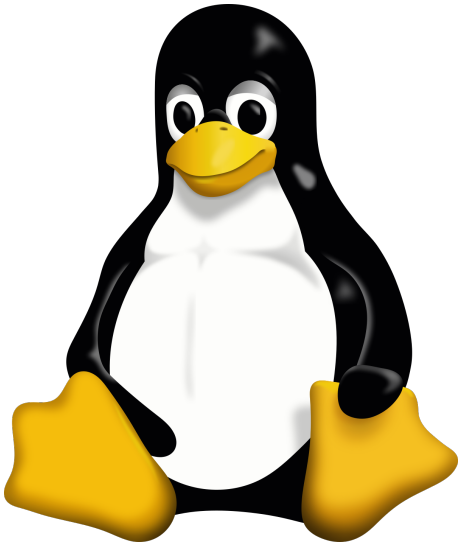
Sistemas Operativos: Resumen

Facultad de Ingeniería de la Universidad de Buenos Aires

Cátedra Mendez

2C 2023



Contents

1 Programa

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

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

2 Proceso

Proceso: Un archivo con información para generar el programa en memoria. Tiene las instrucciones en assembler, punto de partida, datos, símbolos y tabla de realocalizacion, etc... Toda esta metadata, es la mitad de los datos necesarios para lograr que el archivo se ejecute (convirtiéndose así en un proceso), la otra mitad la hace el kernel.

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

El kernel almacena la lista de procesos en una lista circular doblemente enlazada llamada task list.

2.1 Pasos en la creación de un proceso

1. El kernel carga las instrucciones en memoria. Hay distintas partes del archivo que corresponden a distintos componentes.
   a. .text: Codigo
   b. .dato: variables globales
   c. .heap
   d. .stack
2. Crea el stack y el heap
3. Transfiere el control al programa actual
4. Protege al SO y al programa

Cada proceso viene acompañado de varias abstracciones que el SO brinda para la ejecución correcta del programa. Las abstracciones habituales son: La memoria, el cpu, file descriptors, señales, etc.

Además, en el address space (posiciones de memoria al que pueden acceder) del proceso, se incluye una porción del código del kernel. Este incluye código que procesa la mayoría de syscalls (entre otras cosas). Esto se hace para minimizar la cantidad de veces que se tiene que llamar al "verdadero" kernel.

2.2 Virtual memory

Cada programa tiene un **address space propio.** El programa CREE que tiene acceso a la totalidad de la memoria. Este mecanismo está relacionado con el trabajo de ilusionista del sistema operativo.

En realidad, el SO mapea dicha memoria virtual a un espacio de la memoria física.

2.2.1 Mapeo de virtual a fisica

La traducción de memoria virtual a física es realizada por la MMU (Memory management system) en conjunto con el sistema operativo. Esta unidad de hardware se encarga de traducir direcciones virtuales a direcciones físicas.

2.3 Virtual processor

De la misma manera que cada programa cree que tiene acceso total a la memoria, pasa lo mismo con el procesador. Cada proceso cree que es el único que usa el CPU.

Esto es hasta cierto punto "verdadero". Cuando el programa está siendo ejecutado (en un momento determinado) SI es el único proceso que está siendo ejecutado. Sin embargo, NO es el único proceso que accede al CPU en el sistema. Como el cambio entre proceso y proceso es constante, decimos que cada proceso tiene un **contexto**

2.3.1 Contexto

Consiste en un conjunto de datos que un proceso tiene en un momento determinado. Cuando se cambia de proceso a proceso, se da un **context switch**: Se almacena el contexto de un proceso en memoria y entra el nuevo proceso. Este proceso es **caro**. Es por esto que se incluye parte del kernel en los procesos.

Hay muchos componentes que se guardan en el contexto:

- Nivel usuario
- Nivel registro
- Nivel Sistema

2.4 Estados

Un proceso puede estar en varios estados. Los principales son: Running (1 a la vez), ready (varios) y blocked (varios). El resto de los estados posibles son variaciones de estos (también hay otros, pero esos son los estados importantes).

