

Resumen Sis Op

Introducción a los Sistemas Operativos

¿Qué es un Sistema Operativo?

Software que **gestiona los recursos físicos** de una computadora. Permite **la ejecución concurrente** de programas. Abstacta el hardware para simplificar el desarrollo de aplicaciones. Proporciona **servicios controlados** mediante una interfaz. **Ciclo de ejecución:** Fetch (traer la instrucción) Decode (decodificar) Execute (ejecutar) Comparte hardware entre múltiples programas. Abstacta el hardware → más fácil de usar. Facilita la colaboración entre procesos.

Funciones del SO

- **Árbitro (Referee):** gestiona recursos y aísla procesos.
- **Ilusionista:** crea la ilusión de recursos infinitos.
- **Conector (Glue):** servicios comunes como copiar/pegar, acceso a archivos, I/O.

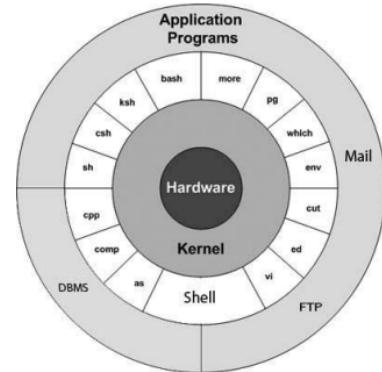
Virtualización: Virtualización es una Abstracción. Transforma recursos físicos en virtuales: CPU, Memoria, Tiempo, Periféricos.

Requisitos Clave de un SO

- **Multiplexación:** varios procesos comparten recursos (CPU, RAM). Compartir el tiempo y recursos de la computadora entre los procesos para asegurar que todos tengan la oportunidad de ejecutarse
- **Aislamiento:** evita que errores de un proceso afecten a otros. Si un proceso falla, no debe impactar a los procesos independientes. Sin embargo, este aislamiento no debe ser absoluto.
- **Interacción:** procesos se comunican (ej: pipes). El sistema operativo permite la comunicación intencionada entre procesos.

El Kernel del Sistema Operativo: núcleo central del SO. Se ejecuta en modo privilegiado (**kernel mode**). Controla directamente el hardware y coordina:

- Memoria
- Procesos
- Dispositivos
- Archivos
- E/S (input/output)



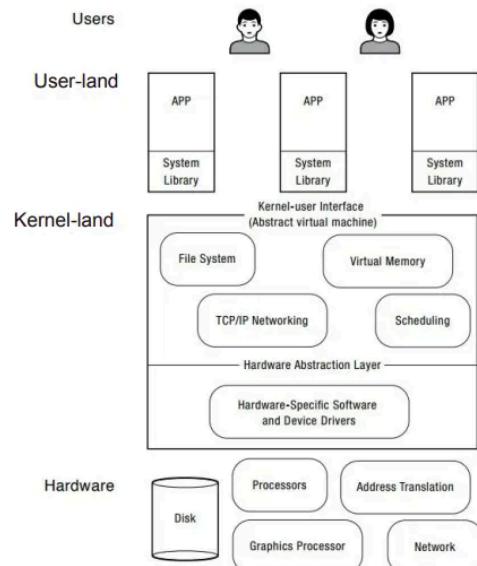
🐧 Kernel en Unix/Linux

Separa **user-land** (apps) y **kernel-land**.

Usa **System Calls** para brindar acceso controlado a los recursos:

- `fork()`, `execve()`, `read()`, `write()`, etc.

Seguridad: Aísla procesos, protege memoria, evita accesos indebidos. Administra recursos compartidos.



Caso de Estudio: UNIX: Cada programa hace **una cosa bien**. Salidas simples, reutilizables como entrada. Uso intensivo de herramientas pequeñas.

File Descriptors: Números enteros que representan recursos abiertos. FDs estándar:

File descriptor	Purpose	POSIX name	stdio stream
0	standard input	STDIN_FILENO	<i>stdin</i>
1	standard output	STDOUT_FILENO	<i>stdout</i>
2	standard error	STDERR_FILENO	<i>stderr</i>

Todo en Unix (archivos, terminales, sockets) puede ser accedido mediante FDs. Los FD abstraen la interfaz de lo que hay detrás. Si tengo un File Descriptor, tengo un “archivo (en el sentido abstracto)”

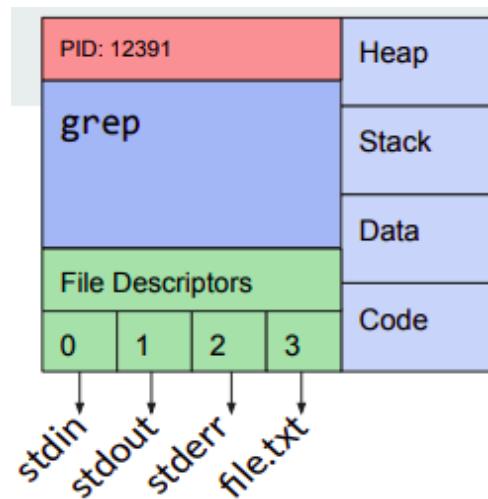
Cosas que puede representar un file descriptor: Archivos normales, Dispositivos de entrada/salida (I/O), Sockets de red, Conexiones TCP y UDP, Pipes y FIFOs, Pseudoterminales (PTY), etc.

Procesos

- Un proceso = programa en ejecución, dinámico, los procesos menos el kernel viven en user-land

Componentes:

- PID
- Memoria (código, datos, stack, heap)
- Tabla de FDs



El kernel los administra en una **Process Table**.

🔧 System Calls clave

Llamada	Función
<code>fork()</code>	Crea un nuevo proceso (copia del actual)
<code>execve()</code>	Reemplaza el proceso actual con otro
<code>wait()</code>	Espera que un proceso hijo finalice
<code>exit()</code>	Termina el proceso actual
<code>getpid()</code>	Devuelve el PID del proceso actual
<code>pipe()</code>	Crea un canal de comunicación entre procesos
<code>dup()</code>	Duplica un file descriptor (útil para redirección de E/S)

El Kernel: Profundizando en su Rol y Funcionamiento

Ejecución Directa vs. Ejecución Directa Limitada

- **Ejecución Directa:** los programas de usuario tienen acceso irrestricto al hardware. En este escenario, el sistema operativo provee bibliotecas, pero carece de mecanismos de aislamiento. La ausencia de aislamiento permite que un programa modifique a otros o al propio sistema operativo, y que acceda directamente al almacenamiento.
- **Ejecución Directa Limitada:** el hardware impone restricciones a ciertas operaciones, y el sistema operativo, actuando como un árbitro, controla las operaciones de riesgo.

La Abstracción User/Kernel

User space	Kernel space
● Una aplicación puede ejecutarse solo en modo usuario.	● Solo el kernel se ejecuta en modo supervisor.
● No puede ejecutar instrucciones privilegiadas.	● Puede ejecutar instrucciones privilegiadas.

User-land: espacio donde viven las aplicaciones de usuario. Cada proceso o programa en ejecución posee memoria con las instrucciones, los datos, el stack y el heap. En el User-Space se ejecutan todos los programas. Se ejecutan en un contexto: Aislado, Protegido, Restringido.

Modos de Operación y Protección del Hardware: User-space vs Kernel-space

Las CPUs proveen **modos duales de operación** que permiten al sistema operativo implementar un aislamiento fuerte entre el código del usuario y el del kernel, protegiendo así al sistema.

- **Modos de ejecución:** modo máquina, modo supervisor y modo usuario. En **x86**, hay cuatro anillos de privilegio, usan 2: **modo usuario** (Ring 3) y **modo supervisor o kernel** (Ring 0).
- **Instrucciones privilegiadas:** Solo pueden ejecutarse en **modo supervisor**. Pueden manipular registros de control, modificar tablas de páginas, habilitar/deshabilitar interrupciones o ejecutar instrucciones. Si un programa en **modo usuario** intenta ejecutar estas instrucciones, el procesador lanza una **excepción**.
- **Transición entre modos:** las aplicaciones deben hacer una syscall, que genera una transición controlada al modo supervisor. En **RISC-V** `ecall`. En

x86, tradicionalmente se usa `int 0x80` (más modernos pueden usar `sysenter` o `syscall`). El **kernel controla el punto de entrada**, lo cual es crítico para la seguridad.

- **Protección de memoria:** Implementada mediante el uso de **tablas de páginas**, las cuales traducen direcciones virtuales a físicas y permiten controlar qué regiones de memoria puede leer, escribir o ejecutar cada proceso.
- **Timer Interrupts:** El hardware puede generar interrupciones periódicas que interrumpen al proceso actual y devuelven el control al kernel, permitiendo implementar **scheduling**.

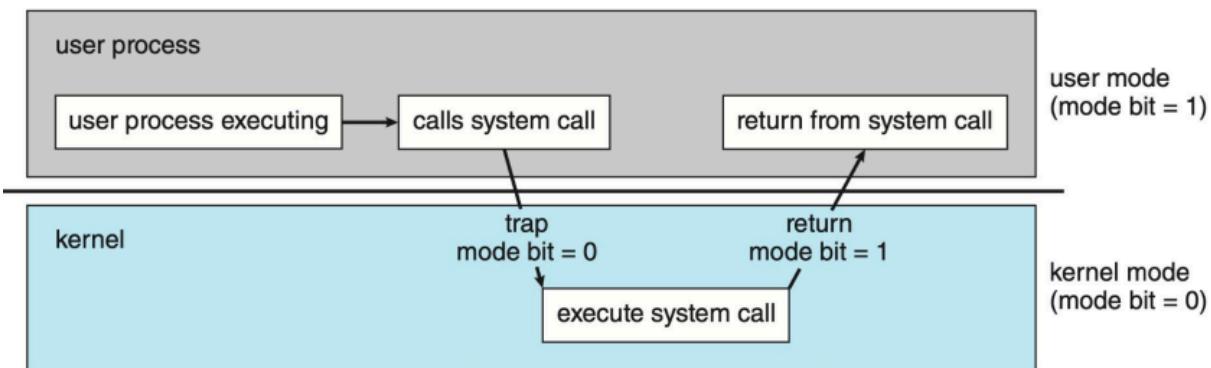


Figure 1.13 Transition from user to kernel mode.

System Calls

- Las "System Calls" son la interfaz controlada que los programas de usuario utilizan para solicitar servicios del kernel. Implican una transición del modo usuario al modo kernel.
- Una system call cambia el modo del procesador de user mode a kernel mode, por ende la CPU podrá acceder al área protegida del kernel. El conjunto de system calls es fijo. Cada system call está identificada por un único número, que por supuesto no es visible al programa, éste sólo conoce su nombre. Cada system call debe tener un conjunto de parámetros.

Mecanismos de protección del hardware

Ejecución Directa

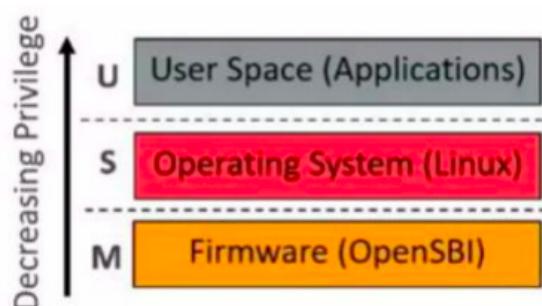
- El sistema operativo principalmente provee rutinas como una biblioteca, pero no ofrece aislamiento entre programas. El problema principal de la ejecución directa es la falta de aislamiento y protección. Esto significa que un programa podría modificar a otro programa o al sistema operativo, o dañar el sistema de archivos.

Modo Dual de Operaciones

- El concepto de "Modo Dual de Operaciones" significa que el procesador puede estar en diferentes estados. Esto es fundamental para la protección del sistema, permitiendo que el kernel opere en un modo privilegiado y las aplicaciones en un modo restringido.

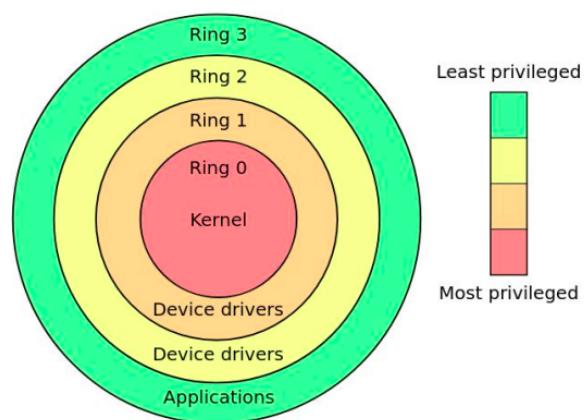
Modos en Risc-V

Risc-V: Cada nivel de privilegio puede ejecutar distintas instrucciones:



Modos en x86

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



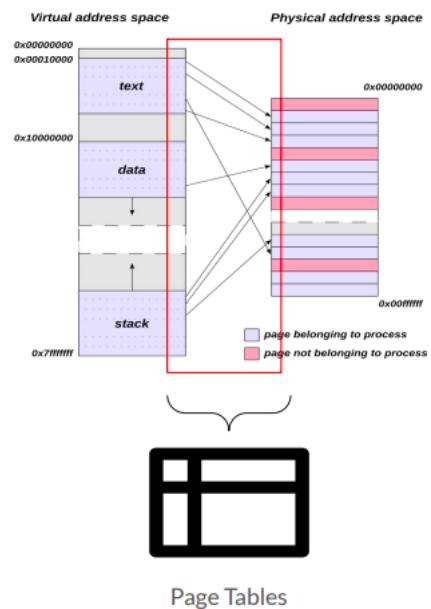
Protección del Kernel

Instrucciones Privilegiadas:

- Son instrucciones que el modo usuario no puede ejecutar. Si un programa en modo usuario intenta ejecutar una instrucción privilegiada, el procesador genera una excepción.

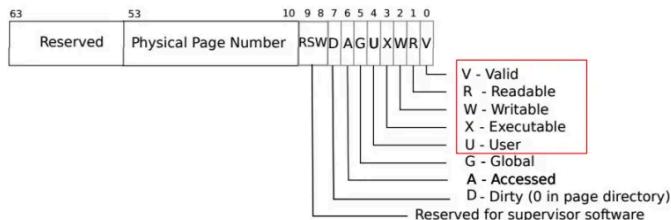
Protección de Memoria:

- La memoria se gestiona mediante "Page Tables," que realizan el mapeo de direcciones virtuales a físicas. Cada entrada tiene direcciones virtuales y físicas, así como flags y conf de cada página. Acciones prohibidas para un usuario: acceder a una página no de usuario, escribir en una página de solo lectura, acceder a una página no mapeada, ejecutar una página no ejecutable. Se generaliza que cualquier acceso de usuario no permitido a la memoria resulta en una excepción.



