

# Esercitazione: Utilizzo delle Pipe

- Specifica del problema
- Struttura generale del programma
- Esempio per Unix
- Esempio per Windows

# Esercizio

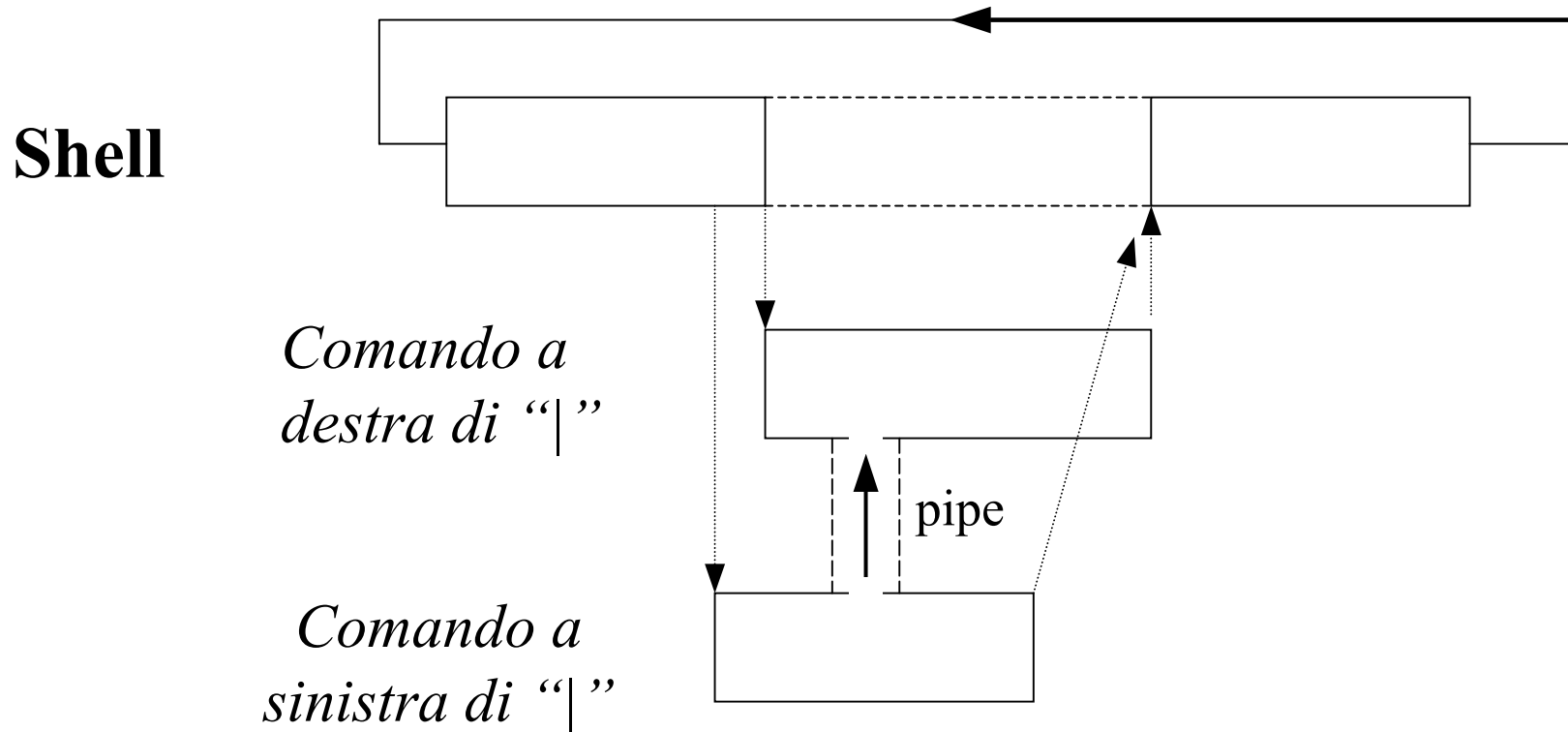
Realizzazione di una semplice shell che sia in grado di accettare sulla stessa linea più comandi separati dal simbolo “|”.

## **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 standard input del comando a destra di “|”.

**Nota:** questo utilizzo del simbolo “|” é molto diffuso sia in shell Unix che Windows

# Struttura del programma



**Unix:** generazione di nuovi processi con accoppiata `fork()` - `execvp()`

**Windows:** generazione di nuovi processi con `CreateProcess()`;

# Descrizione del main()

```
int main ()
{
    /* Dichiarazione variabili */

    while (1) {

        /* Legge la linea di comando -      *
         * esce se il comando e' "exit"     */

        do {

            /* Genera un processo con gli eventuali pipe sullo *
             * stdout eo sullo stdin                               */

        } while (next_command != NULL);

        /* Attende che i comandi lanciati siano terminati */

    }

    /* Terminazione della shell */
}
```

# Unix - Inclusioni

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

#define COMMAND_LENGTH 1024
```

## Dichiarazioni di variabili del main()

```
char *command_pointer, *next_command;
char line[COMMAND_LENGTH+1];
char **arg;
int *old_pipe_descriptors;
int *new_pipe_descriptors;
int arg_num;
int pipe_present=0, pipe_pending=0;
int pending_processes;
int i, status;
```

```

int main ()
{
    /* Dichiarazione variabili */

    while (1) {

        /* Legge la linea di comando -      *
         * esce se il comando e' "exit"    */

        do {

            /* Genera un processo con gli eventuali pipe sullo *
             * stdout eo sullo stdin                               */

