

Esercitazione sulle Pipe

Davide Lamanna

lamanna@dis.uniroma1.it

MIDLAB

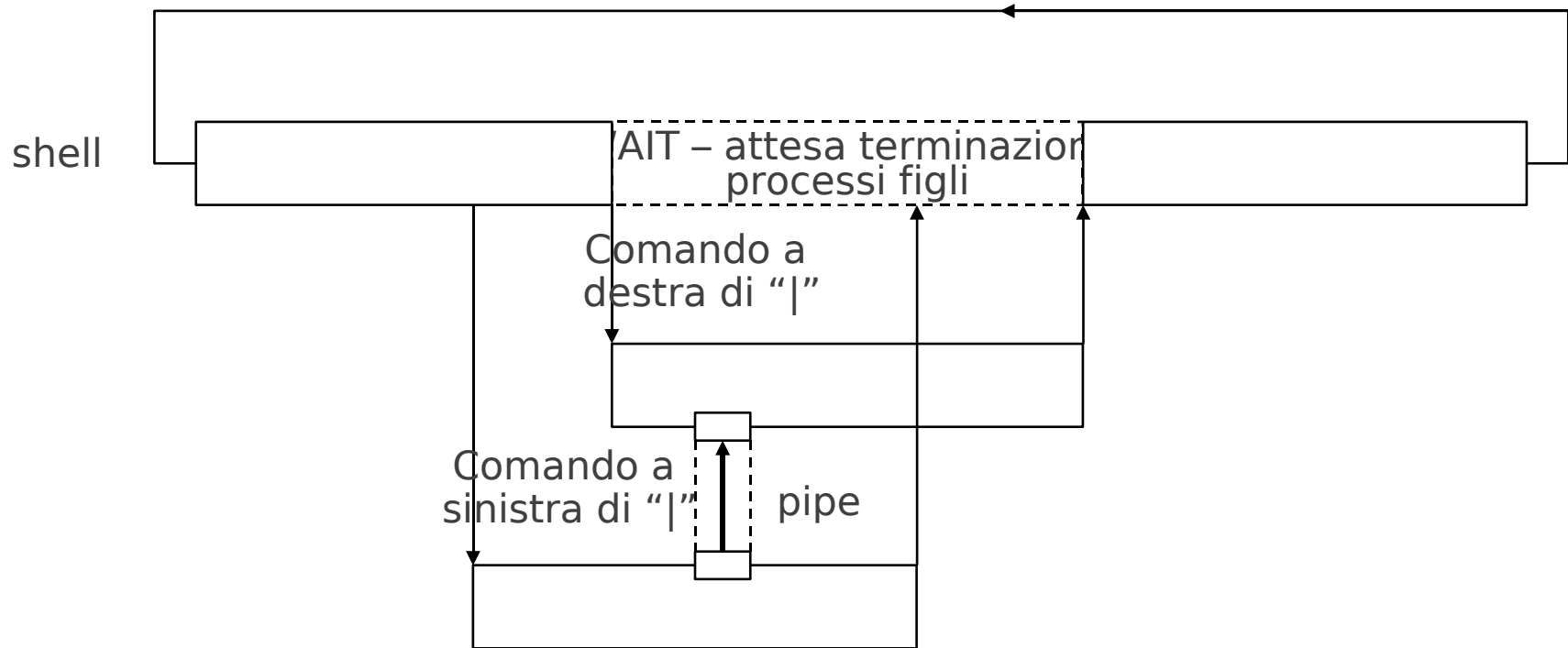
Contenuti

- ❑ Specifica del problema
- ❑ Struttura generale del programma
- ❑ Implementazione Linux
- ❑ Implementazione Windows

Specifica del problema

- Realizzazione di una semplice shell che sia in grado di accettare sulla stessa linea più comandi separati dal simbolo “|”
- Semantica del simbolo “|”
 - L’output generato dal comando a sinistra di “|” non deve andare sullo standard output della shell (schermo), ma deve essere usato come input del comando a destra di “|”
 - Questo utilizzo del simbolo “|” è molto diffuso sia in shell Unix/Linux che Windows