Page Tables

Un registro de la tabla de páginas (RISC-V)



Timer Interrupts:

- Casi todos los procesadores contienen un dispositivo llamado Hardware Timer, cada timer interrumpe a un determinado procesador mediante una interrupción por hardware. Cuando una interrupción por tiempo se dispara se transfiere desde el proceso de usuario al Kernel

Modos de Transferencia (cambios de contexto)

Transiciones de Modo Usuario a Modo Kernel: inducidas por: Interrupciones, Excepciones del Procesador, System Calls

Interrupciones

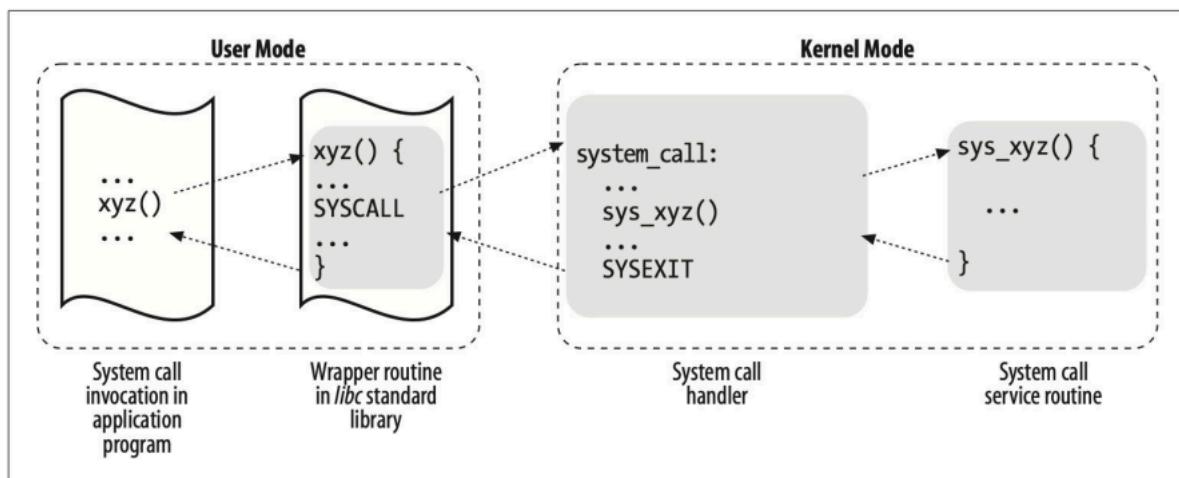
- Una interrupción es una señal asincrónica al procesador, dice que un evento externo requiere atención. El kernel mantiene una tabla de interrupciones y sus correspondientes manejadores.

- Las interrupciones tienen un orden de importancia: Errores de la Máquina—>Timers→Discos→Network devices→Terminales→Interrupciones de Software
- Cada interrupción está asociada a un número. El "Vector de Interrupciones" puede conceptualizarse como un array en la memoria. Cada entrada de este array se asigna a un número de interrupción. Cada entrada contiene la dirección de la función que la CPU ejecutará al recibir esa interrupción, junto con opciones como el nivel de privilegio.

Excepciones del Procesador

- Una excepción es un evento de hardware causado por una aplicación de usuario que provoca la transferencia del control al Kernel.

System Calls: servicio proporcionado por el kernel que puede ser invocado desde el nivel de usuario. Se invocan de manera similar a como un dispositivo de E/S solicita una interrupción. Se utiliza mecanismo de interrupciones excepciones traps.



- System Call se asemeja a la invocación de una función de C. Internamente, el programa realiza una llamada a la System Call mediante la invocación de una función "wrapper" en la biblioteca de C. Esta función wrapper debe proporcionar los argumentos al manejador de la System Call ("trap_handling"). Los argumentos se pasan al wrapper a través del stack, pero el kernel espera en registros específicos, y la función wrapper copia los valores a los registros.

Transiciones de Modo Kernel a Modo Usuario

- Las transiciones de Modo Kernel a Modo Usuario ocurren en los siguientes casos: Al continuar la ejecución después de una interrupción, excepción o System Call. Al realizar un cambio de contexto entre diferentes procesos. Al iniciar un nuevo proceso.

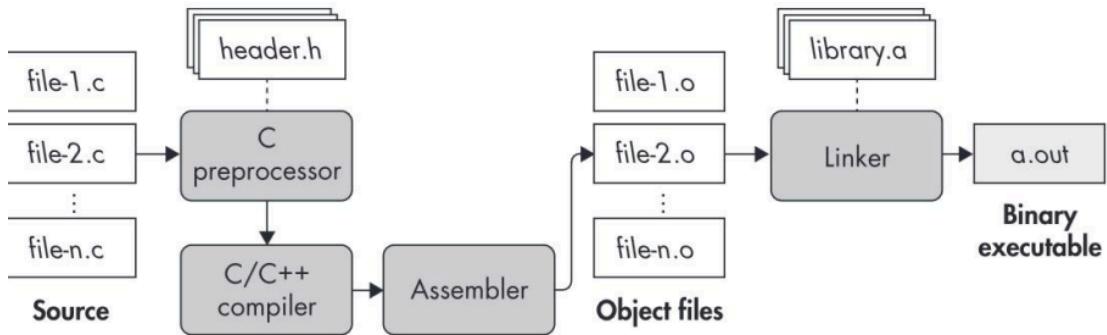
El proceso

El Programa

- El proceso de creación de un programa implica los siguientes pasos: El programador edita el código fuente. El compilador traduce el código fuente a una secuencia de instrucciones de máquina. El compilador guarda esta secuencia, junto con los datos y metadatos del programa, en un archivo ejecutable.
- **Compilación:** El proceso de compilación se describe en varias fases:

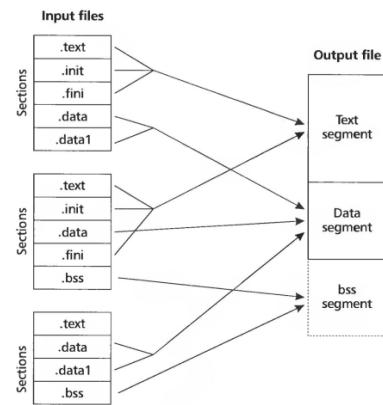


- **Fase de Procesamiento (Pre):** modifica el código fuente original de C según las directivas que comienzan con el carácter #. El resultado es otro programa en C con la extensión .i.
- **Fase de Compilación:** El compilador traduce el programa .i a un archivo de texto .s que contiene el programa en lenguaje assembly. Ejemplo [cat](#).
- **Fase de Ensamblaje:** El ensamblador traduce el archivo .s a instrucciones de lenguaje de máquina, empaquetándolas en un formato llamado programa objeto relocatable. Este archivo tiene la extensión .o y está en formato ELF. Ejemplo: [file ejemplo.o](#). Los archivos objeto, se compilan independientemente, por lo que el ensamblador no conoce las direcciones de memoria de otros archivos objeto. Por eso deben ser reubicables, permite enlazarlos en cualquier orden para formar un ejecutable binario completo.



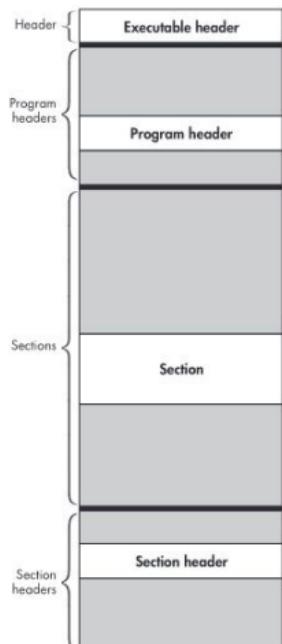
- **Fase de Link Edición:**

- El linkeditor es la última fase de la compilación. Enlaza todos los archivos objeto en un único archivo binario ejecutable. El linkeditor ordena los archivos objeto y enlaza las funciones de las bibliotecas estáticas (.a en C) y deja referencias simbólicas a las bibliotecas compartidas.

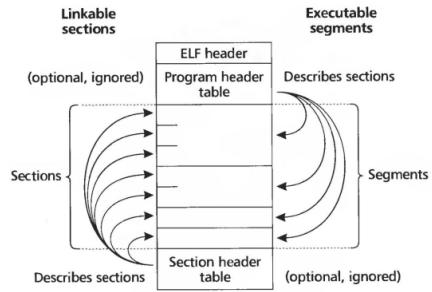


- **Formato Ejecutable:**

- Un programa es un archivo que contiene la información para construir un proceso en memoria.
- Componentes:
 - Instrucciones de Lenguaje de Máquina:
 - Dirección del Punto de Entrada:
 - Datos: valores iniciales de variables, constantes y literales.
 - Símbolos y Tablas de Realocación
 - Bibliotecas Compartidas o Dinámicas
 - Otra información necesaria.

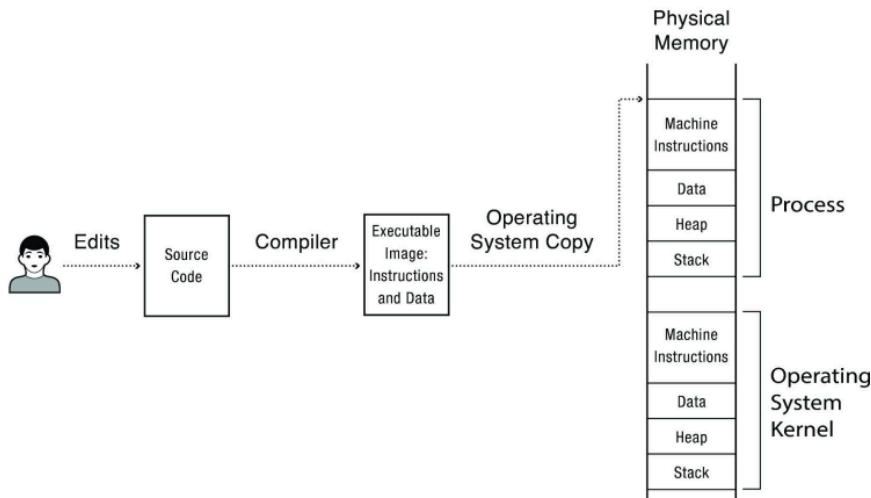


- **Formato ELF:** En Linux, el formato ejecutable es ELF (Executable and Linking Format). El archivo ELF es como un "plano" que el Loader (parte del sistema operativo) usa para "construir" el proceso durante `execve()`.



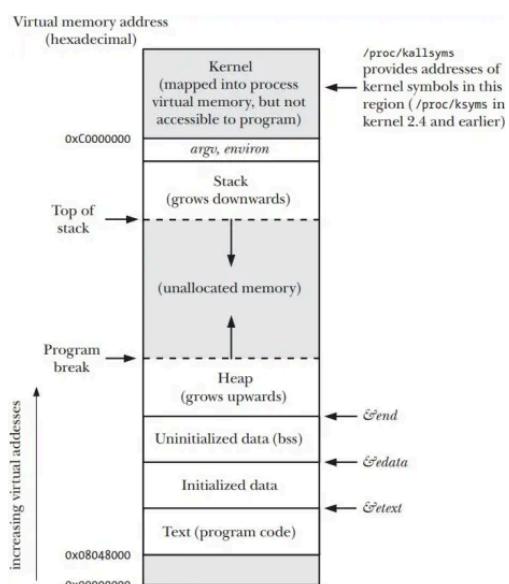
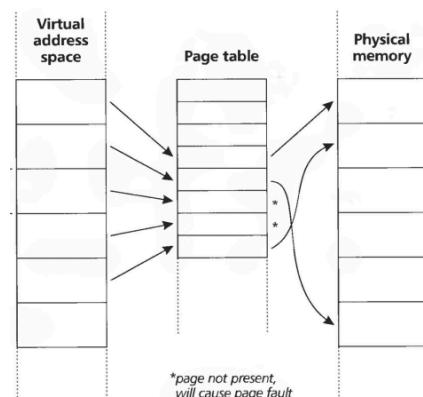
El Proceso

- No es más que solo un programa en ejecución. Incluye: Archivos abiertos, Señales pendientes, Datos internos del kernel, El estado completo del procesador, Un espacio de direcciones de memoria, Uno o más hilos de ejecución (threads), cada uno con: un único contador de programa, un Stack, un Conjunto de Registros, una sección de datos globales.
- **De Programa a Proceso:** El Kernel se encarga de: Cargar instrucciones y Datos de un programa ejecutable en memoria. Crear el Stack y el Heap. Transferir el Control al programa. Proteger al SO y al Programa. El Loader utiliza el archivo ELF para armar espacio de dir.



Virtualización de Memoria:

- La memoria física es compartida por diversos procesos mediante la "Memoria Virtual".
- Las "direcciones virtuales" permiten que la memoria de cada proceso comience en la misma dirección (0), dando la ilusión de que cada proceso tiene toda la memoria.
- El hardware (MMU) traduce las direcciones virtuales a direcciones físicas.



La virtualización de memoria le hace creer al proceso que este tiene toda la memoria disponible para ser reservada y usada como si este estuviera siendo ejecutado sólo en la computadora (ilusión).

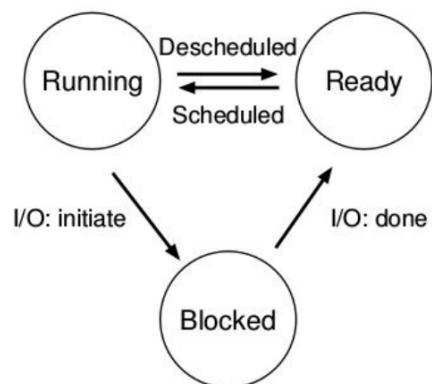
Todos los procesos en Linux, está dividido en 4 segmentos: Text: Instrucciones del Programa. Data: Variables Globales. Heap: Memoria Dinámica. Stack: Variables Locales y trace de llamadas.

Todos los procesos tienen la misma estructura del Address Space.

- System call `brk()` para la asignación de memoria. Es una System call que opera con bloques grandes (páginas). `malloc()` se implementa en User Space y usa `brk()`, manejando estructuras de datos propias para optimizar la memoria.

Estados de un Proceso

- **Corriendo (Running):** El proceso se está ejecutando actualmente en un procesador, instrucción por instrucción.
- **Listo (Ready):** El proceso está preparado para ejecutarse, pero el sistema operativo ha decidido no asignarle un procesador en este momento.
- **Bloqueado (Blocked):** El proceso no puede continuar ejecutándose hasta que ocurra algún evento, como la finalización de una operación de E/S.



Ejemplo Histórico: System V (ver si es necesario)

System V, un sistema Unix, tenía una variedad más extensa de estados de proceso:

- **Corriendo User Mode (Running User Mode):** El proceso se está ejecutando en modo usuario.
- **Corriendo Kernel Mode (Running Kernel Mode):** (El documento no da más información)
- **Listo para Correr en Memoria (Ready to Run on Memory):** El proceso está listo para ejecutarse y reside en la memoria principal.
- **Durmiente en Memoria (Asleep In Memory):** El proceso está bloqueado y reside en la memoria principal.
- **Listo para Correr pero Sustituido (Ready to Run but Swapped):** El proceso está listo para ejecutarse, pero ha sido movido a la memoria secundaria (swap).
- **Durmiente en Memoria Secundaria (Asleep Swapped):** El proceso está bloqueado y reside en la memoria secundaria.
- **Preempt:** Similar al estado "Listo", pero se aplica a un proceso que anteriormente estuvo en modo Kernel.
- **Creado (Created):** El proceso está en el proceso de creación y en un estado de transición.

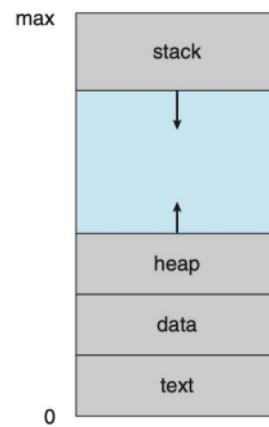
- **Zombie:** El proceso ha terminado su ejecución, específicamente al ejecutar la llamada al sistema `exit()`.

Estructuras del Kernel

- El **contexto de un proceso** es la información necesaria para describir completamente el estado de un proceso. Cada proceso tiene su propio contexto y ve su propio contexto.
- El contexto de un proceso incluye: El contenido del espacio de direcciones (memoria). El contenido de los registros de hardware (CPU). Las estructuras de datos del kernel relacionadas con el proceso. El contexto de un proceso es la unión de: Contexto a nivel de usuario. Contexto de registros. Contexto a nivel de sistema.

