

#### Compartición de Recursos: Sistema Operativo como Réferi

La compartición de recursos supone ciertos desafíos para el sistema operativo:

**Alojamiento de Recursos**

¿Cómo se debería elegir cuántos recursos le llega a cada tarea?

El sistema operativo debe mantener a todas las actividades simultáneas de forma separada, alojando recursos para cada una según sea apropiado. Si el sistema le da muy poca memoria a un programa, puede que no sólo faliente a ese programa sino también a la performance de toda la máquina.

**Aislamiento**

Un error en una aplicación no debería interrumpir a otras aplicaciones, o ni siquiera al sistema operativo en sí. A esto se le llama **aislamiento de fallas**.

Ningún usuario debería tener acceso o ser capaz de cambiar la información de otro usuario sin permisos. El **aislamiento de fallas** requiere de poder restringir el comportamiento de las aplicaciones tal que no puedan acceder a la capacidad total del hardware. Sin este aislamiento, cualquier bug de cualquier programa podría causar que el disco se vuelva irrecuperablemente corrupto.

**Comunicación**

¿Cómo se logra tener una comunicación tal que se puedan compartir resultados?

La otra cara de la moneda del aislamiento es la necesidad de comunicar entre distintos usuarios y aplicaciones: los programas deben poder comunicarse entre sí. Si el sistema operativo fue diseñado para prevenir bugs, aplicaciones y usuarios maliciosos, se necesita poner límites para esta comunicación: el sistema operativo debe poner estos límites y permitir que estos se crucen de forma controlada según la necesidad y el contexto.

#### Disimulando las limitaciones del Hardware: Sistema Operativo como Ilusionista

El hardware está necesariamente limitado por las restricciones físicas del mismo. Como el SO es el que decide cómo se van a separar los sets de recursos entre varias apps que corren al mismo tiempo, una misma app puede tener distinta cantidad de recursos asignados en dos tiempos distintos, aún cuando se está corriendo en el mismo hardware.

Si bien un uniprocessador sólo puede correr un programa a la vez, la mayoría de los sistemas operativos permite que varias aplicaciones parezcan, para el usuario, estar corriendo al mismo tiempo. Esto se hace a través del concepto de:

##### Virtualización

Provee a una aplicación la ilusión de que tiene una cantidad de recursos que, físicamente, no son reales o no están disponibles. El SO puede presentar a cada aplicación la abstracción de que tiene un procesador enteramente dedicado a ella, aunque a un nivel físico puede haber en realidad un sólo procesador compartido. La mayoría de los recursos físicos pueden ser virtualizados.

Algunos sistemas operativos virtualizan incluso una computadora entera, corriendo un sistema operativo como si fuese una aplicación, por encima de otro sistema operativo. A esto se le llama **máquina virtual**. Al sistema operativo corriendo en la máquina virtual se le llama **sistema operativo huésped** y cree estar corriendo en una máquina real y física, pero esta es una ilusión presentada por el verdadero sistema operativo que corre debajo.

Los sistemas operativos pueden enmascarar o disimular otras limitaciones relacionadas al hardware físico, proveyendo a las aplicaciones de la ilusión de capacidades de hardware que no están realmente físicamente presentes. Por ejemplo, en una computadora con múltiples procesadores compartiendo memoria, cada procesador podría updatear únicamente una única memoria a la vez. El sistema de memoria en el hardware asegura que cualquier update a la misma palabra de memoria va a ser **atómica**.

##### Atómica

Cuando el valor guardado en memoria es el último valor guardado por cualquiera de los procesadores, y no una mezcla de updates de diferentes procesadores.

¿Qué pasa si tenemos dos procesadores que tratan de hacer un update casi al mismo tiempo?

Los resultados pueden ser bastante inesperados y diferentes de lo que pasaría si cada uno hubiese updateado la información en turnos. Idealmente, el programador va a querer tener una abstracción tal que se maneje un update atómico para toda la estructura de datos siendo actualizada, no sólo para la única memory word.

La ilusión de actualizaciones atómicas para las estructuras de datos viene dada por el sistema operativo usando unos ciertos mecanismos especializados provistos por el hardware.

#### Servicios Comunes: Sistema Operativo como Pagamento

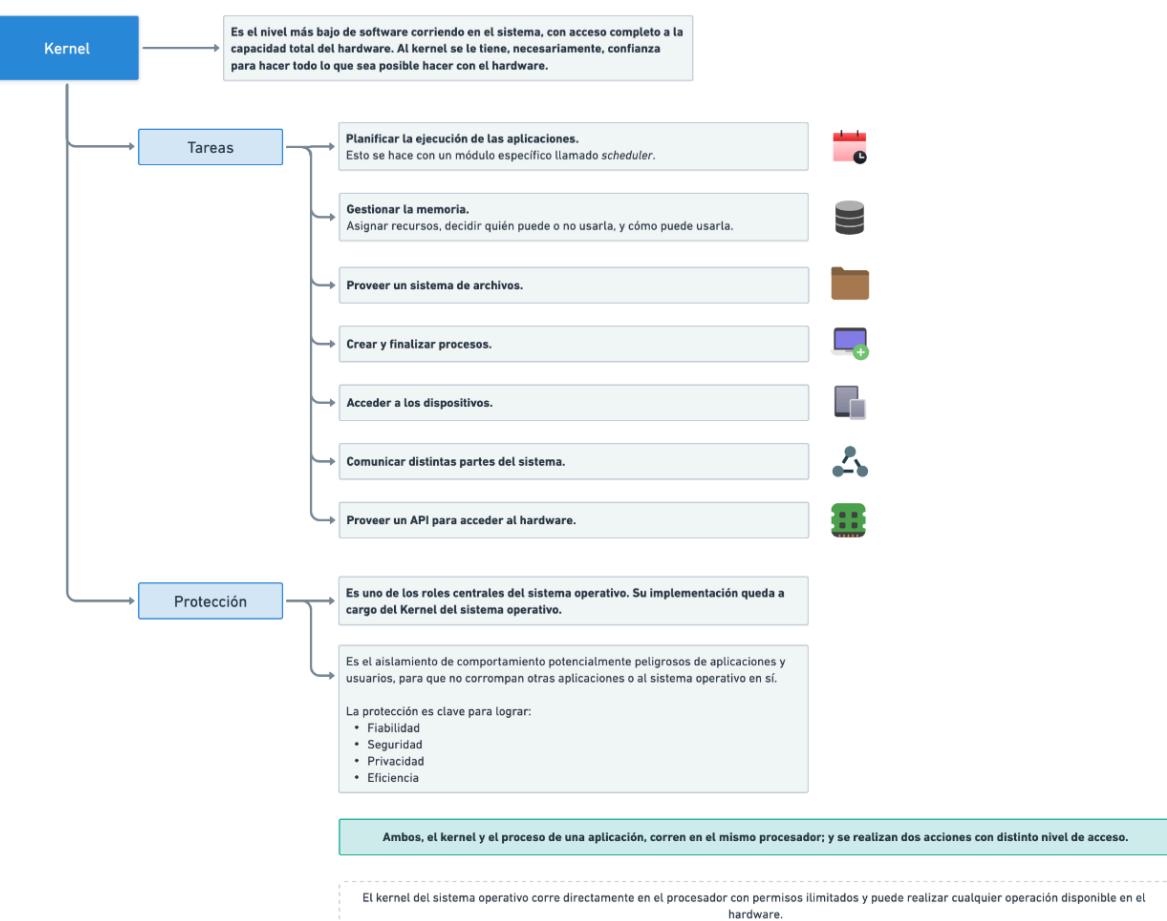
Los sistemas operativos también juegan este tercer y último rol: el de **proveer un set en común de servicios estándares a las aplicaciones**, para simplificar y regularizar su diseño.

Una razón importante para que el sistema operativo sea el que provee de servicios comunes es para facilitar la **compartición entre aplicaciones**: si las aplicaciones van a compartir archivos, por ejemplo, deben poder guardarse en un formato estándar, con un sistema estándar de administración de directorios. De la misma forma, la mayoría de los sistemas operativos provee de una forma estándar para que las aplicaciones se pasen mensajes y comparten memoria, facilitando la compartición.

##### Dispositivos:

Las computadoras pueden llegar a venir con una gran cantidad de diferentes dispositivos, y la mayoría de las aplicaciones van a ser capaces de ignorar estas diferencias usando una interfaz genérica provista por el SO. Sin embargo, va a haber aplicaciones que no puedan ignorarlas y a las que la unidad de disco específica en uno le afecte en funcionamiento.

Para aquellas aplicaciones que puedan operar en un nivel de abstracción más alto, el sistema operativo provee de una capa de interoperabilidad para que ambas aplicaciones -y los dispositivos involucrados- puedan evolucionar de forma independiente sin requerir cambios simultáneos en ambos lados.



### El Concepto de Proceso

1. Un programador crea código en un lenguaje high-level.
2. El compilador convierte ese código en una secuencia de instrucciones a máquina, y las guarda en un archivo llamado ejecutable *image* o *imagen ejecutable*.
3. Para empezar a correr el programa, el SO copia las instrucciones del mismo, junto con su data de la imagen ejecutable, en la memoria física. También se reserva una porción de memoria para el stack de ejecución (*heap*) y otra data o estructuras que puedan ser necesarias.
4. El sistema operativo puede empezar a correr el programa setando el stack pointer y saltando a la primera instrucción.

Un proceso es una instancia de un programa, como un objeto es una instancia de una clase en POO.

El sistema operativo hace un seguimiento de los varios procesos de la computadora usando una estructura de datos llamada **Bloque de Control de Procesos (PCB)**.

#### Bloque de Control de Procesos

Guarda el estado de los procesos (ready, blocked, running). En definitiva, es una tabla de procesos; un arreglo donde cada elemento es un **struct** que guarda los contenidos como:

- El ID
- La memoria
- Los registros
- El estado
- Etc

Que, en definitiva, es toda la información que el SO necesita acerca de un proceso en particular:

- Dónde está guardado en memoria
- Dónde está su imagen ejecutable en disco
- Qué usuario solicitó ejecutarlo
- Qué privilegios tiene el proceso
- Etc

En JOS a los procesos se los llama **environments**. Por lo que de ahora en más, **proceso** y **environment** serán sinónimos.

