Le system call nel sistema Unix

Antonio Rocchia

April 25, 2022

Contents

1 Le system call di Unix

Il kernel di Linux espone un API di funzioni che un utente o un processo può chiamare per richiedere che un servizio venga svolto dal sistema operativo a livello kernel.

Ogni linguaggio di programmazione fa uso di queste system call in tantissimi modi. Ad esempio per aprire e creare file, leggere da un file, inizializzare un processo e così via.

Nei linguaggi di programmazione moderni le system call sono esposte al programmatore tramite una serie di funzioni(wrapper) che ne semplificano l'uso ed impediscono che vengano commesse violazioni dell'API.

2 Lista delle system call Unix

- Controllo dei processi/thread:
  - Creazione processi
  - Terminazione processi
  - Sospensione processi
  - Attesa terminazione
  - Sostituzione di un processo
- Gestione del file system
- Gestione dei dispositivi
- Comunicazione

3 Wrapper delle system call nel linguaggio C

3.1 fork()

fork - Crea un processo figlio ,Includere fork

```
#include <unistd.h>

int fork();
```

Descrizione: `fork()` genera un processo figlio duplicando il processo che chiama questa funzione. Il nuovo processo è chiamato *figlio*, il processo chiamante è chiamato *padre*

Il processo figlio ed il padre *continuano l'esecuzione in spazi di memoria diversi.* Al momento della fork() *il figlio ottiene una copia dello spazio di memoria del padre* Il padre ed il figlio sono identici eccetto per queste differenze:

- Il figlio ha il suo PID unico e diverso dal padre
- il figlio non eredita allarmi (*alarm()*) dal padre
- Il segnale di terminazione del figlio è sempre *SIGCHLD*
- Il figlio eredita le copie dei descrittori(*file descriptor*) aperti dal padre
- Il figlio eredita una copia dello stream di direttori aperti dal padre (*opendir()*)

Valore di ritorno: In caso di successo, il processo padre riceve il PID del processo figlio appena generato, e il figlio riceve 0. In caso di fallimento, il padre riceve *-1*, non viene generato nessun processo figlio e *errno* viene impostata correttamente.

Esempi: Di seguito vengono riportati alcuni esempi di codice che mostrano gli utilizzi tipici di fork(). Si cerca dove possibile di spiegare al meglio cio che succede durante l'esecuzione

fork_base.c

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    //  Inizializzo e controllo il valore di ritorno di fork()
    int pid = 0;
    pid = fork();
    if (pid == 0)
```

Il processo figlio ottiene una copia dello spazio di memoria del padre, in questo caso la variabile pid inizializzata a 0. Il processo padre ed il processo figlio proseguono l'esecuzione allo stesso momento (in parallelo), ma con una differenza, *fork restituisce il PID del processo figlio generato al padre e restituisce 0 al figlio.* Questo valore viene assegnato alla variabile pid. Ora possiamo scrivere del codice che permetta al processo di capire se è il padre o il figlio.

fork_base.c

```
    if (pid == 0)
    { // Sono il figlio
        // Codice del figlio
        printf("Sono il figlio\n");
        return 0;
    }
    else if (pid > 0)
```

```
    { // Sono il padre
        // Codice del padre
        printf("Sono il padre\n");
    }
    return 0;
}
```

Il processo figlio esegue lo stesso codice del processo padre. Pertanto nella maggior parte dei casi è necessario terminare l'esecuzione del figlio prima di uscire dal blocco *if(pid==0)*

Output fork_base.c

```
Sono il padre
Sono il figlio
```

3.2 getpid(),getppid()

getpid, getppid - restituisce l'identificatore del processo(PID) ,Includere getpid()

```
#include <unistd.h>

int getpid();
int getppid();
```

Descrizione: *getpid()* restituisce al processo che chiama la funzione il sui identificatore di processo *(PID)*.

getppid() restituisce al processo che chiama la funzione l'identificativo di processo *(PID)* del suo processo padre.

Controllo degli errori Questa system call non fallisce mai. Non è necessario alcun controllo.

Valore di ritorno

- getpid(): Restituisce il pid del processo che la chiama
- getppid(): Restituisce il pid del padre del processo che la chiama.

Esempi: Di seguito alcuni esempi di codice che mostrano l'utilizo di getpid() e getppid().

getpid() in azione