3 Kernel

3.1 Punto de vista del usuario

- El Sistema de Archivos o File System.
  - Características:
    - Estructura jerárquica
    - Habilidad de crear y eliminar archivos
    - Tratamiento de los dispositivos periféricos como si fueran archivos.
    - protección de los archivos de datos
- El Entorno de Procesamiento.
  - El shell ejecuta los comandos buscando en ciertos directorios en una determinada secuencia.
  - Dado que el shell es un programa no forma parte del Kernel del sistema operativo
- Los Bloques Primitivos de Construcción
  - Proveer ciertos mecanismos para que los programadores construyan a partir de pequeños programas otros más complejos.

3.2 Cómo protege el kernel de un SO las aplicaciones y los usuarios

- Instrucciones Privilegiadas
  - Distintos set de instrucciones para usar en distintos rings
- Protección de Memoria
  - Dirección Virtual vs Dirección Física
  - Espacio de direcciones
- Timer Interrupts

  - Hardware Timer: cada timer interrumpe a un determinado procesador mediante una interrupción por hardware.
  - Cuando una interrupción por tiempo se dispara se transfiere el control desde el proceso de usuario al Kernel

3.3 Cambio de contexto

3.3.1 User Mode to Kernel Mode

- Interrupciones
  - Señal asincrónica hacia el procesador avisando que algún evento externo requiere su atención
- Excepciones del Procesador
  - Evento de hardware causado por una aplicación de usuario que causa la transferencia del control al Kernel.
- System Call
  - Algún procedimiento provisto por el kernel que puede ser llamada desde el user level.
  - Dado que todas las system calls son accedidas de la misma forma, el kernel tiene que saber identificarlas de alguna forma. Para poder hacer esto, la función wrapper **copia el número de la system call a un determinado registro de la CPU (%eax).**

3.3.2 Kernel Mode to User Mode

- Nuevo Proceso
- Continuar luego de una interrupción, excepción o una sys call
- Cambiar entre diferentes procesos

4 Signal

Una signal (también llamadas interrupciones por software) es una notificación que envía a un proceso cuando un determinado evento ocurre.

- Son análogas a las interrupciones por hardware, interrumpen el flujo normal de la ejecución de un programa, pero en la mayoría de los casos no es posible predecir el arribo de una señal.
- Un proceso puede enviar una señal a otro proceso (se usa para sincronización).
- Un proceso puede enviarse señales a él mismo.

Normalmente la fuente de envío de señales hacia un proceso es el Kernel.

4.1 Motivos por los que el Kernel envía señales a un proceso

- Una excepción de software (div por 0).
- El usuario presiona algún carácter especial en la terminal (CRTL-C).
- Un evento que tiene que ver con el software ocurrió, un proceso hijo terminó.

4.2 Ciclo de vida de una signal

- Generada
- Entregada
- Pendiente

Una señal es generada por un evento, una vez generada, esta es entregada a un determinado proceso, que tomará en algún momento una acción. El tiempo entre la generación y la entrega de la señal está pendiente.

Una señal pendiente es entregada al proceso tan pronto como este esté planificado para ser el próximo en correr o inmediatamente si este se encuentra en ejecución.

A veces es necesario que un proceso no sea interrumpido por la entrega de una señal. Para ello se agrega una máscara de señal.

Una vez que la señal se entrega, el proceso puede llevar a cabo una de las siguientes acciones por defecto:

- La señal es ignorada: es descartada por el kernel y no tiene efecto sobre el proceso.
- El proceso es terminado. Muchas veces se denomina abnormal process termination.
- Un archivo de core dump es generado y el proceso terminado. (imagen de la memoria virtual)
- El proceso es detenido: ejecución suspendida.
- El proceso continúa su ejecución.

Un programa puede modificar la acción por defecto de una determinada señal cuando ésta es entregada. Esto se denomina definir la **disposition**

4.3 Signal handler

Un signal handler es una función, escrita por el programador, que realiza la tarea apropiada en respuesta a la entrega de una señal.

5 Scheduling

Es la parte del kernel que decide que se va a ejecutar y cuando.

