get_next_line – Dossier de defensa

Proyecto 42

28 de octubre de 2025

1. Contexto general

- Objetivo: implementar `get_next_line(int fd)` que devuelve la siguiente línea disponible (incluye el `\n` si existe) en cada llamada.
- Todas las lecturas pasan por `read(2)` y usan un `BUFFER_SIZE` definido en compilación.
- Reto principal: mantener entre llamadas el texto leído que aún no se devolvió (estado persistente).
- Dos variantes: versión obligatoria (un descriptor) y bonus (varios descriptores en paralelo).

2. Funcionamiento interno

1. Se invoca `get_next_line(fd)`.
2. Se acumula texto en un buffer estático `tmp` leyendo bloques de `BUFFER_SIZE` hasta encontrar `\n` o EOF.
3. Se construye la línea a devolver (hasta el primer `\n` incluido, si existe).
4. Se actualiza `tmp` para conservar solo el sobrante posterior a la línea devuelta.
5. La siguiente llamada continúa desde ese estado.

3. Estructura del repositorio

- `get_next_line.c`: lógica principal para un descriptor (incluye `read_file`, `get_new_line`, `get_reminder` y la API `get_next_line`).
- `get_next_line_bonus.c`: misma lógica con `static char *tmp[OPEN_MAX];` (un buffer por descriptor).
- `get_next_line_utils.c` y `_utils_bonus.c`: `ft_strchr`, `ft_strlen`, `ft_strjoin`.
- `get_next_line.h` y `_bonus.h`: prototipos y macros (`BUFFER_SIZE`, `OPEN_MAX`).

4. Versión obligatoria (`get_next_line.c`)

- Estado único: `static char *tmp;`.
- Validación inicial: `fd <0` o `BUFFER_SIZE` $\leq 0 \Rightarrow$ `NULL`.
- `read_file: bucle de read + ft_strjoin hasta detectar \n o EOF.`
- `EOF sin datos pendientes: libera memoria y devuelve NULL.`

Funciones clave

`read_file(int fd, char *tmp)`
: Lee bloques de `BUFFER_SIZE`, termina en `\0`, concatena en `tmp` y se detiene al ver `\n`, EOF o error (`read` = -1). En error, limpia y retorna `NULL`.

`get_new_line(char *tmp)`
: Reserva y copia desde el inicio hasta el primer `\n` (incluyéndolo si existe) o hasta `\0`.

`get_reminder(char *tmp)`
: Recorta `tmp` dejando únicamente lo posterior al `\n`; si no hay `\n`, libera y retorna `NULL`.

`get_next_line(int fd)`
: Orquesta: valida, llama a `read_file`, extrae línea, actualiza estado y maneja EOF/errores.

5. Versión bonus (`get_next_line_bonus.c`)

- Estado por descriptor: `static char *tmp[OPEN_MAX];`.
- Permite alternar `get_next_line(fd1)` → `get_next_line(fd2)` sin mezclar datos.
- Validación adicional: `fd <0`, `BUFFER_SIZE` $\leq 0 o$`fd >OPEN_MAX`$\Rightarrow$ `NULL.`
- `Lógica de lectura, extracción y recorte idéntica, aplicada a tmp[fd].`

6. Helpers compartidos (`get_next_line_utils*.c`)

- `ft_strchr(char *str)`: retorna 1 si encuentra `\n`, 0 en otro caso; tolera `NULL`.
- `ft_strlen(char *str)`: cuenta caracteres hasta `\0`.
- `ft_strjoin(char *s1, char *s2)`: concatena en buffer nuevo; si `s1` es `NULL`, lo trata como cadena vacía; termina en `\0` y libera `s1`.

7. Gestión de memoria y errores

- `read_file` libera su buffer temporal siempre (éxito o error).
- `ft_strjoin` libera su primer argumento: el estado `tmp` siempre apunta a memoria válida y actualizada.
- `get_reminder` libera el `tmp` anterior y entrega el nuevo resto (o `NULL` si no hay más).
- En EOF con buffer vacío, se devuelve `NULL` y el estado queda limpio.
- Errores de `read` (`-1`): se limpian y se propagan como `NULL`. El usuario distingue error de EOF consultando `errno` si lo necesita.

8. Casos límite y cómo defenderlos

- `BUFFER_SIZE = 1`: más iteraciones, mismo resultado; correctness sobre rendimiento.
- **Línea final sin** `\n`: se devuelve esa última línea (sin salto) y luego `NULL`.
- **Líneas muy largas**: múltiples concatenaciones con `ft_strjoin`; limitado por memoria disponible.

- **Descriptor inválido o `BUFFER_SIZE` no positivo**: retorno inmediato `NULL`.
- **Bonus con muchos FD**: uso de `OPEN_MAX` para no salir de rango; si se supera, retorno `NULL`.

9. Preguntas frecuentes

¿Lee más si ya hay `\n` en `tmp`?
No. El bucle de lectura se detiene al detectarlo.

¿Qué pasa si `ft_strjoin` recibe `NULL`?
Se inicializa como cadena vacía y se concatena con el bloque leído.

¿Por qué liberar `tmp` cuando queda vacío?
Para evitar fugas y dejar el estado listo para futuras llamadas.

¿Cómo usar la API correctamente?
`while ((line = get_next_line(fd))) { ... free(line); }`

¿Cómo influye `BUFFER_SIZE`?
Rendimiento (menos llamadas a `read`) vs. memoria temporal; cualquier valor positivo es válido.

10. Mains de práctica (opcional durante la defensa)

- Obligatoria: abrir un archivo (o usar `stdin`) e imprimir cada línea tal cual la devuelve `get_next_line`.
- Bonus: abrir dos archivos y alternar llamadas, etiquetando la salida ("A:", "B:") para demostrar independencia entre descriptores.
- Antes de entregar, elimina o desactiva estos `main` si tu repositorio debe quedar limpio.

11. Comandos útiles

```
# Compilación (GNU11 + warnings estrictos)
gcc -std=gnu11 -Wall -Wextra -Werror -pedantic \
    get_next_line.c get_next_line_utils.c \
    -o gnl_std

# Ejecutar con archivo o por stdin
./gnl_std texto.txt
./gnl_std < texto.txt

# Versión bonus
gcc -std=gnu11 -Wall -Wextra -Werror -pedantic \
    get_next_line_bonus.c get_next_line_utils_bonus.c \
    -o gnl_bonus
./gnl_bonus texto1.txt texto2.txt

# Opcional: fijar BUFFER_SIZE y desactivar mains de práctica si los hubieras añadido
gcc -std=gnu11 -Wall -Wextra -Werror -pedantic \
    -DBUFFER_SIZE=64 \
    get_next_line.c get_next_line_utils.c -o gnl_std_bs64
```

12. Checklist previo a la evaluación

- Explicar qué almacenan los `static` y cuándo se liberan.
- Describir claramente `read_file`, `get_new_line` y `get_reminder`.
- Justificar el array `tmp[OPEN_MAX]` en el bonus y cómo evita mezclas entre `fd`.
- Comentar el impacto de `BUFFER_SIZE` y por qué tu implementación es robusta.
- Recordar que el usuario debe `free()` en cada línea devuelta.

Con este dossier puedes defender con claridad tanto la versión normal como la bonus, argumentar gestión de memoria/errores y demostrar el uso con compilación y ejecución rápidas.