Corso di Laboratorio di Sistemi Operativi

Lezione 7

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Introduzione al linguaggio C

Il linguaggio C

- Il **C** è un linguaggio **imperativo** legato a Unix, adatto all'implementazione di compilatori e sistemi operativi.
- È stato progettato da D. Ritchie per il PDP-11 (all'inizio degli anni '70). Nel 1983 l'**ANSI** ne ha definito una versione standard **portabile** (ANSI C).
- A differenza dei linguaggi da cui ha tratto le idee fondamentali, ovvero, BCPL (M. Richards) e B (K. Thompson), è un linguaggio **tipato**.
- Il C è **compilato**; la compilazione è preceduta da una fase di *preprocessing* (sostituzione di macro, inclusione di file sorgenti ausiliari e compilazione condizionale).
- Il C è considerato un linguaggio ad *alto livello, ma non "troppo"* in quanto fornisce le primitive per manipolare numeri, caratteri ed indirizzi, ma non oggetti composti come liste, stringhe, vettori ecc.
- Il C è un linguaggio "piccolo": non fornisce direttamente nemmeno delle primitive di input/output. Per effettuare queste operazioni si deve ricorrere alla **Libreria Standard**. Si può pensare al C come al nucleo imperativo di Java più i **puntatori** e la gestione a **basso livello** di numeri, caratteri e indirizzi.

Perché imparare il C

- Linguaggi derivati dal C: C++, Objective C, C♯, ...
- Linguaggi con sintassi ispirata al C: Java, JavaScript, Perl, PHP, C shell di UNIX/Linux, ...
- Qualunque linguaggio deve fare i conti con l'hardware: avere una buona conoscenza del C aiuta a capire cosa sta succedendo dietro le quinte.
- Linguaggio cardine dei sistemi UNIX:
  - Le chiamate di sistema sono definite come funzioni C
  - Qualsiasi linguaggio quindi alla fine passa per il C per interfacciarsi con il sistema operativo
  - Questo vale anche su Windows
- Spesso è necessario scrivere in C parti delicate di software scritto in altri linguaggi meno efficienti (Python, Java, ecc...)

Utilizzi del linguaggio C (e C++)

- Programmazione di sistema, sistemi embedded.
- Software numerico/scientifico: MATLAB, NumPy, SciKit, ...
- Implementazione di compilatori, interpreti, librerie per altri linguaggi di programmazione (e.g., Python, Perl, PHP).
- *Lingua franca* per la comunicazione tra linguaggi diversi
- 3D graphics APIs: OpenGL, DirectX/Direct3D
- 3D game engines (in C++, in realtà): Ogre3D, Unreal Engine, NetImmerse Engine, Unity, Bullet, PhysX, ...
- Computazione su GPU: OpenCL, CUDA
- Computer Vision e elaborazione dati audio/video, es. OpenCV, OpenAL
- Qualunque ambito in cui siano richieste prestazioni, efficienza nell'utilizzo delle risorse e portabilità del codice.

Lo Standard ISO e le sue implementazioni

Il linguaggio C è definito da uno **standard internazionale ISO**.

- Quattro versioni sono state ratificate finora: ISO C89, **C99**, C11 e C17.
- Ogni versione aggiunge qualche funzionalità lasciando completa compatibilità all'indietro.
- Il produttore dell'hardware e/o del sistema operativo fornisce il proprio **compilatore** e/o la propria implementazione della **libreria standard**.
  - Linux: gcc, clang, icc, ...
  - Windows: MSVC, icc, gcc, ...
  - macOS: clang, gcc, icc, ...
- Esiste quindi un linguaggio, ma molte implementazioni diverse.
- Attenzione a non confondere il **compilatore** con **l'ambiente di sviluppo**, es. MSVC vs. Visual Studio, clang vs Xcode, ...

Struttura di un programma C

Consideriamo il programma C che stampa a video la stringa `ciao, mondo!` seguita da un avanzamento del cursore all'inizio della linea successiva:

```c
#include <stdio.h>

int main()
{
  printf("ciao, mondo!\n");

  return 0;
}
```

- La prima riga è una **direttiva al preprocessore** che dice di includere le funzioni per l'input/output della libreria standard prima di compilare il programma.

Struttura di un programma C

Consideriamo il programma C che stampa a video la stringa `ciao, mondo!` seguita da un avanzamento del cursore all'inizio della linea successiva:

```c
#include <stdio.h>

int main()
{
  printf("ciao, mondo!\n");

  return 0;
}
```

- Ogni programma C è composto da **dichiarazioni** di variabili, tipi, e funzioni (contenenti istruzioni).
- fra queste ne esiste una particolare, chiamata `main`, da cui **inizia l'esecuzione** e che quindi deve essere presente in ogni programma.
- Le due parantesi `()` vuote dopo il `main` significano che quest'ultimo non prende alcun parametro in input.
- La parola chiave `int` indica che `main` **restituisce** un numero intero.

Struttura di un programma C

Consideriamo il programma C che stampa a video la stringa `ciao, mondo!` seguita da un avanzamento del cursore all'inizio della linea successiva:

```c
#include <stdio.h>

int main()
{
  printf("ciao, mondo!\n");

  return 0;
}
```

- La funzione `printf` della libreria standard stampa a video (standard output) la stringa fornita come argomento.
- All'interno di quest'ultima la **sequenza di escape** `\n` specifica il carattere speciale di "avanzamento all'inizio della linea successiva" o "newline".

Struttura di un programma C

Consideriamo il programma C che stampa a video la stringa `ciao, mondo!` seguita da un avanzamento del cursore all'inizio della linea successiva:

```c
#include <stdio.h>

int main()
{
  printf("ciao, mondo!\n");

  return 0;
}
```

- L'istruzione `return 0;` restituisce il numero zero, e termina l'esecuzione della funzione (e quindi del programma).
- Zero significa che il programma è andato a buon fine, come per convenzione nel mondo Unix.

Compilazione di programmi C

In ambiente Linux e macOS esistono due compilatori C **open source** e liberamente disponibili: GCC (GNU C Compiler) e Clang.

- Funzionano entrambi su Linux, macOS, Windows, iOS, Android, quasi tutti gli UNIX e tante piattaforme embedded.
- Supportano entrambi tutte le versioni di ISO C (e C++).
- GCC è il compilatore di "default" su sistemi Linux (e Android).
- Clang è il compilatore ufficiale su macOS (e iOS).

Compilazione di programmi C

In ambiente Linux e macOS esistono due compilatori C **open source** e liberamente disponibili: GCC (GNU C Compiler) e Clang.

- L'uso di uno o dell'altro è equivalente:
  - piccole differenze nell'interpretazione degli standard;
  - opzioni da riga di comando uguali o molto simili;
  - nella maggioranza dei casi completamente intercambiabili.
- I nostri esempi useranno Clang perché:
  - i messaggi di errore sono più chiari e comprensibili;
  - compila di default in modalità C11.

Compilazione di programmi C

Invocare il compilatore

Per compilare un programma, dopo averlo salvato in un file, ad esempio `programma.c`, si invoca il compilatore:

```
$ clang programma.c
```

Se il programma è sintatticamente corretto, il compilatore produrrà un file **eseguibile** chiamato `a.out`, che può essere eseguito:

```
$ ./a.out
Ciao mondo!
```

Compilazione di programmi C

Invocare il compilatore

L'opzione `-o` permette di dare un nome diverso al file di output:

```
$ clang programma.c -o programma
$ ./programma
Ciao mondo!
```

I tipi base del C

Tipo	Significato
`char`	carattere (un singolo byte)
`short int`	intero corto
`int`	intero
`long int`	intero lungo
`float`	floating-point a precisione singola
`double`	floating-point a precisione doppia
`unsigned int`	intero senza segno (anche `long` o `short`)

In C esistono due tipi di **conversioni di tipo**:

1. **Promozioni**: conversioni automatiche

   `char` $\rightsquigarrow$ `short` $\rightsquigarrow$ `int` $\rightsquigarrow$ `long` $\rightsquigarrow$ `float` $\rightsquigarrow$ `double`

2. **Cast**: conversione esplicita (nel verso opposto); per esempio:

   ```
   x=(int)5.0;
   ```

Sintassi di base

La sintassi di base è molto familiare per chi conosce già, ad esempio, il Java:

- Dichiarazione di variabili (`int x = 0;`)
- Costrutti di controllo e cicli (`if`, `for`, `while`)
- Espressioni e operatori:
  - Operatori aritmetici (`+`, `-`, `*`, `/`, `%`, ecc...)
  - Operatori logici (`&&`, `||`, `!`, ecc...)
  - Precedenza degli operatori
- Costanti numeriche (`42`, `3.14`), stringhe (`"Hello world"`), caratteri (`'A'`), ecc...
- **Attenzione**: Le somiglianze si limitano alla sintassi di base. In tutto il resto i due linguaggi sono profondamente diversi.

Espressioni booleane

La prima differenza importante rispetto al C++ e al Java è che in C *non esiste* il tipo `bool`.

- Il costrutto `if`, e i cicli `for` e `while`, si aspettano delle espressioni di tipo intero nelle proprie condizioni di controllo.
- Il valore **zero** viene interpretato come **falso**.
- Qualsiasi valore diverso da zero viene interpretato come **vero**.

```c
#include <stdio.h>

int main() {

  int x = 0;

  if(x)
    printf("x e' falso\n");
  else
    printf("x e' vero\n");

  return 0;
}
```

Un esempio di programma C

Il seguente programma stampa la tabella Fahrenheit-Celsius per l'intervallo di valori Fahrenheit da 0 a 300:

```c
#include <stdio.h> // Funzioni di I/O dalla libreria standard

int main()
{
  float fahr = 0; // dichiarazione di una variabile di tipo float

  printf("Tabella Fahrenheit-Celsius:\n");

  while(fahr <= 300) // ciclo while
  {
      float celsius = (5.0 / 9.0) * (fahr - 32.0);
      printf("%3.0f °F -> %6.1f °C\n", fahr, celsius);
      fahr = fahr + 20;
  }

  return 0;
}
```

La funzione `printf`

L'istruzione

```
printf("%3.0f °F -> %6.1f °C\n", fahr, celsius);
```

chiama la funzione `printf`, comunemente usata per stampare messaggi sullo standard output nei programmi C.

- Il primo argomento è una stringa di caratteri da stampare (`"%3.0f °F -> %6.1f °C\n"`) in cui ogni occorrenza del simbolo `%` indica il punto in cui devono essere sostituiti, nell'ordine, il 2°, 3°, ... argomento
- I caratteri successivi ad ogni `%` indicano il **tipo** dell'argomento e il formato in cui deve essere stampato.
- Ad esempio, `%3.0f` indica che l'argomento deve essere di tipo `float` e che devono essere stampati almeno 3 caratteri e nessun carattere per la parte decimale.

Sequenze di escape

All'interno delle stringhe, le **sequenze di escape** permettono di specificare caratteri particolari:

Sequenza	Significato
`\n`	Newline (a capo)
`\t`	Tab
`\\`	Singola backslash `'\'`
`\"`	Doppi apici

Attenzione

Non vanno confuse le sequenze di escape, che fanno parte della sintassi del **linguaggio**, con gli specificatori di formato di **printf**, che invece fanno parte del modo in cui una singola funzione interpreta il proprio input.

Il preprocessore

Ogni file C, prima della compilazione, viene **preprocessato**.

- Il preprocessore trasforma il codice interpretando delle direttive.
- Le direttive sono righe di codice che cominciano con un cancelletto #, seguito dal nome della direttiva vera e propria
- Il compilatore vede solo il risultato del preprocessing.

Alcune utili direttive del preprocessore

```
#include <nomefile.h>
```

Include testualmente il file `nomefile.h`. Usato per includere i **file di intestazione** della libreria standard.

```
#define NOME valore
```

Definisce **costanti simboliche**.

- Dopo tale direttiva, il preprocessore sostituirà testualmente ogni occorrenza di `NOME` con `valore`.
- La sostituzione non avviene all'interno di stringhe o all'interno di altri identificatori.
- Le costanti simboliche **non sono** variabili; infatti per distinguerle vengono convenzionalmente scritte in maiuscolo.

Un altro programma per la conversione Fahrenheit-Celsius

```c
#include <stdio.h>

#define LOWER   0
#define UPPER 300
#define STEP   20

int main()
{
  for(float fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
    printf("%3.0f °F -> %6.1f °C\n", fahr, (5.0/9.0) * (fahr - 32));

  return 0;
}
```

Esempi

Esempio I

I/O di caratteri

I seguenti programmi implementano una versione semplificata del comando `cat`: leggono caratteri dallo standard input e li ripetono tali e quali sullo standard output, finché l'input non termina (EOF, ovvero End Of File).