Struttura del programma

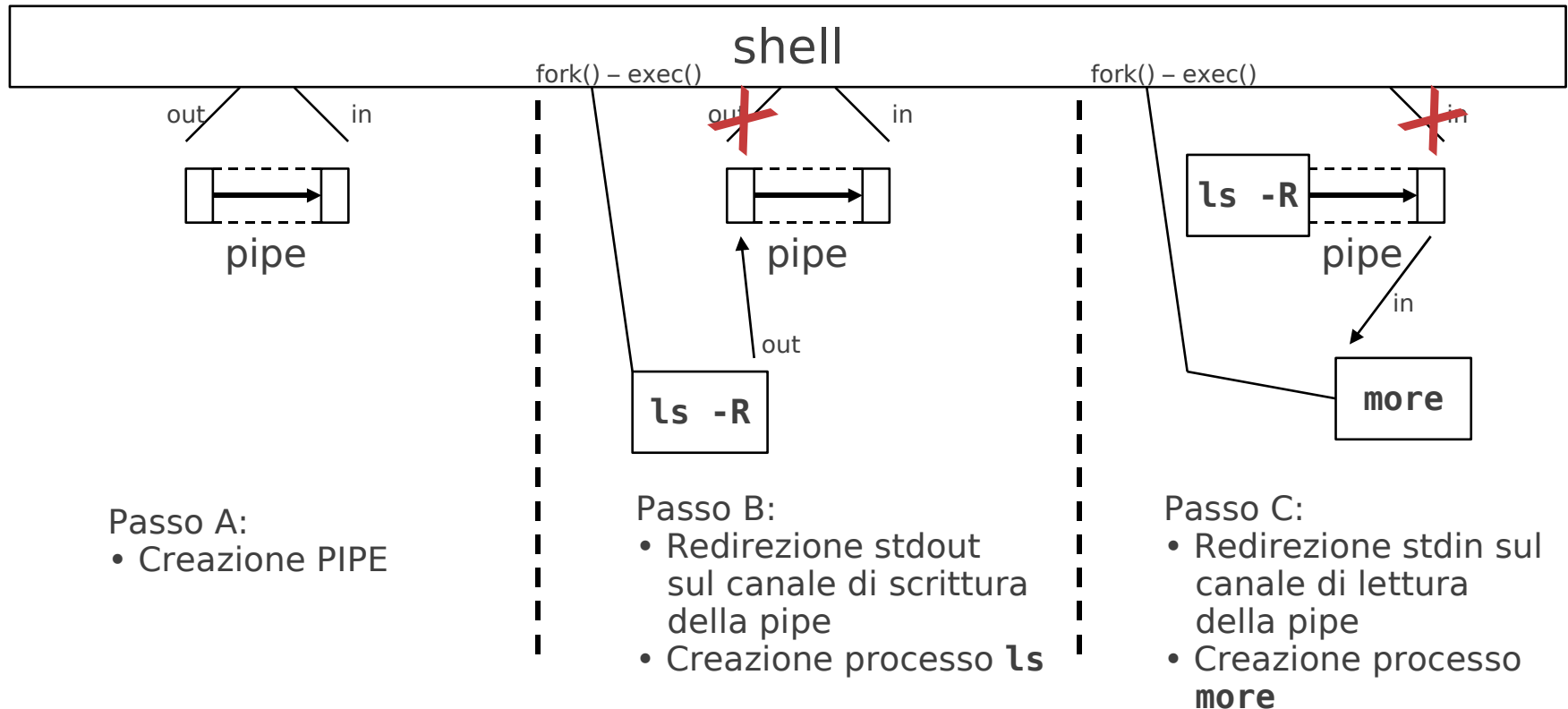


- ❑ UNIX: generazione di nuovi processi con accoppiata `fork()` – `execvp()`
- ❑ WINDOWS: generazione di nuovi processi con `CreateProcess()`

Generazione dei processi connessi da PIPE

□ Comando:

▪ `ls -R | more`



Implementazione shell - UNIX (1)

