Laboratorio di Sistemi Operativi

Luca De Paulis

6 maggio 2021

Indice

1

Lezione 1

1.1 FUNZIONI CON UN NUMERO VARIABILE DI ARGOMENTI

Alcune funzioni del C hanno un numero variabile di argomenti: ad esempio la `printf` può accettare un singolo argomento (solo una stringa), ma anche 2, 3, ..., a seconda di quanti placeholder (come `%d`, `%s`, eccetera) ci sono.

```c
char* str = "Mario";
int a = 3;
int b = 5;

printf("Ciao!");
printf("Ciao %s!", str); // prints "Ciao Mario!"
printf("%d + %d = %d", a, b, a + b) // prints "3 + 5 = 8"
```

Per dichiarare funzioni con un numero variabile di argomenti bisogna usare l'*include file* `stdarg.h`.

Facciamo un esempio con una funzione che prende un intero `count` che rappresenta il numero di argomenti passati alla funzione e fa la somma di tutti gli argomenti passati successivamente.

```c
#include <stdarg.h>

int va_sum(int count, ...){
    // va_list is the type for varadic arguments
    va_list ap;
    // every va_list must be initialized with va_start!
    va_start(ap, count);
    //             |- - - > last non-variadic argument
    int sum = 0;
    for(int i = 0; i < count; i++)
        sum += va_arg(ap, int); // gets the next argument
                /*    |    |- - > type of the arg, must be known statically!
                      |- - > va_list
                */
    // every va_list must be "freed" with va_end
    va_end(ap);
}
```

In questo programma abbiamo usato diverse funzionalità forniteci da `stdarg.h`:

- il tipo `va_list` rappresenta la lista di tutti gli argomenti variabili;
- la macro `va_start` che serve ad inizializzare una lista e prende due argomenti: una lista di argomenti variabili (di tipo `va_list`) e **l'ultimo argomento non-variabile**;
- la macro `va_arg` serve a ottenere il prossimo argomento: prende la `va_list` e un **tipo**, che deve essere conosciuto a tempo di compilazione;
- la macro `va_end` che serve a liberare la memoria usata dalla `va_list`.

2

Lezione 2

2.1 PUNTATORI A FUNZIONE

Il C permette di definire puntatori a diversi tipi di oggetti: puntatori a `int`, a `char`, puntatori a puntatori, ecc... In particolare, consente anche la definizione di **puntatori a funzione**:

```
    int somma (int a, int b) {
        return a + b;
    }
    /*
    - - - - > return type
    |           - - - - -> parameter types
    |           |    |     */
    int (*fun) (int, int);
    fun = somma;

    int a = fun(3, 6); // a <- 9
```

A cosa servono i puntatori a funzione? Dopotutto, il codice precedente poteva essere semplificato usando direttamente `somma` invece di `fun`.

Un possibile caso d'uso dei puntatori a funzione è nel caso delle cosiddette *funzioni di ordine superiore*, ovvero funzioni che prendono come argomento una funzione. Un classico esempio di funzione di ordine superiore è `map`: matematicamente, data una collezione di dati S (che sia un insieme, una lista, un array, ecc...) e una funzione f, l'applicazione `map(f, S)` mi restituisce la collezione S dove ad ogni elemento $x \in S$ sostituisco il numero $f(x)$.

Esempio 2.1.1. Dato l'insieme $S = \{1, 2, 3, 4\}$ e la funzione $f(x) = 2x + 1$, il risultato di `map(f, s)` è $\{3, 5, 7, 9\}$: ho *mappato* la funzione f su tutti gli elementi della collezione S.

Una possibile implementazione della `map` (di funzioni `int` $\longrightarrow$ `int` su array di interi) in C è la seguente.

```
        /*    fun has type
                int -> int
              ------------------  */
    void map(int (*fun)(int), int arr[], size_t len){
        for(int i = 0; i < len; i++)
            arr[i] = fun(arr[i]);
}
```

Dato che i tipi di puntatori a funzione possono essere poco chiari è possibile definire degli *alias* tramite la `typedef`:

```c
// defines a new type, called myFuncType
typedef int (*myFuncType)(int);
// now we can define map like this
void map(myFunctType fun, int arr[], size_t len){
    for(int i = 0; i < len; i++)
        arr[i] = fun(arr[i]);
}
```

2.2 PUNTATORI GENERICI

Tra tutti i tipi di puntatore ne esiste uno particolare, indicato dal C come `void*`. Una variabile di tipo `void*` è un *puntatore a tipo generico*, nel senso che può contenere un puntatore a qualsiasi tipo.

```c
void* ptr;
int a = 50;
char c = 'C';

// both allowed!
ptr = &a;
ptr = &c;

// not allowed!
printf("\%c", *ptr);
// a void* pointer must be cast before dereferenced!
printf("\%c", *(char*)ptr); // prints 'C'
```

Un puntatore di tipo `void*` non può essere dereferenziato: siccome non ne conosciamo il tipo non sappiamo quanto grande è l'area di memoria da esso puntata. Bisogna quindi prima convertirlo in un puntatore con un tipo ben definito e poi dereferenziarlo per accedere al contenuto del puntatore.

