



Sistema de Ficheros

Mejora de rendimiento. Buffers de E/S





Lecturas aconsejadas

- Linux System Programming (R. Love)
 - Capítulos 2 y 3



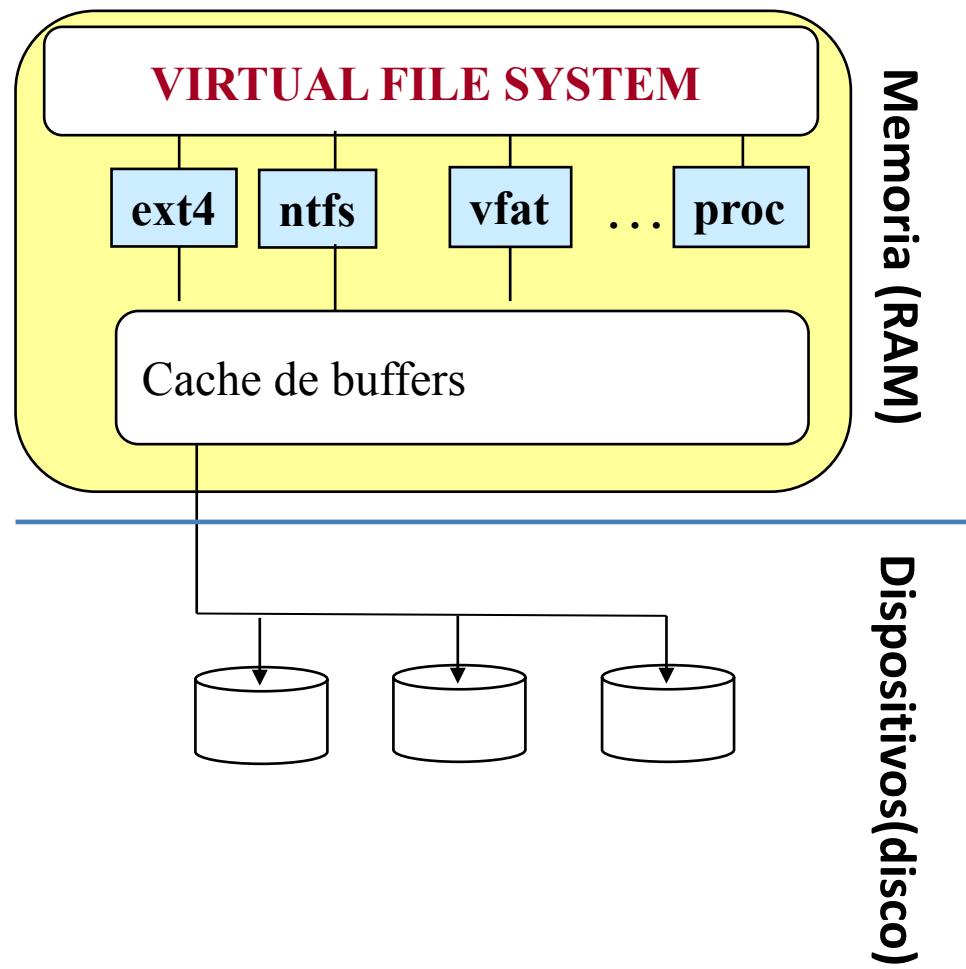
Introducción

- El acceso a disco (u otro dispositivo de almacenamiento) es extremadamente lento
 - Órdenes de magnitud respecto a un acceso a memoria principal
- Si cada solicitud de lectura/escritura supusiese un acceso al dispositivo físico, el rendimiento sería bajísimo
- ¿Ideas?
 - Reutilizar el concepto de *cache* de EC