(Struttura del main)

```
/* inclusioni e dichiarazioni iniziali */
int main () {
    /* dichiarazioni di variabili */
    while (1) {

        /* lettura linea di comando */

        do {
            /* generazione processi figli */
        } while (next_command != NULL);

        /* attesa terminazione dei processi lanciati */
        for (i=0; i<pending_processes; i++) {
            wait(&status);
        }
    }
    printf("\nLeaving Shell\n\n");
    return(0);
}
```

Implementazione shell - UNIX (2)

(Inclusioni e dichiarazioni iniziali)

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

```
/* massima lunghezza del comando accettato dalla shell */
```

```
#define COMMAND_LENGTH 1024
```

```
/* conta gli argomenti di un comando da eseguire */
```

```
int count_args(char *start_command, int *pipe_present, char **next_command);
```

```
/* restituisce il vettore da passare alla execvp() */
```

```
char **build_arg_vector(char *command, int arg_num, int pipe_present);
```

Implementazione shell - UNIX (3)

(Dichiarazione di variabili)

```
char *command_pointer; /* puntatore al comando a sinistra di "|" */
char *next_command; /* puntatore al comando a destra di "|" */
char line[COMMAND_LENGTH+1]; /* serie di comandi inseriti da tastiera */
char **arg; /* vettore di parametri da passare alla execvp */
int arg_num; /* numero di elementi di arg */
int *old_pipe_descriptors; /* puntatore alla pipe */
int *new_pipe_descriptors; /* puntatore corrente alla pipe */
int pipe_present=0; /* indica se è presente "|" o no */
int pipe_pending=0; /* indica se è stata aperta una pipe o no */
int pending_processes; /* indica il numero di processi aperti */
int i; /* contatore */
int status; /* status dei processi figli*/
```

Implementazione shell - UNIX (4)

(Struttura del main)

```
/* inclusioni e dichiarazioni iniziali */
int main () {
    /* dichiarazioni di variabili */
    while (1) {
        /* lettura linea di comando */

        do {
            /* generazione processi figli */
        } while (next_command != NULL);

        /* attesa terminazione dei processi lanciati */
        for (i=0; i<pending_processes; i++) {
            wait(&status);
        }
    }
    printf("\nLeaving Shell\n\n");
    return(0);
}
```

Implementazione shell - UNIX (5)

(Lettura linea di comando)

```
printf("\nExample Shell>");
fflush(stdout);

fgets(line, COMMAND_LENGTH, stdin);
line[COMMAND_LENGTH] = '\0';

if (strlen(line) == COMMAND_LENGTH) {
    printf("\nCommand too long!\n");
    continue;
}

pending_processes = 0;
command_pointer = line;

/* esce se il comando è exit */
if (strcmp(command_pointer, "exit\n") == 0) break;
```

Implementazione shell - UNIX (6)

(Struttura del main)

```
/* inclusioni e dichiarazioni iniziali */
int main () {
    /* dichiarazioni di variabili */
    while (1) {

        /* lettura linea di comando */
        do {
            /* generazione processi figli */
        } while (next_command != NULL);

        /* attesa terminazione dei processi lanciati */
        for (i=0; i<pending_processes; i++) {
            wait(&status);
        }
    }
    printf("\nLeaving Shell\n\n");
    return(0);
}
```

Implementazione shell - UNIX (7)