El proceso tiene varios estados:

1. Running user Mode
2. Ready to Run on Memory
3. Asleep in Memory
4. Ready to Run but Swapped
5. Asleep Swapped
6. Preempt
7. Created
8. Zombie

Y cualquier interfaz de SO debe incluir:

- Una forma de crear un nuevo proceso
- Una forma de destruirlo por la fuerza
- Una forma de esperar a que termine su ejecución
- Una forma de suspender su ejecución por un tiempo y luego reanudarla
- Una forma de saber sobre la situación del proceso y su estado

#### Creación de un Proceso

La única forma de que un usuario cree un proceso en el sistema es llamando a la system call **fork**. El proceso que invoca a fork es llamado **proceso padre**, y el nuevo proceso creado es llamado **proceso hijo**.

Fork entonces:

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

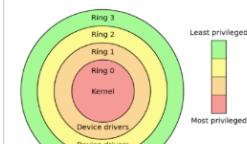
## Kernel

### El Concepto de Proceso (cont.)

La mayoría de las instrucciones que provienen de un proceso son perfectamente seguras, entonces: **¿Podemos modificar al procesador de alguna forma para permitir que instrucciones seguras se ejecuten directamente en el hardware?**

#### Modo Dual

En las arquitecturas x86 se tienen 4 modos de operaciones vía el hardware. Van del 0 al 3 y se denominan **rings**, siendo el ring 0 el nivel más privilegiado (el del **kernel** - modo supervisor) y el 3 el menor privilegiado (el de las aplicaciones - modo **usuario**). La diferencia entre modos está en un bit en el registro de control del procesador.



#### Representación de Modo Dual

Es un único bit en el procesador, en el registro de status, que representa en qué modo se encuentra el procesador ejecutando.

##### Mode

User Mode	El procesador chequea cada instrucción antes de ejecutarla, para verificar que la misma está permitida por ese proceso.
Kernel Mode	El sistema operativo ejecuta con las protecciones apagadas.

Para que el hardware permita al kernel proteger aplicaciones y usuarios entre sí, pero aún dejando al usuario correr código de forma directa en el procesador, necesita como mínimo soportar tres cosas:

#### Instrucciones privilegiadas

Todas las instrucciones potencialmente inseguras están prohibidas cuando se ejecuta en modo usuario.

#### Protección de Memoria

Todos los accesos a memoria fuera de la región de memoria válida de un proceso están prohibidos cuando se ejecuta en modo usuario.

#### Interrupciones con Timer

Independientemente de qué hace el proceso, el kernel debe interrumpir de forma periódica para volver a obtener el control desde el período actual.

Aicionalmente, el hardware debe proveer de una forma segura de transferir el control desde user-mode hasta kernel-mode y viceversa.

#### Instrucciones Privilegiadas

El aislamiento de los procesos sólamente es posible si existe una forma de limitar a los programas que están corriendo en user-mode para que no puedan cambiar su nivel de privilegios.

Los procesos pueden cambiar indirectamente su nivel de privilegio ejecutando instrucciones especiales llamadas **system calls**, para transferir el control al kernel en una ubicación en especial definida por el sistema operativo.

Más allá de convertirse en el kernel en estas ubicaciones fijas, un proceso de una aplicación no tiene permitido cambiar su nivel de privilegio. Tampoco tiene permitido cambiar el set de memoria al que puede acceder y no puede deshabilitar las interrupciones del procesador.

##### Instrucciones Privilegiadas

Son aquellas instrucciones disponibles en el kernel-mode pero no en user-mode. El kernel necesita poder ejecutar estas instrucciones para poder hacer su trabajo: cambiar el nivel de privilegios, ajustar el acceso a memoria, deshabilitar y habilitar las interrupciones.

##### Excepción

Cuando una aplicación trata de acceder a memoria a la que no debería tener acceso para cambiar su nivel de privilegio, se produce una excepción del procesador. Dicha excepción causa que el procesador la transfiera el control al **Excepcion Handler** o al Administrador de Excepciones del Kernel.

#### Memory Protection o Protección de Memoria

La mayoría de los procesadores introduce un nivel de indirección llamado **Virtual Adresses o Direcciones Virtuales**. Con las direcciones virtuales, la memoria de cada proceso empieza en el mismo lugar: **zero**.

##### Base

El lugar de inicio de la región de memoria de un proceso.

##### Bounds

El largo de la región de memoria de un proceso.

Cada proceso piensa que tiene la máquina completa para sí mismo, cuando claramente esa no es la realidad. El hardware traduce estas direcciones virtuales a la ubicación física de los procesos en memoria, por ejemplo, añadiendo el registro de base a cada una de las direcciones virtuales.

Las memorias virtuales permiten, por ejemplo, que el heap y el stack empiecen en puentes separadas de la región de memoria, para que puedan crecer todo lo que sea necesario respecto de las necesidades del programa.

#### Interrupciones del Timer

El aislamiento de los procesos requiere que el hardware provea de una forma para que el kernel periódicamente recupere control del procesador.

Cuando el SO ejecuta un programa a nivel de usuario, el procesador es libre de ejecutar cualquier instrucción no privilegiada que deseé. Para el programa del usuario, parece como si el mismo tuviese control completo sobre el hardware dentro de los límites de su espacio de memoria. Esto, por supuesto, también es una ilusión.

Sin embargo, si una aplicación controla al procesador, el SO por definición **no** está corriendo.

##### Hardware Timer

Casi todas las computadoras incluyen un dispositivo llamado **Hardware Timer**, que puede ser seteado para interrumpir al procesador después de cierto tiempo, que se puede medir tanto en tiempo como en cantidad de instrucciones ejecutadas.

Resetear el timer es una operación privilegiada accesible únicamente por el kernel, para que los procesos a nivel de usuario no puedan, maliciosamente o por error, deshabilitar el timer.

Cuando una interrupción ocurre, el control es transferido por el hardware desde el usuario hacia el kernel.

Otras interrupciones de hardware (como el aviso de que un dispositivo I/O ha completado su trabajo) suelen también generar una transferencia de control del usuario al kernel.

El hecho de que ocurra una interrupción de timer no significa que algo haya salido mal con el proceso. Y en esos casos, el timer simplemente se reiniciará y reanudará la ejecución del proceso que fue pausado, devolviendo el program counter y el valor de los registros a los valores que tenían inmediatamente antes de ser interrumpidos.

## Kernel

### Transferencia de Control de Forma Segura



#### User a Kernel

Transicionar de Kernel a User es "deshacer" la transición de User a Kernel. Por lo que primero vamos a ver este caso y luego, consecuentemente, el camino opuesto.

Hay tres razones por las cuales el kernel puede llegar a querer quitar o tomar control desde un proceso de usuario:

#### Excepciones

Una excepción de un procesador es una condición inesperada causada por el comportamiento de un programa de usuario. En una excepción, el hardware va a parar de ejecutar el proceso actual y empezar a correr un manejador de Excepciones especialmente diseñado en el kernel.

Otra excepción puede hacer que el sistema operativo detenga el proceso y devuelva un código de error al usuario; o puede hacer que el kernel le dé el poder a otro proceso a ser ejecutado (por ejemplo, un debugger).

#### Interrupciones

Una interrupción es una señal asíncrona enviada al procesador indicándole que ocurrió un evento externo que requiere de su atención. El procesador detiene entonces el proceso que está actualmente ejecutándose y empieza a correr el Manejador de Excepciones en el kernel.

Las interrupciones también son usadas para informar al kernel que se ha completado una request de los dispositivos I/O.

Otra alternativa a las interrupciones es el *polling*, que básicamente es el chequeo en loop por parte del kernel de cada dispositivo I/O para ver si ocurrió algún evento que requiera de manejo. Cuando el kernel está haciendo polling, no se puede correr código a nivel de usuario.

Otra fuente de interrupciones son las interrupciones interprocesador. Esto es, cuando existen múltiples procesadores, uno de ellos puede causar una interrupción en cualquiera de los otros. El kernel usa estas interrupciones para coordinar acciones a través del multiprocesador.

#### System Calls

Los procesos de usuario también pueden transicionar al kernel de forma voluntaria, para pedirle al kernel que realice alguna operación de parte del usuario.

Una system call es un procedimiento previsto por el kernel que puede ser llamado desde el nivel de usuario. La mayoría de los procesadores implementan system calls usando una instrucción especial llamada *trap*.

Sin embargo, la instrucción *trap* no es necesariamente requerida; nosotros podríamos voluntariamente causar un *trap* ejecutando cualquier instrucción que cause una excepción, y luego la instrucción *trap* cambia el modo del procesador user a kernel, y comienza a ejecutar en el mismo el Exception Handler.

Para proteger al kernel de programas de usuarios dañinos -o mal utilizados- es esencial que las aplicaciones sólo puedan hacer *trap* para una dirección de memoria predefinida, pero no para saltar de forma arbitraria a cualquier parte del kernel.



#### Kernel a User

Hay distintas causas por las cuales pasaremos de kernel a user:

#### Nuevo Proceso

Para empezar un nuevo proceso, el kernel copia el programa a memoria, setea el program counter para ser la primera instrucción del proceso, setea el stack pointer para ser la base del user stack, y luego cambia a user mode.

#### Reanudar

Esto puede ser para reanudar luego de: una excepción, una interrupción o una system call.

Cuando el kernel termina de manejar la request, reanuda la ejecución del proceso interrumpido; restaurando su program counter y sus registros, y cambiando de nuevo el modo a nivel usuario.

#### Cambiar a otro proceso diferente

En algunos casos, el kernel puede decidir cambiar a otro proceso diferente al que estaba ejecutando antes de la interrupción, excepción o system call.

Y como el kernel eventualmente quiere reanudar el proceso viejo, necesita cambiar el estado del proceso (program counter, registros, etc) en el control block del proceso.

#### Llamada ascendente (User Level)

Muchos sistemas operativos proveen a los programas de usuario la habilidad de recibir de forma asíncrona notificaciones de eventos.

Este mecanismo es muy similar al manejo de interrupciones del kernel, excepto que es a nivel de usuario.



#### Context Switch

Cada proceso tiene un contexto bien definido que comprende la información necesaria para descubrir completamente al mismo. El contexto incluye tres partes:

#### User-Level Context

Consiste en las secciones que forman parte del Virtual Address Space del proceso:

- Text
- Data
- Stack
- Heap

#### Register Context

El contexto de registro consiste en los siguientes componentes:

- Contador de Programa
- Registro de estado del procesador (PS)
- Stack Pointer
- Los Registros de Propósito General.