Con ciclo `while`:

```
#include <stdio.h>

int main()
{
  int c = getchar();

  while(c != EOF)
  {
    putchar(c);
    c = getchar();
  }

  return 0;
}
```

Con ciclo `for`:

```
#include <stdio.h>

int main()
{
  for(int c = getchar(); c != EOF; c = getchar())
  {
      putchar(c);
  }

  return 0;
}
```

Nota: la shortcut da tastiera per immettere il carattere speciale di EOF sul terminale è `Ctrl-D`.

Esempio II

Conteggio di caratteri

I seguenti programmi implementano la funzionalità del comando Unix `wc -c`:

Con ciclo `while`:

```
#include <stdio.h>

int main()
{
  long nc = 0;

  while(getchar() != EOF)
  {
    ++nc;
  }

  printf("%ld\n",nc);

  return 0;
}
```

Con ciclo `for`:

```
#include <stdio.h>

int main()
{
  long nc = 0;
  for(; getchar() != EOF; ++nc);

  printf("%ld\n",nc);

  return 0;
}
```

Esempio III

Conteggio di linee

Il seguente programma implementa la funzionalità del comando Unix `wc -l`:

```
#include <stdio.h>

int main()
{
  long nc = 0;

  for(int c = getchar(); c != EOF; c = getchar())
    if(c == '\n')
      ++nc;

  printf("%ld\n",nc);

  return 0;
}
```

Esercizi

Esercizi

1. Scrivere un programma C che stampi il valore della costante simbolica `EOF`.
2. Scrivere un programma C che conti il numero di spazi, tab e newline (*whitespace characters*) presenti nei caratteri immessi sullo standard input.
3. Scrivere un programma C che stampi un istogramma orizzontale (utilizzando il carattere `'-'`) raffigurante le lunghezze delle parole (delimitate da *whitespace characters*) immesse sullo standard input (parola per parola).
4. Scrivere un programma C che conti il numero di parole immesse sullo standard input (si considerino come delimitatori di parola i *whitespace characters*).

Corso di Laboratorio di Sistemi Operativi

Lezione 8

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Funzioni

Le funzioni in C

I programmi C sono costituiti da **dichiarazioni** di variabili, tipi, e **funzioni**, e da **definizioni** di variabili e funzioni.

La **dichiarazione** di una funzione (chiamato anche **prototipo**) ha la seguente sintassi:

```
tipo_ritornato nome_funzione(lista_parametri);
```

Esempio:

```
int factorial(int n);
```

Le funzioni in C

I programmi C sono costituiti da **dichiarazioni** di variabili, tipi, e **funzioni**, e da **definizioni** di variabili e funzioni.

È possibile scrivere funzioni che non ritornano nulla, dichiarando il tipo `void`. Esempio:

```
void putchar(int c);
```

Le funzioni in C

I programmi C sono costituiti da **dichiarazioni** di variabili, tipi, e **funzioni**, e da **definizioni** di variabili e funzioni.

La **definizione** di una funzione implementa effettivamente una funzione dichiarata precedentemente:

```
tipo_ritornato nome_funzione(lista_parametri)
{
  istruzione;
  istruzione;

  ...

  return valore;
}
```

Le funzioni in C

I programmi C sono costituiti da **dichiarazioni** di variabili, tipi, e **funzioni**, e da **definizioni** di variabili e funzioni.

La **definizione** di una funzione implementa effettivamente una funzione dichiarata precedentemente. Esempio:

```
int factorial(int n) {
  if(n == 0)
    return 1;
  else
    return n*factorial(n - 1);
}
```

Esempio

Il seguente programma è costituito da una funzione `int power(int m, int n)` che calcola la potenza m^n, e dalla funzione principale `main` che la utilizza.

```c
#include <stdio.h>

int power(int, int); // dichiarazione

int main()
{
   for(int i = 0; i < 10; ++i)
      printf("%d %3d %6d\n", i, power(2,i), power(-3,i));

   return 0;
}

int power(int m, int n) // definizione
{
   int p = 1;
   for (int i = 0; i < n; ++i)
      p = p * m;
   return p; // restituisce il valore di p al chiamante
}
```

Alcuni dettagli

Alcune cose importanti da tenere a mente:

- La dichiarazione, o la definizione, di una funzione deve **precedere** il punto in cui la funzione viene **chiamata**.
- La dichiarazione è necessaria se la funzione viene definita e chiamata in file diversi.
- In assenza della dichiarazione, il compilatore non può controllare che la funzione venga chiamata con tipi corretti.
- Se una funzione indica un tipo di ritorno, **deve** contenere tutte le istruzioni `return` necessarie.

Attenzione: Il compilatore **non** controlla questi obblighi.
In caso di violazioni il programma può comportarsi in modo errato durante l'**esecuzione**, in modo imprevedibile.

Passaggio per valore

- In C tutti gli argomenti delle funzioni sono passati **per valore**, ovvero le funzioni lavorano su copie dei parametri e non modificano i parametri passati dal chiamante.
- Il passaggio **per riferimento**, presente in altri linguaggi, non esiste in C. Si può simulare per mezzo dei **puntatori**, ovvero passando l'indirizzo in memoria dell'argomento.
  Sarà argomento della prossima lezione.

Passaggio per valore

Esempio

```c
#include <stdio.h>

void fake_swap(int x, int y) {
  int z = x;
  x = y;
  y = z;
}

int main()
{
  int x = 42, y = 0;

  fake_swap(x,y);

  printf("x = %d, y = %d\n", x, y);

  return 0;
}
```

Variabili e scope

Lo **scope** di una variabile è la parte di codice in cui essa è **visibile**.

- Le variabili dichiarate all'**interno** delle funzioni sono **locali** a queste ultime e sono dette **automatiche**, in quanto sono create al momento della chiamata della funzione e cessano di esistere quando questa termina.
- Lo stesso discorso vale per le variabili dichiarate in blocchi interni (es. in un ciclo `for`).
- È buona norma **inizializzare** ogni variabile al momento della dicharazione (es. `int x = 0;` invece di `int x;`), perché altrimenti il valore iniziale della variabile non è definito a priori.
- È possibile definire variabili **globali** al di fuori delle funzioni, che saranno accessibili a tutto il programma. L'uso di variabili globali è sconsigliabile dal punto di vista del design del codice.

Array

Array

Un array è un tipo di dato **aggregato** formato da un numero **fissato** di elementi dello stesso tipo.

Per dichiarare un array:

```
tipo nome[N] = { valori };
```

Esempio:

```
int voti[12] = { 18, 30, 27, 20, 22, 28, 25, 25, 30, 27, 18, 26 };
```

Array

Un array è un tipo di dato **aggregato** formato da un numero **fissato** di elementi dello stesso tipo.

È possibile inizializzare tutti i valori a zero in un colpo:

```
int statistiche[300] = { };
```

Se la dimensione non viene specificata, si tratterà del numero di elementi elencati nell'inizializzazione:

```
int voti[] = { 18, 30, 27, 20, 22, 28, 25, 25, 30, 27, 18, 26 };
```

Esempio I

Media pesata dei valori contenuti in un array

```c
#include <stdio.h>

#define N_ESAMI 12

int main() {
  int voti[N_ESAMI] = { 18, 30, 27, 20, 22, 28, 25, 25, 30, 27, 18, 26 };
  int crediti[N_ESAMI] = { 12, 12, 12, 12, 6, 6, 6, 6, 6, 6, 6, 6 };

  int somma = 0, totale = 0;

  for(int i = 0; i < N_ESAMI; ++i) {
    somma += voti[i] * crediti[i];
    totale += crediti[i];
  }

  float media = (float) somma / totale; // Notare il cast esplicito a float

  printf("La mia media pesata e': %2.2f\n", media);

  return 0;
}
```

Esempio II

Tabella dei valori della funzione $\sin x$

```c
#include <stdio.h>
#include <math.h>

#define N_CAMPIONI 10

int main()
{
  float valori[N_CAMPIONI] = { 0.0 };

  float step = 2 * M_PI / N_CAMPIONI;

  for(int i = 0; i < N_CAMPIONI; ++i) {
    valori[i] = sin(i * step);
  }

  for(int i = 0; i < N_CAMPIONI; ++i) {
    printf("sin(%1.3f) = %1.3f\n", i * step, valori[i]);
  }

  return 0;
}
```

Array e problemi di sicurezza

In C gli array, come il resto, vengono gestiti a **basso livello**:

- Il programma fa esattamente ciò che scrivete, né più né meno.
- In particolare, il linguaggio non prevede controlli sulla **validità degli indici** usati per accedere agli array.
- Accedere ad un array con un indice troppo grande (*out of bounds*) comporta seri problemi.
  - Nel migliore dei casi, il sistema operativo uccide il processo.
  - Nella maggioranza dei casi, vengono scritte altre posizioni di memoria non prevedibili a priori.
  - In ogni caso, il comportamento del programma è **indefinito**.

Attenzione: Errori di programmazione che causano la scrittura oltre il termine di un array sono la causa del 90% dei bug e dei problemi di sicurezza del software.

Array e problemi di sicurezza

Esempio

```c
#include <stdio.h>

int main() {
  int risposta = 42;
  int valori[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  // Azzeriamo l'array
  for(int i = 0; i <= 10; ++i) {
    valori[i] = 0;
  }

  printf("La risposta e': %d\n", risposta);

  return 0;
}
```

Qual è l'output di questo programma?

Esercizi

Esercizi

1. Ripetere l'esercizio 2 o 3 della Lezione 7, definendo però una funzione `is_whitespace()` per controllare se un carattere è di spazio bianco oppure no.
   - Nota: non esiste il tipo `bool`, a differenza di C++ o Java.
2. Definire una funzione `int lg(int n)` che trovi il massimo numero m tale che $10^m \leq n$ (ovvero la parte intera di $\log_{10} n$).
3. Scrivere un programma che ordini un array di numeri interi, utilizzando un algoritmo di ordinamento visto a lezione di Algoritmi e Strutture Dati (es. bubble sort o insertion sort).
   - Nota: In attesa di vedere i puntatori (necessari per passare array come argomenti a funzioni), ci si limiti ad ordinare un array dichiarato e inizializzato nel medesimo blocco.

Attenzione: Fornire sempre la dichiarazione di ogni funzione.

Corso di Laboratorio di Sistemi Operativi

Lezione 9

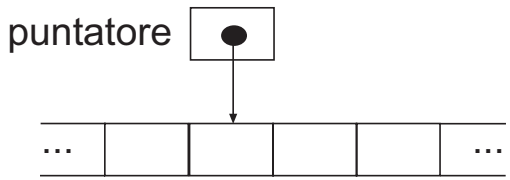
Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Puntatori

I puntatori

Un **puntatore** è una variabile che contiene l'**indirizzo** di un'altra variabile.



L'uso e la manipolazione dei puntatori sono ciò che distingue maggiormente il C dagli altri linguaggi.

I puntatori

La dichiarazione di un puntatore avviene in modo simile a quella di un'altra variabile:

```
int *p = valore iniziale;
```

Il tipo `int *` indica un puntatore a variabili di tipo `int`.

Il puntatore va inizializzato con l'**indirizzo** di una variabile di tipo `int`, oppure con il valore speciale `NULL`.

```
int x = 42;

int *p = &x;   // ora p punta ad x
int *q = NULL; // q non punta a nulla
```

Uso dei puntatori

L'uso dei puntatori gira attorno a due **operatori unari**.

- L'operatore *address of* (&) ottiene un puntatore ad una variabile:

```
int x = 42;
int *p = &x; // ora p punta ad x
```

- L'operatore di *dereferenziazione* (*), permette di accedere alla variabile puntata da un puntatore:

```
int x = 42;
int *p = &x;

printf("%d\n", *p); // stampa 42
```

Uso dei puntatori

Esempio: funzione swap()

Versione non funzionante:

```c
#include <stdio.h>

void fake_swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main()
{
  int x = 42, y = 0;

  fake_swap(x,y);

  printf("x: %d, y: %d\n", x, y);

  return 0;
}
```

Versione funzionante:

```c
#include <stdio.h>

void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main()
{
  int x = 42, y = 0;

  swap(&x, &y);

  printf("x: %d, y: %d\n", x, y);

  return 0;
}
```

Uso dei puntatori

La funzione `scanf()`

La funzione `scanf()` è la controparte di `printf()` per **leggere** valori dallo standard input.

```c
#include <stdio.h>

int main()
{
  int x = 0;

  printf("Inserisci un numero intero: ");
  scanf("%d", &x);

  printf("Il doppio di %d e' %d\n", x, x * 2);

  return 0;
}
```

Uso dei puntatori

La funzione `scanf()`

L'uso di `scanf()` è simile a quello di `printf()`:

```
int x = 0;
scanf("%d", &x);
```

- Il primo argomento è una striga contenente la **specifica di formato** dei dati in input, con sintassi simile a `printf()`
  - `"%d"` per numeri interi, `"%f"` per numeri di tipo `float`, ecc...
- Gli argomenti seguenti sono invece **puntatori** alle variabili dove andranno scritti i valori letti in input.
- La funzione ritorna il numero di elementi letti con successo, oppure `EOF` se l'input è terminato prima di leggere qualcosa.
- Maggiori dettagli alla lezione sulla manipolazione di stringhe.

