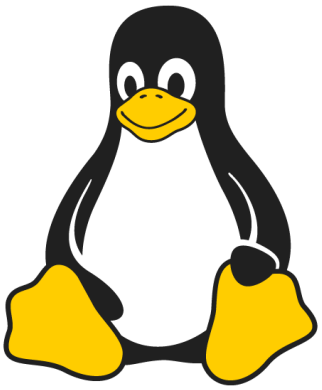
.UBAfiuba
FACULTAD DE INGENIERÍA



Apuntes de Sistemas Operativos

[7508] Sistemas operativos
Curso Méndez
1C 2021

Grassano, Bruno
bgrassano@fi.uba.ar

Índice

1. Introducción

El presente archivo contiene los apuntes que fueron tomados a lo largo de la cursada de la materia sistemas operativos (7508) en el curso del profesor Méndez. Esta formado por parte de los mismos apuntes del curso, de anotaciones tomadas durante la clase, y en algunas partes agregue notas de la bibliografía.

Recomendaciones

Llevarla al día.

Para las teóricas conviene ir ya con el tema leído desde el apunte de Mendez, complementa también tratar de seguir las clases con alguno de los libros de la bibliografía (Sección 11), personalmente fui leyendo el Arpaci, bastante claro y ordenado, los otros no tuve el tiempo de mirarlos durante la cursada.

Para las practicas, se tienen (en modalidad virtual) 2 labs que son unos trabajos prácticos individuales que se realizan al comienzo de la cursada, poniéndose un rato salen sin problemas. Dan 2 semanas para realizar cada uno. Después de estos, sigue el TP JOS del MIT dividido en 4 partes que se realiza en grupos de a 2. Para este prepárense para dedicarle un buen tiempo, en muchos casos es poco código pero muy especifico que requiere comprender muy bien lo que se esta haciendo (me acuerdo de un caso que estuvimos mas de una hora para hacer 2 lineas de código). Para cada parte lean bien las diferentes consignas, ya que aportan diferentes datos (la del curso, la original del MIT, y la del código).

Primera clase

2. ¿Que es un sistema operativo?

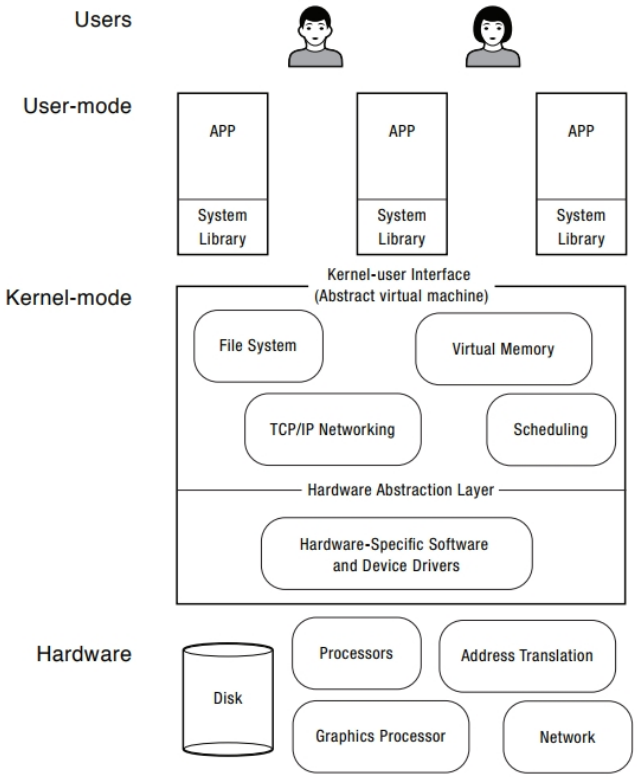
Es la capa de software que maneja los recursos de una computadora para sus usuarios y sus aplicaciones. Normalmente estos son invisibles a los ojos del usuario. (*Auto, Avión, Cámara, etc*)

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.

La principal forma para lograr esto es mediante el concepto de **virtualización**. Esto significa que el sistema operativo toma un recurso físico (*Ej. Memorias, procesadores, persistencia*) y lo transforma en algo mas general y fácil de usar.

Roles del OS

- **Referee**: El sistema operativo gestiona recursos compartidos entre diferentes aplicaciones. Estos están ejecutándose en la misma maquina física. *Acciones posibles: Frenar aplicaciones e iniciar otra, aislar aplicaciones de recursos, decidir que aplicación usa que recurso.*
- **Ilusionista**: Provee la ilusión de que se dispone de toda la memoria para almacenar el programa, cuando realmente la memoria principal es finita. Da la abstracción del hardware para simplificar el diseño de los programas.
- **Pegamento**: Debe proveer una serie de servicios comunes que faciliten un mecanismo para compartir, una API, un conjunto de funciones que parezcan que están diseñadas con el mismo criterio. *Ej. accesos a entrada/salida, ctrl+c/ctrl+v de manera uniforme a lo largo del sistema*



Modos de ejecución

Existen dos modos de ejecución.

- El sistema operativo se ejecuta como la capa **de software** de mas bajo nivel en la computadora. Este contiene una capa para la gestión de dispositivos, y por otro lado una serie de servicios para la gestión de dispositivos agnósticos del hardware. Estas capas son conocidas como el **kernel** del sistema operativo. Si parte del código fuente de esta capa es ejecutado, la computadora pasa a un estado llamado **Modo Supervisor**. Estas capas pueden utilizar primitivas que provee el hardware para el aislamiento de fallas y sincronización. El termino de kernel puede ser equivalente al de sistema operativo. Gracias a la existencia del kernel, los programas son independientes del hardware subyacente. Tener en cuenta que el kernel es un programa. No confundir esto con el modo *root* de linux ni correr con privilegios.
- Las aplicaciones mientras tanto, se ejecutan en un contexto **aislado, protegido y restringido**. Este contexto de ejecución se llama **User Mode** y mediante funciones de bibliotecas pueden utilizar los servicios de acceso al hardware o recursos que el kernel proporciona.

Perspectiva del usuario

El usuario lo que puede 'ver' son tres partes: el sistema de archivos, el entorno de procesamiento, y los bloques primitivos de construcción.

Sistema de archivos

En UNIX es una estructura jerárquica que permite el tratamiento consistente entre diferentes archivos. Permite crear, borrar, proteger o modificar archivos (que crezcan en tamaño de forma dinámica). Esta organizado como un árbol con una raíz (/). Cada nodo no hoja del sistema de archivos se denomina directorio de archivos, y las hojas del árbol pueden ser a su vez archivos, directorios, o archivos especiales. Cada uno de estos tiene un nombre dado por el camino (path) para localizarlo en la estructura.

Entorno de procesamiento

Un programa es un archivo ejecutable. Un proceso es una instancia de un programa en ejecución. Cada programa puede tener muchos procesos ejecutándose a la vez, no hay un limite lógico para la cantidad.

El primer programa en ser ejecutado es el interprete de comandos (shell) en UNIX en modo usuario (no es parte del kernel del sistema operativo). Este una vez cargado ejecuta el comando *login*.

Bloques primitivos de construcción

La idea de UNIX es la de proveer mecanismos para que los programadores construyan programas complejos a partir de programas mas pequeños. *Divide y conquista*

El proceso

El proceso es la parte fundamental de cualquier sistema operativo. Es una abstracción. Un programa en ejecución. El programa almacenado en disco contiene el código fuente compilado y los datos del mismo. A partir de estos los sistemas operativos permiten darle vida a los ejecutables, convirtiéndose en un proceso. Se puede ver como el espacio de memoria donde se aloja el programa que esta siendo ejecutado.

Segunda clase

3. El Kernel

Ejecución directa

La ejecución directa consiste en correr el programa directamente en la CPU. Esto otorga la ventaja de tener rapidez. Aunque también esto viene con algunos problemas. Estos son: ¿Como se asegura el OS que el programa no va a hacer nada que el usuario no quiere que sea hecho? ¿Como hace el OS para pausar la ejecución de ese programa y hacer que se ejecute otro?

Debido a esto se necesita limitar la ejecución directa. Hoy en día esto ya no existe en un sistema operativo serio.

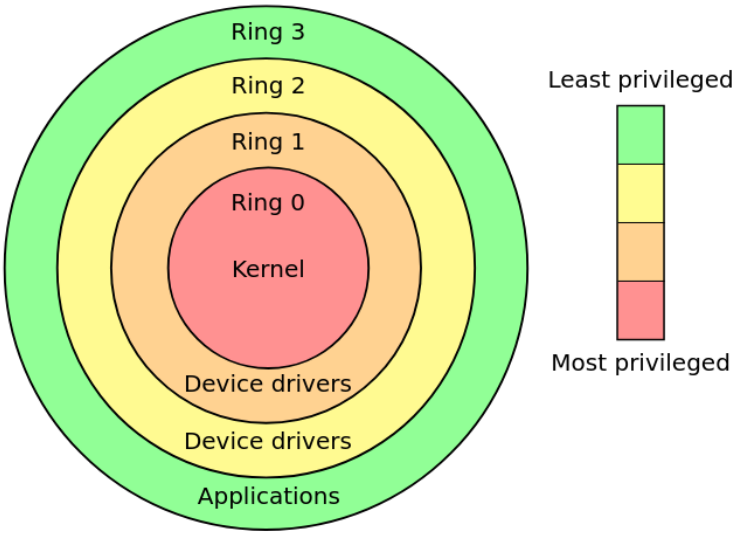
Limitar la ejecución directa

Para poder limitar la ejecución directa, son necesarios mecanismos del hardware.

- Dual Mode Operation - Modo de operación dual
- Privileged Instructions - Instrucciones Privilegiadas
- Memory Protection - Protección de Memoria
- Timer Interrupts - Interrupciones por temporizador

Dual Mode Operation

En las arquitecturas de x86 se tienen 4 modos de operaciones vía el hardware. Van del 0 a 3 y se denominan **rings**, siendo el ring 0 el nivel mas privilegiado (el del kernel - modo supervisor o modo kernel) y el 3 el menos (el de las aplicaciones - modo usuario). El hardware chequea en que modo de operación se esta ejecutando cada instrucción. La diferencia entre ambos modos esta en un bit en el registro de control del procesador. Tener esto permite proteger la memoria, los puertos de I/O y la posibilidad de ejecutar ciertas instrucciones.



Protección del sistema

- Que se tenga el modo dual permite que cada modo tenga su propio set de instrucciones. Con lo cual el bit de modo de operación indica al procesador si la instrucción ejecutada puede ser o no ejecutada, según el modo en que se encuentre. (Instrucciones privilegiadas) *Un ejemplo de estas instrucciones es LGDT*
- En el caso de la protección de la memoria, el sistema operativo debe poder configurar el hardware de forma tal de que cada proceso pueda leer y escribir su propia porción de memoria, ya que ambos, sistema y aplicaciones están en memoria al mismo tiempo.

Timer Interrupts

Cuando se ejecuta una aplicación, esta cree que tiene todo el tiempo para realizar lo que quiera, pero en realidad es una ilusión. El hardware proporciona al kernel un **hardware counter**, el cual una vez seteado, interrumpe el procesador luego de un determinado tiempo. Esto permite que el kernel desaloje el proceso del usuario y tome el control de la maquina.

El programa no se da cuenta que se despierta el kernel.

Con este sistema no hay forma de que un programa se quede con todo el procesamiento, ya que vuelve al kernel despues de un tiempo.

IOPL

El IOPL son los bits que se agregaron para poder identificar en que modo (ring) se esta. Se puede cambiar de privilegio con unas funciones, las cuales solo pueden se usadas desde el modo kernel (POPF(D)). Son los bits 12 y 13 del registro de flags.

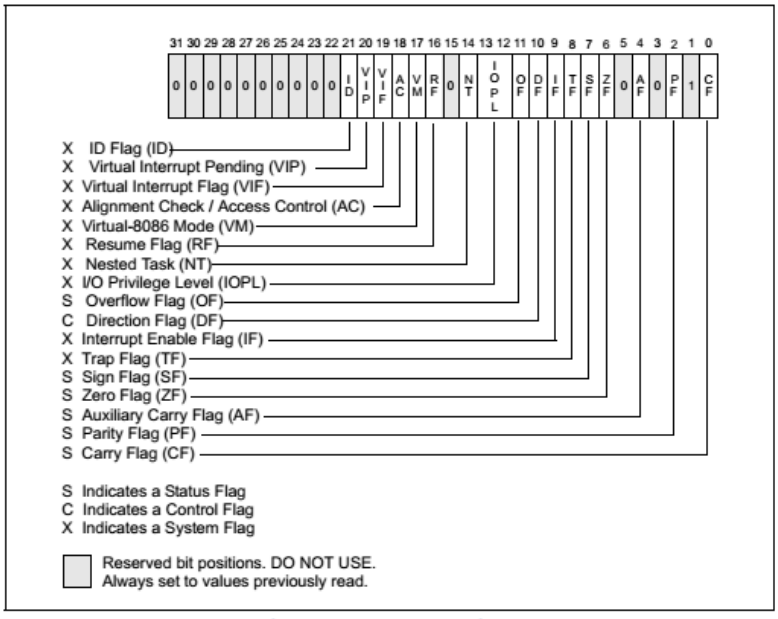


Figure 3-8. EFLAGS Register

Figura 1: Bits del registro de flags

Tareas del kernel

- Planificar la ejecución de las aplicaciones, toda aplicación que se ejecuta, la planifica el kernel. Hay un modulo especifico para esto, el scheduler. *El famoso Árbol rojo negro.*

- Gestionar la Memoria, asignar el recurso, decir quien puede y quien no usarla. También el como puede usarla.
- Proveer un sistema de archivos
- Creación y finalización de procesos
- Acceder a los dispositivos
- Comunicaciones
- Proveer un API para acceder al hardware

Transferencias

Existen tres formas para cambiar de modo usuario a modo kernel: con interrupciones (evento externo), con excepciones del procesador (evento interno), y mediante la ejecución de *system calls* (evento intencional). (6 en total, de usuario a kernel, y de kernel a usuario)

Interrupciones

Una interrupción es una señal asincrónica enviada hacia el procesador que indica que algún evento externo ha sucedido, causando que se pueda requerir de la atención del mismo. El CPU esta revisando continuamente si se dispara alguna interrupción. Si sucede alguna, completa o detiene cualquier instrucción que se este ejecutando, y en vez de ejecutar la siguiente instrucción, guarda el contexto, y comienza la ejecución del manejador de esa interrupción en el kernel.

Es externo al procesador.

Excepciones del procesador

Estas excepciones son causadas por el programa de un usuario. Sigue el mismo funcionamiento que el de una interrupción. *Ej. Acceder fuera de la memoria del proceso, intentar ejecutar una instrucción privilegiada en modo usuario, intentar escribir en memoria de solo lectura, dividir por cero.*

System Calls

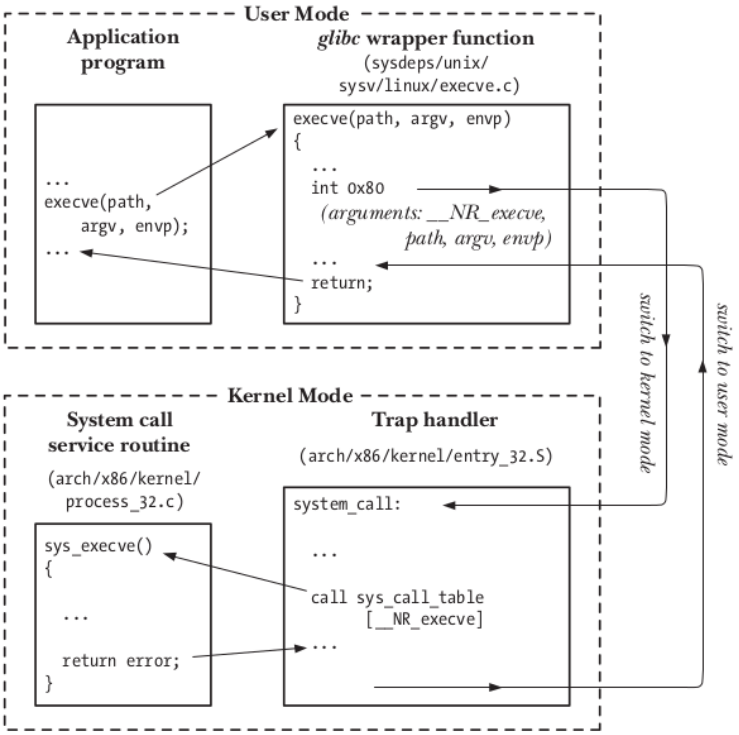
Las System Calls son funciones que permiten a los procesos del usuario pedirle al kernel que realice operaciones en su nombre (puntos de entrada controlados al kernel). Las system calls conforman una API.

