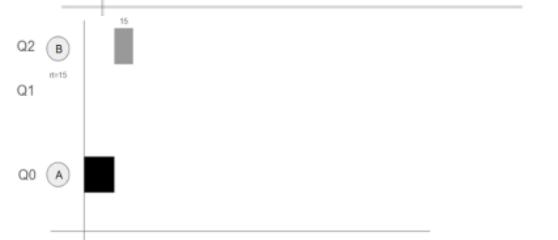
¿Cuanto va a tardar el programa en terminar (turnaround)?

- Asumir que en ciclos de round robin siempre empieza ejecutándose A.
- Asumir que cuando se desbloquea un programa de mayor prioridad, interrumpe inmediatamente a uno de menor prioridad.

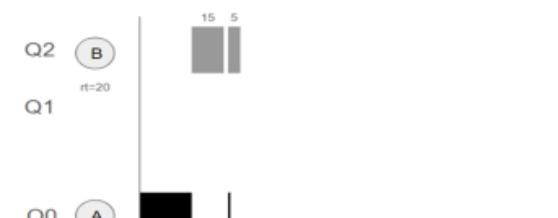
- Como se tiene un programa A que inicio hace mucho tiempo, sin I/O, y no hay boost (regla 5) entonces estará en el nivel mas bajo de prioridad.



- Al iniciar el programa B, se le da la prioridad mas alta por regla 3, y ni bien llega se ejecuta hasta su I/O de 1 ms (por enunciado)



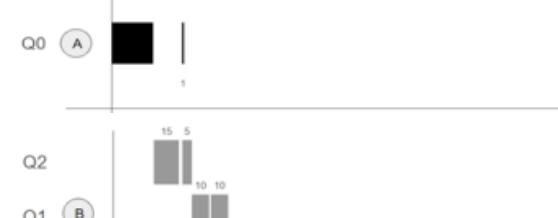
- En el momento del I/O se ejecuta el programa A y luego se vuelve al B, terminando sus 20 ms de timeslice (por enunciado)



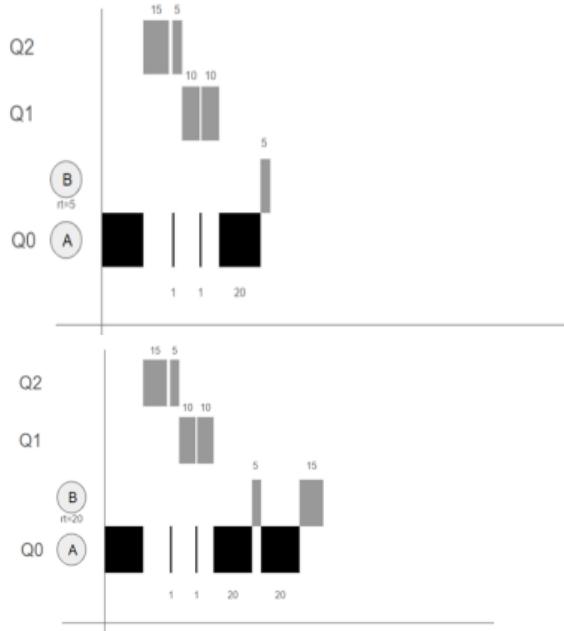
- Al terminar el timeslice de B , se le baja de prioridad/cola, ejecutándose porque sigue teniendo una prioridad mas alta que A. Llega hasta el I/O.



- En el I/O se ejecuta el A, luego se vuelve al B y se termina su timeslice.



- Al terminar el timeslice, se le baja nuevamente de prioridad. Ahora ambos tienen misma prioridad, por lo que se los ejecuta alternadamente (Round Robin) por regla 2. Se empieza por A hasta agotar su timeslice, y luego se ejecuta B hasta su I/O.
- Otra vez se cambia a A y se agota su timeslice, se vuelve a B hasta otro I/O.



Esto se repetirá, otros 20ms de A y luego se terminarán los 5ms faltantes de B.
Al terminarse el programa B, se debe contar cuantos ms suman entre la ejecución de ambos programas ya que para calcular turnaround se calcula con el tiempo de llegada y finalización.
65ms (de la ejecución de B) + 62ms (de la ejecución de A mientras que se cambiaba de programa) = 127ms
Respuesta: El Turnaround es igual a 127ms.

Simulador de Scheduler

Es un loop infinite, con un select next, que selecciona el siguiente proceso a ejecutar. Si hay un proceso siguiente (osea el candidato no es NULL), pasamos su estado a ejecutable y el switch hace un cambio de contexto de ese proceso. Si no hay nada ejecutable, hace idle(); se usa para manejar el estado de inactividad o espera de un programa.
Acá simplemente hace una simulación de que no hubo nada RUNNABLE porque todos los procesos estaban BLOCKED. El switch además devuelve cuánto tiempo se ejecutó ese proceso.

```
C sched.c  X  C sched.h
C sched.c > scheduler()
1 #include "sched.h"
2
3 extern struct proc proc[];
4
5 struct proc* select_next(){
6     return NULL;
7 }
8
9 void scheduler(){
10    while (!done()){
11        struct proc *candidate = select_next();
12
13        if (candidate != NULL) {
14            candidate->status = RUNNING;
15            candidate->runtime += swtch(candidate);
16        } else {
17            idle();
18        }
19    }
20
21 }
22 }
```

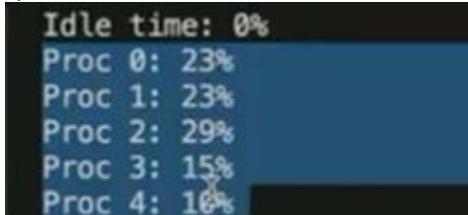
```

C sched.c      C sched.h x
C sched.h > SCHED_H
1 #ifndef SCHED_H
2 #define SCHED_H
3
4 #define NUMPROC 5
5 #define TIMESLICE 20
6
7 enum Status { RUNNING, RUNNABLE, BLOCKED };
8
9 struct proc {
10     int pid;
11     enum Status status;
12
13     int runtime;
14 };
15
16 void scheduler();
17
18#endif

```

EJ1: En esta implementación se elige un proceso al azar, y si este está en estado runnable, se ejecuta.

EJ2: Busquemos un proceso entre los que esté en RUNNABLE. A cada proceso runnable se le asigna un número aleatorio. El número más grande gana y se ejecuta ese.



Aca definimos el struct del proceso. Tenemos seteado en max 5 procesos con un timeslice de 20.

Un ejercicio de parcial podría ser: codee el select_next() para que sea un round robin.

```

struct proc* select_next(){
    int n = rand() % NUMPROC;

    if (proc[n].status == RUNNABLE) {
        return &proc[n];
    } else {
        return NULL;
    }
}

extern struct proc proc[];

struct proc* select_next(){
    int next_rnd = -1;
    struct proc* next = NULL;

    for (int i = 0; i < NUMPROC; i++) {
        if (proc[i].status == RUNNABLE) {
            int random = rand();
            if (random > next_rnd) {
                next_rnd = random;
                next = &proc[i];
            }
        }
    }

    return next;
}

```

Sin embargo, este algoritmo no hace que los procesos se ejecuten uniformemente. Un algoritmo más equitativo hace que cada proceso ocupe 20% del tiempo del CPU. Por ejemplo, el siguiente:

```

struct proc* select_next(){
    int next_runtime = INT_MAX;
    struct proc* next = NULL;

    for (int i = 0; i < NUMPROC; i++) {
        if (proc[i].status == RUNNABLE) {
            if (proc[i].runtime <= next_runtime) {
                next_runtime = proc[i].runtime;
                next = &proc[i];
            }
        }
    }

    return next;
}

```

Caso de estudio: Linux

El planificador de tareas de Linux desde su primera versión hasta la 2.4 fue sencillo.

Planificador de tareas

- Round-robin: Cada proceso se le asigna un intervalo de tiempo fijo (quantum de tiempo) en el cual se puede ejecutar. Cuando este tiempo se agota, se deja en espera y pasa al siguiente proceso en la cola.
- Multinivel de Feedback Queue: Se utilizan múltiples colas, organizando los procesos basándose en prioridades y comportamiento. Un proceso puede cambiar de cola si su prioridad o comportamiento cambia.

Prioridades y Preemption

Aunque el núcleo Linux de la serie 2.4 no era completamente preemptivo en su manejo de procesos en espacio de usuario, sí tenía soporte básico para la preemption en el manejo de interrupciones y algunas operaciones críticas del kernel.

- ciertas tareas críticas podían interrumpir otras menos críticas para mejorar la capacidad de respuesta del sistema.

Sin embargo, en el espacio de usuario, un proceso en ejecución no sería necesariamente interrumpido hasta que finalizara su quantum de tiempo, a menos que esperara por I/O u otra interrupción.

Principios básicos del CFS (Completely Fair Scheduler)

Fairness (equidad)

El CFS busca ofrecer a cada proceso una porción justa del CPU, basándose en el concepto de equidad. No se asignan cuotas fijas de tiempo (time slices), sino que se intenta que cada proceso reciba una porción del tiempo de CPU (proporcional a su peso de prioridad). Eg. Si tengo 5 procesos, en 100 segundos una ejecución “justa” asigna el CPU durante 20 ms a cada proceso.

- Como funcionaría en teoría: Si hay n procesos cada proceso se ejecutaría en el procesador todo el tiempo a $1/n$ de la potencia de este.
- Como funciona en la práctica: En una ventana de tiempo, el procesador ejecuta cada tarea durante $1/n$ del tiempo.
- ★ **EL SCHEDULER RESUELVE UN PROBLEMA DE OPTIMIZACIÓN: REDUCIR LA VARIANZA DEL VRUNTIME DE TODOS LOS PROCESOS. ES DECIR, TRATA DE MANTENER EL RUNTIME DE CADA PROCESO IGUAL.**

Algoritmo de selección:

Cada vez que tenga que elegir un proceso, el scheduler elige siempre el que esté más atrás en el vruntime consumido.

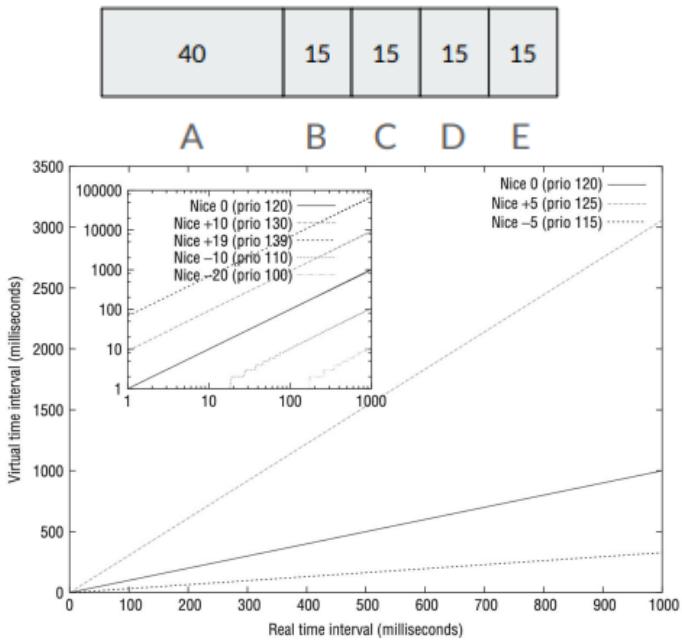


Proceso	Runtime
A	32
B	31
C	26
D	35
E	37

¿Cómo modelamos el concepto de prioridad? La prioridad se corresponde con pesos de los procesos. En el ejemplo de abajo, el proceso A pesa más que los demás.

El reloj de los procesos con más prioridad va a ir más lento. Entonces como consumen menos vruntime van a ser seleccionados más frecuentemente por el mismo algoritmo. En consecuencia, terminan teniendo un share mayor del CPU.

Virtual Runtime (vruntime): Cada proceso tiene asociado un contador llamado vruntime, que representa la cantidad de tiempo que el proceso ha estado ejecutándose en el CPU. El vruntime se incrementa con cada tick del reloj mientras el proceso está en ejecución.



LOS PROCESOS QUE CONSUMEN VRUNTIME MÁS LENTAMENTE (CON RESPECTO AL TIEMPO REAL) SON LOS QUE TIENEN MAYOR PRIORIDAD PARA SER EJECUTADOS.

El vruntime se calcula en función del tiempo que un proceso ha ejecutado en el CPU, ajustado por su prioridad.

$$\text{vruntime}+ = \frac{\text{delta_exec}}{\text{weight}}$$

- **delta_exec** es la cantidad de tiempo que el proceso ha estado ejecutando desde la última actualización.
- **weight** es el peso asociado a la prioridad del proceso.

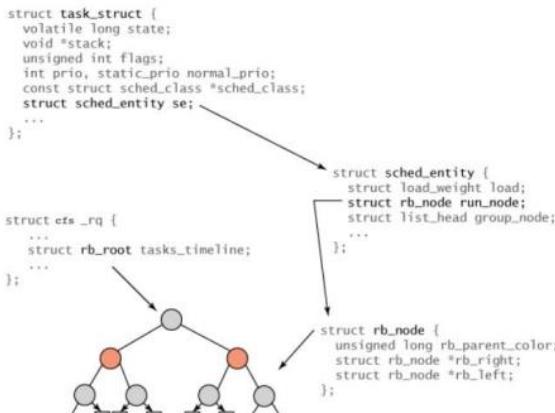
Runqueue

- Estructura de datos utilizada por el planificador (scheduler) para gestionar y hacer un seguimiento de todos los procesos (tareas) que están listos para ejecutarse en un núcleo de CPU.
- Cada núcleo de CPU tiene su propia runqueue, que contiene las tareas que están listas para ejecutar pero que no están ejecutándose en ese momento.
- Los procesos ejecutables se colocan en la runqueue, y se extrae de la misma el que tiene menor vruntime. Es esencialmente una priority-queue, donde la prioridad es el menor vruntime.

Como hacemos para elegir el proceso con menor vruntime de una runqueue, en un tiempo menor que O(n)?

Red-Black Tree: Tipo de árbol binario de búsqueda auto-balanceado que se utiliza en informática para mantener datos ordenados y permitir inserciones, eliminaciones y búsquedas eficientes.

- Acceso O(1) al nodo más a la izquierda (tiempo virtual más bajo).
- Inserción O(log n).
- Eliminación O(log n).
- Autoequilibrado.



Funcionamiento Detallado

- Selección del Proceso:** El scheduler siempre intenta ejecutar el proceso con el menor vruntime. Esto asegura que todos los procesos tengan la oportunidad de ejecutarse de manera justa, proporcionando un tiempo de ejecución proporcional a su peso.
- Uso de Red-Black Trees:** Para organizar eficientemente los procesos por su vruntime, dado que permite inserciones, eliminaciones y búsquedas en tiempo logarítmico, lo que es crucial para mantener el rendimiento del scheduler con un gran número de procesos.
- Sleeping and Waking Up:** Cuando un proceso se bloquea, por ejemplo, esperando por I/O, se elimina del árbol rojo-negro. Cuando se despierta, se vuelve a insertar en el árbol con su vruntime ajustado si es necesario, de manera que no sea penalizado por el tiempo que estuvo bloqueado.
 - Cuando un proceso se está ejecutando, su vruntime aumentará constantemente, por lo que finalmente se moverá hacia la derecha en el árbol rojo-negro.
 - Si un proceso duerme, su vruntime permanecerá sin cambios. Debido a que el min_vruntime de la cola (Arbol) aumenta mientras tanto, el proceso que estaba dormido se colocará más a la izquierda al despertarse, porque su clave se hizo más pequeña.
- Balance de Carga:** El CFS también se encarga de equilibrar la carga de trabajo entre múltiples CPUs. Esto se logra migrando procesos de una CPU a otra si se detecta que una CPU está sobrecargada mientras otra está inactiva o menos cargada.
- Soporte para Multi-threading:** CFS maneja los hilos de ejecución (threads) de manera similar a como maneja los procesos. Cada hilo es tratado como un proceso separado con su propio vruntime.

Nice

- Característica del sistema operativo Linux que se usa para ajustar la prioridad de los procesos en cuanto a su planificación para el uso del CPU.
- Permite a los usuarios manipular la prioridad de un proceso de manera que los procesos con una mayor prioridad obtengan más tiempo de CPU comparado con los que tienen una menor prioridad.
- Se puede configurar con la system call nice()
- Características:

Rango de valores: El valor nice puede variar de -20 hasta 19. Un valor nice de -20 es el más alto nivel de prioridad (menos "nice", es decir, menos amable con los demás procesos), y 19 es el nivel más bajo (más "nice", más amable). Por defecto, los procesos suelen iniciarse con un valor nice de 0.

Asignación de CPU: Un valor nice bajo (más cercano a -20) hace que el proceso sea más "egoísta", obteniendo más tiempo de CPU. Por el contrario, un valor nice alto (más cercano a 19) hace que el proceso sea más "generoso", cediendo el tiempo de CPU a otros procesos.

Modificación del valor nice: Los usuarios pueden cambiar el valor nice de un proceso para afectar su prioridad de planificación. Esto se hace generalmente a través de la línea de comandos con herramientas como nice y renice. Por ejemplo, para iniciar un proceso con un valor nice específico, se utiliza el comando nice -n [valor] [comando]. Para cambiar el valor nice de un proceso ya en ejecución, se usa renice [nuevo_valor] -p [pid].