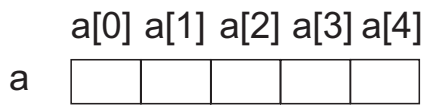
Puntatori e array

Puntatori e array

I puntatori sono lo strumento principale per la manipolazione degli array in C.

Supponiamo di dichiarare un array come segue:

```
int a[5] = { 0, 1, 2, 3, 4 };
```

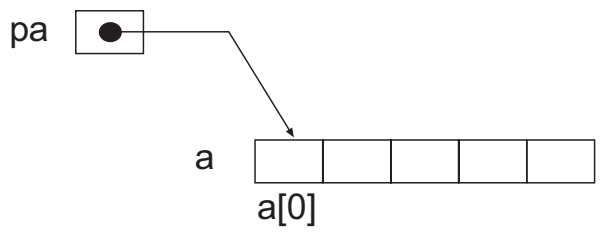


Puntatori e array

I puntatori sono lo strumento principale per la manipolazione degli array in C.

Ora, estraiamo un puntatore al suo primo elemento:

```
int *pa = &a[0];
printf("%d\n", *pa); // stampa zero
```



Per ottenere il puntatore al primo elemento di un array si può anche menzionare semplicemente il nome dell'array:

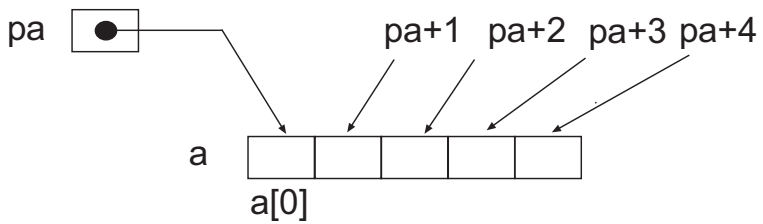
```
int *pa = a; // equivalente ad &a[0]
```

Puntatori e array

I puntatori sono lo strumento principale per la manipolazione degli array in C.

Incrementando il puntatore è possibile ottenere puntatori agli elementi successivi:

```
pa = pa + 1;
printf("%d\n", *pa); // stampa 1
```



Puntatori e array

Esempio

```c
#include <stdio.h>

#define SIZE 10

int main() {
  int array[SIZE] = { };
  int val = 0;

  printf("Inserisci un valore: ");
  scanf("%d", &val);

  for(int *p = array; p < array + SIZE; ++p) {
    *p = val;
  }

  for(int i = 0; i < SIZE; ++i) {
    printf("array[%d] = %d\n", i, array[i]);
  }

  return 0;
}
```

Puntatori e array

Esempio 2

```c
#include <stdio.h>
#define SIZE 10

void fill(int *, int, int);

int main() {
  int array[SIZE] = { }, val = 0;

  printf("Inserisci un valore: ");
  scanf("%d", &val);

  fill(array, SIZE, val);
  for(int i = 0; i < SIZE; ++i)
    printf("array[%d] = %d\n", i, array[i]);

  return 0;
}

void fill(int *begin, int size, int value) {
  for(int *p = begin; p < begin + size; ++p)
    *p = value;
}
```

Puntatori e array

Passaggio di array a funzioni

```
void fill(int *begin, int size, int value) {
  for(int *p = begin; p < begin + size; ++p)
    *p = value;
}
```

I puntatori sono l'unico modo per "passare" array alle funzioni:

- Passare direttamente l'array non è possibile.
- Invece, per scrivere una funzione che operi su un array di input è necessario passare:
  - Un puntatore al primo elemento
  - Un numero intero che indichi il numero di elementi dell'array

In ogni caso, è sempre necessario comunicare la lunghezza dell'array a chi ci opera.

Puntatori e array

Passaggio di array a funzioni

Attenzione: il C supporta anche la seguente sintassi nella dichiarazione delle funzioni:

```
int func(int array[42]);
```

che però è obsoleta e perfettamente equivalente a:

```
int func(int *array);
```

Quindi, l'uso di questa sintassi è **fuorviante**:

- Dà l'idea che la dimensione dell'array sia nota, mentre invece viene **ignorata**.
- L'array non viene passato per valore come ci si aspetterebbe per coerenza con variabili di altro tipo.

Puntatori e array

Aritmetica sui puntatori

```
void fill(int *begin, int size, int value) {
  for(int *p = begin; p < begin + size; ++p)
    *p = value;
}
```

L'aritmetica sui puntatori consente di **navigare** all'interno di un array usandoli come cursori da poter spostare a piacimento.

- Aggiungere un numero intero `i` ad un puntatore `p` fa avanzare il puntatore di `i * sizeof(T)`, dove:
  - `T` è il tipo puntato (es. `int`)
  - `sizeof(T)` è la **dimensione** in byte del tipo `T` (es. 4 byte)
- Ad esempio, con due puntatori `float *a;` e `double *b;`
  - `a + 1` punta 4 byte (32bit) più avanti rispetto ad `a`.
  - `b + 1` punta 8 byte (64bit) più avanti rispetto ad `b`.

Puntatori e array

Aritmetica sui puntatori

```
void fill(int *begin, int size, int value) {
  for(int *p = begin; p < begin + size; ++p)
    *p = value;
}
```

L'aritmetica sui puntatori consente di **navigare** all'interno di un array usandoli come cursori da poter spostare a piacimento.

- Non a caso, è quanto basta per avanzare di i posizioni all'interno di un array.
- Di conseguenza, `*(a + i)` equivale ad `a[i]`.
- In effetti, la sintassi a parentesi quadre si applica a puntatori:

```
int a[3] = { 40, 41, 42 };
int *pa = a;

printf("%d %d\n", pa[2], *(pa + 2)); // stampa "42 42"
```

Puntatori e array

Avvertenze

In C, il nome di un array può essere usato per nominare il puntatore al primo elemento dell'array stesso.

Attenzione: ciò genera spesso molta confusione, e soprattutto su internet si trovano molte affermazioni non corrette:

- Puntatori e array in C sono la stessa cosa. **Falso**.
- Un array **è** un puntatore. **Falso**.
- Dichiarare un puntatore ha qualcosa a che fare con l'allocazione di memoria. **Falso**.

Esercizi

Esercizi

1. Scrivere un programma che legga dallo standard input dei numeri e ne stampi la somma totale.
2. Scrivere le seguenti funzioni che manipolino un array passato come argomento (nei modi visti):
   - Una funzione `reverse()` per invertire l'ordine degli elementi in un array.
   - Una funzione `sort()` per ordinare un array di numeri interi (riutilizzare quanto scritto per l'esercizio 3 della Lezione 8).
   - Scrivere una funzione `qsort()` che implementi un algoritmo di ordinamento in maniera ricorsiva.
3. Scrivete tre programmi che utilizzino le funzioni scritte qui sopra per invertire/ordinare/ecc... una sequenza di numeri interi letti da standard input.

Corso di Laboratorio di Sistemi Operativi

Lezione 10

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Manipolazione di stringhe

Gestione delle stringhe in C

Seguendo la sua impostazione a basso livello, il C non fornisce un tipo di dato primitivo per rappresentare stringhe di caratteri.

Array di caratteri

Per rappresentare una stringa ci si affida ad **array di caratteri**.

```
char message[] = "Ciao";
```

Il codice qui sopra è equivalente al seguente:

```
char message[] = { 'C', 'i', 'a', 'o', 0 };
```

- Le stringhe costanti (`"Ciao"`) sono delle scorciatoie per dichiarare array costanti contenenti i caratteri della stringa.
- Per **convenzione**, l'ultimo elemento dell'array è un carattere **nullo**, che segnala la fine della stringa.
- Ciò permette di comunicare la lunghezza della stringa senza utilizzare una variabile aggiuntiva.

Array di caratteri

Esempio

Con scorrimento indicizzato:

```c
#include <stdio.h>

int main() {
  char msg[] = "Ciao";

  printf("Stringa: \"%s\" \n", msg);
  printf("Caratteri: ");

  for(int i = 0; msg[i]; ++i) {
    printf("'%c' ", msg[i]);
  }
  putchar('\n');
  return 0;
}
```

Con scorrimento a puntatori:

```c
#include <stdio.h>

int main() {
  char msg[] = "Ciao";

  printf("Stringa: \"%s\" \n", msg);
  printf("Caratteri: ");

  for(char *p = msg; *p; ++p) {
    printf("'%c' ", *p);
  }
  putchar('\n');
  return 0;
}
```

Funzioni standard per le stringhe

La libreria standard contiene molte funzioni utili a manipolare stringhe, fornite dall'header `<string.h>`:

- Concatenazione di due stringhe, copia di una stringa in un altro array, ...
- Confrontare due stringhe
- Estrarre la lunghezza di una stringa, cercare caratteri o sottostringhe, separare una stringa in base a delimitatori, ...

Tutte queste operazioni vanno eseguite con **attenzione**:

- Il risultato deve sempre avere il terminatore nullo nella posizione corretta.
- Non si deve scrivere mai oltre la fine degli array contenenti le stringhe.

Funzioni standard per le stringhe

Lunghezza di una stringa

La funzione

```
int strlen(char *s);
```

restituisce la lunghezza di una stringa.

Possibile implementazione:

```
int strlen(char *s) {
  int n = 0;
  for(char *p = s; *p; ++p) {
    ++n;
  }
  return n;
}
```

Funzioni standard per le stringhe

Confrontare due stringhe

La funzione

```
int strncmp(char *s1, char *s2, unsigned len);
```

confronta lessicograficamente `s1` e `s2`.

- Restituisce **`0`** se sono **uguali**, `-1` o `1` altrimenti.
- Notare il terzo argomento: serve ad evitare che il confronto esca oltre i limiti di uno dei due array se per errore mancasse uno dei terminatori nulli.
- Esiste la versione "meno sicura", `strcmp`, senza terzo argomento.

Funzioni standard per le stringhe

Esempio: utilizzo di `strncmp()`

```c
#include <stdio.h>
#include <string.h>

#define MAXINPUT 100

int main() {
  char input[MAXINPUT] = "";

  do {
    printf("Password: ");
    scanf("%99s", input);
  } while(strncmp(input, "passw0rd", MAXINPUT) != 0);

  printf("Il segreto e': 42\n");

  return 0;
}
```

Funzioni standard per le stringhe

Copia di stringhe

La funzione

```
char *strncpy(char *dest, char *source, unsigned len);
```

copia al massimo `len` caratteri dalla stringa `source` nella stringa `dest`.

Possibile implementazione:

```
char *strncpy(char *dest, char *source, unsigned len)
{
  int i = 0;
  while(i < len && source[i]) {
    dest[i] = source[i];
    ++i;
  }

  if(i < len)
    dest[i] = 0;

  return dest;
}
```

Funzioni standard per le stringhe

Concatenazione di stringhe

La funzione

```
char *strncat(char *dest, char *source, unsigned len);
```

concatena al massimo `len` caratteri dalla stringa `source` in fondo a `dest`.

Possibile implementazione:

```
char *strncat(char *dest, char *source, unsigned len)
{
  int destlen = strlen(dest);

  strncpy(dest + destlen, source, len);

  return dest;
}
```

Funzioni standard per le stringhe

Esempio: uso di `strncpy` e `strncat`

```c
#include <stdio.h>
#include <string.h>

#define MAXLEN 20

int main() {
  char msg[MAXLEN] = "";
  char msg1[MAXLEN] = "Ciao, ";
  char msg2[MAXLEN] = "mondo!";

  strncpy(msg, msg1, MAXLEN);
  strncat(msg, msg2, MAXLEN-strlen(msg)-1);

  printf("%s\n", msg);

  return 0;
}
```

Funzioni standard per le stringhe

Altre funzioni

È consigliabile prendere dimestichezza con la maggior parte delle funzioni esposte da `<string.h>`.

```
man string
```

Avvertenze

Attenzione: Per via della conversione automatica da `T[n]` a `T*`, il seguente codice compila:

```
char *stringa = "Ciao mondo";
```

Tuttavia, al 99% dei casi non fa quello che intendete.

