

Un semplice esempio

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

int main(int argc, char *argv[]) {
    execlp("ls", "ls", 0);
    printf("La chiamata execlp() ha fallito\n");
}
```

In questa **execlp** chiediamo di eseguire LS, passando come ARGV[0] proprio LS.

Però **execlp**, quando prende il controllo, dato che è P, va a guardare all'interno della variabile d'ambiente PATH, e tra i vari valori registrati all'interno di questa variabile d'ambiente identifica qual'è eventualmente la posizione corretta di ls, all'interno del FileSystem.

Noi stiamo cercando di eseguire LS e quale versione di ls partirà, dipenderà tutto dalla variabile d'ambiente PATH.

Se questa variabile d'ambiente contiene il ./, e noi nel codice stiamo lanciando un programma che compilandolo abbiamo voluto chiamarlo LS, stiamo eventualmente rilanciando ls, questo è un classico problema di sicurezza che noi abbiamo quando gli attaccanti registravano nomi dei file eseguibili con nomi omologhi a comandi di shell e magari invece di eseguire il comando di shell vero, si eseguiva il programma che voleva eseguire l'attaccante.

Ora siamo in grado di capire come funziona una shell. Questa è un'applicazione, ls che è il comando che noi eseguiamo all'interno di un sistema operativo unix, è un programma che sta da qualche parte.

Come fa questa shell a lanciare ls? Se noi lanciamo ls lo facciamo con una exec, e se facciamo una exec sappiamo che il programma viene sostituito con un altro, e allora perché la shell rimane in ascolto attiva?

Le shell lavorano secondo un incrocio di system call o comandi che noi abbiamo già spiegato, che sono la system call FORK, e la system call EXECVE.

Quando abbiamo una shell attiva, quello che succede realmente è questo: la shell chiama un input dal kernel, verrà magari messa in attesa se l'input non arriva e, quando l'input da tastiera arriva il kernel lo consegna all'address space della shell e poi riconsegna il controllo alla shell.

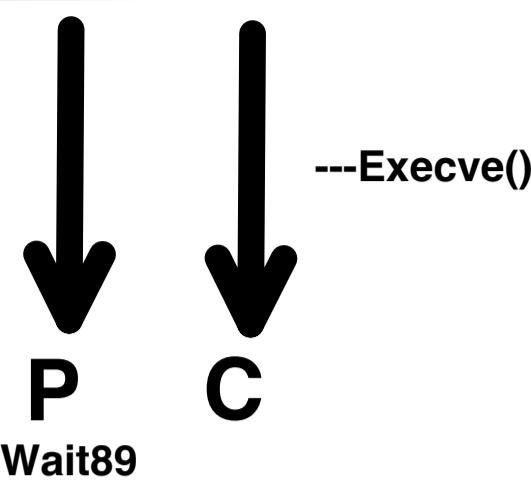
Una volta digitato "LS" la shell si fa carico di attivare il programma, ma NON VUOLE TERMINARE, perché da la possibilità all'utente di fare altre cose.

Più avanti nella sua esecuzione richiamerà il kernel ancora per una fork e quindi diventiamo due, ossia

Parent e Child, e siamo due cloni della shell e il Child esegue la exec: passa il controllo di nuovo al kernel per lanciare il programma LS.

Il meccanismo all'interno di una shell è basato su un ciclo in cui noi eseguiamo una FORK(), diventiamo due, e uno dei due esegue EXECVE(), QUALSIASI SIA IL PROGRAMMA CHE SI PASSA IN QUESTO PROCESSO TRAMITE LA EXECVE(), IL PARENT RIMANE IN WAIT(). Lui aspetta che l'altro processo finisca la sua esecuzione.

FORK



OCESSES-AND-THREADS/UNIX> ./a.out &

La shell che prende questa linea in input, capisce che il parent della shell ossia il processo stesso con cui ora stiamo parlando per fornire queste informazioni in input, magari non deve eseguire la WAIT.

Una semplice shell di comandi per sistemi UNIX