(Generazione processi figli – parsing del comando)

```
arg_num = count_args(
    command_pointer, &pipe_present, &next_command);

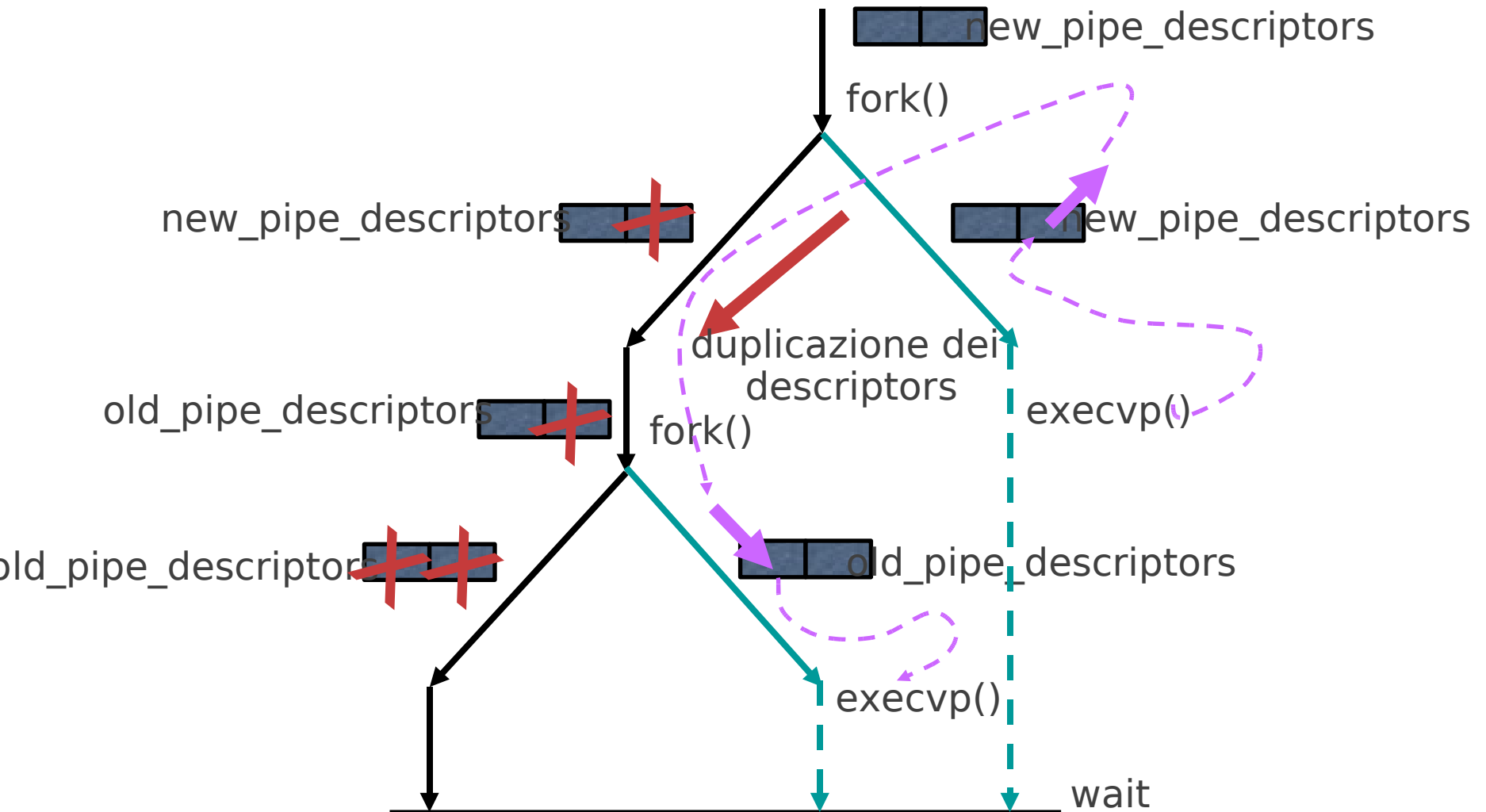
if (arg_num < 0) break;

arg = build_arg_vector(
    command_pointer, arg_num, pipe_present);

if (pipe_present) {
    new_pipe_descriptors = malloc(sizeof(int)*2);
    if (pipe (new_pipe_descriptors) < 0) {
        printf("Can't open a pipe for error %d!\n", errno);
        fflush(stdout);
        exit(-1);
    }
}
```

Implementazione shell - UNIX (8)

(Generazione processi figli)



Implementazione shell - UNIX (9)