5.1 Conceptos relacionados

Multiplexación: Hacer que un recurso esté disponible para varios procesos

Multiprogramación: La idea es que en la RAM haya varios programas cargados en memoria (incluido el SO)

Time sharing: Consiste en establecer distintos intervalos de tiempo para que distintos programas estén en el cpu

5.2 Metricas

Para medir que tan bien anda un scheduler hay dos grandes métricas

5.2.1 Turn around time

"Tiempo que tarda una tarea en realizarse"

Se mide como:

(Tiempo en el que se completó una tarea) - (Tiempo en el que llegó la tarea)

5.2.2 Tiempo de respuesta

Mide que tan interactivo es un programa
Se mide como:
(Tiempo en el que inicia un programa) - (Tiempo llega una respuesta)

5.3 Ejemplos de schedulers

5.3.1 FIFO

- Es un scheduler intuitivo.
- Se procesan los procesos en el orden en el que llegaron.

Sufre de un gran problema, el **efecto convoy:** Si el primer proceso es muy largo, te traba todo. Entonces, un proceso que es cortito tiene que esperar un montón.

Shortest job first Trata de mitigar el efecto convoy. Dice que si llegan dos procesos a la vez, ejecuta el más corto primero.
Sin embargo, ES RARO que lleguen dos procesos A LA VEZ.

Shortest time to completion Consiste en que siempre hace el proceso más corto. Si llega un proceso más corto, para el que está haciendo y hace el que llega.
Sin embargo, esto presupone que el scheduler sabe la longitud del proceso. Esto no suele ser el caso (no pasa).

5.3.2 Round robin

- Se define un time slice (unidad de tiempo fija).
- Después de cada time slice ejecutas un nuevo proceso. Y va cambiando uno a uno en "círculos".

Esto logra un tiempo de respuesta mayor, dando lugar a más "interactividad". Pero viene al costo de **turn around time**: Este cambio constante no es muy performante ya que hay un cambio de contexto constante.

5.3.3 Multi Level Feedback Queue

- Trata de maximizar turn around time y tiempo de respuesta.
- Consiste de muchas colas con prioridades distintas.
- Cada tarea está en una única cola en un momento determinado.
- Solo se procesan los procesos que están en la cola de más alta prioridad.
  - Si hay varios procesos en más de una cola se realiza Round robin

Asignación de prioridades

1. Siempre que llega un proceso se le da la prioridad más alta
2. Si una tarea usa el CPU durante todo su timeslice entonces se le baja la prioridad
3. Si un tarea renuncia al cpu durante su time slice su prioridad aumenta

Si un proceso usa el CPU durante TODO su timeslice significa que es un proceso que es muy intenso y no es muy interactivo.
En cambio, en un proceso que rechaza el CPU, significa que no es un proceso muy intenso y que es interactivo.

Problemas

- Starvation
  - Problema: Los procesos que son muy heavies en cpu no son procesados nunca si hay procesos interactivos constantemente.
  - Solución: Se implementa un **boost**, el cual implica que cada cierto periodo S, todos los procesos pasan a la mayor prioridad
- Gaming (de Gaming the system):
  - Problema: Si un proceso es inteligente, podría, voluntariamente, dejar de pedir CPU para mantenerse siempre en la mayor prioridad
  - Solución: Se implementa **accounting.** Se lleva una cuenta de cuantos time slices un proceso tuvo arriba del todo. Cuando llega a su cuota máxima, se tira para abajo

5.3.4 Completely Fair Scheduler

- Es el scheduler usado hoy en día en Linux.
- Se basa en calcular proporciones de tiempo del procesador en vez de pensar en time slices de la misma longitud.
- Es altamente escalable ya que usa estructuras de datos muy performantes.
- Usa el concepto de virtual runtime.