El uso de estas llamadas cambia el modo del procesador de *user mode* a *kernel mode*. El conjunto que se ofrece de opciones es fijo. Cada system call esta identificada por un numero fijo que no es visible al programa, solamente se conoce su nombre.

La llamada a una system call es similar a la invocación a una función de C, aunque detrás hay varias diferencias. La función que se ve en C actúa como un *wrapper* (envoltorio). Esta función que envuelve a la system call proporciona todos los argumentos al **system call trap_handling**, los cuales son pasados por el stack (hay casos en que los espera en determinados registros, ahí son copiados por el wrapper también).

Al momento de ejecutarse, las system calls son identificadas por el kernel con un numero en un registro (%eax) y se ejecuta una instrucción de código maquina llamada **trap machine instruction** (0x80) (una interrupción) que causa el cambio a *kernel mode* y se ejecute el código del vector de traps el sistema. En respuesta a esto el kernel invoca a su propia función para manejar esa trap finalmente. Si se devuelve error, la función wrapper lo setea en *errno*.

Las syscalls son llamadas desde el modo usuario. (con el wrapper).



De Kernel a usuario

- Crear un nuevo proceso.
- Continuar después de una interrupción, excepción o system call. Una vez que el kernel manejo el pedido, continua con el proceso interrumpido. (cambiando a modo usuario)
- Cambio entre procesos, se ejecuta un proceso, salta la interrupción, cambio a otro debido a que el otro se estaba ejecutando mucho.

Linux

Linux no es un sistema operativo, es solo un kernel (Desarrollado por Linus). El resto de los componentes están hechos por terceros. *Ej. GNU*. Algunas características de este kernel son:

- No tiene acceso a la biblioteca estándar de C, ni a los encabezados.
- Esta codificado con GNU C.
- El kernel no se protege de si mismo.
- No puede realizar fácilmente operaciones de punto flotante.
- El kernel tiene una pila fija de tamaño pequeño, no es dinámica.
- Es un único programa ejecutándose en memoria. (Kernel monolítico)
- El kernel tiene interrupciones sincrónicas, es preentive (el kernel te desaloja) y admite SMP.

Tipos de kernel

- Kernel Monolítico: es un programa único (proceso) que se ejecuta continuamente en memoria intercambiándose con los procesos del usuario. *Ej. Linux*
- Micro Kernel: solo implementa funcionalidad básica (en Ring 0). 4000 lineas de código. *Ej. MINIX*

Inicio

El proceso de inicio comienza con el booteo, denominado bootstrap. Esta parte depende del hardware, ya que acá se realizan chequeos del mismo y se carga el bootloadler, que es el encargado de cargar el kernel del sistema operativo. Esto se divide en 3 partes.

1. Se carga el BIOS (Basic Input/Output System)
2. Se crea la Interrupt Vector Table, y carga las rutinas de manejo de interrupciones en Modo Real.
3. El BIOS genera una interrupción (19), la cual hace ejecutar el servicio de interrupciones.

Luego del booteo, el BootLoader pasa a modo supervisor (posible por ser vía hardware), busca el kernel en el dispositivo de almacenamiento correspondiente y lo carga a memoria principal (esta como un archivo imagen, comprimido). Una vez hecho eso, setea el registro de la próxima instrucción y ejecuta la primer linea del kernel.

Una vez cargado y ejecutado el kernel, se cargan a memoria las aplicaciones que se deben ejecutar (shell), se setea la siguiente instrucción, y se pasa a modo usuario, dándole control a la aplicación.

Fase de inicio

La función de arranque para el kernel (intercambiador o proceso 0) establece la gestión de memoria (tablas de paginación y de memoria), detecta el tipo de CPU, funcionalidades adicionales (*ej. punto flotante*), y después cambia a funcionalidades del kernel no especificas de Linux a través de *start_kernel()*.

Esta función establece el manejo de interrupciones, configura memoria adicional, y comienza el proceso de inicialización. El primer proceso ejecutado por el sistema es *Init* (PID = 1).

El proceso Init se encarga de comprobar y montar los sistemas de archivos y poner en marcha los servicios de usuario necesarios y, en última instancia, cambiar al entorno de usuario cuando el inicio del sistema se ha completado.

Tercera y cuarta clase

4. Introducción x86

Ley de Moore

En el 1965 Gordon Moore formulo la ley empírica que se ha podido constatar hasta hoy en día. Esta dice: *'Aproximadamente cada dos años se duplica el número de transistores en un microprocesador por unidad de área'.*

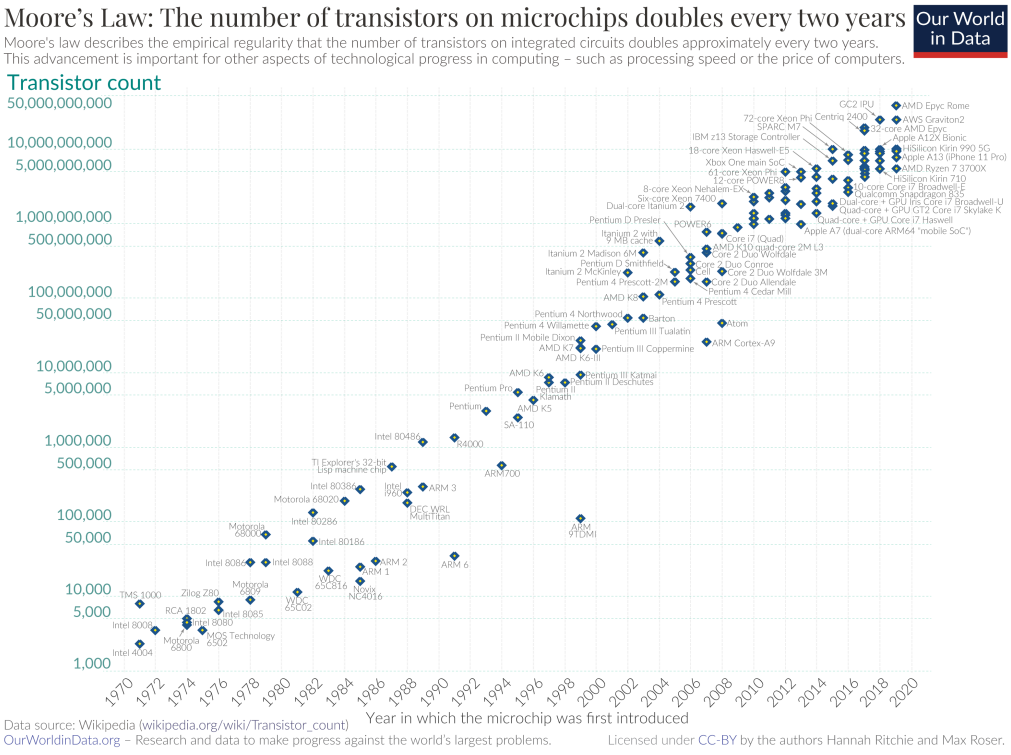


Figura 2: Evolución de la cantidad de transistores.

Ley de Bell

Esta ley establece que aproximadamente cada diez años una forma nueva de computadoras basadas en una nueva forma de programación, networking e interface se establece como nuevo uso de la industria.

Arquitectura x86: Hardware

El primer modelo de programación se denomino *program visible* debido a que sus registros utilizados durante la ejecución del programa son especificados por las instrucciones. *Ej. ADD dx,cs*

Existe otros tipos de registros que son *program invisible* ya que no son direccionables directamente en tiempo de ejecución del programa. Únicamente a partir del procesador 80286 existen registros program-invisible, utilizados exclusivamente para control y para operar con memoria protegida, entre otras cosas.

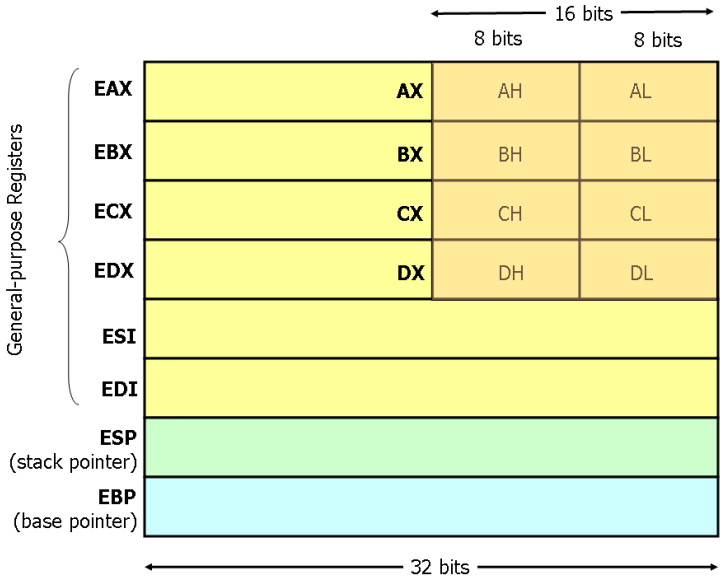


Figura 3: Parte de los registros de x86, tener en cuenta para el TP de JOS

El Stack

En la arquitectura x86 los programas utilizan el stack del programa para soporta la llamada a funciones. La maquina utiliza el stack para:

- pasar la información de los parámetros.
- almacenar la información de retorno.
- almacenar los valores de ciertos registros para su posterior utilización.
- para almacenamiento local.

La porción del stack para realizar esto se llama *stack frame*. La estructura del stack frame esta delimitada por 2 valores, el stack frame pointer, registro %EBP y el stack pointer, registro %ESP. El stack pointer puede moverse durante la ejecución de la función, toda la información es accedida en forma relativa con el stack frame pointer.

El conjunto de registros de un procesador es una fuente limitada de recursos que son compartidos por las distintas funciones que se están ejecutando. Cuando se ejecuta una función, hay que asegurarse que la función que llama (caller) a otra función (callee), esta ultima no sobre escriba todos los valores de los registros, ya que algunos de estos van a ser necesitados por la función llamadora. Por lo tanto se define una convención.

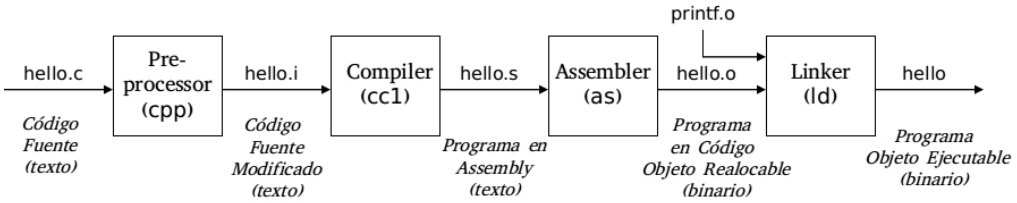
- Los registros %EAX, %EDX y %ECX son denominados Caller-save, es decir, cuando Q es llamado por P, el llamado puede sobre-escribir esos registros sin miedo de destruir datos de P.
- Los registros %EBX, %ESI y %EDI son denominados Callee-save, esto significa que Q debe guardar los valores de estos registros en la pila antes de sobrescribirlos y restaurarlos antes de retornar

5. El proceso

La compilación

Esta compuesta por 4 fases.

- La fase de procesamiento: El preprocesador (cpp) modifica el código fuente original del programa escrito en C comenzando con las directivas (#). El resultado es otro programa C (.i).
- La fase de compilación: El compilador (cc) traduce el programa .i a un archivo .s que tiene el programa en assembly.
- La fase de ensamblaje: El ensamblador traduce el .s en lenguaje de maquina y los almacena en un formato como programa objeto relocable (.o).
- La fase de link edición: Junta las bibliotecas utilizadas.



El programa en UNIX

Un programa es un archivo que tiene toda la información de como construir un proceso en memoria. Este contiene:

- Formato de identificación binaria: Es la META información que describe el formato ejecutable. Le permite al kernel interpretar la información contenida en el mismo archivo.
- Instrucciones de Lenguaje de Maquina: Almacena el código del algoritmo del programa.
- Dirección del punto de entrada del programa: Indica la dirección de la primera instrucción que el programa debe ejecutar.
- Datos: Son los valores con los que se deben inicializar variables, constantes, y literales.
- Símbolos y Tablas de Realocación: Describe la ubicación y los nombres de las funciones y variables del programa. Tiene otra información para debug también.
- Bibliotecas compartidas: Describe los nombres de las bibliotecas compartidas que son usadas durante la ejecución. También tiene la ruta del linker dinámico que debe ser usado.
- Otra información necesaria para terminar de construir el proceso en memoria.

Un programa es algo 'sin vida', un conjunto de instrucciones y datos que esperan en algún lugar del disco para saltar a la acción. El OS es quien lo toma y lo transforma en algo útil, mediante el Kernel.

El Kernel realiza lo siguiente.

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

El proceso

'Un proceso es la ejecución de un programa de aplicación *con derechos restringidos*; el proceso es *la abstracción* que provee el Kernel del sistema operativo para la ejecución protegida'- [DAH]

El con derechos restringidos nos dice que esta en una maquina con dual mode, y que se ejecuta en el ring 3. La abstracción equivale a la virtualización que se tiene, que provee el kernel.

Un proceso en realidad es mas que un simple programa en ejecución. Un proceso incluye muchas mas cosas.

- Los archivos abiertos
- Señales pendientes
- Datos internos del kernel
- El estado completo del procesador
- Un espacio en memoria
- Uno o mas hilos en ejecución, los cuales tienen su contador de programa, stack, y registros
- Datos globales

Leer: No silver Bullet – Essence and Accident in Software Engineering

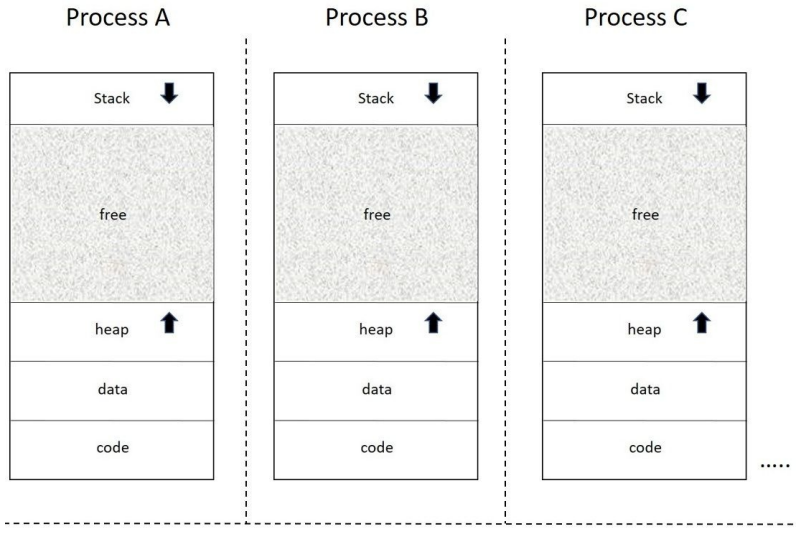
La virtualización

Es crear una abstracción que haga que un dispositivo de hardware sea mucho mas fácil de utilizar.