(Generazione processi figli – generazione processo (1))

```
if ((i=fork()) == 0) {
    if (pipe_present) { /* se sto eseguendo il primo comando */
        close(1); /* chiudo standard output */
        /* redireziona il canale di out della pipe come stdout */
        dup(new_pipe_descriptors[1]);
        close(new_pipe_descriptors[1]);
    }
    if (pipe_pending) { /* se sto eseguendo il secondo comando */
        close(0); /* chiudo standard input */
        /* redireziona il canale di in della pipe come stdin */
        dup(old_pipe_descriptors[0]);
        close(old_pipe_descriptors[0]);
    }

    execvp(arg[0],arg);
    printf("Can't execute file %s\n",arg[0]);fflush(stdout);
    exit(-1);
}
else .....
```

Implementazione shell - UNIX (10)

(Generazione processi figli – generazione processo (2))

```
if (i>0) {
    pending_processes++;
    if (pipe_present) { /* se sto eseguendo il primo comando */
        close(new_pipe_descriptors[1]);
        old_pipe_descriptors = new_pipe_descriptors;
        pipe_pending = 1;
    }
    if (pipe_pending) { /* se sto eseguendo il secondo comando */
        close(old_pipe_descriptors[0]);
        free(old_pipe_descriptors);
        pipe_pending = 0;
    }
}
else {
    printf("Can't spawn process for error %d\n", errno);
    fflush(stdout);
}
command_pointer = next_command;
```

Implementazione shell - UNIX (11)

(Funzione `count_args` – logica di funzionamento)

- Conta gli argomenti del comando da eseguire (compreso il comando stesso)
 - Separo i 2 comandi
 - Divido la stringa in token (lo spazio) fino a quando non trovo il carattere “|” e nel frattempo conto
 - Restituisco
 - Il numero di argomenti del primo comando (quello alla sinistra di “|”)
 - Se è presente un secondo comando (a destra di “|”) restituisco il puntatore al comando, e setto il flag **pipe_present** per la creazione della pipe

□ Esempio:

- `ls -R | more`
 - Per `ls -R` restituisco 2 come numero di argomenti, **pipe_present = 1** e il puntatore a “more”
 - Per `more` restituisco 1 come numero di argomenti, **pipe_present = 0** e il puntatore a NULL (che segnalerà la fine dell'esecuzione)

Implementazione shell - UNIX (12)

(Funzione count_args – codice (1))

```
int count_args(char *start_command, int *pipe_present, char
**next_command) {
    char copied_string[COMMAND_LENGTH + 1];
    char *temp, *temp_next;
    int args = 0, i;

    strcpy(copied_string, start_command);

    if (((temp = strtok(copied_string, " ")) == NULL) ||
        (strcmp(temp, "\0") == 0)) return(-1);
    args++;

    while (((temp = strtok(NULL, " ")) != NULL) &&
           (strcmp(temp, "|") != 0 )) {
        if (strcmp(temp, "\0") == 0 ) break;
        args++;
    }
}
```

Implementazione shell - UNIX (13)

(Funzione count_args – codice (2))

```
if (temp == NULL) {
    *pipe_present = 0;
    *next_command = NULL;
}
else {
    if ((temp = strtok(NULL, " \n")) != NULL) {
        *pipe_present = 1;
        for (i=0; ; i++)
            if (copied_string[i] == '|') break;
        *next_command = start_command + i + 2;
    }
    else {
        *pipe_present = 0;
        *next_command = NULL;
    }
}
return(args);
}
```

Implementazione shell - UNIX (14)

(Funzione `build_arg_vector` – codice)

```
char **build_arg_vector(char *command, int arg_num, int
    pipe_present) {

    char **arg_vector;
    char *temp;
    int i;

    arg_vector = malloc((arg_num+1) * sizeof(char *));

    for(i=0; i < arg_num; i++) {
        if (i==0) temp = strtok(command, " \n");
        else temp = strtok(NULL, " \n");
        arg_vector[i] = temp;
    }

    arg_vector[i] = NULL;
    return(arg_vector);
}
```

Implementazione shell - WINDOWS (1)