Cache de bloques/buffers

■ Copiamos bloques de disco en memoria



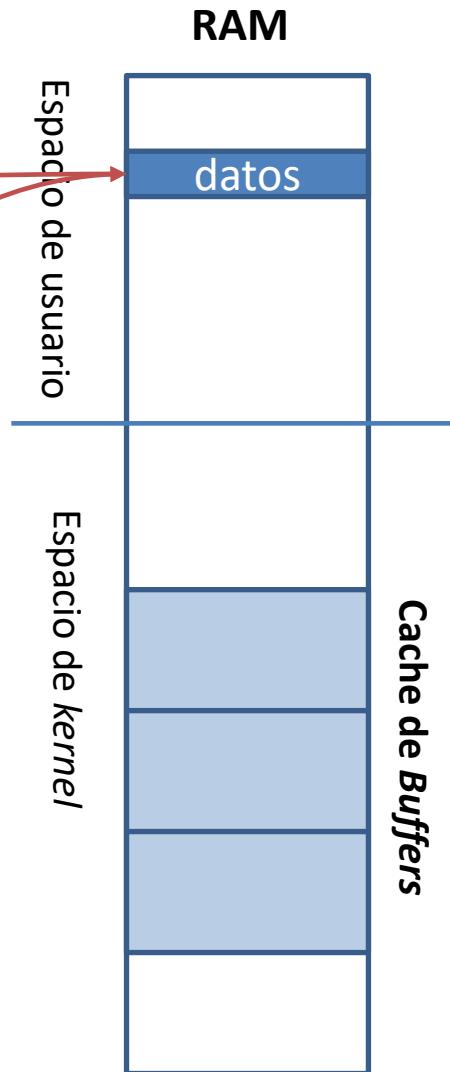
- Cada vez que se realiza un acceso a un fichero (*read/write*)...
 - Se calcula en qué bloque está la información que queremos
 - Se comprueba si ya está en la *cache*. Si es así, finaliza el acceso
 - Si no, se accede a disco y traemos un bloque completo, no sólo lo que pide el usuario



Ejemplo de escritura (*delayed write*)

- Las operaciones con disco NO se realizan cuando las solicitamos

```
char datos[128];  
  
int main() {  
    datos = ...;  
    fd=open(...);  
  
    write(fd,datos,128);  
...}
```

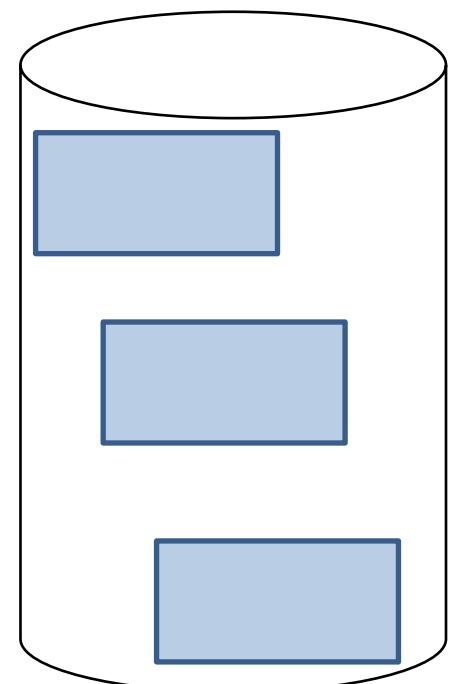
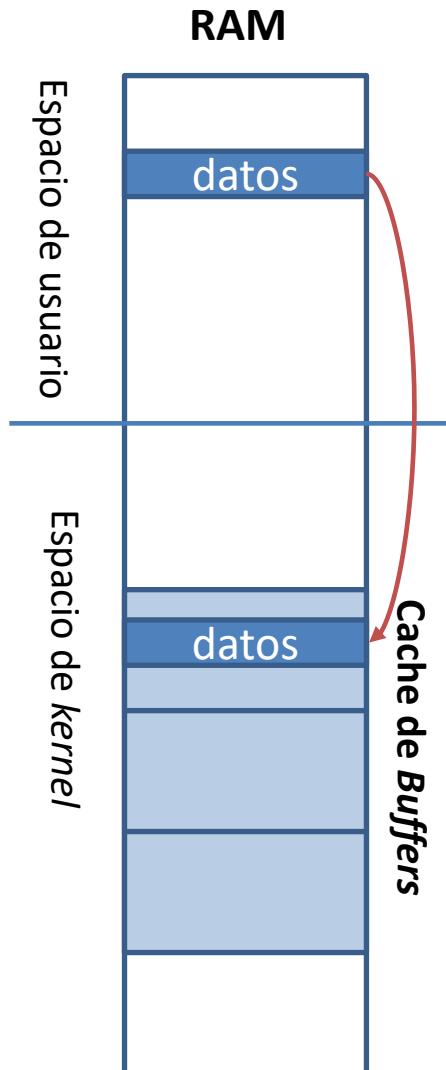




Ejemplo de escritura (*delayed write*)

- Las operaciones con disco NO se realizan cuando las solicitamos

```
char datos[128];  
  
int main() {  
    datos = ...;  
    fd=open(...);  
  
    write(fd,datos,128);  
    ...  
}
```

