Laboratorio di Sistemi Operativi

Luca De Paulis

9 aprile 2021

Indice

INDICE i

1 LEZIONE 1 1
1.1 Funzioni con un numero variabile di argomenti 1

2 LEZIONE 2 3
2.1 Puntatori a funzione 3
2.2 Puntatori generici 4
2.3 Funzioni rientranti 5
2.4 Preprocessore 6

3 LEZIONE 6 8
3.1 Processi 8
3.2 Fork 8
3.3 Exec 9
3.4 Terminazione 11
3.5 Attesa dei processi figli 13

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`:

```
#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:

```
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*.