```
#include <unistd.h>
#include <stdio.h>

int main(void)
{
    int pid = 0;

    pid = fork();
    if (pid == 0)
    {
        // processo figlio
        printf("La variabile pid = %d\n", pid);
        printf("Il mio pid: %d\n", getpid());
        printf("Il pid del mio processo padre: %d\n",
               getppid());
        return 0;
```

```
    }
    else if (pid < 0)
    {
        perror("Errore durante la fork");
        return -1;
    }
    else
    { // if(pid>0)
        // processo padre
        printf("La variabile pid = %d\n", pid);
        printf("Il mio pid: %d\n", getpid());
        printf("Il pid del mio processo padre: %d\n",
               getppid());
        return 0;
    }
}
```

Output getpid_base.c

```
La variabile pid = 22687
Il mio pid: 22686
Il pid del mio processo padre: 21857
La variabile pid = 0
Il mio pid: 22687
Il pid del mio processo padre: 22686
```

3.3 exit()

exit - Causa la terminazione *volontaria* di un processo.

Un processo può terminare involontariamente, se si tentano azioni illegali o se il processo viene interrotto tramite *segnale* (*kill()*), o volontariamente, se viene eseguita l'ultima istruzione o si esegue la funzione exit() ,

```
#include <stdlib.h>

void exit(int status);
```

exit o return? Se una procedura raggiunge un return, restuisce un valore alla procedura che l'ha invocata, il processo termina solo se il main effettua un return. exit() invece termina il processo in esecuzione, anche se non viene chiamato dal main.

Descrizione La funzione exit() causa la normale terminazione di un processo.

- Il byte meno significativo di *status* è restituito al processo padre.
- Tutte le stream di stdio *(stdin,stdout,stderr)* vengono chiuse.
- La libreria standard di C *stdlib.h* specifica due costanti `(EXIT_SUCCESS, EXIT_FAILURE)` che possono essere usate per specificare l'esito della terminazione.
- Se un processo padre termina prima della terminazione di un figlio. Il processo figlio viene "*adottato*" dal processo init (pid==1), che ne rileva automaticamente lo stato di terminazione.

Controllo degli errori Questa system call non fallisce main. Non è necessario alcun controllo.

Valore di ritorno Questa system call non ha nessun valore di ritorno.

Esempi Todo esempi

3.4 signal()

signal - gestione dei segnali secondo ANSI C ,

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

L'uso di *sighandler_t* è un estensione di GNU, senza la dichiarazione di signal risulta essere un po più difficile da leggere.

,

```
void (* signal(int sig, void (*func))) (int);
```

Descrizione: signal() ridireziona la gestione del segnale *signum* all'handler specificato.

L'handler può essere:

- SIG_IGN, il segnale è ignorato.
- SIG_DFL, la gestione del segnale è impostata su default (Vd. sezione successiva)
- Funzione specificata dal programmatore.

Se la gestione del segnale è impostata su una funzione specificata dal programmatore:

1. Si interrompe l'esecuzione del processo che riceve il segnale.
2. Viene eseguito l'handler
3. Dopo l'esecuzione dell'handler il processo riprende l'esecuzione da dove si è fermato.

I segnali UNIX Di seguito una lista di tutti i segnali con i relativi codici in un sistema UNIX

Lista dei segnali

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5)
    ↪ SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10)
    ↪ SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15)
   ↪ SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20)
   ↪ SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25)
   ↪ SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30)
   ↪ SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37)
   ↪ SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42)
   ↪ SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47)
   ↪ SIGRTMIN+13
```

```
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52)
    ↪ SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57)
    ↪ SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62)
    ↪ SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Impostare un handler personalizzato