```

#include <stdio.h>
#include <stdlib.h>
void main() {
    char comando[256]; pid_t pid; int status;
    while(1) {
        printf("Digitare un comando: ");
        scanf("%s", comando);
        pid = fork();
        if (pid == -1) {
            printf("Errore nella fork\n");
            exit(EXIT_FAILURE);
        }
        if (pid == 0) {
            execlp(comando, comando, 0);
            exit(EXIT_FAILURE);
        }
        else wait(&status);
    }
}
  
```

A meno che qualcuno non intervenga dall'esterno questa applicazione non termina mai, nel while acquisiamo il comando e lo scarichiamo all'interno dell'array di 256 byte. Questo è il nome di un comando che deve partire, lanciamo una fork e verifichiamo se non ci sono errori, i soliti controlli di routine, nel momento in cui siamo in due e siamo il Child, eseguiamo la execvp del comando, passiamo come argomento del main quella stringa e basta, altrimenti se siamo il parent attendiamo che il Child abbia completato la sua esecuzione.

MINI-SHELL ESEMPIO

```

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

#define PARAMS // this allows running commands with parameters, but still limited to foreground execution

#define MAX_ARGS 128
#define AUDIT

int gets(char*);

char* s = " ";

int main(int argc, char** argv) {
    char command_line[4096];
    char* p;
    char* args[MAX_ARGS];
    int pid, status;
    int i;

    printf("Welcome to mini-shell\n");

    while(1) {
        printf("Type a command line: ");

#ifndef PARAMS
        scanf("%s", command_line);
#else
        gets(command_line);
        p = (char*)strtok(command_line, s);
        i = 0;
        args[i] = p;
        i++;
#endif
    }
}
  
```

Se compilando io non definisco "PARAMS" sto generando, nel momento in cui compilo, una versione esegibile di questo programma in cui quello che posso fare è lanciare programmi senza passare parametri.

Ecco qui, nell'esempio sopra nella EXEC, stiamo lanciando un programma sostituendolo ovviamente al CHILD, non passiamo argomenti se non il nome del programma che è stato attivato.

Nel codice sopra non abbiamo la possibilità di inserire la stringa: "ls -la", è già un'espressione molto complessa, però in QUESTA applicazione abbiamo la possibilità di effettuare attività più complesse.

```

    argv[i] = p;
    while(p){
        #ifdef AUDIT
            printf("%s ", p);
        #endif
        fflush(stdout);
        p = (char*)strtok(NULL,s);
        args[++i] = p;
    }
    args[++i] = NULL;

    printf("\n");
#endif
    pid = fork();
    if ( pid == -1 ) {
        printf("Unable to spawn new process\n");
        exit(EXIT_FAILURE);
    }
    if ( pid == 0 ){
#ifndef PARAMS
        execvp(command_line,command_line,0);
#else
        execv(args[0],args);
#endif
        printf("Unable to run the typed command\n");
    } else wait(&status);
}

```

Se non è definito PARAMS, abbiamo solo la linea della scanf che viene compilata, quindi all'interno del while(1) noi andiamo a prendere ad ogni iterazione andiamo a prendere una nuova stringa che rappresenta un nuovo comando che vogliamo eseguire.

E poi ovviamente abbiamo la parte della fork, controlliamo se non ci sono errori nella generazione del processo Child, abbiamo tolto la definizione di params e quindi andiamo ad eseguire in execvp la command_line che abbiamo letto nella scanf precedente, passando soltanto argv[0] al main del programma che sta partendo.

L'else non viene considerato, perché params non è definito.

Se la execvp va a buon fine, questo programma non è più attivo.

OCESSES-AND-THREADS/UNIX> ./a.out
Welcome to mini-shell
Type a command line: █

Esecuzione di questo codice lungo sopra.

Welcome to mini-shell
Type a command line: ls
a.out fork-example.c mini-shell.c thread-exit.c
cascade-fork.c fork-ppid-example.c mini-shell-V1.c threads-interference.c
cascade-fork-V1.c fork-wait-example.c multi-thread-search threads-performance.c
environment.c iterative-fork.c strtok-example-usage.c threads-sort-strings.c
Type a command line: █

Type a command line: ps
PID TTY TIME CMD
3518 pts/0 00:00:00 bash
5926 pts/0 00:00:00 a.out
5935 pts/0 00:00:00 ps
Type a command line: █

Type a command line: ^C
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io,
OCESSES-AND-THREADS/UNIX> █

COME FACCIAMO A FARE COSE Più COMPLESSE CON QUESTA MINI SHELL?

Se noi andiamo a definire PARAMS, andiamo a cercare di poter gestire una linea di comando che non è più vero che è una semplice stringa acquisita in command_line, ma è una cosa più complessa, con una gets(command_line) acquisiamo un'intera linea, e su questa linea ci possono essere scritte più informazioni che possono rappresentare sicuramente il nome del comando che vogliamo lanciare, e altre informazioni che rappresentano i parametri di questo comando che vogliamo lanciare.

Ora per capire gli sviluppi andiamo a vedere un esempio nuovo, che riguarda la funzione "STRTOK".

Noi abbiamo un array di caratteri, un'area di memoria, dove eventualmente chiamando qualche API, andiamo in questa area di memoria a scaricare qualcosa di complesso, quindi c'è una parte poi un blank, poi ancora un'altra parte, etc..! Quindi abbiamo un'intera linea qui dentro, prima di \n.



Di questa linea, che di fatto è un'unica stringa, vogliamo identificare delle porzioni, suddivisa appunto dal blank.

Ad esempio io voglio identificare una stringa che rappresenta il comando: ls
una stringa che rappresenta il parametro: la.
"ls -la".

Come facciamo a fare questa cosa qua?

L'operazione classica per fare questo tipo di attività, si chiama "Tokenizzazione".

Vogliamo modificare il contenuto dell'array precedente, inserendo al posto del blank, degli "\0".



Se riusciamo a fare questo abbiamo tokenizzato la stringa originale in due stringhe diverse.

STRTOK()