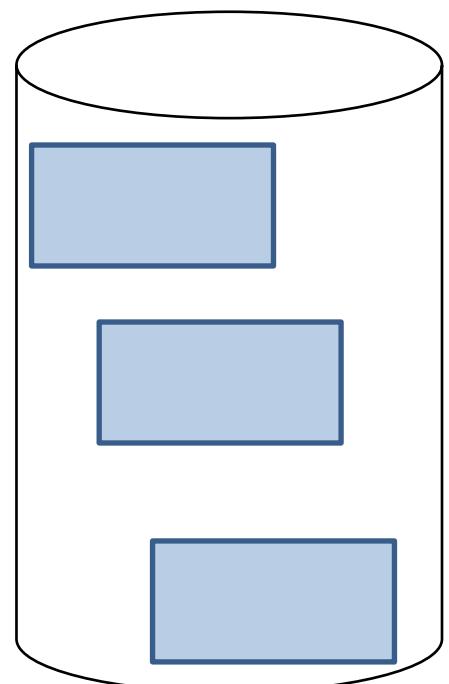
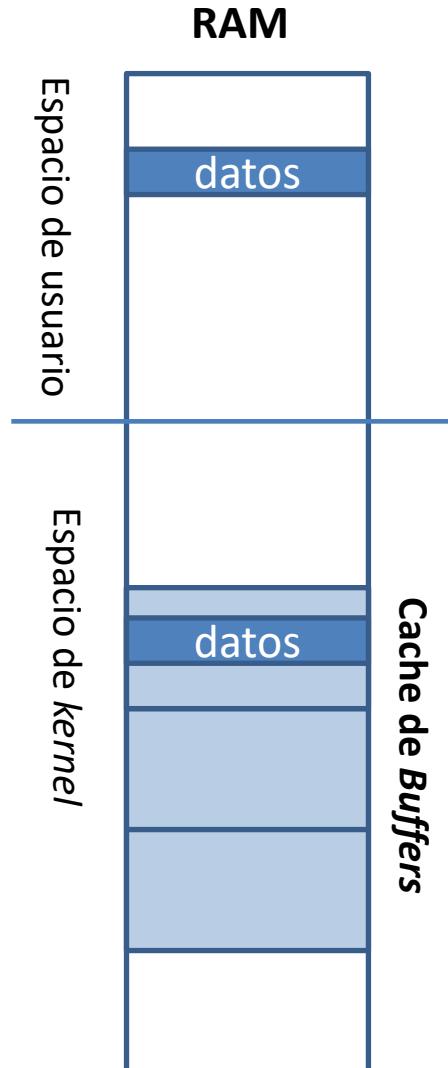




Ejemplo de escritura (*delayed write*)

- Las operaciones con disco NO se realizan cuando las solicitamos
 - Cuando el SO decida, actualiza el disco

```
char datos[128];  
  
int main() {  
    datos = ...;  
    fd=open(...);  
  
    write(fd,datos,128);  
    ...  
}
```





¿Problemas del *delayed write*?

- ¿Qué pasa si se va la luz?
- ¿Cómo se notifica un fallo de escritura?
- ¿En qué orden ocurren realmente las escrituras?
 - ¿Y si hay una lectura posterior a la escritura actual?
- El sistema operativo marca una *edad* máxima para un *buffer* antes de enviarlo a disco
 - Antes de que expire ese tiempo, se envía el *buffer* a disco
 - Se puede configurar en
`/proc/sys/vm/dirty_expire_centisecs`
- También es posible forzar que se escriba un determinado *buffer*
 - O incluso hacer que todas las escrituras sean *síncronas*



E/S sincronizada

- Para forzar la escritura de datos y metadatos:
 - `int fsync(int fd);`
 - `int fdatasync(int fd);`
- La segunda puede ser más rápida porque no garantiza que metadatos no esenciales se escriban en ese momento
- Códigos de Error
 - **EBADF** el descriptor de fichero no es válido para escritura
 - **EINVAL** el descriptor de fichero no soporta sincronización
 - **EIO** error de E/S a bajo nivel
- Otras alternativas
 - `void sync(void)` para sincronizar todos los buffers
 - `O_SYNC` en `open()` equivalente a llamar a `fsync` tras cada `write`
 - `O_DIRECT` en `open()` supone prescindir de todas las capas del E/S. Impone muchas restricciones al uso de `read/write`.

<unistd.h>

SV+POSIX

Entrada/Salida en memoria



<sys/mman.h>

```
void *mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off )
```

- **addr** → dirección en la que se mapeará el fichero. Se puede dejar a 0 y la elije el sistema.
- **prot** → OR de PROT_READ PROT_WRITE PROT_EXEC
- **filedes** → descriptor de fichero (que debe estar abierto ya)
- **len-off** → permiten indicar qué parte del fichero se lleva a memoria
- **flag** → MAP_SHARED | MAP_PRIVATE permite indicar si la región es una copia privada (no afecta al fichero original) o compartida

- A partir de ese punto, podemos acceder al fichero como si fuera un *array*
- Si estamos modificando un fichero, nos debemos asegurar de que la escritura se lleva a cabo llamando explícitamente a **munmap()** o **msync()**



Buffers en espacio de usuario

- Los *buffers (page cache)* del *kernel* mejoran enormemente el rendimiento
 - Pero aún queda sobrecarga
 - Considera el código

```
...
char c;
fdi = open("entrada",...);
fdo = open("salida"),...);
while ( read(fdi,&c,1) )
    write(fdo, &c,1);
```

- ¿Qué se podría mejorar? ¿Dónde está la sobrecarga?