Pesos de Prioridad (Weight): Cada proceso tiene un peso asignado basado en su prioridad de nice, que puede variar de -20 (más favorable) a 19 (menos favorable). Estos valores de nice se mapean a un peso específico utilizando una tabla predefinida en el kernel. Los pesos ayudan a determinar cuán rápido un proceso acumula vruntime en comparación con otros procesos. 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,
};
```

$$\text{vruntime}+ = \frac{\text{delta_exec}}{\text{weight}}$$


Timeslice dinámico

YA ELEGÍ UN PROCESO. ¿CUÁNTO TIEMPO VA A EJECUTAR?

- Latencia objetivo (sched_latency): Este es un valor que define el período de tiempo durante el cual el scheduler intenta que todos los procesos ejecutables se ejecuten al menos una vez. Por defecto, este período es de alrededor de 48 ms en sistemas con un número considerable de tareas ejecutándose (el default es 20ms). Este valor se ajusta según el número de tareas activas para mantener el rendimiento del scheduler.
- Para calcular por cuánto tiempo se va a correr un proceso al ser elegido se toma proporcional del sched latency, la prioridad del proceso dividida por el total de todas las prioridades de todos los procesos runnables.

Ej1: sched_latency es de 20 ms: Proceso A: nice-0, Proceso B: nice-5

nice-0 = 1024

Nice-5 = 335

- timeslice(A) = 20 ms * 1024/(1024 + 335) = 15 ms
- timeslice(B) = 20 ms * 335/(1024 + 335) = 5 ms

Ej2: sched_latency es de 20 ms: Proceso A: nice-10, Proceso B: nice-15

nice-10 = 110

nice-15 = 36

$$\rightarrow \text{timeslice}(A) = 20 \text{ ms} * 110 / (110 + 36) = 15 \text{ ms}$$

$$\rightarrow \text{timeslice}(B) = 20 \text{ ms} * 36 / (110 + 36) = 5 \text{ ms}$$

EL VALOR DE NICE TIENE SENTIDO RELATIVO. NO ABSOLUTO!!!

Constantes y Factores en el CFS

MIN_GRANULARITY: Mínimo tiempo de ejecución garantizado

Este valor define el mínimo tiempo de ejecución que un proceso debería tener antes de ser desalojado del CPU., para evitar que los procesos sean interrumpidos demasiado rápidamente debido a que el timeslice calculado fue demasiado pequeño, lo cual podría llevar a un aumento en el overhead por cambios de contexto.

- El sched_latency trabaja en conjunto con otro parámetro llamado min_granularity.
- min_granularity suele ser una fracción del sched_latency, típicamente alrededor del 10-20%.

Ejemplo de Ajuste de sched_latency: Supongamos que un sistema tiene 100 tareas activas y un sched_latency estándar de 48 ms. Si el número de tareas supera el valor ideal que el scheduler puede manejar efectivamente dentro de ese período, digamos 25 tareas, el sistema puede decidir duplicar el sched_latency a 96 ms para permitir que todas las tareas sean planificadas adecuadamente sin aumentar demasiado la frecuencia de cambio de contexto.

NORMALIZACION DE TIEMPOS

Para garantizar que los cálculos se mantengan dentro de límites manejables y evitar el desbordamiento de enteros, los tiempos de ejecución pueden ser normalizados periódicamente. Esto implica ajustar los vruntime de todos los procesos, reduciendo todos los valores en función del proceso con el menor vruntime. Esto evita que haya overflows de las variables.

Esquema del Funcionamiento del CFS

- **Inicialización:** Cada proceso recibe un valor vruntime inicial, que es cero cuando el proceso es creado.
- **Cálculo de Pesos:** Basado en el valor nice de cada proceso, se asigna un peso específico. La tabla de mapeo de nice a pesos determina cuán rápido se acumula vruntime para un proceso dado.
- **Gestión de vruntime:** El vruntime de un proceso se incrementa en función del tiempo que el proceso pasa ejecutándose en el CPU. La tasa de incremento se ajusta por el peso del proceso: procesos con mayor peso (menos nice) acumulan vruntime más lentamente.
- **Uso de Red-Black Tree:** Todos los procesos listos para ejecutarse son almacenados en un árbol rojo-negro, organizados por su vruntime. El proceso con el menor vruntime se encuentra siempre el más a la izquierda de la raíz del árbol y es seleccionado para ejecutarse.
- **Selección de Procesos:** El scheduler selecciona el proceso con el menor vruntime del árbol para ejecución. Este proceso se ejecuta hasta que su vruntime ya no es el mínimo, o hasta que se bloquea o termina su quantum de ejecución.

- **Preemptividad y Voluntariedad:** Si un proceso se bloquea esperando I/O, se retira del árbol temporalmente. Cuando un proceso despierta o es creado, se calcula su vruntime ajustado y se inserta en el árbol.
- **Ajuste Dinámico:** `sched_latency` y `min_granularity` se utilizan para ajustar la cantidad de tiempo que cada proceso puede ejecutarse. En sistemas con muchos procesos, estos valores se ajustan para balancear equidad y eficiencia.
- **Balanceo de Carga entre CPUs:** El CFS también maneja la distribución de procesos entre múltiples CPUs. Los procesos pueden ser migrados entre CPUs para optimizar la carga y minimizar la latencia.

EJEMPLO

Detalles de los Procesos y Valores Nice:

- Proceso A: nice = 0
- Proceso B: nice = -5
- Proceso C: nice = 10
- Proceso D: nice = 0
- Proceso E: nice = -10

Cálculo de pesos:

- Valor nice de -10 (Proceso E) corresponde a un peso de 1024.
- Valor nice de -5 (Proceso B) corresponde a un peso de 512.
- Valor nice de 0 (Proceso A y D) corresponde a un peso de 256.
- Valor nice de 10 (Proceso C) corresponde a un peso de 128.

Inicio: Todos los procesos empiezan con vruntime = 0.

- Ciclo 1: Todos los procesos tienen el mismo vruntime inicial, pero el Proceso E tiene el peso más alto y, por lo tanto, es el menos penalizado en la acumulación de vruntime. El Proceso E se ejecuta, digamos, por una unidad de tiempo. Su vruntime incrementa menos debido a su alto peso. Nuevo vruntime de E = $0 + (1 \text{ unidad de tiempo} / 1024) = 0.0009765625$.
- Ciclo 2: El Proceso B tiene el siguiente peso más alto. Se ejecuta por una unidad de tiempo. Nuevo vruntime de B = $0 + (1 \text{ unidad de tiempo} / 512) = 0.001953125$.
- Ciclo 3: Proceso A o D (ambos tienen nice = 0 y el mismo peso). Supongamos que se selecciona el Proceso A. Se ejecuta por una unidad de tiempo. Nuevo vruntime de A = $0 + (1 \text{ unidad de tiempo} / 256) = 0.00390625$
- Ciclo 4: Se ejecuta el Proceso D bajo las mismas condiciones que A. Nuevo vruntime de D = $0 + (1 \text{ unidad de tiempo} / 256) = 0.00390625$.
- Ciclo 5: El Proceso C tiene el peso más bajo y por lo tanto, su vruntime crece más rápidamente. Se ejecuta por una unidad de tiempo. Nuevo vruntime de C = $0 + (1 \text{ unidad de tiempo} / 128) = 0.0078125$.
- Revisión y Selección Continua: Al final de estos ciclos, el scheduler revisa todos los vruntime. El Proceso E, que ha acumulado el menor vruntime (0.0009765625), sería normalmente seleccionado nuevamente, ya que su vruntime sigue siendo el más bajo.

Group Scheduling

Los procesos se colocan en diferentes grupos, y el planificador es primero justo entre estos grupos y luego justo entre todos los procesos dentro del grupo.

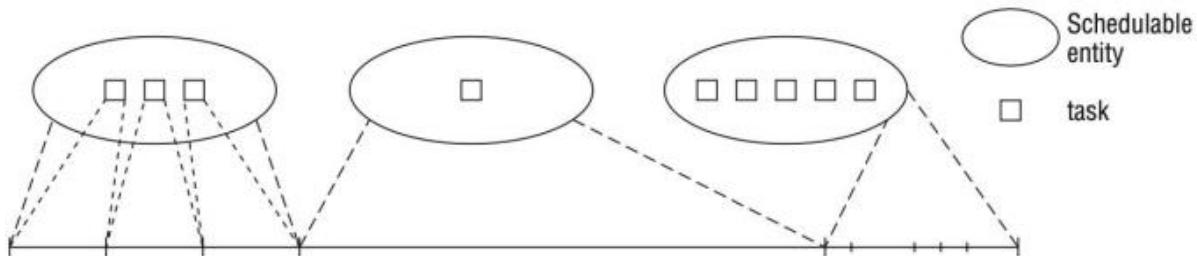
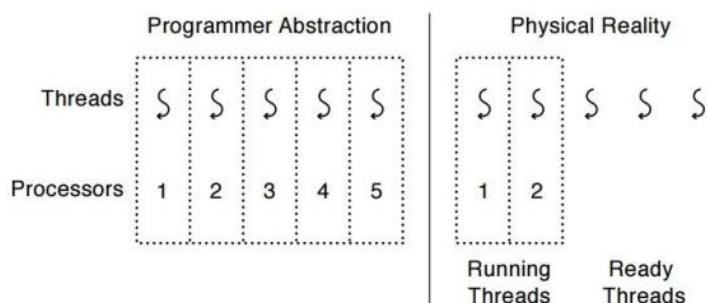


Figure 2-29: Overview of fair group scheduling: The available CPU time is first distributed fairly among the scheduling groups, and then between the processes in each group.

Esto permite, por ejemplo, otorgar partes iguales del tiempo de CPU disponible a cada usuario. Una vez que el planificador ha decidido cuánto tiempo obtiene cada usuario, el intervalo determinado se distribuye de manera justa entre los procesos del usuario. Esto implica naturalmente que, cuanto más proceso ejecute un usuario, menor será el share de CPU que obtendrá cada proceso. Sin embargo, la cantidad de tiempo total para el usuario NO se ve influenciada por el número de procesos.

THREADS

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:



La abstracción

Un thread es una SECUENCIA de EJECUCION ATOMICA que representa una TAREA PLANIFICABLE de EJECUCION.

- Secuencia de ejecución atómica: Cada thread ejecuta una secuencia de instrucciones como lo hace un bloque de código en el modelo de programación secuencial. El thread empieza y termina.
- Tarea planificable de ejecución: El sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando él deseé.

Threads vs. Procesos

Proceso: un programa en ejecución con derechos restringidos.

Thread: una secuencia independiente de instrucciones ejecutándose dentro de un programa.

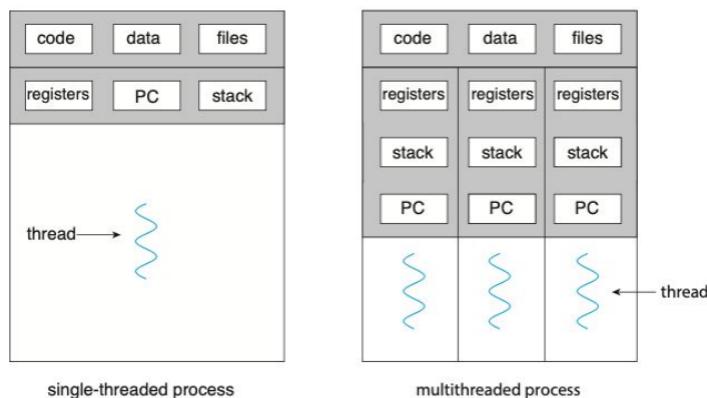
Esta abstracción se caracteriza por:

- Thread id

- un conjunto los valores de registros => Tiene un CONTEXTO
- stack propio
- una política y prioridad de ejecución
- un propio errno
- datos específicos del thread

Podemos tener:

1. **One thread per process:** un proceso con una única secuencia de instrucciones ejecutándose de inicio a fin. Para los que vieron Pascal o C sería el equivalente a un bloque de instrucciones delimitado por begin-end o { }. Lo que todos los programadores de modelo secuencial conocemos.
2. **Many thread per process:** un programa es visto como threads ejecutándose dentro de un proceso con derechos restringidos. En dado un determinado instante, algunos threads pueden estar corriendo y otros estar suspendidos. Cuando se detecta por ejemplo una operación de I/O por alguna interrupción, el kernel desaloja (preempt) a algunos de los threads que están corriendo, atiende la interrupción, y al terminar de manejar la interrupción vuelve a correr el thread nuevamente.



Thread Scheduler

¿CÓMO HACE EL S.O. PARA CREAR LA ILUSIÓN DE MUCHOS THREADS CON UN NÚMERO FIJO DE PROCESADORES?

Obviamente es necesario un **planificador de thread o threads 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**

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.

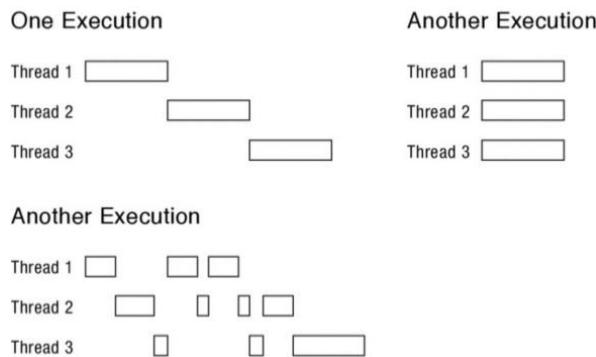
Por ejemplo:

```
...  
...  
x = x + 1;  
y = x + y;  
z = x + 5y;  
...  
...
```

Entonces en base a lo antedicho, se pueden encontrar los siguientes escenarios de ejecución:

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended. Other thread(s) run. Thread is resumed. Thread is suspended. Other thread(s) run. Thread is resumed.
. y = y + x; z = x + 5y;
.	.	z = x + 5y;	
.	.		

Y los siguientes interleaves o entrelazado pueden suceder, con estos distintos threads:



En la actualidad hay dos formas de que los threads se relacionen entre sí:

1. Multi-threading Cooperativo: no hay interrupción a menos que se solicite. Problema: Cheating
2. Multi-threading Preemptivo: Es el más usado en la actualidad. Consiste en que un threads en estado de *running* puede ser movido en cualquier momento.

El API de Threads

Para la programación utilizando threads se utilizará la biblioteca pthread donde la p es de POSIX Threads.

1. Creación de un Thread

```
#include<pthread.h>
int pthread_create(pthread_t * thread, const pthread_attr_t * attr,
                  void * (start_routine) (void *), void * arg)
```

- **thread:** Es un puntero a la estructura de tipo pthread_t, que se utiliza para interactuar con el threads.
- **attr:** Se utiliza para especificar los ciertos atributos que el thread debería tener, por ejemplo, el tamaño del stack, o la prioridad de scheduling del thread. En la mayoría de los casos es NULL.
- **start_routine:** Sea tal vez el argumento más complejo, pero no es más que un puntero a una función, en este caso que devuelve void.
- **arg:** Es un puntero a void que debe apuntar a los argumentos de la función.

Devuelve 0 si se ha creado el thread con éxito, si hubo error devuelve otro valor. Ejemplo básico de creación de un thread:

```
#include <pthread.h>
typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
}
```

2. Terminación de un Thread

Muchas veces es necesario esperar a que un determinado thread finalice su ejecución, para ello se utiliza la función pthread_join(), que toma dos argumentos:

1. **thread** es el thread por el que hay que esperar y es de tipo pthread_t.
2. **value_ptr** es el puntero al valor esperado de retorno.

Ejemplo de creación:

```

#include <pthread.h>
#include <stdio.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1,p2;
    int rc;

    printf("inicia main()\n");
    rc=pthread_create(&p1, NULL, mythread, "A");
    rc=pthread_create(&p2, NULL, mythread, "B");

    rc(pthread_join(p1,NULL));
    rc(pthread_join(p2,NULL));

    printf("termina main()\n");
    return 0;
}

```

Tabla de equivalencia entre procesos y threads

Process primitive	Thread primitive	Descripción
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 control
getpid	pthread_self	obtiene el id de un determinado flujo de control
abort	pthread_cancel	terminación anormal de un flujo de control

Estructura y Ciclo de Vida de un Thread

Como se ha visto, cada thread es la representación de una secuencia de ejecución de un conjunto de instrucciones. El S.O. provee la ilusión de que cada uno de estos threads se ejecutan en su propio procesador, haciendo de forma transparente que se ejecuten o paren su ejecución.

Para que la ilusión sea creíble, el sistema operativo debe guardar y cargar el estado de cada thread. Como cualquier thread puede correr en el procesador o en el kernel, también debe haber estados compartidos, que no deberían cambiar entre los modos. Para poder entender la abstracción hay que comprender que existen dos estados: El estado per thread y el estado compartido entre varios threads.

El Estado Per-thread y Threads Control Block (TCB)

Cada thread debe tener una estructura que represente su estado. Esta estructura se denomina **Thread Control Block (TCB)**, se crea una entrada por cada thread. La TCB almacena el estado per-thread de un thread: El estado del Cómputo que debe ser realizado por el thread.

Para poder crear múltiples threads y pararlos y arrancarlos, el S.O. debe poder almacenar en la TCB el estado actual del bloque de ejecución:

- El puntero al stack del thread
- Una copia de sus registros en el procesador.

Metadata del thread

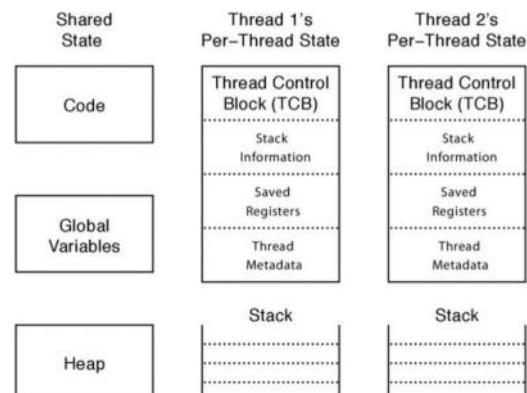
Por cada thread se debe guardar determinada información sobre el mismo:

- ID
- Prioridad de scheduling
- Status

Shared State o Estado Compartido

De forma contraria al per-thread state se debe guardar cierta información que es compartida por varios Threads:

- El Código
- Variables Globales
- Variables del Heap



Memoria en un proceso multithread

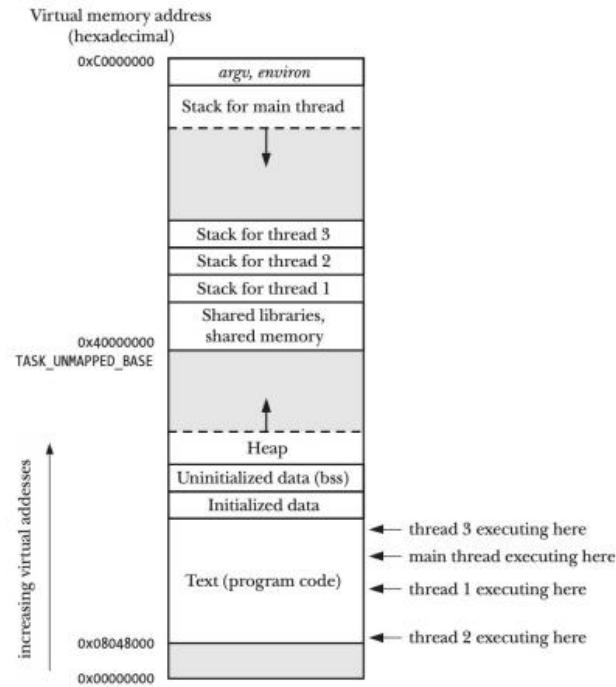
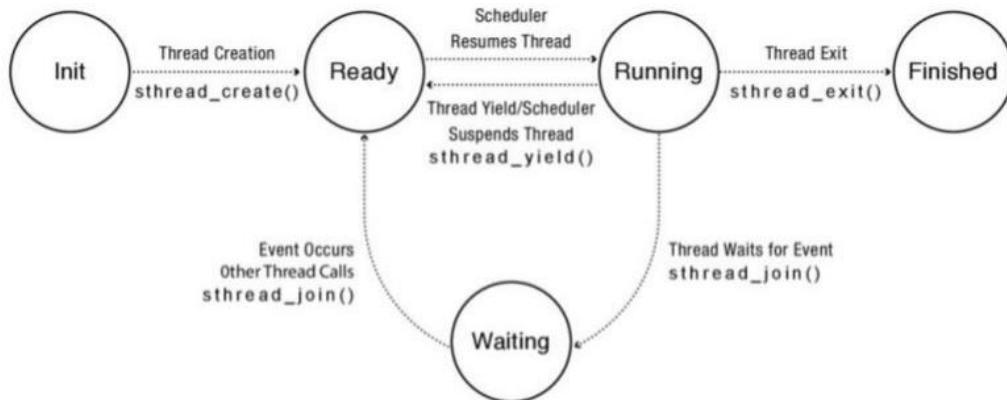


Figure 29-1: Four threads executing in a process (Linux/x86-32)

Estados de un Thread



- ◊ **Init:** Un thread se encuentra en estado INIT mientras se está inicializando el estado per-thread y se está reservando el espacio de memoria necesario para estas estructuras. Una vez que esto se ha realizado el estado del thread se setea en READY. Además, se lo pone en una lista llamada ready list en la cual están esperando todos los thread listos para ser ejecutados en el procesador.
- ◊ **Ready:** Un thread en este estado está listo para ser ejecutado, pero no está siendo ejecutado en ese instante. La TCB está en la ready list y los valores de los registros está en la TCB. En cualquier momento el thread scheduler puede transicionar al estado RUNNING.

- ◊ **Running:** Un thread en este estado está siendo ejecutado en este mismo instante por el procesador. En este mismo instante los valores de los registros están en el procesador. En este estado un RUNNING THREAD puede pasar a READY de dos formas:
 - El scheduler puede pasar un thread de su estado RUNNING a READY mediante el desalojo o preemption del mismo mediante el guardado de los valores de los registros y cambiando el thread que se está ejecutando por el próximo de la lista.
 - Voluntariamente un thread puede solicitar abandonar la ejecución mediante la utilización de `thread_yield`, por ejemplo.
- ◊ **Waiting:** En este estado el Thread está esperando que algún determinado evento suceda. Dado que un thread en WAITING no puede pasar a RUNNING directamente, estos thread se almacenan en la lista llamada **waiting list**. Una vez que el evento ocurre el scheduler se encarga de pasar el thread del estado WAITING a RUNNING, moviendo la TCB desde el waiting list a la ready list.
- ◊ **Finished:** Un thread que se encuentra en estado FINISHED nunca más podrá volver a ser ejecutado. Existe una lista llamada ***finnished list*** en la que se encuentran las TCB de los threads que han terminado.

Diferencias Proceso/Thread

Los Threads COMPARTEN por defecto lo siguiente:

- memoria
- descriptores de archivos
- el contexto del filesystem
- el manejo de señales

Los procesos NO COMPARTEN nada de lo anterior por defecto.

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.

Creación de procesos y threads en Linux

En linux existe una system call llamada `clone()` que es utilizada por `fork()` y por `pthread_create()`

Los flags disponibles son:

CLONE_VM - share memory
 CLONE_FILES - share file descriptors
 CLONE_SIGHAND - share signal handlers
 CLONE_VFORK - allow child to signal parent on exit
 CLONE_PID - share PID (not implemented yet)
 CLONE_FS - share filesystem

Creacion de Threads (LinuxThreads)