```c
#include <signal.h>

void gestore(int);
void gestore2(int);

int main(void) {
    /* ... */
    signal(SIGUSR1, gestore);
    signal(SIGUSR2, gestore);
    signal(SIGCHLD,gestore2);

    signal(SIGCHLD,SIG_DFL); /* Reimposto il valore di default */

    /* ... */
}
```

I segnali SIGKILL e SIGSTOP non possono essere ignorati e non possono essere gestiti esplicitamente dai processi.

Valore di ritorno Il valore di ritorno in caso di errore è SIG_ERR e errno viene correttamente impostato.

Persistenza dell'handler Alla fine dell'esecuzione di un handler definito dall'utente il sistema si occupa di reinstallarlo automaticamente.

Se durante l'esecuizone di un handler, arriva un secondo segnale uguale a quello che ha causato la sua esecuzione, il segnale viene accodato e gestito una volta terminato il primo handler.

Segnali che arrivano mentre c'è un segnale uguale accodato vengono accorpati, come se fosse un solo segnale. Quindi un solo segnale verrà consegnato al processo.

Gestire un segnale durante una system call Cosa succede se arriva un segnale mentre c'è una system call in esecuzione.

Le system call possono essere classificate come slow o non slow. *Le system call slow possono essere interrotte da un segnale*, in questo caso la system call fallisce.

3.5 alarm()

alarm - imposta un timer per l'invio di un segnale

,

```c
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Descrizione alarm() invia un segnale SIGALRM al thread che chiama la funzione dopo che sono passati un certo numero di secondi. *Se i secondi sono 0, tutti gli alarm() del thread vengono cancellati.*

Gli alarm() vengono preservati anche dopo un exec, ma non sono ereditati dai processi creati con fork().

Su alcuni sistemi sleep() potrebbe essere implementata tramite alarm(), *pertanto NON mischiare chiamate a sleep() e ad alarm().*

Valore di ritorno: alarm() restituisce il numero di secondi rimanenti prima una precedente chiamata ad alarm() inviasse SIGALRM, oppure 0 se nessun timer era stato precedentemente impostato.

3.6 sleep()

sleep - ferma l'esecuzione per un certo numero di secondi.

,

```
#include <unistd.h>

unsigned int sleep(insigned int seconds);
```

Descrizione: sleep() "addormenta" il thread che chiama questa system call per un numero di secondi reali pari all'intero specificato.

Su linux, sleep() è implementata tramite nanosleep(). Su altri sistemi invece potrebbe essere implementata tramite alarm(), *pertanto NON mischiare chiamate a sleep() e ad alarm().*

Valore di ritorno La funzione restituisce 0 se il numero di secondi passati è effettivamente quello richiesto. Il numero di secondi rimasto se la chiamata viene interrotta da un segnale inviato al thread che chiama sleep.

3.7 kill() e raise()

kill - Invia un segnale ad un processo. raise - Invia un segnale al processo che la invoca. ,

```
#include <signal.h>

int kill(int pid, int sig);
int raise(int sig);
```

Descrizione kill() può inviare un segnale ad processo o un process group in base al pid specificato:

- Se pid>0, sig è inviato al processo con quel pid.
- Se pid=0, sig è inviato ad ogni processo del process group corrente.
- Se pid=-1, sig è inviato ad ogni processo eccetto per il primo. L'ordine di invio del segnale va dal processo che ha il pid più grande nella tabella dei processi, a quello avente il pid più piccolo.
- Se pid<-1, il segnale è inviato ad ogni processo che ha il process group uguale a valore assoluto del pid.

Se sig=0 il segnale non viene inviato, tuttavia viene fatta ugualmente la ricerca degli errori. Ciò è utile per verificare se un pid è associato a qualche processo.

Il processo che invia il segnale e i processi che lo ricevono devono avere lo stesso user ID effettivo, a meno che il segnale non sia inviato dal super user.

raise() invece invia il segnale sig al processo che la invoca.

Tavola dei segnali UNIX

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5)
    ↪ SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10)
    ↪ SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15)
   ↪ SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20)
   ↪ SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25)
   ↪ SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30)
   ↪ SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37)
   ↪ SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42)
   ↪ SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47)
   ↪ SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52)
   ↪ SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57)
   ↪ SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62)
   ↪ SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Valore di ritorno: Il caso di successo kill() e raise() restituiscono 0. In caso di fallimento restituiscono un valore diverso da 0 e la variabile errno è impostata correttamente.

3.8 open(), creat()

open, openat, creat - aprono e (nel caso di creat) possibilmente creano un file.

,

```c
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);
```

Descrizione: La system call *open()* apre il file specificato da *pathname*. Se il file specificato non esiste e viene specificata la flag *O_ CREAT*, il file viene anche creato.

open() *restituisce un descrittore file*, un intero piccolo e non negativo che rappresenta un elemento nella process's table dei descrittori file aperti. Il descrittore sarà poi usato da altre system call quali read(), write(), lseek(), etc....

