ELEMENTI DI PROGRAMMAZIONE NEI S.O.

Danilo Croce

Ottobre 2024

IL MONDO SECONDO IL LINGUAGGIO C

- C è stato creato da Dennis Ritchie nel 1972 per sviluppare programmi UNIX.
- Alcune delle caratteristiche originali di UNIX sono ancora visibili
- “Everything is a file”

UNIX: EVERYTHING IS A FILE

- Sockets
- Devices
- Hard drives
- Stampanti
- Modems
- Pipes
- …

C: EVERYTHING IS A FILE

Descriptive Name	Short Name	File Number	Description
Standard In	stdin	0	Input from the keyboard
Standard Out	stdout	1	Output to the console
Standard Error	stderr	2	Error output to the console

Per impostazione predefinita ogni processo inizia con questi 3 "file"… "aperti"

HELLO WORLD? (1 OF 3)

Cosa dobbiamo fare per stampare "Hello World" sulla console (output standard)?

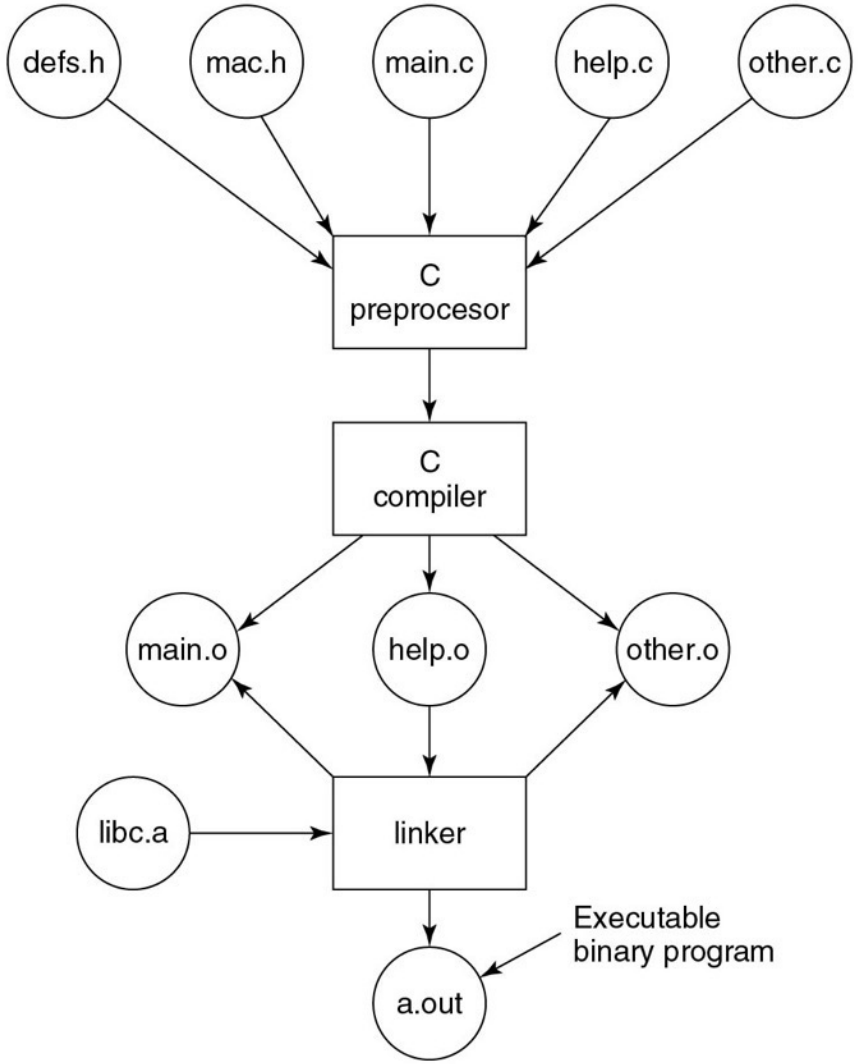
- `printf(char *str, …);`

HELLO WORLD! (1 OF 3)