En los OS modernos se proporciona la virtualización de memoria y del procesador.

En memoria

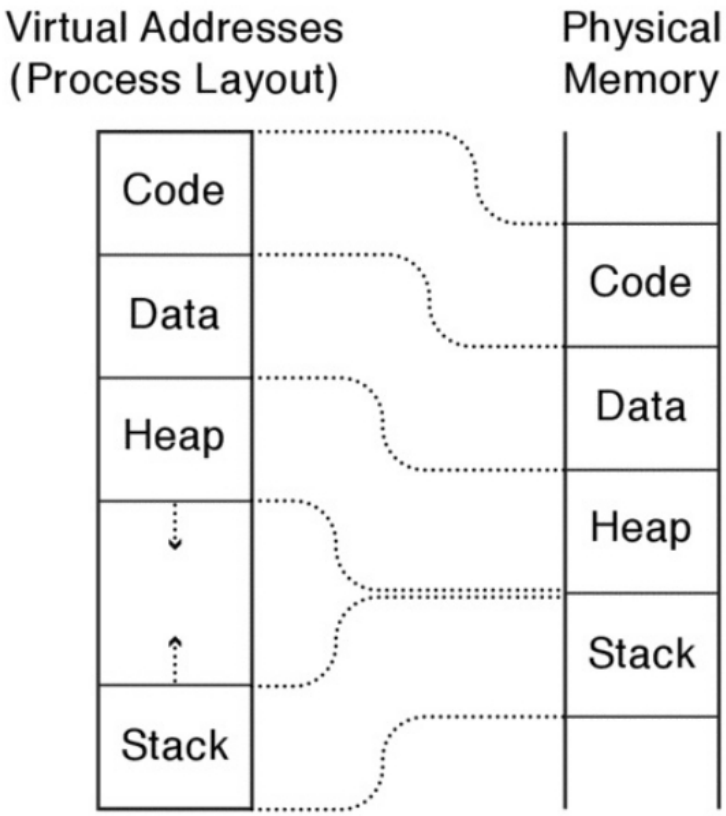
La virtualización de memoria le hace creer al proceso que este tiene toda la memoria disponible para ser reservada y usada como si este estuviera siendo ejecutado sólo en la computadora. Estos se dividen en texto (instrucciones del programa (.text o .code)), .data (variables globales), el heap, y el stack. Estas secciones se llaman espacio de direcciones del proceso. Están guardadas y el kernel sabe cuales son. Apenas el proceso quiere salir de esa sección de memoria, el kernel le tira una interrupción.



Para que un proceso se ejecute tiene que estar residente en memoria junto con el sistema operativo. Incluso pueden estar otros procesos simultáneamente. Para lograr esto, el OS tiene que configurar el hardware de forma tal de que cada proceso solo pueda ver y escribir lo suyo. El hardware provee mecanismos de protección para esto.

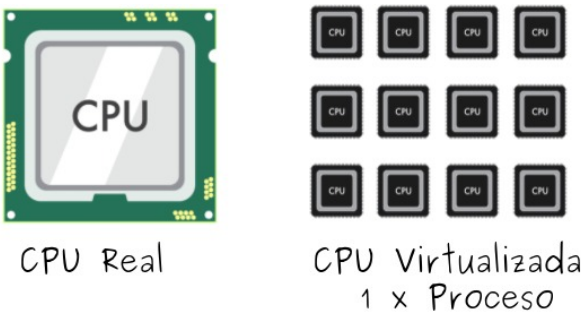
Uno de los mecanismos es la memoria virtual, esta es una abstracción por al cual la memoria física puede ser compartida por diversos procesos. El componente clave en este caso son las direcciones virtuales, para cada procesos su memoria empieza en el mismo lugar, la dirección 0. Esto hace que cada proceso piense que tiene toda la memoria para el.

El mapeo de una dirección virtual a una física ocurre a través del hardware con una Memory Managment Unit (MMU).



En CPU

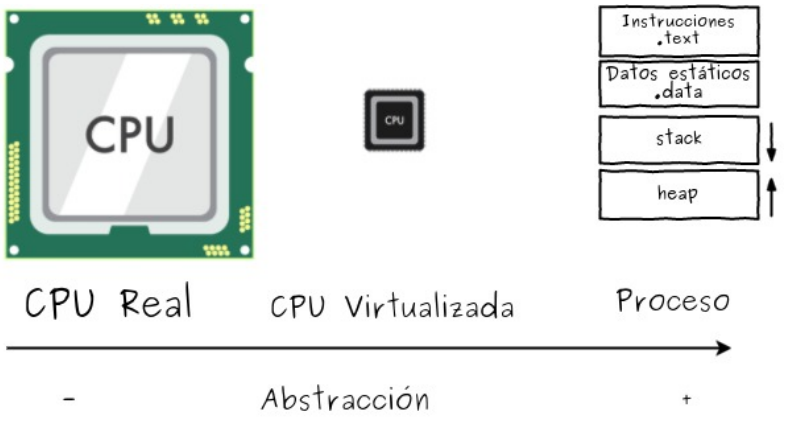
La virtualización del procesador es la forma de virtualización más primitiva, consiste en dar la ilusión de la existencia de un único procesador para cualquier programa que requiera de su uso.



Esto permite que cada proceso crea que tiene toda la CPU y que le pertenecen todos los dispositivos. También provee una ilusión en cuanto a que todos los dispositivos tiene el mismo nivel de interfaces y que son mas potentes para su uso.

Otra ventaja de este sistema es que otorga un gran aislamiento ante fallas, ya que los procesos no pueden afectar directamente a otros procesos, y los errores no colapsan toda la maquina.

Esta gran ilusión se crea mediante el kernel.



El proceso por dentro

La idea general detrás de la abstracción es la de cómo virtualizar una CPU o procesamiento, es decir cómo hacer para que un único procesador actúe como tal para varios programas que requieren ser ejecutados utilizando el mismo hardware.

Cualquier proceso necesita permisos del kernel para poder realizar cualquiera de las siguientes acciones.

- Acceder a memoria de otro proceso
- Escribir o leer en el disco
- Cambiar algún seteo del hardware.
- Enviar información a otro proceso.

API

- Create: fork(), un proceso en UNIX se crea siempre a partir de otro proceso.
- Destroy: exit(), kill()
- Wait: wait()
- Controles varios: exec(), sbrk() (aumenta el address space en malloc())
- Status: getpid(), getppid()

El contexto

Cada proceso tiene un contexto bien definido que tiene toda la información necesaria para describirse.

- User Address Space: text, data, stack, y heap

- Control Information: el kernel utiliza dos estructuras principales para mantener la información contenida: la *u area* y *proc*. Cada proceso tiene su propio kernel stack y mapas de traducción de direcciones.
- Credentials: son las ids del proceso asociado a el, de grupos y de usuario.
- Variables de entorno: strings variables heredadas del padre.
- Hardware context: es el contenido de los registros de propósito general, junto con algunos registros especiales del sistema (PC, SP, PWD, registros del manejo de memoria, y de la unida de punto flotante).

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

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

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

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

U Area y Proc

Estas dos estructuras dependen de la implementación misma del kernel.

La proc structure es una entrada en la process table. Contiene información a la cual se puede acceder cuando el proceso no se esta ejecutando. Contiene el PID, ubicacion del mapa de direcciones del kernel del u area del proceso, el estado actual, prioridad, información para el manejo de señales. Es conocido también como *task structure* en Linux.

La U area (user area) es parte del espacio del proceso. Esta es vista solo por el proceso cuando esta siendo ejecutado. Contiene Process Control Block (guarda el hardware context cuando el proceso no se ejecuta), un puntero a la proc structure, argumentos para, y retorno o errores hacia la system call actual, manejadores de señales, información de las areas de memoria, tabla de open fd.

fork

Crea un nuevo proceso. Para esto se realizan varias acciones.

- Se chequea que hay recursos.
- Obtiene una entrada en la Process Table (PID unico).
- Verifica que no se ejecuten demasiados procesos.
- Cambia el estado del proceso hijo a 'siendo creado'. Permite que el scheduler no le haga nada, ya que lo esta creando el kernel.
- Copia datos de la process table del padre al hijo.
- Aumenta contadores (directory inode, file table).
- Hace una copia del contexto del padre en memoria.
- Crea un contexto a nivel sistema falso para el hijo. (para que el hijo se reconozca y tenga un punto de inicio)

Ambos procesos ocurren de forma concurrente, haciendo que no haya determinismo en el orden de ejecución. (Están vivos al mismo tiempo)
Ejemplo del Arpaci.

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

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {// fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

exit

Esta system call realiza:

- Ignora todas la signals
- Cierra todos los archivos abiertos

- Se liberan los locks sobre los archivos
- Libera el directorio actual
- Se separan segmentos de memoria compartida
- Se actualizan contadores
- Libera memoria
- Pone el estado del proceso en modo zombie, asignándole como proceso padre a init, casi toda la información del proceso murió, queda algo de meta-data solamente.
- Manda señal de muerte al proceso padre
- Context switch

Estados

De forma simple se puede mostrar como sigue, en realidad hay mas estados posibles.

- Running
- Ready
- Blocked

El que hace que pase de ready a running es el planificador.

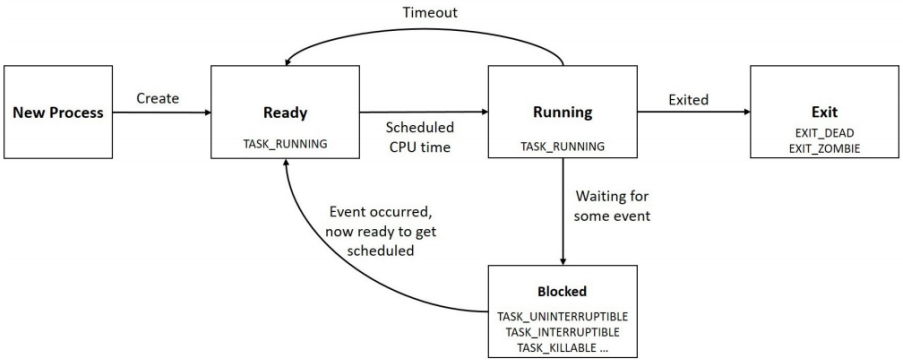


Figura 4: Estados en los que puede estar un proceso en Linux

Quinta y sexta clase

6. La memoria

El tema principal sobre la administración de la memoria se encuentra en como representarla.

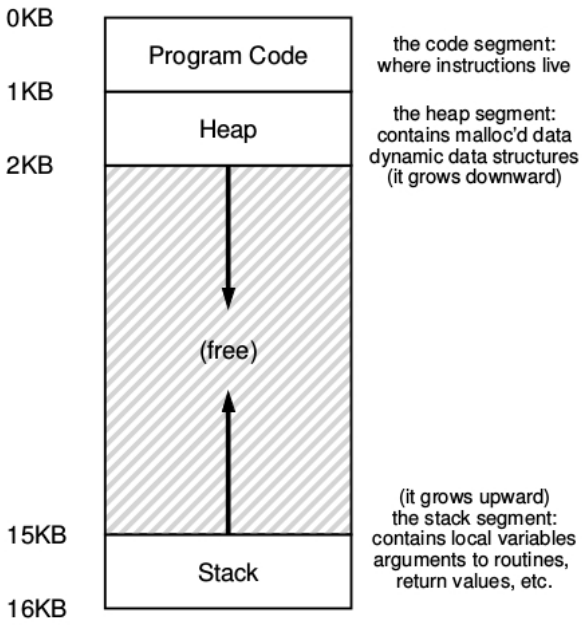
La memoria física se imagina como un arreglo de direcciones de memoria.

Esto resulta manejable mientras la cantidad de memoria este en un rango adecuado, dado por la arquitectura.

Address Space

Crear un mecanismo que permita que la memoria física de una computadora sea utilizada de forma fácil y eficiente llevo 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.



Cuando se esta describiendo así el espacio de direcciones, se esta describiendo la abstracción que provee el sistema operativo, ya que el programa piensa que esta cargado a partir de la dirección 0 y que tiene toda la memoria para el.

Cuando por ejemplo un proceso trata de cargar el contenido de la dirección 0 (la dirección virtual 0), de alguna forma el sistema operativo con ayuda de algún tipo de mecanismo de hardware que tendrá que asegurarse que no se cargue la dirección física 0 real más bien que se cargue la dirección física en la cual el espacio de direcciones de ese proceso se encuentra alojado.

Esta implementación de la memoria virtual esta hecha de forma tal de que sea invisible al programa que se ejecuta, pero detrás de escena, el OS y el hardware llevan a cabo la ilusión.

Otras cuestiones importantes de la implementación son el tiempo (que los programas no sean mas lentos) y el espacio (no usar demasiada memoria para las estructuras de la memoria virtual), la memoria virtual debe ser eficiente.

La ultima cuestión que debe de tener es la de la protección. El OS debe de asegurarse de proteger a los procesos entre si, y de protegerse a si mismo. Esto habilita el aislamiento entre procesos.

La API

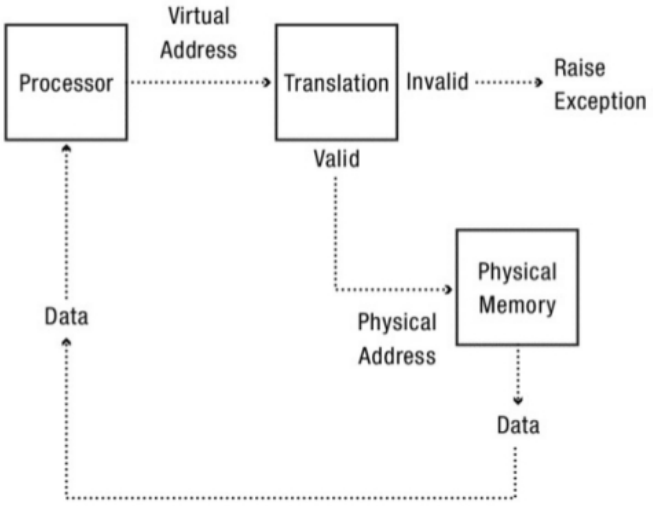
En un programa en ejecución existen 2 tipos distintos de memoria, el heap y el stack.

- La del stack es reservada y liberada implícitamente por el compilador. Su declaración es a través de la declaración de variables.
- La del heap se obtiene y libera explícitamente por el programador. *Ej. malloc() y free()* Estas funciones internamente hacen llamadas a las syscalls *brk() y sbrk()*

Traducción

Se hace a través de una técnica llamada Hardware-Based Address Translation, o directamente Address Translation.

Esta técnica transforma la dirección virtual en una física. El hardware provee un mecanismo eficiente de bajo nivel, y el sistema operativo setea al hardware de la forma correcta, gerencia la memoria manteniendo información de los lugares libres y en uso, y mantiene el control sobre la memoria usada.



Cuestiones que provee la traducción de las direcciones

- Aislamiento de procesos
- Comunicación entre procesos, si los procesos deben de coordinar-se entre si, una forma eficiente de hacerlo es que tengan una región compartida de memoria.
- Segmentos de código compartidos para instancias de un mismo programa, haciendo mas eficiente el uso de la cache.
- Inicialización de programas, se puede empezar un programa antes de que se haya cargado a memoria todo el código.
- Uso eficiente de la memoria de forma dinámica
- Administración de la cache, ubica los programas en memoria física de forma eficiente para que mejore la eficiencia del cache. Se hace con un sistema llamado *Page coloring*
- Debugging de programas, puede evitar que programas con bugs sobre escriban secciones de su propio código, atrapando el error de forma temprana.