- Non alloca un array per contenere la stringa `"Ciao mondo"`.
- La stringa `"Ciao mondo"` è già presente in memoria, nel **testo** del programma.
- Il puntatore viene inizializzato a puntare alla lettera `'C'` nel testo del programma.
- Tale zona di memoria è **a sola lettura**, e quindi manipolare tale stringa potrebbe portare a undefined behavior.
  - Il tipo dei *literal* dovrebbe essere `const char[n]`, in modo da convertirsi solo in `const char *`.
    Non è così per retrocompatibilità.

Avvertenze

Attenzione: Per via della conversione automatica da `T[n]` a `T*`, il seguente codice compila:

```
char *stringa = "Ciao mondo";
```

Esempio di codice **errato**:

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

int main() {
  char *ciao = "Ciao mondo!";

  strncpy(ciao, "Hello world!", 11);

  printf("%s\n", ciao);

  return 0;
}
```

Argomenti da riga di comando

Array di puntatori

Essendo i puntatori delle variabili, possono essere a loro volta memorizzati in array; ad esempio la dichiarazione seguente:

```
char *line[42] = { };
```

dichiara un array di 42 elementi, ognuno dei quali è un `char *`.

- In particolare, se ogni `line[i]` punta al primo elemento di un ulteriore array di caratteri, il risultato complessivo sarà che l'array `line` potrà essere utilizzato per memorizzare delle righe di testo.

Array multidimensionali

In C è anche possibile dichiarare array multidimensionali:

```
float matrix[4][3] = {
  { 1, 0, 0 },
  { 0, 1, 0 },
  { 0, 0, 1 },
  { 1, 1, 1 }
};
```

dichiara una matrice di 4 righe e 3 colonne di elementi `float`.

- Per accedere all'elemento in posizione (i, j) si scrive

```
matrix[i][j] = x;
```

- **Attenzione**: La sintassi `matrix[i,j]` compila ma è scorretta (quiz: cos'è?).

Array di puntatori e array multidimensionali

Le dichiarazioni

```
int a[10][20];
int *b[10];
```

differiscono per i motivi seguenti:

- La prima dichiarazione alloca la memoria necessaria per contenere 200 interi (10×20), mentre nel caso della seconda vengono allocate le locazioni necessarie per contenere 10 puntatori ad interi;
- nonostante le espressioni `a[3][4]` e `b[3][4]` denotino entrambe un `int`, nel caso di `b` ogni elemento `b[i]` può puntare ad un array di lunghezza diversa.

Argomenti da riga di comando

Come ogni comando Unix, anche un programma C può ricevere argomenti da riga di comando:

- Vengono passati come argomenti alla funzione `main()`, che può avere questa dichiarazione:

```
int main(int argc, char **argv);
```

- Il primo parametro (`argc`) indica il numero di parametri presenti.
- Il secondo parametro (`argv`) punta ad un **array** di **puntatori a carattere**, ognuno dei quali punta una stringa contenente un argomento.

Argomenti da riga di comando

Come ogni comando Unix, anche un programma C può ricevere argomenti da riga di comando:

- Vengono passati come argomenti alla funzione `main()`, che può avere questa dichiarazione:

```
int main(int argc, char **argv);
```

- Il primo parametro è sempre il nome del programma
- I restanti sono disposti in ordine, così come passati dalla shell.
- L'ultimo elemento (`argv[argc]`) è `NULL`.

Argomenti da riga di comando

Esempio

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  int somma = 0;

  if(argc >= 2 && strcmp(argv[1], "-s") == 0)
    somma = 1;

  int x = 0, y = 0;

  printf("Inserire due numeri: ");
  int n = scanf("%d %d", &x, &y);

  if(somma)
    printf("%d + %d = %d\n", x, y, x + y);
  else
    printf("%d * %d = %d\n", x, y, x * y);

  return 0;
}
```

Qualche informazione in più su `scanf()`

Qualche informazione in più su `scanf()`

La funzione `scanf()` può essere usata in modi più complessi di quelli visti finora. Ricapitolando le cose già viste:

- La stringa di formato indica quanti input e di che tipo ci si aspetta di leggere.
- Gli argomenti successivi sono **puntatori** alle variabili dove tali valori verranno scritti.

```
int x = 0, y = 0;
scanf("%d %d", &x, &y);
```

- La funzione ritorna il numero di valori letti e convertiti.
  - Due nell'esempio qui sopra
  - Meno di due se non viene letto un numero, o se termina l'input

Sintassi della stringa di formato

La funzione `scanf()` è più complessa di quanto visto finora.

- Per default, gli spazi bianchi tra due valori in input vengono ignorati.
- È possibile specificare altri caratteri che devono essere inseriti in input, e che verranno ignorati (ma richiesti), durante la lettura:

```
float real = 0, float imag = 0;
scanf("( %f , %f )", &real, &imag);
```

  In questo esempio, la funzione si aspetta in input due valori numerici, ma separati da una virgola e racchiusi tra parentesi.
- Qualsiasi specificatore, se arricchito con il carattere `'*'`, indica di leggere e convertire un valore secondo il tipo specificato, ma poi di ignorarlo:

```
scanf("(%f %*c %f)", &real, &imag);
```

Funzione `sscanf()`: leggere da una stringa

A volte è utile leggere dall'input un'intera riga di testo, e poi analizzarne il contenuto a parte:

- Per sapere in anticipo la lunghezza e quindi allocare sufficiente memoria
- Per velocità: leggere un blocco di dati in una sola volta è più veloce che leggere un carattere alla volta.

Funzione `sscanf()`: leggere da una stringa

La funzione `sscanf()` supporta questa necessità. Opera come `scanf()`, ma leggendo i dati da una stringa fornita come parametro invece che dallo standard input.

```
char *valori="(3.14, 0)";
float real = 0, imag = 0;

sscanf(valori, "(%f, %f)", &real, &imag);
printf("c = %f + i%f\n", real, imag);
```

Le funzioni `sprintf()` e `snprintf()`

Dualmente, esiste la funzione `sprintf()`, che opera come `printf()` ma scrivendo su una stringa invece che sullo standard output:

```
char valori[42] = "";
float real = 3.14, imag = 0;

sprintf(valori, "(%f, %f)", real, imag);
```

Una variante **raccomandata**, `snprintf()`, accetta un ulteriore argomento con la lunghezza massima della stringa di output:

```
snprintf(valori, 42, "(%f, %f)", real, imag);
```

Esercizi

Esercizi

1. Scrivere le funzioni (ispirate alla libreria standard)

```
char *strchr(char *str, char c);
char *strstr(char *str, char *pattern);
```

che restituiscono il puntatore rispettivamente alla prima occorrenza del carattere `c` e alla prima occorrenza della stringa `pattern` all'interno della stringa puntata da `str`. Se un'occorrenza non viene trovata, si restituisca un puntatore nullo.

Esercizi (2)

2. Scrivere una funzione

```
int readline(char *line, unsigned len);
```

che riempia la stringa puntata da `line` con caratteri letti dallo standard input, fino a che non viene letto `EOF`, il carattere di new line `'\n'`, o finché non vengono letti `len` caratteri. Ci si assicuri che la stringa venga terminata correttamente. La funzione deve restituire il numero di caratteri letti, oppure `-1` se nessun carattere è stato letto perché è stata trovata subito la costante `EOF`.

Esercizi (3)

3. Scrivere una funzione:

```
void capitalize(char *str);
```

che sostituisca, nella stringa puntata da `str`, ogni occorrenza di una lettera latina minuscola all'inizio di una parola con la corrispondente maiuscola.

Esercizi (4)

4. Scrivere un programma che legga una sequenza di numeri dallo standard input, e:
   - Se l'utente ha passato l'opzione `-r` sulla riga di comando, stampi i numeri in ordine inverso.
   - Se l'utente ha passato l'opzione `-s`, stampi i numeri ordinati in senso crescente.
   - Se l'utente ha passato l'opzione `-S`, stampi i numeri ordinati in senso decrescente.
   - Se l'utente non ha passato alcuna opzione, stampare un messaggio di errore e non fare nulla (ancora prima di leggere alcunché).

Corso di Laboratorio di Sistemi Operativi
A.A. 2019–2020

Lezione 11

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
nicola.gigante@uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Allocazione dinamica della memoria

Allocazione dinamica della memoria

Nello scrivere codice che tratti array e stringhe ci si sarà resi conto di una limitazione: la dimensione va dichiarata **staticamente**.

Com'è possibile quindi scrivere programmi che manipolino una quantità arbitraria e ignota a priori di valori?

- Nei programmi dati come esercizio nella lezione 9 e 10, è necessario prevedere quanti numeri verranno letti dallo standard input.
- In tutti gli esempi fatti finora riguardo le strighe, è sempre necessario indicare la lunghezza a priori.

La soluzione è ricorrere ad uno schema di allocazione **dinamica** della memoria.

Allocazione dinamica della memoria

La funzione `malloc()`

La funzione `malloc()` è dichiarata come segue:

```
void *malloc(unsigned n);
```

- La funzione accetta come argomento il numero di **byte** di memoria di cui si ha bisogno.
- Alloca una zona di memoria **contigua** della dimensione richiesta e restituisce un puntatore all'inizio di tale zona.
- Il tipo di ritorno `void *` è un puntatore ad un tipo qualsiasi.

Allocazione dinamica della memoria

Esempio dell'uso di `malloc()`

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

int somma(int *array, int size) {
  int s=0;
  for(int i=0; i<size; i++) s+=array[i];
  return s;
}

int main() {
  int n = 0;

  printf("Quanti numeri verranno inseriti? ");
  scanf("%d", &n);
  if(n == 0) return 0;

  int *elementi = malloc(n * sizeof(int));

  printf("Inserire i numeri: ");
  for(int i = 0; i < n; ++i) scanf("%d", elementi + i);
  printf("La somma dei numeri inseriti e': %d\n", somma(elementi, n));
  return 0;
}
```

Liberare la memoria allocata

La funzione `free()` serve a **liberare** la memoria allocata dinamicamente con `malloc()`.

```
int *elementi = malloc(n * sizeof(int));
// ...
free(elementi);
```

- A differenza delle variabili automatiche, la memoria allocata dinamicamente non ha uno **scope** preciso.
- Se non viene esplicitamente liberata dal programma, una zona di memoria allocata dinamicamente resterà assegnata al processo fino alla fine.

Liberare la memoria allocata

Il problema dei memory leak

In software complessi, dove le allocazioni dinamiche sono molte, è difficile tenere traccia di quando liberare la memoria:

- Ad ogni chiamata di `malloc()` dovrebbe seguire una chiamata a `free()`.
- Ma se alloco della memoria in una funzione e restituisco il puntatore al chiamante, chi dovrà poi liberare la memoria?
- Se il puntatore stesso poi andasse fuori scope, non sarebbe più possibile liberare la memoria allocata.
- Errori di questo tipo vengono detti **memory leak**.

Esempio

Creazione dinamica della concatenazione di due stringhe

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

char *concat(char *str1, char *str2)
{
  int len = strlen(str1) + strlen(str2);
  char *result = malloc(len + 1);

  strcpy(result, str1);
  strcat(result, str2);

  return result;
}
```

Riallocazione

La funzione `realloc()`

Tramite la funzione `realloc()` è possibile ridimensionare un'area di memoria allocata dinamicamente con una precedente chiamata a `malloc()`, restringendola o allargandola.

```
void *realloc(void *ptr, unsigned new_size);
```

La funzione ritorna un **nuovo** puntatore, perché i dati potrebbero dover essere spostati se intorno all'area già allocata non c'è spazio sufficiente.

Esempio

Lettura di un numero arbitrario di valori

```c
#include <stdio.h>
#include <stdlib.h>
void reverse(int *array, int size); // codice omesso per brevita'

int main() {
  int size = 10;
  int *array = malloc(size * sizeof(int));

  int read = 0;
  while(scanf("%d", &array[read]) == 1) {
    if(++read == size) {
      size *= 2;
      array = realloc(array, size * sizeof(int));
    }
  }

  reverse(array, read);
  for(int i = 0; i < read; i++) {
    printf("%d\n", array[i]);
  }
  free(array);
  return 0;
}
```

Azzerare la memoria allocata

È buona norma, come al solito, inizializzare ad un valore noto la memoria.

- Nell'esempio precedente, nessun elemento veniva letto prima di essere scritto, quindi il programma era corretto.
- In codice più complesso potrebbe essere difficile esserne sicuri.
- È possibile allocare direttamente memoria già azzerata:

```
void *calloc(unsigned count, unsigned size);
```

- La funzione è equivalente a:

```
void *calloc(unsigned count, unsigned size) {
  unsigned len = count * size;
  char *mem = malloc(len);
  for(int i = 0; i < len; ++i) {
    mem[i] = 0;
  }
  return mem;
}
```

- Esempio:

```
int *array = calloc(n, sizeof(int));
```

Le strutture

Le strutture

Le strutture sono un tipo **aggregato**, che raggruppa variabili di tipo diverso in un'unica entità.