Il descrittore file che viene restituito corrisponderà al più piccolo valore di file descriptor disponibile.

All'inizio dell'esecuzione di un processo vengono automaticamente aperti stdin(fd=0), stdout(fd=1), stderr(fd=2). Per approfondire.

Una chiamata ad open *crea un nuovo descrittore di file aperto*, un elemento in una tabella dell'intero sistema che tiene traccia di tutti i file aperti. Il descrittore di file aperto tiene traccia dell'offset(vd. lseek()) e del flag di stato del file(spiegati giù).

Aprire file con open

```c
#include <fcntl.h>

int main()
{
    int fd1,fd2,fd3;

    fd1=open("path/to/file/", O_RDONLY);
    if(fd1<0) {
        perror("open fallita");
    }
    /* ... */
    fd2=open("path/to/file2/", O_WRONLY);
    if(fd2<0) {
        perror("open in scrittura fallita");

        fd2=open("path/to/file2/", O_WRONLY|O_CREAT, 0777);
    }
    /* Omogeneo: apertura di un dispositivo */
    fd3=open("/dev/dispositivo", O_WRONLY);
    /* ... */
}
```

L'argomento `open(..., int flag, ...)` DEVE INCLUDERE UNO dei seguenti *access mode:*

- O_RDONLY
- O_WRONLY
- O_RDWR

Questi access mode specificano se il file sarà aperto in read-only, write-only o read/write.

In aggiunta si possono opzionalmente concatenare zero o più "file creation flag" e/o "file status flag". Si possono concatenare più flag tramite l'operatore bitwise '|'. `open(..., O_RDONLY|O_CREAT|O_LARGEFILE , ...);`

Le flag di creazione file ("file creation flag") sono:

- O_CLOEXEC
- O_CREAT
- O_DIRECTORY
- O_EXCL
- O_NOCTTY
- O_NOFOLLOW
- O_TMPFILE
- O_TRUNC

La differenza tra i "file creation flag" e le "file status flag" è che i flag di creazione coinvolgono la semantica dell'operazione di apertura del file (es. Apri il file in modalità write-only e se non esiste crealo `open("path/to/file/", O_WRONLY|O_CREAT)`), mentre le flag di stato coinvolgono la semantica delle successive operazioni di I/O (es. Apri il file in modalità write-only e appendi al file ogni volta che scrivo `open("path/to/file/", O_WRONLY|O_APPEND)`).

Una chiamata a creat() è equivalente a chiamare open() con i flag `O_CREAT|O_WRONLY|O_TRUNC`

Il parametro `open(...,...,mode_t mode);` fa riferimento ai permessi del file. Nel caso di open() questo parametro è opzionale, nel caso di creat() bisogna specificare i permessi del file.

Questi permessi vengono specificati in formato ottale (es. 0777, 0766, etc...). In C abbiamo una libreria che definisce delle macro che possiamo comporre come i flag tramite l'operatore bitwise '|'.

#include <sys/stat.h>

```
#define S_IRWXU 0000700    /* RWX mask for owner */
#define S_IRUSR 0000400    /* R for owner */
#define S_IWUSR 0000200    /* W for owner */
#define S_IXUSR 0000100    /* X for owner */

#define S_IRWXG 0000070    /* RWX mask for group */
#define S_IRGRP 0000040    /* R for group */
#define S_IWGRP 0000020    /* W for group */
#define S_IXGRP 0000010    /* X for group */

#define S_IRWXO 0000007    /* RWX mask for other */
#define S_IROTH 0000004    /* R for other */
#define S_IWOTH 0000002    /* W for other */
#define S_IXOTH 0000001    /* X for other */

#define S_ISUID 0004000    /* set user id on execution */
#define S_ISGID 0002000    /* set group id on execution */
#define S_ISVTX 0001000    /* save swapped text even after use */
```

Uso di stat.h

```
#include <sys/stat.h>
#include <stdio.h>

int main(void) {
    printf("Mode 1: %4o\t", S_IRWXU);
    printf("Mode 2: %4o\t", S_IROTH|S_IWOTH);
    printf("Mode 3: %4o\t", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
}
```

output

```
Mode 1:  700    Mode 2:    6    Mode 3:  776
```

3.9 pipe()

pipe - Crea una pipe. ,name=pipe()

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

Descrizione: pipe() crea una pipe, un *canale di comunicazione tra processi unidirezionale* per la trasmissione dei dati.