Virtual runtime: Este es el runtime (tiempo que estuvo en el procesador), normalizado por la cantidad de procesos runnable. Cuando tiene que cambiar de proceso en ejecución, elegiría **el proceso con menos vruntime para que sea el próximo en ser ejecutado.**

Parametros

- **Sched_latency:**
  - Mide la cantidad de tiempo antes del switcheo de proceso.
  - Es dinámico y cambia con la cantidad de procesos.
    - Problema: Si hay muchos procesos el cambio de contexto sería muy a menudo y habría una menor performance
    - Solución: **min_granularity**
- **Min_granularity:**
  - Es una medida que define la MÍNIMA cantidad de tiempo que un time slice puede durar.
  - Este valor es modificable. Por defecto vale 6ms
- **Niceness:**
  - Mide la prioridad de un proceso.
  - Va de -20 (mayor) a +19 (menor prioridad). Por defecto es 0.
  - Cada valor de nice mapea a un **weight**
- **Time slice**:
  - En CFS, es variable y se calcula así:
    - $time\ slice\ =\ \frac{weight_k}{\sum_0^{n-1} weight} \cdot\ sched\ latency$
    - En espanol": Su peso normalizado por el sched latency

Arbol rojo y negro Un árbol rojo-negro es un árbol binario de búsqueda en el que cada nodo tiene un atributo de color cuyo valor es rojo o negro. En adelante, se dice que un nodo es rojo o negro haciendo referencia a dicho atributo.

Además de los requisitos impuestos a los árboles binarios de búsqueda convencionales, se deben satisfacer las siguientes reglas para tener un árbol rojo-negro válido:

- Todo nodo es o bien rojo o bien negro.
- La raíz es negra.
- Todas las hojas (NULL) son negras.
- Todo nodo rojo debe tener dos nodos hijos negros.
- Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

Cuando el scheduler es invocado para correr un nuevo proceso, la forma en que el scheduler actúa es la siguiente:

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

6 Memoria

Espacio de direcciones: Representa el sandbox donde se aloja cada proceso.

6.1 Virtualización de memoria

Mapeo de n elementos virtuales a m elementos físicos. Ningún proceso sabe que la memoria está virtualizada.

La **virtual address** tiene que ser traducida a la **physical address** por la MMU. El SO delega esa responsabilidad a la MMU.

6.1.1 Trabajo del SO

El sistema operativo se encarga de manejar las direcciones de memoria para minimizar la fragmentación y ser eficiente en tiempo.

El sistema operativo tiene que involucrarse en los puntos claves de la address translation para:

- Setear al hardware de forma correcta para que esta traducción se de lugar;
- Gerenciar la memoria
  - Manteniendo registro de qué parte está libre.
  - Qué parte está en uso.
  - Manteniendo el control de la forma en la cual la memoria está siendo utilizada.
- Intervenir de forma criteriosa como mantener el control sobre toda la memoria usada.

6.2 Implementaciones

6.2.1 Segmentacion/ Base + Bound

- Cada proceso tiene su propio address space.
  - Address space: contiene todo el estado de la memoria de un programa en ejecución.
- Los límites de ese address space están delimitados por dos registros. **Base & Bound**.
  - Base contiene la dirección inicial del address space
  - Bound tiene el tamaño del address space o el límite del mismo.

Para traducir de la dirección virtual a la física se usa la cuenta:
$\text{Registro}_{\text{base}}$ + offset.

Si eso daba mayor al bound, entonces tenías un error (estás tratando de acceder a memoria que no le pertenece al proceso.)

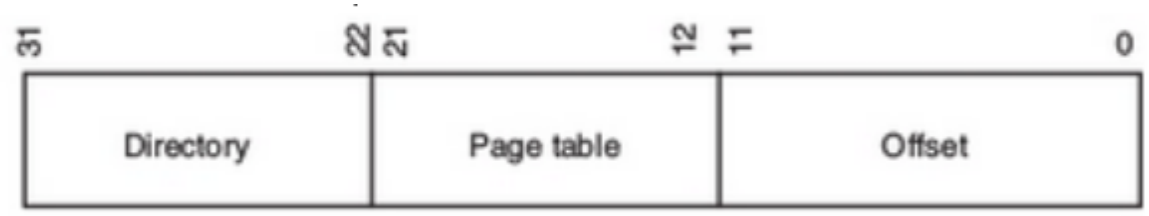
En x86 En x86 había más de dos registros para la segmentación, hay 4. CS (code segment), DS (data segment), SS (stack segment), ES (Extra segment). Cada uno de esos registros tiene su propio offset. 2 bits de dirección y los otros 30 de offset (La longitud de cada dirección es la longitud de la arquitectura, en x86 es 32 bits).