- Dichiarazione di una struttura:

```
struct point {
  float x;
  float y;
};
```

La dichiarazione di una struttura definisce un **tipo di dato**.

- Dichiarazioni di variabili di tipo `struct point`:

```
struct point p = { 3, 4 };
printf("%f, %f\n", p.x, p.y);
```

Uso di strutture

Data una struttura è possibile accedere liberamente alle sue componenti oppure operare sull'intera struttura.
Esempi:

- Uso individuale:

```
struct point p = { };
scanf("{ %f , %f }", &p.x, &p.y);
```

- Passaggio di un'intera struttura ad una funzione:

```
#include <math.h>
#include <stdio.h>

float abs(struct point p) {
  return sqrt(p.x * p.x + p.y * p.y);
}
...
struct point p = { 0.707, 0.707 };
printf("%f\n", abs(p)); // Stampa ~ 1
```

Puntatori e array di strutture

Le strutture sono da considerarsi allo stesso livello dei tipi base:

- Possono essere contenuti in **array**:

```
struct point points[] = { { 3, 4 }, { 12, 15 }, { 0, -1 } };
printf("%d %d", points[0].x, points[0].y);
```

- Si possono dichiarare **puntatori** a strutture:

```
struct point   p = { 0, 0 };
struct point *pp = &p;
```

- L'accesso alle componenti di una struttura passando per un puntatore è un'operazione così comune da meritare un **operatore** apposta (`s->var`):

```
printf("%f %f\n", pp->x, pp->y); // Equivalente alla seguente
printf("%f %f\n", (*pp).x, (*pp).y);
```

Uso delle strutture

Le strutture sono uno strumento per organizzare il codice, aggregando dati correlati tra loro.

- Si possono vedere come un antenato del concetto di **classe** della programmazione orientata agli oggetti.
- Attenzione però a non trasferire troppi concetti: il C non è orientato agli oggetti.
- In congiunzione con l'allocazione dinamica, sono i mattoni di base per l'implementazione di qualsiasi **struttura dati** complessa.

Esempio di struttura dati

Lista concatenata

Una lista concatenata può essere descritta come una catena di nodi, ognuno dei quali punta al successivo nella catena.

In C, questa idea si può implementare in modo molto diretto:

```
struct node {
  int data;
  struct node *next;
};
```

Esempio di struttura dati (2)

Lista concatenata

I nodi della lista andranno aggiunti e rimossi in modo non prevedibile a priori, quindi vanno **allocati dinamicamente**.

- Ogni nuovo nodo deve essere inizializzato con il valore del campo `data` e il puntatore `next` impostato a `NULL`.
- Meglio raggruppare queste operazioni in una funzione:

```
struct node *create(int data) {
  struct node *ptr = malloc(sizeof(struct node));
  ptr->data = data;
  ptr->next = NULL;

  return ptr;
}
```

Esempio di struttura dati (3)

Lista concatenata

Esempio di navigazione della lista, la funzione `length()`:

```
int length(struct node *head) {
  int len = 0;
  for(struct node *n = head; n; n = n->next) {
    ++len;
  }

  return len;
}
```

Esempio di struttura dati (4)

Lista concatenata

Ricerca di un elemento in una lista:

```
struct node *find(struct node *head, int data) {
  for(struct node *n = head; n; n = n->next) {
    if(n->data == data)
      return n;
  }
  return NULL;
}
```

Esempio di struttura dati (5)

Lista concatenata

Concatenazione di due liste:

```
struct node *last(struct node *head) {
  for(struct node *n = head; n; n = n->next) {
    if(n->next == NULL)
      return n;
  }
  return NULL;
}

struct node *append(struct node *head1, struct node *head2) {
  struct node *last1 = last(head1);
  last1->next = head2;

  return head1;
}
```

Esempio di struttura dati (6)

Lista concatenata

La memoria occupata da tutti i nodi allocati dinamicamente, alla fine del proprio utilizzo, deve essere rilasciata al sistema operativo.

È opportuno raggruppare in una funzione il rilascio di tutta la lista in un colpo solo:

```
void destroy(struct node *head) {
  struct node *next = head;

  while(next) {
    struct node *n = next;
    next = n->next;
    free(n);
  }
}
```

Esercizi

Esercizi

1. Dichiarare una struttura `struct complex` per rappresentare numeri complessi a partire dalla parte reale e immaginaria:
   - Utilizzare il tipo di dato `float` per le componenti.
   - Scrivere le funzioni `cabs()` e `angle()` per il calcolo del modulo e dell'argomento di un numero complesso.
   - Scrivere una funzione `from_polar()` che restituisca un numero complesso a partire da modulo e argomento.

   Nota: Le funzioni trigonometriche si trovano in `<math.h>`
   Nota: In questo esercizio non servono allocazioni dinamiche.
2. Riscrivere le funzioni `length()`, `find()`, `last()`, `append()`, e `destroy()` per le liste concatenate utilizzando un approccio ricorsivo.

Esercizi (2)

3. Dichiarare una struttura per la rappresentazione di un **albero binario di ricerca** (non necessariamente bilanciato), ed implementare le relative operazioni:
   - Una funzione `create()` per creare un albero con solo la radice.
   - Una funzione `insert()` per inserire un numero intero all'interno dell'albero.
   - Una funzione `find()` per trovare un valore nell'albero.
   - Una funzione `remove()` per rimuovere un valore dall'albero.
   - Una funzione `destroy()` per liberare la memoria occupata da tutti i nodi dell'albero.
   - Una funzione `to_list()` che restituisca una lista concatenata (implementata a lezione o nell'esercizio 2), contenente i valori dell'albero in ordine di visita.

   Nota: La funzione `remove()` deve occuparsi di liberare la memoria del nodo contenente il valore che viene rimosso.

Corso di Laboratorio di Sistemi Operativi

Lezione 12

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Compilazione separata C e gestione del progetto

Distribuire il codice su diversi file

In progetti appena più grandi di un esercizio è impensabile scrivere tutto in un unico file. I programmi C possono essere scritti su più file, adottando alcuni accorgimenti:

- Uno dei file del progetto dovrà contenere la funzione `main()`.
- Una funzione definita in un file può chiamarne una definita in un file diverso, purché sia comunque visibile la sua **dichiarazione**. Esempio:

```c
int add(int, int); // Solo la dichiarazione

int add3(int x, int y, int z) {
  return add(x, add(y,z));
}
```

Distribuire il codice su diversi file

File di intestazione (header)

In ogni file è quindi necessario avere la dichiarazione delle funzioni definite altrove che verranno chiamate.
Non solo: lo stesso vale per tipi di dato (`struct`), costanti, ecc...

Per non ripetere tutte le dichiarazioni in molti posti, la pratica vuole che si riuniscano nei cosiddetti **file di intestazione**:

- Un file di intestazione (chiamato anche **header**), solitamente chiamato con l'estensione `.h`, contiene solo **dichiarazioni** di un codice.
- Il file viene incluso con la direttiva `#include <file.h>` (o `#include "file.h"`) del preprocessore, che include testualmente il contenuto del file incluso nel file corrente.
- Il famoso `stdio.h` non è altro che questo.

Distribuire il codice su diversi file

Esempio

File `add.h`:

```
// Funzione che permette
// di sommare due numeri interi
int add(int x, int y);
```

File `add.c`:

```
#include "add.h"
int add(int x, int y) {
  return x + y;
}
```

File `main.c`:

```
#include <stdio.h>

#include "add.h"

int main()
{
  printf("3 + 4 = %d\n", add(3,4));

  return 0;
}
```

Si compila con:

```
$ clang main.c add.c -o add
$ ./add
3 + 4 = 7
```

Includere più volte lo stesso header

Spesso gli header saranno scritti da una persona/team, e usati da un'altra. Ipotizziamo che:

- un header `A.h` ne include un altro `B.h`
- anche chi usa `A.h` include `B.h` (ignorando, com'è giusto che sia, che `A.h` già lo include)

Allora, il file `A.h` verrà incluso due volte e le dichiarazioni contenute saranno processate due volte, portando ad errori.

Serve un meccanismo per includere una volta sola il contenuto di un header anche quando questo viene potenzialmente incluso più volte.

Includere più volte lo stesso header

Le direttive #ifdef/#ifndef e le header guards

La direttiva del preprocessore #ifdef serve a scopi come questo:

```
#define SIMBOLO
#ifdef SIMBOLO
  codice...
#endif
```

Il codice tra #ifdef SIMBOLO e #endif verrà incluso solo se è stata prima definita la costante del preprocessore SIMBOLO.
La direttiva #ifndef funziona ugualmente, ma **esclude** il codice se la costante del preprocessore è stata definita.
Nota: È anche possibile specificare un ramo #else.

Includere più volte lo stesso header

La direttiva `#ifdef` e le header guards

Sfruttando questo meccanismo possiamo scrivere un header file che si accorga nel caso venga incluso più volte:

Esempio sul file `add.h` visto prima:

```
#ifndef ADD_H__
#define ADD_H__

// Funzione che permette di sommare due numeri interi
int add(int x, int y);

#endif
```

Il processo di compilazione

La compilazione di un programma C avviene in più fasi:

1. Preprocessing
2. Compilazione
3. Assemblaggio
4. Linking

Il processo di compilazione

Compilazione

La compilazione, in senso più stretto, riguarda la traduzione del programma da codice C a codice *assembly*. È possibile chiedere al compilatore di fermarsi dopo questa fase con l'opzione `-S`.

Ad esempio, se in un file si scrive una sola funzione:

```
int add(int x, int y) {
  return x + y;
}
```

Si può ottenere il relativo codice assembly:

```
$ clang -S add.c
$ cat add.s
<codice asm...>
```

Il processo di compilazione

Assemblaggio

L'assemblaggio consiste nel trasformare il codice assembly (che è ancora testo) in codice macchina in formato binario eseguibile.
Si può chiedere al compilatore di fermarsi a questa fase con l'opzione `-c`:

```
$ clang -c add.c
$ ls
add.c add.o
```

Il file risultante, `add.o`, è detto **file oggetto**. Contiene codice eseguibile, ma non è ancora un programma completo.

Il processo di compilazione

Linking

Il linking, effettuato da un programma chiamato **linker**, prende un **insieme** di file oggetto e di **librerie dinamiche** e compone un programma eseguibile completo:

- I file oggetto vengono riuniti in un tutt'uno, e i **riferimenti** a chiamate di funzioni chiamate in un file, ma non definite vengono risolti con le funzioni definite in altri file.
- Le chiamate a funzioni ancora irrisolte vengono collegate a funzioni esportate dalle librerie dinamiche elencate (es. la libreria standard).

Compilazione separata

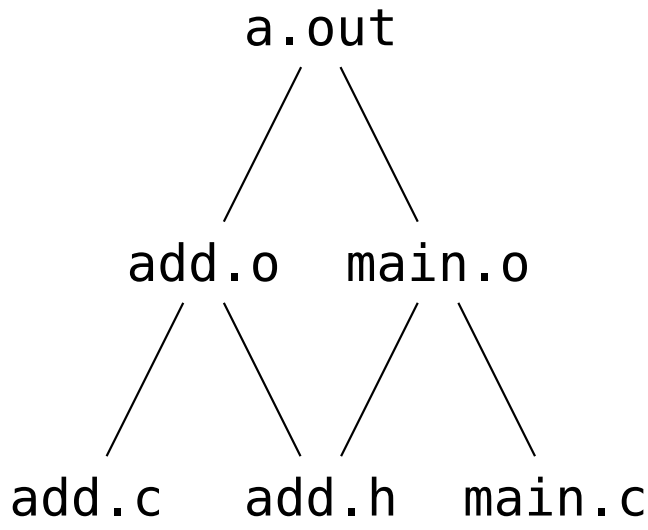
La compilazione del programma add dell'esempio precedente può essere quindi fatta anche per stadi, un file alla volta:

```
$ clang add.c -c
$ clang main.c -c
$ clang add.o main.o -o add
```

Solitamente si fa proprio in questo modo, per efficienza:

- Si possono compilare più file contemporaneamente.
- Se viene cambiato un file, basta ricompilare solo quello e poi rifare il linking.
- Va però tenuto traccia di quali file sono stati modificati dall'ultima compilazione, e in quali file vengono inclusi gli header.

Grafo delle dipendenze



Il grafo mette in luce le dipendenze fra i vari file, ad esempio da `add.h` dipende sia da `add.o` che da `main.o`.