Proceso: El Contexto User-level

- Consiste en las secciones que conforman el espacio de direcciones virtual del proceso:
 - Text
 - Data
 - Stack
 - Heap



Contexto de registros: Representa el estado de la CPU, es decir, los registros de la CPU.

Contexto System-level: Process Table Entry: La entrada en la tabla de procesos. Configuración de memoria: Define el mapeo de la memoria virtual a la memoria física. Kernel Stack: Contiene los stack frames de las llamadas a funciones hechas dentro del kernel. La u-area en desuso.

System-level context: U Area

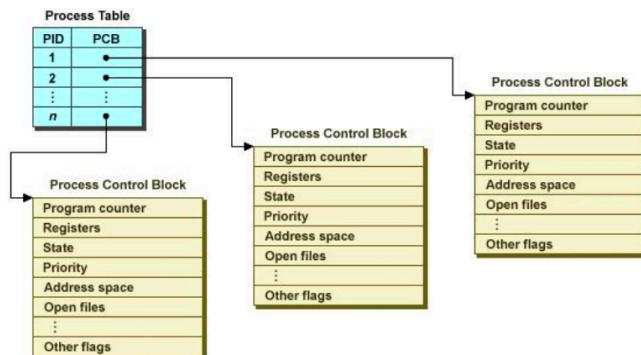
process state
process number
program counter
registers
memory limits
list of open files
...

- La u-area era una estructura por proceso usada en sistemas antiguos (UNIX Version 6) para almacenar información específica de cada proceso (descriptores de archivos abiertos, registros en modo usuario, manejo de señales).

- Los sistemas Unix modernos, como Linux, han reestructurado la gestión de procesos, y la Process Table Entry generalmente contiene todo lo que antes estaba en la u-area.

System-level context: Process Table Entry (o PCB)

- Cada proceso está representado en el sistema operativo por un Process Control Block (PCB).
- El PCB contiene información asociada con un proceso específico.
- Los PCBs se guardan en una tabla, por lo que también se les llama Process Table Entries. Array o lista enlazada.



El Contexto System-level: Process Table:

Un PCB podría contener:
 Identificación: PID y grupo de procesos. Ubicación del mapa de direcciones del Kernel y la u-area del proceso. Estado actual del proceso. Punteros al siguiente y anterior proceso en el planificador. Prioridad. Información para el manejo de señales. Información para la administración de memoria.

Ejemplo PCB: struct proc (xv6): es un array de 64 entradas. Contiene: Process Id Nombre Process State Exit state Parent Process Open files Current directory Page Table Kernel Stack Estructuras auxiliares para context switch y locking.

El context switch: El context switch es el proceso de cambiar de un proceso a otro. Implica cambiar: User-level context (memoria), Register context (CPU). Realizar context switches rápidamente da la ilusión de concurrencia.

Kernel stack: separado del user space, por razones de seguridad y para evitar interferencias con el user space. Se utiliza uno por proceso, cada proceso "dormido" puede estar realizando diferentes operaciones en el kernel. Durante un context-switch se restauran los registros del proceso entrante, se cambia la zona de usuario. No se modifica la memoria del kernel. Se apunta el stack pointer al kernel stack del proceso entrante.

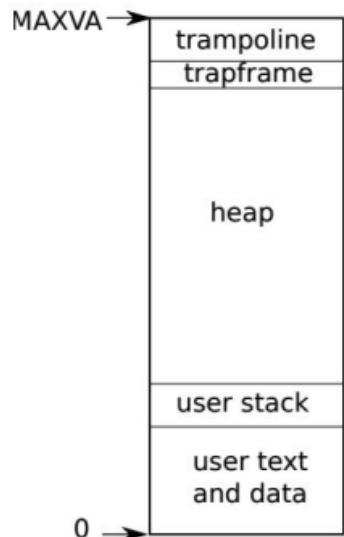
Scheduler I

Cambio de contexto

Un cambio de contexto ocurre cuando el sistema operativo suspende la ejecución de un proceso y restaura la de otro. Implica transiciones entre el modo usuario y el modo kernel.

- **Proceso:**

1. Comienza con una transición a Kernel-Space (el cambio no puede iniciar desde User-Space). Esto puede deberse a interrupciones, excepciones o llamadas al sistema (system calls). No todas estas transiciones resultan en un cambio de contexto.
2. Si es necesario un cambio de contexto, se invoca al scheduler.
3. El scheduler elige el siguiente proceso y llama a `switch`.
4. `Switch` guarda el estado actual del kernel-space y user-space y restaura el del siguiente proceso (que pasa a Kernel-Space).
5. Finalmente, se regresa al User-Space del nuevo proceso.



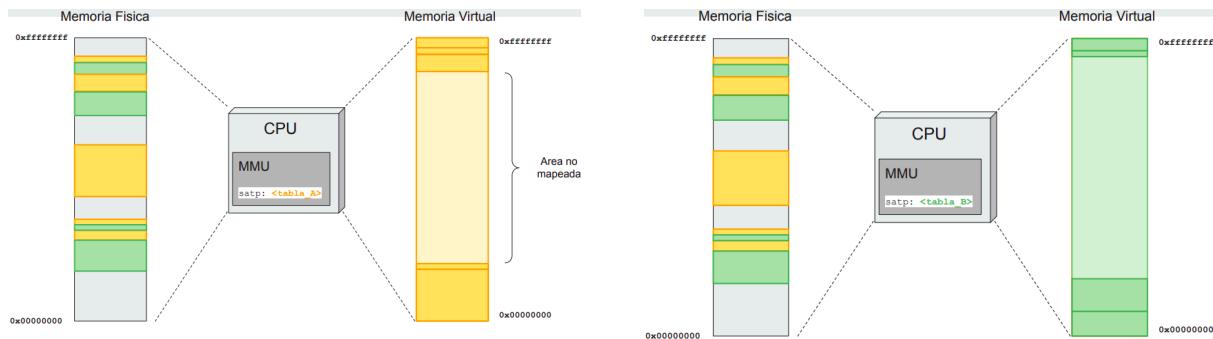
Trampoline

- Es código assembler para la transición de modo usuario a modo kernel.
- Es la única página del kernel cargada continuamente en el user-space del proceso.

Trapframe

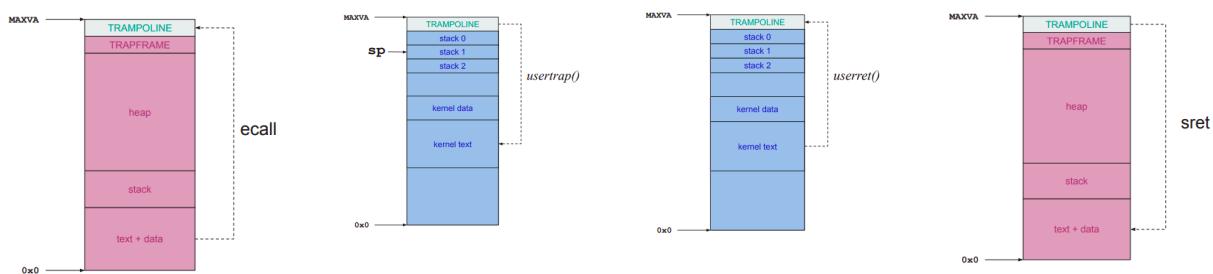
- Es donde el kernel guarda el "estado de los registros" para poder restaurarlos después.
- Es un espacio donde el Kernel guarda el estado de

- Es la única página compartida entre el espacio de direcciones del usuario y el del kernel. El resto del kernel se carga solo cuando es necesario.
 - Los Linux modernos usan Kernel Page-Table Isolation (KPTI), una técnica similar a la de xv6.
- todos los registros del procesador antes de pasar al modo kernel.
- Al restaurar un proceso, el Kernel lee los registros guardados en el Trapframe y los carga nuevamente en el procesador.

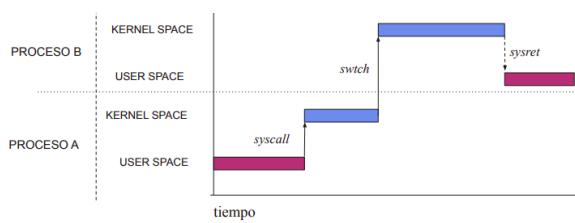


Transiciones User-Kernel en detalle (xv6)

- La transición se inicia con una interrupción de software (`ecall`). El procesador salta al trampolíne y cambia a Kernel Mode.
- El trampolíne: Guarda los registros en el trapframe. Cambia el registro `satp` (para cambiar la tabla de páginas y el espacio de direcciones). Establece el stack pointer (`sp`) al kernel stack del proceso. Invoca la función `usertrap()`.
- Para volver de Kernel a User Mode: Se restauran los registros desde el trapframe. Se cambia el registro `satp` para volver al espacio de direcciones del usuario. Se invoca la función `userret()` (en el trampolíne). Se ejecuta `sret` para volver al punto original en el User Mode.

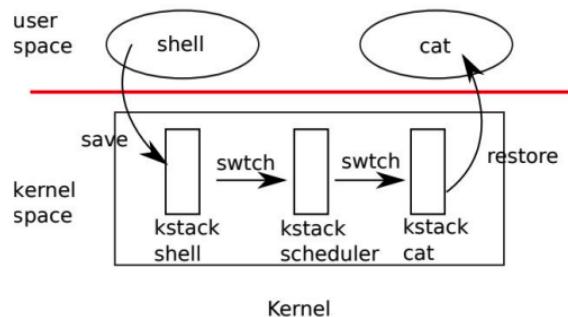


Cambio de contexto en xv6



- En xv6, cada CPU tiene un proceso del kernel llamado el planificador (scheduler), que opera sin espacio de usuario. Un cambio de contexto en xv6 implica dos cambios de contexto: Proceso A al Planificador, Planificador al Proceso B

- **Switch** guarda el estado de la CPU (registros) en `(struct proc) p->context` (dentro del kernel). Carga el estado del planificador. Como hay un planificador por CPU, este contexto se guarda en `(struct cpu) c->context`.
- El estado de la CPU se almacena en: Espacio de Usuario: `(struct proc) p->trapframe` (también mapeado en el espacio de direcciones del usuario, uno por proceso). Kernel en el contexto del proceso: `(struct proc) p->context` (uno por proceso). Kernel en el contexto del planificador: `(struct cpu) c->context` (uno por CPU)



Scheduling

Planificación (Scheduling)

- El planificador (scheduler) es la parte del sistema operativo que decide cuándo se ejecutan los procesos. La planificación determina cuánto tiempo de CPU obtiene cada proceso, a menudo llamado "time slice" o "time quantum". "Workload" se refiere al trabajo que realiza un proceso en ejecución, y cómo se calcula afecta las políticas de planificación.
- Los planificadores se pueden clasificar por tipo de sistema:
 - Los sistemas interactivos priorizan los tiempos de respuesta bajos (Round Robin).

- Los sistemas por lotes (batch) optimizan el tiempo total de ejecución (FIFO, SJF).
- Los planificadores también pueden ser:
 - Preemptivos: el sistema operativo puede interrumpir un proceso en ejecución (RR, STCF).
 - No preemptivos: los procesos se ejecutan hasta su finalización (por ejemplo, FIFO y SJF).

Métricas de planificación

- Primeramente las planificaciones iniciales se basan en suposiciones poco realistas, pero para hacer algún tipo de análisis se debe tener algún tipo de métrica estandarizada . El "tiempo de respuesta" (turnaround time) se utiliza como métrica, calculado como el tiempo de finalización menos el tiempo de llegada.

Bajo las suposiciones simplificadas, el tiempo de llegada es 0

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

Políticas de planificación

- Lotes (Batch): First In, First Out (FIFO). Shortest Job First (SJF). Shortest Time-to-Completion First (STCF)
- Interactiva: Round Robin (RR)

First In, First Out (FIFO)

- FIFO es un algoritmo de planificación simple. Ventajas: simple, fácil de implementar, funciona bien con las suposiciones iniciales. FIFO puede sufrir del "efecto convoy", donde un trabajo largo bloquea a los más cortos.

Shortest Job First (SJF)

- SJF ejecuta el trabajo más corto primero para mejorar el tiempo de respuesta. La ventaja de SJF se reduce si los trabajos llegan en diferentes momentos.

Shortest Time-to-Completion First (STCF)

- STCF aborda el problema del tiempo de llegada adelantando el trabajo actual si llega uno más corto. SJF es una política no preemptiva, mientras

que STCF es preemptiva. (se puede cortar el proceso largo para que ocurran los más pequeño y luego continuar con el más largo)

Tiempo de respuesta (Response Time)

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

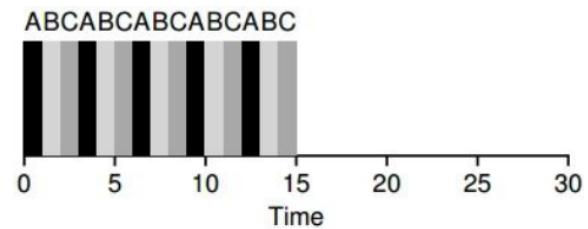
El "tiempo de respuesta" es importante en los sistemas de tiempo compartido, ya que mide el tiempo que tarda un proceso en comenzar a responder a un usuario. Ej: Round Robin.

Round Robin (RR)

- RR asigna a cada proceso un "time slice" y cambia al siguiente proceso después de ese slice. El time slice es un múltiplo del intervalo de interrupción del temporizador.

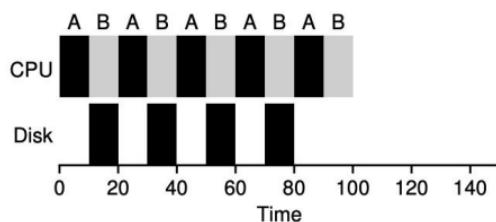
Ejemplo: Si el timer interrupt es 10 ms, el time slice podrá ser 10ms, 20ms y cualquier n*10ms

Elegir un time slice apropiado es crucial para equilibrar la sobrecarga del cambio de contexto y la capacidad de respuesta.



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

Considerando la E/S



Los procesos alternan entre el uso de la CPU y la espera de la E/S. El objetivo es mantener la CPU ocupada superponiendo la E/S y la computación.

Superponer (overlap): mientras se ejecuta I/O, ejecutar otros procesos

Utilización de la CPU

- Los programas pueden ser intensivos en CPU (usando principalmente la CPU) o intensivos en E/S (usando principalmente la E/S).
- Los procesos intensivos en E/S pueden desperdiciar tiempo de CPU si no se gestionan cuidadosamente. Ejecutar múltiples procesos intensivos en E/S puede maximizar la utilización de la CPU.
- Los procesos intensivos en CPU siempre competirán por el tiempo de la CPU. Las GPU se pueden usar para descargar la computación de la CPU, haciendo que las aplicaciones sean más intensivas en E/S.

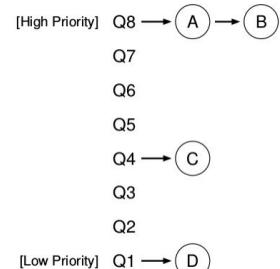
Estrategias para procesos intensivos en E/S

- Con una planificación perfecta, si un programa dedica el 20% de su tiempo a la computación y el 80% a las operaciones de E/S, 5 de estos procesos pueden utilizar completamente la CPU. Las operaciones de E/S son mucho más lentas que las instrucciones de la CPU y normalmente son bloqueantes.
- *Cálculo probabilístico de la utilización de la CPU:* Si un proceso está bloqueado esperando la E/S una fracción p del tiempo, entonces la prob de que esté bloqueado en cualquier momento dado es p . Con n procesos idénticos, la prob de que todos estén bloqueados es p^n . Por lo tanto, la utilización de la CPU es $1 - p^n$.