```

clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

fork()

clone(SIGCHLD, 0);

vfork()

clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);
  
```

vfork

Las primeras implementaciones de BSD estuvieron entre aquellas en las que fork() realizaba una duplicación literal de los datos, heap y stack del proceso padre. Esto es ineficiente, especialmente si el fork() es seguido por un exec() inmediato.

Por esta razón, versiones posteriores de BSD introdujeron la llamada al sistema vfork(), que era mucho más eficiente que el fork() de BSD, aunque operaba con una semántica ligeramente diferente (de hecho, algo extraña).

Al igual que fork(), vfork() es utilizado por el proceso que lo llama para crear un nuevo proceso hijo. Sin embargo, vfork() está diseñado específicamente para ser usado en programas donde el hijo realiza una llamada exec() inmediata. Dos características distinguen la llamada al sistema vfork() de fork() y la hacen más eficiente:

- ✓ No se duplica ninguna página de memoria virtual ni tablas de páginas para el proceso hijo. En su lugar, el hijo comparte la memoria del padre hasta que realiza un exec() exitoso o llama a _exit() para terminar.
- ✓ La ejecución del proceso padre se suspende hasta que el hijo haya realizado un exec() o _exit()

COW: Copy on write

Inicialmente, el kernel configura las cosas de manera que las entradas de la tabla de páginas para estos segmentos se refieren a las mismas páginas de memoria física que las entradas correspondientes de la tabla de páginas del padre, y las páginas en sí mismas se marcan como de solo lectura.

Después del fork(), el kernel intercepta cualquier intento del padre o del hijo de modificar una de estas páginas y hace una copia duplicada de la página que está a punto de ser modificada. Esta nueva copia de la página se asigna al proceso que causó el fault, y la entrada correspondiente de la tabla de páginas para el hijo se ajusta de manera apropiada. A partir de este punto, el padre y el hijo pueden modificar sus copias privadas de la página sin que los cambios sean visibles para el otro proceso.

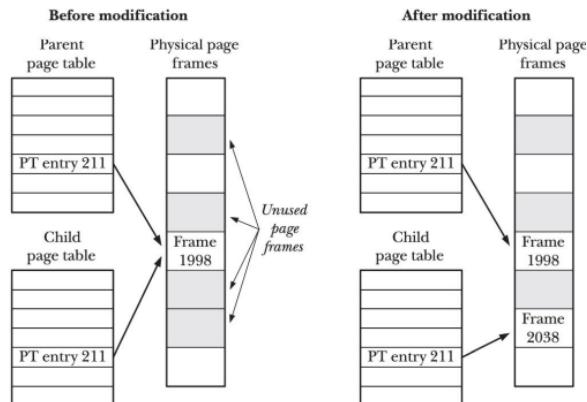


Figure 24-3: Page tables before and after modification of a shared copy-on-write page

Thread Groups: CLONE_THREAD

```
clone(CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND |  
CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID |  
CLONE_CHILD_CLEARTID | CLONE_SYSVSEM, 0)
```

Se puede comprobar si se está usando en el sistema ejecutando:

```
$ /lib/libc.so.6
```

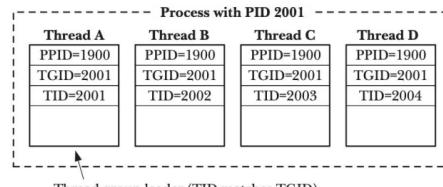


Figure 28-1: A thread group containing four threads

Viendo los Threads: ps m