Buffer en espacio de usuario

- Podemos crear una nueva copia de cada bloque en el espacio de usuario, evitando llamadas al sistema para cada pocos bytes
 - Así se implementa en **la librería estándar**

```
FILE*    fopen(const char path, const char *acesto);  
FILE*    fdopen(int fd, const char *mode);  
int      fclose( FILE *fp);  
int      fseek( FILE *fp, long offset, int whence);  
size_t   fread(void *ptr, size_t itemsize, size_t  
items, FILE *fp);  
size_t   fwrite(void *ptr, size_t itemsize, size_t  
items, FILE *fp);
```

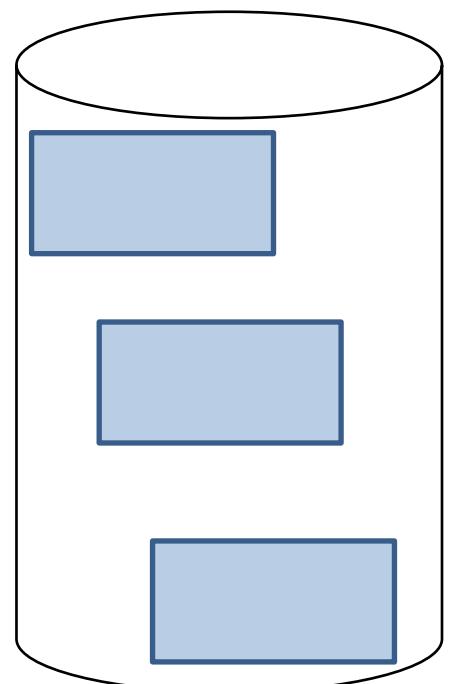
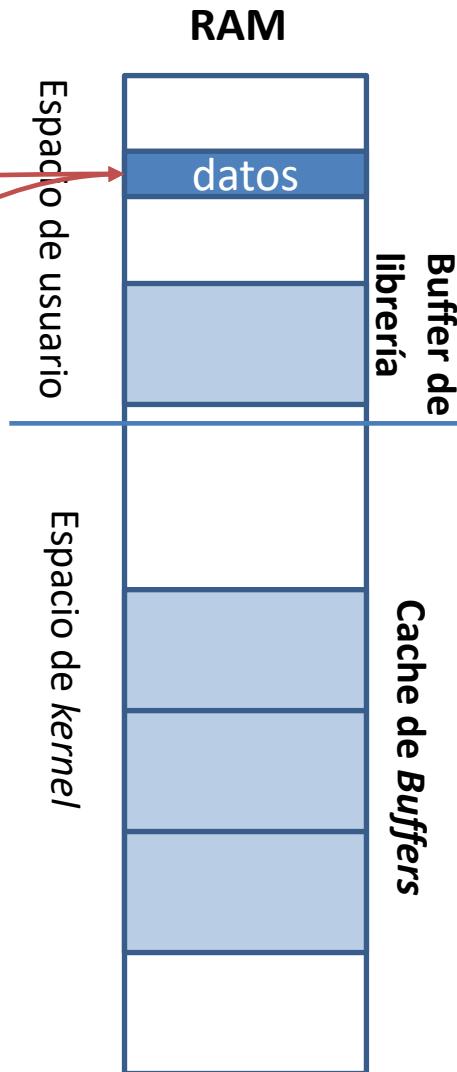
- Otras: `fgetc()`, `fgets()`, `ungetc()`, `fputc()`, `fputs()`, `rewind()`, `ftell()`, `feof()`, `ferror()`, `fileno()`
- Familia de Funciones `*printf`, `*scanf`.