#### System-level Context

- La entrada en la Process Table Entry
- La u área
- La Process Region Entry, Region Table y Page Table que definen el mapeo de la memoria virtual vs memoria física del proceso.

Para asegurarnos que ningún programa malicioso pueda corromper al kernel cuando transicionamos desde el usuario, necesitamos que el procesador guarde su estado y switchee lo que está haciendo, todo mientras continúa ejecutando instrucciones que pueden llegar a alterar el estado de lo que sea que esté en proceso de guardado.

Un cambio de contexto implica congelar el estado del CPU y reemplazarlo por otro. Además, puede llegar a ocurrir un cambio de privilegio. El cambio de contexto requiere del soporte del hardware y son los dos que ya vimos: de user a kernel (gana prioridad), y de kernel a user ( pierde prioridad).

Si ocurrió una interrupción, no sólo se accede a la tabla de interrupciones para obtener un handler sino también se guardará el estado del proceso interrumpido en el Stack de Interrupciones. Este stack está separado del stack del kernel. Si el kernel está corriendo en un sistema multiprocesador, cada procesador tiene su propio stack de interrupciones en el kernel, para que se puedan manejar múltiples traps y excepciones a través de los múltiples procesadores.



#### Linux

Linux no es un sistema operativo, es sólo un kernel. El resto de los componentes están hechos por terceros.

Algunas características de este kernel son:

- No tiene acceso a la biblioteca estándar de C, ni a los encabezados.
- Está codificado con GNU C.
- El kernel no se protege de sí mismo.
- No puede realizar fácilmente operaciones de punto flotante.
- El kernel tiene una pila fija de tamaño pequeño, no es dinámico.
- Es un único programa ejecutándose en memoria (monolítico).
- El kernel tiene interrupciones sincrónicas, es preemptive (propiedad de algunos kernels en donde la CPU puede interrumpir en el medio de una ejecución de código de kernel y asignar otras tareas) y admite Symmetrical Multiprocessing.

#### Tipos de Kernel:

##### Kernel Monolítico

Es un programa único (proceso) que se ejecuta continuamente en memoria, intercambiándose con los procesos de usuario.

##### Micro Kernel

El kernel sigue existiendo pero sólo implementa funcionalidad básica en el ring 0. Otros servicios se implementan en ring 1 o ring 2, pero no es imprescindible que se ejecuten en modo kernel exclusivamente.

## Kernel

### Transferencia de Control de Forma Segura (cont.)

#### Linux (cont.)

##### Proceso de Inicio

En el proceso de inicio comienza el booteo, denominado bootstrap. Esta parte depende del hardware, ya que acá se realizan chequeos del mismo y se carga el bootloader, 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.

Una tabla de vectores de interrupción (IVT) es una estructura de datos que asocia una lista de handlers de interrupciones con una lista de solicitudes de interrupción. Cada entrada de la tabla, llamada vector de interrupción, es la dirección de un handler.

3. El BIOS genera una interrupción (19), la cual hace ejecutar el servicio de interrupciones.

→ Cuando el sistema se inicia, éste carga el Kernel desde el disco, usando un procedimiento especial llamado bootstrapping.

- El kernel inicializa el sistema y setea el ambiente para correr procesos.
- El kernel crea algunos procesos iniciales, los cuales a su vez crean otros procesos.
- Una vez cargado en memoria, el kernel permanece allí hasta que se apague la computadora.
- El kernel administra los procesos y provee de varios servicios.

##### ↑ Fase de Inicio del Kernel

La función de arranque para el kernel establece la gestión de memoria (tablas de paginación y paginación de memoria), detecta el tipo de CPU y cualquier funcionalidad adicional como capacidades de punto flotante. Después cambia las funcionalidades del kernel para arquitectura no específica de Linux, a través de una llamada a la función `start_kernel()`.

El núcleo **inicializa los dispositivos, monta el sistema de archivos raíz**, y ejecuta `init`, que es designado como el primer proceso ejecutado por el sistema (PID = 1).

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

El trabajo de `init` es "conseguir que todo funcione como deberá ser" una vez que el kernel está completamente en funcionamiento. En esencia, establece y opera todo el espacio de usuario. Esto incluye:

- La comprobación y montaje de sistemas de archivos
- La puesta en marcha de los servicios de usuarios necesarios
- Cambiar al entorno de usuario cuando el inicio del sistema se ha completado.

#### ⚙️ → 🖥️

### De Programa a Proceso

#### El Programa

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

#### Compilación



#### ⇒ De Programa a Proceso

El SO, más precisamente el kernel, se encarga de:

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

#### Intro a x86

##### Ley de Moore

Ley empírica que se ha podido constatar hasta hoy en día. Dice: "aproximadamente cada dos años se duplica el número de transistores en un microprocesador por unidad de área".

##### Ley de Bell

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.

##### El Stack

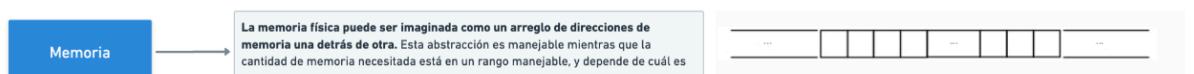
En la arquitectura x86 los programas utilizan el stack para soportar la llamada a funciones. La máquina usa el stack para:

- pasar 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**, y su estructura está delimitada por 2 valores: el stack frame pointer (%EBP) y el stack pointer (%ESP).

Asimismo, existen distintos tipos de registros:

- Multipropósito
- De propósito especial
- De segmento



Antes

El sistema operativo era más o menos un conjunto de funciones o rutinas que se alojaban en la memoria, empezando en la dirección física 0. Después existía un único programa o proceso que se encontraba en la memoria física de la computadora. Esta memoria era el resto no utilizado por el sistema operativo.

### Multiprogramación y Time Sharing

#### Multiprogramación

Después de un tiempo, las computadoras evolucionaron a un modelo en el cual más de un proceso estaba preparado para ser ejecutado en algún determinado momento, y el sistema intercalaba dicha ejecución según la circunstancia. Esto significa que múltiples procesos están listos para ser ejecutados en un determinado tiempo, y el SO es el que decide, en base a ciertas políticas de planificación o scheduling, qué proceso es el siguiente.

#### Time Sharing

- Se empieza luego a dar una mayor importancia a la noción de interacción del usuario con la computadora, y se empieza a implementar el time sharing:
  - Se tiene un proceso corriendo por un determinado **time slice** o **quantum** de tiempo, al cual se le da acceso a toda la memoria y recursos.
  - Cuando su período de tiempo asignado termina, se graba su estado en algún lugar (como por ejemplo, el disco).
  - Se carga otro proceso.
  - Se ejecuta por un rato, y así sucesivamente.

El problema de este método es que es demasiado lento, sobre todo cuando la memoria de la computadora comienza a crecer.

Teniendo en cuenta que hacer un cambio de contexto a nivel registros es relativamente rápido, no tiene sentido grabar información de memoria en disco como mencionamos arriba. Es más fácil mantener los procesos en memoria mientras se realizan los cambios de quien se está ejecutando y, de esta forma, se permite implementar de forma eficiente el **time sharing**.

La idea es, entonces, darle a cada proceso una pequeña parte de la memoria física.

Vamos a ver más de esto en **Scheduling**.

### El espacio de Direcciones o Address Space

La idea del espacio de direcciones y/o abstracción de memoria surge inevitablemente de querer crear un mecanismo que permita que la memoria física de una computadora sea utilizada de forma fácil y eficiente.

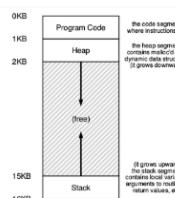
El **address space** de un proceso contiene todo el estado de la memoria de un programa en ejecución.

Así, decimos que:

- El código del programa debe estar alojado en la **memoria**, y vive en lo alto del espacio de direcciones, comenzando desde 0. Como es estático, no necesitará más espacio en tiempo de ejecución.
- Se usa el **heap** para reservar memoria de forma dinámica.
- El programa, mientras se está ejecutando, usa el **stack** para mantener registro de dónde se encuentra en la cadena de llamadas a funciones o procedimientos; para reservar espacios para variables locales, pasar parámetros y devolver valores.
- Tanto stack como heap pueden crecer o achicarse.
- Existen por supuesto otras cosas en la memoria: variables estáticas, constantes, etc.

Cuando se describe de esta forma al espacio de direcciones, lo que se está describiendo es la **abstracción** que el sistema operativo provee para ejecutar el programa (porque en realidad, el programa no se encuentra en la dirección de memoria física 0).

El sistema operativo está **virtualizando memoria**, porque el programa que se está ejecutando piensa que está cargado en una dirección de memoria particular y que tiene potencialmente un espacio de direcciones muy grande, cuando la realidad es bastante diferente.



#### Conclusiones

El sistema operativo debe implementar la virtualización de forma tal que sea invisible al programa que se está ejecutando. El programa debe comportarse como si estuviese alojado en su propia área de memoria física privada. Sin embargo, por detrás, el sistema operativo y el hardware hacen todo el trabajo de multiplexar memoria a lo largo de diferentes procesos y por ende, se implementa así la ilusión.

También se debe ser **eficiente**: el SO debe esforzarse porque la virtualización sea eficiente en términos de tiempo y espacio.

Por último, se debe cuidar la **protección**: el SO debe asegurarse de proteger a los procesos entre sí y de proteger al sistema de dichos procesos; cada proceso debe ejecutarse en su propio caparazón aislado y seguro de los errores o intentos maliciosos que puedan ocurrir por parte de otros procesos.

### El API de Memoria

Es importante saber cuáles son las funciones que permiten obtener y liberar memoria, así como los errores más comunes al utilizar estas herramientas.

#### Tipos de Memoria

En un programa en ejecución, existen dos tipos de memoria:

##### Stack

La reserva y liberación de espacio en el **stack** es manejada **implícitamente** por el compilador. La declaración de memoria en el stack en C es sencilla: se declara una variable de algún tipo y el compilador automáticamente le reserva un espacio en memoria al compilar el código.

Una vez llamada la función en donde se usa dicha variable, el compilador va y busca esa memoria por el programador.

##### Heap

Es un tipo de memoria que es obtenida y liberada **explícitamente** por el programador (por ejemplo, con `malloc`). El usuario es responsable de pedir y liberar la memoria almacenada en el heap.

