# Informe - Proyecto Final - Memory Map
# Sistemas Operativos 2023

Gonzalez Ignacio - Nolasco Agustín
43.131.850 - 42.142.489

A continuación se detallan las decisiones de diseño tomadas a la hora de implementar las llamadas al sistema `mmap` y `munmap`. Además, de comentar algunos detalles de implementación.

Los perfiles de las llamadas al sistema son los siguientes:

- `void* mmap(int fd)`
  donde `fd` es el descriptor del archivo el cual queremos mapear en memoria. La llamada al sistema retorna la dirección de memoria base de la primera página del mapeo. En caso de que ocurra algún error la llamada al sistema retornara `MAP_FAILED`. Las razones por las cuales puede fallar son que el descriptor del archivo sea invalido, que el archivo tenga tamaño `0` o que no haya un *slot* disponible en el proceso para alojar el nuevo mapeo.

- `int munmap(void *addr)`
  donde `addr` es la dirección base de la primera página de un mapeo, en principio debería ser la misma dirección que fue retornada por la llamada `mmap`. La llamada al sistema devuelve un `int` que nos dice si la operación finalizó con éxito (`0`) o no (`-1`). La razón por la cual puede fallar es que la dirección `addr` no corresponde a una dirección base de un mapeo realizado previamente.

Ahora se detallan algunas decisiones de diseño tomadas para la implementación de las llamadas al sistema definidas anteriormente.

La constante `MAP_FAILED` utilizada para representar el fallo en la operación de `mmap`, se encuentran definida en `kernel/fcntl.c`.

Se agrego una macro `PTE_D` en `kernel/riscv.h` la cual define una mascara que nos permite extraer el bit dirty de la `pte` para verificar si una pagina fue modificada, y así determinar si debe ser escrita en disco al hacer `munmap`.

Las implementaciones de rutinas auxiliares se encuentran en `kernel/mapfile.c`, y se detallan a continuación:

- `int getmd(uint64 addr)`
  dada una dirección virtual, retorna el *map descriptor* al que pertenece, en caso de que la dirección no corresponda a ninguno devuelve `-1`. Esta rutina es utilizada para determinar si una dirección pertenece a un mapeo a la hora de capturar una

excepción, así como también en la llamada `munmap` para determinar que mapeo desallocar.

- `int mfilealloc(struct proc *p, int fd)`
  dado un proceso `p` y un *file descriptor* `fd`, busca un *slot* disponible en el proceso para allocar el mapeo del archivo, luego extiende el tamaño del proceso, incrementa el *contador de referencias* del `inode` correspondiente al archivo y nos devuelve la dirección base de la primera página del mapeo. En caso de que no haya un *slot* disponible retorna `-1`. Esta rutina es utilizada principalmente en la llamada al sistema `mmap`.

- `int loadblock(struct proc *p, int md, uint64 va)`
  dado un proceso `p`, un *map descriptor* `md` y una dirección virtual `va`, se carga en memoria el contenido de una pagina (`4KB`) del archivo que se corresponda con la `va` dada y se lo mapea en la pagina correspondiente del proceso `p`, en caso de tener éxito o fallar retornara `0` o `-1` respectivamente. Esta rutina se utiliza cuando se produce una excepción por *page fault* y se debe cargar el contenido del archivo en memoria.

- `void applymodif(struct mapfile *mf, pagetable_t pagetable, uint64 va)`
  dado el `mapfile`, la `pagetable` del proceso actual y la dirección base `va` del mapeo a desallocar, carga los bloques que fueron modificados en disco. Esta rutina se utiliza en la llamada al sistema `munmap`, siempre y cuando el mapeo se haya realizado sobre un archivo abierto en modo escritura (`writable`).