Quindi, in caso di modifiche ad `add.h`, percorrendo il grafo verso l'alto notiamo che devono essere aggiornati sia `add.o` che `main.o`, mentre invece modificando `main.c` va ricompilato solo il file stesso.

Il comando make

Il comando make è un programma che si occupa di gestire la compilazione di programmi C (o di altri linguaggi).

- Il grafo delle dipendenze viene codificato in un file di testo chiamato `Makefile` che risiede nella stessa directory dei file sorgente.
- Esempio di `Makefile`:

```
add: add.o main.o
    clang add.o main.o -o add
add.o: add.h add.c
    clang -c add.c
main.o: add.h main.c
    clang -c main.c
```

- Invocando il comando make direttamente, viene interpretato il `Makefile` e compilato il programma.

Il comando make

La sintassi di base dei `Makefile` è molto semplice.

- Un `Makefile` è composto da regole specificate dalla seguente sintassi (di default si comincia ad eseguire le regole dalla prima, l'ordine delle rimanenti non è importante):

```
target : source file(s)
    command
```

- Nota: Il comando deve essere preceduto da un <Tab>.
- Il comando `make` controlla le date di ultima modifica dei file. Se un file (source) ha una data di modifica più recente di quella dei file che da esso dipendono (target), questi ultimi vengono aggiornati eseguendo i comandi specificati nelle regole del `Makefile`.

Esercizi

1. Riscrivere alcuni dei programi scritti nelle settimane passate come esercizio, dividendo il codice in più file (separando, ad esempio, la funzione `main()` dalle altre) con relativi file di intestazione e `Makefile`.
2. Riscrivere alcuni dei programmi scritti nelle settimane passate come esercizio, facendoli però operare su argomenti passati sulla riga di comando.

Corso di Laboratorio di Sistemi Operativi

Lezione 13

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Introduzione alla programmazione di sistema

Programmazione di sistema

Il C è un linguaggio legato a doppio filo con i sistemi `UNIX`:

- Il 99% del sistema operativo è scritto in C
- L'interfaccia che il sistema operativo fornisce ai programmi utente è definita in termini di funzioni C.

Chiamate di sistema e libreria standard

Le chiamate di sistema sono il modo in cui un processo chiede al sistema operativo di fare qualcosa.

- Si attivano tramite degli interrupt della CPU.
- Al programmatore C, però, vengono fornite delle **funzioni**, dichiarate in file di intestazione e implementate in una libreria, solitamente chiamata `libc`, che implementa anche le funzioni della libreria standard.
- Sono quindi due le fonti di funzioni "standard" che un programmatore C ha a disposizione:
  - Funzioni della libreria standard del linguaggio, definite dallo standard ISO C.
  - Funzioni di interfaccia per le chiamate di sistema, definite dallo standard POSIX (Portable Operating System Interface for Unix).

Programmazione di sistema in ambiente UNIX

Programmazione di sistema significa gestire ed effettuare operazioni di basso livello, messe a disposizione dal sistema operativo, per operare su:

- file
- directory
- processi
- comunicazione tra processi
- altro...

Funzionalità e operazioni solitamente disponibili all'utente tramite comandi della shell, sono implementate in C tramite chiamate di sistema apposite.

Accesso ai file

Accesso a file

Un primo esempio di accesso a servizi esposti dal sistema operativo è l'accesso ai file.

Esistono due modi complementari per accedere ai file in un programma C:

- Funzioni della libreria standard (`fopen()`, `fread()`, ecc...), dichiarate da `<stdio.h>`
- Chiamate di sistema POSIX (`open()`, `read()`, ecc...), più a basso livello, ma più flessibili, dichiarate da `<unistd.h>`

Aprire un file

Prima di leggere/scrivere su un file, un programma C deve **aprire** il file tramite la funzione `fopen()`:

```
FILE *fopen(char *name, char *mode);
```

La funzione `fopen` prende come parametri:

- il nome del file, `name`
- una stringa, `mode`, che indica il **modo** di utilizzo del file:
  - `"r"` (lettura)
  - `"w"` (scrittura)
  - `"a"` (append)

e restituisce un **puntatore** ad una struttura di tipo `FILE`, detto "file pointer" che identificherà il file aperto nelle successive chiamate ad altre funzioni di manipolazione di file.

Lettura e scrittura

Una volta aperto, un file può essere letto/scritto in vario modo.
Il modo più facile sono alcune funzioni molto familiari:

```
int fprintf(FILE *fp, char *format, ...);
int fscanf(FILE *fp, char *format, ...);
int fgetc(FILE *fp);
int fputc(int c, FILE *fp);
```

Le quattro funzioni operano esattamente come `printf()`, `scanf()`, `getchar()` e `putchar()`, rispettivamente, ma su un file qualsiasi invece che sullo standard input/output.

Chiudere un file

Al termine delle operazioni di lettura/scrittura di un file, è buona norma rilasciare il file pointer, utilizzando la funzione

```
int fclose(FILE *fp);
```

Si applicano le stesse avvertenze riguardo la coppia di funzioni `malloc()`/`free()`.

Accorgersi della fine del file

Alcune funzioni di lettura, come `fgetc()` ritornano la costante `EOF` quando incontrano la fine del file o quando incontrano un **errore**.

Per distinguere i due casi è comoda la funzione `feof()`:

```
int feof(FILE *fp);
```

La funzione restituisce "vero" (un numero positivo) se la posizione di lettura del file è arrivata alla fine.

Lettura e scrittura

I file predefiniti

In un programma C sono sempre presenti tre file pointer standard, già aperti e pronti all'uso:

- `stdout` è un file aperto in scrittura, puntato sullo **standard output** del processo.
- `stdin` è un file aperto in lettura, puntato sullo **standard input** del processo.
- `stderr` è un file aperto in scrittura, puntato sullo **standard error** del processo.

Lettura e scrittura

I file predefiniti (2)

In effetti, `printf()` e `scanf()` sono equivalenti ad una chiamata a `fprintf()` e `fscanf()` passando `stdout` e `stdin` come file.

```
printf("Ciao mondo!");           // Queste due righe...
fprintf(stdout, "Ciao mondo!"); // ...sono equivalenti
```

Accesso ai file

Esempio

```c
#include <stdio.h>

int main(int argc, char **argv) {
  if(argc < 2) {
    fprintf(stderr, "Fornire il nome del file\n");
    return 1;
  }

  char *filename = argv[1];

  FILE *file = fopen(filename, "r");
  if(!file) {
    fprintf(stderr, "Errore nell'apertura del file!\n");
    return 2;
  }

  int n = 0, sum = 0;
  while(fscanf(file, "%d", &n) == 1) {
    sum += n;
  }

  if(!feof(file)) {
    fprintf(stderr, "Il file non conteneva solo numeri.\n");
    return 3;
  }

  printf("Somma dei numeri contenuti: %d\n", sum);

  return 0;
}
```

Gestione degli errori

`ferror()` e `feof()`

Una funzione come `fgetc()` potrebbe aver restituito `EOF` anche perché il file è effettivamente terminato.

Le funzioni `feof()` e `ferror()` servono a distinguere i due casi:

```
char c = fgetc(file);
if(c == EOF) {
  if(feof(file))
    printf("Il file e' terminato\n");
  else if(ferror(file))
    printf("La lettura ha causato un errore\n");
}
```

Gestione degli errori

La variabile errno

Per riportare **quale** errore si è verificato, le varie funzioni impostano la variabile globale **errno**, dichiarata in <errno.h>, con un valore che rappresenta il motivo dell'errore.

```
FILE *file = fopen("/my/file", "r");
if(file == NULL) {
  if(errno == EACCES)
    fprintf(stderr, "Si e' verificato un errore di permessi\n");
  if(errno == EISDIR)
    fprintf(stderr, "Il file richiesto e' una directory\n");
}
```

Il file errno.h dichiara tutte le costanti del preprocessore corrispondenti ai possibili errori riportati dalle funzioni standard (come EACCES o EISDIR nell'esempio).

Gestione degli errori

Le funzioni `perror()` e `strerror()`

Una stringa standard di descrizione della variabile `errno` si può ottenere tramite la funzione `strerror()`.

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(int argc, char **argv) {
  if(argc < 2) {
    fprintf(stderr, "Specificare il nome di un file\n");
    return 1;
  }

  FILE *file = fopen(argv[1], "r");
  if(file == NULL) {
    fprintf(stderr, "%s: Impossibile aprire %s: %s\n",
            argv[0], argv[1], strerror(errno));
    return 2;
  }
  return 0;
}
```

Anche la funzione `perror()` serve proprio a questo scopo.

Gestione degli errori

Le funzioni `perror()` e `strerror()`

Una stringa standard di descrizione della variabile `errno` si può ottenere tramite la funzione `strerror()`.

Output:

```
$ ./programma pippo.txt # Inesistente
./programma: Impossibile aprire pippo.txt: No such file or directory
```

La posizione di lettura/scrittura

Lettura e scrittura di un file avvengono in maniera sequenziale, dall'inizio alla fine, ma è possibile spostarsi arbitrariamente con la funzione `fseek()`:

```
int fseek(FILE *file, long offset, int whence);
```

La funzione `fseek` imposta la posizione attuale a `offset` byte di distanza dalla posizione indicata da `whence`, che può essere:

- `SEEK_SET`: Inizio del file
- `SEEK_CUR`: Posizione attuale
- `SEEK_END`: Fine del file

Ad esempio, per tornare all'inizio del file:

```
fseek(file, 0, SEEK_SET);
```

La funzione `ftell()` restituisce la posizione attuale:

```
int ftell(FILE *file);
```

La posizione di lettura/scrittura

Esempio: conoscere a priori la lunghezza di un file

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)
{
  char *filename = argv[1];
  FILE *file = fopen(filename, "r");
  if(!file) {
    fprintf(stderr, "%s: Impossibile aprire il file %s: %s",
            argv[0], filename, strerror(errno));
    return 2;
  }

  fseek(file, 0, SEEK_END);
  long bytes = ftell(file);

  printf("Il file e' lungo %ld bytes\n", bytes);

  fclose(file);

  return 0;
}
```

Input/Output binario

La lettura e scrittura di dati testuali non è l'unico modo di accesso ai file. Specificando `"rb"` o `"wb"` come parametro `mode` di `fopen()` è possibile effettuare Input/Output di dati **binari**.

- I dati vengono scritti e letti senza intermediazioni (es. traduzioni di codifica dei caratteri). Per l'I/O binario si usano funzioni apposite.
- La lettura si effettua tramite la funzione `fread()`:

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *file);
```

  Legge `size * nitems` byte dal file e scrive nella memoria puntata da `ptr`.

- La scrittura si effettua tramite la funzione `fwrite()`:

```
size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *file);
```

  Scrive sul file `size * nitems` byte dalla memoria puntata da `ptr`.

Accesso ai file (funzioni POSIX)

Funzioni POSIX per l'accesso ai file

Mentre le funzioni ISO C sono specificate dallo standard del linguaggio, le funzioni più di basso livello fornite dallo standard POSIX sono specifiche dei sistemi UNIX.

Non sono un'interfaccia completamente ridondante:

- le funzioni POSIX permettono di scrivere e leggere da fonti diverse da file regolari: pipe, socket, ecc...
- sui sistemi Unix, le funzioni POSIX sono usate per implementare le funzioni ISO. Queste ultime adottano un **buffer** interno, mentre le funzioni POSIX effettuano direttamente le relative system call.
- le funzioni POSIX permettono di gestire i **permessi** dei file, i link, e altri attributi dei file specifici di UNIX.

System call POSIX per l'accesso ai file

```
#include <unistd.h>
```

Funzione	Scopo
`open()`	apre un file in lettura e/o scrittura o crea un nuovo file
`creat()`	crea un file nuovo
`close()`	chiude un file precedentemente aperto
`read()`	legge da un file
`write()`	scrive su un file
`lseek()`	sposta la posizione di lettura/scrittura
`unlink()`	rimuove un file
`fcntl()`	controlla alcuni attributi associati ad un file
`chmod()`	cambia i permessi di un file

Aprire un file

La funzione `open()` serve ad aprire un file:

```
int open(const char *path, int openflags);
```

Apre il file `path`, nel modo specificato da `openflags`, che può essere una **combinazione** dei seguenti **flag**:

Funzione	Scopo
`O_RDONLY`	open for reading only
`O_WRONLY`	open for writing only
`O_RDWR`	open for reading and writing
`O_APPEND`	append on each write
`O_CREAT`	create file if it does not exist
`O_TRUNC`	truncate size to 0
`O_EXCL`	error if O_CREAT and the file exists

Restituisce un **file descriptor** che rappresenta il file aperto.

Aprire un file

Esempio

```c
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
  // Apriamo in scrittura, appendendo, creando il file se non esite
  int fd = open(argv[1], O_WRONLY | O_APPEND | O_CREAT, 0644);

  char contents[13] = "Hello World\n";

  write(fd, contents, 12);

  close(fd);

  return 0;
}
```

File descriptors

La funzione `open()` restituisce un numero intero chiamato **file descriptor**.

- Rappresenta il file aperto
- Non solo veri file possono essere associati a file descriptor, ma anche altro, come pipes, socket, ecc...
- In Unix, "tutto è un file" è un motto ricorrente, quindi i file descriptor compaiono dappertutto.
- Tre file descriptor sono già aperti all'inizio del programma: lo standard input (0), standard output (1) e standard error(2):

```
write(1, "Hello world\n", 12);
```

Lettura e scrittura

Lettura e scrittura avvengono tramite le funzioni `read()` e `write()`, usate in modo analogo a `fread()` e `fwrite()`.