- Uso eficiente de I/O
- Archivos mapeados a memoria
- Memoria virtual
- Puntos de control y reinicio, el estado de un programa puede ser guardado de forma periódica en caso de que se rompa el programa o sistema. Esto permite que se pueda reiniciar de ese punto guardado.
- Estructuras de datos persistentes
- Migración de procesos
- Control del flujo de la información, actúa como una capa de seguridad extra, para evitar que se envié información a donde no tiene que ir.
- Distribución de la memoria compartida

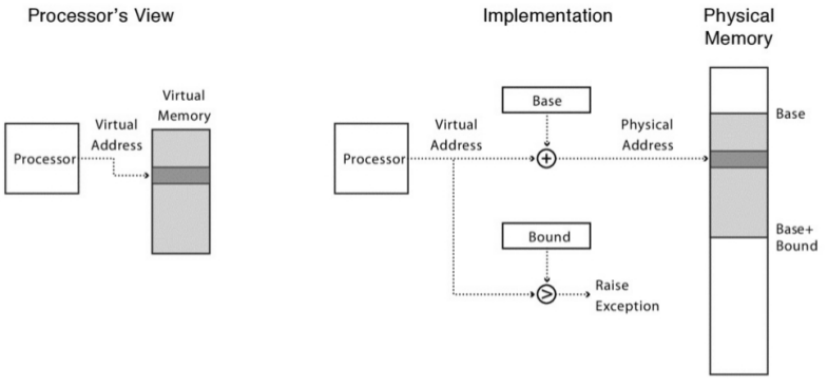
Dynamic Realocation o Memoria segmentada

Base and Bound

Se necesitan dos registros, el *base* y el *limite o segmento*. Estos registros permiten que el address space pueda ser ubicado en cualquier lugar de la memoria física.

$PhysicalAddress = VirtualAddress + base$

El registro bound (limite o segmento), es verificado antes de realizar alguna operación en memoria.



Esto forma parte de la MMU (Memory Managment Unit)

Tabla de segmentos

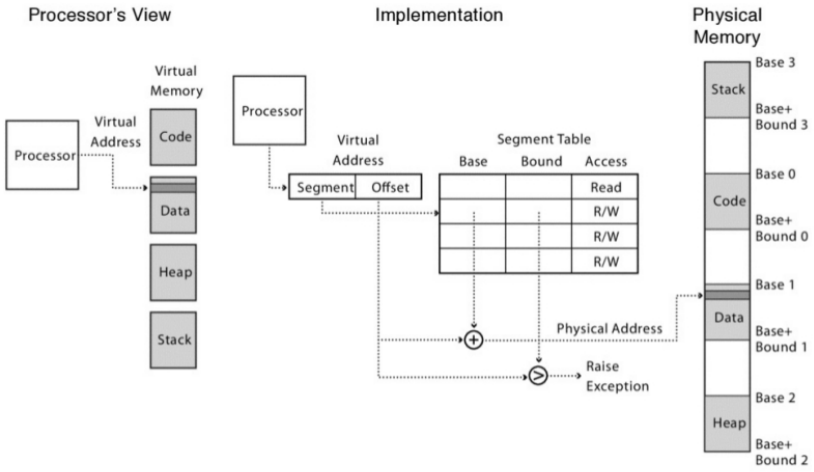
El problema de Base and Bound es que se tiene un solo registro base y un solo segmento. Se mejora teniendo un arreglo de pares base-segmento por cada proceso, donde cada entrada controla una parte del virtual address space.

Las direcciones virtuales tienen un numero de segmento (index) y un offset.

Se tienen bits de protección también, lectura, escritura, ejecución.

Segmentación de grano fino vs segmentación de grano grueso. La primera consiste en tener muchos segmentos pequeños y la segunda consiste en tener pocos segmentos grandes.

El problema de la segmentación es la fragmentación, precisamente la fragmentación externa.



Paginada

En este caso la memoria se reserva en pedazos de tamaño fijo llamados page frames. La traducción se realiza de forma similar al segmentada. (4KB generalmente)

Hay una tabla de paginas por cada proceso con punteros a las page frames. Como tienen un tamaño fijo, y son potencia de 2,las entradas en la page table sólo tienen que proveer los bits 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.

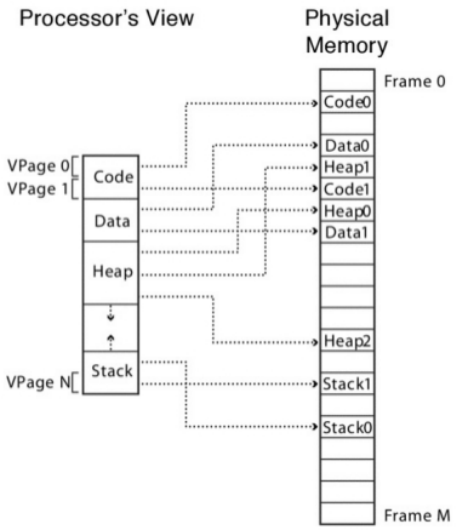


Figura 5: El modelo en la memoria.

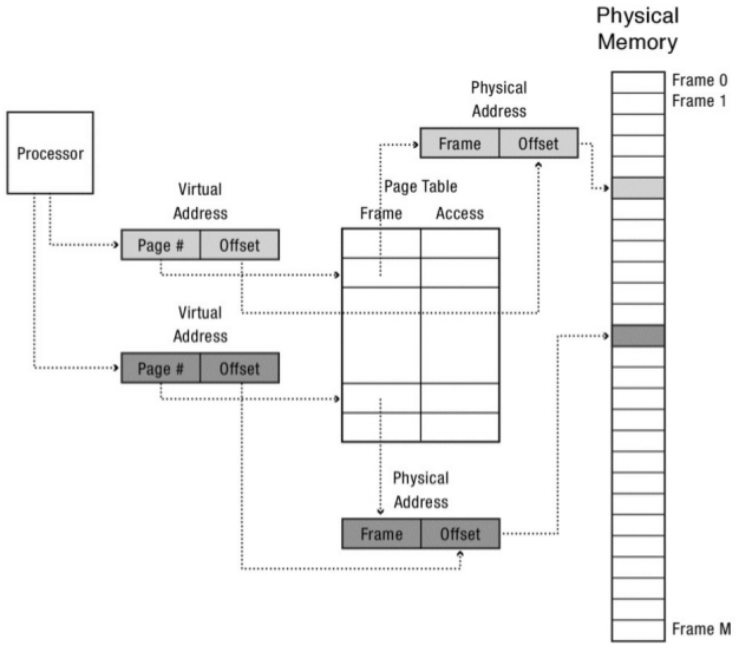


Figura 6: El proceso de traducción con memoria paginada.

Multinivel

Es un mecanismo mas eficiente de paginación, basado en un árbol o hash table. Proporcionan

- Protección de memoria de grano fino
- Memoria Compartida
- Ubicación de memoria flexible
- Reserva eficiente de memoria
- Un sistema de búsqueda de espacio de direcciones eficiente

Segmentación paginada (paged segmentation)

Este sistema usa dos niveles de un árbol. Con paged segmentation, la memoria esta segmentada, pero en vez de que cada entrada en la pagina de segmentos apunte directamente a una región continua de la memoria física, cada entrada en la tabla de segmentos apunta a una tabla de paginas, que a su vez apunta a la memoria correspondiente a ese segmento. La tabla de segmentos tiene una entrada llamada limite o tamaño que describe la longitud de la pagina de tablas, osea la longitud de los segmentos en las paginas

Existen también casos con mas niveles, y sistemas de hash.

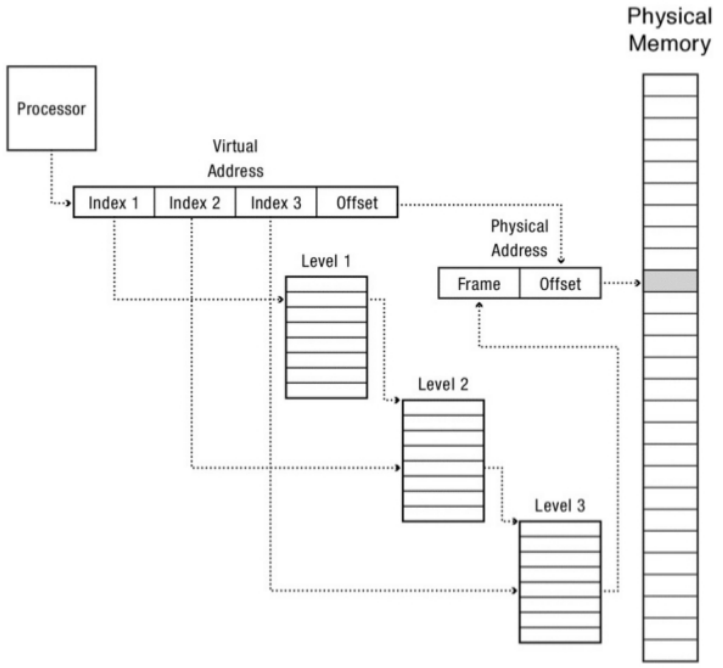


Figura 7: El proceso de traducción con tres niveles.

Mejora del rendimiento: TLB

Como se vio que había mejoras de localidad y temporalidad, se utiliza un cache llamado Translation-Lookaside Buffer el cual es parte de la MMU. Este sistema guarda los pares de direcciones virtuales a físicos utilizadas. Por cada referencia a memoria, el hardware se fija primero la TLB. Si esta guardada la referencia (TLB hit), va a memoria sin tener que realizar la traducción, resultando un proceso mas rápido. (ya que la traducción es ir a buscar a memoria la tabla, TLB miss)

Una cuestión a tener en cuenta es la consistencia con los datos originales cuando las entradas son modificadas.

En general tienen 32-64 entradas.

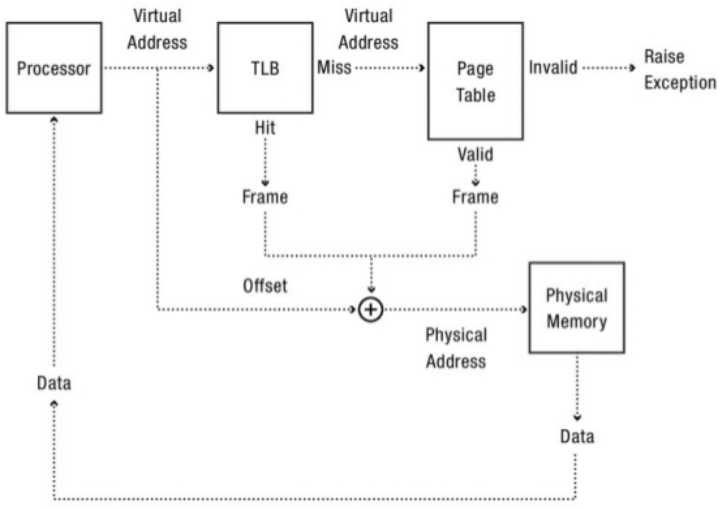
- El context switch: Las direcciones virtuales de un viejo proceso ya no son más válidas, y no deben ser válidas para el nuevo proceso. De otra forma, el nuevo proceso sería capaz de leer las direcciones del viejo proceso. Frente a un context switch, se necesita descartar el contenido de TLB en cada 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.

Funcionamiento

Extraer VPN (Virtual Page Number) de la VA, y chequear si la TLB tiene la traducción para esa VPN.

- Si la tiene es TLB HIT.
  1. Se extrae la PFN (Page Frame Number) de la entrada de la TLB.
  2. Se concatena el offset de la VA y se tiene la PA, con lo que se puede acceder a memoria.
- Si fue TLB MISS, haces un page table lookup.
  1. El hardware accede a la page table para encontrar la traducción.
  2. Si la referencia a la memoria virtual es valida y accesible, se actualiza la TLB con la traducción nueva.
  3. El hardware vuelve a buscar la instrucción en la TLB, y de ahí ir a la memoria.



Séptima y octava clase

Nota: Se ve después de memoria por que el TP1 consiste en la implementación de parte del sistema de memoria de JOS.

7. Scheduling o Planificación de Procesos

Cuando se tienen muchos programas, tiene que haber alguna forma de poder elegir cual se ejecuta primero o cuanto tiempo de CPU tienen. Este periodo se llama **time slice** o **time quantum**. El planificador tiene que tender a O(1), debe de ser lo mas eficiente posible. En el caso de Linux *tiende* a O(1).

Time Sharing

Surge de la idea de que los programadores tienen la computadora toda para el. Este sistema da la ilusión de que la computadora es toda tuya, pero no es así. Se comparte el tiempo de procesamiento.

Multiprogramación

Permite que si alguien esta haciendo E/S alguien mas haga computo.

Uso del CPU

Si un determinado programa que se estaba ejecutando debía realizar operaciones de entrada y salida de datos, debía interactuar con un dispositivo de entrada y salida. Esta interacción en términos de tiempos de ejecución de instrucciones del procesador podía parecer infinito. Además, con el agravante de que el tiempo de E/S solía ser en promedio de entre el 80 % y 90 % del tiempo total del programa, la CPU permanecía inactiva un gran período de tiempo.

Ej. Si se asume que el 20 % del tiempo de ejecución de un programa es solo cómputo y el 80 % son operaciones de entrada y salida, con tener 5 procesos en memoria se estaría utilizando el 100 % de la CPU.

El cálculo es más realista si se supone que un proceso gasta una fracción p, bloqueado en E/S. Por lo tanto, la probabilidad de que n procesos estén en E/S es p^n. Entonces, el uso del procesador queda definido como $1 - p^n$.

Multiple Fixed Partitions

Dividís en particiones la memoria, y los procesos se asignaban a esas particiones. En una se ejecutan los procesos grandes (partición grande) y en otras particiones chicas los procesos mas chicos. El problema esta en cuanto se acaba uno de los tipos, si tengo muchos procesos chicos tengo mucho espacio de memoria (de partición grande) desperdiciado.

Leer 'An Experimental Time-Sharing System' de Corbato.

Multiple variable Partitions

Cada tarea entra en memoria y ocupa un espacio. En este caso se puede realocar la memoria donde hay un espacio para aprovecharla mejor (reubica para reducir la fragmentación interna de la memoria). Esto elimina la fragmentación interna y externa.

Time Sharing

Cuando un Sistema Operativo se dice que realiza multi-programación de varios procesos debe existir una entidad que se encargue de coordinar la forma en que estos se ejecutan, el momento en que estos se ejecutan y el momento en el cual paran de ejecutarse. En un sistema operativo

esta tarea es realizada por el Planificador o Scheduler que forma parte del Kernel del Sistema Operativo.

Workload

Es la carga de trabajo de un proceso corriendo en el sistema. Normalmente a los procesos se les dice *jobs*.

Metricas

Para poder hacer algún tipo de análisis se necesitan métricas para poder comparar diferentes políticas de planificación.

Turnaround

Es el tiempo en que se completa un *job* menos el tiempo en que llega. Normalmente la unidad de tiempo serian nano segundos, para lo que vamos a necesitar la usamos sin unidad.

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

Mide performance, cuan rápido se va a ejecutar un proceso. Si Todos los procesos llegan al mismo tiempo, $T_{arrival} = 0$

Tiempo de respuesta

Nace con el Time Sharing, mide cuan rápido el usuario va a sentir que su proceso se esta ejecutando.

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

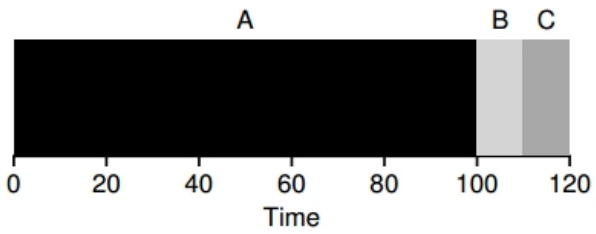
Surge debido a que los usuarios pretenden interacción con rapidez.

Políticas

Mientras mas inteligentes son, tienden a tardar mas. Hay que analizar el tradeoff entre la decisión y la la ejecución planificada.

