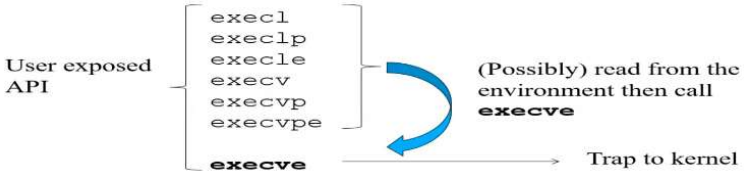
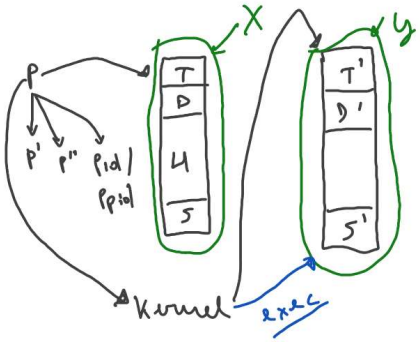
Processi e thread 4 Exec+ Env

martedì 14 ottobre 2025 14:38

Vediamo ora come definire il contenuto di un address space di un'applicazione (quindi di definire il programma ospitato all'interno di questo contenuto) :



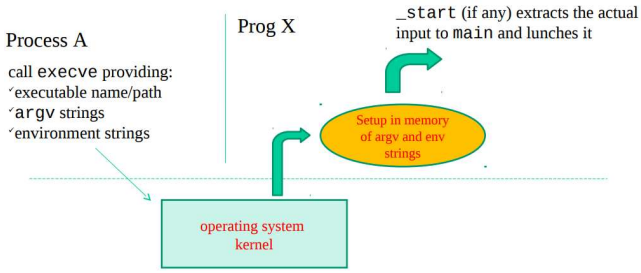
Quindi l'attivazione di un programma avviene attraverso l'ultima istruzione, ma non il clonaggio (cambiando il suo address space). Quindi in dettaglio si ha che :



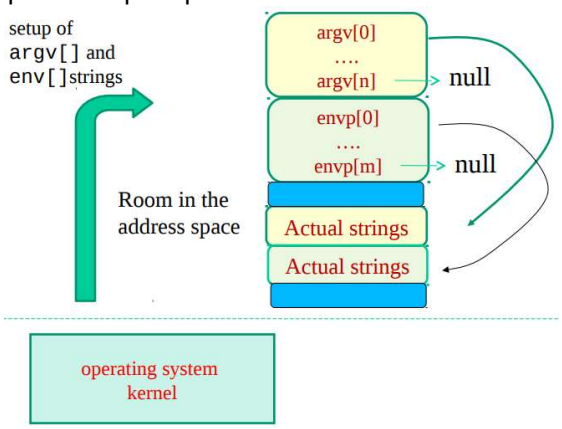
Quindi se chiamo questa syscall si ha che il vecchio processo non esiste più. Vediamo degli esempi:

```
#include <unist.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

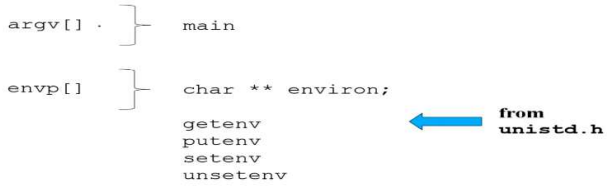
Dove il primo parametro è un puntatore a carattere e rappresenta il nome del programma che vogliamo far partire all'interno dell'address space (sostituzione address space). Mentre per gli altri parametri passiamo delle stringhe . Vediamo in dettaglio come funziona :



Ed in dettaglio , soprattutto per i parametri :



Quindi le API sono le seguenti :



Vediamole in dettaglio :

int execl(char *file_name, [char *arg0, ... ,char *argN,] 0)	
Descrizione	invoca l'esecuzione di un programma
Parametri	1) *file_name: nome del programma 2) [*arg0, ... ,*argN,]: parametri della funzione main()
Restituzione	-1 in caso di fallimento

È una exec basica , con la quale cerchiamo di lanciare un programma il cui nome è il primo parametro (se all'interno del direttorio corrente). Se variante **execlp** va a fare la ricerca nella variabile PATH. Gli altri parametri sono stringhe che vengono passate al main dell'applicazione che va a sostituire l'applicazione corrente. **Nota questa syscall invoca execve**. Vediamo altro esempio :

int execv(char *file_name, char **argv)	
Descrizione	invoca l'esecuzione di un programma
Parametri	1) *file_name: nome del programma 2) **argv: parametri della funzione main()
Restituzione	-1 in caso di fallimento

Dove il primo parametro è e una stringa e rappresenta il nome del file che venga attivato , il quale viene cercato nel direttorio corrente ; esiste la variante **execvp** che cerca nel direttorio PATH. In questo caso si ha uno schema fail-retry : si parte dalla prima variabile e poi se non trovata vado alla prossima. Per quanto riguarda il secondo parametro passiamo un array di puntatore a caratteri. Vediamo un esempio :

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

extern char** environ;


void main(int argc, char **argv){
        char ** addr=environ;
        char * const * p = NULL;


        printf("process %d - environ head pointer is at address:
%lu\n",getpid(),(unsigned long)environ);
        fflush(stdout);

        while(*addr){
             printf("%s\n",*(addr));
             fflush(stdout);
             addr++;
        }
       //argv[0] è il nome del programma


//      execve(argv[0],argv,NULL);
//      execve(argv[0],argv,environ);
        execve(argv[0],p,NULL);
  //    execve("/usr/bin/ls",NULL,environ);

        argv[0] = "/usr/bin/ls";
        argv[1] = NULL;
        execve("/usr/bin/ls",argv,environ);
        printf("execve failed\n");
        fflush(stdout);
}
```

il cui output mostra il contenuto di environment; mentre ultime righe di codice eseguono il programma che fa vedere il contenuto del direttorio attuale. Vediamo ora ulteriore variante :

int execve(char *file_name, char **argv, char **envp)	
Descrizione	invoca l'esecuzione di un programma
Parametri	1) *file_name: nome del programma 2) **argv: parametri della funzione main() 3) **envp: variabili d'ambiente
Restituzione	-1 in caso di fallimento