```
$ ps m  
PID TTY      STAT   TIME COMMAND  
3587 pts/3    -      0:00 bash①  
      - -     Ss      0:00 -  
3592 pts/4    -      0:00 bash②  
      - -     Ss      0:00 -  
12534 tty7   - 668:30 /usr/lib/xorg/Xorg -core :0③  
      - -    Ss+ 659:55 -  
      - -    Ss+  0:00 -  
      - -    Ss+  0:00 -  
      - -    Ss+  8:35 -
```

Listing 8-1: Viewing threads with ps m

This listing shows processes along with threads. Each line with a number in the PID column (at ①, ②, and ③) represents a process, as in the normal ps output. The lines with dashes in the PID column represent the threads associated with the process. In this output, the processes at ① and ② have only one thread each, but process 12534 at ③ is multithreaded, with four threads.

MEMORIA

¿Qué es la Memoria?

- El tema FUNDAMENTAL sobre la administración de memoria se encuentra en la FORMA adecuada de REPRESENTARLA.
- La MEMORIA FISICA puede ser imaginada como un ARREGLO de DIRECCIONES de memoria una detrás de otra.
- Esta “abstracción” es manejable mientras la cantidad de memoria necesitada está en un rango “manejable” y cuál es ese rango.
- En DOS (Disk Operating System) ese rango lo daba la elección de una arquitectura determinada, la del 8086, que permite tener direcciones de memoria de hasta 220 bits, es decir 1 MegaByte de memoria.

Los Early Days

En los primeros Sistemas Operativos, en “the Early Days”, en los cuales el mismo S.O. se desplegaba en solo 64 KiloBytes de memoria, el resto de esta estaba disponible para un único proceso ejecutándose a partir de los 64Kb de memoria física.

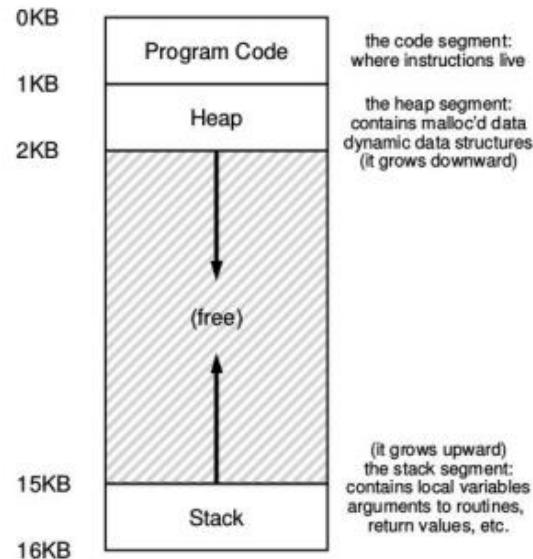
El sistema operativo era más o menos un conjunto de funciones o rutinas (específicamente una biblioteca) que se alojaba en la memoria empezando en la dirección física 0, y después existía un único

programa en ejecución (un proceso) que se encontraba en la memoria física de la computadora, esta memoria era el resto no utilizado por el sistema operativo.

El Espacio de Direcciones o Address Space

Crear un mecanismo que permita que la memoria física de una computadora sea utilizada de forma fácil y eficiente llevó paulatinamente a concebir el concepto de espacio de direcciones, 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

Metas principales:

TRANSPARENCIA

El sistema operativo debería implementar la virtualización de memoria de forma tal que sea INVISIBLE al PROGRAMA que se está ejecutando; es más, el programa debe comportarse como si él estuviera alojado en su propia área de memoria física privada. Pero detrás de escena, el sistema operativo y el hardware hacen todo el trabajo para multiplexar memoria a lo largo de los diferentes procesos y por ende implementa la ILUSION.

FLEXIBILIDAD y la EFICIENCIA

El sistema operativo debe esforzarse para hacer que la virtualización sea lo más eficiente posible en términos de tiempo (no hacer que los programas corran más lentos) y espacio (no usar demasiada memoria para las estructuras necesarias para soportar la virtualización).

PROTECCION

El sistema operativo tiene que asegurarse de proteger a los procesos unos de otros como también proteger al sistema operativo de los procesos. Cuando un proceso realiza un load, un store, o un fetch de una instrucción este no tiene que ser capaz de hacerlo o afectar de ninguna forma al contenido de la memoria del proceso o del sistema operativo.

Address Translation

Con esta técnica el hardware transforma cada acceso a memoria, transformando la virtual address que es provista desde dentro del espacio de direcciones en una physical address en la cual la información deseada se encuentra realmente almacenada.

Entonces, en todos y por cada una de las referencias a memoria un address translation es realizada por el hardware para redireccionar las referencias a la memoria de la aplicación hacia las posiciones reales en la memoria física.

Es evidente, que el hardware por sí solo no puede virtualizar la memoria. Éste, sólo provee un mecanismo de bajo nivel para poder hacerlo eficientemente. El sistema operativo tiene que involucrarse en los puntos claves para:

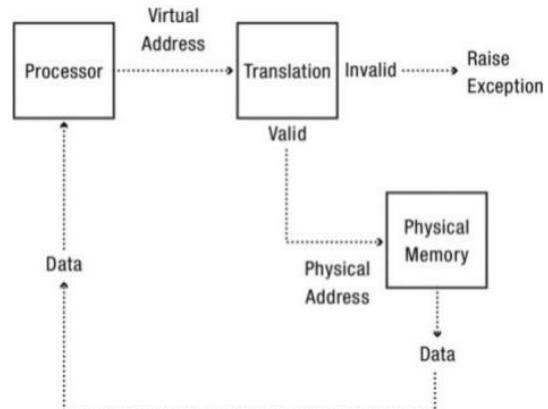
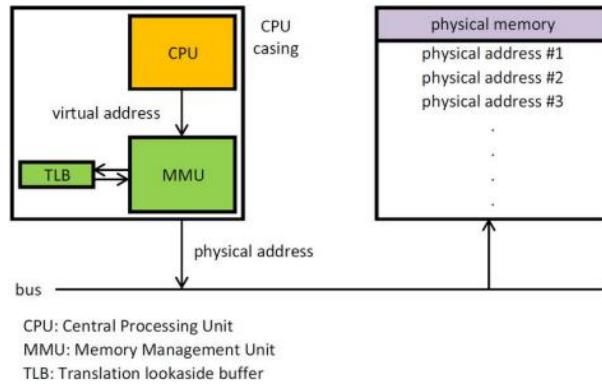
- setear al hardware de forma correcta para que esta traducción se dé lugar;
- por eso debe entonces gerenciar la memoria, manteniendo información de en qué lugar hay áreas libres y en qué lugar hay áreas en uso;
- y por último intervenir de forma criteriosa como mantener el control sobre toda la memoria usada.

El S.O. está en el medio de ese proceso y determina si el mismo se ha realizado correctamente. Lo que hace es gerenciar el manejo de la memoria:

- Manteniendo registro de qué parte está libre.
- Qué parte está en uso.
- Manteniendo el control de la forma en la cual la memoria está siendo utilizada.

Formalmente, el proceso de address translation es un mapeo entre los elementos de un espacio de direcciones virtuales de N-elementos (VAS) y un espacio de direcciones físicas de M-elementos (PAS): $MAP: VAS \Rightarrow PAS \cup \emptyset$

$$MAP(A) = \begin{cases} A' & \text{si los datos en la dir. virtual } A \text{ están presentes la dir. física } A' \text{ de PAS} \\ \emptyset & \text{si los datos de la dir. virtual } A \text{ no están presentes en la memoria física.} \end{cases}$$



Implementaciones de memoria virtual

HACIA UNA FLEXIBLE ADDRESS TRANSLATION

Para ir ganando comprensión sobre el hardware-based address translation, se verá su primera implementación, introducida en las primeras máquinas que utilizaban time-sharing hacia el fin de los años 50, es una idea simple llamada base y segmento también puede ser vista como dynamic relocation. Específicamente solo se necesitan dos registros de hardware dentro de cada cpu: Uno llamado registro base y el otro registro límite o Segmento.

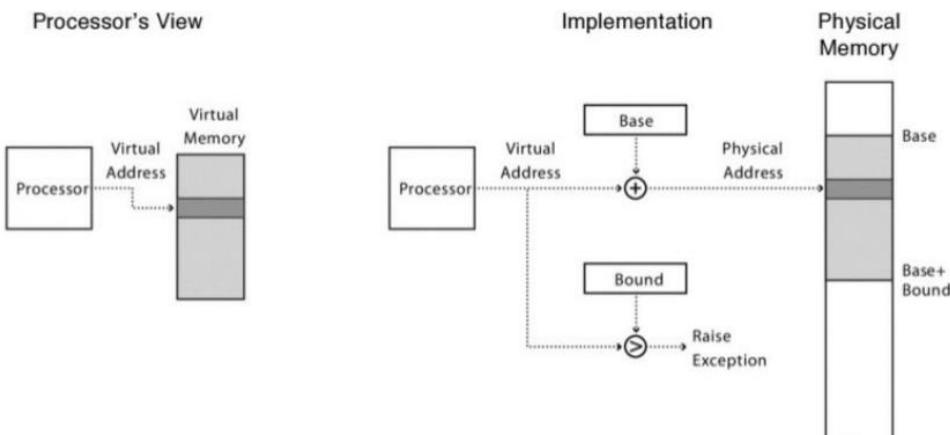
Base and Bound

Especificamente solo se necesitan DOS REGISTROS de hardware dentro de cada cpu:

- registro BASE
- 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.

PROBLEMA: tiene un solo registro base y solo un segmento.

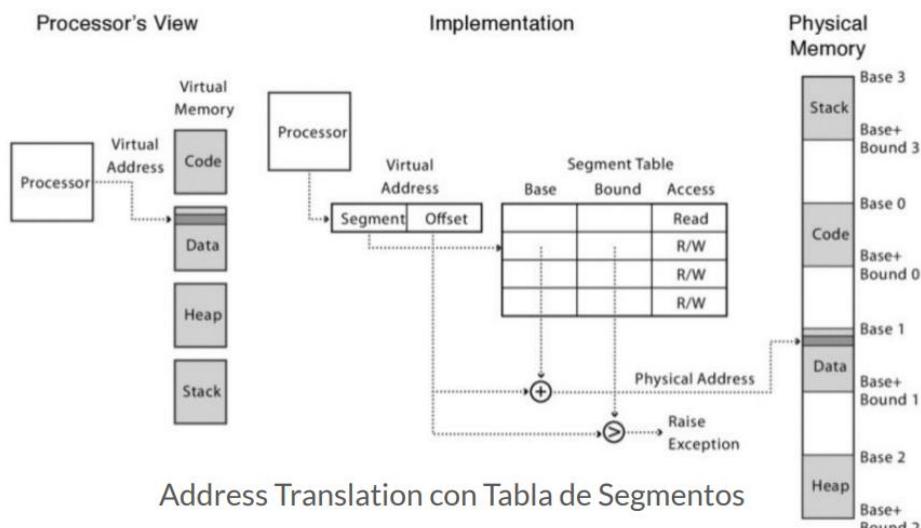


Base and Bound (segmentación)

Address Translation con Tabla de Segmentos

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*

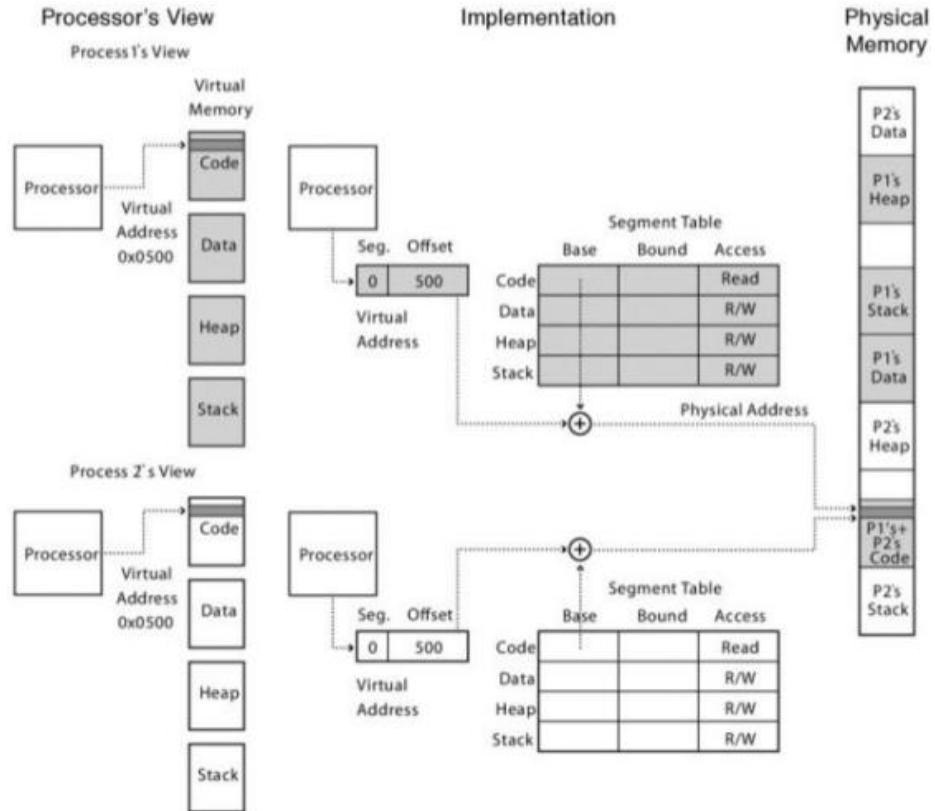
El número de segmento es el índice de la tabla para ubicar el inicio del segmento en la memoria física. El registro bound es chequeado contra la suma del registro base+offset para prevenir que el proceso lea o escriba fuera de su región de memoria.



Address Translation con Tabla de Segmentos

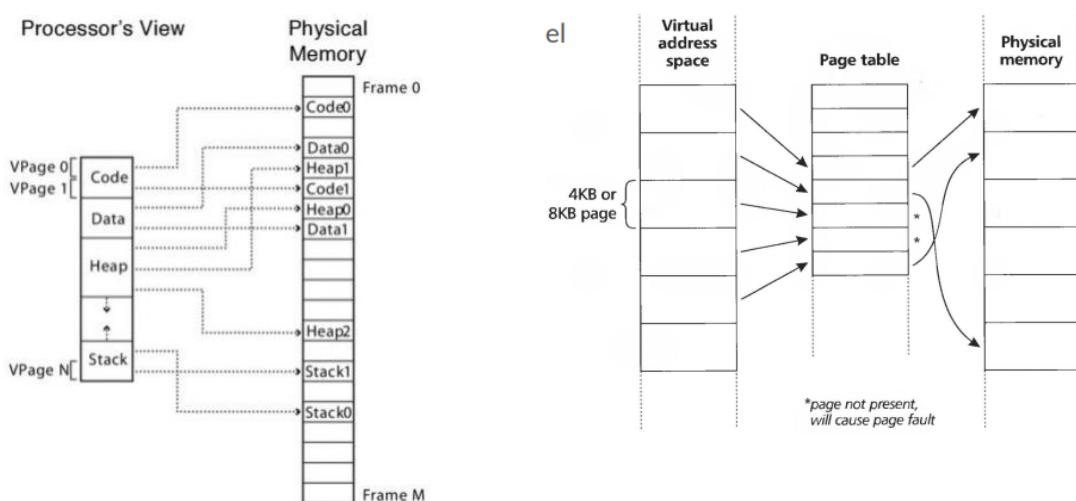
En una dirección virtual utilizando esta técnica, los bits de más alto orden son utilizados como índice en la tabla de segmentos. El resto se toma como offset y es sumado al registro base y comparado contra el registro bound.

- El número de segmentos depende de la cantidad de bits que se utilizan como índice.
- **Segmentation Fault:** error que se daba cuando, en las máquinas que implementan segmentación, se quería direccionar una posición fuera del espacio direccional. Increíblemente este error aún se utiliza en sistemas operativos que no usan segmentación.



Memoria Paginada

Con la paginación, la memoria es reservada en pedazos de tamaño fijo llamados page frames. Las páginas y los frames tienen el mismo tamaño (eg. 4 Kb)



- Una alternativa a la memoria segmentada es la memoria paginada.

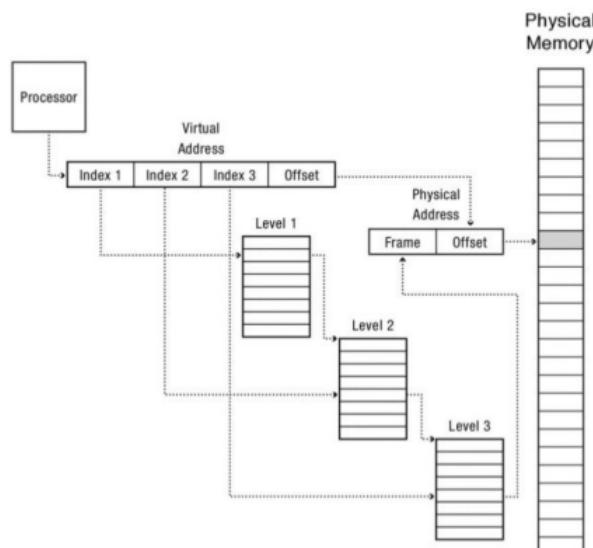
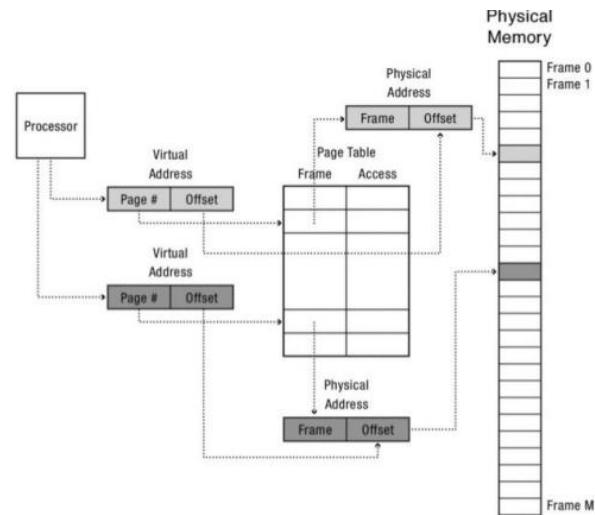
- El address translation es SIMILAR a como se trabaja con la SEGMENTACION.
- TABLA de páginas por CADA PROCESO cuyas entradas contienen punteros a las page frames.
- Teniendo en cuenta que los page frames tienen un tamaño fijo, y son potencia de 2, las entradas en la page table sólo tienen que proveer los bit superiores de la dirección de la page frame. De esta forma van a ser más compactos. No es necesario tener un límite; la página entera se reserva como una unidad.

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 concatenado con el offset de la página que se obtiene de la virtual address. El sistema operativo maneja los accesos a la memoria.

Una de las cosas que pueden parecer raras sobre la paginación es que, **SI BIEN EL PROGRAMA CREE QUE SU MEMORIA ES LINEAL, DE HECHO, SU MEMORIA ESTÁ DESPARRAMADA POR TODA LA MEMORIA FÍSICA COMO SI FUERA UN MOSAICO.**

Memoria paginada multinivel

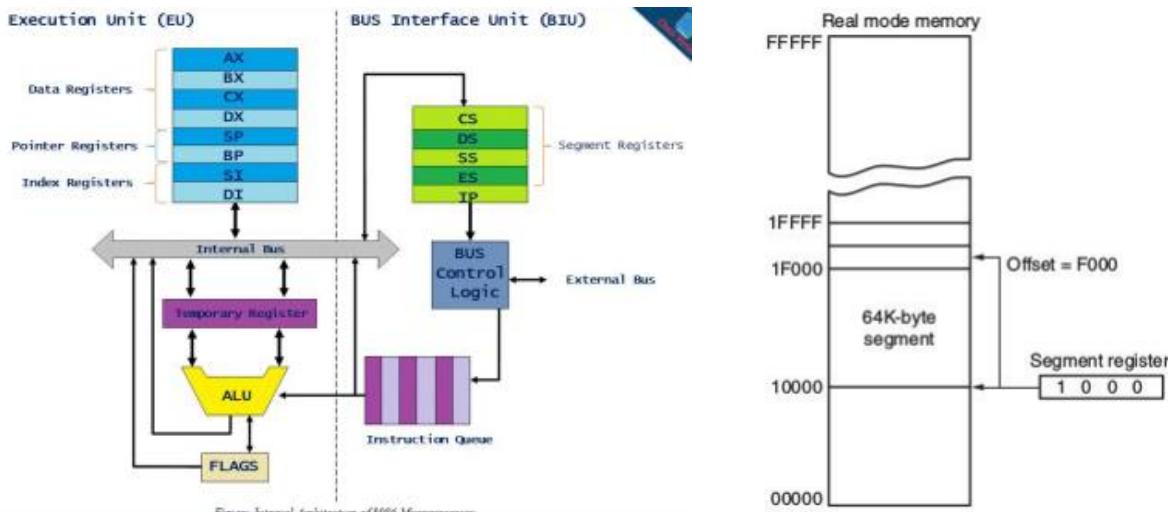


Casos de Estudio

Base and Bound en 8086

La arquitectura del procesador 8086 tiene un bus de datos de 20 bits, los registros tienen una longitud de 16 bits. Utiliza base and bound

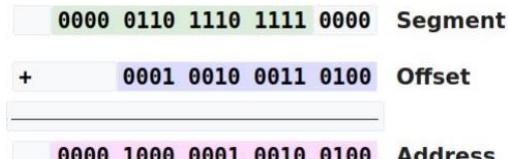
- Registros Generales: AX, BX, CX, DX
- Registros de Segmento CS: Code Segment DS: Data Segment SS: Stack Segment ES : Extra Data Segment
- SI, DI, BP, SP, IP: Registros de Punteros y Registro de Índices. (Base Pointer, Stack Pointer y Instruction Pointer)



Dado que era una arquitectura de 16 bits la cantidad máxima direccionable era 2¹⁶ que es 65536 bytes. Como se quedaban cortos porque eso era solo 64k de memoria, decidieron agregar 4 bits más a las direcciones de memoria llevando el bus de datos a 20 bits Una dirección virtual se considera como el valor del registro de segmento shifted 4 bit a izquierda, y luego se le suma el offset

1000:F00F virtual equivale a la dirección física:

→ 10000 + F00F 1F00F dirección de 20 bits



Combinaciones especiales:

- **CS:IP** localiza la próxima instrucción a ser ejecutada en modo real.
- **SS:SP** localiza la dirección del puntero al stack, a veces también puede ser SS:BP.
- **DS: BX,DI,SI** localizan el puntero a una dirección de memoria dentro del data address.
- **ES:DI** puntero al extra data address donde van los strings.

Modo Protegido x86

En x86 además del modo nativo, real mode, donde las direcciones accedidas son direcciones físicas, existe otro modo llamado Modo Protegido en el cual se proveen los mecanismos necesarios para la protección de memoria. Se implementó a partir del 80286.

- ✓ Permite direccionar a datos y programas más allá de 1 Mb de memoria física, así como también dentro del primer mega de memoria física.
- ✓ Es el sistema de virtualización que se utiliza es el de tabla de segmentos

Memoria segmentada en x86



La Unidad de Gestión de Memoria (MMU) transforma una dirección lógica en una dirección lineal mediante un circuito de hardware llamado unidad de segmentación; posteriormente, un segundo circuito de hardware llamado unidad de paginación transforma la dirección lineal en una dirección física.

Dirección lógica

Incluida en las instrucciones del lenguaje de máquina para especificar la dirección de un operando o de una instrucción. Este tipo de dirección encarna la conocida arquitectura segmentada 80 × 86, que obliga a los programadores de MS-DOS y Windows a dividir sus programas en segmentos. Cada dirección lógica consta de un segmento y un desplazamiento (o "offset") que denota la distancia desde el inicio del segmento hasta la dirección real.

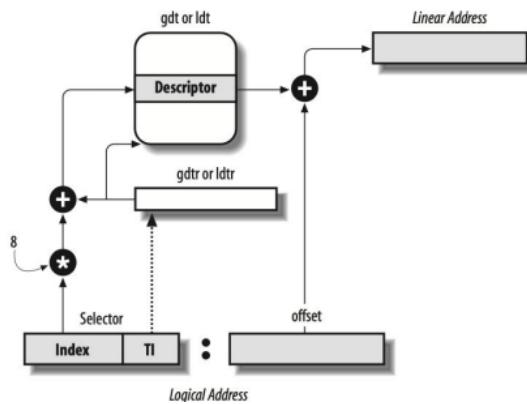
Dirección lineal (también conocida como dirección virtual)

Un único número entero sin signo de 32 bits que puede utilizarse para direccionar hasta 4 GB, es decir, hasta 4,294,967,296 celdas de memoria. Las direcciones lineales suelen representarse en notación hexadecimal; sus valores varían desde 0x00000000 hasta 0xffffffff. (equivalente para 64 bits)

Dirección física

Se utiliza para direccionar las celdas de memoria en los chips de memoria. Corresponden a las señales eléctricas enviadas a lo largo de los pines de dirección del microprocesador al bus de memoria. Las direcciones físicas se representan como números enteros sin signo de 32 bits o 36 bits.

- Examina el campo TI del Selector de Segmento para determinar en qué Tabla de Descriptores se almacena el Descriptor de Segmento. Este campo indica si el Descriptor está en la GDT (en cuyo caso la unidad de segmentación obtiene la dirección base lineal de la GDT desde el registro gdtr) o en la LDT activa (en cuyo caso la unidad de segmentación obtiene la dirección base lineal de esa LDT desde el registro ldtr).
- Calcula la dirección del Descriptor de Segmento a partir del campo de índice del Selector de Segmento. El campo de índice se multiplica por 8 (el tamaño de un Descriptor de Segmento), y el resultado se suma al contenido del registro gdtr o ldtr.
- Suma el desplazamiento ("offset") de la dirección lógica al campo Base del Descriptor de Segmento, obteniendo así la dirección lineal.



Global Descriptor Table (GDT): tabla guardada en memoria, apuntada por el registro llamado GDTR.

Una por todo el sistema y siempre accesible. Kernel y memoria ccompartida.

Local Descriptor Table (LDT): normalmente una por tarea, programa o proceso. Puede haber varias en el sistema pero solo una está activa en un momento dado.

Segment Selector: índice en la GDT que apunta a un Segment Descriptor, cada tabla tiene 8192 entradas

Segment Descriptor : es una entrada de 8 bytes en la GDT: 32 bits segmento. 20 bits de segmento límite dependiendo del G-bit

Segmentación en Linux

Linux utiliza la segmentación de una manera muy limitada dado que la segmentación y la paginación son algo redundantes, ya que ambas pueden utilizarse para separar los espacios de direcciones físicas de los procesos: la segmentación puede asignar un espacio de direcciones lineales diferente a cada proceso, mientras que la paginación puede mapear el mismo espacio de direcciones lineales en diferentes espacios de direcciones físicas.

Linux prefiere la paginación a la segmentación por las siguientes razones:

1. La gestión de memoria es más simple cuando todos los procesos usan los mismos valores de los registros de segmento, es decir, cuando comparten el mismo conjunto de direcciones lineales.
2. Uno de los objetivos de diseño de Linux es la portabilidad a una amplia gama de arquitecturas; en particular, las arquitecturas RISC tienen un soporte limitado para la segmentación.

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

TODOS LOS SEGMENTOS COMIENZAN EN 0x00000000 Y ESO HACE QUE EN LINUX LAS DIRECCIONES LÓGICAS COINCIDEN CON LAS DIRECCIONES LINEALES; ES DECIR, EL VALOR DEL CAMPO DE DESPLAZAMIENTO ("OFFSET") DE UNA DIRECCIÓN LÓGICA SIEMPRE COINCIDE CON EL VALOR DE LA DIRECCIÓN LINEAL CORRESPONDIENTE.

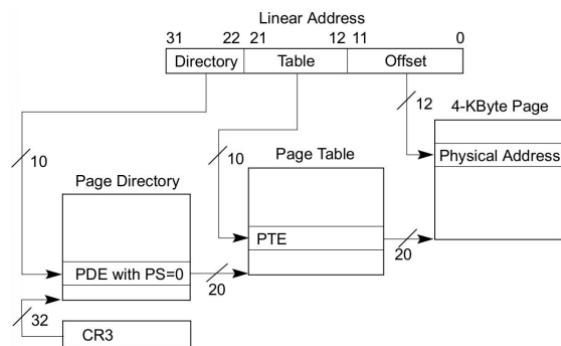
OBSERVA QUE LAS DIRECCIONES LINEALES ASOCIADAS CON DICHOS SEGMENTOS COMIENZAN EN 0 Y ALCANZAN EL LÍMITE DE DIRECCIONAMIENTO DE $2^{32} - 1$.

Memoria Paginada en x86

Page Directory Entry: Una entrada de la page directory, ocupa 4 bytes. La page directory posee 1024 entradas

Page Table Entry: Una entrada de la page table, ocupa 4 bytes. La page directory posee 1024 entradas

REGISTROS IMPORTANTES para la paginación: CR0 y CR3.



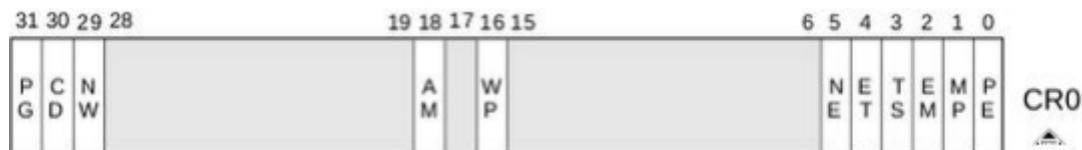
REGISTRO CR3

Es uno de los REGISTROS de CONTROL y es fundamental para la ADMINISTRACION de la MEMORIA VIRTUAL. El bit más a la izquierda del registro CR0 si está en 0 determina que la lineal address se convierte directamente en physical address para acceder a la memoria. Si PG está en 1 la lineal address debe ser convertida en physical address a través del mecanismo de paginación. CR3 contiene page directory base que contiene 1024 entradas de 4 bytes cada una, cada entrada en el page directory ocupa 4 bytes y direcciona a una page table que contiene 1024 entradas.

- **Función Principal:** 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.
- **Bits de Dirección de la Base de la Tabla de Directorios de Páginas (PDBR):** Estos bits contienen la dirección base de la tabla de directorios de páginas. En sistemas x86 sin PAE, los bits 31-12 de CR3 apuntan a la base de la tabla de directorios de páginas y los bits 11-0 son ignorados (y generalmente se establecen en 0).
- **PCD (Page-Level Cache Disable, Bit 4):** Si se establece, la paginación no se cachea. Sin embargo, este bit es a menudo ignorado, con las características de cacheo determinadas por las entradas individuales de la tabla de páginas.
- **PWT (Page-Level Write-Through, Bit 3):** Controla las características de caching para las tablas de páginas. Si se establece, se utiliza la política de cacheo de Write-Through; si se borra, se utiliza Write-Back. Son políticas de caché utilizadas en sistemas informáticos, especialmente en relación con la memoria caché de la CPU y algunas veces en sistemas de almacenamiento.
- **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: 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:

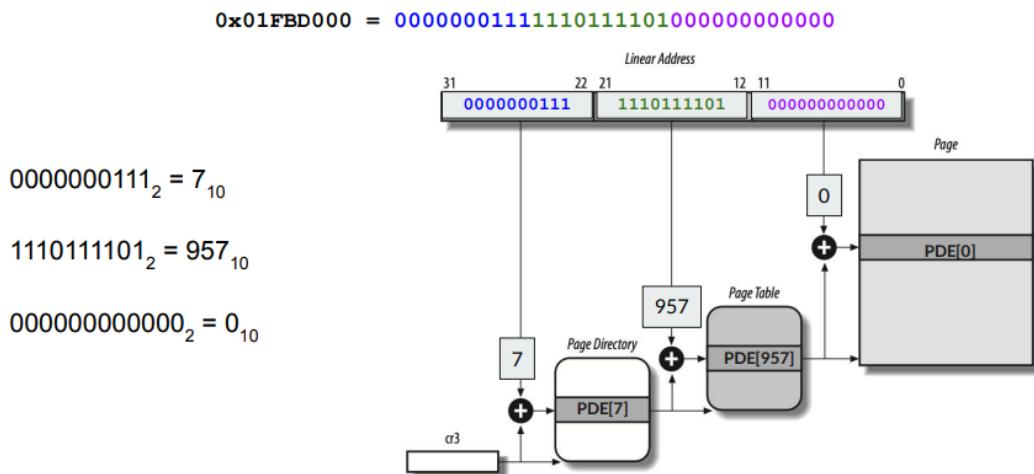
- **PE (Bit 0, Modo Protegido):** Cuando este bit está establecido, el procesador opera en modo protegido. De lo contrario, opera en modo real.
- **WP (Bit 16, Protección de Escritura):** Cuando está establecido, determina el comportamiento de las páginas de solo lectura en modo supervisor.
- **PG (Bit 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.

Cada uno de estos bits tiene un propósito específico y, en conjunto, determinan cómo el procesador x86 interactúa con la memoria, cómo maneja las excepciones y cómo opera en general.

El registro CR0 es esencial para inicializar el procesador al arrancar. Por ejemplo, al pasar de modo real a modo protegido (una etapa crucial durante el arranque de muchos sistemas operativos), el bit PE de CR0 se establece. Además, el bit PG es fundamental para habilitar o deshabilitar la paginación, una característica vital en sistemas operativos modernos.

Ejemplo de traducción

Considere un sistema x86 de memoria virtual paginada de dos niveles con un espacio de direcciones de 32 bits, donde cada página tiene un tamaño de 4096 bytes. Como se realiza la traducción de 0x01FBD000 Dirección virtual: 0x01FBD000 = 0000000111110111101000000000000



Ejemplo de parcial

Considere un sistema x86 de memoria virtual paginada de dos niveles con un espacio de direcciones de 32 bits, donde cada página tiene un tamaño de 4096 bytes. Un entero ocupa 4 bytes y se tiene un array de 50.000 enteros que comienza en la dirección virtual 0x01FBD000.

El arreglo se recorre completamente, accediendo a cada elemento una vez. En este proceso, **¿a cuántos frames de memoria física distintos (no la cantidad total de accesos) necesita acceder el sistema operativo para conseguir esto?** Recuerde contar las tablas de páginas intermedias, no solo las páginas que contienen los elementos del array

Cantidad total de bytes que ocupa el array: $50,000 \times 4 = 200,000$

Posición del primer byte: 0x01FBD000 (definido en el enunciado)

Posición del último byte: $0x01FBD000 + (50000 \times 4 - 1) = 0x01FEDD3F$

El rango de direcciones que el sistema operativo va a recorrer es entonces: [0x01FBD000, 0x01FEDD3F]

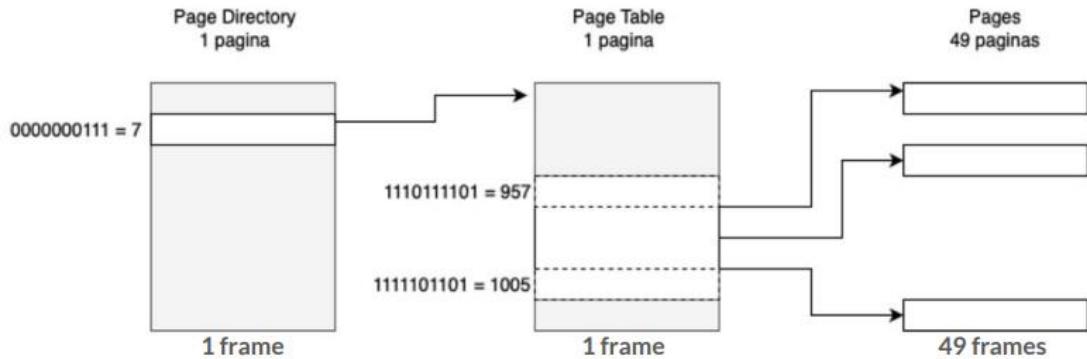
$0x01FBD000 = 0000000111\ 1110111101\ 000000000000$

$0x01FEDD3F = 0000000111\ 1111101101\ 110100111111$

Expresados en decimal para mayor claridad:

$0x01FBD000 = [7]\ [957]\ [0]$

$0x01FEDD3F = [7]\ [1005]\ [3391]$



1er nivel (el directorio de páginas): se usa un solo registro, en el índice 7. Este contiene la dirección (y metadata) de la única tabla de páginas que será necesaria en el segundo nivel. Si se requirieran más tablas de páginas de segundo nivel, este índice cambiaría para algunas direcciones involucradas en el rango. Esto no ocurre para el rango propuesto en este ejercicio.

2do nivel: Se utilizan varios registros, cada uno contiene la dirección de las páginas donde están los datos del array. Concretamente, se usan los registros desde el 957 hasta el 1005, es decir, 49 registros, lo que implica 49 páginas de datos.

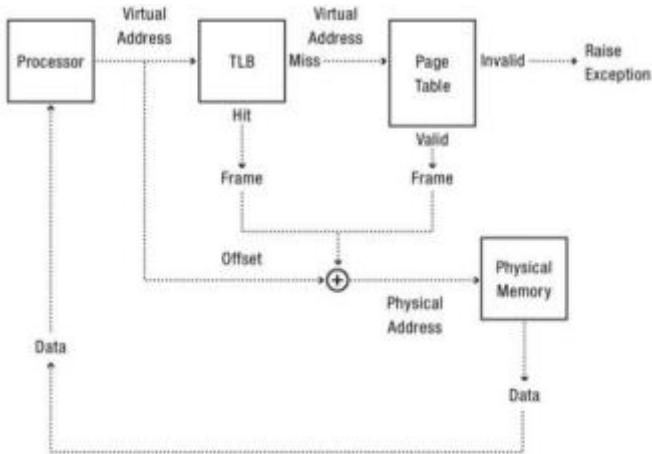
3er nivel: Cada página contiene 4096 bytes de datos. Si queremos guardar 200,000 bytes entonces necesitamos 48.8 páginas, es decir, necesitamos utilizar 49 páginas. Esto es consistente con el cálculo que hicimos antes a partir de las direcciones de los extremos del rango de bytes del array.

Resultado: 1 page directory (por definición de la arquitectura x86) + 1 page table (justificado anteriormente) + 49 páginas de datos (justificado anteriormente) = 51 páginas en total que el sistema operativo necesita utilizar para guardar y acceder a este array.

Translation Lookaside Buffer

HACIA UNA EFICIENTE ADDRESS TRANSLATION

- MECANISMO de hardware que hace uso del Cache para resolver el problema de la velocidad de traducción.
- Es parte de la MMU
- Pequeña tabla a nivel hardware que contiene los resultados de las recientes traducciones de memorias realizadas. Cada entrada de la tabla mapea una virtual page a una physical page
- Por cada referencia a la memoria virtual, el hardware primero chequea la TLB para ver si esa traducción está guardada ahí; si es así la traducción se hace rápidamente sin tener que consultar a la page table (la cual tiene todas las traducciones)
- TLB hit: Cuando existe matcheo y el procesador utiliza ese matcheo para formar la physical address, ahorrándose todos los pasos de la traducción.
- TLB miss: Cuando del proceso anterior no existe matcheo en la TLB y se deben chequear todas las tablas contra la virtual page.



Para que sea útil la búsqueda de la TLB, las entradas de la tabla son implementadas en una memoria muy rápida, memoria estática on-chip, situada muy cerca del procesador. De hecho, para mantener esta búsqueda rápida, muchos sistemas en la actualidad incluyen múltiples niveles de TLB. En general, cuanto más pequeña es la memoria, más rápida es la búsqueda. Si el primer nivel de la TLB produce un TLB miss existe otro nivel, que guarda más datos, y este es consultado y la traducción se realiza si se falla en ambos niveles.

Consistencia de la TLB

Cada vez que se introduce un cache en el sistema, se necesita considerar la forma de asegurar la consistencia del cache con los datos originales cuando las entradas en el mismo son modificadas. Una TLB no es la excepción. Para una ejecución correcta y segura de un programa, el sistema operativo tiene que asegurarse que cada programa ve su propia memoria y la de nadie más.

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.

Reducción de Permiso: ¿Qué sucede cuando el sistema operativo modifica una entrada en una page table? Normalmente NO SE PROVEE consistencia por hardware para la TLB; mantener la TLB consistente con la page table es responsabilidad del sistema operativo.

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.

Administración de Memoria Kernel

Mapeo Directo en xv6

El kernel accede a la RAM y a los registros de los dispositivos mapeados en memoria utilizando "mapeo directo"; es decir, ASIGNANDO los recursos a DIRECCIONES VIRTUALES que son iguales a las direcciones físicas. Por ejemplo, el propio kernel está ubicado en KERNBASE=0x80000000 tanto en el espacio de direcciones virtuales como en la memoria física.

El mapeo directo SIMPLIFICA EL CODIGO DEL KERNEL que lee o escribe en la memoria física. Por ejemplo, cuando fork asigna memoria de usuario para el proceso hijo, el asignador devuelve la dirección física de esa memoria; fork utiliza esa dirección directamente como una dirección virtual cuando copia la memoria de usuario del padre al hijo.

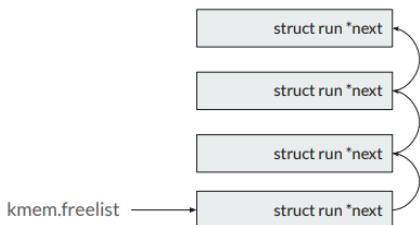
Hay un par de direcciones virtuales del kernel que no están mapeadas directamente:

- La página trampolín. Está mapeada en la parte superior del espacio de direcciones virtuales tanto para el espacio de kernel como usuario.
- Las páginas de pila del kernel. Cada proceso tiene su propia pila del kernel, que está mapeada en una dirección alta para que debajo de ella xv6 pueda dejar una página de protección sin mapear.

Kalloc: Alocador de memoria física

La estructura de datos del asignador es una lista libre de páginas de memoria física que están disponibles para asignación. Cada elemento de la lista de páginas libres es un struct run.

¿De dónde obtiene el asignador la memoria para almacenar esa estructura de datos? Almacena la estructura run de cada página libre en la propia página libre, ya que no hay nada más almacenado allí.