First In, First Out - FIFO

El primero en llegar se ejecuta primero, es un cola. Los problemas que tiene este método es si llega primero un proceso que ocupa una mayor cantidad de tiempo en comparación a los otros. El T_{around} empeora considerablemente. Esto se conoce como *convoy effect*. Esta es una planificación *non-preemptive*, el kernel no puede desalojar un proceso (tiene todo el control del CPU).

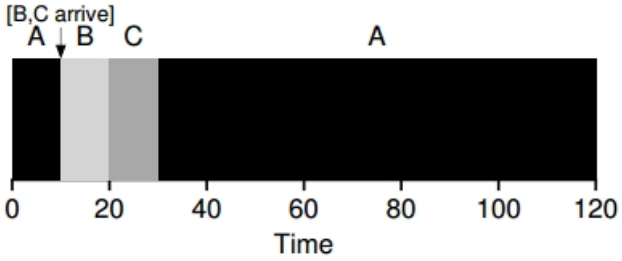


Shortest Job First - SJF

Busca resolver el problema de FIFO. Ejecuta primero el trabajo mas corto. El problema esta en que no se conoce de antemano cuanto va a durar un proceso y que todavía puede llegar un proceso largo sin que haya uno mas corto para ejecutar. También se le sumaria el tiempo que tarda en decidir cual proceso ejecutar. *non-preemptive*

Shortest Time-to-Completion - STCF

En este caso el planificador se adelanta y si hay un proceso que puede terminar antes, ejecuta ese primero, posponiendo la ejecución del que se estaba ejecutando. Esto lo realiza con un context switch. Es *pre-empitve*.



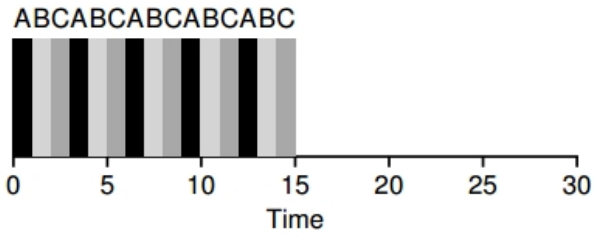
Round Robin - RR

El algoritmo consiste en ejecutar un proceso por un periodo de tiempo determinado (slice), y transcurrido pasar a otro proceso en la cola de ejecución. Trata a todos los procesos igual.

Es importante en este caso elegir un buen time slice. El time slice tiene que amortizar el cambio de contexto sin hacer que esto resulte en que el sistema no responda más.

Ej. 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.

Con este método de planificación, la métrica de *response* mejora, pero la de *turnaround* empeora, ya que se atrasa el retorno de todos los programas.



Multi-Level Feedback Queue - MLFQ

Este planificador intenta atacar principalmente 2 problemas.

- Intenta optimizar el turnaround time mediante la ejecución de la tarea mas corta primero. (No sabemos cual es)
- Intenta hacer que el sistema tenga un tiempo de respuesta interactivo para los usuarios. (Minimizar el response time)

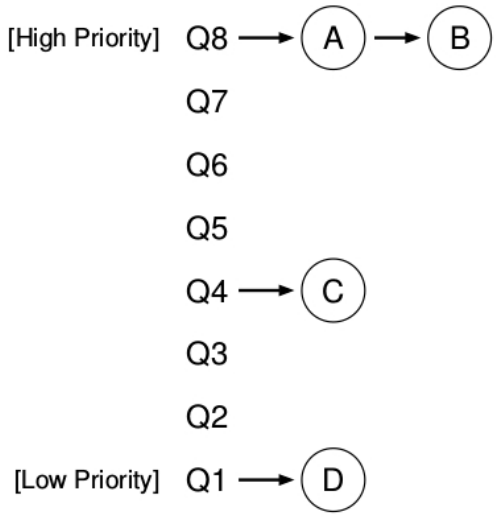
Para lograr esto, tiene un conjunto de distintas colas, cada una con un nivel de prioridad distinto.

En un determinado tiempo, una tarea que esta lista para ser ejecutada esta en una única cola. MLFQ usa las prioridades para decidir cual tarea debería correr en un determinado tiempo $t0$. Se puede elegir la tarea con mayor prioridad o la que esta en la cola mas alta.

La clave en este planificador esta en que no otorga prioridades fijas. Estas son variables con el tiempo.

Lo que queremos es que:

1. Si una tarea no utiliza el CPU (espera I/O), MLFQ le mantenga una prioridad alta, ya que vendría a ser un proceso interactivo. *Ej. Una tarea espera entrada del teclado.*
2. Si una tarea usa mucho el CPU, MLFQ le reduzca su prioridad. (No suelta el CPU) *Ej. Cualquier programa que haga mucho calculo.*



Posibles problemas

Hay que tener algunas consideraciones en la elección de prioridades.

- Starvation: Si hay demasiadas tareas interactivas, se van a combinar consumiendo todo el tiempo de la CPU. Esto evitaría que algún proceso que usa mucho la CPU se llegue siquiera a ejecutar.
- Un programador podría escribir su programa para que se ejecute mas de lo que debería. *Ej. Se escribe a un archivo que no contiene nada justo antes de que se acabe el time slice.*
- Los programas son dinámicos, puede pasar que un programa use mucho el CPU al comienzo y después pase a usar mas I/O, volviéndose mas interactivo.

Reglas

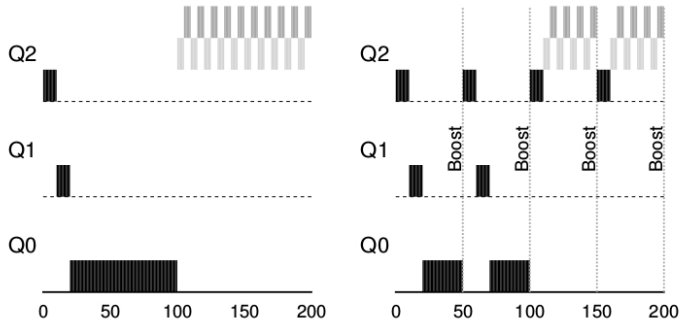
Las reglas que tiene MLFQ, siendo A y B dos procesos.

1. Si la prioridad de A es mayor que la de B. Se ejecuta A, B no se ejecuta.
2. Si la prioridad de A es igual a la de B, A y B se ejecutan en Round Robin.
3. Si entra un proceso nuevo, se le asigna la prioridad mas alta.

4. Una vez que una tarea usa su asignación de tiempo en un nivel dado, su prioridad se reduce independientemente de si renuncio al CPU. Esto evita que los programadores tomen ventaja del planificador.

5. Después de cierto periodo S, se mueven todas las tareas a la cola mas alta, se les da un boost de prioridad. Esto arregla el problema de Starvation y de los cambios de formas de ejecución de los programas.

La constante S es una VOO-DOO CONSTANT, requiere de magia negra para ser determinada correctamente. Si S es muy alta, los procesos que requieren mucha ejecución caen en Starvation, si es baja, las tareas interactivas no van a poder compartir correctamente el CPU. La obtención de S es estadístico mediante diferentes pruebas de ejecución.



Estas reglas permiten que MLFQ aprenda continuamente sobre los procesos para poder predecir su comportamiento. Hoy en día ya no se utiliza, son mas complejos los planificadores ahora. Se implemento en MULTICS.

Proportional Sharing

Este tipo de planificador cambia la forma en que encara el problema. En vez de optimizar turnaround o response time, intentara que cada tarea obtenga un porcentaje de uso del tiempo de CPU.

El concepto se conoce también como planificación por lotería. Cada cierto tiempo, se hace un sorteo para determinar quien se ejecuta a continuación. Los que deban de ejecutarse mas frecuentemente tienen que tener mas posibilidades de ganar la lotería.

El concepto que subyace en el algoritmo de planificación por lotería es muy básico: los boletos, son utilizados para representar cuanto se comparte de un determinado recurso para un determinado proceso. El porcentaje de los boletos que un proceso tiene es el porcentaje de cuanto va a compartir el recurso en cuestión.

Ej. Suponiendo que existen dos procesos A y B y que un boleto ganador esta entre 0 y 99 podría suponerse que el proceso A tiene el 75 % de posibilidades de recibir el recurso y el proceso B tiene el 25 % restante. En términos de boletos de la lotería el proceso A tendría los boletos del 0 al 74 y el proceso B tendría los boletos del 75 al 99. Un boleto ganador determina si A o B son ejecutados. Entonces por ejemplo de boletos ganadores de la lotería podrían ser: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 49. Entonces el resultado de la planificación es: A B A A B A A A A A B A B A A A A A A

Utilizar la aleatoriedad lleva a una correcta visión desde el punto de vista probabilístico pero no garantiza que esa proporción deseada se lleve a cabo. De hecho en el ejemplo anterior no sucede que se ejecute 25 75. No se garantiza que salgan esos valores.

Existen distintos mecanismos para manipular los boletos.

- Ticket Currency: Existen como en la realidad distintos tipos de moneda y las tareas pueden tener los tickets comprados con distintos valores de moneda; el sistema automáticamente los transforma en un tipo global de moneda.

- Transferencia de boletos: Este mecanismo permite que un proceso temporalmente transfiera sus boletos a otro proceso. Este mecanismo es útil cuando se esta utilizando la arquitectura cliente/servidor.
- Inflación: Con la inflación un proceso puede aumentar o disminuir la cantidad de boletos que posee esto lo puede hacer de forma temporal. Este proceso obviamente no puede realizarse en un sistema en el cual las tareas compiten entre ellas, ya que una tarea muy avara podría captar todos los boletos. Sin embargo, este método puede ser utilizado en un ambiente en el cual los procesos confían entre ellos.

Lo interesante de este método es la facilidad con la que se puede implementar. Necesita lo siguiente:

- Un buen generador de números aleatorios que determine cual es el numero de la lotería ganador
- Una estructura de datos para mantener la información de los procesos del sistema.
- Un numero total de tickets.

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to get a value,
// between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Planificación multiprocesador

Hoy en día con procesadores multicore hay que planificar de alguna forma también. (Las aplicaciones normalmente nacen secuencialmente y después se las hace multithreading para aprovechar las ventajas que da el multicore). Los planificadores no suelen mejorar el rendimiento de las aplicaciones haciéndolas multithreading, eso queda para el programador. (No tiene la inteligencia de pasar de secuencial a paralelo un programa)

Arquitectura multiprocesador

La diferencia principal se centra alrededor del hardware llamado cache y de que forma los datos que tiene son compartidos a través de los multiprocesadores.

La memoria principal, por el contrario mantienen todos los datos del sistema pero el acceso a los mismos es lento. A través de mantener los datos que son frecuentemente utilizados en la cache, el sistema puede hacer que una memoria larga y lenta parezca una memoria rápida.

Las caches aprovechan la localidad temporal y la espacial.

La localidad temporal se aprovecha cuando cierta cantidad de datos son accedidos, es muy probable que sean accedidos otra vez en un futuro cercano. *Ej. variables o instrucciones que se ejecutan una y otra vez en un ciclo.*

En la localidad espacial cuando un programa que accede a una dirección X es muy probable que necesite volver a acceder cerca de X. Acá podría pensarse en un programa sobre un arreglo. Teniendo en cuenta que este tipo de localidad existe en la mayoría de los programas los sistemas de hardware pueden hacer buen uso de las cache.

```
for(i = 0; i < 20; i++)
    for(j = 0; j < 10; j++)
        a[i] = a[i]*j;
```

En el caso de los multiprocesadores esto se complica, ya que al traer un dato de memoria y guardarlo en su cache, este puede sufrir modificaciones con el tiempo. Supongamos que este programa se cambia a otro procesador por el planificador con un cache diferente, ¿Que sucede con el dato? Puede ocurrir que el programa vaya leer devuelta a memoria el valor viejo si no se tiene cuidado.

Este problema se llama *coherencia del cache*. La solución básica que da el hardware es monitoreando los accesos a memoria, asegurándose que la vista de una memoria única compartida sea preservada. Esto lo puede hacer con una técnica llamada *Bus Snooping*, en donde cada cache pone atención a las actualizaciones de memoria mediante la observación del bus que los conecta. Cuando ocurre algún cambio la CPU anula su propio cache, provocando que se actualice.

Afinidad de Cache

Lo que si sucede en los planificadores, es que cuando el proceso corre sobre una CPU en particular, se va construyendo una parte del estado en la cache de esa CPU. Lo que se intenta hacer es que se mantenga esta afinidad para no tener que reconstruirlo.

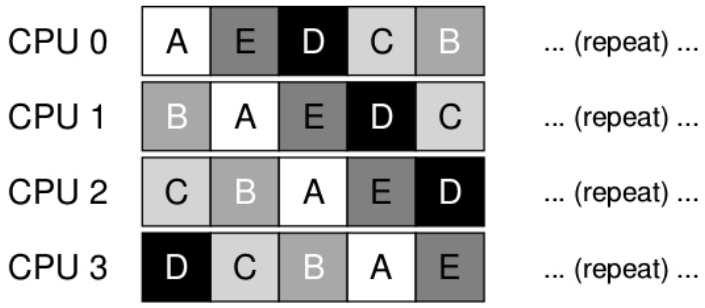
Si se ejecutase el proceso en un procesador diferente siempre, el rendimiento del proceso va a ser peor, ya que tiene que volver a cargar el estado.

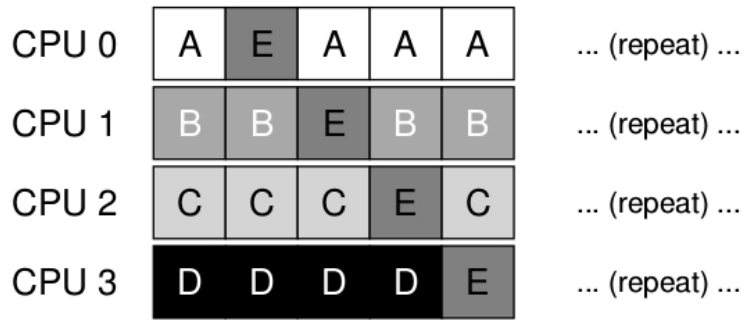
Planificación de cola única

Es la forma mas fácil de tener un planificador para un sistema multiprocesador. Se ponen todos los jobs en una única cola llamada Single Queue Multiprocessor Scheduling (SQMS). Esto tiene la ventaja de ser simple ya que no requiere mucho trabajo tomar la política existente y adaptarla a mas de un CPU.

Las desventajas de eso son que no es escalable, que se pierde la afinidad del cache, y que se tiene que agregar algún código para bloquear el acceso a la cola (solamente accede uno). El bloqueo este reduce mucho la performance.

Ej. La perdida de afinidad se puede ver con 5 procesos y 4 CPUs, a medida que terminan los time slice van cambiando los CPUs donde se ejecutan los procesos. Realizaría lo opuesto a la afinidad del cache, ya que saltan de CPUs.





Multi-Queue planification

Llamada Multi-Queue Multiprocessor Scheduling (MQMS), agrega múltiples colas, donde cada cola sigue una determinada disciplina de planificación. Este sistema es completamente escalable, a medida que crecen las CPUs, crecen las colas, sacando del problema los bloqueos y los caches. Otra ventaja que tiene es que provee afinidad de cache intrínsecamente, las tareas intentan mantenerse en la CPU en la que fueron planificadas.

El problema que presenta es el desbalance de ejecución (load imbalance). Esto sucede cuando un CPU esta sobrecargada de procesos mientras que otra no hace nada, llevando a un mayor desgaste al primero. Una solución seria moviendo las tareas de un CPU a otro, técnica llamada *migration*.

Figura 8: Desbalance de ejecución.

El planificador de Linux: Completly Fair Scheduler

Este planificador introdujo un algoritmo que calculaba en tiempo constante el time-slice y las colas de ejecución por proceso. Introdujo el concepto de Rotating Staircase Deadline Scheduler, que lleva al concepto de fair-share scheduler.