```

6 #define MAX_TOKENS 128
7
8 int gets(char* );
9
10 char line[4096];
11
12 char* s = " "; //blank is used as the tokenizer character
13
14 #ifdef SEGVFAULT_TEST
15 char * pointer = "ciao a tutti";
16#endif
17
18 int main(int argc, char** argv) {
19     char **token_vector;
20     char* p;
21     char* tokens[MAX_TOKENS];
22     int i;
23
24 #ifdef SEGVFAULT_TEST
25     p = (char*)strtok(pointer,s); //you should never try to do this
26#endif
27
28     gets(line);
29     p = (char*)strtok(line,s); //Tokenize and get the pointer to the first token
30
31     i = 0;
32     tokens[i] = p;
33
34     while(p){ //get the pointers to the other tokens
35         p = (char*)strtok(NULL,s);
36         tokens[++i] = p;
37
38     }
39     tokens[++i] = NULL;
40
41     token_vector = tokens;
42
43     while(*token_vector){ //audit the tokens on the standard output
44         printf("%s\n",*token_vector);
45         token_vector++;
46
47     }
48 }
49

```

Noi sicuramente abbiamo l'idea di andarcia a prendere un'intera linea con GETS().

Andiamo a scaricare questa linea in un'area di 4096 byte.

Ora questa linea la facciamo TOKENIZZARE/SUDDIVIDERE in stringhe, dividendole in base a quello che vogliamo, passando ovviamente il puntatore all'inizio dell'area che deve essere tokenizzata, ed ovviamente indichiamo quali sono gli elementi che vanno sostituiti con il sterminatore di stringhe, che è il secondo parametro, che è un puntatore a carattere, ossia una stringa.