#### Funciones de Manejo de Memoria

En un programa en ejecución, existen dos tipos de memoria:

##### `malloc`

Devuelve un puntero `void*` a un bloque de memoria de tamaño, como mínimo, igual al que se había solicitado. Si falla, devuelve `NULL`. No inicializa el bloque de memoria (viene con basura).

##### `free`

El puntero recibido debe haber sido inicializado con `malloc`, `calloc` o `realloc`. De lo contrario, tiene comportamiento indefinido. No avisa si algo salió mal.

## Memoria

### El API de Memoria (cont.)

#### Funciones de Manejo de Memoria (cont.)

Por dentro, las funciones `malloc` y `free` hacen llamadas a system calls que terminan de ejecutar el pedido o la liberación de memoria. Estas system calls son `brk` y `sbrk`

##### brk

Lo que hace esta system call (break) es sustituir la dirección del brk, que apunta al tope del heap. Para esto, recibe una nueva dirección de tipo `void*`. Entonces, básicamente, se extiende el heap apuntando a la nueva dirección del fin del mismo.

##### sbrk

El parámetro en este caso es la cantidad de bytes, y lo que hace es mover o achicar el heap por esa cantidad de bytes particular.

### Address Translation

#### Definición de Address Translation

Es la conversión desde la dirección de memoria a la cual el programa cree que está referenciando; al espacio físico de esa celda de memoria.

Existen dos puntos importantes a la hora de virtualizar memoria:

- La Flexibilidad
- La Eficiencia

Para llegar a ellos, un buen mecanismo de virtualización de memoria debe ser lo más flexible y eficiente posible.

Esto se logra a través de una técnica llamada **Hardware-Based Address Translation** o, simplemente, **Address Translation**.

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

Entonces, en todo y para cada una de las referencias a memoria, el hardware realiza un **address translation** para redireccionar las referencias desde la **memoria de la aplicación** hacia las posiciones reales en la **memoria física**.

El hardware por si sólo no puede virtualizar memoria, ya que este sólo provee un mecanismo de bajo nivel para poder hacerlo eficientemente.

El sistema operativo debe entonces involucrarse en los puntos claves para:

- Setear al hardware de forma correcta para que esta traducción se de lugar.
- Gerenciar la memoria, manteniendo información de en qué lugar hay áreas libres y en qué lugar hay áreas en uso. Para eso debe mantener registro:
  - De qué parte está libre
  - De qué parte está en uso
  - Del control de la forma en la cual la memoria está siendo utilizada.
- Intervenir de forma criteriosa, como manteniendo el control sobre toda la memoria usada.
- Determinar si la traducción se ha realizado correctamente.

La idea es, como siempre, crear una ilusión:

- Que el programa tenga su propia memoria privada donde reside su propio código y datos.
- Pero que detrás de esa realidad virtual, existan varios programas compartiendo memoria al mismo tiempo que la CPU los cambia de estado, ejecutando un único programa y pasando luego al siguiente.

#### Cosas Provistas por el Address Translation

1. **Aislamiento de procesos:** protegiendo al kernel y otras aplicaciones al limitar la memoria accesible por una aplicación.
2. **Comunicación entre procesos:** haciendo que los procesos comparten una región de memoria en común.
3. **Segmentos de código compartidos:** instancias del mismo programa pueden compartir las instrucciones del programa, reduciendo la cantidad de memoria que se necesita. También distintos programas pueden compartir las librerías comunes.
4. **Inicialización de un programa:** se puede empezar a ejecutar un programa antes de que todo su código sea cargado hacia la memoria desde el disco.
5. **Alojamiento de memoria dinámica eficiente:** un proceso puede crecer su heap o su stack, y se puede usar address translation para pedirle al kernel que aloje memoria para esas razones.
6. **Manejo de la caché:** el SO puede organizar cómo quiere que los programas estén posicionados en la memoria física para mejorar la eficiencia de la caché, usando algo llamado *page coloring*.
7. **Debugging de Programas:** el SO puede usar memory translation para prevenir que un programa bugged pise su propia región de código. Pudiendo atrapar los errores de puntero antes, se hace mucho más fácil debuggear. También se usa para instalar data breakpoints.
8. **Dispositivos I/O eficientes:** permite a la data ser transferida de forma segura entre aplicaciones de user-mode y dispositivos I/O.
9. **Archivos mapeados en memoria:** una abstracción conveniente y eficiente para muchas aplicaciones es mapear archivos en el address space, para que los contenidos del archivo puedan ser directamente referenciados con instrucciones del programa.
10. **Virtual memory:** el SO puede proveer a las aplicaciones de la abstracción de que se le está dando más memoria de la que está presente de forma real en la computadora.
11. **Checkpoint and restart:** el estado de un programa que corre durante mucho tiempo puede ser periódicamente guardado en un checkpoint para que, en caso de que falle, pueda ser restaurado desde ese último punto.
12. **Persistent data Structures:** el SO puede proveer de la abstracción de que existe una región de memoria persistente, donde cambios a las estructuras de datos en esa región sobreviven a crashes del programa y el sistema.
13. **Migración de procesos:** un programa siendo ejecutado puede, de forma transparente, ser movido de un servidor a otro.
14. **Control de Flujo de Información:** una capa extra de seguridad verifica que un programa no esté mandando información privada a terceros.
15. **Memoria compartida distribuida:** se puede, de forma transparente, convertir una red de servidores en una memoria compartida a gran escala.

💡 Si se quiere ver un ejemplo de memoria, se puede ver [acá](#).

### Dynamic Reallocation o Memoria Segmentada

#### Base and Bound (Base y Segmento) o Dynamic Reallocation

Como vimos en el capítulo de Kernel y Booteo:

##### Base

Es el lugar de inicio de la región de memoria de un proceso.

##### Bounds

Es el largo de la región de memoria de un proceso.

💡 Como primer acercamiento, vamos a ver una primera implementación de Dynamic Reallocation, introducida en las primeras máquinas con time sharing que se introdujeron hacia fines de los años 50. Sólo se necesitan estos dos registros de hardware dentro de cada CPU. Este va a permitir que el **address space** pueda ser ubicado en cualquier lugar deseado de la memoria física, y se hará mientras el sistema operativo pueda asegurarse que el proceso sólo puede acceder a su propio address space.

En esta configuración, cada programa es escrito y compilado como si fuera cargado en la dirección física 0. A su vez, cuando se inicia la ejecución del programa, el SO decide en qué lugar va a cargar el mismo, y, para hacerlo, setea el registro base en un determinado valor. A partir de ahora, cuando cualquier referencia es generada por el proceso, además es traducida por el procesador de la siguiente manera:

$$\text{physical address} = \text{virtual address} + \text{base}$$

Cada referencia de memoria generada por el procesador es una dirección virtual, y cada vez que se hace referencia a esta dirección, el hardware tiene que sumar el contenido del registro base y su resultado es la dirección física que tiene que ser utilizada en la memoria del sistema.

💡 Es decir, es como sumar un offset, sabiendo que el base es donde comienza la región y que todo se va a guardar de forma secuencial.

## Memoria

### Dynamic Realocation o Memoria Segmentada (cont.)

#### Base and Bound (Base y Segmento) o Dynamic Realocation (cont.)

El proceso que transforma la **virtual address** en la **physical address** es exactamente la técnica conocida como **address translation**. Esto es simplemente la traducción de una dirección virtual en una física, independientemente de la implementación de esta traducción.

Debido a que esta ubicación sucede en **run-time** y debido a que se puede mover el address space incluso una vez que el proceso empezó a ejecutarse, la técnica es referenciada como **realocación dinámica**.

Sin olvidarnos del registro **bound**, podemos decir que ayuda con la protección: el procesador, antes que nada, va a checar que la referencia de memoria esté dentro de los límites del address space. Si un proceso genera una dirección virtual que es mayor que los límites o una dirección que es negativa, la CPU va a generar una excepción y el proceso va a terminarse.

El punto es que los límites están ahí para asegurarse que todas las direcciones generadas por el proceso sean legales y estén dentro del límite del mismo.

El registro base y el límite son estructuras de hardware mantenidas dentro del mismo procesador o chip. Antiguamente, a esa parte del procesador -que ayudaba a la address translation- se llamaba **memory management unit (MMU)**. A medida que se fueron sofisticando las técnicas de gerenciamiento de memoria, se fueron agregando más cosas a lo que se llamaba MMU.

El **bound register** puede ser definido de dos formas diferentes:

1. El registro mantiene el tamaño del address space, entonces el hardware chequea la dirección virtual contra el bound register, sumándole primero el registro base.
2. El registro mantiene la dirección física del fin del espacio de direcciones.

#### Address Translation con Tabla de Segmentos

El problema de la técnica anterior es que se tiene un sólo registro base y un sólo segmento, y se debe almacenar la memoria toda junta de forma seguida. La mejora a este método es mediante la aplicación de un pequeño cambio: en vez de tener un sólo registro límite, se tiene un arreglo de pares **[registro base, segmento]** por cada proceso. Cada entrada en el arreglo controla una porción del virtual address space. La memoria física de cada segmento es almacenada continuamente, pero distintos segmentos pueden estar ubicados en distintas partes de la memoria física.

Una dirección virtual tiene dos componentes:

- Un número de segmento
- Un offset de segmento

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

En una dirección virtual usando esta técnica, los bits de más alto orden son utilizados como índice en la tabla de segmentos. El resto se toma como offset y es sumado al registro base y comparado contra el registro bound. El número de segmentos depende de la cantidad de bits que se utilizan como índice. Ver ejemplo acá.

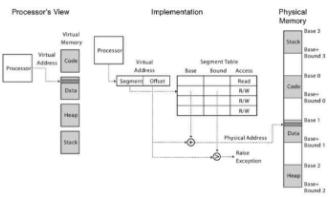
El error de **Segmentation Fault** sucede cuando se quiere direccionar una posición fuera del espacio direccional. Este error aún se utiliza incluso en SO que no usan segmentación.

Algunas notas:

1. El stack crece hacia atrás o backward, por ende, el hardware necesita saber sobre eso en un bit de información.
2. A veces es necesario poder compartir segmentos entre varios procesos, y para eso el hardware proporciona un bit que permite saber si se puede ejecutar, escribir o leer un segmento por varios procesos a la vez. Estos se llaman **protection bits**.
3. Todo lo que hace el hardware en estos casos, en realidad lo está haciendo la Memory Management Unit.
4. La segmentación de grano fino consiste en tener muchos segmentos pequeños y la de grano grueso en tener pocos segmentos grandes.