Una característica importante de este Completly Fair Scheduler (CFS) es que no otorga un determinado time-slice a un proceso, si no que le otorga una proporción del procesador, dependiente de la carga del sistema.

Para poder ser eficiente, CFS intenta gastar poco tiempo tomando decisiones de planificación. Esto lo consigue gracias a su diseño y al uso inteligente de las estructuras de datos.

Funcionamiento

Mientras que los planificadores tradicionales otorgan un time-slice fijo, CFS divide el tiempo del procesador entre los procesos que están compitiendo por ella. Esto lo hace mediante una técnica llamada virtual runtime (Vruntime). El vruntime es el runtime que corrió cada proceso normalizado por el numero de procesos runnable (medido en nanosegundos).

A medida que un proceso se ejecuta acumula vruntime, y al momento de decidir cual correr, CFS selecciona al proceso que menos vruntime tiene. Esto tiene dos cuestiones clave:

- Si CFS cambia de proceso en tiempos muy pequeños estará garantizando que todos los procesos se ejecuten a costa de perdida de performance debido a la cantidad de context switches.

- Si CFS cambia pocas veces, la performance del scheduler es buena, pero el costo esta puesto del lado de la equidad entre procesos (fairness).

En un estudio realizado por Google en uno de sus servidores encontró que incluso después de varias optimizaciones, el tiempo de planificación uso el 5% del tiempo.

El CFS se maneja entre estos valores mediante varios parámetros.

- **sched_latency**: Determina cuanto tiempo un proceso tiene que ejecutarse antes de considerar un switch. (time slice dinámico) Un valor común es 48ms, este es dividido entre todos los procesos ejecutándose en la CPU. CFS va a ser completamente justo con todos. Claramente el problema acá surge si hay muchos procesos, el time-slice seria muy pequeño, para eso esta el siguiente parámetro.
- **min_granularity**: Establece un valor mínimo del time-slice, asegurándose que nadie tenga menos de este tiempo de ejecución y de que no haya un overhead por el context switch. Normalmente 6ms.

El CFS tiene una interrupción periódica de tiempo cada 1ms, para que pueda tomar decisiones.

Prioridades: Weighting - Niceness

Algo que tiene el CFS es que se pueden asignar prioridades sobre los procesos por parte del usuario y administrador. El valor va de -20 a +19. El valor por defecto es 0, el que mas prioridad tiene es -20, y el que menos tiene es +19.

```
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, /* 0 */
};
```

Para obtener el time slice queda de la forma:

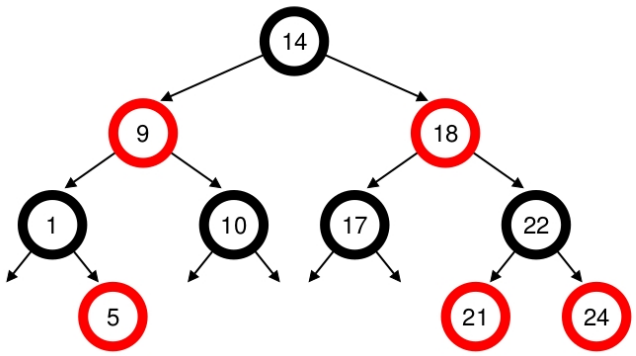
$$time\_slice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} \cdot sched\_latency$$

$$vrntime_i = vrntime_i + \frac{weight_0}{weight_i} \cdot runtime_i$$

Arboles Rojo-Negro

Para que el sistema sea eficiente, se utiliza esta estructura, ya que sus tiempos son muy buenos, O(log(n))
El algoritmo es el siguiente:

1. El nodo mas a la izquierda del árbol es el elegido, ya que es el que menos tiempo se ejecuto.
2. Si el proceso termina, se elimina del árbol.
3. Si el proceso alcanza su máximo tiempo de ejecución o para su ejecución voluntariamente, se reinserta en el árbol basado en su nuevo tiempo.
4. Repetir para el nuevo nodo mas a la izquierda.



API

- nice(): establece un valor de niceness.
- sched_yield(): permite a un proceso ceder el uso de la CPU.

unix Xv6

El enfoque que utilizan los sistemas operativos para crear la ilusión de que cada proceso tiene su propio procesador virtual es mediante la multiplexación de los procesos sobre el hardware disponible.

Xv6 multiplexa cambiando en cada procesador un proceso por otro en dos ocasiones:

- Los mecanismos de xv6 sleep y wakeup cambian cuando un proceso espera que se complete una operación de I/O, cuando espera por la finalización de un proceso hijo (wait) o cuando el proceso espera en la system call sleep().
- Xv6 periódicamente fuerza un cambio cuando un proceso de usuario se está ejecutando

Esto crea la ilusión de que cada proceso tiene su propia CPU, entre otras cosas. Este cambio entre procesos se traduce en un cambio entre modo usuario - modo kernel - modo usuario .

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns. It loops, doing:
// - choose a process to run
// - switch to start running that process
// - eventually that process transfers control
// via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
      // Enable interrupts on this processor.
      sti();

      // Loop over process table looking for process to run.
```

```
            acquire(&ptable.lock);
            for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->state != RUNNABLE)
                    continue;

                // Switch to chosen process. It is the process's job
                // to release ptable.lock and then reacquire it
                // before jumping back to us.
                c->proc = p;
                switchuvm(p);
                p->state = RUNNING;

                swtch(&(c->scheduler), p->context);
                switchkvm();

                // Process is done running for now.
                // It should have changed its p->state before coming back.
                c->proc = 0;
            }
            release(&ptable.lock);
        }
    }
```

Novena y décima clase

8. Concurrencia

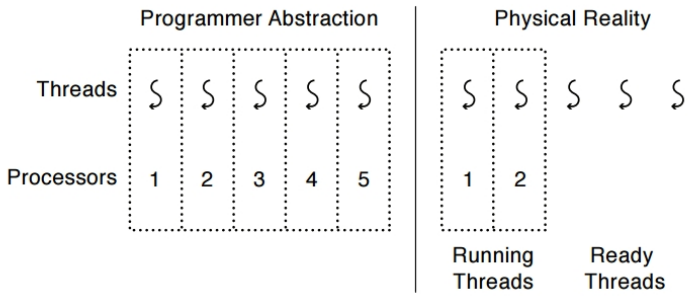
Definición de Thread: Un thread es una secuencia de ejecución atómica que representa una tarea planificable de ejecución.

Secuencia de ejecución atómica porque cada thread ejecuta una secuencia de instrucciones como lo hace un bloque de código en el modelo de programación secuencial (Desde el punto de vista del thread, tiene un inicio y fin y nunca para). Y tarea planificable de ejecución ya que el sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando él desee.

'Pretender estar haciendo varias cosas a la vez, pero siempre le dedicamos el tiempo de procesamiento a una cosa.' Es distinto a la ejecución paralela, en ese caso si se hacen a la vez, en el mismo tiempo t.

```
{
instruccion1
instruccion2
instruccion3
instruccion4
...
instruccionN
}
```

Como se vería un thread: 'secuencia independiente de instrucciones que representa una tarea planificable de ejecución'.



Threads vs procesos

Un proceso es un *programa* en ejecución con derechos restringidos, mientras que un thread es una secuencia independiente de instrucciones ejecutándose dentro de un programa.

Threads

Los threads cuentan con las siguientes características:

- Thread ID
- Un conjunto de valores de registros
- Un stack propio
- Una política y prioridad de ejecución

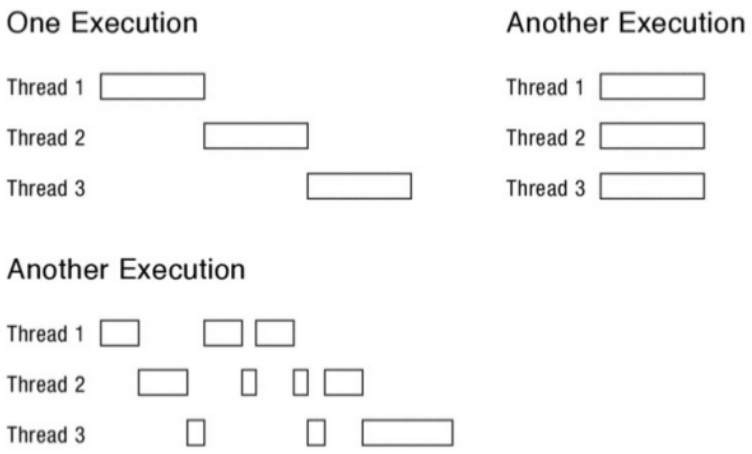
- Su propio *errno*
- Datos específicos del thread.

Hay diferentes casos de ejecución, un Thread por proceso, muchos Threads por proceso, muchos procesos de un solo Thread, y muchos Kernel Threads (el kernel puede ejecutar varios para aprovechar recursos).

Thread Scheduler

Ya que generalmente se trabajan con menos procesadores que Threads, es necesaria alguna forma de planificar quien se ejecuta, esta forma tiene que ser transparente, el programador no debe preocuparse de cuando es suspendido o no el thread. Esto hace que los Threads provean de un modelo de ejecución en el cual cada thread corre en un procesador virtual dedicado (exclusivo) con una velocidad variable e impredecible. (No podemos decir con que velocidad y cuando se va a ejecutar.)

Desde el punto de vista del thread cada instrucción se ejecuta inmediatamente una detrás de otra (atómico). Pero el que decide cuando se ejecuta es el planificador de threads o thread scheduler. El thread piensa que nunca paro de ejecutarse. Los mismos pueden ser ejecutados de 2 formas, de forma cooperativa (no hay interrupción a menos que se solicite) y forma preemptiva (un thread en estado running puede ser movido en cualquier momento).



La API

Tienen una biblioteca, no forma parte de la entandar.

Principalmente los threads se deben de poder crear y hacer join (esperar a que termine otro thread).

Un ejemplo utilizando la biblioteca *pthread*.

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>


typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;
```

```c
typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

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

    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    free(m);
    return 0;
}
```

¿Por que se usan los threads?

- Paralelismo: Se tiene el potencial de acelerar el rendimiento del programa.
- Si un thread realiza I/O, otro thread podría realizar otras operaciones mientras este listo para correr.

¿Por que se complica con los recursos compartidos?

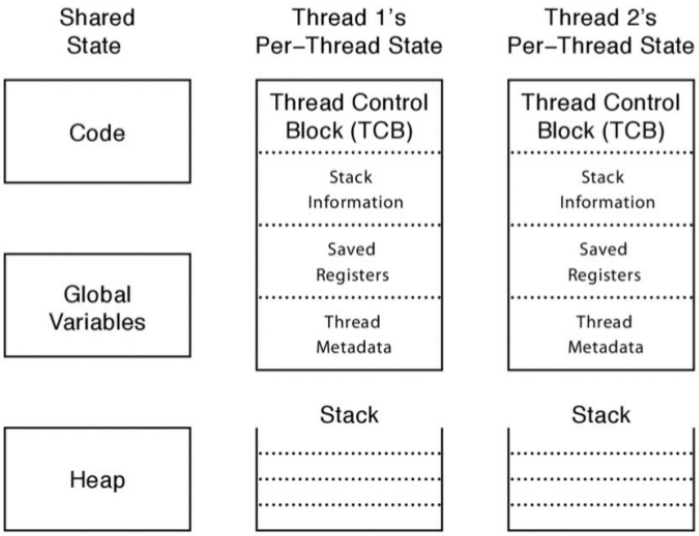
Porque no hay sincronización entre los threads. Corren de forma diferente dependiendo de lo que decida el planificador.

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. Este bloque debe almacenar el puntero al stack del thread y una copia de los registros en el procesador. De esta forma el sistema operativo puede crear, parar, y volver a iniciar múltiples threads sin problemas de estado de cada uno.

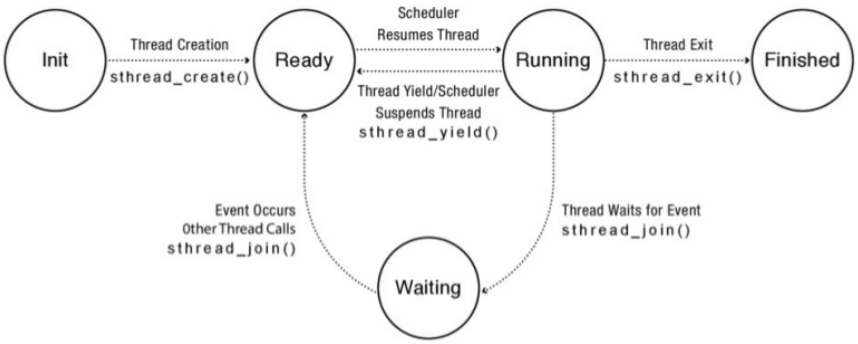
Por cada thread se guarda información del mismo también, ID, prioridad del planificador, y el estado.

De forma compartida se debe guardar también información, el código, variables globales, variables del heap. Todos los threads verían esta información.



Estados de un thread

- INIT: sucede mientras se está inicializando el estado per-thread y se está reservando el espacio de memoria necesario para estas estructuras. Una vez hecho, se setea en READY y se lo pone en una lista llamada *ready list* en la cual esperan todos los threads listos para ejecutarse.
- READY: Indica que el thread esta listo para ser ejecutado. La TCB esta en la *ready list* y los valores de los registros están en la TCB. Puede transicionar a RUNNING.
- RUNNING: El thread esta siendo ejecutado en este mismo instante por el procesador. Los valores de los registros están en el procesador. Puede pasar a READY si lo desalojan o preemption del mismo, o si el thread deja la ejecucion (con thread_yield).
- WAITING: El thread esta esperando que algun evento suceda. No puede pasar a RUNNING directamente. Se almacena en la *Waiting list*. Cuando ocurra el evento, el planificador lo pasa a estado RUNNING.
- FINISHED: El thread termino, no vuelve a ejecutarse. Pasa a la lista *finnished list* en la que están los TCB de los threads que terminaron.



Diferencias con los procesos

- Los threads
  - Por defecto comparten memoria
  - Por defecto comparten los descriptores de archivos
  - Por defecto comparten el contexto del filsystem
  - Por defecto comparten el manejo de señales
- Los Procesos
  - Por defecto no comparten memoria
  - Por defecto no comparten los descriptores de archivos
  - Por defecto no comparten el contexto del filsystem
  - Por defecto no comparten el manejo de señales

Linux utiliza un modelo 1-1 (proceso-thread) con lo cual para el kernel de Linux, un thread y un proceso son lo mismo, todo es una tarea ejecutándose. (No es así en todos los sistemas operativos). El scheduler es el mismo de procesos (sabe distinguir entre los tipos solamente).

System Call: clone()

La creación de un proceso y un thread llaman de fondo a esta system call (clone). Lo que las diferencia son los flags que le mandan.

```
#define _GNU_SOURCE
#include <sched.h>

int __clone(int (*fn) (void *arg), void *child_stack, int flags, void *args, ...
/* pid_t *ptid, void *newtls, pid_t *ctid */ );
```

- 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

En el caso de un thread, se la llama con mas permisos, y en un proceso lo menos posible.

Sincronización

Los programas con varios threads se pueden diferenciar en 2 tipos. Los que están compuestos por threads independientes que operan sobre datos separados entre si, y los que están compuestos por threads que trabajan de forma cooperativa entre si, compartiendo memoria y datos. Ambos casos tienen distintas formas de tratarse.

Hay que tener especial atención a como se programa cuando se sigue el modelo cooperativo, ya que la ejecución del programa depende de la forma en que los threads se intercalan en la ejecución, esta misma puede no ser determinística debido a decisiones del scheduler. Y además, los

compiladores pueden reordenar instrucciones para mejorar la performance[1], algo que es invisible generalmente con un solo thread. Estos problemas pueden llevar a bugs que se caracterizan por ser sutiles, no deterministicos, y no reproducibles.