I puntatori di tipo `void*` possono quindi essere usati per creare funzioni e algoritmi che non dipendono dal tipo, degli elementi considerati. Un esempio classico di ciò è la funzione di libreria `qsort` che implementa un *quicksort generico*: la firma della funzione è infatti

```c
void qsort(
    void* base,                 // array to sort
    size_t nmemb,               // number of elements in the array
    size_t size,                // size of a single element of the array
    int (*cmp) (void*, void*)   // auxiliary function that compares two elements
);
```

La funzione `qsort` prende quindi un puntatore `void*`, ovvero un array di tipo generico, insieme al numero di elementi che contiene (`size_t nmemb`) e alla grandezza in byte di ogni elemento (`size_t size`) e ad un **puntatore a funzione** che prende due elementi di tipo generico e restituisce un intero (che, come spiegato dal manuale, deve essere `0` se i due elementi sono uguali, un numero positivo se il primo è più grande del secondo e un numero negativo altrimenti).

Possiamo usare la `qsort` in questo modo:

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

```c
int cmp_int(void* ptr1, void* ptr2){
    // casting both pointers to int* and then deferencing them to get the int value
    int n = *(int*)ptr1;
    int m = *(int*)ptr2;
    // n - m = 0 if they are equal, > 0 if n > m and < 0 otherwise, as requested
    return n - m;
}
int cmp_float(void* ptr1, void* ptr2){
    // casting both pointers to float* and then deferencing them to get the float
    ↪  value
    float x = *(float*)ptr1;
    float y = *(float*)ptr2;

    // x - y = 0 if they are equal, > 0 if x > y and < 0 otherwise, as requested
    return x - y;
}

int main(int argc, char* argv[]){
    int array_int[] = {2, 3, 1, 8, -4};
    float array_float[] = {3.14, 2.1828, -1.61};

    // sorting the two arrays with different types
    qsort(array_int, 5, sizeof(int), cmp_int);
    qsort(array_float, 3, sizeof(float), cmp_float);
}
```

2.3 FUNZIONI RIENTRANTI

Ad esempio la seguente funzione non è rientrante:

```c
int x = 5;

int func(){
    printf("Beginning: x = %d\n", x);
    x = x+2;
    printf("End: x = %d\n", x);
}
```

Infatti se interrompiamo la funzione dopo l'assegnamento `x = x+2` ed eseguiamo qualche altra cosa, il contenuto della variabile globale `x` può essere modificato e la funzione non stampa più il numero `x+2`.

Una funzione rientrante non può fare uso di variabili globali o statiche.

Nel resto del corso varemo sempre e solo uso di funzioni rientranti, spesso indicate con il suffisso `_r`.

Tokenizzazione di stringhe

Tokenizzare una stringa significa spezzarla ad ogni occorrenza di un determinato carattere. Per far ciò può essere utile usare le funzioni `strtok` e la sua corrispondente versione rientrante, `strtok_r`.

La versione non rientrante è la seguente:

```c
int main(){
    char string[] = "Hello crazy world";
    // token represents the single portion of the string
    char* token = strtok(string, " ");
                /*          |     | - -> separator, here it's a space
                            | - -> string to tokenize
                */
    // keep reading single tokens
    while(token){
        printf("%s\n", token);
        // calling strtok with NULL uses the string tokenized before,
        // using a hidden global variabile
        token = strtok(NULL, " ");
    }
}
```

Per renderla rientrante dobbiamo eliminare la variabile globale nascosta: la versione rientrante `strtok_r` ha infatti un parametro aggiuntivo, usato nel seguente modo.

```c
int main(){
    char string[] = "Hello crazy world";
    // save variable, used by strtok_r
    char* save = NULL;
    // token represents the single portion of the string
    char* token = strtok_r(string, " ", &save);
                //                      | - -> local state!
    // keep reading single tokens
    while(token){
        printf("%s\n", token);
        // calling strtok with NULL uses the string tokenized with the state saved
        ↪  in "save"
        token = strtok(NULL, " ", &save);
    }
}
```

2.4 PREPROCESSORE

Il preprocessore è un programma invocato prima della compilazione di un file C. Il suo scopo è eseguire **sostituzioni testuali** nel codice, e le **direttive al preprocessore** iniziano tutte con il simbolo `#`.

Include files

Il primo scopo del preprocessore è quello di includere i cosiddetti *include files* tramite la direttiva `#include`: la direttiva `#include <file.h>` copia il contenuto del file `file.h` all'interno del file corrente.

La forma vista sopra (`#include <file.h>`) cerca il file `file.h` nelle directory standard, come ad esempio `/usr/bin`. Ne esiste anche un'altra (`#include "file.h"`) che cerca il file header nella directory corrente inizialmente, e passa alle directory standard solo se non trova l'header nella cartella corrente.

Macro

La direttiva `#define` serve a definire delle macro. Ve ne sono di tre tipi:

- `#define DEBUG` definisce una macro (cioè un nome) che non ha nessun valore associato. Vedremo l'utilità di questa direttiva parlando di *compilazione condizionale*.
- `#define SIZE 10` definisce una macro `SIZE` che verrà sostituita dal preprocessore con il valore `10`.
- `#define PROD(X, Y) (X)*(Y)` definisce una macro con *parametri*: al momento della sostituzione testuale il preprocessore sostituirà ogni occorrenza di `PROD(a, b)` con `(a)*(b)` *prima di compilare il programma*.

La sostituzione testuale è un procedimento potente, ma anche molto pericoloso: ad esempio se definissimo la macro `#define PROD(X, Y) X*Y`, che sembra identica a quella di prima, commetteremmo un errore:

```
x = PROD(3 + 1, 5)
// is substituted with
x = 3 + 1 * 5   // = 8
// and not with
x = (3 + 1) * 5 // = 20
```

Compilazione condizionale

Un uso molto comodo delle direttive al preprocessore è la *compilazione condizionale* tramite le direttive `#if`, `#ifdef`, `#ifndef` e `#endif`.