El problema de la segmentación es la **fragmentación**, precisamente la fragmentación externa:

 La **fragmentación externa** ocurre cuando la memoria se divide en particiones de tamaño variable según el tamaño de los procesos. Los espacios no utilizados que se forman entre fragmentos de memoria no contiguos son demasiado pequeños para dar servicio a un nuevo proceso, y a esto se le denomina **fragmentación externa** [más info acá].



Process C7 needs 50KB memory space

Assigned Space

Fragment 40 KB

Assigned Space

Fragment 10 KB

Assigned Space

Fragment 5 KB

Assigned Space

Fragment

Used Space

Assigned Space

## Memoria

### Memoria Paginada (cont.)

#### Translation Multilevel

Se usa para implementar un mecanismo eficiente de paginación. Un árbol o un hash resultan ser más apropiados que un arreglo. Muchos sistemas usan técnicas de address translation basados en árboles. Estas técnicas soportan:

- 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.

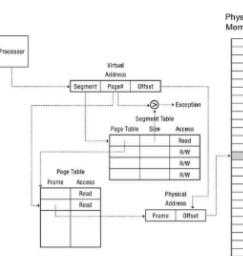
Todos los sistemas multilevel usan paginación en el nivel más bajo del árbol, pero la principal diferencia entre ellos es cómo se llega a la page table a nivel de las hojas del árbol: segmentación + paginación, múltiples niveles de paginación o segmentación, + múltiples niveles de paginación.

#### Segmentación Paginada

Dos niveles de un árbol. La memoria está segmentada, pero cada entrada en la tabla de segmentos apunta a una tabla de páginas, que a su vez apunta a la memoria correspondiente a ese segmento.

La virtual address tiene 3 componentes:

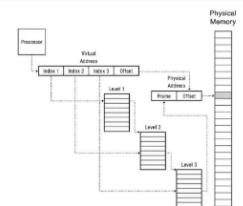
- Un número de segmento
- Un número de página virtual en ese segmento
- El offset



Physical Memory

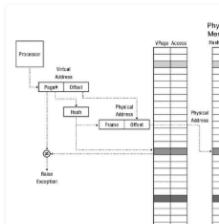
#### Address Translation con Tres Niveles de Page Tables

Este es un ejemplo conocido como multi-level paging. Implementa múltiples niveles de page table.



Physical Memory

#### Address Translation con Tabla de Hash por Software



Physical Memory

#### Hacia una eficiente Address Translation

Uno de los problemas del address translation reside en la **velocidad** de la traducción: para ello se utilizan técnicas que mejoran la velocidad de esta traducción. Para mejorar el address translation, se utiliza un mecanismo de hardware llamado **Translation-Lookaside Buffer**, o también conocido como **TLB**. La TLB es parte de la MMU y es simplemente un mecanismo de cache de las traducciones más utilizadas entre los pares virtual to physical address. Por ende, un mejor nombre para este mecanismo podría ser **address translation cache**. Por cada referencia a la memoria virtual, el hardware primero chequea la TLB para ver si esa traducción está guardada ahí. Si es así, la traducción se hace rápidamente sin tener que consultar a la page table (la cual tiene todas las traducciones).



Podés ver un ejemplo que tomaron en el parcial del 2C del 2022 [acá](#).

A nivel procesador, este proceso tiene una ganancia muy buena en prestaciones, ya que a nivel hardware está implementado en **memoria estática on chip** que está localizada muy cerca del procesador.

#### Consistencia de la TLB

Cada vez que se introduce un cache en el sistema, se necesita considerar la forma de asegurar la consistencia del cache con los datos originales cuando las entradas en el mismo son modificadas. Una TLB no es la excepción.

Para una ejecución correcta y segura del problema, el sistema operativo tiene que asegurarse que cada programa ve su propia memoria y la de nadie más.

Se consideran:

- **Context Switch**: las direcciones virtuales del viejo proceso ya no son más válidas y no deben ser válidas para el nuevo proceso. Frente a un context switch, se necesita descartar el contenido de la TLB (hacer un **flush**). La TLB contiene al **id** del proceso que produce cada transacción.
- **Reducción de Permisos**: cuando el sistema operativo modifica una entrada en una page table, no se provee consistencia por hardware para la TLB. Mantener la TLB consistente con la Page Table es responsabilidad del SO.
- **TLB Shutdown**: En un S0 multiprocesador, cada uno puede tener cachear una copia de una transacción en su TLB. Para seguridad, 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. Para esto, se manda una interrupción a cada procesador y se le pide que esa entrada de la TLB sea eliminada.



### Multiprogramación

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 ejecución de instrucciones del procesador, podía parecer infinito; y la CPU parecía inactiva por un gran período de tiempo.

#### Utilización de la CPU

Se asume que el **20% del tiempo de ejecución** de un programa es sólo cálculo y el **80% son operaciones de entrada y salida**, con tener 5 procesos en memoria se estaría utilizando el 100% de la CPU. Siendo un poco más realista, se supone que las operaciones de E/S son **bloqueantes** (una operación de lectura a disco tarda 10 miliseg y una instrucción registro-registro tarda 1-2 nanoseg), es decir, paran el procesamiento hasta que se haya realizado la operación de E/S.

Entonces, el cálculo es más realista si se supone que un proceso gasta una **fracción p** bloqueado en E/S. De esta manera, si tenemos **n procesos** esperando para hacer operaciones de entrada y salida, la probabilidad de que los **n procesos** estén haciendo E/S es  $p^n$ .

Por ende, la probabilidad de que se esté ejecutando algún proceso es  $(1-p^n)$ , esta fórmula es conocida como utilización de CPU.

#### Multiple Fixed Partitions

**Multitasking with a Fixed number of Tasks (MFT)**: se tenían varias particiones de memoria, cada una de tamaño fijo, que se establecían cuando el sistema operativo se instalaba o podían ser redefinidas por el operador.

#### Multiple Variable Partitions

**Multitasking with a Variable number of Tasks (MvT)**: trataba a toda la memoria no utilizada por el sistema operativo como un único gran espacio desde el cual se podían asignar "regiones" contiguas tanto como lo requieren un número limitado de aplicaciones y programas simultáneamente.



### Números y Workload

El **workload** es la carga de trabajo de un proceso corriendo en el sistema.

Determinar cómo se calcula el **workload** es fundamental para determinar partes de las políticas de planificación. Cuanto mejor es el cálculo, mejor es la política. Las suposiciones que se harán para el cálculo del **workload** son **más que irreales**.

- Los supuestos sobre los procesos o **jobs** que se encuentran en ejecución son:
  1. Cada proceso se ejecuta la misma cantidad de tiempo.
  2. Todos los jobs llegan al mismo tiempo para ser ejecutados.
  3. Una vez que empieza un job, sigue hasta completarse.
  4. Todos los jobs usan **únicamente** CPU.
  5. El tiempo de ejecución (run-time) de cada job es conocido.

#### Métricas de Planificación

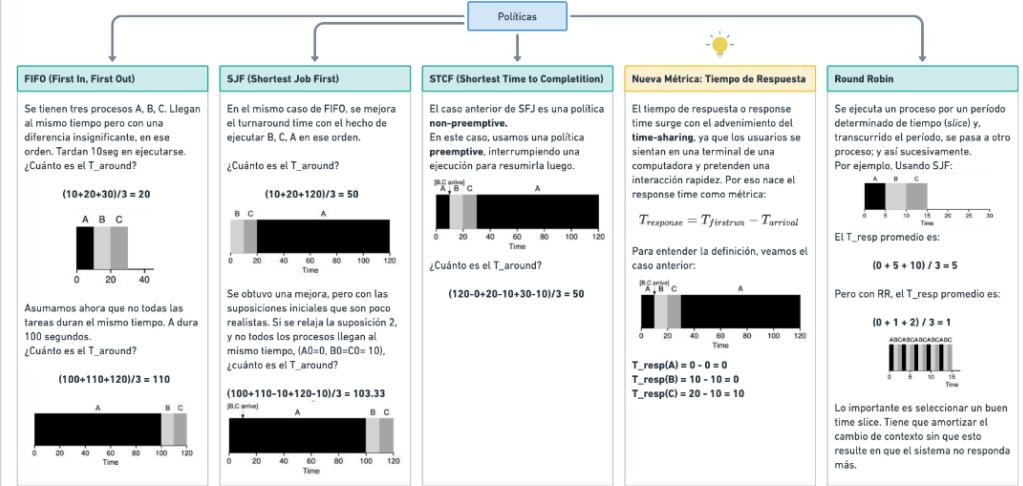
Para poder hacer algún tipo de análisis, se debe tener algún tipo de métrica estandarizada para comparar las distintas políticas de planificación o scheduling. Bajo estas premisas, por ahora, para que todo sea simple, se utilizará una única métrica llamada **turnaround time**. Que se define como el tiempo en el cual el proceso se completa, menos el tiempo de arribo al sistema:

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

Debido a la regla 2, el **T\_arrival = 0**.

Hay que notar que el turnaround time no es una métrica que mide performance.

### Políticas para Sistemas Mono-Procesadores



### Multi-Level Feedback Queue (MLFQ)

MLFQ intenta atacar principalmente dos problemas:

#### Turnaround time

Intenta optimizar el turnaround time, que se realiza mediante la ejecución de la tarea más corta primero. Desafortunadamente, el sistema operativo nunca sabe a priori cuánto va a tardar en correr una tarea.

#### Response Time

MLFQ intenta que el planificador haga sentir al sistema con un tiempo de respuesta interactivo para los usuarios, por ende, minimizar el response time. Desafortunadamente los algoritmos como Round-Robin reducen el **response time** pero tienen un terrible turnaround time.

#### Reglas Básicas

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

En un determinado tiempo, una tarea que está lista para ser corrida está en una única cola. MLFQ usa las prioridades para decidir cuál tarea debería correr en un determinado tiempo t0: la tarea con mayor prioridad (la primera tarea en la cola más alta) será elegida para ser corrida.