L'array *int pipefd[2]* è usato per specificare due *descrittori file (file descriptors)* che rappresentano gli estremi della pipe:

- pipefd[0] rappresenta il lato di lettura della pipe.
- pipefd[1] rappresenta il lato di scrittura della pipe.

Quando tutti i descrittori di file che si riferiscono al lato scrittura della pipe vengono chiusi, viene scritto *EOF*(end of file) sulla pipe. Il lato lettura potrà leggerlo tramite *read* e restitirà 0.

Chiusura lato scrittura

```c
#include <unistd.h>

int main(void) {
    int pipefd[2];

    pipe(pipefd);

    close(pipefd[1]); /* tutti i file descriptor per il lato scrittura
       ↪ sono chiusi*/
                      /* Scrivo EOF sulla pipe */

    char c;
    read(pipefd[0],&c,1); /* restituisce 0 */
}
```

Quando vengono chiusi tutti i descrittori di file che si riferiscono al lato lettura, ogni write verso la pipe causera un *SIGPIPE* verso il processo che prova a leggere. Questa situazione potrà quindi essere gestita come ogni altro segnale.

Chiusura lato lettura

```c
#include <unistd.h>

int main(void) {
    int pipefd[2];

    pipe(pipefd);

    close(pipefd[0]); /* tutti i file descriptor per la lettura sono
       ↪ chiusi */

    char myString[5] = "Ciao\0";
    write(pipefd[1],myString,5); /* Causa SIGPIPE */
}
```

La pipe non ha una dimensione. Il canale di comunicazione della pipe è una *byte stream*, quindi non esiste il concetto di limite del file. Il kernel si occupa di gestire la pipe.

- Se la pipe è piena, write sarà sospensiva.
- Se la pipe è vuota o non ha la quantità di byte richiesti da una read, read sarà sospensiva.

La capacità effettiva della pipe dipende dall'implementazione. In ogni caso è preferibile che il programma non sia progettato facendo affidamento ad una particolare capacità: Un applicazione dovrebbe essere progettata in modo che il processo che legge consumi i dati appena sono disponibili, cosi che il processo di scrittura non rimanga bloccato.

Un applicazione che usa pipe() e fork() dovrebbe usare close() per chiudere file descriptor duplicati di cui non ha bisogno. Se il processo figlio deve solo scrivere dati, chiudera il lato lettura della pipe. close(pipefd[0]);

Chiudere i file descriptor inutilizzati

```c
#include <unistd.h>
#include <stdlib.h>

int main(void) {
    int pipefd[2];

    pipe(pipefd);
```

```
    int pid = fork();
    if(pid == 0) {
        close(pipefd[0]); /* Chiudo il lato lettura */

        /* Scrivo nella pipe */

        exit(EXIT_SUCCESS);
    } else if(pid < 0) {
        /* fork error recovery */
        exit(EXIT_FAILURE);
    } /* if pid > 0: Sono il padre*/
        close(pipefd[1]); /* Chiudo il lato scrittura */

        /* Leggo dalla pipe */

        exit(EXIT_SUCCESS);
}
```

Valore di ritorno: In caso di successo, pipe() restituisce 0. In caso di errore restituisce -1, pipefd[2] rimane invariato e viene impostato un valore per *errno*

Esempi: Nel seguente esempio il programma crea una pipe, dopo effettua una fork per creare un processo figlio. Il figlio ottiene un duplicato dei descrittori dei file che fanno riferimento alla pipe creata dal padre. Dopo la fork ogni processo chiude i descrittori di cui non hanno più bisogno. Il padre scrive poi una stringa passata al programma per argomento nella pipe, il figlio la legge e la scrive su *stdout*

Tipico uso di pipe()

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int pipefd[2];
    int cpid;
    char buf;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd) == -1)
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1)
    {
        perror("fork");
```

```
        exit(EXIT_FAILURE);
    }

    if (cpid == 0)
    {                       /* Child reads from pipe */
        close(pipefd[1]); /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);
    }
    else
    {                       /* Parent writes argv[1] to pipe */
        close(pipefd[0]); /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]); /* Reader will see EOF */
        wait(NULL);         /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```