Escritura con librería estándar

- Tenemos un nuevo nivel de *buffering*

```
char datos[128];  
  
int main() {  
    datos = ...;  
    fd=fopen(...);  
  
    fwrite(fd,datos,...);  
    ...  
}
```

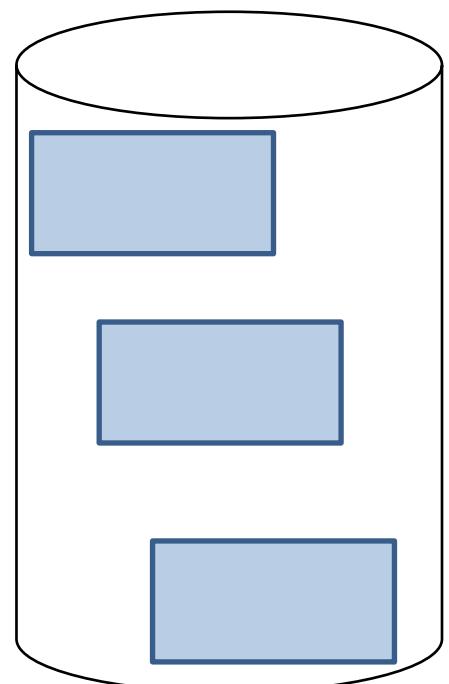
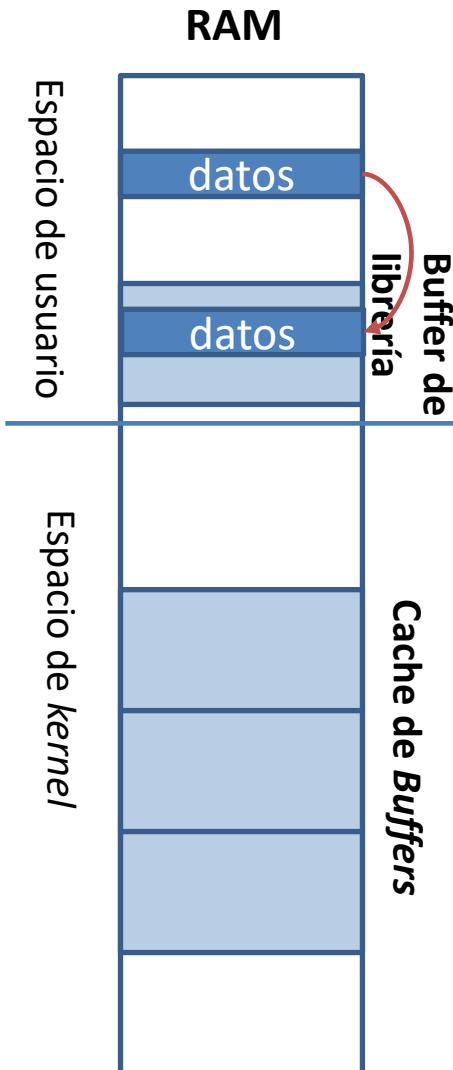




Escritura con librería estándar

- Al llamar a `fwrite()` escribimos en espacio de usuario
 - Sólo cuando sea necesario, escribimos todo ese *buffer* en la parte del *kernel*

```
char datos[128];  
  
int main() {  
    datos = ...;  
    fd=fopen(...);  
  
    fwrite(fd,datos,...);  
    ...  
}
```





Estrategias de buffering de librería estándar

- *Unbuffered* : los bytes se transmiten individualmente tan pronto como sea posible
- *Line buffered*: los bytes se transmiten al fichero cuando se detecta un carácter de fin de línea ('\n')
- *Fully buffered*: los bytes se transmiten en bloques del tamaño escogido
 - Forzar el volcado (*flushing*): **fflush(FILE* stream)**

Cuestión: en UNIX todo es un fichero... incluido la salida estándar. ¿Qué efecto tiene esto en el comportamiento de una función como *printf()*? 

Lectura interesante: uso de *streams* (*FILE*) en C

https://www.gnu.org/software/libc/manual/html_node/I_002fO-on-Streams.html#I_002fO-on-Streams

Más específico sobre *buffers*:

https://www.gnu.org/software/libc/manual/html_node/Stream-Buffering.html#Stream-Buffering

Cuestión: ¿Cuál es la diferencia entre *fsync()* y *fflush()*? 



Elegir la estrategia de *buffering*

- Tras abrir un *stream* (FILE) y ANTES de hacer cualquier operación sobre él, se puede especificar qué buffering se quiere usar

```
int setvbuf (FILE *stream, char *buf, int mode, size_t size)


- stream es el puntero que devuelve fopen
- mode
  - _IOFBF → full buffering
  - _IOLBF → line buffering
  - _IONBF → unbuffered
- buf debe ser un array de caracteres capaz de albergar al menos size caracteres. Se puede dejar a NULL para que setvbuf se encargue de hacer malloc
- size es el tamaño que queremos para el buffer si usamos full buffering.
  - Aconsejable usar la macro BUFSIZ

```