## Scheduling

### Multi-Level Feedback Queue (MLFQ) (cont.)

#### Reglas Básicas (cont.)

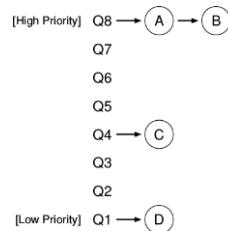
Dado el caso que exista más de una tarea con la misma prioridad, entonces se utilizará el algoritmo de **Round Robin** para planificar estas tareas entre ellas. Las reglas de MLFQ son:

- 1 Si la prioridad de A es mayor que la prioridad de B, A se ejecuta y B no.
- 2 Si la prioridad de A es igual a la prioridad de B, A y B se ejecutan en **Round Robin**.
- 3 Cuando una tarea entra en el sistema, se pone con la más alta prioridad.
- 4 Una vez que una tarea usa su asignación de tiempo en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU) su prioridad se reduce (por ejemplo, baja un nivel en la cola de prioridad).
- 5 Despues de cierto periodo de tiempo 5, se mueven las tareas a la cola con más prioridad.

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

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

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



### Planificación: Proportional Share

Este algoritmo es también conocido como **fair-share**. Este es basa en un concepto simple: en vez de optimizar el **turnaround** o el **response time**, el scheduler intentará garantizar que cada tarea obtenga cierto porcentaje de tiempo de CPU.

El concepto también es conocido como planificador por lotería: cada tanto se realiza un sorteo para determinar qué proceso tiene que ejecutarse a continuación. Por lo tanto, aquellos procesos que deben ejecutarse más seguido, tienen más probabilidad de ganar el sorteo.

De esta forma, la prioridad se distribuye en forma de **tickets de lotería**, que son utilizados para representar cuánto se comparte un determinado recurso para un determinado proceso. El porcentaje de los boletos respecto del total de boletos entregados a todos los procesos es el porcentaje de cuánto va a poder hacer uso del recurso en cuestión.



Al mismo tiempo, cada proceso puede tener distinto tipo de **cambio** de boletos, para subdividir el proceso en subprocessos y asignarle prioridades a cada uno de ellos. Este tipo de cambio es automáticamente transformado luego al tipo de cambio global del scheduler.



Además, el proceso puede transferir sus boletos a otro proceso, es decir, las prioridades son variables. Por otro lado, en un ambiente en el cual los procesos confían entre ellos, un proceso podría aumentar o disminuir la cantidad de boletos que posee, de forma temporal. A esto se le llama **inflación**.

### Planificación Avanzada: Planificación Multiprocesador.

Para entender de planificación multiprocesador, hay que entender primero la diferencia fundamental entre **hardware monoprocesador y multiprocesador**.

La diferencia se centra básicamente alrededor de un tipo de hardware llamado cache, y de qué forma exactamente son compartidos los datos en la cache a través de los multiprocesadores.

En un sistema con un único CPU, hay una jerarquía del hardware de caché que ayuda al procesador a correr los programas más rápidamente. Las caches son pequeñas y rápidas memorias que mantienen copias de datos comúnmente utilizados.

En la memoria principal se mantienen todos los datos del sistema, pero el acceso es más lento.

Entonces las cache se basan en la noción de **localidad**, de la cual hay dos tipos:

#### Localidad Temporal

Cuando cierta cantidad de datos son accedidos, es muy probable que sean accedidos otra vez en un futuro cercano. Entonces, cuando la misma posición de memoria es referenciada muchas veces en un lapso de tiempo muy corto, se llama **localidad temporal**.

#### Localidad Espacial

Cuando un programa accede a una dirección X, es muy probable que necesita volver a acceder cerca de X. La **localidad espacial** entonces tiene que ver con la referencia de variables que están en posiciones contiguas de memoria.

#### Coherencia de Caché

Problema que surge cuando las cachés de los distintos CPUs no son coherentes entre sí, por lo que se accede demasiado a memoria y el proceso se vuelve lento.

#### La solución Bus Snooping.

Cada caché pone atención a las actualizaciones de memoria mediante el bus que las conecta a ellas y a la memoria principal. Cuando una CPU ve que se actualizó un dato en una caché, va a dar cuenta del tal cambio e invalidar o actualizar su copia.

### Afinidad de Cache

- Cuando un proceso corre sobre una CPU en particular, va construyendo un cachito del estado de sí mismo en las cachés de esa CPU.
- Entonces, la próxima vez que el proceso se ejecuta, sería bastante interesante que se ejecutara en la misma CPU, ya que se va a ejecutar más rápido si parte de su estado en esa CPU.
- Si, en cambio, se ejecuta el proceso en una CPU diferente cada vez, el rendimiento del proceso va a ser peor, ya que tendrá que volver a cargar su estado o parte del mismo cada vez que se ejecute.

Por lo tanto, un planificador multiprocesador debería considerar la afinidad del cache cuando toma sus decisiones de planificación, quizás prefiriendo mantener a un proceso en un CPU si lo es más conveniente.

#### Planificación de Cola Única

La forma más fácil de tener un Scheduler en un sistema Multiprocesador es reutilizando el marco de trabajo básico que se tiene en un Scheduler de monoprocesadores.

## Scheduling

### Afinidad de Cache (cont.)

#### Planificación de Cola Única (cont.)

Se ponen entonces todos los trabajos que tienen que ser planificados en una **única cola**, que se llama **Single Queue Multiprocessor Scheduling**, o SQMS.

Esta forma de plantear la planificación tiene la ventaja de la simplicidad: no requiere demasiado trabajo agarrar a la mejor tarea, ejecutarla y adaptarla para que trabaje con más de una CPU.

Pero, hay limitaciones:

1. No es Escalable.
2. Se debe insertar algún tipo de bloqueo en su código fuente, para asegurarse que cuando SQMS accede a una única cola, se obtiene un resultado correcto. Desafortunadamente, el bloqueo reduce mucho la performance, y se pierde más tiempo sobrecargándose en el bloqueo y menos en lo que debería estar haciendo.
3. Cada tarea termina saltando de CPU en CPU, haciendo exactamente lo opuesto que recomienda la afinidad de caché.

### Multi Queue Planification

Debido a los problemas que tiene el **Single Queue Scheduler**, se terminó creando el **Multi-Queue Multiprocessor Scheduling**, o MQMS.

El esquema básico de la planificación consiste en múltiples colas. Cada cola va a seguir una determinada disciplina de planificación, por ejemplo, round robin. Cuando una tarea entra en el sistema, ésta se coloca exactamente en una única cola de planificación, de acuerdo con alguna heurística. Esto implica que es esencialmente planificada de forma independiente.

- MQMS tiene la ventaja sobre SQMS que es **escalable**. A medida que las CPUs van creciendo, también lo hacen las colas, lo que conlleva a que los locks y las caches no sean un problema.
- MQMS provee de una **afinidad de cache**, es decir, las tareas intentan mantenerse en la CPU en la que fueron planificadas.

El único problema de MQMS es el **load imbalance**.



El **load imbalance** se da cuando una CPU queda ociosa frente a las demás que están sobrecargadas. Este problema se resuelve moviendo las tareas de un CPU a otro, y se llama **migración**. Mediante la migración, se obtiene un verdadero balance de carga.

### Linux: Completely Fair Scheduler (CFS)

Linux implementa el Fair Share de forma altamente eficiente y escalable. Lo hace por medio de:

1. Su diseño
2. El uso inteligente de estructuras de datos para esa tarea.

#### Modo de Operación Básico

Divide de forma justa la CPU entre todos los procesos que están compitiendo por ella. Esto lo hace con una técnica para contar llamada **virtual runtime**, que no es más que el runtime **normalizado** por el número de procesos en estado **runnable**.

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

También hay un punto de tensión entre performance y equitatividad:

1. Si el CFS cambia de proceso en tiempos muy pequeños, estará garantizado que todos los procesos se ejecutan, pero a costa de pérdida de performance por tener demasiado context switches.
2. Si el CFS cambia de proceso pocas veces, la performance es buena, pero no hay tanta equitatividad.

Esto se maneja mediante varios parámetros de control:

#### Sched\_latency

Determina por cuánto tiempo un proceso tiene que ejecutarse antes de considerar el cambio de contexto. El valor típico es de 48ms, y el CFS divide este valor por el número de procesos para determinar el time-slice de un proceso. Así, se asegura que por ese período de tiempo, CFS va a ser completamente justo.

#### Min\_granularity

Normalmente se setea en 6ms. El min\_granularity especifica el periodo mínimo, deseado por el programador, en el que se ejecutará una sola tarea. Este parámetro ajustable sólo se usa cuando la carga es alta. A diferencia del sched\_latency\_ns, este parámetro ajustable especifica el periodo de destino asignado para que se ejecute cada tarea, en lugar del tiempo en el que todas las tareas deben ejecutarse una vez.

#### Weighting (Niceness)

CFS tiene control sobre las prioridades de los procesos, de forma tal que los usuarios y administradores puedan asignar más CPU a un determinado proceso. Esto se hace con un mecanismo de cálculo de unix llamado **nivel de proceso nice**. Este valor va de -20 a +19, con un valor por defecto de 0. Con una característica un poco extraña: los valores positivos de nice implican una prioridad más baja, y los valores negativos de nice implican una prioridad más alta.

CFS mapea el nice value con un peso, según la siguiente tabla:

```
static const int prio_to_weight[40] = {  
    /* -20 */ 88761, 71755, 56483, 46273, 36291,  
    /* -15 */ 29154, 23254, 18705, 14949, 11916,  
    /* -10 */ 9548, 7620, 6108, 4984, 3906,  
    /* -5 */ 3121, 2581, 1991, 1586, 1277,  
    /* 0 */ 1024, 820, 655, 526, 423,  
    /* 5 */ 335, 272, 215, 172, 137,  
    /* 10 */ 118, 87, 70, 56, 45,  
    /* 15 */ 36, 29, 23, 18, 15,  
};
```

Y los pesos permiten calcular el **time slice** de cada proceso, pero teniendo en cuenta las distintas prioridades para cada proceso.

$$times\_lice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} * sched\_latency$$

Con esto se calcula el **vruntime**.