La stringa s, contiene i caratteri che, nella stringa registrata e scaricata in line, devono essere eliminati: in particolare modo devono essere utilizzati per la "Tokenizzazione".

Nella stringa S, abbiamo inserito solo il blank.

Stiamo suddividendo tutta la linea che abbiamo acquisito utilizzando gets(), utilizzando il blank come separatore delle informazioni.

Strtok una volta che ha finito di tokenizzare questa linea, ci può restituire gli indirizzi delle stringhe suddivise all'interno dello schema di tokenizzazione?



Questa Tokenizzazione ha dato luogo ad A,B e C stringhe diverse, ciascuna con un proprio indirizzo iniziale.

Io voglio sapere dove sono queste stringhe all'interno di queste aree di memoria che vengono tokenizzate.

"La funzione strtok() viene utilizzata per isolare i token sequenziali in una stringa con terminazione null, str. Questi token sono separati nella stringa da almeno uno dei caratteri in sep. La prima volta che viene chiamato strtok(), dovrebbe essere specificato str; le chiamate successive, volendo ottenere ulteriori token dalla stessa stringa, dovrebbero invece passare un puntatore nullo. La stringa di separazione, sep, deve essere fornita ogni volta e può cambiare tra le chiamate."

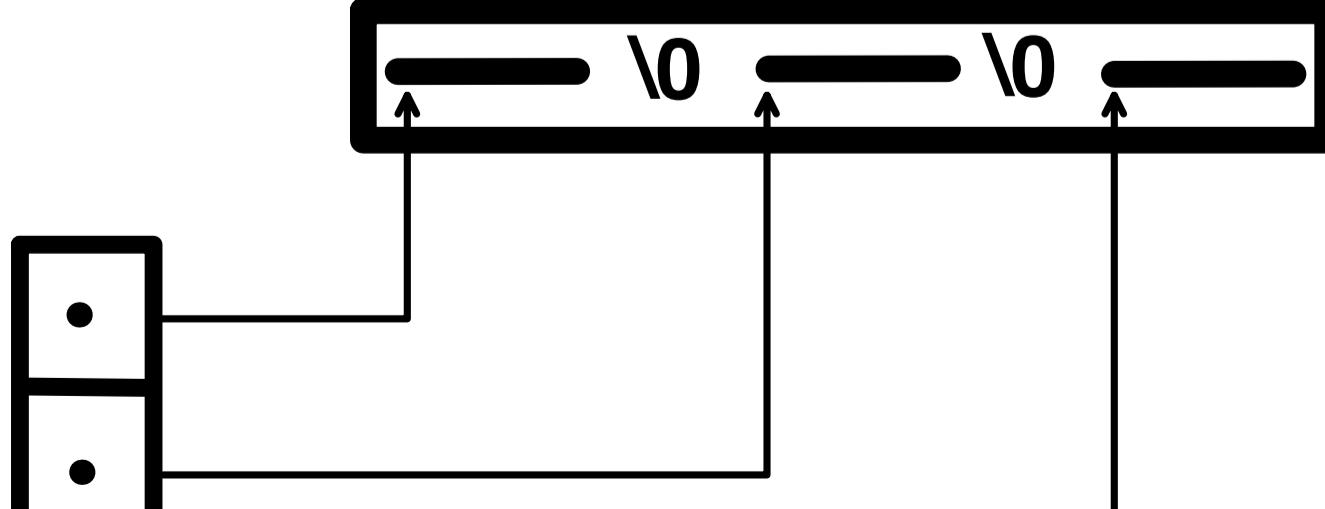
Tranquillo. È sempre STRTOK che fa questo lavoro.

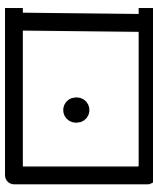