Para evitar esto, hay que estructurar el programa para que resulte fácil el razonamiento concurrente, y 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.

Operaciones Atómicas

Ocurre en algunos casos que algunas instrucciones no pueden dividirse en otras al momento del ensamblado. En estos casos se garantiza la ejecución de las mismas sin tener que intercalar ejecución.

Para los problemas de este tipo, los programas tienen que ser **seguros (safety)** (nada malo va a pasar), donde el programa nunca termina en un estado incorrecto , y **Liveness** (si algo va a pasar tiene que ser bueno), donde el programa eventualmente siempre está en un estado correcto.

Revisar el algoritmo de Peterson. Trata sobre la exclusión mutua, que permite a dos o mas procesos/hilos de ejecución compartir recursos sin conflictos, usando solo memoria compartida para la comunicación. Los algoritmos involucrados tienen que ser distintos entre los procesos, donde uno para y espera al otro

Para lograr que sea atómico, un bloque se debe transformar a una sola.

Locks

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

Estos locks están asociados a determinados estados o partes de código, y requieren que el thread posea el lock para entrar en ese estado. Esto logra que solo un thread acceda a un recurso compartido a la vez, permitiendo la exclusión mutua.

Estos locks tienen dos estados, *busy* o *free*, empezando siempre en *free*. Para usarlos, se pide acceso al lock con *lock()* y se libera con *unlock()*. Para que esto funcione, la verificación y seteo del estado son operaciones atómicas. Si un thread ya tiene acceso a esa región, todos los demás threads esperan a que quede libre (es bloqueante).

Otras posibles acciones con locks son *trylock() y timedlock()*.

Los locks deben asegurar la exclusión mutua, progress (si no esta en uso, alguien debe poder tomarlo), y bounded waiting (Si T quiere acceder al lock y existen varios threads en la misma situación, los demás tienen una cantidad finita (un límite) de posible accesos antes que T lo haga.)

La Sección Crítica es aquella sección del código fuente que se necesita que se ejecute en forma atómica. Para ello esta sección se encierra dentro de un lock.

El sistema operativo no puede inferir donde poner un lock en un programa.

Nota: Una implementación básica es escribiendo y leyendo de memoria (JOS implementa esta). En técnicas de programación concurrente se ven en detalle implementaciones.

Condition variables

En ciertas ocasiones puede ser necesario que un thread espere que cierta condición o estado se dé para que este continúe su ejecución, por ejemplo, un servidor web necesita esperar a que

[1] Para evitarlo en C compilar con -O0

alguien le avise que hay una nueva petición web. Ahora mandarlo a esperar girando (spinning) en su propio lugar no es eficiente.

Una condition variable permite a un thread esperar a otro thread para tomar una accion. Son un objeto sincronizado que permiten de forma eficiente esperar por un cambio para compartir algún estado que está protegido por un lock.

Su API es *wait()*(suelta al mutex, suspende la ejecucion, y vuelve a tomar el mutex cuando ocurre el signal()), *signal()* (toma a un thread y lo pone en la ready list), y *broadcast()* (pone a todos en la ready list). No confundir signal y wait con las system calls de UNIX.

Una condition variable se utiliza cuando algún tipo de señal tiene que suceder entre los threads, por ejemplo esperar por un cambio en un estado compartido o variable compartida, por ende un lock siempre debe proteger la actualización de dicho estado. Las condition variables o también llamados monitores están diseñadas para trabajar en concordancia con los lock.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

En otro hilo.

```
pthread_mutex_lock(&lock);
ready = 1;
pthread_mutex_unlock(&lock);

pthread_cond_signal(&cond);
```

Internamente mientras que esta esperando, utiliza la operación NOP.

Ejemplo en tipos de datos sincronizados

Algunos ejemplos posibles son, una cola o un contador. Ejemplo sacado del Arpaci.

```
typedef struct __counter_t {
    int value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    pthread_mutex_init(&c->lock, NULL);
}


void increment(counter_t *c) {
    pthread_mutex_lock(&c->lock);
    c->value++;
Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
    pthread_mutex_lock(&c->lock);
    c->value--;
    pthread_mutex_unlock(&c->lock);
```

```
}

int get(counter_t *c) {
    pthread_mutex_lock(&c->lock);
    int rc = c->value;
    pthread_mutex_unlock(&c->lock);
    return rc;
}
```

Errores comunes en concurrencia

Non Deadlocks

- Atomicity violation: "el deseo de la serialización entre múltiples accesos a memoria es violado".
- Order Violation: "El orden deseado entre accesos a memoria se ha cambiado"

Deadlocks

Ocurre cuando dos o mas threads se bloquean entre si. Se puede dar cuando uno obtiene el lock, y por algún motivo nunca lo libera (exclusión mutua). Hold and wait, un thread mantiene un recurso reservado para si mismo mientras espera que se de alguna condición. No preemption, los recursos adquiridos no pueden ser desalojados por la fuerza (preempted). Circular wait, un conjunto de threads que de forma circular cada uno reserva recursos que necesita otro thread.

La circular wait se previene escribiendo código que nunca induzca a esperas circulares, por ejemplo con el establecimiento de un orden total, este orden asegurará que no se caiga en espera circular.

El hold and wait se previene haciendo que los lock se tomen en forma atómica:

Undécima clase

9. Sistema de archivos

Un sistema de archivos (file system) permite a los usuarios organizas sus datos, para que estos puedan persistir a través del tiempo. La definición formal es: *Una abstracción del sistema operativo que provee datos persistentes con un nombre.*

Los datos persistentes son aquellos que se almacenan explícitamente hasta ser borrados.

Los nombres están para facilitar la identificación por parte de los usuarios, y para tener la posibilidad de identificación que permite que una vez que un programa termina de generar un archivo otro lo pueda utilizar, permitiendo así compartir la información entre los mismos. (El nombre no es necesario realmente para un sistema de archivos.)

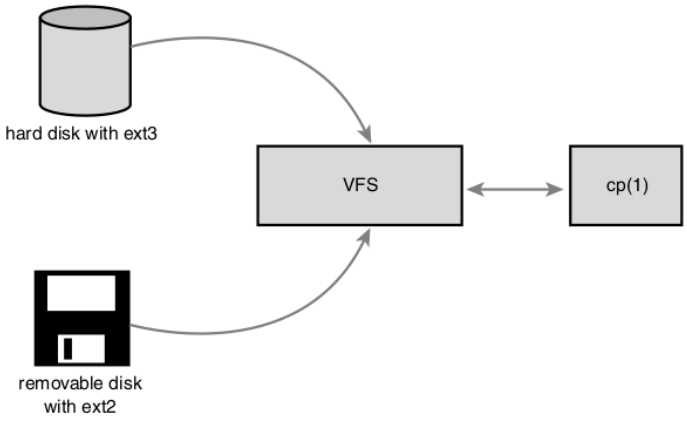
Esta abstracción esta compuesta por dos partes.

- Los archivos, que están compuestos por un conjuntos de datos.
- Los directorios

Virtual File System

El VFS es el subsistema del kernel que implementa las interfaces que tienen que ver con los archivos y el sistema de archivos provisto a los programas corriendo en modo usuario. Los sistemas de archivos deben basarse en el VFS para coexistir y poder inter-operar, y así poder usar las *syscalls* para leer/escribir en diferentes sistemas y medios.

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



Cualquier cosa que vaya a ser persistida va a pasar por el VFS. Actúa como proxy, para que el kernel pueda hablar con todos. (No todos los OS tienen un VFS)

File System Abstraction Layer

Es una interfaz genérica que para cualquier tipo de filesystem el kernel implementa con el sistema de archivos de bajo nivel. Esta capa de abstracción habilita a Linux a soportar sistemas de archivos diferentes, incluso si 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, por lo que cada filesystem adapta su forma de trabajo a lo que espera el VFS (por lo que todos los sistemas aceptan nociones como

archivos, directorios, y diferentes operaciones). 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.

Establece la linea base a cumplir, después la implementación puede variar dependiendo de cada sistema.

Estructuras

VFS tiene estructuras que modelan un file system.

- Super bloque: representa a todo un sistema de archivos.
- Inodo (inode): representa un determinado archivo dentro de un sistema de archivos. (físico en disco)
- Dentry: representa una entrada de directorio, que es un componente simple de un path. (sucesión del path hasta llegar al archivo)
- File: representa un archivo asociado a un determinado proceso. (el proceso abrió un archivo)

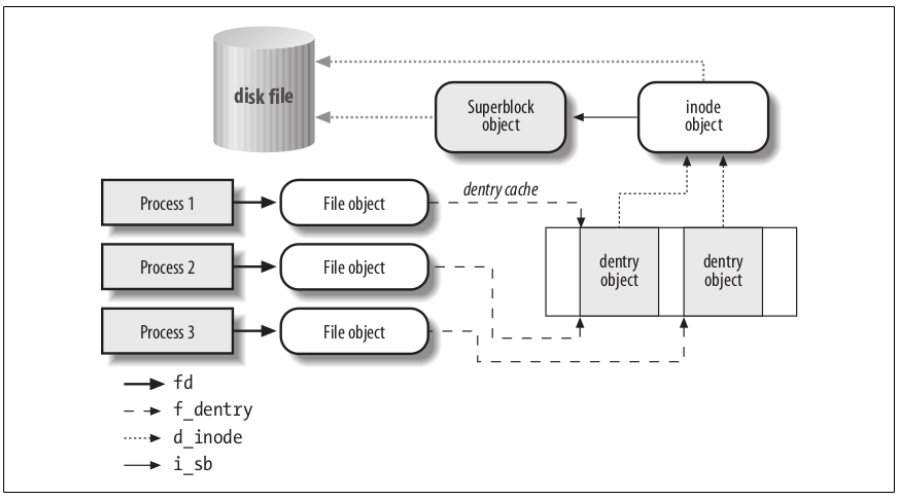


Figura 9: Notar las relaciones.

Tener en cuenta que un directorio es tratado como un archivo normal, no hay un objeto especifico para directorios. En unix los directorios son archivos normales que listan los archivos contenidos en ellos. Guardan la relación nombre del archivo-inodo. (Indica que archivos contiene el directorio)

Abrir un directorio seria ver un archivo binario, que tiene 4 bytes para ver el inodo, la longitud del archivo, la longitud del nombre, el tipo de archivo, y el nombre. (Todos tienen un '.' y '..' para representar al actual y al padre.)

Operaciones

- super_operations: Son métodos que aplica el kernel sobre un determinado sistema de archivos. *Ej. write_inode() o sync_fs().*
- inode_operations: Son métodos que aplica el kernel sobre un archivo determinado. *Ej. create() o link().*
- dentry_operations: Son métodos que se aplican directamente por el kernel a una determinada *directory entry. Ej. d_compare() o d_delete(), compara elimina directorios.*

- file_operations: Son métodos que aplica el kernel sobre un archivo abierto por un proceso. *Ej. read() o write(), las de archivos e siempre.*

Archivos

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

Ej. /home/mariano/MisDatos.txt, cada archivo tiene un nombre único y un significado para referirse a datos. Estos nombres proveen la abstracción de mas alto nivel del dispositivo de almacenamiento. Es mas fácil manejar el nombre que el bloque en que esta guardado en disco.

Cada archivo esta dividido en 2 partes, la metadata que contiene información acerca del archivo *(Ej. tamaño, fecha de modificación, propietario, seguridad)* guardada en la región conocida como **inodo**, y los datos que quieren ser almacenados (desde el OS es un arreglo de bytes sin tipo) guardados en un **data region**.

Directorios

Los directorios proveen los nombres para los archivos, es una lista de nombres *human-friendly* y un mapeo a un archivo o a otro directorio.

Definiciones:

- path: es el string que identifica unívocamente a un directorio o archivo dentro de un dispositivo.
- root directory: es el directorio de que cuelgan todos los demás. (la raíz)
- absolute path: es la ruta desde el directorio raiz. *Ej. home/hola/hola2*
- relative path: es el path relativo que se interpreta desde el directorio actual.
- current directory: es el directorio en el cual se ejecuta el proceso.
- hard link: es el mapeo entre el nombre y el archivo subyacente. Si se tiene un file system que permite muchos mapeos de estos, la estructura ya no seria un árbol (cuidado con los ciclos en estos casos). (un archivo que contiene la ruta a un archivo - un inodo puede ser apuntado por un montón de nombres - los links que tiene el inodo - es una referencia)
- soft link: un archivo que puede ser llamado con distintos nombres. (un archivo que apunta a otro archivo, en vez de tener un archivo directamente - un delete de un soft link no borra el archivo)
- volumen: es una una abstracción que corresponde a un disco lógico. En el caso más simple un disco corresponde a un disco físico. Es una colección de recursos físicos de almacenamiento.
- mount point: es un punto en el cual el root de un volumen se engancha dentro de la estructura existente de otro file system. *Ej. Seria la conexión que ocurre entre el disco de la computadora y el volumen de un USB, el disco de la computadora tiene directorios preparados para realizar la unión de directorios.*

Cuando los links hacia un archivo llegan a 0, se borra el archivo.

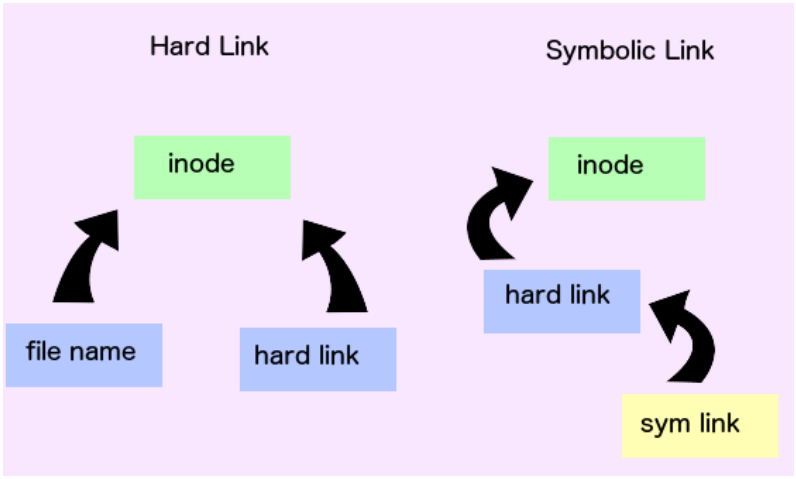


Figura 10: Hard y Soft Link.

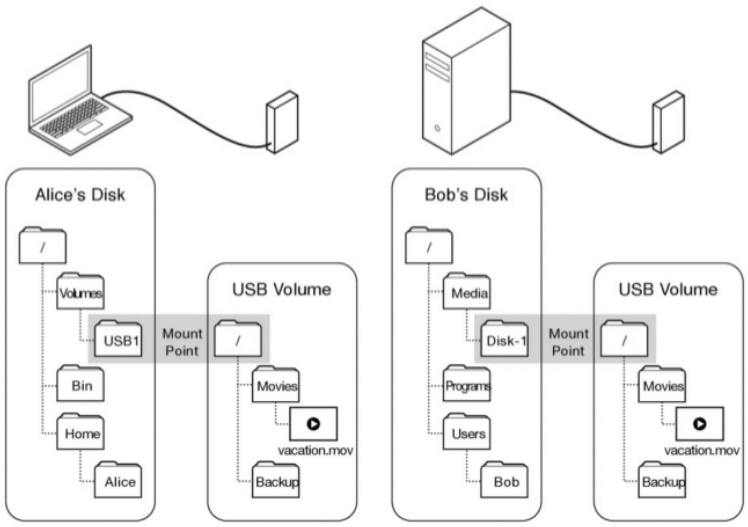


Figura 11: Mount Point

La API

Varias de las mencionadas en esta lista son utilizadas en una de las tareas del lab Fork (find). Sobre los archivos se tienen las siguientes llamadas:

- open(): devuelve un file descriptor
- creat(): equivale a open() con unos flags en especifico, es para crear un archivo.
- close()
- read(): no garantiza la lectura total de los bytes, hace intentos de lectura.
- write()
- lseek(): reposiciona el desplazamiento de un archivo abierto.