(Struttura del main)

```
/* inclusioni e dichiarazioni iniziali */
int main () {
    /* dichiarazioni di variabili */
    while (1) {

        /* lettura linea di comando */

        do {
            /* generazione processi figli */
        } while (next_command != NULL);

        /* attesa terminazione dei processi lanciati */
        WaitForMultipleObjects( pending_processes,
            pending_processes_handles, TRUE, INFINITE);
    }
    printf("\nLeaving Shell\n\n");
    return(0);
}
```

Implementazione shell - WINDOWS

(2)

(Inclusioni e dichiarazioni iniziali)

```
#include <windows.h>
```

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

```
#include <string.h>
```

```
#include <malloc.h>
```

```
/* massima lunghezza del comando accettato dalla shell */
```

```
#define COMMAND_LENGTH 1024
```

```
/* conta gli argomenti di un comando da eseguire */
```

```
int count_args(char *start_command, int *pipe_present, char **next_command);
```

```
/* restituisce il vettore di argomenti di un comando */
```

```
char **build_arg_vector(char *command, int arg_num, int pipe_present);
```

```
/* costruisce la stringa da passare alla CreateProcess */
```

```
void build_command_line(char **arg, char *line, int num);
```

Implementazione shell - WINDOWS

(3)

(Dichiarazione di variabili)

```
char *command_pointer, /* puntatore al comando a sinistra di "|" */
char *next_command; /* puntatore al comando a destra di "|" */
char line[COMMAND_LENGTH+1]; /* serie di comandi inseriti da tastiera */
char **arg; /* vettore di parametri da passare alla execvp */
int arg_num; /* numero di elementi di arg */
int pipe_present=0; /* indica se è presente "|" o no */
int pipe_pending=0; /* indica se è stata aperta una pipe o no */
int pending_processes; /* indica il numero di processi aperti */
int i; /* contatore */

HANDLE *old_pipe_handles; /* vettore di handle della pipe */
HANDLE *new_pipe_handles; /* vettore di handle della pipe corrente */
SECURITY_ATTRIBUTES security;
HANDLE temp_readHandle; /* handle temporaneo al canale di in della pipe */
HANDLE temp_writeHandle; /* handle temporaneo al canale di out della pipe */
HANDLE /* vettore di handle che memorizza
    pending_processes_handles[ /* gli handle dei processi creati */
        MAXIMUM_WAIT_OBJECTS];

BOOL newprocess; /* handle alla pipe */
STARTUPINFO si; /* handle alla pipe corrente */
PROCESS_INFORMATION pi; /* handle alla pipe corrente */
```

Implementazione shell - WINDOWS (4)

(Struttura del main)

```
/* inclusioni e dichiarazioni iniziali */
int main () {
    /* dichiarazioni di variabili */
    while (1) {

        /* lettura linea di comando */

        do {
            /* generazione processi figli */
        } while (next_command != NULL);

        /* attesa terminazione dei processi lanciati */
        WaitForMultipleObjects( pending_processes,
            pending_processes_handles, TRUE, INFINITE); }
    }
    printf("\nLeaving Shell\n\n");
    return(0);
}
```

Implementazione shell - WINDOWS

(5)

(Lettura linea di comando)

```
printf("\nExample Shell>");  
fflush(stdout);
```

```
fgets(line, COMMAND_LENGTH, stdin);  
line[COMMAND_LENGTH] = '\0';
```

```
if (strlen(line) == COMMAND_LENGTH) {  
    printf("\nCommand too long!\n");  
    continue;  
}
```

```
pending_processes = 0;  
command_pointer = line;
```

```
/* esce se il comando è exit */
```

```
if (strcmp(command_pointer, "exit\n") == 0) break;
```

Implementazione shell - WINDOWS (6)

(Struttura del main)

```
/* inclusioni e dichiarazioni iniziali */
```

```
int main () {
```

```
    /* dichiarazioni di variabili */
```

```
    while (1) {
```

```
        /* lettura linea di comando */
```

```
        do {
```

```
            /* generazione processi figli */
```

```
        } while (next_command != NULL);
```

```
        /* attesa terminazione dei processi lanciati */
```

```
        WaitForMultipleObjects( pending_processes,  
                                pending_processes_handles, TRUE, INFINITE);
```

```
    }
```

```
    printf("\nLeaving Shell\n\n");
```

```
    return(0);
```

```
}
```