Multi-Level Feedback Queue

- MLFQ intenta abordar principalmente 2 problemas:
 - Optimizar el "turnaround time", lo que se logra ejecutando primero los trabajos más cortos. El desafío es que el sistema operativo generalmente no sabe la duración de un trabajo, información requerida por algoritmos como Shortest Job First (SJF) o Shortest Time-to-Completion First (STCF).

- Hacer que el sistema sea "responsive" a los usuarios interactivos, minimizando el "response time". Algoritmos como Round Robin reducen el "response time" pero pueden ser malos para el "turnaround time".
- MLFQ utiliza múltiples colas, cada una con un nivel de prioridad asignado.
- Una tarea lista para ejecutarse se encuentra en una única cola. MLFQ usa las prioridades para decidir qué tarea ejecutar en un momento dado: la tarea con la prioridad más alta se ejecuta. Si hay múltiples tareas con la misma prioridad, se utiliza el algoritmo Round Robin para planificarlas.



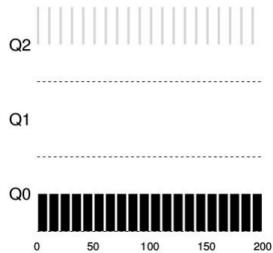
MLFQ: Las reglas básicas

- Las 2 reglas básicas de MLFQ son:
 - **REGLA 1: Si la prioridad (A) es mayor que la prioridad de (B), (A) se ejecuta y (B) no.**
 - **REGLA 2: Si la prioridad de (A) es igual a la prioridad de (B), (A) y (B) se ejecutan en Round-Robin.**
- La clave de la planificación MLFQ es cómo el "scheduler" establece las prioridades, que varían según el comportamiento observado de la tarea. MLFQ "aprende" el comportamiento de los procesos y utiliza esta información para predecir su comportamiento futuro.
- Ejemplos de cómo MLFQ ajusta las prioridades: Una tarea que no usa la CPU mientras espera entrada del teclado mantiene su prioridad alta (comportamiento de proceso interactivo). Una tarea que usa intensivamente la CPU durante largos períodos reduce su prioridad.

Primer intento: ¿Cómo cambiar la prioridad?

- Se debe decidir cómo MLFQ cambiará el nivel de prioridad de una tarea a lo largo de su vida. Esto implica considerar el "workload": una mezcla de tareas interactivas cortas y tareas largas intensivas en CPU.
- Un primer intento de algoritmo de ajuste de prioridades:
 - **REGLA 3: Una tarea que llega al sistema se coloca en la cola de mayor prioridad.**

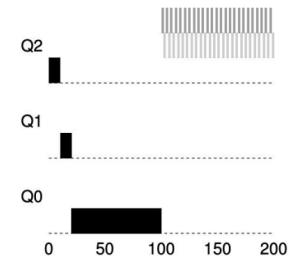
- REGLA 4a: Si una tarea usa su "time slice" completo, su prioridad se reduce en una unidad (pasa a la cola inferior). REGLA 4b: Si una tarea renuncia a la CPU antes de completar un "time slice", permanece en el mismo nivel de prioridad.



- Ejemplo con dos tareas: una de larga ejecución (A) y otra corta e interactiva (B). El algoritmo asume inicialmente que las tareas son cortas y les da alta prioridad. Las tareas cortas terminan rápidamente, y las tareas largas se mueven a colas de menor prioridad (como un proceso BATCH).
- La Regla 4 considera que si una tarea renuncia a la CPU antes de un "time slice" (por ejemplo, por E/S), mantiene su prioridad. Esto beneficia a las tareas interactivas.

PROBLEMA con este Approach de MLFQ

- "Starvation": Las tareas interactivas pueden consumir todo el tiempo de CPU, y las tareas largas nunca se ejecutan.
- Un usuario podría "hackear" el "scheduler" para obtener más tiempo de CPU (ejemplo, realizando operaciones E/S antes del final del "time slice" para quedar en la misma cola de prioridad).

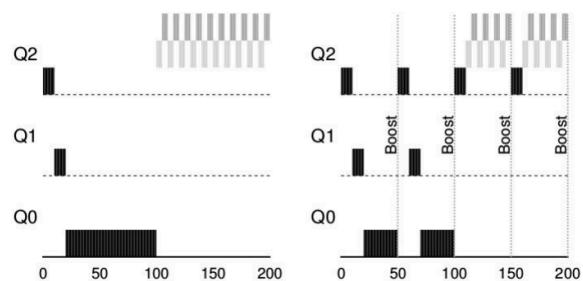


Segundo Approach

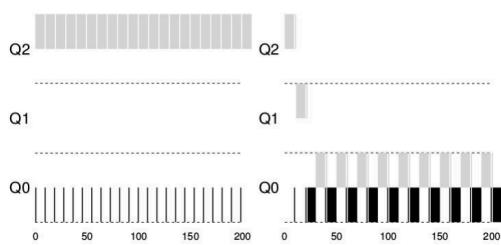
- Para solucionar el problema de "starvation", se mejora la prioridad de todas las tareas en el sistema periódicamente. Se agrega una nueva regla:
 - **Regla 5: Después de un período de tiempo S, las tareas se mueven a la cola de mayor prioridad.**

Segundo Approach: Boost

- El "boost" garantiza que los procesos no sufren "starvation" (se ejecutan mediante Round Robin en la cola de mayor prioridad).



- Si un proceso intensivo en CPU se vuelve interactivo, el "scheduler" lo tratará como tal después del "boost".
- El período de tiempo S es una "VOO-DOO CONSTANT" (difícil de determinar correctamente). Si S es demasiado grande, los procesos largos pueden sufrir "starvation"; si S es demasiado pequeño, las tareas interactivas no comparten adecuadamente la CPU.



Cada proceso tiene un contador que rastrea los "time slices" consumidos

- Para tratar de evitar que los procesos "ventajeen" (game) al "scheduler", se puede llevar un mejor registro del tiempo de uso de la CPU en todos los niveles de MLFQ. El "scheduler" realiza un seguimiento del tiempo que un proceso ha consumido en un nivel, desde que se asigna a una cola hasta que se traslada a otra.

- La Regla 4a y 4b se reescriben en una única regla:
 - **Regla 4: Una vez que una tarea consume su "time-slice" en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU), su prioridad se reduce; baja un nivel en la cola de prioridad.**

Scheduler II

Caso de Estudio: El Planificador de Tareas de Linux

El planificador de tareas es un componente crítico del kernel de Linux, responsable de decidir qué proceso se ejecutará en la CPU en un momento dado. Su evolución a lo largo de las diferentes versiones del kernel ha sido marcada por la búsqueda de eficiencia, equidad y capacidad de respuesta.

Evolución Histórica del Planificador de Linux:

- **Primeras Versiones (hasta Kernel 2.4): Un Enfoque Sencillo:** El planificador inicial de Linux se basaba principalmente en el algoritmo

Round-Robin. Además, se utilizaba un sistema de **Multinivel de Feedback Queue**, donde los procesos se organizaban en diferentes colas según su prioridad y comportamiento (uso intensivo de CPU o cesión frecuente). Sin embargo, este enfoque mostraba limitaciones en sistemas con una alta carga de trabajo.

- **Linux 2.6: El Planificador O(1):** La versión 2.6 introdujo un planificador revolucionario con una complejidad de tiempo constante $O(1)$. Esto implicaba que el tiempo para tomar decisiones de planificación era independiente del número de procesos. La mejora se logró mediante el uso de estructuras de datos más sofisticadas y optimizaciones algorítmicas, teniendo un rendimiento significativamente mejor, especialmente en entornos con alta concurrencia.
- **Linux 2.6.23: El Completely Fair Scheduler (CFS):** En 2007, la versión 2.6.23 marcó un hito con la introducción del **CFS**, se basa en el principio de **planificación proporcional (fair scheduling)**, donde cada proceso recibe una porción del tiempo de CPU que es proporcional a su prioridad.
 - **vruntime (Tiempo Virtual de Ejecución):** CFS introduce el concepto de **vruntime**, un contador virtual asociado a cada proceso. Este contador representa la cantidad de tiempo que el proceso ha estado ejecutándose en la CPU, pero ajustado por su prioridad. Los procesos con mayor prioridad acumulan vruntime más lentamente en relación con el tiempo real.
 - **Equidad (Fairness):** La filosofía central de CFS es asegurar que, en una ventana de tiempo, cada proceso reciba una porción del tiempo de CPU que le corresponda. En teoría, si hay ' n ' procesos con la misma prioridad, cada uno debería ejecutarse durante aproximadamente $1/n$ del tiempo total disponible. En la práctica, CFS busca minimizar la diferencia entre el vruntime de todos los procesos ejecutables.



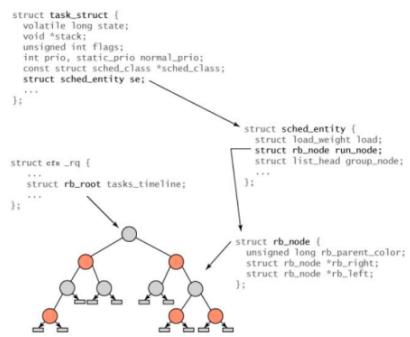
Proceso	Runtime
A	32
B	31
C	26
D	35
E	37

- **Algoritmo de Selección:** En cada decisión de planificación, CFS selecciona el proceso que tiene el valor de vruntime más bajo, es decir, el proceso que ha "quedado más atrás" en términos de tiempo de ejecución proporcional.

Árbol Rojo-Negro (Red-Black Tree) tipo de runqueue:

CFS lo utiliza para gestionar eficientemente los procesos y encontrar rápidamente el que tiene el menor vruntime. Esta estructura de datos auto-balanceada permite:

- Acceso al nodo más a la izquierda (menor vruntime) en tiempo O(1).
- Inserción y eliminación de procesos O(log n).
- Mantiene los procesos ordenados por su vruntime de manera eficiente.



○ Funcionamiento Detallado de CFS:

1. **Selección:** Siempre se elige el proceso con el menor vruntime.
 2. **Árbol Rojo-Negro:** Los procesos se organizan en este árbol por su vruntime.
 3. **Sleeping and Waking Up:** Los procesos bloqueados se eliminan del árbol y se reinsertan al despertar con su vruntime ajustado para no ser penalizados por el tiempo inactivo.
 4. **Incremento de vruntime:** Mientras un proceso se ejecuta, su vruntime aumenta. Los procesos con mayor prioridad ven su vruntime aumentar más lentamente. Si un proceso duerme, su vruntime permanecerá sin cambios.
 5. **Balance de Carga:** CFS también participa en el balanceo de carga entre múltiples CPUs, migrando procesos para optimizar la utilización.
 6. **Soporte para Multi-threading:** Los threads se tratan de manera similar a los procesos, cada uno con su propio vruntime.
- EEVDF en Linux 6.6: Reemplazó a CFS. Basado en:
Tiempo Virtual: Tiempo de ejecución ajustado por prioridad.
Tiempo Elegible: Momento en que un proceso puede ejecutarse.
Solicitudes Virtuales: Demanda de CPU del proceso.
Plazos Virtuales: Momentos esperados para recibir CPU en

tiempo virtual. Asegura que el servicio recibido esté dentro de un límite máximo permitido y asigna recursos de forma proporcional y precisa.

Conceptos Clave en la Planificación de Linux (especialmente relevantes para CFS y EEVDF):

- **nice() y Prioridades:** La llamada al sistema `nice()` permite a los usuarios ajustar la prioridad de un proceso, influyendo en la cantidad de tiempo de CPU que recibirá. Los valores nice varían de -20 (mayor prioridad) a 19 (menor prioridad), con 0 como valor por defecto. Un valor nice más bajo se traduce en un proceso menos "amable" con los demás y con más preferencia del CPU.
- **Pesos de Prioridad (Weight):** Internamente, el kernel de Linux mapea los valores nice a **pesos**. Estos pesos determinan la tasa a la que se acumula el vruntime de un proceso. Un peso mayor (correspondiente a un valor nice más bajo) implica una acumulación de vruntime más lenta.
- **Latencia Objetivo (sched_latency):** Es un parámetro que define el intervalo de tiempo durante el cual el scheduler intenta que todos los procesos ejecutables se ejecuten al menos una vez. Su valor se ajusta dinámicamente en función del número de tareas activas (o prioridad, además de la latencia objetivo) para mantener un buen equilibrio entre equidad y eficiencia. El tiempo durante el cual un proceso se ejecuta antes de que el scheduler considere otro proceso no es fijo en CFS. Un proceso con mayor prioridad recibirá una porción proporcionalmente mayor de la latencia objetivo como su timeslice.
- **Mínimo Tiempo de Ejecución Garantizado (min_granularity):** Este valor establece un límite inferior para el tiempo que un proceso debe ejecutarse antes de ser desalojado, incluso si su timeslice calculado es menor. Esto ayuda a reducir la sobrecarga causada por cambios de contexto frecuentes cuando hay muchos procesos.
- **Normalización de Tiempos:** Para evitar desbordamientos en los contadores de vruntime, el kernel puede normalizar periódicamente los valores, restando un valor base común a todos los vruntime.

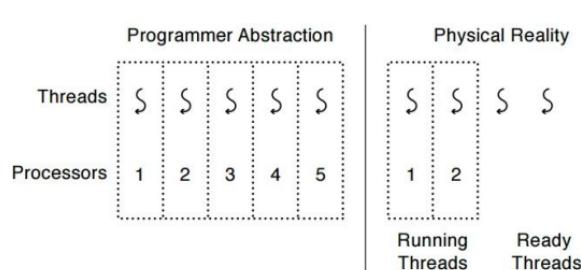
Group Scheduling y Cgroups:

- **Scheduling de Grupos:** Linux permite organizar los procesos en grupos. El scheduler primero asegura una distribución justa del tiempo de CPU entre estos grupos y luego distribuye el tiempo asignado a cada grupo de acuerdo a sus prioridades y pesos.

manera justa entre los procesos que contiene. Esto es útil para implementar políticas de calidad de servicio (QoS) o para asegurar una cierta cantidad de recursos a diferentes usuarios o contenedores.

- **Cgroups (Control Groups):** Los cgroups son una potente funcionalidad del kernel que permite crear jerarquías de colecciones de tareas (procesos y threads) y aplicarles límites y controles sobre el uso de diversos recursos del sistema, incluyendo la CPU. El scheduling de grupos se integra con cgroups, permitiendo una gestión granular de la asignación de CPU a conjuntos de procesos. Un caso de uso importante es la agrupación de los threads de un mismo proceso dentro de un cgroup, lo que permite gestionar el uso de CPU a nivel de proceso.

Threads: Hebras de Ejecución Concurrente



Los threads, o hilos de ejecución, son una abstracción fundamental para lograr la **conurrencia** en la programación. El objetivo principal de utilizar threads es escribir programas que puedan realizar múltiples tareas de manera aparentemente simultánea, mediante la interacción y el compartimiento de datos entre diferentes flujos de ejecución.

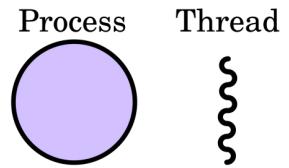
La Abstracción del Thread:

- **Secuencia de ejecución atómica:** Cada thread ejecuta una serie de instrucciones de manera secuencial, similar a un bloque de código en la programación tradicional.
- **Tarea planificable de ejecución:** El sistema operativo tiene la capacidad de controlar la ejecución de un thread en cualquier momento, pudiendo iniciarla, suspenderla y reanudarla según sus políticas de planificación.