```
struct run {
    struct run *next;
};

void * kalloc(void) {
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
        kmem.freelist = r->next;
    release(&kmem.lock);

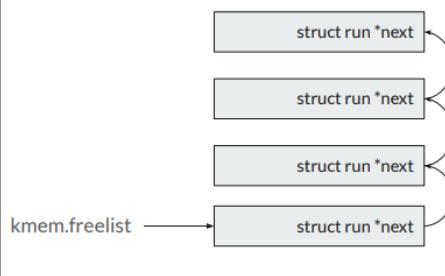
    if(r)
        memset((char*)r, 5, PGSIZE);
    return (void*)r;
}
```

Kfree

```
void kfree(void *pa) {
    struct run *r;
    ...

    r = (struct run*)pa;

    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
}
```



OBSERVACIONES:

- ★ kalloc no recibe parámetros porque siempre aloca de a una página y la pagina siempre tiene tamaño fijo (eg. 4 Kb)
- ★ Tanto kalloc como kfree operan con direcciones físicas, no virtuales. Otra función se encarga de mapearlas en las tablas de páginas
- ★ La lista en realidad funciona como una pila
- ★ Para que no se corrompa la estructura de la pila, la modificación se realiza en una región crítica, que se construye con locks (ver clase sobre concurrencia)

Mapeo de páginas

Objetivo: La dirección virtual de una página, alojar una página física y mapearla a esa dirección virtual. Esto es lo que hace el kernel cada vez que un proceso pide más memoria.

```
int mappage(pte_t *pagetable, uint32 va)
```

- pte_t *pagetable representa un array de pte_t, es decir una tabla de páginas. pte_t es un tipo de dato abstracto que representa una page table entry.
- uint32 va es la dirección de la página virtual que se desea mapear. Asumir que no se encuentra ya mapeada

Asuma que dispone de las siguientes funciones (en verde en el código siguiente):

- void* kalloc(): aloca y devuelve la dirección física de un frame de memoria disponible

Tipo de dato abstracto pte_t

- void pte_init(pte_t* pte, uint32 pa): inicializa la pte para apuntar a la dirección física pa
- uint32 pte_pa(pte_t* pte): devuelve la dirección física asociada al pte, o 0 si la pte no está mapeada.

En este ejemplo vamos a ignorar los permisos y flags de las páginas:

Para simplificar, vamos a usar un TDA pte_t que es una entrada de la tabla de páginas. Estas son dos funciones que van a operar sobre el tda. La estructura del pte_t es abstraída por el TDA, pero sabemos que ocupa 4 bytes en memoria

```
1 #define PAGE_SIZE 4096      // Tamaño de la página: 4KB
2 #define PTE_COUNT 1024      // Número de entradas por tabla (2^10)
3
4 typedef uint32 pte_t;        // Definimos pte_t como un entero de 32 bits
5
6
7 int mappage(pte_t *pagetable, uint32 va) {
8
9
10    // Máscaras para extraer los índices de las tablas de página
11    uint32 idx_11 = (va >> 22) & 0x3FF;    // Los primeros 10 bits
12    uint32 idx_12 = (va >> 12) & 0x3FF;    // Los siguientes 10 bits
13
14    ...
15 }
```

Notar las operaciones bitwise para obtener los índices de las tablas de páginas de nivel 1 y 2. Son corrimientos para obtener los primeros y segundos 10 bits de los 32 bits. Con esa máscara se hace un AND de 0X3FF, prende lo que tiene q prender y apaga lo q tiene q apagar.

```

...
14 // Obtenemos la entrada de la tabla de primer nivel (nivel 1)
15 pte_t *l2_pagetable = (pte_t *) pte_pa(&pagetable[idx_11]);
16
17 // Si la tabla de nivel 2 no existe, la creamos
18 if (l2_pagetable == 0) {
19
20     // Asignamos un nuevo frame de memoria física para la tabla de segundo nivel
21     l2_pagetable = (pte_t *)kalloc();
22
23     if (l2_pagetable == 0) {
24         // Error: No se pudo asignar memoria
25         return -1;
26     }
27
28     // Inicializamos la entrada de la tabla de nivel 1
29     // para apuntar a la nueva tabla de nivel 2
30     pte_init(&pagetable[idx_11], (uint32)l2_pagetable);
31 }
32
...

```

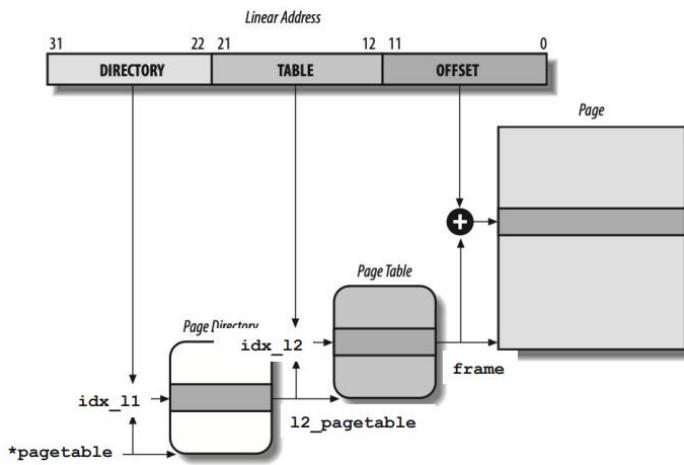
En este caso la tabla de nivel 1 en el índice que necesitábamos correspondiente a la tabla de nivel 2, no estaba mapeada. Es decir, es la primera vez que tratábamos de mapear una página con este índice en la tabla de nivel 1. Este código obtiene un frame y lo inicializa como tabla de nivel 2. Posteriormente inicializa el índice de la tabla de nivel 1 correspondiente. Pte: pa = page table to physical address: con el índice de la 1ra entrada, transformo eso en una physical address. De ese índice de la page table, agarro solo los primeros bits, usando la máscara. Me quedo con los primeros 22 bits (porque estoy en 4k). Una vez que tengo eso, ya tengo la entrada a la tabla del 1er nivel. Si me da 0 es que no hay página reservada para esa dirección de memoria (la busco en el page directory y me da un índice, si ese índice me da 0 es que no hay nada, lo reservo), entonces la debería crear.

```

...
33     // Asignamos un nuevo frame de memoria física para la página
34     void *frame = kalloc();
35     if (frame == 0) {
36         // Error: No se pudo asignar memoria
37         return -1;
38     }
39
40     // Inicializamos la entrada de la tabla de nivel 2
41     // para apuntar al frame de memoria física
42     pte_init(&l2_pagetable[idx_12], (uint32)frame);
43
44     // Éxito
45     return 0;
46 }

```

Resumen de punteros:



Páginas en xv6 RISC-V

Las páginas en RISC-V de 64 bits tienen 3 niveles en lugar de 2.

Los 25 bits superiores de la dirección virtual se ignoran

El bit V - Valid indica si la página apunta a otra página válida o si no está mapeada

Función walk en xv6

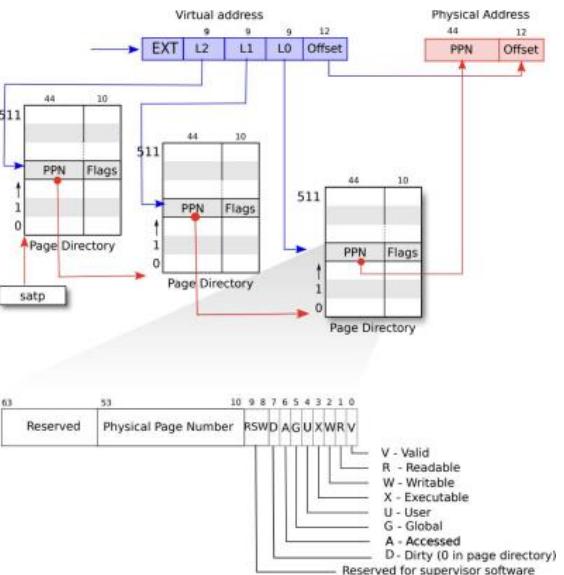
Se usa para obtener la pte_t de nivel inferior dada una virtual address

```
pte_t * walk(pagetable_t pagetable, uint64 va, int alloc)
{
    if(va >= MAXVA)
        panic("walk");

    for(int level = 2; level > 0; level--) {
        pte_t *pte = &pagetable[PX(level, va)];

        if(*pte & PTE_V) { // Si la página está presente
            pagetable = (pagetable_t)PTE2PA(*pte);

        } else {
            if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
                return 0;
            memset(pagetable, 0, PGSIZE);
            *pte = PA2PTE(pagetable) | PTE_V;
        }
    }
    return &pagetable[PX(0, va)];
}
```



Administración de Memoria Usuario

malloc()

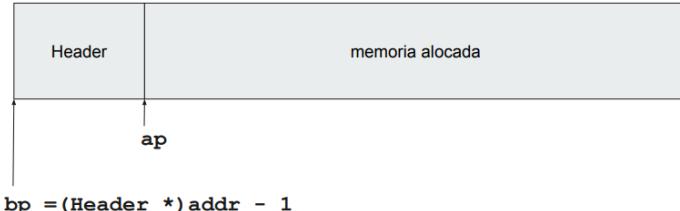
- En unix malloc() devuelve un bloque de size bytes alineado a 8-bytes (double word).
- No inicializa la memoria devuelta.
- Utiliza la system call sbrk o mmap.

free()

- Liberá bloques reservados en el heap.
- El ptr debe haber sido reservado previamente con malloc(), calloc() o realloc(). Si esto no sucede el comportamiento de free es INDEFINIDO

Hacer un malloc, te devuelve la cantidad de memoria que le pediste, pero también va a reservar espacio para un header, donde guarda metadata para que el free cuánta memoria hay que liberar. Malloc te devuelve el puntero a donde arranca el espacio reservado.

```
ap = malloc(n)
```

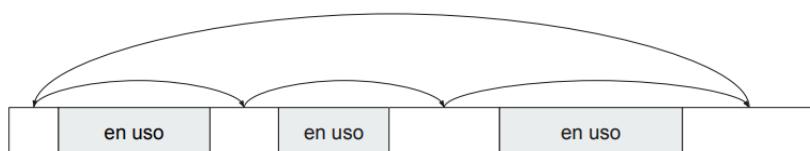


Cada vez que se solicita memoria mediante malloc, la libc produce un bloque de memoria alocada del tamaño solicitado (en este caso n) y devuelve un puntero al inicio de este bloque. Este bloque puede ser de cualquier tamaño, no necesariamente de una página o múltiplo de ella. Antes del bloque, libc construye un header para el bloque de memoria. Ese header será utilizado por la libc para gestionar la memoria alocada y no se supone que sea modificable por el usuario.

- ptr solo es importante en el caso de que el header corresponda a un bloque libre. En este caso apunta al siguiente bloque libre
- size siempre es relevante e indica el tamaño del bloque.
- size no es en bytes sino en múltiplos del tamaño del header mismo. Esto va a ser necesario y evidente cuando se quiera dividir un bloque

```
struct header {
    struct header *ptr;
    unsigned int size;
};

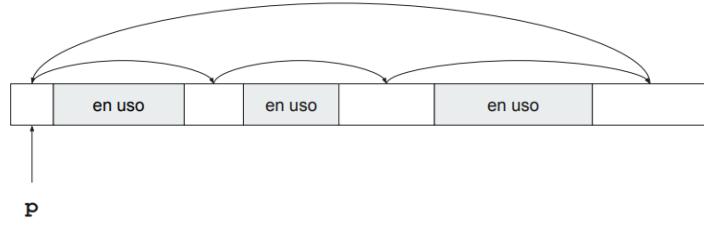
typedef struct header Header;
```



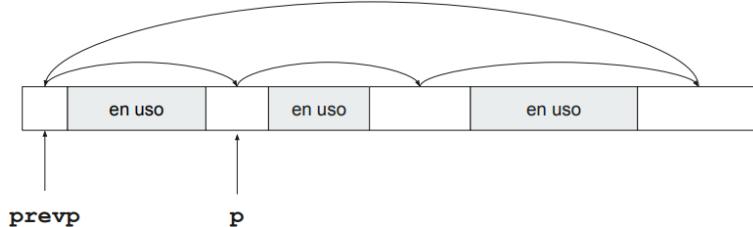
Se indican en blanco los bloques libres y en gris los bloques en uso. No están ilustrados, pero al principio de cada bloque (tanto blanco como gris) existe un header. En todos ellos size indica el tamaño de ese bloque. En los bloques blancos, ptr forma una lista enlazada circular de bloques libres. Los grises no están enlazados por lo que ptr no es relevante, independientemente del valor que tenga.

Buscar en bloque libre

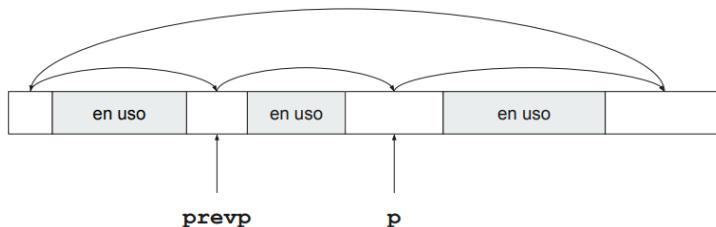
Cuando se quiere obtener memoria el primer paso es buscar un bloque libre. Se realiza un algoritmo de recorrido de lista usando un puntero p.



Se va a colocar un puntero prevp con la invariante de que siempre está en el bloque libre anterior a p. Este puntero va a ser necesario para “re-cablear” los bloques cuando se encuentre un bloque candidato.



Ambos punteros avanzan hasta que p se encuentra con un bloque libre que tiene al menos el espacio requerido por el usuario. Este es el bloque candidato de donde se devolverá una porción al usuario.

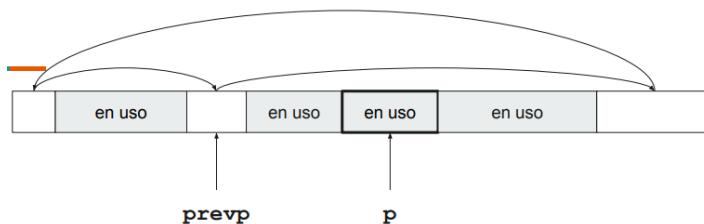


Ambos punteros avanzan secuencialmente. Esto sucede hasta que p encuentra un bloque donde size es mayor a la solicitada por el usuario.

Opción 1: Reservar el bloque completo

En este caso, se da la gran coincidencia de que el bloque encontrado tiene exactamente la memoria solicitada por el usuario. Entonces, el cableado es simple: el bloque anterior debe apuntar al siguiente de p. malloc devuelve el bloque. Como p es el header del bloque, p+1 por aritmética de punteros va a apuntar al espacio libre en sí del bloque, no al header.

OBS: Notar como el bloque se deslinquea de la lista y pasa a estar en uso

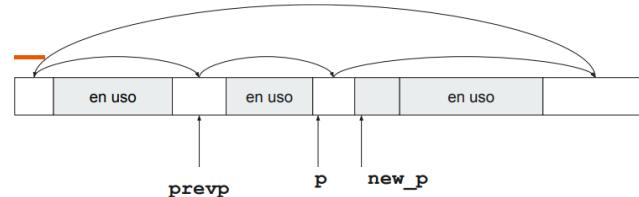


```
prevp->ptr = p->ptr;
return (p+1); // p es de tipo Header
```

Opción 2: Split

Ocurre cuando el bloque es más grande que lo necesario por el usuario, por ende, se va a dividir en lugar de devolverse completo. El cableado es más complicado. Se va a devolver espacio en la cola del bloque, no al principio

- Se reduce el tamaño del bloque actual en la cantidad de unidades que estamos extrayendo de la cola.
- Se construye un puntero que apunta al header del bloque a devolver. Esto se logra nuevamente con aritmética de punteros y considerando que size indica el tamaño tomando como unidad de medida no bytes, sino el tamaño de un header.
- Se define el tamaño de este bloque como nunits, es decir, lo que estamos devolviendo.
- No hace falta tocar los punteros, el siguiente sigue siendo el siguiente.
- Se devuelve el espacio que está después del nuevo header



```

p->size -= nunits;
new_p = p + p->size;
new_p->size = nunits;

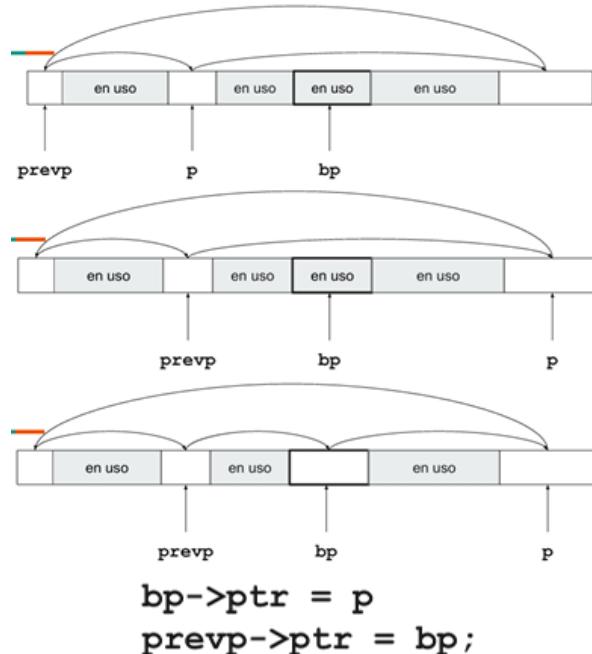
return (new_p + 1); // new_p es de tipo Header
  
```

Liberar el bloque

Debe comenzar con una búsqueda.

Necesitamos obtener el bloque libre que está antes y después del bloque a liberar (No consideraremos los casos de borde donde no existen bloques libres antes o después del bloque a liberar. El algoritmo es similar).

Una vez que encontramos que el bloque a liberar está entre **prevp** y **p** procedemos a recablear los punteros.