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World!\n");
    return 0;
}
```


BUILD PROCESS



Il processo di compilazione dei file C e degli header file per creare un programma binario eseguibile.

HELLO WORLD? (2 OF 3)

Cosa dobbiamo fare per stampare "Hello World" sulla console (output standard)?

- `printf(char *str, …);`
- `int write(int fd, char *buf, size_t len);`

IN EFFETTI: EVERYTHING IS A FILE!

Per impostazione predefinita ogni processo inizia con 3 "file" aperti

Descriptive Name	Short Name	File Number	Description
Standard In	stdin	0	Input from the keyboard
Standard Out	stdout	1	Output to the console
Standard Error	stderr	2	Error output to the console

HELLO WORLD! (2 OF 3)

```
#include <unistd.h>
#define STDOUT 1

int main(int argc, char **argv)
{
    char msg[] = "Hello World!\n";
    write(STDOUT, msg, sizeof(msg));
    return 0;
}
```


HELLO WORLD? (3 OF 3)

Cosa dobbiamo fare per stampare "Hello World" sulla console (output standard)?

- `printf(char *str, …);`
- `int write(int fd, char *buf, size_t len);`
- `int syscall(int number, ...);`

HELLO WORLD! (3 OF 3)

```
#include <unistd.h>
#include <stdio.h>
#include <sys/syscall.h>
#define STDOUT 1

int main(int argc, char **argv)
{
     char msg[] = "Hello World!\n";
     int nr = SYS_write;
     syscall(nr, STDOUT, msg, sizeof(msg));
     return 0;
}
```


ESEMPI

- Fare riferimento ai file di esempio
  - `5.1_hello_world_1.c`
  - `5.1_hello_world_2.c`
  - `5.1_hello_world_3.c`

Mostrati a lezione

STANDARD LIBRARY

- Libc fornisce utili wrapper intorno alle syscall
  - Ad esempio `write`, `read`, `exit`
- È necessario chiamare la `syscall` oppure l'istruzione `int 0x80`
  - Fatto in assembly

```
syscall(int nr, …)
```


... TANTE SYSCALL

https://filippo.io/linux-syscall-table/

Se fate click su Entry Point
finirete nel GITHUB di Linus Torvalds

The **return value** is placed in `%rax`.

%rax	Name	Manual	Entry point
0	read	read(2)	sys_read

%rdi	%rsi	%rdx
unsigned int fd	char *buf	size_t count

%rax	Name	Manual	Entry point
1	write	write(2)	sys_write
2	open	open(2)	sys_open
3	close	close(2)	sys_close

%rdi
unsigned int fd

%rax	Name	Manual	Entry point
4	stat	stat(2)	sys_newstat
5	fstat	fstat(2)	sys_newfstat
6	lstat	lstat(2)	sys_newlstat
7	poll	poll(2)	sys_poll

UNA LUNGA ATTRAVERSATA... DA `read()` A `sys_read()`

Cosa succede quando viene invocata la funzione read( )?

- Di seguito viene riportata la sequenza dei passaggi
- Disclaimer: alcuni punti sono approssimati ma danno l'idea della sequenza di funzioni invocati

1. Funzione `read()` in glibc

- La funzione read() è implementata in `glibc` e funge da wrapper per la chiamata di sistema read.
- **File:** `glibc/sysdeps/unix/sysv/linux/read.c`
- Link: `https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/read.c.html`

2. Macro `SYSCALL_CANCEL`

- La macro `SYSCALL_CANCEL` viene utilizzata in `read.c` per effettuare la chiamata di sistema effettiva in modo sicuro (e.g., problemi sul multithreading).
- **File:** `glibc/sysdeps/unix/sysdep.h`
- **Link:** `https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysdep.h.html`

UNA LUNGA ATTRAVERSATA... DA `read()` A `sys_read()`

3. Macro `INTERNAL_SYSCALL`

- `SYSCALL_CANCEL` si basa su `INTERNAL_SYSCALL` per configurare i registri e invocare l'istruzione syscall, che fa passare il controllo dal contesto utente a quello kernel.
- **File:** `glibc/sysdeps/unix/sysv/linux/x86_64/sysdep.h`
- **Link:** `https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/x86_64/sysdep.h.html`

4. Punto di ingresso nel kernel (`entry_SYSCALL_64`)

- L'istruzione syscall provoca il passaggio alla modalità kernel e indirizza l'esecuzione a entry_SYSCALL_64, definito in assembly.
- **File:** `arch/x86/entry/entry_64.S` nel kernel Linux
- **Link:** `https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_64.S`

5. Funzione `do_syscall_64`

- `entry_SYSCALL_64` chiama `do_syscall_64`, una funzione C che determina quale funzione kernel chiamare basandosi sul numero della chiamata di sistema.
- **File:** `arch/x86/entry/common.c`
- **Link:** `https://github.com/torvalds/linux/blob/master/arch/x86/entry/common.c`

UNA LUNGA ATTRAVERSATA... DA `read()` A `sys_read()`

6. Tabella delle chiamate di sistema (`sys_call_table`)

- `do_syscall_64` **consulta** `sys_call_table` **per mappare il numero di system call alla funzione kernel appropriata. Per** `read`**, il numero è 0 e punta a** `__x64_sys_read`**.**
- **File:** `arch/x86/entry/syscalls/syscall_64.tbl`
- **Link:** `https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl`

7. Funzione `__x64_sys_read`

- **La funzione** `__x64_sys_read` **è il wrapper per read nel kernel, definita con** `SYSCALL_DEFINE3`**, che richiama** `ksys_read` **per la logica di lettura.**
- **File:** `fs/read_write.c`
- **Link:** `https://github.com/torvalds/linux/blob/master/fs/read_write.c`

CREAZIONE DI PROCESSI

PROCESS MANAGEMENT SYSTEM CALL

- Creiamo una shell minimale:
  - Attende che l'utente digiti un comando
  - Avvia un processo per eseguire il comando
  - Attende che il processo sia terminato

`(fork, wait, execv)`

CREAZIONE DEL PROCESSO (1 OF 2)

- `pid_t fork()`
  - Duplica il processo corrente
    - Restituisce il pid del figlio nel chiamante (genitore)
    - Restituisce 0 nel nuovo processo (figlio)
- `pid_t wait(int *wstatus)`
  - Attende che i processi figli cambino stato
  - Scrive lo stato in `wstatus`
  - Ad esempio, a causa di un exit o segnale

FORK, WAIT