El SO se encarga de que las direcciones virtuales no mapeen a lago por fuera. El error "Segmentation fault" se daba justamente cuando tratabas de acceder a memoria por fuera de tu segmento.

La segmentación permitía tener procesos mezclados en memoria. Nos es como el base and bound tradicional en donde había un único "bloque".

6.2.2 Memoria paginada

- Se divide la memoria física y virtual.
- La memoria virtual está dividida en **pages** y la física en **frames.**
  - Cada frame, mide lo mismo que cada page que mide lo mismo que 1k.
- Se mapea cada frame a cada pagina.
- La virtual address está partida en 3 sectores.



(a)

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

Page directory

- Esta primera sección apunta a una tabla llamada La **Page Directory.**
- Esta tabla tiene elementos de 4 bytes (32 bits). Tiene 1024 entradas (de 0 a 1023). La tabla en total mide 4096 bytes.
- **Cada elemento apunta a OTRA tabla.** Es decir, que tenes 1024 entradas a 1024 tablas. Vamos a decir que guarda la dirección de la tabla B.
- Cada tabla tiene 4mb de almacenamiento. Si el proceso requiere de más memoria, se le asigna otra tabla.
- De los 32 bits de la entrada en la page directory, solo son los primeros 20 que apuntan a la otra tabla. Los otros 12 se usan para metadata de la tabla en cuestión.

Page Table Entry

- En estos 10 bits, está **EN QUE POSICIÓN** de la tabla B fijarte.
- Entonces vos tenes la dirección de la tabla b (que lo obtuviste arriba) + la dirección en la que te tenes que fijar.
- Esa dirección es el **frame** en el que tenes la memoria deseada

Memory page offset

- Indica el offset de la página.

Este esquema provee una granularidad mucho mayor. Hay un registro (CR3) que guarda la dirección de la page directory de cada proceso.

7 Cache

TODO: Copiar [1]

8 Threads

TODO: Copiar [2]

9 Filesystem

Forma de organizar los datos de forma persistente.
Un dato es persistente cuando está almacenado hasta que es borrado (la ram no aplica porque se borra cuando se le corta la corriente).
A nivel SO es una abstracción que provee datos con un nombre.
Contiene:

- Archivos: Contienen información
- Directorios: Contenedor de archivos y otros directorios.

[1] Esto no lo esta resumido porque justo lo vimos en taller; y el concepto es el mismo que en taller. Si tenes resumido esto, los PR son mas que bienvenidos: PONERLINK

[2] Ver footnote 1

10 Unix filesystem hierarchy

En Unix hay un estándar de directorios. Esto quiere decir que hay directorios que contienen archivos con un propósito común. Ejemplos:

- /dev: Dispositivos
- /etc: Archivos de configuración
- /proc: Metadata sobre los procesos, como fd, etc
- /lib: Bibliotecas.
- /bin: Binarios genericos.
- /sbin: Binarios relacionados con el mantenimiento del sistema
- /usr: Un montoooon de cosas. Hoy en dia, la mayoria de las distros, guardan los binarios de /bin y /sbin aca.
- /mnt: Directorio para montar dispositivos externos (USB, Harddrives, etc)
- /boot: Archivos necesarios para el booteo
- /sys: Tiene archivos "virtuales" que contienen información del kernel.
- /tmp: Archivos temporales
- /root: Directorio "HOME" de root. Cuando root se logea, se logea aca.
- /home: Aca estan los directorios de los distintos usuarios

Etc, hay mas y pueden variar. Es solo un estandar.

10.1 Virtual filesystem

Subsistema del kernel que estandariza el procesado y tratamiento de archivos de todos los filesystems posibles.

Este hace de interfaz para poder realizar operaciones intra filesystems distintos.

Cuando interactúa con el SO, uno simplemente llama a las syscalls "tradicionales": open, close, write. Es el virtual filesystem el que se encarga de hacer la llamada a la función correspondiente.

-> write() -> sys_write() -> write específico al filesystem (ejemplo write_xfs) -> escribir al disco.