Desafíos de la Concurrencia: Inteligibilidad, Predecibilidad, No-Determinismo

Threads vs. Procesos:

- **Proceso:** Se considera un programa en ejecución con derechos y recursos restringidos por el sistema operativo (espacio de memoria propio, descriptores de archivos, etc.).
- **Thread:** Es una secuencia independiente de instrucciones que se ejecuta dentro de un proceso. Múltiples threads pueden coexistir dentro del mismo proceso y compartir su espacio de memoria y otros recursos.



Características de un Thread:

Cada thread se caracteriza por: **Thread ID, Conjunto de valores de registros, Política y prioridad de ejecución, errno propio, Datos específicos del thread.**

Modelos de Ejecución con Threads:

- **One thread per process:** Un modelo donde cada proceso contiene una única secuencia de instrucciones, ejecutándose de principio a fin. Este es el modelo tradicional de programación secuencial.

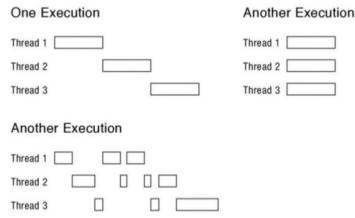
A diagram showing a single purple circle labeled "Process" containing a wavy line labeled "Thread", representing a process with one thread.
- **Many threads per process:** Un modelo donde un programa se estructura como múltiples threads que se ejecutan dentro del contexto de un único proceso con derechos restringidos. En un momento dado, algunos threads pueden estar en ejecución, mientras que otros pueden estar suspendidos. El kernel puede **desalojar (preempt)** threads en ejecución (por E/S o una interrupción), atender eventos y luego permitir que los threads continúen su ejecución.

A diagram showing a large purple circle labeled "Process" containing three smaller purple circles, each with a wavy line labeled "Thread", representing a process with multiple threads.

Thread Scheduler:

El sistema operativo utiliza un **thread scheduler** para crear la ilusión de que muchos threads se ejecutan simultáneamente. El cambio entre threads es transparente para el programador, quien se enfoca en la lógica secuencial de cada thread. Desde la perspectiva de un thread, cada instrucción parece ejecutarse inmediatamente después de la anterior, aunque con una velocidad

de ejecución que puede variar e ser impredecible debido a la planificación del scheduler.



Interleaving (Entrelazamiento):

El scheduler puede entrelazar instrucciones de distintos threads, generando múltiples posibles órdenes de ejecución, lo que complica el razonamiento sobre el comportamiento concurrente.

Modelos de Multi-threading: *Multi-threading cooperativo*: los threads ceden el control voluntariamente; si uno no lo hace, puede bloquear a todos. *Multi-threading preemptivo*: el sistema operativo puede interrumpir threads en cualquier momento para repartir mejor la CPU.

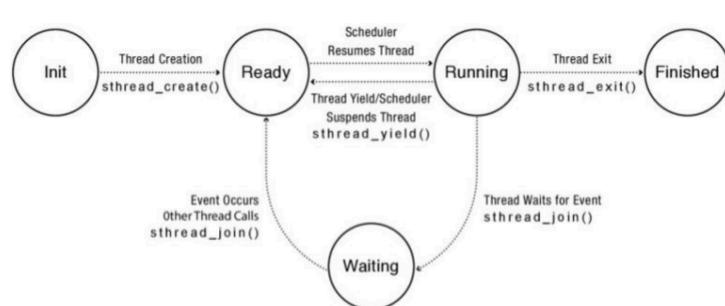
Threads: Implementación en el Sistema Operativo

El sistema operativo debe mantener información tanto a nivel de cada thread como a nivel compartido entre los threads de un mismo proceso.

- **Estado Per-thread y el Thread Control Block (TCB):** Para cada thread, el SO crea una estructura **TCB**. Esta estructura almacena el estado específico de cada thread, permitiendo al SO detener y reanudar su ejecución de forma transparente. La TCB contiene: **Estado del Cómputo, Puntero al stack del thread, Copia de sus registros en el procesador, Metadata del thread**: id, prioridad de scheduling, status.
- **Estado Compartido entre Threads:** Cierta información es compartida por todos los threads dentro de un proceso: **El Código del programa, Variables Globales, Variables del Heap**.

Estados de un Thread:

Un thread puede pasar por diferentes estados durante su ciclo de vida:



- **INIT:** Estado inicial donde se configura el estado per-thread y reservando la memoria necesaria para la TCB y el stack. Una vez completado, pasa al **READY**.

- **READY:** El thread está listo para ser ejecutado y se encuentra en una **ready list**, esperando su turno en el procesador. Los valores de sus registros están guardados en su TCB. El **thread scheduler** transiciona un thread de READY a **RUNNING**.
- **RUNNING:** El thread se está ejecutando actualmente en el procesador. Los valores de sus registros están cargados en la CPU. Un thread en estado RUNNING puede volver a **READY** de dos maneras:
 - **Preemption:** El scheduler decide desalojar el thread en ejecución, guarda su estado en la TCB y selecciona otro thread de la ready list para ejecutar.
 - **Voluntariamente:** Voluntariamente un thread puede solicitar abandonar la ejecución como con `thread_yield`.
- **WAITING:** El thread está esperando que ocurra un evento específico (finalización de uE/S). Los threads en estado WAITING se almacenan en una **waiting list**. Una vez que el evento sucede, el scheduler mueve el TCB del thread de la waiting list a la ready list.
- **FINISHED:** El thread ha completado su ejecución y no se ejecutará más. La TCB del thread terminado se puede encontrar en una **finished list**.

Diferencias entre Procesos y Threads:

Característica	Threads (dentro del mismo proceso)	Procesos (separados)
Memoria	Compartida por defecto	No compartida por defecto
Descriptoros de Archivos	Compartidos por defecto	No compartidos por defecto
Contexto del Filesystem	Compartido por defecto	No compartidos por defecto
Manejo de Señales	Compartido por defecto	No compartidos por defecto

Implementación de Threads en Linux:

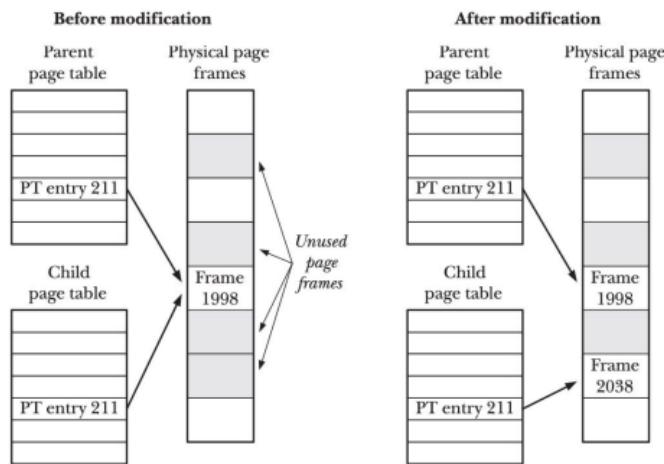
En este modelo, no existe una distinción fundamental entre un thread y un proceso; ambos son tratados como **tareas ejecutables**.

Creación de Procesos y Threads en Linux:

- **`vfork()`**: tipo de `fork()` es más eficiente cuando el hijo realiza una llamada a `exec()` inmediatamente después. **`vfork()` no copia las páginas de memoria del padre; en su lugar, el hijo comparte la memoria del padre hasta que realiza un `exec()` o termina.**

Copy on Write (COW):

Inicialmente, en lugar de realizar una copia física inmediata de las páginas de memoria del padre al hijo, el kernel marca estas páginas como de solo lectura y hace que las entradas de la tabla de páginas de ambos procesos apunten a las mismas páginas físicas.



Si cualquiera de los procesos (padre o hijo) intenta modificar una de estas páginas marcadas como COW, el kernel intercepta este intento (page fault), crea una copia duplicada de la página en memoria física y actualiza la tabla de páginas del proceso que intentó la modificación para que apunte a esta nueva copia. Así ambos procesos tienen su propia copia privada de la página.

Thread Groups (`CLONE_THREAD`): El flag `CLONE_THREAD` es crucial para agrupar los threads que pertenecen al mismo proceso.

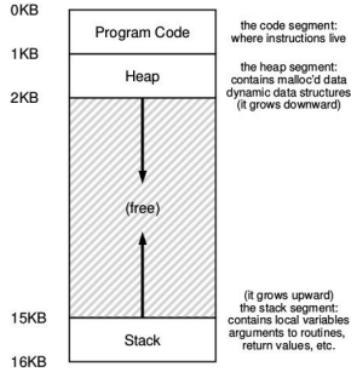
Memoria I

Administración de Memoria: La Abstracción del Espacio de Direcciones

A nivel físico, la memoria puede concebirse como un arreglo lineal de direcciones consecutivas.

Tiempo Compartido (Time Sharing): comparte de forma concurrente los recursos computacionales entre muchos usuarios. Esto se logra mediante la multiprogramación y la introducción de interrupciones de reloj por parte del

sistema operativo, acota el tiempo de respuesta del sistema y limita el uso de la CPU por parte de un proceso individual. IBM 704.



El Concepto de Espacio de Direcciones (Address Space): La Abstracción de la Memoria:

La necesidad de un mecanismo que permitiera utilizar la memoria física de manera fácil y eficiente condujo gradualmente a la concepción del **espacio de direcciones**, una abstracción para la memoria. El **espacio de direcciones (Address Space)** de un proceso contiene todo el estado de la memoria de un programa en ejecución.

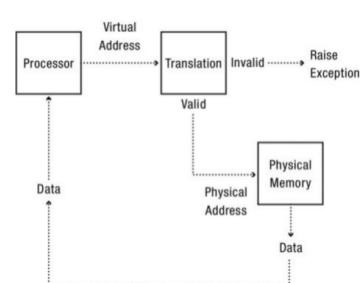
Virtualización de Memoria: Transparencia, Eficiencia y Protección:

El sistema operativo debe **virtualizar la memoria** para lograr varios objetivos clave:

- **Transparencia:** La virtualización debe ser invisible para el programa en ejecución. El programa debe comportarse como si estuviera alojado en su propia área de memoria física privada, sin saber que esta compartiendo la memoria física con otros procesos.
- **Eficiencia:** El sistema operativo debe esforzarse por hacer que la virtualización sea lo más eficiente posible en términos de tiempo (minimizando la ralentización de los programas) y espacio (no utilizar demasiada memoria para las estructuras de soporte de la virtualización).
- **Protección:** El sistema operativo debe garantizar la protección de los procesos entre sí y la protección del propio sistema operativo de los procesos. Un proceso no debe poder acceder o modificar la memoria de otro proceso o del kernel.

Traducción de Direcciones (Address Translation): La Magia del Hardware:

La técnica de **traducción de direcciones** permite que el hardware transforme cada acceso a memoria. La **dirección virtual (virtual address)**, proporcionada desde dentro del espacio de direcciones del proceso, se convierte en una **dirección física (physical address)**, que es la



ubicación real en la memoria física donde se encuentra la información deseada.

En cada referencia a la memoria, el hardware realiza esta traducción de direcciones, redirigiendo las solicitudes de memoria de la aplicación hacia las posiciones reales en la memoria física.

El Rol del Sistema Operativo en la Traducción:

El sisop. debe ayudar al hardware a poder realizar estos procesos, determinando si la traducción se ha realizado correctamente y gestionando uso de la memoria física. Algunas de sus funciones:

- **Configuración del Hardware:** Establecer correctamente el hardware para que la traducción de direcciones se lleve a cabo.
- **Gestión de la Memoria:** Mantener información sobre qué áreas de la memoria física están libres y cuáles están en uso.
- **Control del Uso de la Memoria:** Intervenir de manera criteriosa para mantener el control sobre toda la memoria utilizada.

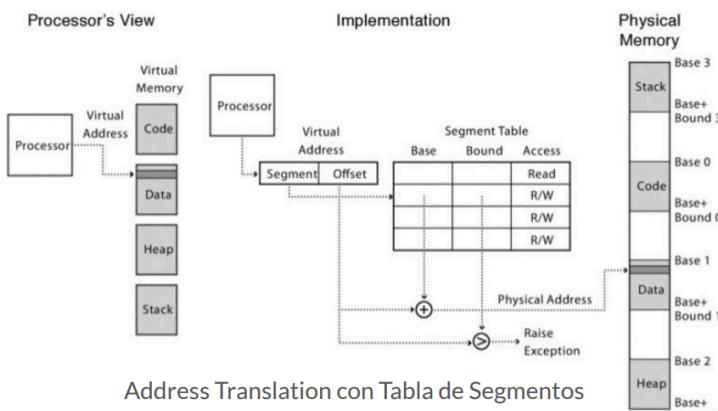
Mapeo Formal de Direcciones: el proceso de traducción de direcciones es un mapeo entre los elementos de un **espacio de direcciones virtuales de N elementos** y un **espacio de direcciones físicas**. Este mapeo puede resultar en una dirección física o en un conjunto vacío (emptyset) si la dirección virtual no es válida.

Implementaciones de memoria virtual

Base and Bound:

Este esquema requiere únicamente dos registros de hardware dentro de cada CPU: **Registro Base, Registro Límite (o Segmento)**.

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

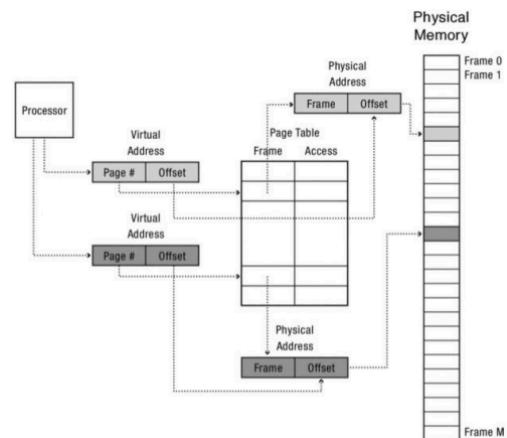


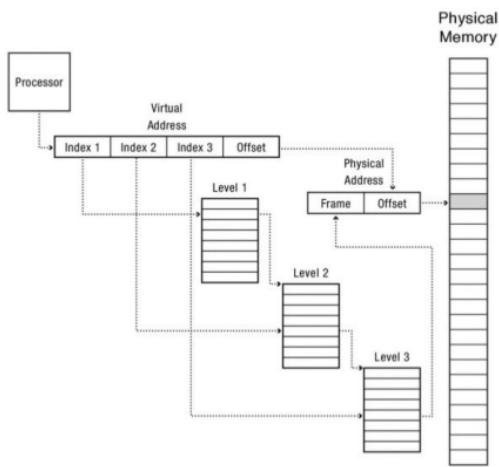
Memoria Segmentada:

Imagina que la memoria de tu computadora es como un armario grande. La memoria segmentada te permite dividir ese armario en secciones de diferentes tamaños (segmentos) para guardar diferentes tipos de cosas de un programa (como las instrucciones, los datos, etc.).

Para encontrar algo, primero buscas la sección correcta (número de segmento) y luego buscas dentro de esa sección (offset). Si intentas buscar fuera de tu sección asignada: Segmentation Fault.

Memoria Paginada: Ahora imagina que el mismo armario está organizado en estantes del mismo tamaño (páginas). La memoria paginada divide tanto la memoria del programa como la memoria física en estos estantes iguales. Para encontrar algo, primero buscas en qué estante virtual está (número de página) y luego el sistema te dice en qué estante físico real se encuentra. Lo bueno es que los estantes virtuales no tienen que estar uno al lado del otro en el armario real.