```
ssize_t read(int fd, void *buffer, size_t nbytes);
```

Legge `nbytes` byte dal file descriptor `fd`, li scrive sulla memoria puntata da `buffer`, e restituisce il numero di byte letti, o -1 se avviene un errore.

```
ssize_t write(int fd, void *buffer, size_t nbytes);
```

Scrive `nbytes` byte dalla memoria puntata da `buffer` sul file descriptor `fd`, e restituisce il numero di byte scritti, o -1 se avviene un errore.

Le chiamate `stat` e `fstat`

Le chiamate di sistema `stat` e `fstat` permettono di accedere in lettura alle informazioni e proprietà associate ad un file (dispositivo del file, numero di inode, tipo del file, numero di link, UID, GID, dimensione in byte, data ultimo accesso/ultima modifica, informazioni sui blocchi che contengono il file):

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

int stat(const char *pathname, struct stat *out);
int fstat(int filedes, struct stat *out);
```

La funzione `stat()` prende come primo argomento un pathname, mentre `fstat()` opera su un descrittore di un file già aperto.

Il risultato viene scritto in una struttura di tipo `struct stat` puntata dal parametro `out`

La struttura stat

stat è una struttura definita in <sys/stat.h> che definita in questo modo:

```
struct stat {
  dev_t st_dev;     // device id
  ino_t st_ino;     // inode number
  mode_t st_mode;   // tipo di file e permessi
  nlink_t st_nlink; // numero di link non simbolici
  uid_t st_uid;     // UID
  gid_t st_gid;     // GID
  dev_t st_rdev;    // device type
  off_t st_size;    // dimensione del file
  time_t st_atime;  // tempo di ultimo accesso
  time_t st_mtime;  // tempo di ultima modifica
  time_t st_ctime;  // tempo di creazione
  long st_blksize;  // dimensione del blocco
  long st_blocks;   // numero di blocchi
};
```

La funzione `stat()`

Esempio

Il programma `lookout.c`, dato un percorso di un file passato su linea di comando, controlla ogni 5 secondi se il file è stato modificato. Nel caso ciò avvenga termina l'esecuzione stampando un messaggio che informa l'utente dell'evento.

La funzione `stat()`

Esempio

La chiamata a `stat` ci dà informazioni sul file.

```
struct stat sb;
if(stat(file, &sb) == -1) {
  fprintf(stderr, "%s: errore nell'accesso al file %s: %s\n",
          argv[0], file, strerror(errno));
  return 1;
}

time_t mtime = sb.st_mtime;
```

La funzione `stat()`

Esempio

Ogni 5 secondi confrontiamo il valore di `st_mtime` con quello che avevamo salvato.

```
while(1) {
  if(stat(file, &sb) == -1) {
    fprintf(stderr, "%s: errore nell'accesso al file %s: %s\n",
            argv[0], file, strerror(errno));
    return 1;
  }

  if(sb.st_mtime != mtime) {
    printf("Il file %s e' stato modificato\n", file);
    return 0;
  }
  sleep(5);
}
```

Esercizi

Esercizi

1. Scrivere un programma C, versione semplificata del comando Unix `cat`, per l'append di uno o più file su standard output.
2. Scrivere un programma C, versione semplificata del comando Unix `cmp` per il confronto di due file, che stampa la prima linea su cui i file differiscono.
   Nota: Si può reimplementare un analogo della funzione `readline()` vista negli esercizi precedenti, che però utilizzi `fgetc()` invece di `getchar()`, oppure investigare l'utilizzo della funzione `fgets()`.

Esercizi (2)

3. Si scriva un programma che legga in modalità binaria dieci numeri interi dal file `/dev/urandom`, e li stampi sullo schermo, utilizzando funzioni ISO per l'accesso ai file (**attenzione:** utilizzare `rb` come modalità di apertura perché `/dev/urandom` è assimilabile ad un file binario).
4. Si riscriva lo stesso programma utilizzando funzioni POSIX per l'accesso ai file.
5. Si scriva un programma che conti le modifiche ad un file (specificato come primo argomento sulla riga di comando) nell'arco di un intervallo di tempo specificato in secondi come secondo argomento sulla linea di comando.
   Suggerimento: la chiamata `time(NULL)` restituisce un valore dello stesso tipo di `stat->st_mtime`. Si tratta del numero di secondi trascorsi dalla mezzanotte di capodanno del 1970.

Corso di Laboratorio di Sistemi Operativi

Lezione 14

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Gestione dei processi

Processi in un sistema UNIX

Il **processo** è l'unità base di esecuzione di un sistema UNIX:

- Ogni processo corrisponde all'esecuzione di un programma.
- Un programma può essere eseguito in più processi diversi.
- Ogni processo è isolato dagli altri. Le risorse assegnate al processo dal sistema operativo sono inaccessibili agli altri processi:
  - Memoria
  - Registri della CPU
  - File aperti
  - ecc. . .

Controllo di processi

Esistono svariate **system call** per creare, controllare e gestire processi:

- `getpid()`, `getppid()`, `getpgrp()` ecc... forniscono degli attributi dei processi (PID, PPID, gruppo ecc.).
- `fork()`: crea un processo figlio duplicando il chiamante.
- `exec()`: trasforma un processo lanciando l'esecuzione di un nuovo programma al posto del programma chiamante.
- `wait()`: permette la sincronizzazione fra processi; il chiamante "attende" la terminazione di un processo correlato.
- `exit()`: termina un processo.

La system call `fork()`

- La chiamata di sistema basilare è `fork()`:

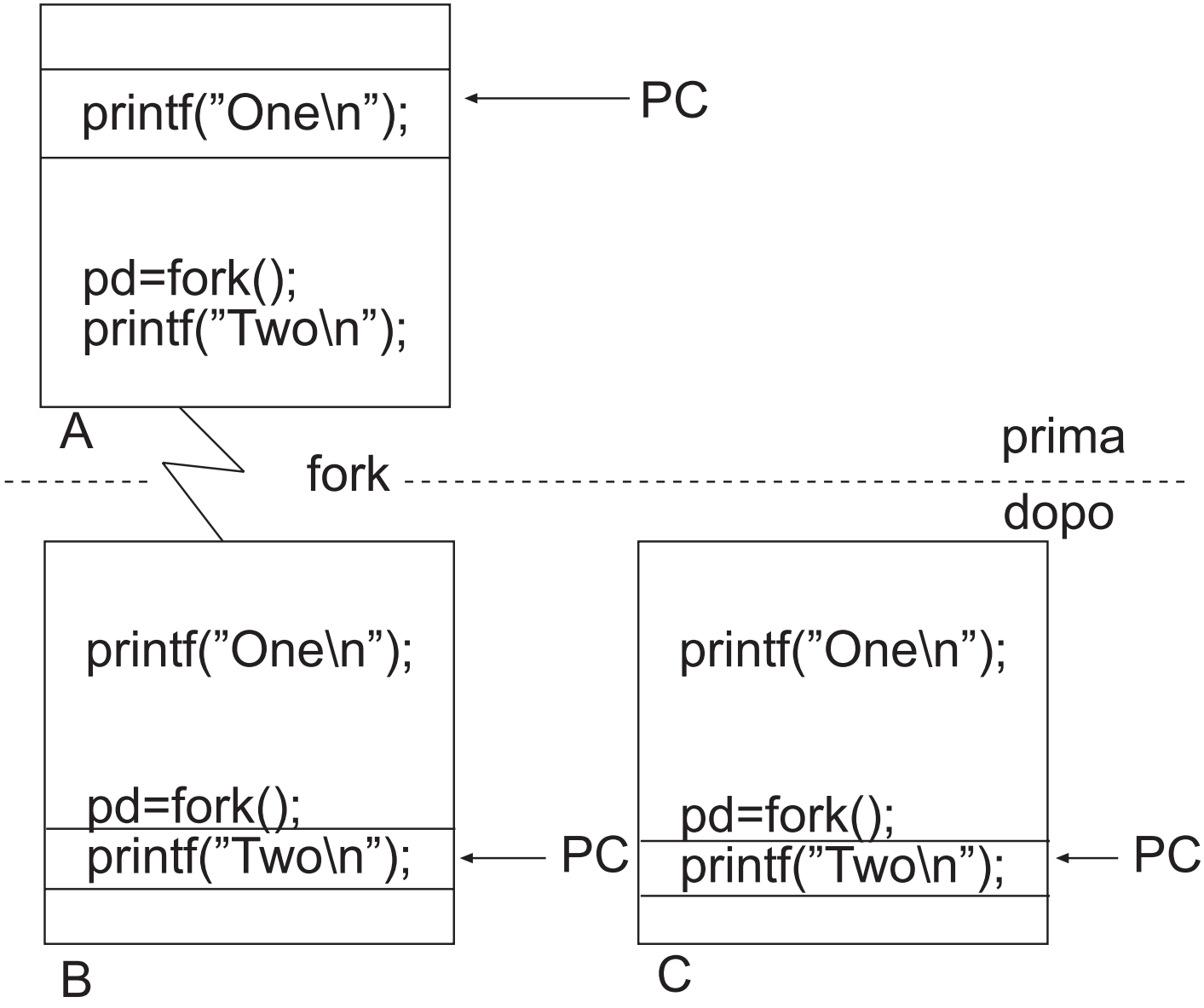
```
pid_t fork();
```

  dove `pid_t` è un tipo speciale definito in `<sys/types.h>` e, solitamente, corrisponde ad un tipo intero.

- La chiamata **crea una copia** del processo chiamante:
  - Stesso contenuto di registri, memoria ecc...
  - Stessi file aperti
  - Stesse variabili d'ambiente
  - Stesso utente, gruppo, ecc...
- Il processo figlio prosegue l'esecuzione in modo totalmente **indipendente** e **isolato**.
- Il valore restituito da `fork()` serve a distinguere tra processo genitore e processo figlio; infatti al genitore viene restituito il PID del figlio, mentre a quest'ultimo viene restituito 0.

La system call `fork()`

Fork di due processi



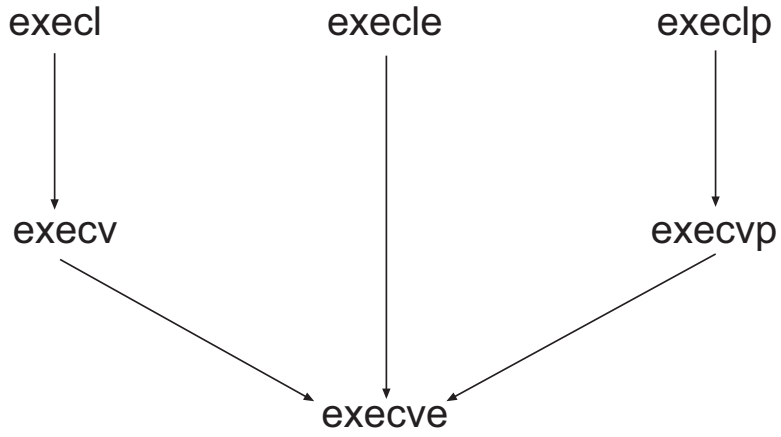
La system call `fork()`

Esempio di utilizzo

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

int main()
{
  printf("Un solo processo con PID %d.\n", (int)getpid());
  printf("Chiamata a fork...\n");

  pid_t pid=fork();

  if(pid == 0)
    printf("Sono il processo figlio (PID: %d).\n", (int)getpid());
  else if(pid > 0)
    printf("Sono il genitore del processo con PID %d.\n",pid);
  else
    fprintf(stderr,
            "Si e' verificato un errore nella chiamata a fork.\n");

  return 0;
}
```

La famiglia di system call exec

- La chiamata di sistema `fork()` permette di creare nuovi processi, ma si limita a copiare processi già esistenti.
- La famiglia di funzioni `exec()` viene utilizzata per lanciare processi che eseguano **programmi diversi** da quello chiamante.
- La funzione non crea un nuovo processo: **sostituisce** il processo attuale con l'esecuzione di un altro file eseguibile.
- In realtà tutte le funzioni chiamano in ultima analisi execve che è l'unica vera system call della famiglia. Le differenze tra le varianti stanno nel modo in cui vengono passati i parametri.



La funzione execl()

La versione più semplice da utilizzare della famiglia è execl():