10.1.1 Objetos del filesystem

Una implementación de filesystem necesita de estas cosas para ser considerado un vfs:

- Super bloque: Representa en file system en sí mismo.
- Inodo: Representa un archivo en el sistema.
- Dentry: Un conjunto de entradas en un directorio. Cada una de estas representa los archivos presentes en el directorio.
- File: Representación de un archivo en memoria utilizado por un proceso.

Contenido de archivo

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

  - Tamaño
  - fecha de modificación
  - Propietario
  - información de seguridad (que se puede hacer con el archivo.

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

10.2 Implementación de filesystem

Filesystem, refiere a dos cosas. Al **concepto** del filesystem y la **implementación**.
Hay muchas implementaciones del "filesystem". Cada una funciona "igual" pero almacena metadata distinta y tiene algunos pros/contras.

10.3 API del filesystem

10.3.1 API archivos

open Devuelve el file descriptor del archivo abierto

creat Crea un archivo y devuelve la el file descriptor

close Le pasas un file descriptor y lo cierra

read Trata de leer count bytes del fd y los guarda en el buffer. Devuelve la cantidad de bytes leidos.

write Trata de escribir count bytes del fd y los guarda en el buffer. Devuelve la cantidad de bytes escritos.

lseek Sobre escribe el offset de un file descriptor.

dup Duplica un fd, devolviendo el numero mas chico disponible

dup2 Duplica un fd, el que vos le especificas

link Crea un hard link.

unlink Elimia un nombre de un archivo del filesystem.
Si además ese nombre era el último nombre o link del archivo y no hay nadie que tenga el archivo abierto lo borra completamente del sistema de archivos.

10.3.2 API directorios

mkdir Crea un directorio

rmdir Borra un directorio

opendir Abre y devuelve un stream que corresponde al directorio que se está leyendo.
Devuelve un DIR * (steam que representa e ldirectorio que se está leyendo)

readdir Lee un DIR * y devuelve unn dirent *
El dirent tiene el nombre del archivo y un inodo

closedir Dada un close DIR*, lo cierra

10.3.3 API metadatos

stat Dado un pathname, devuelve estadisticas sobre el archivo
Devuelve un struct stat

access Dado un pathname, dice si un proceso tiene permiso a usar el archivo de una manera expecica.
Le podes preguntar si existe, lo podes leer, escribir o ejecutar.

chmod Dado un pathname y un modo, le cambia los bits del modo de acceso.

chown Dado un pathname, puede cambiar el owner (user y o group)

10.3.4 Ejemplos

ls

```
int main() {
  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;
}
```

10.4 File descriptors y archivos

No hay una correspondencia 1 a 1 entre archivos e inodos. A veces, es muy útil y necesario tener varios file descriptors referenciando al mismo archivo abierto.
Por esto existen mecanismos en el kernel para esto:

- Per process file descriptor table
- System wide file descriptor table
  - El offset actual del archivo (que se modifica por read(), write() o por lseek();
  - los flags de estado que se especificaron en la apertura del archivo (los flags arguments de open());
  - el modo de acceso (solo lectura,solo escritura, escritura-lectura);
  - una referencia al objeto i-nodo para este archivo.
- File system inode table

10.5 Very simple filesystem

Ejemplo de implementación de filesystem.
La unidad básica es el bloque. Tiene 4k de bytes de tamaño cada bloque.

10.5.1 Division data y metadata

Como sabemos cuanto dedicarla al almacenamiento de datos y cuánto darle al almacenamiento de metadatos?

Tenes que saber cuantos inodos vas a tener en el filesystem.
Fórmula para cantidad de inodos en un bloque = Tamaño bloque / tamaño inodo (256).
La suposición inicial es que como mínimo un archivo ocupa un bloque.
Sin embargo, no podes usar todo el disco para inodos. Necesitas espacio para la metadata.

Super bloque El VFS tiene una referencia a este superbloque.
Contiene "toda" la información del sistema de archivos. Además, tiene una tabla donde se encuentran los inodos. Sabe dónde empiezan los bloques de inodo y, a través de un offset, puede encontrar el resto

Bloque de Bitmap de bloques y bloque bitmap de inodos Tienes que saber qué bloques están ocupados y que inodos están ocupados.
Para esto se usa un bitmap. En 32 bit, podes mapear hasta 32768 inodos y bloques.
Si tuvieses un disco con más espacio necesitarías más bloques.