Memoria Paginada Multinivel: Si el armario anterior es enorme, tener una lista gigante de todos los estantes virtuales y dónde están los reales sería muy pesado. La memoria paginada multinivel organiza esa lista de estantes en varios niveles. Imagina tener un índice general que te dice en qué sección del índice más detallado buscar, y así sucesivamente hasta que encuentras el estante físico. Esto ahorra espacio cuando un programa no está usando mucha de su memoria virtual.

Casos de Estudio: Implementaciones de Memoria Virtual en x86

Base and Bound en el 8086:

- **Registros de Segmento:** CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extra Data Segment).
- **Registros de Punteros e Índices:** SI, DI, BP, SP, IP.

0000 0110 1110 1111 0000	Segment
+ 0001 0010 0011 0100	Offset
<hr/>	
0000 1000 0001 0010 0100	Address

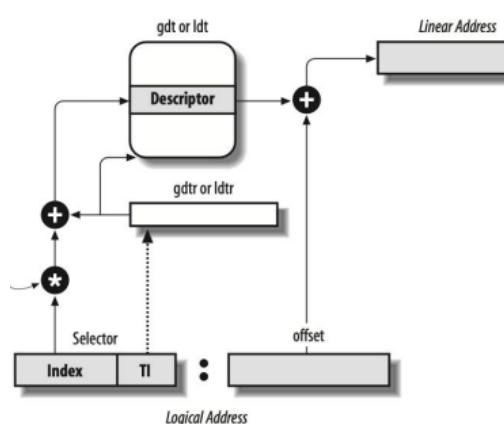
Dado que una dirección de 16 bits solo permitía direccionar 64 KB de memoria, se extendió el direccionamiento utilizando un esquema de segmentación simple. Una **dirección virtual** se calculaba tomando el valor del registro de segmento, desplazándolo 4 bits a la izquierda (multiplicándolo por 16), y luego sumándole un **offset** de 16 bits. En el **modo real** del x86, las direcciones son directamente direcciones físicas, se calculan utilizando el esquema de base + offset. No hay protección de memoria estricta en este modo.

Memoria Segmentada en x86 (Modo Protegido):

El **modo protegido** del x86 proporcionó mecanismos para la protección de memoria y la capacidad de direccionar más allá de 1 MB de memoria física. En este modo, se implementó un esquema de **tabla de segmentos**.

La **Unidad de Gestión de Memoria (MMU)** realiza la traducción de una **dirección lógica** (segmento:offset) a una **dirección lineal** (virtual) mediante la

unidad de segmentación. Posteriormente, una **unidad de paginación** (si está habilitada) transforma la dirección lineal en una **dirección física**.



- **Global Descriptor Table (GDT)**: Una tabla en memoria, apuntada por **GDTR**, que contiene descriptores de segmentos accesibles por todo el sistema (kernel y memoria compartida). Solo hay una GDT.
- **Local Descriptor Table (LDT)**: Normalmente una por tarea, programa o proceso, apuntada por **LDT**. Puede haber varias LDTs en el sistema, pero solo una está activa.
- **Segment Selector**: índice en la GDT que apunta a un Segment Descriptor, cada tabla tiene 8192 entradas.
- **Segment Descriptor**: entrada de 8 bytes en la GDT o LDT describe un segmento de memoria,

Segmentación en Linux:

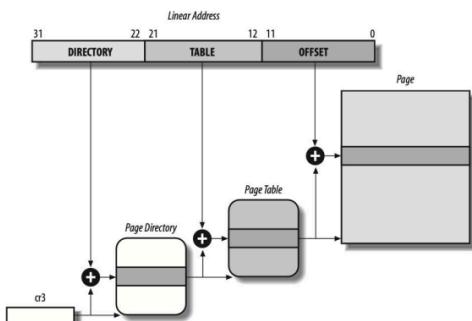
Para simplificar la gestión de la memoria, Linux configura todos los segmentos para que comiencen en la dirección lineal **0x00000000** y abarquen todo el espacio de direcciones lineal. Esto hace que las direcciones lógicas en Linux coincidan con las direcciones lineales; el offset dentro del segmento es la dirección lineal.

Memoria Paginada en x86:

La arquitectura x86 también implementa un esquema de **memoria paginada multinivel**. En el modo protegido con paginación habilitada, la dirección lineal generada por la unidad de segmentación se traduce a una dirección física mediante la unidad de paginación.

En un sistema de paginación, dos niveles, la dirección lineal de 32 bits se divide en tres:

- **Page Directory Index:** Bits de orden superior (10 bits).
- **Page Table Index:** Bits intermedios (10).
- **Page Offset:** Bits de orden inferior (12 bits, ya que el tamaño de página es $2^{12} = 4096$ bytes).



Page Directory Entry

31	...	12	11	...	8	7	6	5	4	3	2	1	0
Bits 31-12 of address		AVL	P S V (0) L	A	P C W D T	P U R S W	/	P					

P: Present	D: Dirty
R/W: Read/Write	PS: Page Size
U/S: User/Supervisor	G: Global
PWT: Write-Through	AVL: Available
PCD: Cache Disable	PAT: Page Attribute
A: Accessed	Table

Page Table Entry

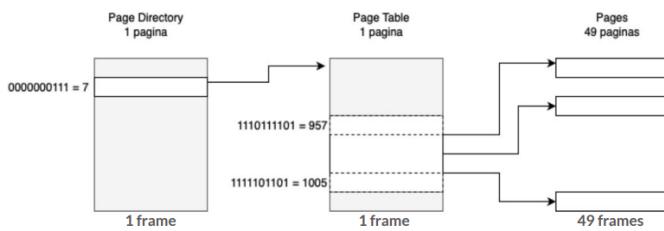
31	...	12	11	...	9	8	7	6	5	4	3	2	1	0
Bits 31-12 of address		AVL	P G A T	D	A	P C W D T	P U R S W	/	P					

P: Present	D: Dirty
R/W: Read/Write	G: Global
U/S: User/Supervisor	AVL: Available
PWT: Write-Through	PAT: Page Attribute
PCD: Cache Disable	Table
A: Accessed	

La traducción se realiza de la siguiente manera:

- El **registro CR3** contiene la dirección física base del **Page Directory**. El índice del page directory se utiliza como índice para encontrar una **Page Directory Entry**. La PDE contiene la dirección física base de una **Page Table**. El índice de la page table se utiliza como índice en la para encontrar una **Page Table Entry**. La PTE contiene la dirección física base de un **Page Frame**. El Offset de Página se añade a la dirección base del frame de página para obtener la **dirección física final**.

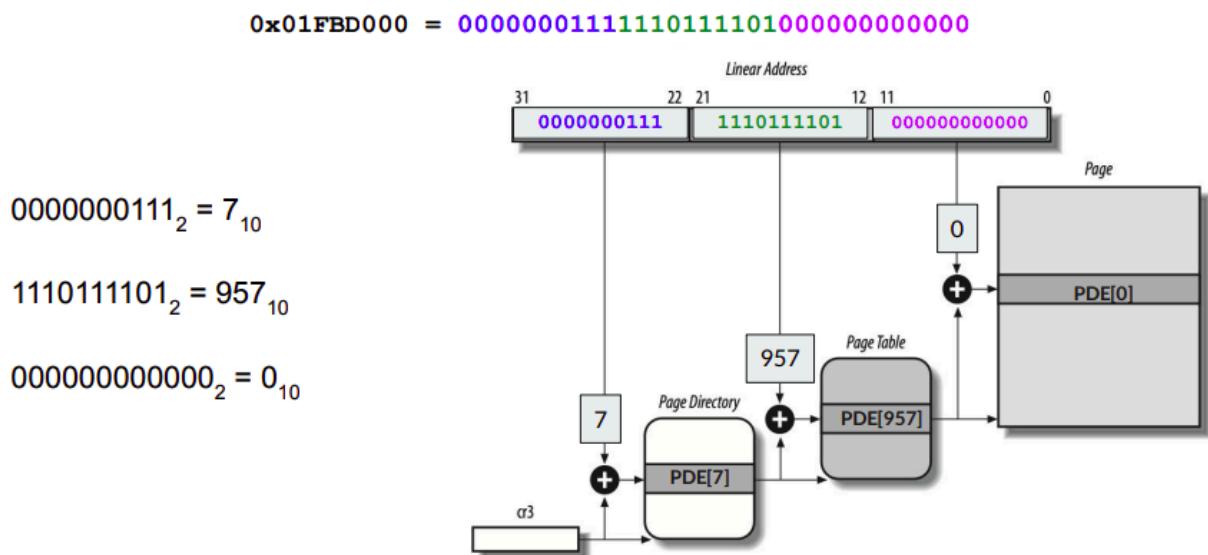
Entradas del Directorio de Páginas (PDE) y Entradas de la Tabla de Páginas (PTE):



Tanto las PDEs como las PTEs son entradas de 4 bytes que contienen información sobre la presencia de la página, permisos de acceso, políticas de caché, bits de uso, y la dirección física del siguiente nivel.

Registros Especiales para la Página:

- CRO (Registro de Control 0):** Contiene varios flags de control del procesador: **PE (Bit 0):** Habilita el modo protegido. **PG (Bit 31):** Habilita la paginación. Si PG es 1, la dirección lineal se traduce a física mediante la paginación. Si PG es 0, la dirección lineal es la dirección física.
- CR3 (Registro de Control 3):** Contiene la dirección física base del Directorio de Páginas. Al cambiar el valor de CR3, se cambia el espacio de direcciones virtuales activo. La Traducción de la Tabla de Búsqueda (TLB) se invalida al escribir en CR3.

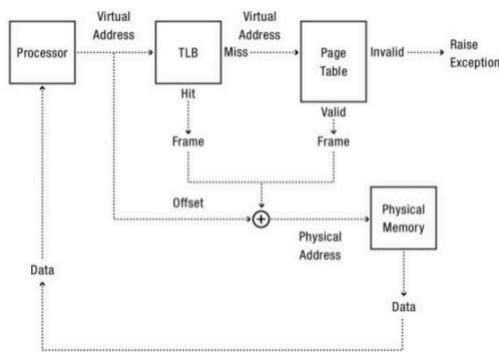


Translation Lookaside Buffer (TLB): Caché para la Traducción de Direcciones

El Translation Lookaside Buffer (TLB): Caché de Traducciones de direcciones:

TLB es un caché de hardware especializado que forma parte de la MMU. Su función principal es almacenar las traducciones recientes entre direcciones virtuales y direcciones físicas.

Por cada acceso a una dirección virtual, el hardware primero consulta la TLB para verificar si la traducción ya está almacenada allí. Si la traducción se encuentra en la TLB (**TLB hit**), la dirección física se obtiene rápido. Si la traducción no está en la TLB (**TLB miss**), el hardware debe realizar la traducción completa, y la traducción resultante se almacena en la TLB para futuras referencias.



Las entradas de la TLB se implementan en memoria estática on-chip de alta velocidad, ubicada cerca del procesador. Muchos sistemas modernos incluyen múltiples niveles de TLB (L1 TLB, L2 TLB) para mejorar aún más el rendimiento. Los niveles más pequeños suelen ser más rápidos.

Consistencia de la TLB:

Al igual que con cualquier caché, es crucial mantener la **consistencia** de la TLB con los datos originales (en este caso, las tablas de páginas).

- **Context Switch:** Las direcciones virtuales del viejo proceso ya no son más válidas, y no deben ser válidas, para el nuevo proceso. De otra forma, el nuevo proceso sería capaz de leer las direcciones del viejo proceso. Frente a un context switch, se necesita descartar el contenido de TLB en cada context switch. Este approach se denomina flush de TLB.
- **TLB Shutdown:** En un sistema con múltiples procesadores, cada uno puede tener en su TLB una copia de una traducción. Si una entrada en una tabla de páginas es modificada, la entrada correspondiente en todas las TLB de todos los procesadores debe ser invalidada para garantizar la coherencia. Esto a menudo requiere que el sistema operativo envíe una interrupción a cada procesador para solicitar la invalidación de la entrada de la TLB, una operación costosa conocida como **TLB shutdown**.

Memoria II

Administración de Memoria en el Kernel

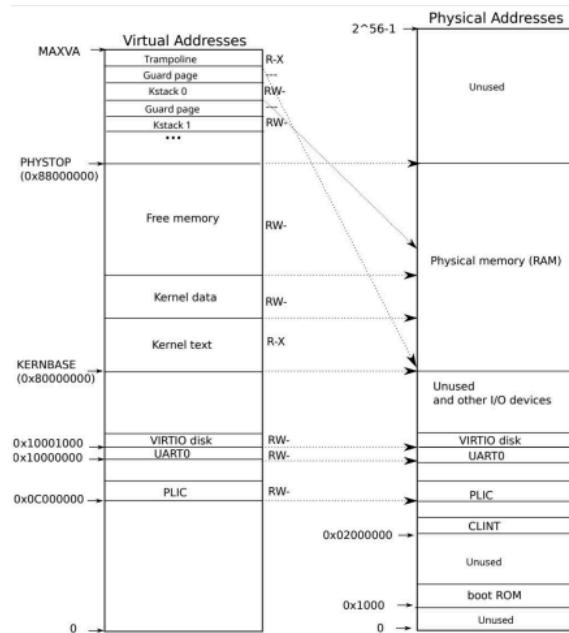
Mapeo Directo: En xv6, el kernel utiliza extensamente el **mapeo directo**. Esto significa que la memoria RAM y los registros de los dispositivos mapeados en memoria son accesibles a través de direcciones virtuales que son idénticas a sus direcciones físicas.

El mapeo directo simplifica el código del kernel. Cuando el kernel asigna memoria física, la dirección física devuelta por el asignador puede utilizarse directamente como una dirección virtual para acceder a esa memoria.

Excepciones al Mapeo Directo en xv6:

Existen algunas áreas de las direcciones virtuales del kernel que no están mapeadas directamente:

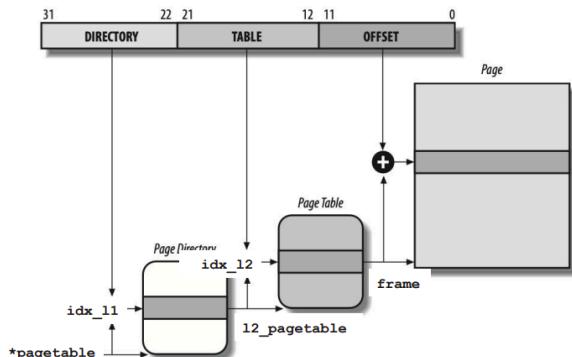
- **La página trampolín:** Está mapeada en la parte superior del espacio de direcciones virtuales tanto para el kernel como para el espacio de usuario.
- **Las páginas de pila del kernel:** Cada proceso tiene su propia pila del kernel, que se mapea en una dirección alta, permitiendo que xv6 coloque una página de protección no mapeada justo debajo.



kalloc: El Asignador de Memoria Física del Kernel:

kalloc es el asignador de memoria física del kernel en xv6. Su estructura de datos principal es una **lista libre** de páginas de memoria física que están disponibles para ser asignadas. Cada elemento de la lista de páginas libres es un struct run. El asignador obtiene memoria para almacenar estas estructuras **run** directamente de las propias páginas libres, ya que inicialmente no contienen otra información.

Es importante notar que `kalloc` y `kfree` operan directamente con **direcciones físicas**, no virtuales. Otra parte del kernel se encarga de mapear estas direcciones físicas a direcciones virtuales en las tablas de páginas. Además, la lista libre funciona como una pila (LIFO).