- Il codice tra `#if` e `#endif` è mantenuto se e solo se la condizione che segue `#if` è vera (ovvero è non-zero), altrimenti il preprocessore lo cancella.
- Il codice tra `#ifdef` e `#endif` è mantenuto se e solo se la macro che segue `#ifdef` è definita, altrimenti il preprocessore lo cancella.
- `#ifndef` si comporta dualmente a `#ifdef`: il codice viene cancellato quando la macro è definita.

Questo è utile per il debug:

```
#ifdef DEBUG
    // print statements or other debugging things
#endif

#if defined(DEBUG)
    // exactly as above
#endif
```

Quindi per debuggare il codice è sufficiente aggiungere `#define DEBUG` all'inizio del programma, e cancellarlo quando non si vuole più eseguire il codice di debugging.

Macro predefinite

Il C definisce alcune macro, come

- `__FILE__` che ci dà il nome del file;
- `__LINE__` che ci dà la linea corrente del file sorgente;
- molte altre.

3

Lezione 6

3.1 PROCESSI

Come visto nel modulo di Teoria, ogni volta che un programma viene eseguito il Sistema Operativo crea un **processo**, cioè una struttura dati contenente i dati necessari all'esecuzione di quel determinato programma.

I dati di un processo, contenuti nel PCB (*Process Control Block*), sono organizzati nel kernel dell'OS in una tabella, chiamata *Process Table*: la posizione di un processo in questa tabella è il suo identificativo, chiamato PID (*Process Identifier*).

Inoltre i processi sono organizzati in una struttura gerarchica: ogni processo ha un processo **padre** e può avere dei processi figli. Il primo processo, cioè il processo che non ha padre, si chiama init.

Per ottenere il PID di un processo o del processo padre in C si possono usare le funzioni `getpid()` e `getppid()`:

```c
#include <unistd.h>

// pid_t is a type alias for unsigned int
pid_t gitpid(); // returns process ID (no error return)
pid_t getppid(); // returns parent process ID (no error return)
```

3.2 FORK

Per creare un nuovo processo si usa la funzione `fork`: essa **duplica** tutto il contenuto del processo corrente (che diventa il padre nel nuovo processo). I due processi hanno quindi due PCB distinti (anche se uguali) e anche due tabelle dei descrittori dei file diverse; tuttavia condividono la tabella dei file aperti e il puntatore alla posizione corrente di ciascun file.

La funzione `fork` ha la seguente *signature*:

```c
pid_t fork();
```

La `fork` ritorna due valori diversi al processo padre e al processo figlio:

- al padre ritorna il PID del figlio;
- al figlio ritorna 0.

In questo modo possiamo distinguere tra il processo padre e il processo figlio e sfruttarli per cose diverse.

Se invece la `fork` fallisce ritorna -1 e setta `errno`.

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

int main(){
    pid_t pid;
    if( (pid = fork()) == -1){ // fork failed!
        perror("fork failed, main");
        exit(EXIT_FAILURE);
    }

    if(pid == 0){ // child process
        printf("I'm the child! My PID is %d, whereas my father's is %d.\n",
        ↪  getpid(), getppid());
        exit(EXIT_SUCCESS);
    } else {      // father process
        sleep(5); // sleeps for 5 seconds
        printf("I'm the father process! My PID is %d, whereas my child's is %d.\n",
        ↪  getpid(), pid);
    }

    return 0;
}
```

La `fork` è un'operazione costosa: infatti l'unica parte dei due PCB che può essere conservata è il segmento Text, che corrisponde al codice, mentre i segmenti Data, Heap e Stack devono essere duplicati poiché i due processi sono separati.

Nelle versioni moderne di UNIX la `fork` viene quindi implementata con un meccanismo *copy-on-write*: la copia dei dati del PCB viene fatta solo quando il figlio cerca di scrivere dei dati (ad esempio una variabile).

Questo meccanismo è molto conveniente poiché nella maggior parte dei casi non vogliamo avere due copie dello stesso processo in esecuzione, ma vogliamo eseguire un altro programma tramite le funzioni `exec*` (che vedremo nella prossima sezione).

3.3 EXEC

Spesso quello che vogliamo fare quando creiamo un nuovo processo non è avere due copie dello stesso processo, ma eseguire un programma diverso. Per far ciò sfruttiamo una sequenza di due funzioni: prima usiamo la `fork` per creare un nuovo PCB e dopo usiamo una funzione della famiglia delle `exec*` per *cambiare eseguibile*.

In generale una qualsiasi funzione della famiglia delle `exec*` segue il seguente procedimento:

- trova il file eseguibile da eseguire;
- usa i contenuti di quel file per sovrascrivere lo spazio di indirizzamento del processo che ha invocato la `exec`;
- carica nel program counter PC l'indirizzo iniziale.

Le funzioni della famiglia `exec*` sono 6 e sono `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`.

▸ `execl` Vediamo la `execl`:

```c
#include <unistd.h>

int execl(
    char* path, // path to the executable
    char* arg0, // first arg: name of the file
    char* arg1, // second arg
    ...
    char* argN, // Nth argument
    (char*) NULL   // the list must be NULL-terminated!
);
```

Se la `execl` fallisce ritorna -1 e setta `errno`, invece se ha successo **non ritorna**: infatti l'eseguibile è cambiato, quindi il vecchio eseguibile non esiste più e non potrebbe vedere il risultato restituito.

La lista di argomenti di lunghezza variabile passata ad `execl` deve essere terminata da un valore `NULL` e potrà essere usata dal nuovo programma attraverso `argv`.

Infine, il *path* specificato come primo argomento della `execl` deve essere un file eseguibile e deve avere i permessi per l'esecuzione per l'effective-user-ID del processo che invoca la `execl`.

▸ ALTRE EXEC Le altre funzioni della famiglia delle `exec*` possono essere distinte sulla base delle lettere che seguono "exec":

- se vogliamo passare gli argomenti al nuovo programma come una lista `NULL`-terminated usiamo la lettera `l`;
- se vogliamo passarli come array di argomenti che rispetta il formato di `argv` (ovvero l'ultimo argomento deve essere `NULL`) possiamo usare la lettera `v`;
- se vogliamo che la `exec` cerchi il file nelle directory specificate dalla variabile di ambiente PATH possiamo usare la lettera `p`;
- se vogliamo passare un array di stringhe che descrivono l'ambiente (*environment*) possiamo usare la lettera `e`; se non lo facciamo, l'ambiente viene preservato.

In generale il processo rimane lo stesso, quindi diversi attributi (come il PID, PPID, descrittori dei file aperti, ecc...) rimangono invariati. Gli attributi che possono cambiare sono

- la gestione dei segnali (la vedremo più avanti);
- effective-user-ID e effective-group-ID, se il nuovo eseguibile li ha settati;
- le funzioni registrate in `_atexit`;
- i segmenti di memoria condivisa;
- i semafori POSIX (che vengono resettati).

▸ FLUSH DEI BUFFER Osserviamo che se cambiamo eseguibile con la `exec` non stiamo terminando il programma precedente, quindi non c'è il *flush* automatico delle funzioni della libreria `stdio.h`.

Ad esempio il programma execl-test.c:

```c
int main () {
    printf("The quick brown fox jumped over");
    // calling "echo", the program that prints things
    execl("/bin/echo", "echo", "the", "lazy", "dogs", (char*)NULL);
    // if execl returns then an error occurred!
```

```
    perror("execl");
    return 1;
}
```

ha un effetto indesiderato:

```
$ ./execl-test
the lazy dogs
$
```

La chiamata a `printf` non ha avuto alcun effetto.

Infatti come abbiamo visto la `printf` (come molte delle funzioni della libreria `stdio.h`) è *bufferizzata*: le stringhe da scrivere non vengono subito inviate allo `stdout`, ma vengono memorizzate in un buffer e inviate quando il buffer è pieno oppure quando viene fatto un *flush* del buffer.

Generalmente il *flush* viene fatto ogni volta che un programma termina, ovvero quando chiama la funzione `exit`, tuttavia in questo caso a causa dell'`execl` la `exit` non viene chiamata.

La soluzione è quindi fare un flush manuale tramite la funzione `fflush`:

```
int main () {
    printf("The quick brown fox jumped over");
    fflush(stdout);

    execl("/bin/echo", "echo", "the", "lazy", "dogs", (char*)NULL);
    // if execl returns then an error occurred!
    perror("execl");
    return 1;
}
```

3.4 TERMINAZIONE

Un processo UNIX può terminare solo in 4 modi:

- chiamando `exit()`;
- chiamando `_exit()` (UNIX) oppure `_Exit()` (standard C), che sono chiamate di livello più basso rispetto alla `exit()`;
- ricevendo un segnale (li vedremo in seguito);
- per un crash del sistema (spegnimento forzato del PC, bug dell'OS, ecc).

Le tre funzioni `exit()`, `_exit()` e `_Exit()` sono molto simili, a partire dalla *signature*:

```
#include <unistd.h>

void _exit(
    int status  // exit status
);

void _Exit(
    int status  // exit status
```

```
);

void exit(
    int status  // exit status
);
```

Nessuna delle 3 funzioni ha un valore di ritorno, in quanto terminano il processo corrente. La `exit()` fa tutto quello che fa la `_exit()`, ma inoltre

- chiama la funzione `atexit()`, se esiste;
- esegue il flush dei buffer di I/O tramite `fflush` e `fclose`.

La `exit` viene chiamata automaticamente quando il programma termina con `return` dal `main`.

▸ FUNZIONE `atexit()` La funzione `atexit()` serve a *registrare* le azioni da compiere alla chiusura del programma. Ha la seguente *signature*:

```
#include <stdlib.h>

int atexit(
    void *function (void);
)
```

Se la `atexit` ha successo ritorna 0, altrimenti ritorna un valore diverso da 0 ma **NON** setta `errno`.

La funzione `function` deve essere una funzione che non prende argomenti e non restituisce alcun valore, il cui codice conterrà le operazioni da compiere alla chiusura del programma. Le tipiche operazioni che vengono inserite in questa funzione sono

- cancellazione di file temporanei;
- stampa di messaggi sull'esito della computazione;
- *pipes* (le vedremo più avanti);
- eccetera.

Inoltre possiamo registrare più di una funzione, tramite chiamate multiple alla `atexit`: alla terminazione del programma le funzioni registrate saranno chiamate in ordine **inverso**.

▸ FUNZIONE `_exit()` La funzione `_exit(status)` svolge le seguenti operazioni:

- termina il processo;
- chiude tutti i descrittori di file;
- libera lo spazio di indirizzamento;
- invia un segnale SEGCHILD al padre;
- salva il byte meno significativo di `status` nella tabella dei processi, in attesa che il padre lo accetti tramite le funzioni `wait` / `waitpid` (prossima sezione);
- i figli, diventati a questo punto orfani (*orphans*), vengono *adottati* da init, ovvero il loro PPID diventa uguale a 1.

3.5 ATTESA DEI PROCESSI FIGLI

Spesso vogliamo che dopo una `fork` il processo padre si metta in pausa e aspetti la terminazione del figlio. La funzione per realizzare questa funzionalità è `waitpid`:

```
#include <sys/types.h>
#include <sys/wait.h>

int waitpid(
    pid_t pid,      // pid of the child, or process group id
    int* statusp,   // pointer to the status, or NULL
    int options     // options
);
```

La funzione `waitpid` fa in modo che il processo corrente si metta in attesa fintanto che il processo con PID `pid` non è terminato. In particolare `waitpid` restituisce 0 oppure il PID del processo terminato, mentre se c'è un errore ritorna -1 e setta `errno`.

Il valore del parametro `pid` può essere di diversi tipi, e per ogni tipo si ottiene un comportamento diverso:

- se `pid` > 0 allora il processo corrente attende il figlio con PID `pid`;
- se `pid` $= -1$ attende un qualsiasi figlio (e come valore di ritorno restituisce il PID del figlio che ha cambiato stato);
- se `pid` $= 0$ attende un qualsiasi processo figlio nello stesso *process group*;
- se `pid` < -1 attende un qualsiasi processo figlio nel *process group* dato dal **valore assoluto** di `pid`.

Il puntatore a `status` serve a recuperare lo *status* di uscita, settato dal processo figlio tramite la `_exit` o tramite la `exit`. Inoltre `status` conterrà diverse altre informazioni, recuperabili con delle maschere particolari. Le più importanti sono:

- `WIFEXITED(status)` ritorna `true` se il processo figlio è terminato tramite una `exit`; in particolare se vale `WIFEXITED(status)`, allora lo stato di uscita si recupera con `WEXITSTATUS(status)`;
- `WIFSIGNALED(status)` ritorna `true` se il processo figlio è terminato tramite un segnale; in particolare se vale `WIFSIGNALED(status)` lo stato di uscita si recupera con `WTERMSIG(status)`.

Se almeno un figlio è già terminato e il suo stato non è ancora stato letto tramite una `wait*`, `waitpid` termina subito; in caso contrario mette il processo corrente (il padre) in attesa.

L'ultimo parametro (`options`) serve ad aggiungere alcuni flag, come ad esempio `WNOHANG`, che permette al processo padre di non mettersi in attesa se non c'è nessuno stato di un figlio subito disponibile. Se si vogliono inserire più flag vanno messi in *or bit-a-bit*.

4

Lezione 7

4.1 THREADS

Abbiamo visto che possiamo eseguire contemporaneamente più programmi separati sfruttando il meccanismo dei processi e le varie *system call* relative ad essi. In alcuni casi tuttavia vorremmo suddividere il flusso di esecuzione di un singolo programma in più parti separate ed indipendenti, anche se *interne* al programma scelto.

Un **thread** è un modo per eseguire parte di un programma indipendentemente e parallelamente al resto. Ogni thread ha un suo program counter: in questo modo può svolgere operazioni "parallelamente" al resto del programma.

Questo "parallelamente" può essere inteso in diversi modi: se abbiamo più processori (come nella stragrande maggioranza dei computer moderni) il Sistema Operativo può far eseguire più thread dello stesso processo a processori diversi nello stesso momento, con un'esecuzione davvero parallela.

Invece se il nostro computer ha un solo processore possiamo eseguire un singolo thread alla volta: il Sistema Operativo farà eseguire un pezzo di uno e un pezzo di un altro per dare l'impressione che siano tutti contemporanei, anche se sono in **esecuzione concorrente**.

Per far ciò sfrutteremo la libreria POSIX `pthread.h`, che contiene tutte le funzioni necessarie per la creazione e il funzionamento di thread. Attenzione però: la libreria `pthread.h` non è linkata di default dal compilatore GCC, quindi dobbiamo linkarla noi manualmente aggiungendo al comando l'opzione `-lpthread`.

4.2 CREAZIONE DI UN NUOVO THREAD

Per creare un nuovo thread relativo ad un determinato processo possiamo usare la funzione `pthread_create`:

```
#include <pthread.h>

int pthread_create(
    pthread_t* thread_id,       // new thread's ID
    const pthread_attr_t *attr, // attributes
    void* (*start_fcn) (void*), // starting function
    void* arg                   // args to the starting function
);
```

Esaminiamo ora gli argomenti e il valore di ritorno di questa funzione:

- `thread_id` è un puntatore ad un valore di tipo `pthread_t`: se `pthread_create` ha successo la funzione restituirà l'ID del nuovo thread all'interno della variabile `thread_id`;

- `attr` serve a specificare degli attributi per il nuovo thread: noi non useremo questa funzionalità e pertanto metteremo sempre questo argomento a `NULL`;
- `start_fcn` è un puntatore ad una funzione che prende un singolo argomento di tipo `void*` e ritorna un `void*`, che sarà l'*exit status*: questa funzione è la funzione da cui il thread inizierà la sua esecuzione;
- `arg` è l'argomento passato alla funzione `start_fcn`;
- il valore di ritorno di `pthread_create` è 0 se la funzione ha successo, mentre negli altri casi restituisce il codice di errore. In particolare `pthread_create` non setta `errno`, poiché il nuovo thread ha una sua copia della variabile e non può modificare l'`errno` del chiamante, quindi dobbiamo sfruttare il codice di errore per capire quale tipo di problema ha portato alla terminazione del thread.

Osserviamo quindi che, al contrario della `fork`, un thread non parte dall'istruzione che segue la sua creazione, ma da una specifica funzione di partenza: questo ci conferma ancora una volta che i thread hanno lo scopo di eseguire particolari funzionalità, e non devono rieseguire tutto il programma principale.

Facciamo un esempio: il codice è in Code-Snippet 1. Eseguendo il programma otterremo un risultato del genere:

