

ELEMENTI DI PROGRAMMAZIONE NEI S.O.

Danilo Croce

Ottobre 2023



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, ...);`



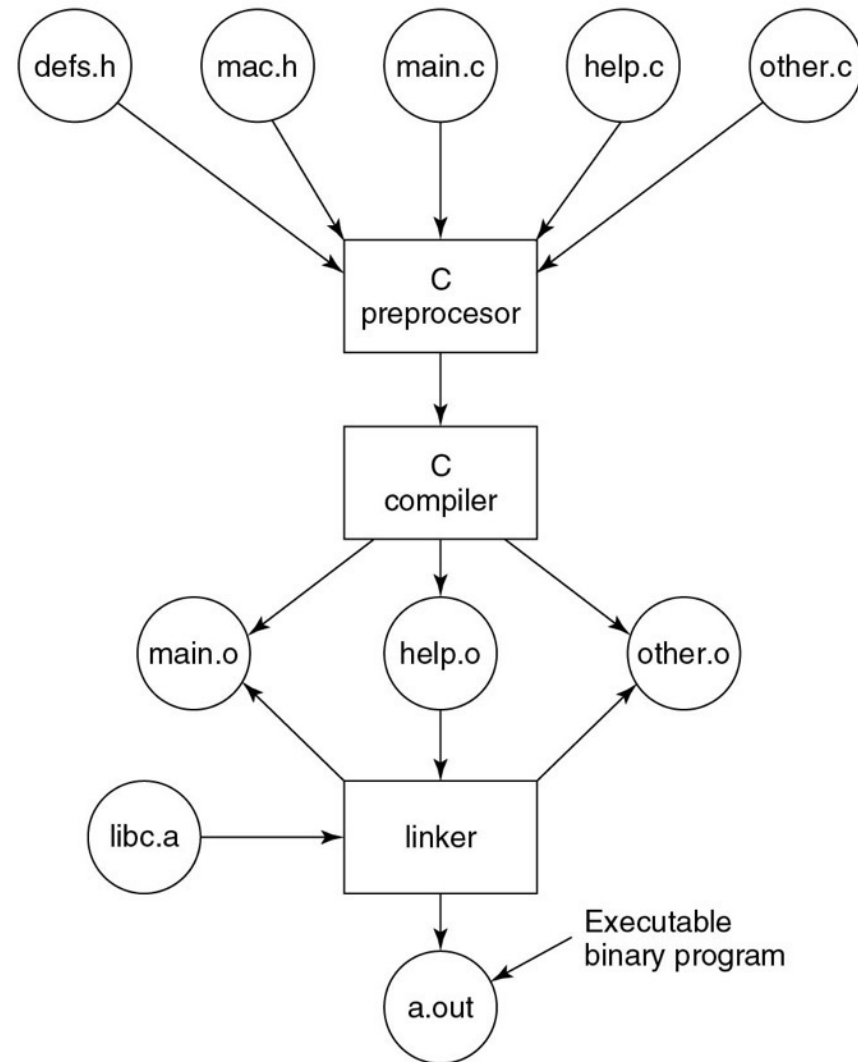
HELLO WORLD! (1 OF 3)

```
#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

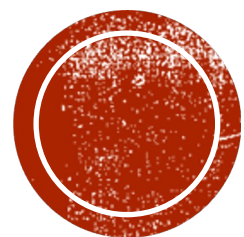


STANDARD LIBRARY

- 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, ...)
```





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 execlv(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, EXEVC

```
void main(void)
{
    int    pid, child_status;
    char    *args[] = {"/bin/ls", "-l", NULL};
    if (fork() == 0) {                // fork creates child process
        exevc(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();

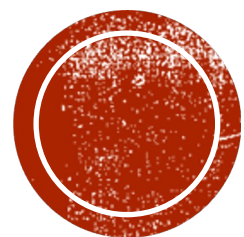
    if (pid == 0) {
        execv(cmd, args);
        exit(1);
    } else {
        wait(&status);
    }
}
```



COME TERMINARE I PROGRAMMI?

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





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>

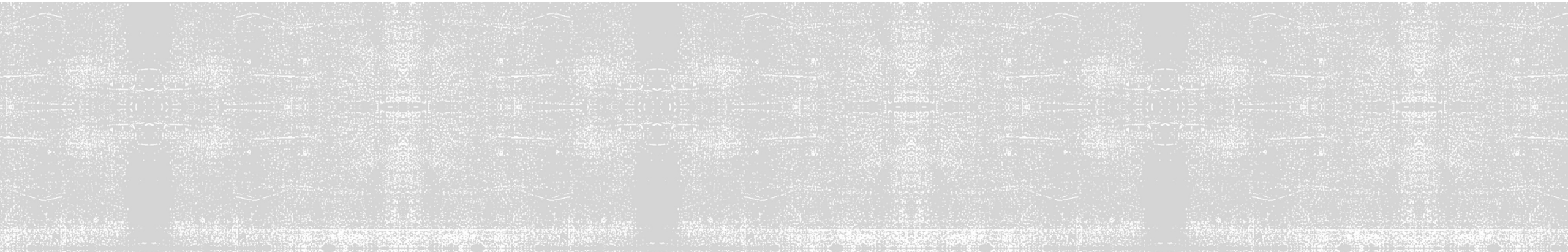
void alarm_handler(int signal)
{
    printf("In signal handler: caught signal %d!\n",signal);
    exit(0);
}
int main(int argc, char **argv)
{
    signal(SIGALRM, alarm_handler);
    alarm(1); // alarm will send signal after 1 sec

    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.**



PIPE EXAMPLE (2 OF 2)

```
#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”?

```
#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;
}
```

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.