Bloques de metadatos La cantidad de bloques de metadatos necesarios se calcula de la siguiente manera:
bloques / 16
Ya que en cada bloque entran 16 inodos

Bloques de datos El resto de los bloques libres es el espacio que se utilizará para los datos.

Inodo Contiene información sobre el archivo.

- 4 bits que te indica qué tipo de archivo es
- Link count (cantidad de hard links)
- User ID y el group de user
- Sticky bit
- 3 bits para el permiso de usuario
- 3 bits para el permiso de grupo
- 3 bits para el permiso de otros
- Punteros a bloques (en el VSFS).
  - Guarda punteros a bloques.
  - Esto te permite que los bloques del archivo puedan estar esparcidos en el discos.
  - Si es más grande, el bloque 13 guarda un puntero a un bloque con punteros a bloques.
  - Si es mas grande, el bloque 14 apunte a un bloque que apunta a bloques que apunta a bloques (con esto podes tener un archivo de *4TB*).
  - Si tenes uno mas grande todavia tenes un puntero de 3ple indirección.

11 Booteo

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

11.1 Memoria

Distintas partes de la memoria (en 32 bits) tienen distintos propósitos.

- Low memory: 0 hasta 640KB
- VGA: 640 a 768
- Low memory: 768 y 960
- BIOS ROM: 960KB a 1M

11.2 Modo de la CPU

- Real: Direcciones de 16 bits. Todas fisicas
- Protegido: Direcciones de 32 bit. Fisicas

11.3 Boot

Proceso que lleva varios pasos

1. Bios:
    a. Arranca en modo real
    b. Cuando prendes la máquina, se lee de la parte ROM de la memoria.
    c. Detecta un disco booteable y lo trata de meter en memoria.
    d. Lee del disco los primeros 512 bytes. Esos 512 bytes son el bootloader. No más no menos.
    e. 0000-7C00
    f. En 7C00 esta es la primera instrucción que se ejecuta (apunta el EIP).
2. Bootloader
    a. Inicializa el stack segment, data y extra
    b. Enciende el modo protegido, lo cual saca el modo real y habilita las instrucciones de 32 bits.
    c. Una vez hecho eso pone en el esp el main del boot.
3. Bootloader en C
    a. Lee el encabezado del ELF. Chequea que el ELF tenga el valor mágico correcto.
    b. Busca todos los segmentos ejecutables del kernel y los carga en memoria.
    c. Después, llama al punto de entrada y ejecuta lo que está en memoria (aka la parte ejecutable del kernel).

12 Máquinas virtuales

Es la virtualización de la totalidad de una computadora.

Definición: Entorno de computación completo. Con sus propias capacidades. Aisladas una de las otras.

12.1 Tipos Lenguaje

12.1.1 Lenguaje

Ej: JVM

No las vamos a ver en clase. Fundamentalmente un intérprete.

12.1.2 Liviana

Docker

El kernel es el que aísla cosas de otras.

12.1.3 De sistema

Queres tener sobre el mismo hardware varios sistemas operativos.

Virtual machine monitor Es el que se encarga de sincronizar las distintas máquinas virtuales.

12.2 Utilidades

- Diversidad: Una sola máquina, varios so.
- Consolidación de servidores: Aislar distintos servicios. Es la forma más fuerte de aislamiento
- Aprovisionamiento rápido: Crear una máquina virtual es trivial. Antes tenía que tener el hardware para correr tus programas.
- Alta disponibilidad: SI tenes que matar una máquina hardware lo podes hacer fácilmente, moves la máquina virtual a otra nueva
- Seguridad: Más aisladas y más fácilmente monitoreable
- Scheduling de recursos compartidos
- Cloud computing: Se basa en las máquinas virtuales.

12.3 Requerimientos

- Fidelidad
  - Es equivalente al hardware real. El SO debería pensar que está corriendo en un hardware real.
  - Tiene que ser totalmente transparente.
- Seguridad
  - Tiene que estar aisladas una de las otras
- Eficiencia
  - En el peor de los casos tiene que andar más lento

12.4 Tipos de VM

12.4.1 SO especial Hypervisor - Tipo 1

Se encarga de supervisar las distintas vms. No hay SO base, solo el hypervisor.
Tenías un SO especial/privilegiado para supervisar y configurar.
Conceptualmente similar a un microkernel.