Implementazione shell - WINDOWS (7)

(Generazione processi figli – parsing del comando)

```
arg_num = count_args(command_pointer, &pipe_present, &next_command);
```

```
if (arg_num < 0) break;
```

```
arg = build_arg_vector(command_pointer, arg_num, pipe_present);  
build_command_line(arg, command_line, arg_num);
```

```
if (pipe_present) {  
    new_pipe_handles = malloc(sizeof(HANDLE)*2);  
    security.nLength = sizeof(security);  
    security.lpSecurityDescriptor = NULL;  
    security.bInheritHandle = TRUE;  
  
    if (CreatePipe (  
        &new_pipe_handles[0], &new_pipe_handles[1], &security, 0) < 0) {  
  
        printf("Can't open a pipe for error %d!\n", GetLastError());  
        fflush(stdout);  
        ExitProcess(-1);  
    }  
}
```

Implementazione shell - WINDOWS

(8)

(Generazione processi figli – generazione processo (1))

```
if (pipe_present) { /* se sto eseguendo il primo comando */
    temp_writeHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    SetStdHandle(STD_OUTPUT_HANDLE, new_pipe_handles[1]);
}
if (pipe_pending) { /* se sto eseguendo il secondo comando */
    temp_readHandle = GetStdHandle(STD_INPUT_HANDLE);
    SetStdHandle(STD_INPUT_HANDLE, old_pipe_handles[0]);
}
```

```
memset(&si, 0, sizeof(si));
memset(&pi, 0, sizeof(pi));
si.cb = sizeof(si);
```

```
newprocess = CreateProcess(
    NULL,
    command_line,
    NULL, NULL,
    TRUE,
    NORMAL_PRIORITY_CLASS,
    NULL, NULL,
    &si, &pi);
```

Implementazione shell - WINDOWS

(9)

(Generazione processi figli – generazione processo (2))

```
if (pipe_present) { /* se sto eseguendo il primo comando */
    SetStdHandle(STD_OUTPUT_HANDLE, temp_writeHandle);
    CloseHandle(new_pipe_handles[1]);
    old_pipe_handles = new_pipe_handles;
    pipe_pending = 1;
}
if (pipe_pending) { /* se sto eseguendo il secondo comando */
    SetStdHandle(STD_INPUT_HANDLE, temp_readHandle);
    CloseHandle(old_pipe_handles[0]);
    free(old_pipe_handles);
    pipe_pending = 0;
}

if (newprocess == 0) {
    printf("Can't spawn executable %s for command \"%s\" (error %d)!\n",
        arg[0], command_line, GetLastError());
    break;
}
pending_processes_handles[pending_processes] = pi.hProcess;
pending_processes++;
command_pointer = next_command;
```

Implementazione shell - WINDOWS (10)

(Funzione count_args – codice (1))

```
int count_args(char *start_command, int *pipe_present, char
**next_command) {
    char copied_string[COMMAND_LENGTH + 1];
    char *temp, *temp_next;
    int args = 0, i;

    strcpy(copied_string, start_command);

    if (((temp = strtok(copied_string, " ")) == NULL) ||
        (strcmp(temp, "\0") == 0)) return(-1);
    args++;

    while (((temp = strtok(NULL, " ")) != NULL) &&
           (strcmp(temp, "|") != 0 )) {
        if (strcmp(temp, "\0") == 0 ) break;
        args++;
    }
}
```

Implementazione shell - WINDOWS (11)

(Funzione count_args – codice (2))

```
if (temp == NULL) {
    *pipe_present = 0;
    *next_command = NULL;
}
else {
    if ((temp = strtok(NULL, " \n")) != NULL) {
        *pipe_present = 1;
        for (i=0; ; i++)
            if (copied_string[i] == '|') break;
        *next_command = start_command + i + 2;
    }
    else {
        *pipe_present = 0;
        *next_command = NULL;
    }
}
return(args);
}
```