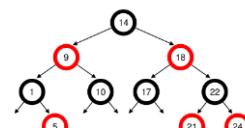
### Árboles Rojo-Negro

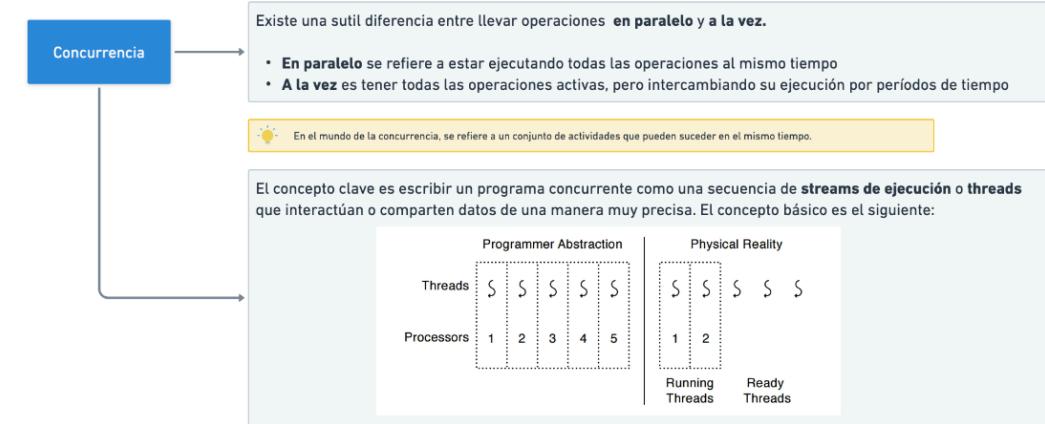
Uno de los focos de eficiencia del CFS está en la implementación de las políticas anteriores. Pero también en una buena selección del tipo de dato cuando el planificador debe encontrar el **próximo job a ser ejecutado**.

- Las listas no escalan bien → O(n)
- Los árboles sí, en este caso los árboles **rojo-negro** → O(log n).

Cuando el scheduler es invocado para correr un nuevo proceso, 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 no), este es reintegrado en el árbol de planificación basado en su nuevo tiempo de ejecución (vruntime).
3. El nuevo nodo que se encuentre más a la izquierda del árbol será ahora el seleccionado, repitiendo la iteración.





## La abstracción

Un thread es una secuencia de ejecución átomica que representa una tarea planificable de ejecución.

### Secuencia de Ejecución Átomica

Cada thread ejecuta una secuencia de instrucciones como lo hace un bloque de código en el modelo de programación secuencial.

### Tarea planificable de ejecución

El sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando él deseé.

### Threads vs Procesos

#### Proceso

Un programa en ejecución con derechos restringidos.

#### Thread

Una secuencia independiente de instrucciones, ejecutándose dentro de un programa.

La abstracción se caracteriza por:

- Thread id
- Un conjunto de los valores de los registros
- Un stack propio
- Una política y prioridad de ejecución
- Un propio error
- Datos específicos del thread

#### One Thread Process

Un proceso con una única secuencia de instrucciones ejecutándose de inicio a fin. Es equivalente a un bloque de instrucciones delimitado por llaves.

#### Many Thread Per Process

Un programa es visto como threads ejecutándose dentro de un proceso con derechos restringidos. En un dato  $t_i$ , algunos threads pueden estar corriendo y otros estar suspendidos. Cuando se detecta, por ejemplo, una operaciones de I/O por alguna interrupción, el kernel desaloja a algunos de los threads, atiende la interrupción y al terminar, vuelve a correr dichos threads.

#### Many Single-Threaded Processes

Limitación de algunos sistemas operativos que permitían varios procesos, pero cada uno con un único thread, lo que implica que puede haber varios threads ejecutándose en kernel mode.

#### Many Kernel Threads

Para aprovechar recursos, también el kernel puede ejecutar varios threads en kernel mode.

### Thread Scheduler

Es necesario un planificador de threads, ya que el SO podría estar trabajando con un único procesador. El cambio entre threads es transparente, es decir, que el programa debe preocuparse de la secuencia de instrucciones y no el cuándo éste debe estar suspendido o no.

Por ende, los threads proveen un modelo de ejecución en el cual **cada thread corre en un procesador virtual dedicado (exclusivo)** con una velocidad variable e impredecible.

Esto quiere decir que, desde el punto de vista del thread, cada instrucción se ejecuta inmediatamente una detrás de otra.

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

#### Multi-threading Cooperativo

No hay interrupción a menos que se solicite.

#### Multi-threading Preemptivo

Es el más usado actualmente. Consiste en que un thread en estado **running** puede ser movido en cualquier momento.

## El API de Threads

Para la programación utilizando threads, se utilizará la biblioteca **pthread**, donde la p es de **POSIX Threads**. El API de **pthread** es muy completa.

### Creación de Un Thread

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

Recibe cuatro argumentos:

1. **thread**: es un puntero a la estructura de tipo **pthread\_t**.
2. **attr**: se utiliza para especificar los ciertos atributos que el thread debería tener.
3. **start\_routine**: es un puntero a una función.
4. **arg**: es un puntero a void que debe apuntar a los argumentos de la función.

### Terminación de un Thread

```
int pthread_join(pthread_t thread, void **value_ptr )
```

1. **thread** es el thread por el que hay que esperar y es de tipo **pthread\_t**.
2. **value\_ptr** es el puntero al valor esperado de retorno.

- Si la función no devuelve nada, NULL.
- Si sólo devuelve un valor, no hay que hacer el empaquetado de los punteros.
- Nunca devolver nada que se encuentre alojado dentro del thread.

## Concurrencia

### Estructura y Ciclo de Vida de un Thread

Recordamos:

#### Thread

Representación de una secuencia de ejecución de un conjunto de instrucciones.

Para que la ilusión sea creíble, el sistema operativo debe guardar y cargar el estado de cada thread. Como cualquier thread, puede correr en el procesador o en el kernel, pero también debe haber estados compartidos, que no deberían cambiar entre los modos.

Para poder entender la abstracción, hay que comprender que existen dos estados:

- El **estado por thread**
- El **estado compartido** entre varios threads

#### Estado Per Thread y TCB: Threads Control Block

##### Threads Control Block

Estructura de un thread que representa su estado. De esta estructura, se crea una entrada por cada thread. La TCB almacena el estado pre-thread de un thread: es decir, el estado de cómputo que debe ser realizado por el thread.

Para poder crear múltiples threads, pararlo y re-arrancarlos, el SO debe poder almacenar en el TCB el estado actual del bloque de ejecución:

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

#### Metadata referente al thread que es utilizada para su administración

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

- ID
- Prioridad de scheduling
- Status



#### Shared State o Estado Compartido

De forma contraria al **pre-thread state**, se debe guardar cierta información que es compartida por varios threads:

- Código
- Variables globales
- Variables del Heap

#### Estados de un Thread

##### Init

Mientras se está inicializando el estado per-thread y se está reservando el espacio de memoria necesario para estas estructuras.

##### Ready

Listo para ser ejecutado, pero no está siendo ejecutado en este instante.

##### Running

Está siendo ejecutado en este mismo instante por el procesador. Los valores de los registros están en el procesador. Se puede volver al estado **ready** de dos formas:

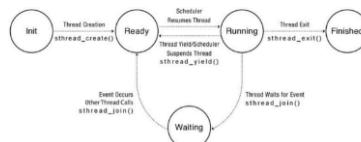
1. El scheduler pasa de running a ready mediante el desalojo.
2. Voluntariamente un thread puede solicitar abandonar la ejecución.

##### Waiting

Esperando que algún determinado evento suceda. Del waiting no se puede pasar al running de forma directa como thread, sino que se tiene que volver a ser elegido por el **scheduler** desde una lista de espera.

##### Finished

Nunca más puede volver a ser ejecutado.



### Threads y Linux

#### Diferencias entre Procesos y Threads

Los threads, por defecto, comparten:

- memoria
- descriptores de archivo (file descriptors)
- contexto del filesystem
- manejo de señales

Los procesos, por defecto, no comparten nada de todo eso.

### Sincronización

La programación multihilo extiende el modelo secuencial de programación de un hilo único de ejecución.

Se pueden encontrar dos escenarios posibles:

1 Un programa está compuesto por un conjunto de threads independientes que operan sobre un conjunto de datos que están completamente separados entre sí y son independientes.

2 Un programa está compuesto por un conjunto de threads que trabajan en forma cooperativa sobre un set de memoria y datos que son compartidos.

Ambos escenarios son distintos y tienen distintas formas de tratamiento.

## Concurrentia

### Sincronización (cont.)

En un programa que utiliza **threads cooperativos**, la forma de pensar secuencial no sirve:  
1. Como el orden de ejecución de los hilos puede no siempre ser la misma, esto influye en los accesos a memoria de recursos compartidos.  
2. La ejecución puede ser no determinística: diferentes corridas producen distintos resultados.  
3. Los compiladores y el procesador físico pueden reordenar las instrucciones.

La solución tiene dos pasos:

1 Estructurar el programa para que resulte fácil el razonamiento concurrente.

2 Utilizar un conjunto de primitivas para sincronizar el acceso a los recursos compartidos.

#### Race Condition

Se da cuando el resultado de un programa depende de cómo se intercalen las operaciones de los threads que se ejecutan en ese proceso. Los threads juegan una carrera entre sus operaciones y el resultado del programa depende de quién gane esa carrera.

#### Operaciones Atómicas

Operaciones que no pueden dividirse en otras y se garantiza la ejecución de las mismas sin tener que intercalar la ejecución.

#### El Problema de la Heladera

Imaginense que dos personas comparten departamento y tienen una única heladera. Como ambos son amigos, los compañeros verifican que nunca falte cerveza en la heladera. La idea es modelizar a cada compañero con un thread y al número de botellas en la heladera como una variable en memoria. Se supone que los **leads** y **stores** son operaciones atómicas.

Dadas operaciones atómicas y con race conditions, se debe garantizar que exista:

- **safety**: que el estado nunca termina en un estado incorrecto.
- **Liveness**: si pasa algo, va a pasar algo bueno y el programa eventualmente siempre está en un estado correcto.

Acá pueden ocurrir varios errores con distintas implementaciones, pero que todos derivan en dos conclusiones:

- O nadie compra cerveza.
- O se compra cerveza dos veces.

La mejor solución para este problema es mediante la utilización de **locks**.

#### Definición de Lock

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

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

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

#### Estados de un Lock

Un lock tiene dos estados:  
• BUSY  
• FREE → Siempre se inicializa en este estado

#### Primitiva obtener()

Usa la primitiva **obtener()** para pedir acceso al lock o a la región de exclusión mutua.