Mapeo de Páginas:

El objetivo del mapeo de páginas es, dada una dirección virtual, asignar una página física y establecer las entradas en las tablas de páginas para que esa dirección virtual apunte a la página física asignada. Esto es lo que hace el kernel cuando un proceso solicita más memoria.

La función `int mappage(pte_t *pagetable, uint32 va)` :

1. Calcula los índices de las tablas de páginas de primer (`idx_I1`) y segundo nivel (`idx_I2`) a partir de la dirección virtual `va`.
2. Obtiene la entrada de la tabla de primer nivel correspondiente.
3. Si la tabla de segundo nivel no existe para esa parte del espacio virtual, asigna una nueva página física usando `kalloc()` para la tabla de segundo nivel y actualiza la entrada de la tabla de primer nivel para que apunte a esta nueva tabla.
4. Finalmente, asigna una nueva página física para la propia página de datos usando `kalloc()` y actualiza la entrada de la tabla de segundo nivel para que apunte a esta página física.

Páginas en xv6 RISC-V:

En xv6 para la arquitectura RISC-V de 64 bits, el esquema de paginación es de **tres niveles** en lugar de dos. Los 25 bits superiores de la dirección virtual se ignoran. Cada entrada de la tabla de páginas incluye un bit **V (Valid)** que indica si la entrada apunta a una página válida o si no está mapeada. Recorre los niveles de la tabla de páginas. Si una entrada no es válida asigna una nueva página física para la siguiente tabla de nivel y actualiza la actual para que apunte a ella.

Administración de Memoria en el Espacio de Usuario

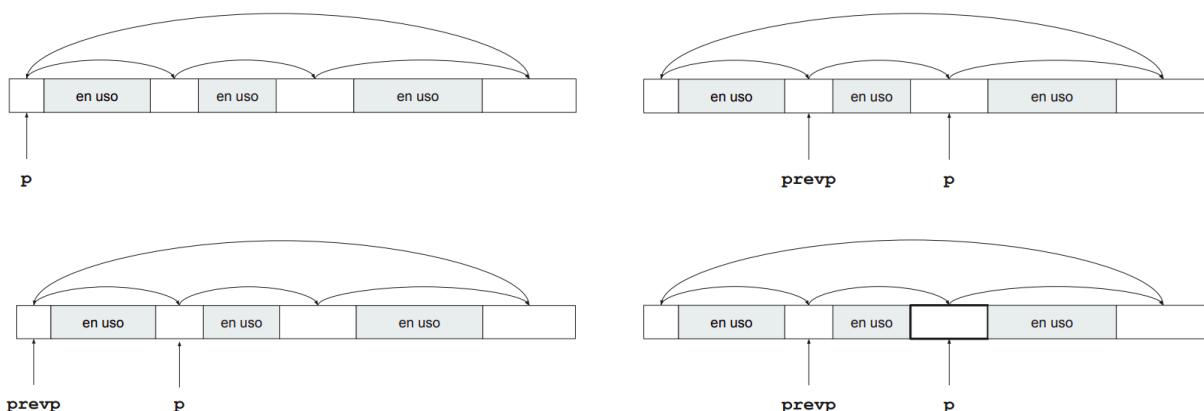
`malloc()` :

- Se utiliza para asignar un bloque de memoria, la memoria devuelta está típicamente alineada a 8 bytes. **No inicializa** el contenido de la memoria que devuelve. Suele utilizar las llamadas al sistema `sbrk()` (para expandir o contraer el segmento de datos del proceso) o `mmap()` (para mapear nuevas regiones de memoria) para obtener memoria del sistema operativo.

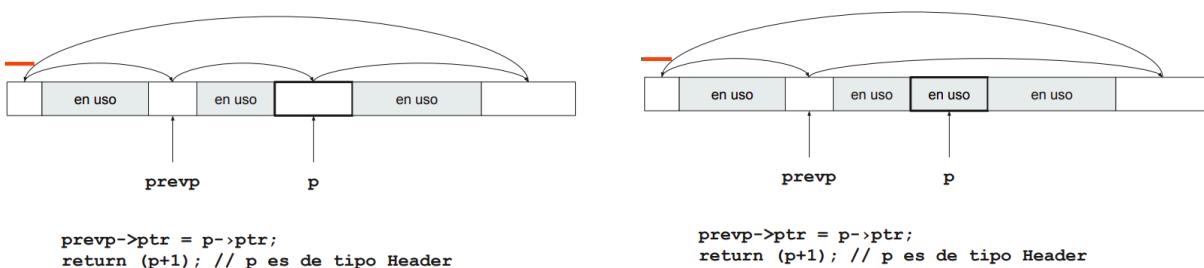
`free()` :

- Se utiliza para liberar un bloque de memoria que fue previamente reservado en el heap. Pasar un puntero que no fue obtenido de estas funciones conduce a un comportamiento **indefinido**, lo que a menudo resulta en errores o corrupción de la memoria.

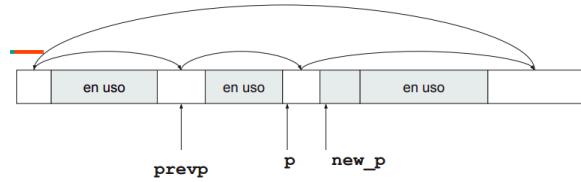
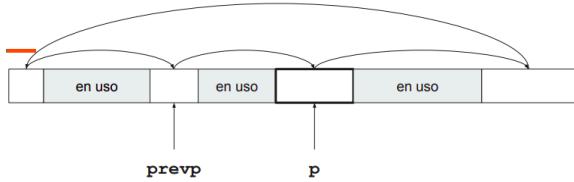
Buscar un bloque libre



Reservar el bloque completo



Split, reservar parte del bloque



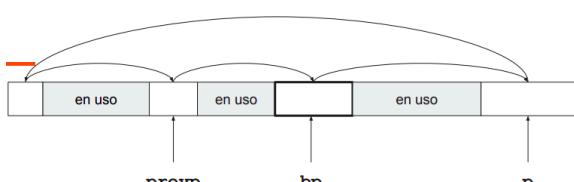
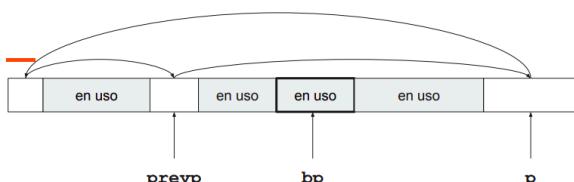
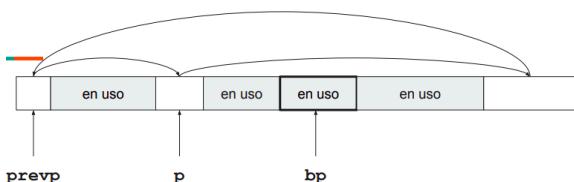
```

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

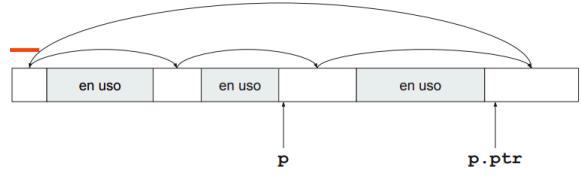
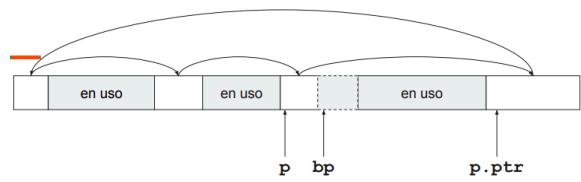
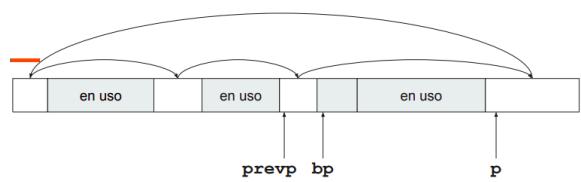
return (new_p + 1); // new_p es de tipo Header

```

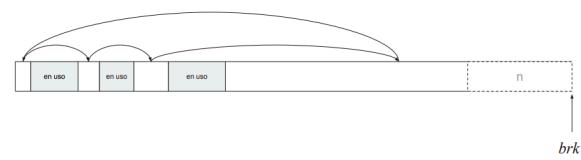
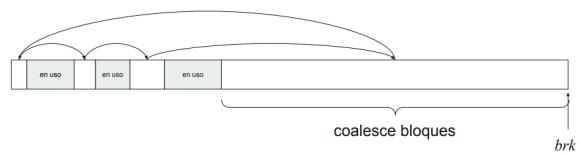
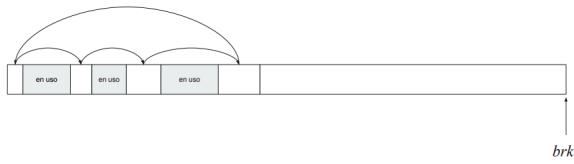
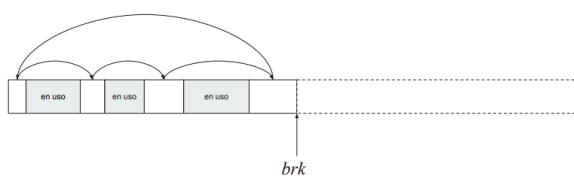
Liberar el bloque

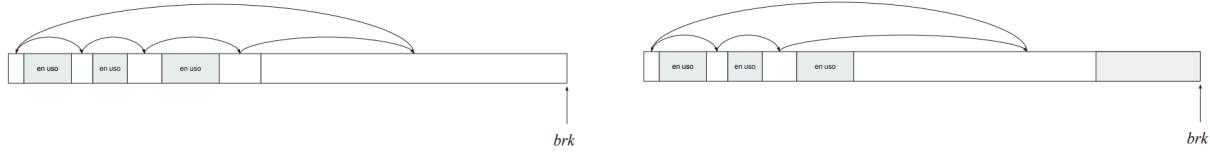


Liberar el bloque (coalesce)



Reservar memoria pidiendo mas al OS





mmap(): La llamada proporciona una forma de mapear archivos o regiones de memoria en el espacio de direcciones de un proceso, permite compartir memoria en varios procesos.

Sincronización

1. **Threads Independientes:** Un programa compuesto por múltiples threads que operan sobre conjuntos de datos completamente separados e independientes entre sí.
2. **Threads Cooperativos:** Un programa compuesto por múltiples threads que trabajan de forma conjunta sobre un conjunto compartido de memoria y datos.

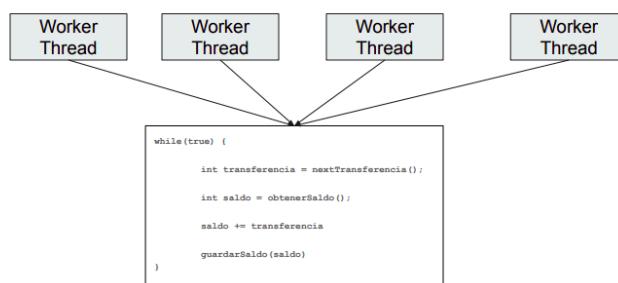
Desafíos de la Programación Concurrente Cooperativa:

La ejecución del programa depende del **entrelazamiento** de los threads, Diferentes ejecuciones del mismo programa pueden producir resultados distintos debido al scheduler. Los compiladores y el procesador físico pueden **reordenar las instrucciones** para mejorar el rendimiento.

El enfoque para manejar estos problemas : Estructurar el programa de manera que facilite el razonamiento sobre la concurrencia y utilizar un conjunto de primitivas estándar.

Race Conditions: ocurre cuando el resultado de un programa depende del orden en que se intercalan las operaciones de los threads que se ejecutan dentro de ese proceso. Los threads compiten entre sus operaciones, y el resultado final depende de quién "gana" esa carrera.

Un típico proceso para procesar transacciones



Sección Crítica:

Es una abstracción, mediante la cual se define que en una sección de código solo puede haber un thread en ejecución a la vez. En una sección crítica se define una sección de código que se ejecuta bajo "exclusión mutua"

Locks

Un **lock** es una variable que permite la sincronización mediante la **exclusión mutua**. Cuando un thread "tiene" el lock, ningún otro thread puede adquirir el mismo lock. La idea principal es asociar un lock con ciertos estados o secciones de código que acceden a recursos compartidos. Un thread debe poseer el lock para entrar. Esto garantiza que solo un thread acceda al recurso compartido a la vez, logrando la atomicidad de las operaciones dentro de la región protegida por el lock.

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

Propiedades Formales de los Locks:

- **Exclusión Mutua:** A lo sumo, solo un thread posee el lock en un momento dado.
- **Progreso:** si ninguno posee el lock y otro quiere adquirirlo, alguno de ellos podrá hacerlo.
- **Bounded Waiting:** Si un thread quiere adquirir el lock y hay otros threads en la misma situación, existe un límite en la cantidad de veces que los otros threads pueden adquirir el lock antes de que el thread original lo haga.

Spinlock: es un tipo de lock, utiliza una técnica de "**busy wait**". En lugar de bloquearse y ceder el procesador, un thread que intenta adquirir un spinlock repetidamente verifica una variable hasta que el lock está disponible. Se utiliza cuando: el tiempo que se mantiene el lock es muy corto, hay poca contención por el lock, en sistemas con paralelismo real donde otro thread podría estar ejecutándose simultáneamente y liberar el lock. Spinlocks en xv6 utilizan **operaciones atómicas**.

Operaciones Atómicas: es una instrucción del procesador que se garantiza que se ejecuta completamente ("todo o nada") sin interrupciones de otros threads o procesadores.

Sleeplock: tipo de lock que hace que el thread que no puede adquirir el lock se **duerma** (cede el procesador). Cuando el lock es liberado, los threads que estaban esperando son despertados. Las alternativas para despertarlos son: el sisop elige uno thread y lo despierta. Ese thread intenta adquirir el lock, el sisop despierta a todos los threads dormidos, y todos compiten nuevamente para obtener el lock. Se utiliza cuando: la espera será larga (espera que el disco lea bloques) y no queremos bloquear el acceso al CPU para los otros procesos, en sistemas de un solo procesador, donde un spinlock sería ineficiente. Utiliza un spinlock interno, en xv6.

Deadlock: es cuando dos o más threads se bloquean mutuamente porque cada uno espera un recurso que otro ya está usando, y ninguno puede avanzar.

FileSystem I

Un **File System** permite a los usuarios organizar sus datos de manera que persistan a lo largo del tiempo. Un file system es una abstracción del sistema operativo que provee datos persistentes con un nombre.

- **Datos Persistentes:** Se almacenan hasta que son explícitamente borrados, incluso ante fallos de energía.
- **Nombres:** Permite acceder a los datos mediante identificadores significativos asociados a los archivos. Esta identificación facilita el uso compartido de información entre programas.

Decisiones de Diseño del File System de Unix:

El sistema de archivos de Unix tomó decisiones de diseño: Estructura Jerárquica, Tratamiento Consistente de Archivos, Creación y Borrado de Archivos, Crecimiento Dinámico de Archivos, Protección de Datos, Dispositivos como Archivos.

Archivos:

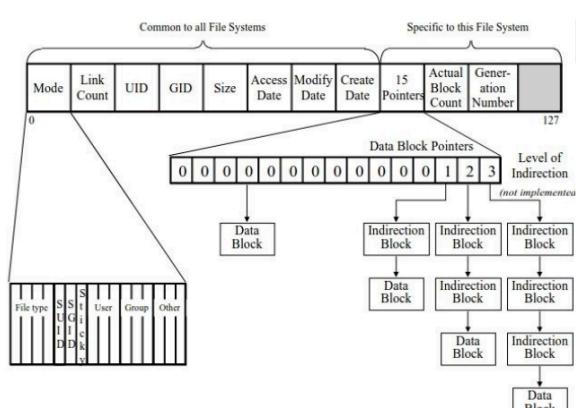
Un archivo es una colección de datos con un nombre específico. Proporciona una abstracción de alto nivel sobre los bloques de almacenamiento. Recordar un nombre de archivo es mucho más sencillo. 3 permisos: read, write y execute. 3 clases de usuarios: owner, grupo, demás.