```
void main(void)
{
     int pid, child_status;
     if (fork() == 0) {
          do_something_in_child();
     } else {
          wait(&child_status); // Wait for child
     }
}
```


CREAZIONE DEL PROCESSO (2 OF 2)

- `int execv(const char *path, char *constargv[ ]);`
  - **Carica un nuovo binario (path) nel processo corrente, rimuovendo tutte le altre mappature di memoria.**
  - `constargv` **contiene gli argomenti del programma**
  - **L'ultimo argomento è** `NULL`
  - **E.g.,** `constargv = {“/bin/ls”, “-a”, NULL}`
  - **Diverse varianti di exec(v)(p) (controllare le pagine** `man`**)**

FORK, WAIT, EXECV

```
void  main(void)
{
     int    pid, child_status;
     char   *args[] = {“/bin/ls”, “-l”, NULL};
     if (fork() == 0) {              // fork creates child process
         execv(args[0], args);       // in child: load+execute program
     } else {
         wait(&child_status);        // Wait for child
     }
}
```


UNA SHELL MINIMALE

```
while (1) {
    char cmd[256], *args[256];
    int status;
    pid_t pid;
    read_command(cmd, args); /* reads command and arguments from command line */

    pid = fork();

    if (pid == 0) {
        execv(cmd, args);
        exit(1);
    } else {
        wait(&status);
    }
}
```


COME TERMINARE I PROGRAMMI?

- Ctrl+C, ma come funziona?
- Risposta: i **signali**

GESTIONE DEI SEGNALI

CHIAMATE DI SISTEMA PER I SEGNALI

- A volte i processi devono essere interrotti durante la loro esecuzione.
- Viene inviato un **segnale** al processo che deve essere interrotto.
- Il processo interrotto può catturare il segnale installando un gestore di segnali (**signal handler)**
- Cosa succede quando l'utente del terminale preme `CTRL+C` o `CTRL+Z` ?

(signal, alarm, kill)

SIGNAL, ALARM, KILL

- `sighandler_t signal(int signum, sighandler_t handler)`
  - **Registra un gestore di segnali per il segnale** `signum`
- `unsigned int alarm(unsigned int seconds)`
  - **Consegna SIGALRM in un numero di secondi specificato**
- `int kill(pid_t pid, int sig)`
  - Consegna il segnale `sig` al processo `pid` **(non uccide!!!)**

ALARM EXAMPLE

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

void alarm_handler(int signal)
{
    printf("In signal handler: caught signal %d!\n",signal);
    exit(0);
}
int main(int argc, char **argv)
{
    signal(SIGALRM, alarm_handler);
    alarm(1); // alarm will send signal after 1 sec


    while (1) {
        printf("I am running!\n");
    }
    return 0;
}
```


