- El inodo de un archivo contiene metadatos críticos sobre el archivo, como sus atributos y punteros a sus bloques de datos.

- Las regiones de datos se dividen en bloques de datos mucho más grandes (normalmente de 8 KB o más) , dentro de los cuales el sistema de archivos almacena datos de archivos y metadatos de directorio

- Las entradas de directorio contienen nombres de archivo y punteros a inodos;

- Hardlink: se dice que un archivo tiene un hardlink si varias entradas de directorio en el sistema de archivos se refieren al inodo de ese archivo. (NO hay en JOS) --> nuestro sistema de archivos no usará inodos en absoluto y, en su lugar, simplemente almacenará todos los archivos (o subdirectorios) metadatos dentro de la (única) entrada de directorio que describe ese archivo

- Los archivos y los directorios consisten lógicamente en una serie de bloques de datos, que pueden estar dispersos por todo el disco

- El FS presenta interfaces para leer y escribir secuencias de bytes en desplazamientos arbitrarios dentro de los archivos

- El tamaño del sector es una propiedad del hardware del disco, mientras que el tamaño del bloque es un aspecto del sistema operativo que usa el disco. --> IMPORTANTE: El tamaño de bloque de un sistema de archivos debe ser un múltiplo del tamaño del sector del disco subyacente.

- Superbloque: contiene metadatos que describen las propiedades del sistema de archivos en su conjunto, como el tamaño del bloque. , el tamaño del disco, los metadatos necesarios para encontrar el directorio raíz, la hora en que se montó el sistema de archivos por última vez, la hora en que se comprobó por última vez que no había errores en el sistema de archivos, etc. --> JOS tiene UN superbloque.

- Metadata de un archivo: Estos metadatos incluyen el nombre del archivo, el tamaño, el tipo (archivo normal o directorio) y punteros a los bloques que componen el archivo (editado)

- En JOS, los sectores son de 512 bytes cada uno

- En JOS el array f_direct de la struct File contiene espacio para almacenar los números de bloque de los primeros 10 (NDIRECT) bloques del archivo, a los que llamamos bloques directos del archivo. [tamaño maximo del archivo 10 * 4096 = 40 KB]. Para archivos mas grandes, asignamos un bloque de disco adicional, denominado bloque indirecto del archivo, para contener hasta 4096/4 = 1024 números de bloque adicionales. Por lo tanto, nuestro sistema de archivos permite que los archivos tengan un tamaño de hasta 1034 bloques, o un poco más de cuatro megabytes.

- Directorios vs Archivos Regulares:
(1) Un struct File en nuestro sistema de archivos (JOS) puede representar un archivo normal o un directorio; se distinguen por el type.

(2) El sistema de archivos gestiona los archivos regulares y los archivos de directorio exactamente de la misma manera.
(3) El FS no interpreta en absoluto el contenido de los bloques de datos asociados con los archivos regulares, mientras que el sistema de archivos interpreta el contenido de un archivo de directorio como una serie de strcut File que describen los archivos y subdirectorios dentro del directorio.
(4) El superbloque en nuestro sistema de archivos contiene una struct File (el campo root en struct Super) que contiene los metadatos para el directorio raíz del sistema de archivos. El contenido de este archivo de directorio es una secuencia de struct File que describen los archivos y directorios ubicados dentro del directorio raíz del sistema de archivos. Cualquier subdirectorio en el directorio raíz puede, a su vez, contener más struct File que representen subdirectorios, y así sucesivamente. (editado)