```
$ ./threadtest
First thread: x = 1
Second thread: x = 2
First thread: x = 3
First thread: x = 4
Second thread: x = 5
...
```

I due thread non si alternano in un modo predefinito, ma vengono eseguiti e sospesi dallo *scheduler*, che opera al di fuori del nostro controllo.

4.3 ATTESA DI UN THREAD

Per attendere un thread, analogamente al caso dei processi, possiamo usare la funzione `pthread_join`:

```
#include <pthread.h>

int pthread_join(
    pthread_t thread_id,
    void** status_ptr
);
```

Come la `pthread_create` la funzione ritorna 0 in caso di successo, altrimenti ritorna il codice dell'errore. La semantica è la seguente: la funzione `pthread_join(tid, &status)`

- sospende il processo che la invoca finché il thread identificato da `tid` termina;
- se `&status` ≠ `NULL` salviamo in `status` lo stato di terminazione;
- quando un thread termina la memoria occupata da suo stack privato e la posizione nella tabella dei thread non vengono rilasciate fino a quando qualcuno non chiama la `pthread_join()` su quel thread: non chiamandola causiamo **memory leak**!

- per liberare le risorse senza la `pthread_join` possiamo usare la `pthread_detach`, ma dopo aver chiamato quest'ultima non possiamo più chiamare la `pthread_join`.

4.4 TERMINARE UN THREAD

Per terminare un thread possiamo:

- usare la `return` nella funzione iniziale del thread;
- usare la funzione `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(
    void* retval    // return value
);
```

Questa funzione non restituisce mai un valore, in quanto termina il thread corrente e restituisce lo status al thread che ha chiamato la funzione `pthread_join`.

Osserviamo che chiamando la funzione `exit` chiudiamo l'intero processo, e non solo il thread che la chiama.

4.5 INTERFERENZE TRA THREAD

Abbiamo già detto che non abbiamo alcun controllo sull'ordine di esecuzione dei thread: questo può portare a dei problemi di *concorrenza* tra thread chiamati **race conditions**.

Consideriamo il Code-Snippet 1. I due thread di quel programma operano sulla variabile condivisa `x` contemporaneamente, sommando 1 e poi stampando il risultato.

Tuttavia in alcuni casi sfortunati il comportamento potrebbe non essere quello che ci aspettiamo. Immaginiamo ad esempio la seguente sequenza di operazioni (chiamando i due thread T_1 e T_2):

1. `x` è inizialmente uguale a 5;
2. il thread T_1 legge `x`, ma prima di poterla incrementare viene *deschedulato*;
3. T_2 legge `x`, che è ancora uguale a 5, incrementa la variabile e la scrive (stampando 6), poi viene sospeso;
4. T_1 viene rieseguito e scrive anche lui 6 in `x`.

In questo caso abbiamo solamente perso un incremento, ma in problemi più seri possiamo causare grandi danni a causa di problemi di concorrenza di questo tipo.

Vogliamo quindi fare in modo che un thread abbia momentaneamente un accesso esclusivo a dei dati condivisi: in questo modo non possono esserci problemi di concorrenza tra thread.

Introduciamo quindi alcuni concetti fondamentali:

- si usa il termine **regione critica** per riferirci al frammento di codice che contiene la lettura/scrittura/modifica della risorsa condivisa;
- vogliamo fare in modo che l'accesso alla regione critica sia sempre **mutuamente esclusivo**: due thread diversi non possono accedere ad una regione critica contemporaneamente.

Il meccanismo base per risolvere il problema delle *race conditions* è il meccanismo delle **lock** (in italiano *lucchetti*): creiamo una variabile speciale, condivisa dai thread, che ha lo scopo di bloccare l'accesso ad una regione critica se qualche altro thread sta già usando la risorsa condivisa.

La libreria `pthread.h` usa il tipo `pthread_mutex_t` per rappresentare un **mutex** (che è un modo per implementare le lock) insieme alle seguenti due funzioni:

```
#include <pthread.h>

int pthread_mutex_lock(
    pthread_mutex_t *mutex  // mutex to lock
);

int pthread_mutex_unlock(
    pthread_mutex_t *mutex  // mutex to unlock
);
```

Le due funzioni servono ovviamente a richiedere l'uso di un lucchetto (tramite la `pthread_mutex_lock`) o a rilasciarlo (tramite la `pthread_mutex_unlock`) e ritornano 0 in caso di successo, il codice dell'errore altrimenti.

4.5.1 Uso dei mutex

Per usare correttamente i mutex bisogna seguire questo schema:

- si crea un mutex per ogni risorsa condivisa a cui vogliamo accedere in mutua esclusione;
- prima di accedere ai dati gestiti dal mutex `mutex_data` si chiama la funzione `pthread_mutex_lock(&mutex_data`
- alla fine della sezione critica si chiama `pthread_mutex_unlock(&mutex_data)`.

La `pthread_mutex_lock` controlla se il lucchetto è occupato da qualche altro thread: se è libero lo prende, lo setta e il thread è libero di entrare nella zona in mutua esclusione; invece se è occupato il thread si **sospende** fino a quando il thread che ha bloccato il mutex non lo rilascia.