Implementazione shell - WINDOWS (12)

(Funzione `build_arg_vector` – codice)

```
char **build_arg_vector(char *command, int arg_num, int
    pipe_present) {

    char **arg_vector;
    char *temp;
    int i;

    arg_vector = malloc((arg_num+1) * sizeof(char *));

    for(i=0; i < arg_num; i++) {
        if (i==0) temp = strtok(command, " \n");
        else temp = strtok(NULL, " \n");
        arg_vector[i] = temp;
    }

    arg_vector[i] = NULL;
    return(arg_vector);
}
```

Implementazione shell - WINDOWS (13)

(Funzione build_command_line – codice)

```
void build_command_line(char **arg, char *line, int num) {
    int i;

    for(i=0; i<num; i++) {
        sprintf(line, "%s", arg[i]);

        line += strlen(arg[i]);
        *line = ' ';
        line += 1;
    }
    *(line - 1) = '\\0';
    return;
}
```

Appendice - strtok()

```
#include <string.h>
```

```
char *strtok(char *s, const char *delim);
```

- ❑ Estrae dei token da una stringa
- ❑ Divide la stringa in una serie di token delimitati dal carattere *delim*
 - La prima chiamata deve avere come parametro una stringa *s*
 - Ritorna il primo token se *delim* è presente nella stringa *s*
 - NULL altrimenti
 - Le successive chiamate possono avere NULL come primo parametro
 - Ad ogni chiamata viene salvato il puntatore al prossimo token
 - Il separatore *delim* può essere diverso ad ogni chiamata
- ❑ Fa effetto collaterale sulla stringa *s*
- ❑ ESEMPIO: *s* = "Ciao come stai? Bene grazie. E tu?"
 - `strtok(s, " ")` restituisce "ciao"
 - `strtok(NULL, " ")` restituisce "come"
 - `strtok(NULL, " ")` restituisce "stai?"
 - `strtok(NULL, " ")` restituisce "Bene"
 - `strtok(NULL, ".")` restituisce "grazie"
 - `strtok(s, ".")` restituisce "Ciao"

Appendice - `execvp()`

```
int execvp( const char *file, char *const  
            argv[]);
```

- ❑ Sostituisce l'immagine del processo corrente con una nuova
 - Permette così di "lanciare" processi diversi da quello in esecuzione
- ❑ Il primo parametro rappresenta il nome del file associato al processo da eseguire
- ❑ Il secondo parametro rappresenta la lista di parametri da passare al processo
 - Il primo elemento dell'array è il comando da eseguire
 - L'ultimo elemento dell'array deve essere **NULL**

Appendice - CreateProcess()

```
BOOL CreateProcess(  
    LPCTSTR lpApplicationName,  
    LPTSTR lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL blInheritHandles,  
    DWORD dwCreationFlags,  
    LPVOID lpEnvironment,  
    LPCTSTR lpCurrentDirectory,  
    LPSTARTUPINFO lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation );
```

- Crea un nuovo processo; parametri principali:
 - *lpApplicationName*: il nome del processo da lanciare (può essere NULL)
 - *lpCommandLine* : specifica la linea di comando da eseguire (stringa separata da spazi)
 - *blInheritHandles* : se TRUE ogni handle ereditabile viene ereditato dal nuovo processo
 - *lpProcessInformation* : memorizza le informazioni riguardanti il nuovo processo
 - l'handle del processo: campo *hProcess* di *lpProcessInformation*

Note

- ❑ Sul sito sono disponibili i codici sorgenti visti
- ❑ Sintassi dei comandi
 - Sotto UNIX/Linux
 - `ls -R | more`: funziona
 - `ls -R|more`: non funziona
 - Sotto WINDOWS:
 - `cmd /c dir /s | cmd /c more`: funziona
 - `dir | more`: non funziona
 - `cmd /c dir /s|more`: funziona
 - `Cmd /c dir /s | more`: non funziona