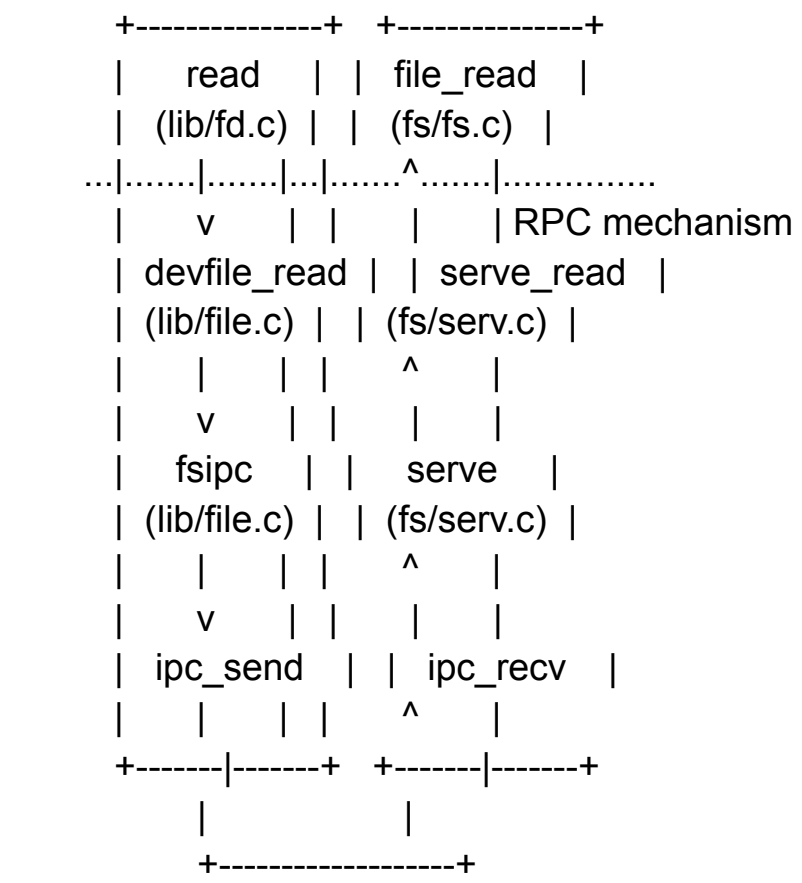
- Acceso al disco: El procesador x86 usa los bits IOPL en el registro EFLAGS para determinar si el código de modo protegido puede realizar instrucciones de E/S de dispositivos especiales, como las instrucciones IN y OUT. Dado que todos los registros del disco IDE a los que necesitamos acceder están ubicados en el espacio de E/S del x86 en lugar de estar asignados a la memoria, otorgar "privilegios de E/S" al entorno del sistema de archivos es lo único que debemos hacer para permitir que el sistema de archivos acceda a estos registros. En efecto, los bits IOPL en el registro EFLAGS proporcionan al núcleo un método simple de "todo o nada" para controlar si el código de modo de usuario puede acceder al espacio de E/S. En nuestro caso, queremos que el entorno del sistema de archivos pueda acceder al espacio de E/S, pero no queremos que ningún otro entorno pueda acceder al espacio de E/S. Entonces:

```
if (type == ENV_TYPE_FS) {
        env->env_tf.tf_eflags |= FL_IOPL_3;
    }
```

- Cache de bloques: se implementa el "caché de búfer" simple (en realidad solo un caché de bloque) con la ayuda del sistema de memoria virtual del procesador. Nuestro sistema de archivos se limitará a manejar discos de 3 GB o menos. Reservamos una gran región fija de 3 GB del espacio de direcciones del entorno del sistema de archivos, desde DISKMAP hasta DISKMAP+DISKMAX, como una versión del disco con "memoria asignada". llevaría mucho tiempo leer todo el disco en la memoria, por lo que implementaremos una forma de paginación por demanda (lazy) , en la que solo asignamos páginas en la región del mapa del disco y leemos el bloque correspondiente del disco en respuesta a una falla de página bc_pgfault en esta región.

- La función flush_block debe escribir un bloque en el disco si es necesario. Dicho esto, flush_block no debería hacer nada si el bloque ni siquiera está en el caché de bloques o si no está sucio. Usaremos el hardware de la máquina virtual (MMU) para realizar un seguimiento de si un bloque de disco se ha modificado desde la última vez que se leyó o se escribió en el disco. Para ver si un bloque necesita escribirse, podemos mirar si el PTE_D bit "sucio" está configurado en la uvpt entrada. Después de inicializar el caché de bloques, simplemente almacena punteros en la región del mapa del disco en la variable global super. Después de este punto, podemos simplemente leer de la struct super como si estuvieran en la memoria y nuestro controlador de errores de página los leerá del disco según sea necesario. (editado)

- Bitmap: Después fs_init de setear el bitmap pointer, podemos tratar al bitmap como una matriz empaquetada de bits, uno para cada bloque en el disco.
- Interfaz del sistema de archivos: ya que los demas entornos no pueden llamar directamente a funciones del entorno del FS, se expone un acceso al entorno del FS mediante remote procedure call (RPC), construida sobre el mecanismo IPC de JOS.