▸ INIZIALIZZAZIONE Bisogna sempre inizializzare i mutex quando li dichiariamo! Se sono globali possiamo usare una macro:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

mentre se non sono globali dobbiamo usare la funzione

```
int pthread_mutex_init(
    pthread_mutex_t *mtx, // mutex to initialize
    const pthread_mutexattr_t *attr // attributes (NULL is default)
); // always returns 0
```

Noi useremo sempre la macro.

Il Code-Snippet 1 può dunque essere migliorato come si vede in Code-Snippet 2: abbiamo circondato la linea di codice che si occupa di interagire con `x` con il mutex. In questo modo otteniamo la mutua esclusione e impediamo le race conditions che erano presenti nella prima versione.

Osserviamo che abbiamo incluso nella regione critica *soltanto* il codice strettamente necessario: se avessimo incluso, ad esempio, anche `sleep(1)` avremmo rallentato il programma inavvertitamente. Infatti in tal caso mentre T_1 è in possesso del lock T_2 è bloccato, anche se T_1 è in `sleep`: abbiamo eliminato il parallelismo fra i thread e il programma viene eseguito esattamente come se fosse un singolo thread.

4.5.2 Versioni "safe" di lock/unlock

Inoltre come al solito le funzioni `pthread_mutex_lock` e `pthread_mutex_unlock` possono restituire degli errori che vanno sempre controllati: è buona pratica quindi creare delle funzioni ausiliarie per farlo automaticamente.

```
void safe_pthread_mutex_lock(pthread_mutex_t *mtx){
    int err;
    if( (err = pthread_mutex_lock(&mtx)) != 0) {
        errno = err;
        perror("lock");
        pthread_exit((void*) errno); // can be changed
    } else {
        printf("locked\n"); // can be commented when the program is finished
    }
}

void safe_pthread_mutex_unlock(pthread_mutex_t *mtx){
    int err;
    if( (err = pthread_mutex_unlock(&mtx)) != 0) {
        errno = err;
        perror("unlock");
        pthread_exit((void*) errno); // can be changed
    } else {
        printf("unlocked\n"); // can be commented when the program is finished
    }
}
```

4.5.3 Restringere l'area d'uso

A questo punto possiamo evitare la duplicazione del codice creando una funzione che incrementi il valore di `x` : in questo modo non serve gestire in due punti diversi il mutex ma basta chiamare la funzione appena definita.

```
static int atomic_incr_x(int incr){
    static int x = 0; // the shared variable
    static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
    int res;

    safe_pthread_mutex_lock(&mtx);
    res = (x += incr);
    safe_pthread_mutex_unlock(&mtx);

    return res;
}
```

Notiamo come abbiamo dovuto usare un'altra variabile `res` , in quanto altrimenti staremmo accedendo a x al di fuori dell'area delimitata dal mutex: se restituissimo il valore di `x` direttamente lo scheduler potrebbe toglierci il controllo del processore appena prima della `return` , dunque qualche altro thread potrebbe modificare x e il valore restituito sarebbe scorretto.

4.6 CONDITION VARIABLES

Supponiamo ora di voler scrivere un programma simile ai precedenti, soltanto che uno dei due thread deve iniziare ad incrementare la variabile condivisa x solo quando sono già stati fatti 10

incrementi.

Una possibile soluzione per la funzione del secondo thread sarebbe la seguente:

```
void* second_thread(void* arg){
    while(atomic_incr_x(0) < 10);

    while (true){
        printf("Second thread: x=%d\n", atomic_incr_x(1));
        sleep(1);
    }
}
```

Infatti chiamando `atomic_incr_x(0)` stiamo incrementando `x` di 0 e restituiamo il risultato dell'incremento, cioè il valore corrente di `x`. In questo modo possiamo controllare se è minore o maggiore di 10 e nel secondo caso possiamo iniziare ad operare effettivamente su `x`.

Tuttavia questo pattern di programmazione, chiamato *busy wait*, è assolutamente da evitare: mentre siamo in attesa prendiamo costantemente il controllo della mutex per leggere il valore di `x` ma senza dover effettivamente operare sulla variabile condivisa.

Per risolvere questo problema possiamo sfruttare le **condition variables**: ogni variabile di condizione rappresenta un evento relativo ad un dato condiviso e associato ad una mutex, come in questo caso il raggiungimento del valore 10 da parte della variabile `x`. Possiamo operare in due soli modi su una variabile di condizione: possiamo metterci in *attesa* che si verifichi un evento oppure possiamo *segnalare* che l'evento si è verificato.

Inizializzazione

Per inizializzare una variabile di condizione possiamo usare una macro fornitaci da `pthread.h`:

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Questa macro può essere usata solamente se la variabile `cond` è globale.

Segnalazione

```
#include <pthread.h>

int pthread_cond_signal(
    pthread_cond_t *cond, // condition variable
);
// 0 on success, error value otherwise
```

La chiamata a `pthread_cond_signal` prende un puntatore ad un oggetto di tipo `pthread_cond_t` (cioè ad una condition variable) e segnala agli altri thread che l'evento indicato dalla variabile `cond` si è verificato, svegliando **uno** dei thread in attesa. Se nessuno è in attesa la *signal* viene persa!