COMUNICAZIONE TRA PROCESSI ATTRAVERSO PIPE

PIPE EXAMPLE (1 OF 2)

Cosa succede se si esegue il seguente comando?

```
$ cat names.txt | sort
```

E i seguenti comandi?

```
$ mkfifo named.pipe
$ echo “Hello World!” > named.pipe
$ cat named.pipe
```

(open, close, pipe, dup)

OPEN, CLOSE, PIPE, DUP

- `int open(const char *pathname, int flags)`
  - **Apre il file specificato dal nome del percorso (`pathname`)**
- `int close(int fd)`
  - **Chiude il descrittore di file specificato `fd`**
- `int pipe(int pipefd[2])`
  - **Crea una `pipe` con due `fd` per le sue estremità**
- `int dup(int oldfd)`
  - **Crea una copia del descrittore di file `oldfd` utilizzando il descrittore di file inutilizzato con il numero più basso per la copia.**

MA DUP A CHE SERVE?

- `dup` e `dup2` sono chiamate di sistema su Unix/Linux per duplicare descrittori di file.
- **Funzione**:
  - `dup(int oldfd)`: Crea un duplicato del descrittore `oldfd`.
  - `dup2(int oldfd, int newfd)`: Duplica oldfd su un descrittore specifico `newfd`.
- **Perché?:** è comune voler reindirizzare l'output di un programma verso un file o un altro programma. `dup` e `dup2` permettono di «agganciare» il descrittore di output standard (`STDOUT_FILENO`, ovvero `1`) o di input standard (`STDIN_FILENO`, ovvero `0`) a un file.
- **Utilità**: Permettono la gestione avanzata di input/output, come il reindirizzamento di file, la creazione di pipeline tra processi, e la gestione di input/output in processi figli.

ESEMPIO 1: REINDIRIZZAMENTO DI OUTPUT

- **Scenario**: Salvare l'output di un programma in un file.
- **Esempio**:

```
int file_fd = open("output.txt", O_WRONLY | O_CREAT, 0644);
dup2(file_fd, STDOUT_FILENO);  // Reindirizza stdout su file_fd
printf("Questo va in output.txt\n");
close(file_fd);
```

- **Risultato**: L'output viene scritto in output.txt invece che in console.
- **Utilizzo tipico**: Registrazione di log, salvataggio di output in file.

ESEMPIO 2: CREAZIONE DI PIPELINE TRA PROCESSI

- **Scenario**: Collegare l'output di un programma all'input di un altro

  Esempio: `ps aux | grep httpd`

```
int fd[2];
pipe(fd);
if (fork() == 0) {
    close(fd[0]);
    dup2(fd[1], STDOUT_FILENO);  // Output di `ps aux` nella pipe
    execlp("ps", "ps", "aux", NULL);
} else {
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);   // Input di `grep` dalla pipe
    execlp("grep", "grep", "httpd", NULL);
}
```

- **Risultato**: `ps aux` manda l'output a `grep`, simulando una pipeline.
- **Utilizzo tipico**: Shell scripting, automazione di comandi.

ESEMPIO 3: GESTIONE DEI PROCESSI FIGLI

- **Scenario**: Reindirizzare l'output di un processo figlio a un file di log.

```
if (fork() == 0) {
    int log_fd = open("logfile.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
    dup2(log_fd, STDOUT_FILENO);  // Output va in logfile.txt
    close(log_fd);
    execlp("ls", "ls", "-l", NULL);  // Comando eseguito dal figlio
}
```

- **Risultato**: L'output del processo figlio `ls -l` va in `logfile.txt`
- **Nota**: `execlp` sostituisce il processo corrente con il processo specificata dal file
- **Utilizzo tipico**: Logging centralizzato, configurazione di ambienti di processo.