Che nell'esempio precedente , a seconda della chiamata che eseguo cambia output. Nota : **nel caso di shell che invoca execve si ha una combinazione di fork e execve :** quindi ho due shell della quale la seconda esegue la execve .Vediamo ora come cambiare le variabili d'ambiente :

char *getenv(char *name)	
Descrizione	richiede il valore di una variabile d'ambiente
Parametri	*name, nome della variabile d'ambiente
Restituzione	NULL oppure la stringa che definisce il valore della variabile

Permette di avere tornato un puntatore alla stringa uguale a name contenuta in enviroment. Analogamente ad ottenere il puntatore, lo possiamo settare :

int putenv(char *string)	
Descrizione	setta il valore di una variabile d'ambiente
Parametri	*string, nome della variabile d'ambiente + valore da assegnare (nella forma "nome=valore")
Restituzione	0 in caso di success – valore diverso da zero in caso di fallimento

Come per esempio : vogliamo aggiungere delle directory (multi valore) :

PATH=/user/local/bin:/bin/:/home/quaglia/bin

Per quanto riguarda invece il settaggio/eliminazione dell'ambiente si ha che :

int setenv(char *name, char *value, int overwrite)	
Descrizione	crea una variabile d'ambiente e setta il suo valore
Parametri	1) *name: nome della variabile d'ambiente 2) *value: valore da assegnare 3) overwrite: flag di sovrascrittura in caso la variabile esista
Restituzione	0 in caso di successo, -1 in caso di fallimento

int unsetenv(char *name)	
Descrizione	elimina una variabile d'ambiente
Parametri	*name: nome della variabile d'ambiente
Restituzione	0 in caso di successo, -1 in caso di fallimento

Dove overwrite sta a significare che comunque la sovrascrivi , **anche se esiste.** Tornando al discorso delle shell, vediamone come implementarne una :

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

//#define PARAMS // this allows running commands with paramenters, 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;

          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
```

Questa versione rappresenta una versione base della shell : è possibile solo utilizzare il primo parametro come comando. Se la define PARAMS viene decommentata ha lo stesso comportamento della v1

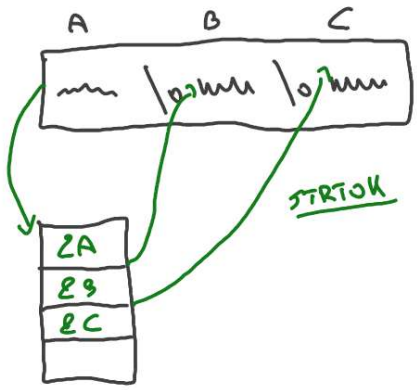
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include<sys/wait.h>

#define PARAMS // this allows running commands with paramenters, but still limited to foreground execution
#define MAX_ARGS 128
#define AUDIT
#define LINE_SIZE (4096)

char* s = " \n";//the \n is because we are reading input data with fgets()
int main(int argc, char** argv) {
      char command_line[LINE_SIZE];
      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
          fgets(command_line,LINE_SIZE,stdin);
          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;
          printf("\n");
#endif
          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
                printf("try to run %s\n",args[0]);
                fflush(stdout);
                execvp(args[0],args);
#endif
                printf("Unable to run the typed command\n");
          }
          else wait(&status);
      }
}
```

Mentre questa seconda versione permette di prendere una intera linea come comando da eseguire.

Ma come facciamo da una riga a prendere sia il comando che i parametri?? Vado a spezzettare (tokenizzare)

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


#define MAX_TOKENS 128

int gets(char*);

char line[4096];

char* s = " ";//blank is used as the tokenizer character

#ifdef SEGFAULT_TEST
char * pointer = "ciao a tutti";
#endif

int main(int argc, char** argv) {
        char **token_vector;
        char* p;
        char* tokens[MAX_TOKENS];
        int i;

#ifdef SEGFAULT_TEST
        p = (char*)strtok(pointer,s);//you should never try to do this
#endif

        gets(line);
        p = (char*)strtok(line,s);//tokenize and get the pointer to the first token


        i = 0;
        tokens[i] = p;

        while(p){//get the pointers to the other tokens
                p = (char*)strtok(NULL,s);
                tokens[++i] = p;

        }
        tokens[++i] = NULL;

        token_vector = tokens;

        while(*token_vector){//audit the tokens on the standard output
                printf("%s\n",*token_vector);
                token_vector++;

        }
}
```

Attenzione se complito con la direttiva SEGFAULT_TEST: vado a scrivere quella stringa in una zona read-only.-> segmentation fault. Quindi l'output è il seguente : se ho in input aaaaa bbbbb cccc dddd , si ha in output aaaa \n bbbb \n cccc \n dddd \n. Ma effettivamente come funziona ?



Torniamo **all'ambiente della shell** : la shell modifica le sue stesse informazioni . in dettaglio:

1. Inserimento/aggiunta
    a. **EXPORT nome=valore**
2. Rimuovere
    a. **UNSET nome**
3. Ottenere nome
    a. **$Nome**

Attenzione al seguente passo : se si parla di comandi interni di parla di comandi che invocano software già presente nella stessa shell (ne fork ne exec), quindi si modificano le informazioni ambientali della shell stessa , altrimenti esterni .