Attesa

```
#include <pthread.h>

int pthread_cond_wait(
    pthread_cond_t *cond, // condition variable
```

```
    pthread_mutex_t *mtx, // mutex linked to CV
);
// 0 on success, error value otherwise
```

La `pthread_cond_wait` viene usata da un thread per mettersi in attesa che una certa condizione si verifichi. In particolare questa funzione deve essere usata mentre il thread è in possesso del mutex relativo alla variabile condivisa: se il thread deve mettersi in attesa la `pthread_cond_wait` rilascia automaticamente il mutex e fa in modo di andare in attesa *passiva* (ovvero senza occupare il processore e il mutex). Tuttavia se qualcuno ha inviato il segnale relativo alla variabile di condizione *prima* che il thread si mettesse in attesa, questo thread non verrà mai svegliato da quella segnalazione: dobbiamo essere attenti alla progrettazione.

Un modo per risolvere questo problema è il seguente: la `pthread_cond_wait` viene messa all'interno di un ciclo `while` la cui guardia serve a controllare la condizione.

```
pthread_mutex_lock(&mtx);
while(condition is false)
    pthread_cond_wait(&cond, &mtx);
// now the condition is true!
pthread_mutex_unlock(&mtx);
```

In questo modo se la condizione è già realizzata quando stiamo per accedere alla sezione critica non è necessario mettersi in *wait*, e quindi non dobbiamo attendere in eterno un segnale che non arriverà mai.

Inoltre questo risolve un altro problema: se non mettessimo il `while` potremmo essere deschedulati appena dopo esserci svegliati dalla *wait*; prima di essere riattivati un altro thread potrebbe modificare la variabile condivisa e far sì che la condizione non sia più verificata, ma dato che ormai abbiamo superato la *wait* non avremmo più modo di controllare la condizione. Sfruttando il ciclo `while` invece se venissimo deschedulati in quel momento rientreremmo esattamente nel punto di controllare la condizione del `while`, e quindi proseguiremmo solo se la condizione è ancora vera.

Possiamo quindi realizzare il programma descritto all'inizio di questa sezione tramite le condition variables: il codice si trova nel Code-Snippet 3.

4.7 CANCELLAZIONE DI UN THREAD

Introduciamo ora un'ulteriore funzione utile per gestire i thread, chiamata `pthreaad_cancel`.

```
#include <pthread.h>

int pthread_cancel(
    pthread_t *tid, // thread id
);
// 0 on success, error value otherwise
```

Questa funzione fa terminare un thread appena raggiunge un *punto di cancellazione*, che in genere è una chiamata di funzione *safe*, nel senso che la funzione chiamata non è una mutex, malloc, o cose simili.

Tuttavia la cancellazione esplicita dei thread deve essere vista come un'ultima spiaggia: nella maggior parte dei casi va evitata in quanto termina brutalmente il thread.

Per renderla più *pulita* possiamo usare i **cleanup handlers**, che sono una versione specifica per i thread della funzione `atexit`: servono a chiamare delle funzioni specifiche alla terminazione di un thread, che sia per `pthread_exit` o `return` dalla funzione principale o per cancellazione.

```c
#include <pthread.h>
void pthread_cleanup_push(
    void (*handler) (void*), // cleanup function
    void *arg, // arguments to the handler
);
```

Come nel caso di `atexit` possiamo registrare più funzioni, e queste vengono messe *in pila* e quindi eseguite in ordine inverso di registrazione. Per rimuovere funzioni dalla pila possiamo usare la seguente funzione:

```c
void pthread_cleanup_pop(
    int execute, // do we want to execute the function?
);
```

A

Vari frammenti di codice

A.1 Short threads example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>

static int x; // shared variable

static void* myfun (void* arg){
    while (true) {
        printf("Second thread: x = %d\n", ++x);
        sleep(1);
    }
}

int main(){
    pthread_t tid;
    int err;

    if( (err = pthread_create(&tid, NULL, &myfun, NULL)) != 0 ) {
        // dealing with errors
        perror("thread");
        exit(EXIT_FAILURE);
    } else { // second thread has been created
        while (true){
            printf("First thread: x = %d\n", ++x);
            sleep(1);
        }
    }

    return 0;
}
```

Code-Snippet 1: Short threads example

A.2 Mutex example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>

static int x; // shared variable
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void* myfun (void* arg){
    while (true) {

        pthread_mutex_lock(&mtx);
        printf("Second thread: x = %d\n", ++x);
        pthread_mutex_unlock(&mtx);

        sleep(1);
    }
}

int main(){
    pthread_t tid;
    int err;

    if( (err = pthread_create(&tid, NULL, &myfun, NULL)) != 0 ) {
        // dealing with errors
        perror("thread");
        exit(EXIT_FAILURE);
    } else { // second thread has been created
        while (true){
            pthread_mutex_lock(&mtx);
            printf("First thread: x = %d\n", ++x);
            pthread_mutex_unlock(&mtx);

            sleep(1);
        }
    }

    return 0;
}
```

Code-Snippet 2: Mutex example

A.3 Condition Variables example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>

static int x; // shared variable
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static void* myfun (void* arg){
    pthraed_mutex_lock(&mtx);
    pthread_cond_wait(&cond, &mtx);
    pthread_mutex_unlock(&mtx);

    while (true) {
        atomic_incr_x(1);
        sleep(1);
    }
}

static int atomic_incr_x(int incr){
    static int x = 0;
    int res;

    pthread_mutex_lock(&mtx);
    res = (x += incr);
    pthread_mutex_unlock(&mtx);

    return res;
}

int main(){
    pthread_t tid;
    int err;

    if( (err = pthread_create(&tid, NULL, &myfun, NULL)) != 0 ) {
        // dealing with errors
        perror("thread");
        exit(EXIT_FAILURE);
    } else { // second thread has been created
        while (true){
            atomic_incr_x(1);
            sleep(1);
        }
    }

    return 0;
}
```

Code-Snippet 3: Condition Variables example