Componentes de un Archivo:

- **Metadata:** Información sobre el archivo comprendida por el SO (tamaño, fecha de modificación, propietario, permisos, etc.).

- **Datos:** El contenido real del archivo, visto por el SO como un arreglo de bytes sin tipo.

Para ver la metadata: `ls -l`. Un archivo se implementa como un **dentry** que apunta a un **inodo**.



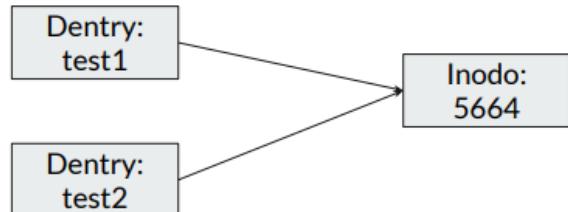
Inodo (Index Node): Almacena la metadata de un archivo, pero **no** su contenido. Tiene: tamaño del archivo, ID del propietario y del grupo, permisos (lectura, escritura, ejecución), tiempos de acceso, modificación y cambio del inodo, número de enlaces al archivo, punteros a los bloques de disco donde se almacenan los datos del archivo. El inodo contiene punteros a los bloques de datos, pero no los datos en sí.

Dentry (Directory Entry): Representa la relación entre el nombre de un archivo y su inodo correspondiente. Los directorios son listas de dentries. Su función principal es traducir nombres de archivos y rutas a sus inodos. Los inodos no están asociados a una estructura jerárquica; los dentries sí.

Directories:

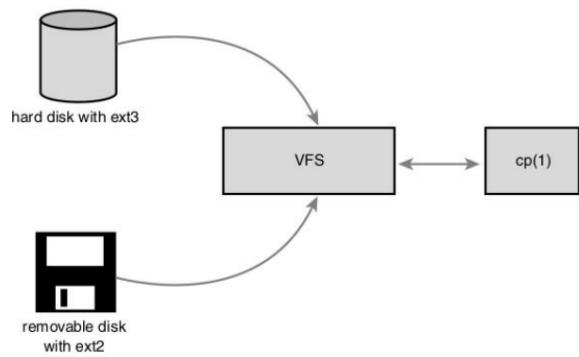
Son archivos especiales de tipo "directorio". Su contenido es una lista de dentries: un dentry apunta a un inodo no a dentries, un inodo apunta a datos pero no a otros inodos. Un dentry puede apuntar a un inodo de tipo directorio, cuyos datos contienen una lista de otros dentries, formando así la estructura jerárquica. Los directorios (inodos de tipo DIRECTORY) contienen dentries que apuntan a otros inodos (que a su vez pueden ser de tipo DIRECTORY).

Open Files (tabla de archivos abiertos): componente que conecta el filesystem con los procesos. Esta tabla vincula un **file descriptor** con el inodo que



representa el archivo. El file contiene el offset. Esto es el "cursor" que indica donde se va a escribir o leer a continuacion.

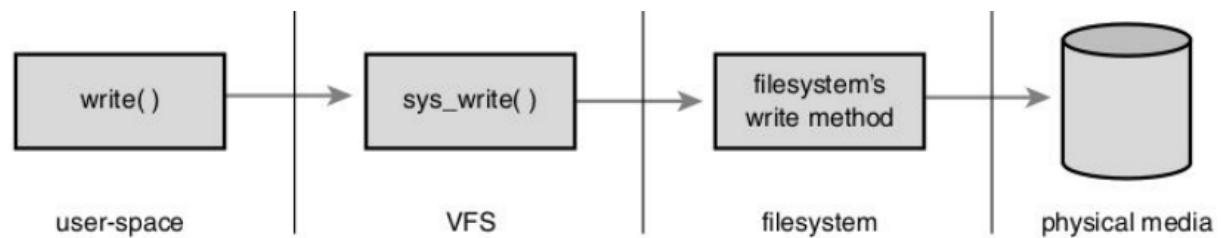
Virtual File System (VFS)



Es un subsistema del kernel que implementa la interfaz relacionada con los archivos y los sistemas de archivos que se presenta a los programas que se ejecutan en modo usuario. Todos los sistemas de archivos deben basarse en el VFS. Permite que los programas utilicen las llamadas al sistema de Unix para interactuar.

FileSystem Abstraction Layer:

Es una interfaz genérica para cualquier tipo de filesystem, capa de abstracción en el kernel que se sitúa entre esta interfaz y el sistema de archivos de bajo nivel. Permite que Linux soporte diferentes sistemas de archivos. VFS provee un modelo común de archivos que puede representar las características y el comportamiento general de cualquier sistema de archivos. Los filesystems adaptan su propia implementación de conceptos como "cómo abrir un archivo" para coincidir con las expectativas del VFS. El resultado es una capa de abstracción general que permite al kernel manejar muchos tipos de sistemas de archivos de forma fácil y limpia.



Objetos del VFS:

Serie de estructuras que modelan un filesystem. Estos **objetos** son:

superbloque: un sistema de archivos, **inode**: archivo individual, **dentry**: entrada de directorio, que es un componente simple de una ruta, **file**: archivo asociado a un proceso específico.

VFS: Directorio: un directorio es tratado como un archivo normal en Unix; no hay un objeto específico para directorios

VFS: Operaciones:

super_operations : Métodos que el kernel aplica sobre un sistema de archivos.

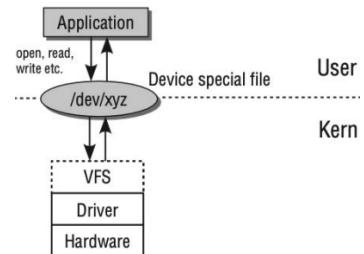
inode_operations : Métodos que el kernel aplica sobre un archivo específico.

dentry_operations : Métodos que el kernel aplica directamente a una entrada de directorio.

file_operations : el kernel aplica directamente sobre un archivo abierto por un proceso.

Dispositivos

Para acceder al hardware, el sistema operativo utiliza una estructura unificada: los dispositivos se representan como **archivos especiales** de tipo **DEVICE**, se utilizan las mismas **syscalls** que para operar con archivos normales. **Drivers**: encargados de implementar las operaciones de bajo nivel que realmente interactúan con el hardware específico.



Tipos de Dispositivos: dispositivos que se pueden vincular al filesystem (los de red existen pero no se vinculan directamente al filesystem):

- **Dispositivos de Carácter:** permiten la transferencia de datos byte a byte, no almacenan datos en bloques, adecuados para dispositivos que no requieren el procesamiento de grandes cantidades de datos a la vez (terminales, puertos serie, impresoras línea a línea).
- **Dispositivos de Bloque:** permiten la transferencia de datos en bloques de tamaño fijo, adecuados para hardware que requiere acceso eficiente a grandes cantidades de datos (discos duros, SSDs, memorias USB).

Vinculación entre los Dispositivos al Filesystem:

Los dispositivos de E/S tienen asociados un par de números. Estos números proporcionan un sistema básico de enrutamiento hacia los dispositivos.

- **Número Mayor:** Identifica el **driver** que maneja un grupo específico de dispositivos. El kernel lo usa para determinar qué controlador debe gestionar las peticiones de E/S.

- **Número Menor:** Identifica un **dispositivo específico** dentro del grupo gestionado por el mismo driver.

Filesystems Virtuales: No todos se corresponden con un dispositivo físico. Algunos son virtuales.

Filesystem API

Unix File Systems System Calls: Las llamadas al sistema relacionadas con archivos se pueden dividir en: las que operan sobre los archivos propiamente dichos, las que operan sobre los metadatos de los archivos

open() : Convierte el nombre de un archivo en una entrada en la tabla de descriptores de archivos del proceso y devuelve este descriptor. Toma flags para especificar el modo de apertura

creat() : como **open()** con los flags **O_CREAT | O_WRONLY | O_TRUNC**. Crea un inodo regular.

close() : Cierra un descriptor de archivo. Si el descriptor ya está cerrado, devuelve un error.

read() : Lee hasta un número dado de bytes desde un archivo, a partir de la posición actual del descriptor de archivo. Después de la lectura, el offset del archivo se incrementa en el número de bytes leídos. Devuelve núm de bytes realmente leídos.

write() : Escribe hasta una cantidad especificada de bytes desde un buffer a un archivo referenciado por un descriptor de archivo. Los bytes escritos puede ser menor que el solicitado.

lseek() : Repositiona el offset de un archivo abierto.

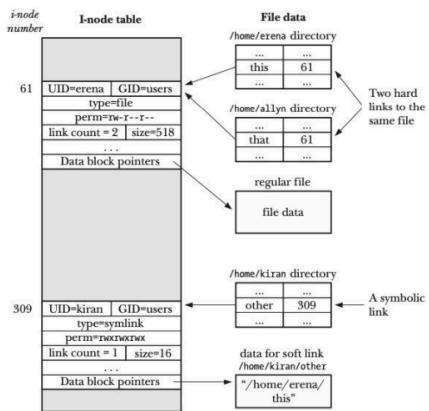
dup() y **dup2()** : Crean una copia de un descriptor de archivo (**oldfd**). Después de una llamada exitosa, el descriptor original y la copia son intercambiables y comparten la misma entrada en la tabla de archivos abiertos del sistema.

dup2(oldfd, newfd) hace lo mismo, pero en lugar de usar el siguiente descriptor disponible, intenta usar **newfd**.

Hard Links y Soft Links (Enlaces Duros y Simbólicos):

- **Hard Link:** Múltiples dentries apuntan al mismo inodo. Todos los nombres son válidos.
- **Soft Link (Symlink):** Un archivo especial que contiene la ruta al archivo de destino. Es un archivo separado con su propio inodo.

Diferencias:



Si hay discos distintos no se puede implementar el hard link. En cambio, el soft link si se puede ya que utiliza paths los cuales son globales, solo se apunta a una referencia.

Característica	Hard Link	Soft Link (Symlink)
El inodo apunta a:	Datos del archivo	Un bloque con la ruta del archivo
Relación con el archivo	Mismo inodo	Archivo separado
Tras borrar el destino	Sigue funcionando	Se rompe
Directorios	No se permite	Sí se permite
Diferentes particiones	No se permite	Sí se permite
Tamaño	No ocupa espacio extra	Ocupa como un archivo pequeño

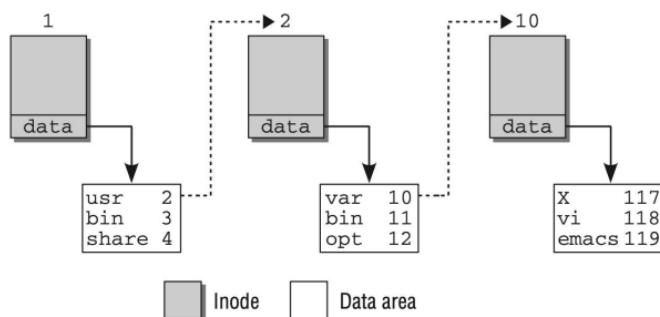
link() : Crea un nombre para un archivo existente, referencian exactamente el mismo archivo.

symlink() : Crea un soft link a un archivo. Toma la ruta del destino como argumento.

unlink() : Elimina un nombre de un archivo del filesystem. Si era el último link al archivo y ningún proceso lo tiene abierto, el archivo se borra completamente.

mkdir() : Crea un nuevo directorio.

Ejemplo /usr/bin/emacs:



API de Metadatos:

- `stat()` : Obtiene información (metadata) sobre un archivo y la guarda en una estructura.
- `access()` : Verifica si un proceso tiene permisos para acceder a un archivo con un pathname.
- `chmod()` : Cambia los bits de modo de acceso de un archivo.
- `chown()` : Cambia el propietario y/o el grupo de un archivo.

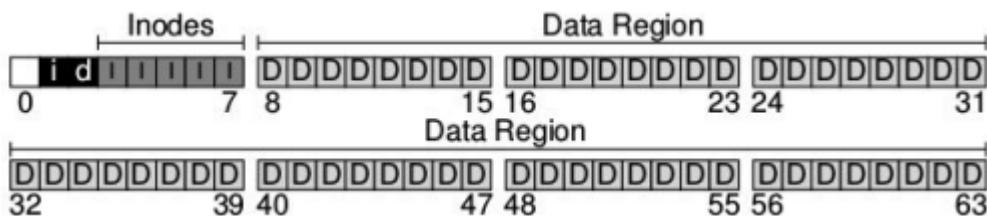
FileSystem II

Implementación de un Filesystem

Requisitos de Diseño Filesystem: abstracción de "archivo" sobre un medio de almacenamiento de bloques: una estructura de índice de archivos, localizar cada bloque que pertenece a un archivo, un mapa de espacio, saber qué bloques están disponibles para expandir archivos.

Organización General de vsfs:

1. **División en Bloques:** El disco se divide en bloques de tamaño fijo (4 KB ejemplo clases).
2. **Data Region:** Mayor parte del espacio en el filesystem almacena los datos de los usuarios.
3. **Inode Table:** Mantener la metadata de cada archivo se utiliza una estructura llamada **inode**. Los inodos se almacenan en el disco en una tabla llamada **inode table**.



1. **Bitmaps de Alocación (Allocation Bitmaps):** Para rastrear qué inodos y qué bloques de datos están libres o en uso, se utilizan **bitmaps**: un **inode bitmap** y un **data bitmap**.
2. **Super Bloque:** Un bloque especial que contiene metadatos sobre todo el filesystem: cant de inodos, cant de bloques, dónde comienza la tabla de inodos, dónde comienzan los bitmaps.

Casos de Estudio

Microsoft File Allocation Table (FAT): arreglo de entradas de 32 bits en un área reservada del volumen. Cada archivo corresponde a una **lista enlazada** de entradas en la FAT, donde cada entrada apunta a la siguiente. Los directorios mapean nombres de archivo a **números de archivo**. Dado el número de un archivo, se puede encontrar la primera entrada FAT y el primer bloque de datos. Siguiendo los punteros en la FAT, se pueden encontrar el resto de los bloques del archivo.

Unix Fast File System (FFS): es un árbol que permite localizar cualquier bloque de un archivo eficientemente. FFS emplea dos heurísticas de localidad: **colocación de grupos de bloques y espacio de reserva**, sirve para archivos grandes como pequeños.

Un inodo típico de FFS contiene 15 punteros. primeros 12 son **directos**, 1 un **puntero indirecto**, 14 un **dblemente indirecto**, 15 es un **triplemente indirecto**.

Buffer Cache

Funciones Principales del Buffer Cache: sincronizar el acceso a los bloques de disco para asegurar que solo haya una copia de un bloque en la memoria y que solo un hilo del kernel use esa copia a la vez, almacena en caché bloques populares.

Relación con Estructuras de Datos: utiliza los **B-Trees** para organizar los metadatos y la ubicación de los datos en el disco. Los B-Trees son eficientes

Primitivas de Sincronización y Persistencia:

- **mmap()** y **msync()** : **mmap()** permite mapear un archivo directamente en el espacio de direcciones de un proceso, permitiendo el acceso a los datos como si estuvieran en memoria. **msync()** permite sincronizar solo la parte del archivo mapeado que interesa.