```

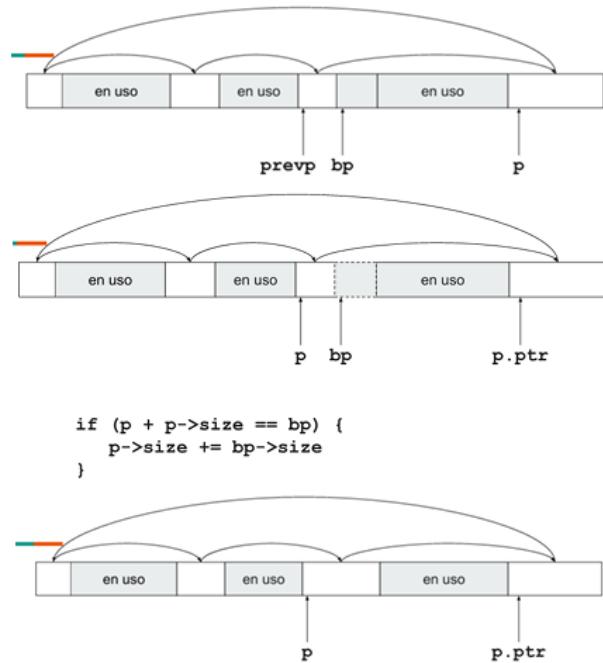
bp->ptr = p
prevp->ptr = bp;
  
```

Liberar el bloque: Coalesce(juntar)

En el caso anterior, el bloque a liberar estaba entre dos bloques en uso. En este caso el bloque a liberar es adyacente a otro bloque libre.

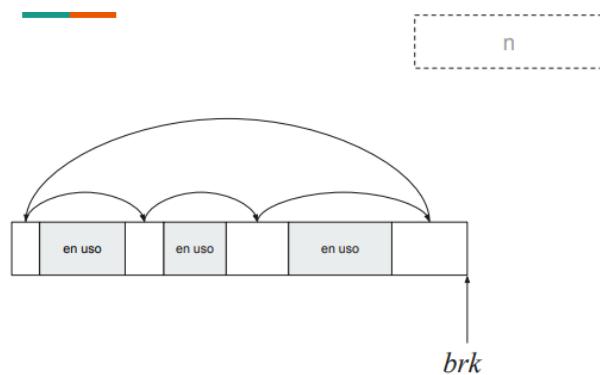
libc va a tratar de juntar el bloque libre que está antes con el nuevo bloque liberado para evitar muchos bloques libres y adyacentes.

Coalesce se puede hacer para bloques a la izquierda o a la derecha. En este bloque juntamos un bloque liberado con un bloque libre a la izquierda, este caso es fácil. Simplemente hay que aumentar el tamaño del bloque libre a la izquierda y todos siguen funcionando.

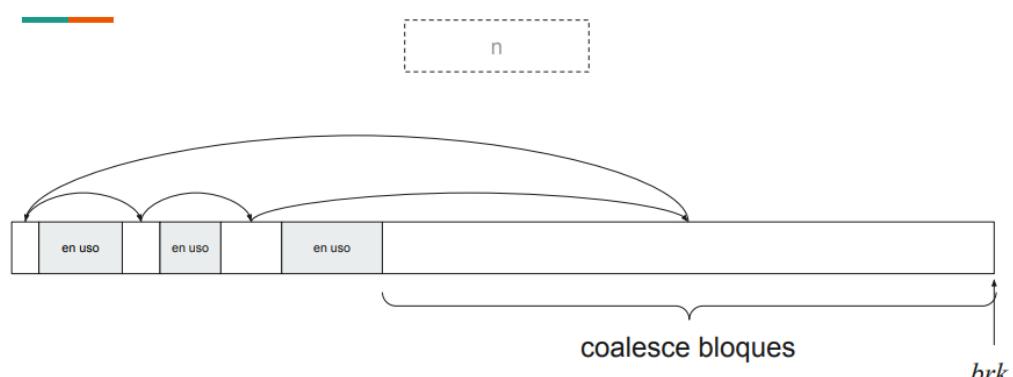


Reservar memoria pidiendo al SO

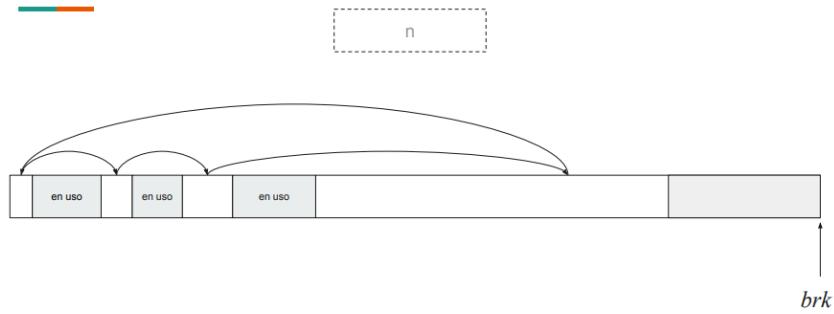
Ocurre cuando se pide un bloque de memoria más grande que todos los disponibles, cuando ya llegue al break, vuelve al primero de manera circular.



La conexión del bloque grande se puede hacer ejecutando free sobre el bloque. Esto además nos permite aplicar optimizaciones de coalesce. En este caso agregamos el bloque y lo juntamos con un bloque libre que estaba antes.



Finalmente, el algoritmo de malloc se ejecuta de nuevo. Ahora si encontrara un espacio disponible.



CONCURRENCIA

DEFINICION: ACCESO A UN RECURSO COMPARTIDO POR VARIAS ENTIDADES → TIENE QUE HABER ALGÚN MECANISMO PARA PODER USAR EL RECURSO SIN QUE HAYA PROBLEMAS DE CONSISTENCIA: SINCRONIZACIÓN.

Sincronización

La programación multihilo extiende el modelo secuencial de programación de un único hilo de ejecución. En este modelo se pueden encontrar dos escenarios posibles:

- Un programa está compuesto por un conjunto de threads independientes que operan sobre un conjunto de datos que están completamente separados entre sí y son independientes.
- Un programa está compuesto por un conjunto de threads que trabajan en forma cooperativa sobre un set de memoria y datos que son compartidos.

En un programa que utiliza un modelo de programación de threads cooperativo, la forma de pensar secuencial no sirve:

- La ejecución del programa depende de la forma en que los threads se intercalan en su ejecución, esto influye en los accesos a la memoria de recursos compartidos.
- La ejecución de un programa es no determinística. Diferentes corridas pueden producir distintos resultados, por ejemplo debido a decisiones del scheduler.
- Los compiladores y el procesador físico pueden reordenar las instrucciones. Los compiladores modernos pueden reordenar las instrucciones para mejorar la performance del programa que se está ejecutando, este reordenamiento es generalmente invisible a los ojos de un solo thread

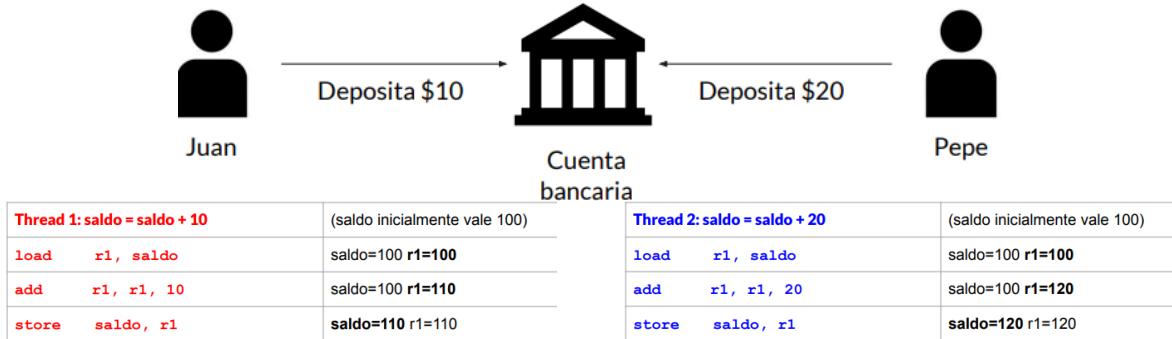
Teniendo en cuenta lo anterior, la programación multithreading puede incorporar bugs que se caracterizan por ser: utiles, no determinísticos, no reproducibles

El approach a seguir en estos casos es:

- 1) estructurar el programa para que resulte fácil el razonamiento concurrente y
- 2) utilizar un conjunto de primitivas estándares para sincronizar el acceso a los recursos compartidos.

Race Conditions

Una race condition se da cuando el resultado de un programa depende en como 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. EJ:



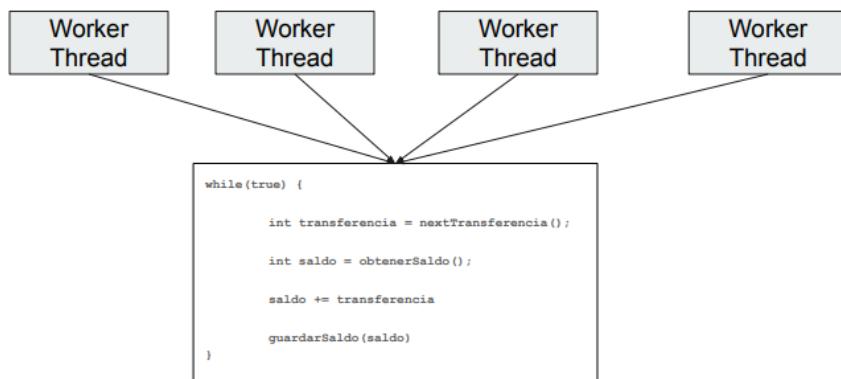
Se podrian ejecutar asi:

load r1, saldo	saldo=100 r1=100
add r1, r1, 10	saldo=100 r1=110
store saldo, r1	saldo=110 r1=110
load r1, saldo	saldo=110 r1=110
add r1, r1, 20	saldo=110 r1=130
store saldo, r1	saldo=130 r1=130

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!!!

Una mezcla donde se pierde plata → Esto es un race condition. Esto pasa porque NO SON operaciones ATOMICAS y se pueden intercalar entre sí, los contextos son distintos, pero están compartiendo un recurso (el saldo). → Como pueden ocurrir de distintas maneras y nunca vamos a tener un resultado asegurado, la ejecución es no determinista.

Un típico proceso para procesar transacciones:



Sección critica: Es una abstracción, mediante la cual se define que en una sección de código solo puede haber un thread en ejecución a la vez. Define una sección de código que se ejecuta bajo “exclusión mutua”

```

while(true) {
    int transferencia = nextTransferencia();
    int saldo = obtenerSaldo();
    saldo += transferencia;
    guardarSaldo(saldo);
}

```

Mientras que solo un thread a la vez entre a esta zona, esta todo bien!

Locks

Variable que PERMITE la SINCRONIZACION mediante la exclusión mutua, cuando un thread tiene el candado o lock ningún otro puede tenerlo.

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.