12.4.2 Hypervisor en espacio de usuario - Tipo 2

El supervisor es instalado como un programa más, corriendo en espacio de usuario.

12.5 Técnicas de virtualización

Todas estas técnicas se usan de forma coordinada.

- Multiplexing: Tenes una única memoria física y la dividis en varias cosas virtuales
- Aggregation: Varias cosas que hacen que se parezca como una.
- Emulation: Emulas algo que no tenes con algo que tenes. Menos eficiente.

12.6 Formas de implementar

12.6.1 Emular todo

- Emulas todo, el hipervisor no le da acceso a ninguna de los componentes reales al SO guest.
- Esto no es particularmente eficiente. Si queres correr varios sistemas operativos se complica.

Qemu

- Tiene un JIT a un lenguaje intermedio.
- Interpreta ese lenguaje en el momento.
  - Eso es lo que permite que el qemu ejecute varias arquitecturas.
  - Si la arquitectura es la misma, se saltea el proceso de traducción, se ejecuta directamente el código.

12.6.2 Ejecución directa

Las instrucciones de la máquina virtual se ejecutan directamente en el cpu host.
Se complica si tenes varias máquinas que quieren ir al mismo hardware (ej, disco rígido).
El VMM hace de scheduler entre las distintas vm del acceso.
La vm le pide al hypervisor, el hypervisor al SO y recién ahí.
Esto rompe la fidelidad, porque el SO tiene que saber pedirle al hipervisor. No anda "asi nomas".
El protocolo de la syscall está definido. Pero no está definido el protocolo para que el kernel le hable al hipervisor.
El truco es usar los discos no usados del cpu (ring 1,2, etc). Esto se debe a que en el cpu cambia de ring en los General Protection Exception. La VMM atrapa esta excepción. Esto se llama *trap and emulate*.

Hay problemas Intel añadió la instrucción popf que si se ejecuta sin los privilegios correctos **NO** hay excepción.
Esto arruina toda la idea del trap and emulate. Si un SO huésped depende de popf, estás jodido.

Soluciones

- Traduccion binaria: Traducción en tiempo de ejecución de las instrucciones sensibles. El hipervisor tiene que ver que se está ejecutando y reemplazar las instrucciones truchas por buenas. Este comportamiento es como una emulación pero a menor escala. No se usa hoy en día. Muy difícil
- Para virtualización: La guest sabe que es una vm. Llama a una system call que es atrapada por el hipervisor. Esto obvia el trucazo. Esto lo que tiene de malo, es que no funciona en cualquier sistema operativo. El so tiene que estar preparado para eso. Fácil para el hypervisor, jodido para el que implementa el so.
- Virtualización asistida por hardware: Extensiones para el procesador. Añadieron un anillo adicional. Un anillo es el root mode y el non root mode es el anillo para las cosas virtualizadas. Si ejecutas popf en non root mode saltas al root mode.
  - En el root mode kernel corre en el ring0 y el usermode en ring3 (como siempre).
  - En el non root mode kernel corre en el ring0 y el usermode en ring3 (como siempre).
  - El que puede poner algo en root mode o en un non root mode es el kernel.
    - Hay instrucciones para esto vmlaunch. El vmexit es el inverso.
- Virtualización moderna (KVM): Podes lanzar las vms como si fuesen procesos. Lanzas qemus que configuran usando la asistencia por hardware. Así, el scheduler del kernel schedulea las máquinas virtuales. El kernel es el que pasa de modo virtual a no virtual.

12.6.3 Teorema de la VMM

Un VMM se puede construir si el conjunto de instrucciones sensibles es un subconjunto de instrucciones privilegiadas

Instrucciones sensibles: Instrucciones que se ejecutan diferente si están en modo usuario o kernel. Eg. I/O, modificar MMU, etc

Instrucciones privilegiadas: Instrucciones que causan traps en modo usuario pero no en modo supervisor