La prima volta che chiamiamo strtok e noi passiamo l'insieme di informazioni da tokenizzare, la funzione ci restituisce il puntatore al primo token che è stato generato. Registrato in P.

Ma possiamo farci restituire anche gli altri, chiamando successivamente strtok però questa volta passando dei parametri diversi, passiamo la stessa regola di tokenizzazione (ossia il secondo parametro) ma preceduto dal primo parametro NULL. (Riga 36).

Possiamo sfruttare tutto questo all'interno di un software che è usato per lanciare programmi. Abbiamo un array di PUNTATORI A CARATTERE, dove ciascun puntatore punta all'inizio di ciascuna stringa tokenizzata.

Strtok è una funzione della libreria standard cosiddetta: STATE FULL. Quando noi la chiamiamo con un certo input la funzione fa un lavoro, quando la chiamiamo con un altro input la funzione fa un altro lavoro.



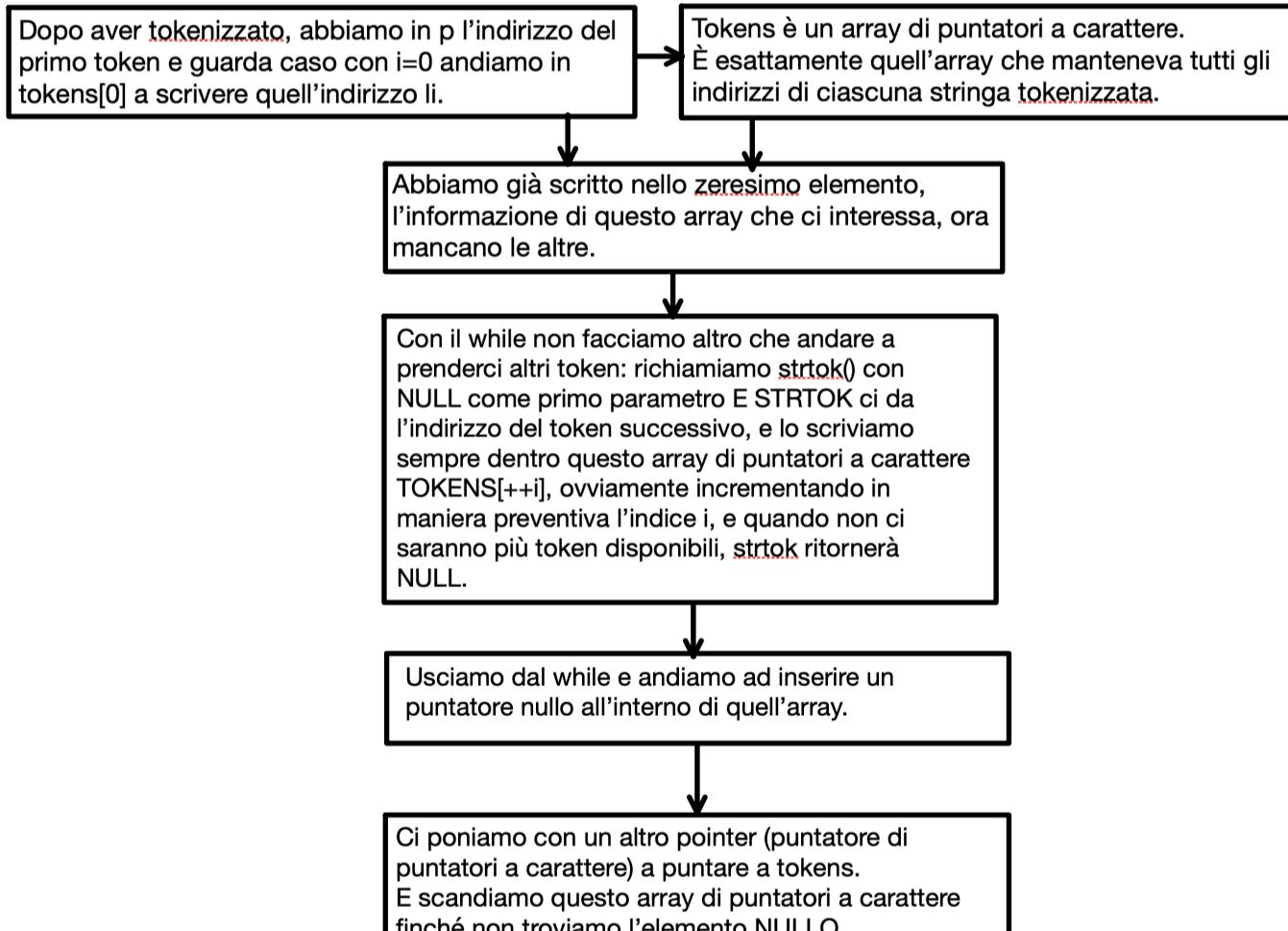


Se noi abbiamo letto una stringa in input e A = "LS", B = "LA", abbiamo ciò che volevamo prima, e abbiamo l'array che ci punta ad ls ed la.

Possiamo utilizzare quell'array in una `execve` per far partire "ls", passando come argomenti del main esattamente queste due stringhe.

In questo modo riusciamo a creare una shell più complessa che ci permette di eseguire anche comandi con parametri.

Riga 24, ci permette di compilare una linea che è un test su `segfault`. Per ora lasciamo la compilazione con il `seg fault` non attivo. Questo lo vedremo dopo. Quindi non attiviamo il simbolo `segfault` in compilazione.