1. Si el estado del lock es FREE, entonces automáticamente pasa a BUSY.
2. Chequear y setear el estado del lock son operaciones atómicas.
3. Si un thread requiere acceso a la región mediante el lock, todos los demás chequean si el lock queda libre y, sino, esperan a que esto suceda.

Un lock utiliza la primitiva **dejar()** la cual pone en estado FREE al lock y, si hubiese otro thread esperando, lo deja entrar.

#### Un Lock debe asegurar:

1. **exclusión mutua**: como mucho, un solo Thread posee el lock a la vez
2. **progress**: si nadie posee el lock y alguien lo quiere, debe poder obtenerlo
3. **bounded waiting**: si un proceso T quiere acceder al lock, pero existen varios threads en la misma situación, los demás tienen una cantidad **finita** de posibles accesos antes de que dicho proceso T lo tome.

 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.

#### Condition Variables

##### Condition Variables

Permiten a un thread esperar a por otro thread para tomar una acción.

Son un objeto sincronizado que permite de forma eficiente esperar por un cambio para compartir algún estado que está protegido por un lock.

Las Condition Variables tienen tres métodos:

##### Wait

- hace unlock
- suspende la ejecución del thread poniéndolo en lista de espera.

##### Signal

- toma un thread de la lista de espera y lo marca como potencialmente seleccionable por el planificador.

##### Broadcast

- Toma todos los threads de la lista y los marca como seleccionables para correr.

### Tipos de Datos Sincronizados

#### Errores comunes de Concurrentia

Se definen dos tipos de errores:

##### Non-deadlock Bugs

Existen dos marcados errores que no están relacionados con deadlock:  
1. **atomicity violation**: "el deseo de serialización entre múltiples accesos a memoria es violado".  
2. **order violation**: "el orden deseado entre accesos a memoria se ha cambiado".

##### Deadlock Bugs

##### Deadlock:

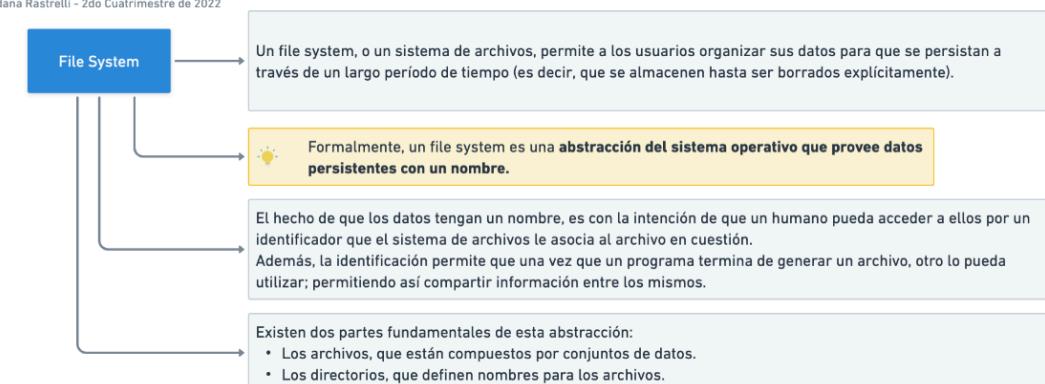
En concurrentia, aparece cuando entre dos o más threads uno obtiene el lock y por algún motivo, nunca libera el mismo haciendo que sus compañeros se bloquen. Según Dahlin, un **deadlock** es un ciclo de espera a través de un conjunto de threads, en el cual cada thread espera que algún otro thread en el ciclo tome alguna acción.

#### Condiciones Necesarias para que se dé un DeadLock

- **Exclusión mutua**: los threads reclaman control exclusivo sobre un recurso compartido que necesitan.
- **Hold-and-wait**: un thread mantiene un recurso reservado para sí mismo mientras espera que se dé una condición.
- **No preemption**: los recursos adquiridos no pueden ser desalojados (preempted) por la fuerza
- **Circular wait**: existe un conjunto de threads que de forma circular cada uno reserva uno o más recursos que son requeridos por el siguiente en la cadena.

#### Cómo prevenir

- **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.
- **Holds-and-wait**: la forma de prevenir el hold and wait es haciendo que los locks se tomen en forma atómica.



### Virtual File System

El Virtual File System (**VFS**) es el subsistema del kernel que implementa las interfaces que tiene que ver con los archivos y el sistema de archivos provistos a los programas corriendo en modo usuario. Todos los sistemas de archivos deben basarse en VFS para coexistir e inter-operar, y así poder utilizar las **system calls de unix** para leer y escribir en diferentes sistemas de archivos y diferentes medios.

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

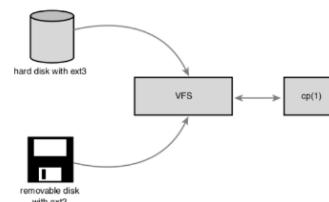
#### File System Abstraction Layer

Es una interface 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 estos difieren en características y comportamiento.

Esto es posible ya que VFS provee un modelo común de archivos que puede 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 de cualquier sistema de archivos soporta, por lo que cada filesystem adapta su forma de trabajo a lo que espera el VFS (es por esto 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 los requerimientos a cumplir, después la implementación puede variar en cada sistema.



#### Estructuras

VFS tiene estructuras que modelan un file system.

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

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

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

#### Operaciones

Operaciones	Definición
<b>super_operations</b>	Son métodos que aplica el kernel sobre un determinado sistema de archivos. Ejemplo: write_inode() o sync_fs().
<b>inode_operations</b>	Son métodos que aplican el kernel sobre un archivo determinado. Ejemplo: create() o link().
<b>dentry_operations</b>	Son métodos que se aplican directamente por el kernel a una determinada directory entry. Ejemplo: d_compare() o d_delete(), comparan y eliminan directorios.
<b>file_operations</b>	Son métodos que aplica el kernel sobre un archivo abierto por un proceso. Ejemplo: read() o write(), las de archivos de siempre.

### Archivos

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

Ejemplo: /home/user/MisDatos.txt.

Cada archivo tiene un nombre único y un significado para referirse a datos. Estos nombres proveen la abstracción de más alto nivel del dispositivo de almacenamiento. Es más fácil manejar el nombre que el bloque en que está guardado en disco. Cada archivo está dividido en 2 partes:

1 La metadata que contiene información acerca del archivo (ejemplo: tamaño, fecha de modificación, propietario y seguridad)

2 Los datos que quieren ser almacenados (desde el OS es un arreglo de bytes sin tipo), guardados en un **data region**.

## File System

### Directarios

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

Definiciones:

Concepto	Definición
<code>path</code>	Es el string que identifica únicamente a un directorio o archivo dentro de un dispositivo.
<code>root_directory</code>	Es el directorio del que cuelgan todos los demás (la raíz).
<code>absolute path</code>	Es la ruta desde el directorio raíz.
<code>relative path</code>	Es el path relativo que se interpreta desde el directorio actual.
<code>current directory</code>	Es el directorio en el cual se ejecuta el proceso.
<code>hard link</code>	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 sería un árbol invertido.
<code>softlinks</code>	Se da cuando un archivo puede ser llamado por distintos nombres (como cuando un archivo apunta a otro archivo).
<code>volumen</code>	Es 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.
<code>mount point</code>	Es un punto en el cual se root de un volumen se engancha dentro de la estructura existente de otro file system.

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

Ejemplo, sería la conexión que ocurre entre el disco de la computadora y el volumen de un USB; el disco tiene directorios preparados para realizar la unión de directorios.

### La API

Las `system calls` de archivos pueden dividirse en dos clases:

1. Las que operan sobre los archivos propiamente dichos.
2. Las que operan sobre los metadatos de los archivos.

#### System Calls sobre los Archivos



#### System Calls sobre los Directarios



#### System Calls sobre los Directarios - incluyendo dirent.h



#### System Calls sobre los Metadatos



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

## File System

### Implementación de un File System

Veremos la implementación de un **vfs** (very simple file system).

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

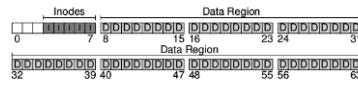
#### Organización General

Se tienen 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 en disco también, en una estructura llamada **Inode Table**, que es un array de inodos almacenados en el disco. Los inodos representan la cantidad máxima de archivos que podrá contener el sistema de archivos. Además, no son estructuras muy grandes, normalmente ocupan entre unos 128 y 256 bytes. Entonces, en un bloque de 4kb se pueden guardar 16 inodos, por lo que tendremos, como máximo en nuestro sistema, 80 inodos.

Los inodos y los bloques o están siendo utilizados, o están libres. La estructura encargada de mantener este registro es la **estructura de aloación**. En este caso, se usará una estructura llamada **bitmap** (una para los datos y otra para los inodos).

Un bitmap es una estructura sencilla en la que se mapea 0 si un objeto está libre y 1 si está ocupado.



Cada bitmap ocupa menos de 4kb, pero se utiliza un bloque por cada uno indefectiblemente.

El único bloque que queda libre en todo el disco se llama **Super Bloque**. El super bloque contiene la información de todo el file system, incluyendo:

- cantidad de inodos
- cantidad de bloques
- dónde comienza la tabla de inodos → bloque 3
- dónde comienzan los bitmaps

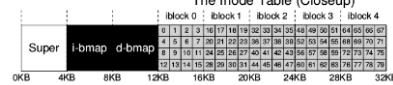
#### Los Inodos

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

Un inodo es referido por un número llamado **inumber** que sería lo que hemos llamado el nombre subyacente en el disco de un archivo.

En este sistema de archivos y en varios otros, dado un **inumber** se puede saber exactamente en qué parte del disco se encuentra el inodo correspondiente.

The Inode Table (Closeup)



Para leer el inodo 32, por ejemplo, el sistema de archivo debe:

1. Calcular el offset en la región de inodos →  $32 * \text{sizeof(inode)} = 8192$
2. Sumarlo a la dirección inicial de la **inode table** en el disco →  $12\text{kb} + 8192 \text{ bytes}$
3. Llegar a la dirección en el disco deseada → que es la **20kb**.

Fin de los temas de la materia

## Más Info

Más apuntes de Sisop [click [aquí](#)]

Más apuntes de Ing. en Informática [click [aquí](#)]

Más material de FIUBA [click [aquí](#)]

Aldana -Aldo- Rastrelli  
2do Cuatrimestre de 2022