A los procesos se les agregó un nuevo campo `mfile[NOMAP]` de tipo `struct mapfile`. El cual es usado para saber que archivos fueron mapeados por el proceso. La constante `NOMAP` fue definida en `kernel/params.h`. El tipo `struct mapfile` está formado por los siguientes campos:

- `va` representa la dirección base de la primera página del mapeo.
- `ip` es el `inode` del archivo que fue mapeado.
- `size` es el tamaño del archivo cuando fue mapeado, esto se guarda ya que este puede verse modificado por otro proceso.
- `writable` es el permiso de escritura. Utilizado para determinar si al hacer un `munmap` debemos guardar los cambios o no.

Todos estos cambios se realizaron en `kernel/proc.h`.

Debido a que los datos son cargados bajo demanda, la primera vez que se solicitan los datos de una página particular, ya sea para lectura o escritura, se producirá una excepción debido a su ausencia. Por lo tanto, hemos tenido que modificar la rutina `usertrap`, que se encuentra en `kernel/trap.c`, para que, en caso de tratarse de un acceso a una dirección de memoria correspondiente a un mapeo válido, se llame a la rutina encargada de cargar los datos del disco a memoria (`loadblock`) y reintentar la operación realizada por el usuario que ocasionó la excepción.

Además de realizar todos los cambios pertinentes para el agregado de nuevas llamadas al sistema, se añadió en `kernel/sysfile.c` dos rutinas:

- `sys_mmap(void)` encargada de registrar el nuevo mapeo en el campo `mfile[NOMAP]` del proceso invocante.
- `sys_munmap(void)` encargada de remover el mapeo solicitado del campo `mfile[NOMAP]` del proceso invocante, a su vez que se hace el guardado de las modificaciones, si las hubo y el archivo mapeado era `writable`. También se decrementa el *contador de referencias* del `inode` del mapeo, para finalmente liberar la memoria utilizada en caso de haberla.

Detalles de la implementación:

- Una vez que un archivo mapeado es desmapeado quedará un *bache* en el espacio de direcciones virtuales, pero la memoria física si será efectivamente liberada. No tomamos mayor complicación debido a que si, por ejemplo, se mapean dos páginas y luego se solicitara más memoria de forma dinámica, al liberar el área de memoria del mapeo tendríamos que mover la nueva información tantas páginas como hayamos desallocado, hacia arriba. Una solución podría ser la de llevar de alguna manera las páginas libres. Otra sería tener un área especial de memoria para los mapeos (`map regions`). De momento lo que hacemos es aumentar el `size` del proceso según cuantas páginas mapeamos, pero nunca lo decrementamos al hacer el `munmap`.
- Actualmente siempre damos los permisos de lectura y escritura a las paginas que cargamos. Pero a la hora de hacer el `munmap` solo guardamos las modificaciones si el archivo que se mapeo estaba abierto en modo escritura (`writable`), en otro caso las modificaciones son ignoradas.
- En el `exit()` agregamos la limpieza del campo `mfile[NOMAP]`, así como también el decremento del *contador de referencias* al `inode` correspondiente al igual que en la llamada al sistema `munmap`.
- En la llamada `fork()` hacemos que se copie el campo `mfile[NOMAP]`, para que los procesos hijos hereden los archivos mapeados. La copia del espacio de memoria del proceso padre al hijo ya se hace por defecto. Además, incrementamos el *contador de referencias* del `inode` correspondiente de cada mapeo.
- Al guardarnos el `inode` del archivo mapeado, podemos independizarnos de si el archivo sigue abierto, o no, por el proceso. De esa forma, el mapeo es independiente del cierre (`close`) del archivo mapeado.
- Los directorios (`ip->type == T_DIR`) no pueden ser mapeados, al igual que en UNIX.
- Se modificaron las rutinas `uvmunmap` y `uvmcopy` en `kernel/vm.c` para que no se produzca un `panic` al momento de desmapear o copiar una pagina que no tiene el bit `V` encendido, ya que esta `pte` podría ser memoria reservada para un mapeo que todavía no fue cargado.