```
while(true) {  
    int transferencia = nextTransferencia();  
    obtener(lock)  
    int saldo = obtenerSaldo();  
    saldo += transferencia  
    guardarSaldo(saldo)  
    dejar(lock)  
}
```

Lock Contention: ocurre cuando múltiples threads o procesos intentan acceder a un recurso compartido simultáneamente, y al menos uno de ellos está bloqueado esperando un lock.

Propiedades

- ◊ Exclusión mutua: como mucho un solo Thread posee el lock a la vez.
- ◊ Progress: Si nadie posee el lock, y alguien lo quiere ... alguno debe poder obtenerlo.
- ◊ Bounded waiting: Si un thread quiere acceder al lock y existen varios threads en la misma situación, los demás tienen una cantidad finita (un límite) de posibles accesos antes que T lo haga.
Eventualmente el procesador le cede el lock a todos los que están esperando. Esta propiedad de los programas se llama liveliness

Spinlock

Lock que se implementa usando un “busy wait” -> Técnica de programación en la que un proceso o hilo de ejecución espera activamente en un loop a que ocurra un evento en lugar de dormirse o bloquearse. Esto significa que el proceso está ocupado en un bucle continuo verificando repetidamente si el evento ha ocurrido, sin realizar ninguna otra tarea útil en ese tiempo. ¿Cuándo se debe usar un spinlock?

- ✓ Cuando el tiempo que se posee el lock es corto
- ✓ Cuando hay poca contention por los locks. Locking optimista, lo más probable es obtener el lock y no competir con otros threads.
- ✓ Cuando hay paralelismo real (multiples procesadores). Sino no habrá ningún otro hilo realmente corriendo que pueda liberar el lock.

Operaciones Atómicas

Es una instrucción del procesador que se garantiza que se ejecuta completa: “todo o nada”.

- No puede haber una lectura parcial de valores, o race conditions
- No puede haber interrupciones entre la lectura y escritura a memoria, tampoco otros procesadores pueden modificar los valores de memoria entre la lectura y escritura de la instrucción. La atomicidad es garantizada por diseño; la arquitectura del procesador define qué instrucciones son atómicas y la implementación del hardware garantiza que la instrucción es realmente atómica.

Las más interesantes para nuestra discusión, son las instrucciones que hacen más de una cosa atómicamente.

Sleep lock

Lock que se implementa cediendo el procesador. El proceso si no logra conseguir el lock se va a dormir. Cuando el lock es liberado los procesos esperando por el lock se despiertan. 2 alternativas para despertarlos:

1. El OS elige uno y lo despierta. Ese obtiene el lock.
2. El OS despierta a todos y todos vuelven a competir para obtener el lock (xv6 hace esto)

¿Cuándo se debe usar un sleep lock?

- ✓ Cuando la espera será larga (eg. cuando se espera que el disco lea bloques) y no queremos bloquear el acceso al CPU para los otros procesos.
- ✓ En sistemas de un solo procesador, donde un spin-lock solo sería desalojado del procesador al acabarse su timeslice.

FILE SYSTEM

UNA ABSTRACCIÓN DEL SISTEMA OPERATIVO QUE PROVEE DATOS PERSISTENTES CON UN NOMBRE

- Permite a los usuarios organizar sus datos para que se persistan a través de un largo período de tiempo.
- Datos persistentes son aquellos que se almacenan hasta que son explícitamente (o accidentalmente) borrados, incluso si la computadora tiene un desperfecto con la alimentación eléctrica.

El hecho de que los datos tengan un nombre es con la intención de que un ser humano pueda acceder a ellos por un identificador que el sistema de archivos le asocia al archivo en cuestión. Además, esta posibilidad de identificación permite que una vez que un programa termina de generar un archivo otro lo pueda utilizar, permitiendo así compartir la información entre los mismos.

Decisiones de Diseño

El sistema de archivos de Unix tomo algunas decisiones de diseño fundamentales que influenciaron los sistemas de archivos posteriores:

- ◊ Una estructura jerárquica
- ◊ Tratamiento consistente de los datos de los archivos
- ◊ La habilidad de crear y borrar archivos
- ◊ Crecimiento dinámico de archivos
- ◊ Protección de los datos del archivo
- ◊ El tratamiento de los periféricos como archivos

Estructura jerárquica

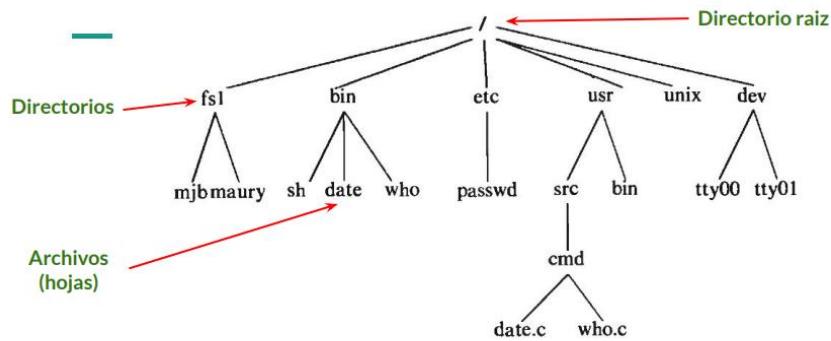


Figure 1.2. Sample File System Tree

Tratamiento consistente de archivos

- Los programas ven a los archivos como streams (flujos) de bytes.
- Al nivel de las system calls (eg. read y write) no se asume que haya una codificación o formato (eg. ASCII, Unicode, etc.) Todos los archivos son una secuencia de bytes, y nada mas

Permisos

Los archivos tienen asociados permisos. El esquema de permisos clásico de UNIX:

- 3 permisos: read, write y execute
- 3 clases de usuarios: owner, grupo, todos los demás

Dispositivos como archivos

Todo es un archivo en UNIX (eg. discos rígidos, CD-Rom, terminales, el proc filesystem).

- Permite un acceso uniforme a los datos de los dispositivos, porque todos los dispositivos son archivos, y los archivos son secuencias de bytes.
- Permite un manejo de permisos unificado, porque todos los dispositivos son archivos y los archivos tienen permisos.

Archivos

Un archivo es una colección de datos con un nombre específico. Proveen una abstracción de más alto nivel que la que subyace en el dispositivo de almacenamiento; un archivo proporciona un nombre único y con significado para referirse a una cantidad arbitraria de datos.

Metadata: información acerca del archivo que es comprendida por el Sistema Operativo, esta información es: tamaño, fecha de modificación, propietario, información de seguridad (que se puede hacer con el archivo).

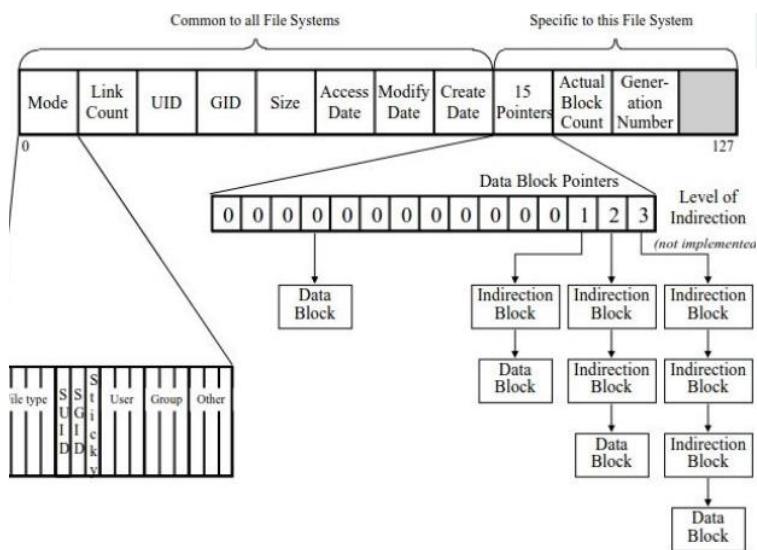
Datos: son los datos propiamente dichos que quieren ser almacenados. Desde el punto de vista del Sistema Operativo, un archivo o file no es más que un arreglo de bytes sin tipo.

UN ARCHIVO SE IMPLEMENTA COMO UN DENTRY APUNTANDO A UN INODO.

Inodo (index node)

Estructura que almacena información sobre un archivo. Contiene metadatos del archivo, pero no el contenido del archivo en sí. Incluye detalles como:

- Tamaño del archivo
- ID del propietario y del grupo
- Permisos del archivo (lectura, escritura, ejecución)
- Tiempos de acceso, modificación y cambio del inodo
- Número de enlaces (enlaces duros) al archivo
- Punteros a los bloques de disco donde se almacena el contenido real del archivo



EL INODO CONTIENE PUNTEROS A BLOQUES DE DATOS QUE CONTIENEN LOS DATOS DEL ARCHIVO EN SÍ.

- ✗ **EL INODO NO CONTIENE LOS DATOS DEL ARCHIVO**
- ✓ **EL INODO SI CONTIENE LA METADATA DEL ARCHIVO**

Tipos de Inodo en Linux

Los tipos de inodo en Linux nos dan idea de la variedad de “objetos” que podemos anclar al filesystem. Se especifican en el campo mode:

Macro	Value	Description
S_IFSOCK	0140000	Socket file
S_IFLNK	0120000	Symbolic link
S_IFREG	0100000	Regular file
S_IFBLK	0060000	Block device file
S_IFDIR	0040000	Directory
S_IFCHR	0020000	Character device file
S_IFIFO	0010000	FIFO (named pipe)

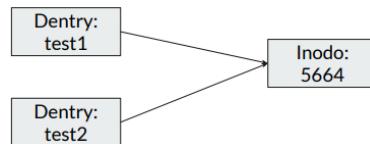
Dentry

Un Dentry (directory entry) representa la relación entre el nombre de un archivo y su inodo correspondiente.

→ Los directorios son listas de dentries.

La función principal de un dentry es AYUDAR a RESOLVER NOMBRES de ARCHIVOS y RUTAS en el sistema de archivos. Cuando el sistema de archivos busca un archivo, utiliza las dentries para traducir la ruta al inodo correspondiente, que contiene la información del archivo.

Puede haber más de un Dentry apuntando a un mismo inodo. Esto es efectivamente un HARD LINK.



Los inodos NO estan asociados a un sistema jerarquico.

Directrios

Son archivos especiales de tipo “directorio”. El contenido de estos archivos es una lista de dentries.

- ◊ Un dentry no apunta a otro dentry, solo a inodos
- ◊ Un inodo no apunta a otros inodos, solo a datos.
- ◊ Pero un dentry puede apuntar a un inodo tipo directorio, que apunta a datos y los datos contienen una lista de dentries. Así se implementa el sistema jerárquico

	inode	rec_len	file_type	name_len	name		
0	21		12	1	.	\0	\0
12	22		12	2	.	.	\0
24	53		16	5	h	o	m
40	67		28	3	u	s	r
52	0		16	7	o	l	d
68	34		12	4	s	b	i
					n		

Recapitulando:

Abstracción	Entidad del Kernel
Los archivos en si (regulares, directorios, dispositivos, o especiales)	Inodo
Los nombres asociados a los archivos	Dentry

- Un inodo apunta a datos
- Un dentry apunta a inodos

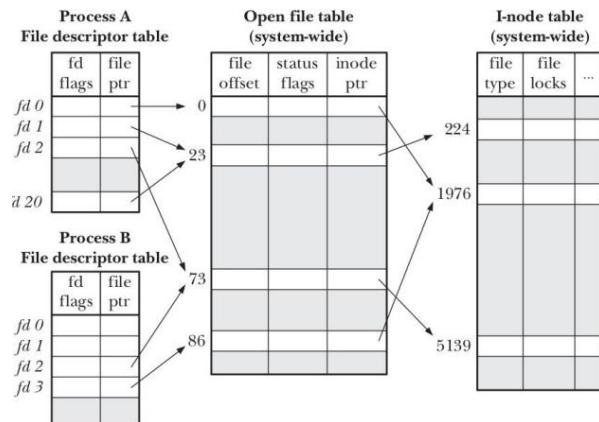
Pero:

- Un inodo nunca apunta a otros inodos
- Un dentry nunca apunta a otros dentries

La jerarquía del filesystem se forma porque un directorio contiene algunos dentries que apuntan a inodos de tipo DIRECTORY, y estos apuntan a bloques de datos que contienen listas de dentries.

El componente que falta para unir el filesystem con los procesos es la tabla de archivos abiertos. Esta tabla vincula un file descriptor (open, read, write, etc.) con el inodo que representa el archivo. El file

(miembro de la open file table) contiene el offset. Esto es el “cursor” que indica donde se va a escribir o leer a continuación.



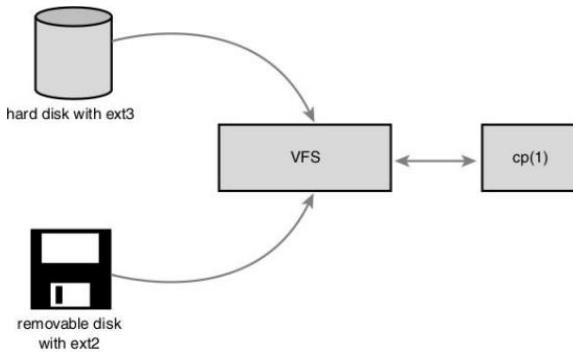
Virtual FileSystem

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
- inter-operar

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

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.

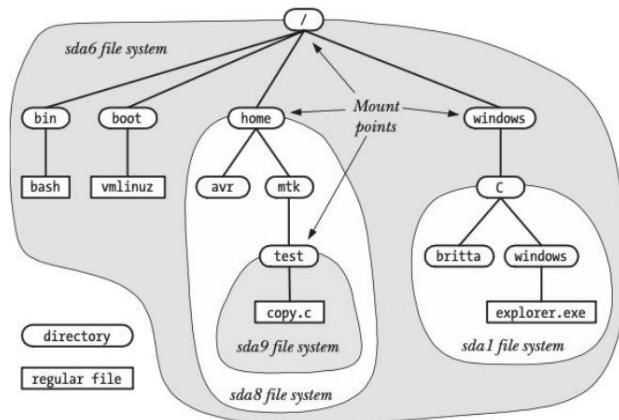


FileSystem Abstraction Layer

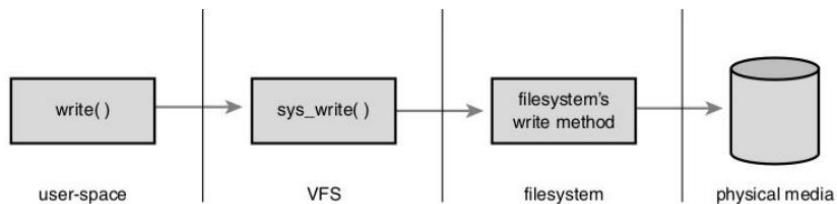
Tipo genérico de interfaz para cualquier tipo de filesystem que es posible sólo porque el kernel implementa una capa de abstracción que rodea esta interfaz para con el sistema de archivo de bajo nivel.

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.



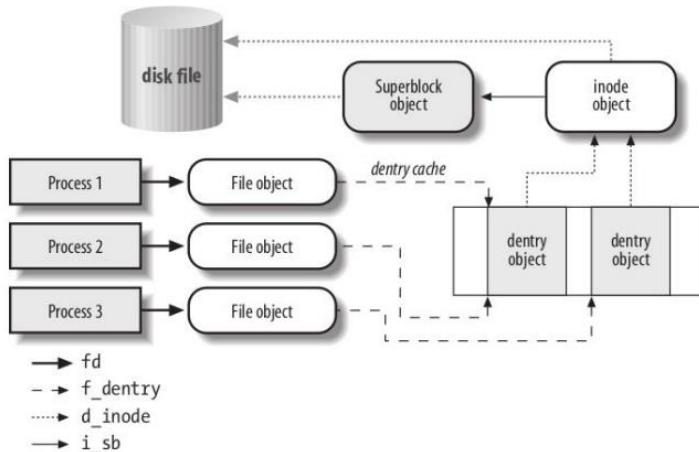
Esta capa de abstracción trabaja mediante la definición de interfaces conceptualmente básicas y de estructuras que cualquier sistema de archivos soporta. Los filesystems amoldan su visión de conceptos como “esta es la forma de cómo abro un archivo” para matchear las expectativas del VFS, todos estos 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.



VFS: Sus Objetos

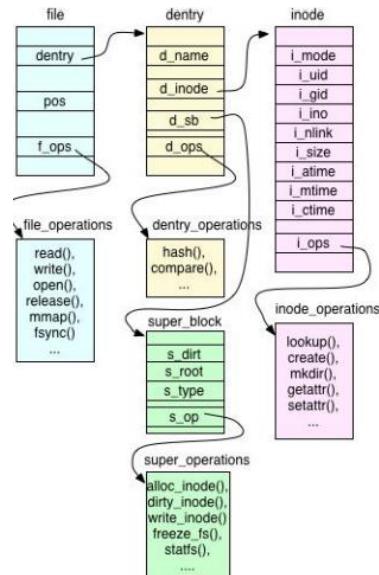
VFS presenta una serie de estructuras que modelan un filesystem, estas estructuras se denominan objetos (programadas en C). Estos Objetos son:

- El super bloque, que representa a un sistema de archivos.
- El inodo, que representa a un determinado archivo.
- El dentry, que representa una entrada de directorio, que es un componente simple de un path.
- El file que representa a un archivo asociado a un determinado proceso



VFS: Operaciones

- 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.



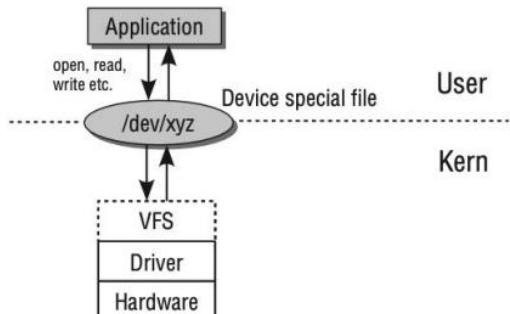
Dispositivos

Estructura para el acceso a dispositivos

Los dispositivos se representan como archivos especiales de tipo DEVICE

Se usan las mismas system calls que para operar con archivos normales

El VFS interactúa con una capa de Drivers
Los Drivers son los que implementan las operaciones de bajo nivel que interactúan con el hardware.



El directorio /dev

En general los dispositivos se cargan como archivos especiales en el directorio /dev Ejemplos de dispositivos que se encuentran ahí (se pueden listar con ls):

- /dev/sda: El primer disco duro en el sistema.
- /dev/sda1: La primera partición del primer disco duro.
- /dev/ttys0: Un puerto serie (como un puerto COM en sistemas Windows).
- /dev/loop0: Un dispositivo de bucle que permite tratar un archivo como si fuera un disco.
- /dev/null: Un "dispositivo" que descarta cualquier dato que se le escriba.
- /dev/random: Proveedor de números aleatorios.

Tipos de dispositivos

Existen dos tipos fundamentales de dispositivos en Unix que se pueden vincular como nodos al filesystem:

Dispositivos de bloque

Permiten la transferencia de datos byte a byte. Estos dispositivos no almacenan datos en bloques, sino que transmiten la información carácter por carácter, lo que los hace adecuados para dispositivos que no requieren procesamiento de grandes cantidades de datos de una sola vez. Ejemplos de dispositivos de

carácter: Terminales (como consolas o dispositivos tty), Puertos serie (para dispositivos como ratones o modems), Impresoras de tipo línea a línea.

En el sistema, estos dispositivos suelen encontrarse en /dev, con nombres como /dev/ttys0 para un puerto serie.

Dispositivos de carácter

Permiten la transferencia de datos en bloques de tamaño fijo (normalmente de varios bytes). Este tipo de dispositivo es adecuado para hardware que requiere el acceso a grandes cantidades de datos de forma eficiente, como los sistemas de almacenamiento. Ejemplos de dispositivos de bloque: Discos duros, Unidades de estado sólido (SSD), Memorias USB

En el sistema, estos dispositivos también se encuentran en /dev, con nombres como /dev/sda para un disco duro o una partición específica.

NOTA: EXISTEN TAMBIÉN LOS DISPOSITIVOS DE RED, QUE NO SE VINCULAN CON EL FILESYSTEM DEL SISTEMA

Cómo se vinculan los dispositivos al filesystem

Los dispositivos de I/O tienen un par de números asociados llamados números mayor y menor. Estos números implementan un sistema muy básico de ruteo hacia los dispositivos en sí.

Major Number: El número mayor identifica driver que maneja un grupo específico de dispositivos. El kernel utiliza este número para determinar qué controlador debe gestionar las solicitudes de entrada/salida (I/O) dirigidas a un dispositivo específico.

Minor Number: El número menor identifica un dispositivo específico dentro del grupo de dispositivos gestionado por un mismo controlador. En otras palabras, mientras que el número mayor se refiere al controlador, el número menor identifica a un dispositivo particular bajo ese controlador.

Los dispositivos son archivos especiales de tipo DEVICE que se crean usando la system call mknod. En general crear nodos de dispositivo no es necesario, esto lo resuelve el programa udev que dinámicamente monitorea y crea nodos de dispositivo. Además, resuelve la gestión de **major y minors** para los cuales “se asume” que el administrador del sistema tiene conocimiento de cómo obtenerlos para cada dispositivo.

Ahora sabemos crear nodos que representan dispositivos, pero el mismo es un archivo es decir bytes. Si es un disco no nos interesan los bloques de bytes, nos interesa el filesystem que esta contenido en los mismos.

```
sudo mount -t ext4 /dev/sda1 /mnt/mi_directorio
```

-t ext4: es el tipo de filesystem que estamos cargando. Esto le permite al kernel saber como interpretar los bloques del dispositivo de bloques.

/dev/sda1: es el dispositivo en bloques origen del filesystem.

/mnt/mi_directorio: es el punto de montaje, es decir, donde se va a colocar el filesystem

No todos los filesystems se corresponden con un dispositivo. ¡Algunos son enteramente virtuales!

Ej: procfs. El procfs (sistema de archivos de procesos) es un sistema de archivos virtual en los sistemas operativos tipo Unix (incluyendo Linux) que proporciona una interfaz para acceder a la información sobre los procesos y otros datos del sistema directamente desde el kernel.

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

Algunas 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. Tiene muchas flags que pueden combinarse. Crea inodos regulares.

creat(): Equivale a llamar a open() con los flags O_CREAT|O_WRONLY|O_TRUNC. Crea inodos regulares.

close(): Cierra un file descriptor. Si este ya está cerrado devuelve un error.

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.

write(): Escribe hasta una determinada cantidad (count) de bytes desde un buffer que comienza en buf al archivo referenciado por el file descriptor.unt. El número de bytes escrito puede ser menor al indicado por count.

lseek(): Repositiona 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)

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.

Podría parecer que existe una correspondencia uno a uno entre un file descriptor y un archivo abierto. Sin embargo, no es así. Es a veces muy útil y necesario tener varios file descriptors referenciando al mismo archivo abierto. Estos file descriptors pueden haber sido abiertos en un mismo proceso o en otros. Para ello existen tres tablas de descriptores de archivos en el kernel:

- Per-process file descriptor table
- Una tabla system-wide de open file descriptors
- La file system i-node table

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 sistema 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 .

- 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 almacenarán:

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

Hard links y Soft links

HARD LINK: se produce cuando mas de un dentry apunta al mismo inodo

SOFT LINK : es un archivo de tipo especial que contiene la ruta al archivo de destino.

Característica	Hard link	Soft Link (Symlink)
El inodo apunta a:	Los datos del archivo	Un bloque con la ruta del archivo
Relacion con el archivo	Mismo inodo	Archivo separado
Funcionamiento tras borrar el destino	Sigue funcionando	Se rompe
Directorios	No se permite	Si se permite
Diferentes particiones	No se permite	Si se permite
Tamaño	No ocupa espacio extra	Ocupa como un archivo pequeño

Otras system call:

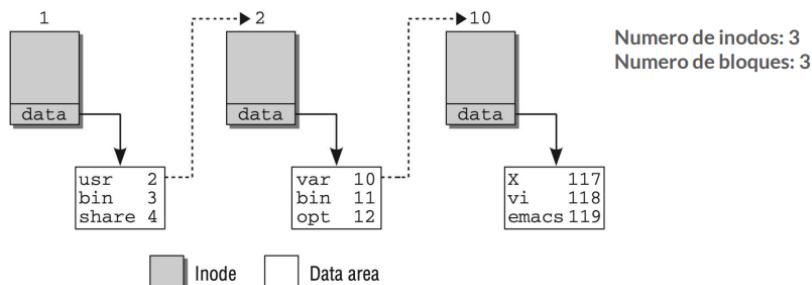
link(): Crea un nuevo nombre para un archivo. Esto también se conoce como un link (hard link). Este nuevo Nombre puede ser usado exactamente como el viejo nombre para cualquier operación, es más ambos nombres se refieren exactamente al mismo archivo y es imposible determinar cuál era el nombre original.

symlink(): Crea un soft link para un archivo. Crea un link simbólico.

unlink(): 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.

mkdir(): Crea directorios.

Como se resuelve /usr/bin/emacs:



Ejemplos de parcial

Se prepara el sistema con los siguientes comandos:

```
# mkdir /dir  
# echo 'hola' > /dir/x
```

A cuantos inodos y bloques accede el siguiente comando

```
# ls -l /dir/x
```

```
Lee inodo /, es de tipo DIRECTORIO  
Lee bloque del directorio /  
Lee inodo dir, es de tipo DIRECTORIO  
Lee bloque del directorio dir  
Lee inodo x, es de tipo REGULAR
```

No necesita leer el bloque de datos de x, ls no accede a los datos:

```
3 inodos  
2 bloques
```

Se prepara el sistema con los siguientes comandos:

```
# mkdir /dir /dir/s  
# echo 'mundo' > /dir/s/y
```

A cuantos inodos y bloques accede el siguiente comando

```
# cat /dir/s/y
```

```
Lee inodo /, es de tipo DIRECTORIO  
Lee bloque del directorio /  
Lee inodo dir, es de tipo DIRECTORIO  
Lee bloque del directorio dir  
Lee inodo s, es de tipo DIRECTORIO  
Lee bloque del directorio s  
Lee inodo y, es de tipo REGULAR  
Lee bloque de datos de y
```

Cat si accede a los datos del archivo

```
4 inodos  
4 bloques
```

Se prepara el sistema con los siguientes comandos:

```
# ln /dir/x /dir/h  
# rm /dir/x
```

A cuantos inodos y bloques accede el siguiente comando

```
# cat /dir/h
```

```

Lee inodo /, es de tipo DIRECTORIO
Lee bloque del directorio /
Lee inodo dir, es de tipo DIRECTORIO
Lee bloque del directorio dir
Lee inodo h, es de tipo REGULAR
Lee bloque de datos de h

3 inodos
3 bloques

No importa que se ejecute rm /dir/x y borre el original porque era un
hard link

```

Se prepara el sistema con los siguientes comandos:

```
# ln -s /dir/s/y /dir/y
```

A cuantos inodos y bloques accede el siguiente comando