        } while (next_command != NULL);

        /* Attende che i comandi lanciati siano terminati */

    }

    /* Terminazione della shell */
}

```

# Unix – lettura linea di comando

```
printf("\nExample Shell");
if (geteuid() == 0) printf("#");
else printf(">");
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;
if (strcmp(command_pointer, "exit\n") == 0) break;
```

```

int main ()
{
    /* Dichiarazione variabili */

    while (1) {

        /* Legge la linea di comando -      *
         * esce se il comando e' "exit"     */

        do {

            /* Genera un processo con gli eventuali pipe sullo *
             * stdout eo sullo stdin                               */

        } while (next_command != NULL);

        /* Attende che i comandi lanciati siano terminati */

    }

    /* Terminazione della shell */
}

```

# Unix – 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);  
    }  
}
```

# Unix - Generazione di un processo (I)

```
if ((i=fork()) == 0) {  
  
    if (pipe_pending) {  
        close(0);  
        dup(old_pipe_descriptors[0]);  
        close(old_pipe_descriptors[0]);  
    }  
  
    if (pipe_present) {  
        close(1);  
        dup(new_pipe_descriptors[1]);  
        close(new_pipe_descriptors[1]);  
    }  
  
    execvp(arg[0], arg);  
    printf("Can't execute file %s\n", arg[0]);  
    fflush(stdout);  
    exit(-1);  
} else .....
```

# Unix – Generazione di un processo (II)

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

```

int main ()
{
    /* Dichiarazione variabili */

    while (1) {

        /* Legge la linea di comando -      *
         * esce se il comando e' "exit"      */

        do {

            /* Genera un processo con gli eventuali pipe sullo *
             * stdout eo sullo stdin                               */

        } while (next_command != NULL);

        /* Attende che i comandi lanciati siano terminati */

    }

    /* Terminazione della shell */
}

```

# Unix - Attesa terminazione processi

```
for (i=0; i<pending_processes; i++) {  
    wait(&status);  
}
```

# Unix - Terminazione Shell

```
printf( "\nLeaving Shell\n\n" );  
return(0);
```

# Funzione count\_args()

```
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, " \n")) == NULL) ||
        (strcmp(temp, "\0") == 0)) return(-1);

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

# Funzione build\_arg\_vector()

```
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);  
}
```

# Differenze Unix-Windows

```
int main ()
{
    /* Dichiarazione variabili */

    while (1) {

        /* Legge la linea di comando -      *
         * esce se il comando e' "exit"    */

        do {
            /* Genera un processo con gli eventuali pipe sullo *
             * stdout eo sullo stdin                               */
        } while (next_command != NULL);

        /* Attende che i comandi lanciati siano terminati */

    }

    /* Terminazione della shell */
}
```

# Windows - Inclusioni

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>
```

```
#define COMMAND_LENGTH 1024
```

## Dichiarazioni di variabili del main()

```
char *command_pointer, *next_command;
char line[COMMAND_LENGTH+1];
char command_line[COMMAND_LENGTH+1];
char **arg;
HANDLE *old_pipe_handles;
HANDLE *new_pipe_handles;
SECURITY_ATTRIBUTES security;
HANDLE temp_readHandle;
HANDLE temp_writeHandle;
HANDLE pending_processes_handles[MAXIMUM_WAIT_OBJECTS];
BOOL newprocess;
STARTUPINFO si;
PROCESS_INFORMATION pi;
int arg_num;
int pipe_present=0, pipe_pending=0;
int pending_processes;
```

# Windows – 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);  
    }  
}
```

# Windows - Generazione di processo (I)

```
if (pipe_pending) {
    temp_readHandle = GetStdHandle(STD_INPUT_HANDLE);
    SetStdHandle(STD_INPUT_HANDLE, old_pipe_handles[0]);
}
if (pipe_present) {
    temp_writeHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    SetStdHandle(STD_OUTPUT_HANDLE, new_pipe_handles[1]);
}

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);
```

# Windows - Generazione di processo (II)

```
if (pipe_pending) {
    SetStdHandle(STD_INPUT_HANDLE, temp_readHandle);
    CloseHandle(old_pipe_handles[0]);
    free(old_pipe_handles);
    pipe_pending = 0;
}
if (pipe_present) {
    SetStdHandle(STD_OUTPUT_HANDLE, temp_writeHandle);
    CloseHandle(new_pipe_handles[1]);
    old_pipe_handles = new_pipe_handles;
    pipe_pending = 1;
}

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

# Windows - Attesa terminazione processi

```
do {  
  
    /* Generazione del processo */  
  
    pending_processes_handles[pending_processes] = pi.hProcess;  
    pending_processes++;  
  
    ....  
  
} while (next_command != NULL);  
  
.....  
.....  
.....  
  
WaitForMultipleObjects(pending_processes,  
                        pending_processes_handles, TRUE, INFINITE);
```

# Windows - Attesa terminazione processi

```
do {  
  
    /* Generazione del processo */  
  
    pending_processes_handles[pending_processes] = pi.hProcess;  
    pending_processes++;  
  
    ....  
  
} while (next_command != NULL);  
  
.....  
.....  
.....  
  
WaitForMultipleObjects(pending_processes,  
                        pending_processes_handles, TRUE, INFINITE);
```

# Windows – build\_command\_line()

```
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;  
}
```