Command	System Call
cat	read() write() open() close()
rm	unlink() stat()
fstat	stat()
tee	open() close() read() write()
touch	creat() stat()
ls	readir() write()

- dup(): duplica el file descriptor.
- link(): crea un nuevo nombre para un archivo.
- unlink(): decrementa el contador de links sobre un inodo.

Sobre los directorios:

- mkdir()
- rmdir()
- chdir()

Para trabajar sobre los directorios, incluyendo *dirent.h*

- opendir()
- readdir()
- closedir()

Sobre los metadatos:

- stat(): devuelve información sobre un archivo.
- access(): determina si un proceso tiene permisos para acceder a un archivo.
- chmod(): altera los modos de acceso.
- chown(): cambia el id del propietario y el grupo de un archivo.
- umask(): setea una marcara con el modo de permisos para la creación de archivos.

Desde unix se tienen algunos comandos que de fondo llaman a estas syscalls.

Ejemplo de ls.

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

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

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

    return 0;
}
```

Ejemplo de Copy.

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

int
main(int argc, char *argv[])
{
    int fdold, fdnew;
    if (argc!=3){
       fprintf(stderr, "Se precisan 2 argumentos\n");
       exit(1);
    }
    fdold=open(argv[1], O_RDONLY);

    if (fdold==-1){
               fprintf(stderr, "No se pudo abrir el fichero %s\n",
               argv[1]);
               exit(1);
    }

    fdnew=creat(argv[2], 0666);

    if (fdnew==-1){
           fprintf(stderr, "No se pudo crear el fichero %s\n",
           argv[2]);
           exit(1);
```

```
    }

    copy(fdold, fdnew);
    exit(0);
}

void
copy(int old, int new)
{
     int cuenta;
     char buffer[2048];

    while ((cuenta=read(old, buffer, sizeof(buffer)))>0)
        write(new, buffer, cuenta);
}
```

Implementación

Se ve un vsfs (very simple file system)

- Se necesita una estructura de datos de un sistema de archivos, la forma de guardar la información en el disco (los datos y metadatos). Este sistema usa un arreglo de bloques.
- Se define un método de acceso, como relacionar las llamadas de los procesos al sistema de archivos (*open(), read(), ...*)

Son N bloques de 4kb. La mayor parte del espacio utilizado va a ser para la data region. Estas regiones tienen sus metadatos, los cuales se guardan en inodos, que deben guardarse a disco también en una inode table (arreglo de inodos almacenados en disco). Los inodos no son estructuras grandes (128,256 bytes). Los inodos representan la cantidad máxima de archivos que podrá contener el sistema de archivos también.

Otra cosa que hay que saber, es que inodos y bloques están siendo utilizados (o están libres), la estructura encargada de esto se llama estructura de alocación. En esta implementación se utiliza un bitmap, una para los datos (data bitmap) y otra para los inodos (inode bitmap). En esta estructura se mapea un 0 si esta libre, y un 1 si esta ocupada. Cada bitmap ocupa menos de 4kb, pero se utiliza un bloque por cada uno. (un bloque para inodos y otro bloque para datos)

También esta el Super Bloque (S), que contiene la información de todo el file system. Esto es, la cantidad de inodos, de bloques, donde comienza la tabla de inodos, donde están los bitmaps.

Se muestra un ejemplo de 64 bloques de 4kb, 56 bloques son para datos. Cada bloque de 4kb puede guardar 16 inodos (suponiendo que ocupan 256 bytes), se tienen 5 bloques de inodos (80 inodos). Un bloque para cada bitmap (I) (D), y el super bloque (S)

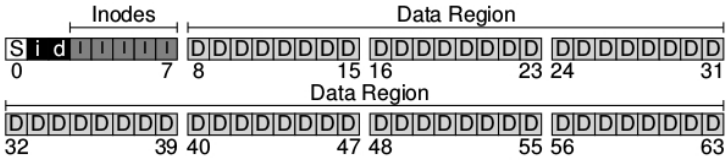


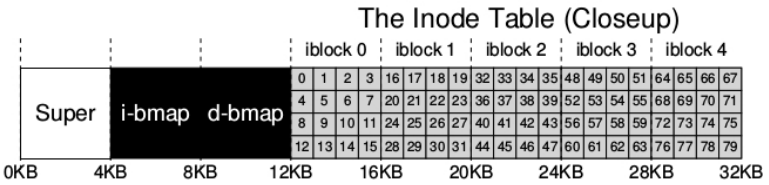
Figura 12: Distribución del ejemplo.

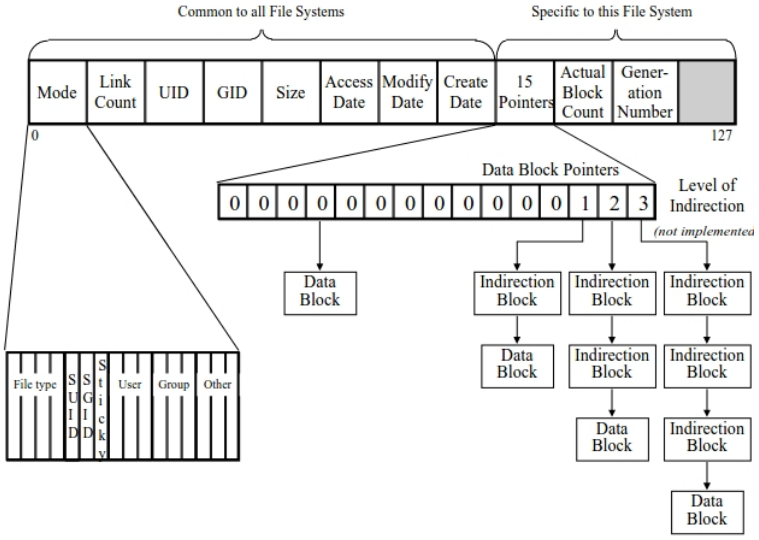
Inodos

Se refiere a un inodo a través de un inumber, dado un inumber se puede saber directamente en que parte del disco se encuentra el inodo.

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

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





10. Extras

Octava clase

10.1. Shell scripting

Algunas notas de shell scripting, se da como extra en la virtualidad. Información completa en el manual.

Para hacer un script normalmente se dice que interprete o shell script se va a usar. Se empieza con #!/bin/bash por ejemplo

Built in

Comandos que ejecuta la propia shell, sin la necesidad de crear un nuevo proceso. *Los siguientes fueron implementados en el lab shell.*

- echo
- cd
- pwd
- ...

Variables

En las shell se pueden definir variables. Todas son de tipo string, para otro caso hay que definirlo especificamente.

```
NOMBRE = "nombre"
```

Se obtiene el valor con el $:

```
$ echo $NOMBRE
nombre
```

Variables magicas

Hay un monton de variables propias de la shell, algunos ejemplos son:

- $? : Tiene el resultado del ultimo programa ejecutado.
- $# : Tiene el numero de argumentos.
- ...

Arreglos

```
array = (1 2 3 4 5)
```

Para verlo:

```
echo array
echo ${array[3]}
```

Estructuras de control

Hay diferentes estructuras de control. Se pueden comparar valores numericos y strings, usar if, y loops.

Funciones

A diferencia de las funciones de los lenguajes de programación 'reales', las funciones Bash no pueden devolver un valor cuando se las llama. Cuando se completa una función bash, su valor de retorno es el estado de la última instrucción ejecutada en la función, 0 para el éxito y un número decimal distinto de cero en el rango de 1 a 255 para el fracaso.

El estado de retorno puede especificarse utilizando la palabra clave return y se asigna a la variable $?. La utilización de return termina la función.

Charla

10.2. Internals de Git

Git hace varias cosas, una de ellas es el control de versiones. ¿Como hace para ir viendo la evolución del archivo? ¿Como se implementaría? Viéndolo con un ejemplo, tengo un único archivo (fisop.txt) y le hacemos un único cambio. Quiero mantener el estado histórico en los tiempos T1 y T2. Hay dos estrategias para guardarlas, una es guardar una copia del archivo entero, otra es guardar la diferencia, no todo el archivo.

Guardando solo la información nueva, se tiene una estrategia mas eficiente en espacio, pero es poco eficiente en la recuperación de estados anteriores, hay que ir reconstruyendo el archivo.

La otra alternativa de guardar las copias del archivo, es mas rápido en mostrar las versiones (no hay que reconstruir desde el comienzo), pero es poco eficiente en el uso de espacio.

Git usa ambas opciones, pero principalmente usa la versión de Snapshots (copias de todos los archivos enteros en todos los momentos). La historia no se puede cambiar, pero si se puede reescribir (esta todo referenciado). Git nunca guarda el mismo contenido mas de una vez, se da cuenta git que lo tenia guardado. De esta forma contra resta la parte mala del uso de Snapshots.

blob

Git se maneja con objetos, necesitamos una forma de modelar los archivos y sus contenidos. Objeto de tipo **blob**, representa los contenidos totales de un archivo. Si se cambia un archivo, se crea un blob nuevo. Todo blob tiene un nombre (no el del archivo), se obtiene con un hash (cadenas hexadecimales). *Ej. 23fca84....*

- git cat-file [-p|-t] <shasum>: Muestra contenido/tipo de un objeto, <shasum>es un hash
- git ls-files –stage : Muestra los archivos index y working directory

Si todos tenemos el mismo contenido, vamos a tener el mismo nombre de blob.
Los blob no pueden representar directorios.

tree

Los directorios se modelan como objeto **tree**. Contiene referencias a otros trees y blocs. Le da nombre a archivos y directorios. Todo tree tiene un nombre (no el del directorio) que se obtiene de un hash también.

- git diff-tree <tree-ish><tree-ish>: Compara dos trees y los contenidos de sus blobs
- git ls-tree : Muestra los contenidos de un tree

commit

Representa un snapshot del repositorio que contiene referencias a un único tree, metadata del snapshot (autos, cuando se hizo), uno o mas padres, y un nombre (no el del snapshot).

Si el archivo no cambia, los blobs se reutilizan. (No importa que nombre tenga el archivo, solo importan los contenidos)

tag

Representa una referencia a un objeto. Contiene referencias a un único objeto, metadata (autor, nombre, mensaje), y un nombre (no el del tag).
No todos los tags son objetos, solamente los **annotated tags**.

¿Como se guardan?

Git mantiene un mapa de objetos dentro de *.git/objects*. Cada objeto tiene como nombre su hash. Se guardan en shards para accederlos de forma mas eficiente. (Primeros 2 dígitos como nombre de una 'carpeta').

Ramas

No son objetos, toda branch es una referencia a un commit. Se guardan en *.git/refs*. HEAD es una referencia a una branch.

Packing objects y pruning

Guarda archivos enteros, no usa deltas, pero permite empaquetar objetos.
Compacta objetos en un único archivo guardándolos como diff a otro objeto.
Es eficiente para almacenar y transferir repositorios grandes.

- git prune: elimina objetos que no son alcanzables de la base de datos de git
- git count-objets: Cuenta la cantidad de objetos que se tienen y muestra cuanto pesa el repositorio
- git pack-objects: Crea manualmente un empaquetamiento de objetos

Si se crea un archivo muy grande y se borra, sigue referenciado, por lo que el repositorio va a seguir pesando eso. Por eso es malo subir imágenes/vídeos grandes a git si no son realmente necesarios.

10.3. Contenerización & Docker

¿Que pasa si tengo 2 procesos corriendo en la misma maquina, cada uno con su contexto y memoria, y quiero correr dos aplicaciones distintas que requieren cierta particularidad? Por ejemplo 2 aplicaciones Java que requieren JVMs distintas, o aplicaciones que escuchan al mismo puerto (ven la misma interfaz de red). Son recursos compartidos. Otro ejemplo es compilar una aplicación C que depende de bibliotecas especificas, distintas a las del sistema

La motivación es que no hay aislamiento a nivel proceso. Cada proceso tiene aislado su memoria, registros de CPU, tabla de archivos abiertos.

Dentro del contenedor estarían corriendo procesos específicos.

El proceso lanzado en Docker tiene una visión completamente distinta del sistema del resto de los procesos. Este contenedor, espacio abstraído no deja de ser un proceso en el fondo. Es un proceso con aislamientos mas fuertes que los demás.

¿Que es la contenerización?

Es un proceso normal corriendo en un kernel normal que tiene un control sobre la visibilidad y acceso a recursos del sistema, es un mejor aislamiento entre procesos dentro de un mismo host. Es una forma de virtualización ligera y rápida.

¿Por que nos conviene contenerizar?

Nos conviene para empaquetar dependencias, dando ambientes de desarrollo y producción reproducibles. Puede correr cierto paquete sin tenerlo instalado en la maquina nativa, un filesystem distinto.

También otorga un mayor control del uso de recursos (limites y cuotas), para la visibilidad y el aislamiento y seguridad. Si alguien compromete un servidor en un Container, no afecta a los recursos del resto de la maquina.

Además son fáciles de provisionar y movilizar.

¿Diferencias con una maquina virtual?

Una maquina virtual simula hardware, es mucho mas pesado y lento que provee un aislamiento mayor, donde cada VM tiene su propio SO.

Un Container es un aislamiento a nivel OS mas ligero y rápido, ya que comparte recursos del SO por debajo. Se pueden tener muchos contenedores sobre un mismo host.

¿Cómo funciona?

Tiene 4 pilares. Aplican al kernel de Linux.

- Namespaces: indica que recursos se pueden ver, mismo concepto que en algún lenguaje que los use. *Ej. C++*. Hay un namespace para cada tipo de recurso. Procesos en distintos namespaces ven distintos recursos. Procesos en distintos namespaces ven distintos recursos, estos son creados durante el fork (flags para la syscall que esta detrás - clone). Los namespaces se pueden referenciar como archivos. Áreas con un namespace son pid, stack de red, nombre de host y dominio, directorios y filesystems montados, ipc, distinto conjunto de usuarios y grupos, y la noción del tiempo del reloj del sistema. Cada namespace tiene una serie de estándares distintos sobre como se inicializa.

- Control groups: definen que recursos se pueden usar. (CPU, memoria, red, dispositivos) Permite al kernel tomar acciones cuando un proceso se excede de recursos. Existen distintas jerarquías dependiendo del recurso. La estructura que tienen es un directorio para cada recurso. Cada recurso es una jerarquía, cada jerarquía tiene uno o mas cgroups. Cada cgroup puede tener procesos. *Ej. Limitar la cantidad de memoria que pide un proceso.*

- Copy on Write: para un manejo eficiente del filesystem. No es parte del kernel, sino que es soportado por el filesystem. Varios containers pueden tener una copia de los mismos archivos. Utiliza mount points.

- Capabilities: es un tema de granularidad de permisos, acceso a ciertas syscalls. *Ej. No quiero que estos procesos tengan accesos a la syscall para reiniciar el sistema.* No tiene nada que ver con un usuario root. Son una serie de bits asociados a un proceso que indican que puede o no hacer un proceso. Se heredan, pero se pueden quitar y agregar.

11. Bibliografía

La bibliografía utilizada en la materia es la siguiente:

- Operating Systems: Principles and Practice - Thomas Anderson y Michael Dahlin
- Operating Systems: Three Easy Pieces - Remzi H, Arpaci-Dusseau y Andrea C. Arpaci-Dusseau[2]
- UNIX Internals: The New Frontiers - Vahalia, Uresh. Pearson Education India
- Computer Systems: A Programmer's Perspective - Randal E. Bryant, David R. O'Hallaron, Prentice Hall
- The Design of the UNIX Operating System - Bach, Maurice J. Prentice-Hall
- The Linux Programming Interface - Michael Kerrisk

[2]Gratuito desde http://ostep.org/