Todo lo que esta por debajo de la linea punteada del RPC, es simplemente la mecánica de obtener una solicitud de lectura del entorno normal al entorno del FS.
El procedimiento: (1) read funciona en cualquier descriptor de archivo y simplemente envia a la funcion de lectura del dispositivo apropiado, en este caso a devfile_read. (2) devfile_read implementa read especificamente para archivos en disco. Esta como las demas devfile_* implementan el lado del cliente de las operaciones del FS (todas las funciones mas o menos de la misma forma, agrupando argumentos en una estrctura de solicitud) (3) Llama a fsipc para enviar la solicitud IPC y desempaqueta y devuelve los resultados. La función fsipc simplemente maneja los detalles comunes de enviar una solicitud al servidor y recibir la respuesta. (4) Realiza un bucle en la función serve, recibe una solicitud sin fin a través de IPC, envía esa solicitud a la función de controlador adecuada y envía el resultado de vuelta a través de IPC. En el ejemplo de lectura, serve se enviará a serve_read, que se encargará de los detalles de IPC específicos para las solicitudes de lectura, como desempaquetar la estructura de la solicitud y, finalmente, llamar file_read para realizar la lectura del archivo.

Recordar que el mecanismo IPC de JOS permite que un entorno envíe un solo número de 32 bits y, opcionalmente, comparta una página. Para enviar una solicitud del cliente al servidor, usamos el número de 32 bits para el tipo de solicitud (los RPC del servidor del sistema de archivos están numerados, al igual que las llamadas al sistema). y almacenamos los argumentos de la solicitud en un union Fsipcen la página compartida a través de la IPC. Del lado del cliente, siempre compartimos la página en fsipcbuf; en el lado del servidor, mapeamos la página de solicitud entrante en fsreq. El servidor también devuelve la respuesta a través de IPC. Usamos el número de 32 bits para el código de retorno de la función. ara la mayoría de los RPC, esto es todo lo que devuelven. FSREQ_READ y FSREQ_STAT también devuelven datos, que simplemente escriben en la página en la que el cliente envió su solicitud. No es necesario enviar esta página en el IPC de respuesta, ya que el cliente la compartió con el servidor del sistema de archivos en primer lugar. Asimismo, en su respuesta, FSREQ_OPEN comparte con el cliente una nueva “página Fd”.
- Spawning Processes: spawn() crea un nuevo entorno, carga una imagen de programa desde el FS y luego inicia el entrno secundario que ejecuta este programa. El proceso principal luego continúa ejecutandose independientemente del proceso secundario. Funciona como un fork() + exec() en UNIX. En JOS se implementa el "exec()" desde el espacio del usuario. - Descriptores de archivos: En JOS, cada uno de estos tipos de dispositivos tiene un correspondiente struct Dev, con punteros a las funciones que implementan lectura/escritura/etc. para ese tipo de dispositivo. Cada uno struct Fd indica su tipo de dispositivo. En lib/fd.c se mantiene la region de la tabla de descriptores de archivo en el espacio de direcciones de cada entorno de aplicación, comenzando en FDTABLE. Esta área reserva el valor de una página (4 KB) de espacio de direcciones para cada uno de los MAXFD descriptores de archivo (actualmente 32) que la aplicación puede tener abiertos a la vez. Una página de tabla de descriptor de archivo en particular se asigna si y solo si el descriptor de archivo correspondiente está en uso. Cada descriptor de archivo también tiene una "página de datos" opcional en la región que comienza en FILEDATA , que los dispositivos pueden usar si así lo desean. El estado del descriptor de archivo se mantiene en la memoria del espacio de usuario. En fork, la memoria se marcará como copia en escritura, por lo que el estado se duplicará en lugar de compartirse. En spawn, la memoria quedará atrás, no se copiará en absoluto. (Efectivamente, el entorno generado comienza sin descriptores de archivos abiertos). En lugar de codificar una lista de regiones en alguna parte, estableceremos un bit que de otro modo no se usaría en las entradas de la tabla de páginas. Estableceremos la convención de que si una entrada de la tabla de páginas tiene este bit establecido, el PTE debe copiarse directamente de padre a hijo tanto en como fork en spawn.

- La interfaz del teclado: En QEMU, la entrada escrita en la ventana gráfica aparece como entrada desde el teclado a JOS, mientras que la entrada escrita en la consola aparece como caracteres en el puerto serie.