```
0CESSES-AND-THREADS/UNIX> gcc strtok-example-usage.c
/tmp/ccBVEj0.o: In function `main':
strtok-example-usage.c:(.text+0x1e): warning: the `gets' function is dangerous and should not be used.
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PR
0CESSES-AND-THREADS/UNIX> ./a.out
```

```
aaaa bbbb cccc dddd
aaaa
bbbbb
cccc
ddddd
```

Abbiamo tokenizzato

Se per caso fosse definito `SEGFAULT_TEST` su questo programma quando compiliamo, in realtà abbiamo anche quel "pointer" all'interno dell'address space, che punta a "ciao a tutti", andremmo a tokenizzare un'altra cosa, ossia andiamo a tokenizzare la stringa puntata da quel puntatore! Tokenizzandola mantenendo la stessa regola che avevamo prima, ossia mantenendo sempre il blank come simbolo da cambiare.

"Ciao a tutti" può essere suddivisa in tre token: "ciao" "a" "tutti".
Però se io cerco di fare questa cosa, ottengo un SEGMENTATION FAULT.

```
0CESSES-AND-THREADS/UNIX> gcc strtok-example-usage.c -DSEGFAULT_TEST
0CESSES-AND-THREADS/UNIX> ./a.out
```

```
0CESSES-AND-THREADS/UNIX> ./a.out
;elfmewlk
Segmentation fault (core dumped)
```

Sto chiamando `strtok()` passando il pointer che punta a "ciao a tutti" e questa stringa qui, non essendo una variabile locale, viene scritta all'interno dell'address space quando l'applicazione va in esecuzione, nella zona READ ONLY. Io sto andando in quella zona per scrivere.

Detto ciò, torniamo al discorso della mini-shell

Ora andiamo ad attivare la #DEFINE PARAMS, perché vogliamo giustamente i parametri della shell.

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

#define PARAMS this allows running commands with parameters, but still limited to foreground execution
#define MAX_ARGS 128

//#define AUDIT

int gets(char*); // This is dangerous and should not be used.

char* s = " ";

int main(int argc, char** argv) {
    char command_line[4096];
    char* p;
    char* args[MAX_ARGS];
    int pid, status;
    int i;

    printf("Welcome to mini-shell\n");

    while(1) {
        printf("Type a command line: ");

        #ifndef PARAMS
            scanf("%s", command_line);
        #else
            gets(command_line); ←
            p = (char*)strtok(command_line, s);
            i = 0;
            args[i] = p;

            while(p){
                #ifdef AUDIT
                    printf("%s ", p);
                #endif
                fflush(stdout);
                p = (char*)strtok(NULL,s);
                args[++i] = p;
            }
            args[++i] = NULL;
        #endif
        printf("\n");
        pid = fork();
        if ( pid == -1 ) {
            printf("Unable to spawn new process\n");
            exit(EXIT_FAILURE);
        }
        if ( pid == 0 ){
            #ifndef PARAMS
                execlp(command_line,command_line,0);
            #else
                execvp(args[0],args);
            #endif
            printf("Unable to run the typed command\n");
        } else wait(&status);
    }
}
```

In modo tale che, quando compiliamo, questa volta non includiamo `scanf()`, ma ora includiamo un blocco di codice che fa esattamente quello di cui parlavamo prima. Prendo un'intera linea con `gets()`, la tokenizzo, comincio a scrivere il primo puntatore al primo dei token all'interno di questo `args`, poi con un `while` mi vado a prendere gli altri token, me li vado a scrivere dentro l'array di puntatori `args`.
Poi eseguo la `fork()` e se sono il Child non eseguo il blocco di codice di `execvp`, ma vado su `execvp`, in cui dico che il programma da lanciare è `args[0]`, quindi lo zero del token che ho generato leggendo quella linea però a quel programma devo passare tutti i token che ho generato, quindi magari anche gli argomenti, ossia `args`.
Siamo in uno scenario interessante.

```
0CESSES-AND-THREADS/UNIX> gcc mini-shell.c
/tmp/ccANUZ3v.o: In function `main':
mini-shell.c:(.text+0x3c): warning: the `gets' function is dangerous and should not be used.
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PR
0CESSES-AND-THREADS/UNIX> ./a.out
Welcome to mini-shell
Type a command line: 
```

```
Type a command line: ls -la
total 84
drwxr-xr-x 3 francesco users 4096 Mar 18 12:42 .
drwxr-xr-x 4 francesco users 4096 Apr  1  2020 ..
-rw-r--r-- 1 francesco users 12920 Mar 18 12:42 a.out
-rw-r--r-- 1 francesco users 269 Apr  8  2018 cascade-fork.c
-rw-r--r-- 1 francesco users 313 Apr  8  2018 cascade-fork-V1.c
-rw-r--r-- 1 francesco users 557 Mar 18 12:00 environment.c
-rw-r--r-- 1 francesco users 254 Mar 21  2019 fork-example.c
-rw-r--r-- 1 francesco users 322 Mar 30  2020 fork-ppid-example.c
-rw-r--r-- 1 francesco users 390 Mar 26  2020 fork-wait-example.c
-rw-r--r-- 1 francesco users 249 Mar 17  2019 iterative-fork.c
-rw-r--r-- 1 francesco users 1216 Mar 18 12:41 mini-shell.c
-rw-r--r-- 1 francesco users 1361 Mar 18 10:43 mini-shell-V1.c
drwxr-xr-x 2 francesco users 4096 May  3  2020 multi-thread-search
-rw-r--r-- 1 francesco users 845 Mar 29  2020 strtok-example-usage.c
-rw-r--r-- 1 francesco users 292 Mar 17  2019 thread-exit.c
-rw-r--r-- 1 francesco users 581 Mar 17  2019 threads-interference.c
-rw-r--r-- 1 francesco users 644 Apr  3  2019 threads-performance.c
-rw-r--r-- 1 francesco users 1371 Apr  4  2019 threads-sort-strings.c
Type a command line: 
```