PIPE EXAMPLE (2 OF 2)

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

#define STDIN  0
#define STDOUT 1

#define PIPE_RD 0
#define PIPE_WR 1

int main(int argc, char** argv)
{
    pid_t cat_pid, sort_pid;
    int fd[2];

    pipe(fd);

    cat_pid = fork();
    if ( cat_pid == 0 ) {
        close(fd[PIPE_RD]);
        close(STDOUT);
        dup(fd[PIPE_WR]);
        execl("/bin/cat", "cat", "names.txt" , NULL);
    }

    sort_pid = fork();
    if ( sort_pid == 0 ) {
        close(fd[PIPE_WR]);
        close(STDIN);
        dup(fd[PIPE_RD]);
        execl("/usr/bin/sort", "sort", NULL);
    }

    close(fd[PIPE_RD]);
    close(fd[PIPE_WR]);


    /* wait for children to finish */
    waitpid(cat_pid, NULL, 0);
    waitpid(sort_pid, NULL, 0);

    return 0;
}
```


POSSIAMO SALTARE QUESTE "CHIUSURE"?

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

#define STDIN  0
#define STDOUT 1

#define PIPE_RD 0
#define PIPE_WR 1

int main(int argc, char** argv)
{
    pid_t cat_pid, sort_pid;
    int fd[2];

    pipe(fd);

    cat_pid = fork();
    if ( cat_pid == 0 ) {
        close(fd[PIPE_RD]);
        close(STDOUT);
        dup(fd[PIPE_WR]);
        execl("/bin/cat", "cat", "names.txt" , NULL);
    }
```

```c
    sort_pid = fork();
    if ( sort_pid == 0 ) {
        close(fd[PIPE_WR]);
        close(STDIN);
        dup(fd[PIPE_RD]);
        execl("/usr/bin/sort", "sort", NULL);
    }

    close(fd[PIPE_RD]);
    close(fd[PIPE_WR]);

    /* wait for children to finish */
    waitpid(cat_pid, NULL, 0);
    waitpid(sort_pid, NULL, 0);

    return 0;
}
```

And why / why not?

POSSIAMO SALTARE QUESTE "CHIUSURE"? (RISPOSTA)

1. **Evitare Blocchi:** Chiudi la fine di lettura di una pipe per impedire al processo di scrittura di rimanere bloccato.
2. **Ricezione EOF:** sort aspetta un EOF per terminare la lettura. Chiudi la fine di scrittura per inviare un EOF a sort.
3. **Evitare Letture Accidentali:** Nel processo cat, chiudi la fine di lettura per evitare letture inaspettate dalla pipe.
4. **Reindirizzamento di STDOUT in cat:** Dopo la duplicazione, chiudi il descriptor originale per garantire che cat scriva solo nella pipe.
5. **Reindirizzamento di STDIN in sort:** Dopo la duplicazione, chiudi il descriptor originale per assicurarti che sort legga solo dalla pipe.
6. **Descriptor nel Processo Padre:** Dopo la fork, il processo padre dovrebbe chiudere entrambe le estremità della pipe.

ESEMPI

- Creazione di processi paralleli con metodo fork
- `5.2_my_first_fork_1.c`
  - un semplice esempio di invocazione del metodo fork
- `5.2_my_first_fork_2.c`
  - creazione dei processi e avvio di applicazioni da codice C
- `5.3_my_signal_1.c`
  - Gestione dei segnali
- `5.3_my_signal_2.c`
  - Gestione degli allarmi

ESEMPI (CONT)

- `5.4_fork_pipe.c`
  - **Comunicazione tra processi attraverso PIPE**
- `5.5_my_first_bash.c`
  - **Sviluppo di una BASH minimale in C**

- NON sottovalutate lo script usato per compilare gli esempi

ESERCIZIO

- Un processo genera due processi figli P1 e P2.
- Il figlio P1 esegue un ciclo indeterminato durante il quale genera casualmente numeri interi compresi tra 0 e 100.
  - P1 comunica, ad ogni iterazione, il numero al padre solo se esso è dispari.
- P2 fa la stessa cosa ma comunica al padre solo i numeri pari.
- Il padre, per ogni coppia di numeri che riceve dai figli ne fa la somma e la visualizza.

- Il programma deve terminare quando la somma dei due numeri ricevuti supera il valore 190: il padre, allora, invia un segnale di terminazione a ciascuno dei figli.