```
# cat /dir/y
```

```

Lee inodo /, es de tipo DIRECTORIO
Lee bloque del directorio /
Lee inodo dir, es de tipo DIRECTORIO
Lee bloque del directorio dir
Lee inodo y, es de tipo SYMBOLIC LINK
Lee bloque de y, obtiene la ruta del link: /dir/s/y

Lee inodo /, es de tipo DIRECTORIO
Lee bloque del directorio /
Lee inodo dir, es de tipo DIRECTORIO
Lee bloque del directorio dir
Lee inodo s, es de tipo DIRECTORIO
Lee bloque del directorio s
Lee inodo y, es de tipo REGULAR
Lee bloque de datos de y

```

```
Cat si accede a los datos del archivo
```

```
7 inodos
7 bloques
```

Dirent.h: struct dirent

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

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

```

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

```

Implementación de un Filesystem

Very Simple File System

Versión simplificada de un típico sistema de archivos unix-like. Existen diferentes sistemas de archivos y cada uno tiene ventajas y desventajas. Para pensar en un file system hay que comprender dos conceptos fundamentales:

- El primero es la estructura de datos de un sistema de archivos. En otras palabras, cómo se guarda la información en el disco para organizar los datos y metadatos de los archivos. El sistema de archivos vsfs emplea una simple estructura, que parece un arreglo de bloques.
- El segundo aspecto es el método de acceso, como se manejan las llamadas hechas por los procesos, como open(), read(), write(), etc. en la estructura del sistema de archivos.

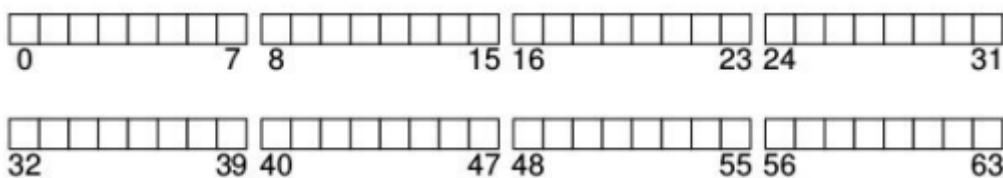
EL OBJETIVO PRINCIPAL DEL DISEÑO DE UN FILESYSTEM ES PODER CONSTRUIR LA ABSTRACCIÓN DE “ARCHIVO” SOBRE UN MEDIO DE ALMACENAMIENTO BASADO EN BLOQUES.

Por eso de alguna u otra forma, todos los filesystems contienen lo siguiente:

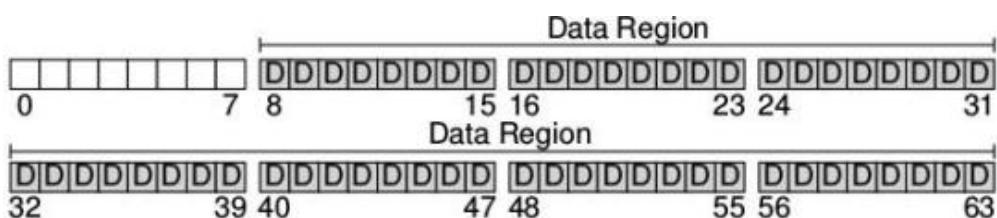
- La estructura de índice de un archivo proporciona una forma de localizar cada bloque del archivo. Las estructuras de índice suelen ser algún tipo de árbol para lograr escalabilidad y soportar la localidad.
- El mapa de espacio libre de un sistema de archivos proporciona una forma de asignar bloques libres para expandir un archivo.

Organización General

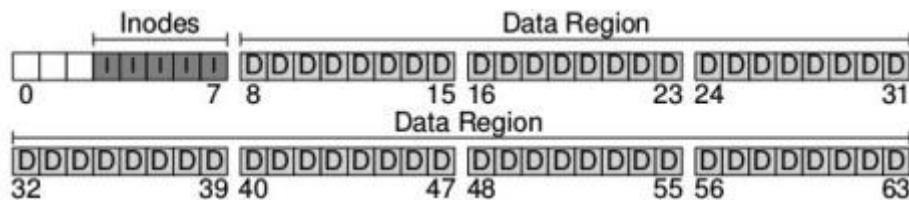
- ◆ Lo primero que se debe hacer es dividir al disco en bloques, los sistemas de archivos simples, como este suelen tener bloques de un solo tamaño. Los bloques tienen un tamaño de 4 kBytes.
- ◆ La visión del sistema de archivos debe ser la de una partición de N bloques (de 0 a N-1) de un tamaño de $N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



- ◆ A la hora de armar un sistema de archivos una de las cosas que es necesario almacenar es por supuesto que 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.
- ◆ Otra vez en nuestro pequeño disco es ocupado por ejemplo por 56 bloques de datos de los 64:



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 (Ojo son 5 bloques con muchos inodos cada uno):



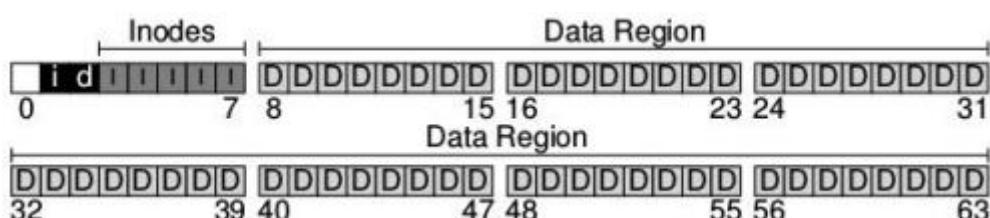
Cabe destacar que los inodos no son estructuras muy grandes, normalmente ocupan unos 128 o 256 bytes. Suponiendo que los inodos ocupan 256 bytes, un bloque de 4KB puede guardar 16 inodos por ende nuestro sistema de archivo tendrá como máximo 80 inodos. Esto representa también la cantidad máxima de archivos que podrá contener nuestro sistema de archivos.

ACLARACION

Aquí tenemos que hacer una aclaración sobre el ejemplo presentado, que esta extraido de [Arpaci] y sobre el cual en nuestra cátedra diferimos. En el ejemplo se puede ver que el autor toma 5 bloques de inodos donde cada bloque puede contener 16 inodos. En este sistema entonces habrá un máximo de 80 inodos posibles. Un inodo se asocia a un (y solo un) archivo. Y el archivo ocupará como mínimo un bloque (no se pueden compartir bloques entre archivos). Si partimos de un disco que tiene 64 bloques, diseñar para 80 archivos es un desperdicio. Con 4 bloques de inodos hubiera sido suficiente. Mas aun, como algunos de esos bloques se usarán no para archivos sino para superbloque, ibitmap, bbitmap y inodos, habrá menor cantidad de archivos posibles (específicamente 56).

El sistema de archivo tiene los datos (D) y los inodos (I) pero todavía nos falta. Una de las cosas que faltan es saber cuáles inodos y cuáles bloques están siendo utilizados o están libres. Esta estructura de alojamiento 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 otra 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 caso i sería el bitmap de inodos y d sería el bitmap de datos:



Se podrá observar que 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:

1. cantidad inodos
2. cantidad de bloques
3. donde comienza la tabla de inodos → bloque 3
4. donde comienzan los bitmaps

Esta es una de las estructuras almacenadas en el disco más importantes. Casi todos los sistemas de archivos unix-like son así. Su nombre probablemente provenga de los viejos sistemas UNIX en los que estos se almacenaban en un arreglo, y este arreglo estaba indexado de forma de cómo acceder a un inodo en particular. 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 qué parte del disco se encuentra el inodo correspondiente

The Inode Table (Closeup)										
			iblock 0	iblock 1	iblock 2	iblock 3	iblock 4			
Super	i-bmap	d-bmap	0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 64 65 66 67	4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 68 69 70 71	8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 72 73 74 75	12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63 76 77 78 79				
0KB	4KB	8KB	12KB	16KB	20KB	24KB	28KB	32KB		

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

1. debe calcular el offset en la región de inodos $32 * \text{sizeof(inode)} = 8192$
2. sumarlo a la dirección inicial de la inode table en el disco o sea $12\text{Kb} + 8192$ bytes
3. llegar a la dirección, en el disco, deseada que es la 20 KB.

Ejercicio de parcial (Superbloque, inodos, bloques)

Diseña un vsfs para un sistema de 1024 bloques, cada bloque 4kb, cada inodo 256 bytes.

Criterio de la catedra. Asumir que en un disco de N bloques entran N archivos*

Independientemente de los bloques especiales (además cuando $N \rightarrow \infty$, la suposición se approxima)

Resolución:

- Cantidad de inodos necesarios: 1024
- Cantidad de inodos por bloque: $4096/256 = 16$
- Cantidad de bloques de inodos: $1024/16 = 64$
- Cantidad de los bitmap: $4096*8=32768$ (sobra! El disco tiene 1024 bloques, y 1024 inodos)

* se puede calcular un óptimo de la cantidad máxima de inodos, que será generalmente menor a la cantidad de bloques. Hacerlo como ejercicio. Pero a menos que lo pidamos explícitamente, el criterio en rojo es lo que vale en los exámenes (además es más fácil)

Solución:

- 1 superbloque
- 1 bloque de bitmap de inodos
- 1 bloque de bitmap de bloques
- 64 bloques de inodos
- $1024 - 64 - 1 - 1 = 957$ bloques de datos

Técnicas de mejora de rendimiento

Las siguientes secciones destacan técnicas para mejorar el rendimiento y la integridad de datos en sistemas de archivos, especialmente en condiciones de fallos y acceso concurrente.

Estrategias de Sincronización y Caché:

- Buffer Cache: Coordina el acceso a bloques de disco, manteniendo solo una copia en memoria y almacenando bloques frecuentemente usados para evitar accesos directos al disco, optimizando así la velocidad.
- Page Cache en Linux: Permite almacenar en RAM datos leídos/escritos en disco, acelerando el acceso a archivos y reemplazando al buffer cache en Linux.

Primitivas de Sincronización:

- fsync() y fdatasync(): Garantizan que datos y metadatos se escriban en el disco inmediatamente, evitando pérdidas en caso de fallas.
- mmap(): Mapea archivos en el espacio de direcciones del proceso, lo que permite manipular datos como si fueran memoria, y msync() asegura la persistencia de cambios realizados con mmap().

Caso de Estudio: xv6:

- xv6 utiliza Buffer Cache con política de reemplazo LRU, moviendo bloques al tope cuando se usan y liberando espacio al desalojar los menos recientes.
- La estructura de capas del sistema de archivos en xv6 coordina el acceso al disco mediante el buffer cache.

Manejo de Fallas y Logging:

- En xv6, el Log permite implementar transacciones. Al registrarse en el log, se asegura que las operaciones se completen totalmente o no se hagan, lo cual es útil en caso de fallas de energía.
- El sistema ejecuta un proceso de recuperación que verifica el log para completar o descartar transacciones incompletas.

ANEXO 1: Iniciar al SO y al Kernel

El inicio del Sistema Operativo involucra el Booteo, la Carga del Kernel, y el Inicio de las Aplicaciones de Usuarios.

1. Booteo o Bootstrap

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 y generalmente depende del hardware de la computadora.

- Booteo 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.
- Booteo 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 (0x000000-0x003FF), 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 0xE05B.
- Booteo 3: La BIOS genera una interrupción 19 (INT 19) de la tabla de interrupciones la cual hace apuntar a CS:IP a 0xE6F2.
- Booteo 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.

2. Fase de 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()`.

- 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).
- 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 esté 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.

3. El proceso de inicio

El trabajo de Init es “conseguir que todo funcione como debe ser” una vez que el kernel está totalmente en funcionamiento. En esencia, establece y opera todo el espacio de usuario. Esto incluye:

- Comprobación y montaje de sistemas de archivos
- Puesta en marcha de los servicios de usuario necesarios y, en última instancia, cambiar al entorno de usuario cuando el inicio del sistema se ha completado.

ANEXO 2: Arquitecturas x86 y risc-V

RISC-V

- El objetivo de los arquitectos de RISC-V es que sea EFICAZ para todos los dispositivos de cómputo, desde los más pequeños hasta los más rápidos.
- Enfatiza SIMPLICIDAD para mantener el COSTO BAJO y MUCHOS REGISTROS y VELOCIDAD de EJECUCION TRANSPARENTE para ayudar a compiladores y programadores a resolver problemas importantes de manera eficiente.

Registros de propósito general

- RISC-V define 32 registros de 32 bits (x0 a x31) en su implementación estándar de 32 bits (RV32I). En implementaciones de 64 bits (RV64I) y 128 bits (RV128I), los registros se amplían a 64 bits y 128 bits respectivamente.
- El registro x0 es un registro constante que siempre contiene el valor 0.
- Los registros x1 a x31 pueden usarse para operaciones aritméticas, de almacenamiento y para guardar direcciones de memoria.

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6-7	t1-2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-11	a0-1	Function arguments/return values
x12-17	a2-7	Function arguments
x18-27	s2-11	Saved registers
x28-31	t3-6	Temporaries

Modularidad y Extensiones

- Conjunto de Instrucciones Base (I): Es el conjunto mínimo de instrucciones necesarias para la mayoría de las aplicaciones, que incluye operaciones aritméticas, lógicas, de control de flujo, y de acceso a memoria. OBLIGATORIO en todas las implementaciones y define el núcleo funcional de RISC-V.
- Extensiones Opcionales:
 - M (Multiplicación y División): Añade instrucciones para multiplicación y división enteras.
 - A (Operaciones Atómicas): Proporciona soporte para operaciones atómicas necesarias en programación concurrente.
 - F (Punto Flotante Simple) y D (Punto Flotante Doble): Añaden soporte para operaciones en punto flotante de 32 y 64 bits respectivamente.
 - C (Comprensión de Instrucciones): Introduce un conjunto de instrucciones de 16 bits para reducir el tamaño del código.
 - V (Vectorial): Permite operaciones vectoriales para procesamiento de datos en paralelo.
 - B (Bit-Manipulation): Incluye instrucciones avanzadas para la manipulación de bits.

Direccionamiento de Memoria

- Espacio de Direcciones:
 - RV32: Usa direcciones de 32 bits, permitiendo un espacio de direcciones de hasta 4 GB.
 - RV64: Usa direcciones de 64 bits, permitiendo un espacio de direcciones de hasta 16 exabytes.
 - RV128: (Propuesta) Usa direcciones de 128 bits, para un espacio de direcciones extremadamente grande, aunque aún es experimental.
- Modos de Dirección:
 - Base+Offset: Direccionamiento sencillo basado en una base y un desplazamiento.
 - Inmediato: Se utiliza un valor inmediato (constante) dentro de la instrucción para operar directamente.

Juegos de Instrucciones

- Instrucciones Aritméticas: ADD, SUB, MUL, DIV, REM (resto de la división).
- Instrucciones Lógicas: AND, OR, XOR, NOT, SLL (desplazamiento lógico a la izquierda), SRL (desplazamiento lógico a la derecha), SRA (desplazamiento aritmético a la derecha).
- Instrucciones de Control de Flujo: JAL (salto y enlace), JALR (salto y enlace a registro), BEQ, BNE, BLT, BGE (instrucciones de comparación y salto condicional).
- Instrucciones de Acceso a Memoria: LW (load word), SW (store word), LB, SB (load/store byte), con soporte para diferentes tamaños de datos.

Virtualización de Memoria

- Paginación y Segmentación:
 - RISC-V soporta la paginación de memoria con tamaños de página configurables (4 KB, 2 MB, 1 GB).
 - El ISA define modos de paginación como Sv32 (32 bits), Sv39 y Sv48 (para 64 bits), que gestionan el espacio de direcciones virtual y permiten la traducción a direcciones físicas.
- Tabla de Páginas: Utiliza una estructura jerárquica de tablas de páginas para traducir direcciones virtuales a físicas, permitiendo un acceso eficiente y seguro a la memoria.

Interrupciones y Excepciones

- Manejo de Excepciones: RISC-V define un conjunto de vectores de excepciones que manejan situaciones como fallos de página, instrucciones ilegales, y desbordamientos.
- CSR (Control and Status Registers): RISC-V usa CSRs para gestionar interrupciones, excepciones y el estado del sistema, permitiendo un control detallado y seguro.

X86

“Las instrucciones CISC complejas del ISA x86 son descompuestas internamente en una serie de micro operaciones más simples (micro-ops) que se asemejan a las instrucciones RISC. Cada microoperación es más sencilla y se puede ejecutar más rápidamente dentro del procesador.”

Registros de propósito general: Todos 32 bits

- EAX, EBX, ECX, EDX: Son de propósito general, utilizados para operaciones aritméticas, lógicas y de almacenamiento temporal. Cada uno se puede subdividir en partes de 16 bits (AX, BX, CX, DX) y a su vez en 8 bits (AL, AH, BL, BH, etc.).
- ESI, EDI: Usados para operaciones de apuntado y manipulación de cadenas.
- EBP (Base pointer): Utilizado para apuntar al inicio del marco de la pila de funciones y subrutinas.
- ESP (Stack pointer): Apunta a la parte superior de la pila. Fundamental para operaciones de gestión de la misma, como llamada a funciones y manejo de interrupciones.
- EIP (Instruction pointer): Contiene la dirección de la próxima interrupción a ejecutar.

Segmentos de memoria:

- CS (Code segment): Contiene instrucciones del programa.
- DS (Data segment): Contiene datos del programa.
- SS (Stack segment): Contiene la pila del programa.
- ES, FS, GS: Adicionales que permiten el direccionamiento mas flexible en operaciones con memoria.

Modos de operación:

- Modo real: Compatibilidad con el x86 original de 16 bits, limitando las direcciones a 20 bits y proporcionando acceso directo al hardware.
- Modo protegido: Introducido con el 80386, es el modo nativo de 32 bits que permite acceso a 4 GB de memoria, protección de la misma, multitarea y control detallado del privilegio del software.
- Modo virtual 8086: Permite la ejecución de aplicaciones de 16 bits dentro de un entorno protegido de 32 bits, combinando la flexibilidad del modo protegido con la compatibilidad hacia atrás.

Juego de instrucciones:

- Aritméticas y Lógicas (ADD, SUB, MUL, DIV, AND, OR, XOR, NOT, etc.)
- Control de Flujo (JMP, JE, JNE, y LOOP, así como llamadas a funciones (CALL) y retornos (RET).)
- Manipulación de la Pila (PUSH, POP, PUSH, POPA)
- Movimiento de Datos (MOV, LEA, XCHG, etc.)
- Entrada/Salida (IN, OUT para la comunicación con dispositivos de hardware.)
- SIMD (Single Instruction, Multiple Data) (Conjunto de instrucciones MMX, SSE, y SSE2, que permiten la operación paralela sobre vectores de datos)

Memoria:

- Dirección de memoria

- Modo de Dirección Lineal: Las direcciones de 32 bits permiten acceso directo a un espacio de 4 GB de memoria.
- Segmentación y Paginación: El modo protegido usa segmentación para definir límites de memoria y paginación para gestionar la memoria virtual, permitiendo la creación de espacios de direcciones virtuales que se traducen a direcciones físicas.
- Paginación
 - Páginas de 4 KB: Estándar donde el espacio de direcciones se divide en bloques de este tamaño.
 - Páginas grandes: Opcionalmente se puede usar páginas de 4 MB para optimizar la gestión de memoria en ciertos escenarios.

Manejo de excepciones e interrupciones:

- Interrupt Descriptor Table (IDT) es una tabla que define manejadores para interrupciones y excepciones permitiendo un manejo estructurado de eventos del sistema, como fallos de página o interrupciones externas.

Convención de llamadas

→ Conjunto de REGLAS que le dicen a la computadora COMO PASAR INFORMACION (o "mensajes") entre DIFERENTES partes de un programa cuando necesita que hagan algo. Son las reglas que debe seguir una parte de un programa para comunicarse con otra parte de este.

Esos pasos son:

1. Pasar Argumentos
2. Recibir valor/res de retorno
3. Guardar Información Importante

→ Hay seis etapas generales al llamar una función [Patterson and Hennessy 2017]:

1. Poner los argumentos en algún lugar donde la función pueda acceder a ellos.
2. Saltar a la función.
3. Reservar el espacio de memoria requerido por la función, almacenando los registros que se requiera.
4. Realizar la tarea requerida de la función.
5. Poner el resultado de la función en un lugar accesible por el programa que invocó a la función, restaurando los registros y liberando la memoria.
6. Dado que una función puede ser llamada desde varias partes de un programa, retornar el control al punto de origen.