```
int execl(const char *path, const char *arg0, ...);
```

L'utilizzo è semplice:

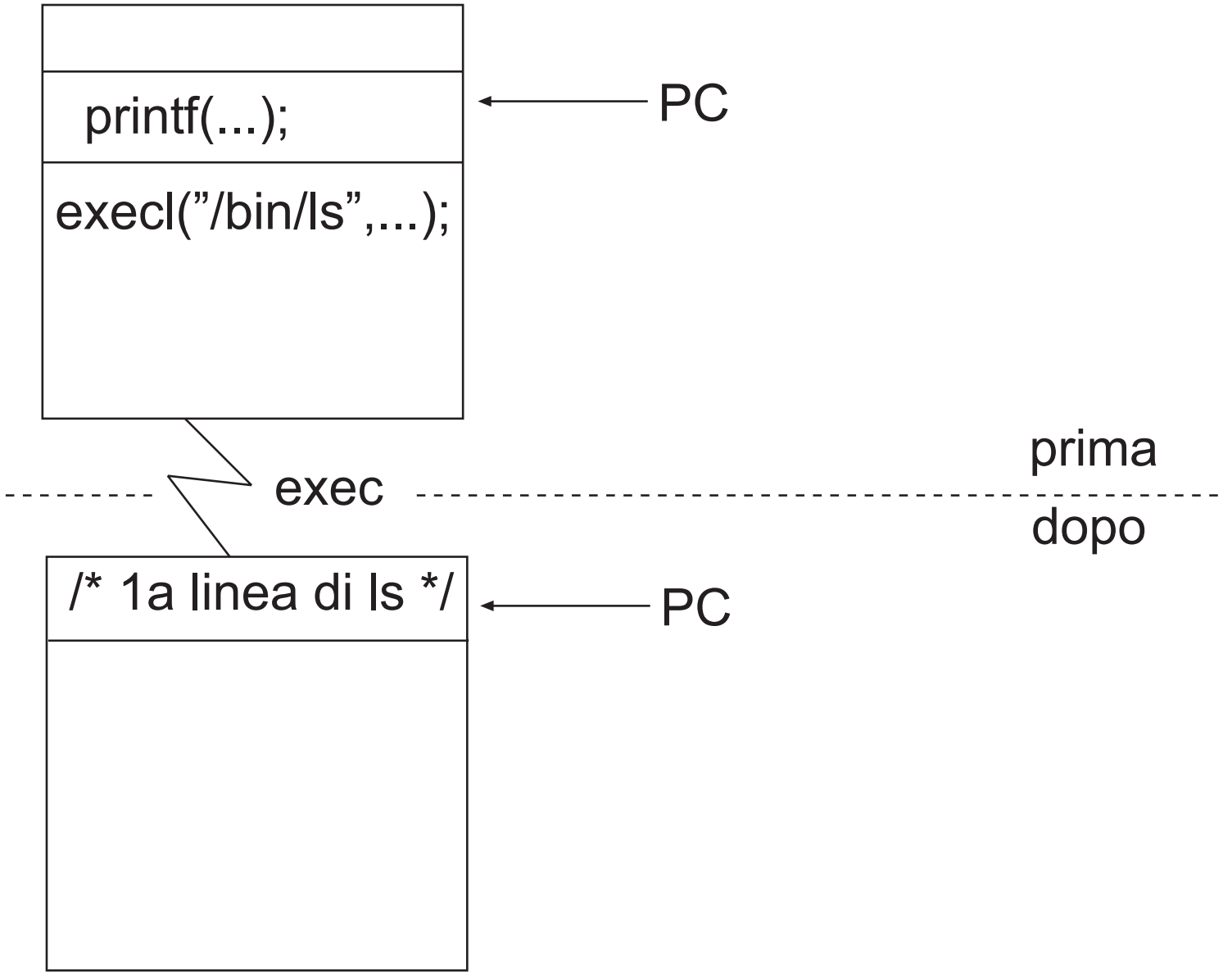
- L'argomento path è l'eseguibile che si vuole lanciare
- Gli argomenti successivi sono gli argomenti da riga di comando che si vogliono passare al programma, terminati da un puntatore nullo.
- execl() elimina il programma originale sovrascrivendolo con quello passato come parametro. Quindi le istruzioni che seguono una chiamata a execl() verranno eseguite soltanto in caso si verifichi un errore durante l'esecuzione di quest'ultima ed il controllo ritorni al chiamante.

La funzione execl()

Esempio

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

int main()
{
    printf("Esecuzione di ls...\n");

    execl("/bin/ls", "ls", "-l", NULL);

    perror("La chiamata di execl ha generato un errore");
    return 1;
}
```

Esempio di utilizzo di `execl`



Utilizzo combinato di `fork` e exec

L'utilizzo combinato di `fork()` per creare un nuovo processo e di `exec()` per lanciare nel processo figlio l'esecuzione di un nuovo programma è uno dei pilastri di UNIX.

Utilizzo combinato di fork e exec

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

int main()
{
  pid_t pid = fork();

  switch(pid) {
    case -1:
      perror("fork() failed");
      return 1;
    case 0:
      printf("Esecuzione di ls...\n");
      execl("/bin/ls", "ls", "-l", NULL);

      perror("exec failed");
      return 1;
    default:
      wait(NULL);
      printf("ls completed\n");
      return 0;
  }
}
```

L'ambiente di un processo

L'ambiente di un processo è un insieme di valori che ogni processo eredita dal padre.

- Viene rappresentato in C come un array di puntatori a stringhe, terminato da un puntatore nullo.
- Ogni puntatore (che non sia quello nullo) punta ad una stringa della forma *NOME=valore*
- L'ambiente del processo viene passato attraverso un **terzo** parametro alla funzione `main()`
- Lo stesso array si trova anche nella variabile globale `environ`:

```
extern char **environ;
```

L'ambiente di un processo

Esempio

```c
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
  while(*envp) {
    printf("%s\n",*envp);
    ++envp;
  }

  return 0;
}
```

L'ambiente di un processo

La funzione `getenv()`

Il contenuto dell'array `environ` è un po' troppo grezzo per la maggior parte degli utilizzi. Esistono alternative più semplici alla manipolazione diretta di `environ`;

```
char *getenv(const char *name);
int setenv(char *name, char *value, int overwrite);
```

L'ambiente di un processo

Esempio di uso della funzione getenv()

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

int main(int argc, char **argv, char **envp)
{
    if(argc <= 1) {
      while(*envp) {
        printf("%s\n",*envp);
        ++envp;
      }
    } else {
      printf("%s=%s\n", argv[1], getenv(argv[1]));
    }

    return 0;
}
```

L'ambiente di un processo

Influenzare l'ambiente dei processi figli

L'ambiente di ogni processo è un dato **privato** del processo:

- L'ambiente di default di un processo coincide con quello del processo padre.
- Solo il processo stesso lo può leggere o modificare.
- Ma il padre può decidere arbitrariamente l'ambiente del figlio **prima** della chiamata ad exec()
- Per specificare un nuovo ambiente è necessario usare una delle due varianti seguenti della famiglia exec:

```
int execle(const char *path, arg1, ..., argn, NULL, const char **envp);
int execve(const char *path, const char **argv, const char **envp);
```

passando in envp l'ambiente desiderato.

L'ambiente di un processo

Esempio di uso di execve()

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

int main()
{
  char *argv[2] = { "env2", NULL };
  char *envp[3] = { "var1=valore1", "var2=valore2", NULL };

  execve("./env2", argv, envp);
  perror("execve fallita");

  return 1;
}
```

Lanciare un processo ricercando nel PATH

La variabile d'ambiente `PATH` viene solitamente usata dalla shell per individuare l'eseguibile corrispondente ad un comando:

- Le chiamate `execv`/`execl`/ecc... non ricercano nel `PATH`: è necessario specificare un path completo.
- Le funzioni `execvp()`/`execlp()` funzionano come le altre ma cercano l'eseguibile nel `PATH`

La famiglia exec()

Per riassumere

Diverse varianti di execve() utili:

```
int execl(const char *name, ...);
int execv(const char *name, const char **argv);
int execlp(const char *name, ...);
int execvp(const char *name, const char **argv);
int execle(const char *name, ..., /* envp */);
int execve(const char *name, const char **argv, const char *envp);
```

Rispettivamente:

- Argomenti sulla riga di comando passati alla funzione, senza enviroment
- Argomenti sulla riga di comando passati come array, senza environment
- Argomenti sulla riga di comando passati alla funzione, senza enviroment, ricercando nel path
- Argomenti sulla riga di comando passati come array, senza environment, ricercando nel path
- Argomenti sulla riga di comando passati alla funzione, con enviroment
- Argomenti sulla riga di comando passati come array, con environment

Current working directory e root directory

Ad ogni processo sono associate due **directory**:

- La **current working directory** è la directory corrente, con cui siamo abituati a lavorare anche quando si opera con la shell.
- La **root directory** è la directory che il processo vede come directory radice dell'albero del file system, ovvero quella che per lui corrisponde al percorso `/`.
- Le due funzioni seguenti permettono di cambiare queste due directory per il processo corrente:

```
#include <unistd.h>

int chdir(const char *path);
int chroot(const char *path);
```

- **Attenzione**: Cambiare directory di lavoro è un'operazione molto comune, mentre cambiare la radice è un'operazione utile e necessaria in pochissimi casi molto particolari.

User ID e Group ID

Il sistema di permessi di UNIX si basa sull'associazione di un ID utente/gruppo ad ogni processo.

- Processi con UID uguale a zero hanno privilegi di utente **root**.
- Ogni file ha il proprio ID utente e gruppo, e i propri permessi di accesso.
- Se l'UID e il GID del processo corrispondono con i permessi di accesso del file, il processo può accedere al file.
- Due funzioni permettono di conoscere il proprio UID/GID:

```
uid_t getuid();
gid_t getgid();
```

- UID e GID del processo si possono cambiare, con la funzione `setuid()/setgid()` solo **abbassando** i propri privilegi, ossia un processo privilegiato può declassarsi ad un utente non privilegiato.

User ID e Group ID

Esempio:

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

int main() {
  uid_t uid = getuid();
  gid_t gid = getgid();

  printf("UID=%d, GID=%d\n", uid, gid);

  return 0;
}
```

User ID e Group ID Effettivi

Se un processo può solo abbassare i propri privilegi, allora come sono implementati i comandi **su**, **sudo**, ecc. . . ?

- Ad ogni processo sono associati un **real** UID ed un **real** GID che coincidono con quelli dell'utente che ha lanciato il processo.
- Esistono però l'**effective** UID e l'**effective** GID, che determinano effettivamente i privilegi del processo. Nella maggior parte dei casi essi coincidono i real ID.
- Tuttavia se il file eseguibile di un programma ha attivo il bit dei permessi noto come Set UID (Set GID), allora, quando sarà invocato con una chiamata exec, l'effective UID (effective GID) del processo sarà quello del proprietario del file e non quello dell'utente che lo ha lanciato.
- I processi figli ereditano l'effective UID/GID.

Esempio: implementazione del comando su

Ora implementeremo una semplice versione del comando su.

Il nostro programma esegue il comando che gli viene passato sulla riga di comando (compresi tutti gli argomenti), con l'identità dell'utente passato come primo argomento:

```
$ ./su utente comando arg1 arg2 arg3
```

Il sorgente completo è allegato alle slide della lezione.

Esempio: implementazione del comando su

Le funzioni `getpwuid()/getpwnam()`

Usiamo due chiamate di sistema particolari per ottenere l'UID e il GID dell'utente chiamato per nome:

```
struct passwd { // in <sys/types.h>
  // ...
  uid_t pw_uid;
  gid_t pw_gid;
  // ...
};

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

Esempio: implementazione del comando su

Conoscendo l'UID e il GID dell'utente di cui vogliamo prendere le sembianze, possiamo cambiare il nostro EUID/EGID:

```
seteuid(new_uid);
setegid(new_gid);
```

Esempio: implementazione del comando su

La chiamata a execvp

La riga di comando ci fornisce il nome dell'utente, e direttamente i parametri per la funzione execvp():

```
char *cmd = argv[2];
char **argvcmd = argv + 2;

execvp(cmd, argvcmd);
```

Esercizi

Esercizi

1. Scrivere un programma C chiamato `file_exec`, che riceva sulla riga di comando:
   - Il nome di un file.
   - Un comando da eseguire completo di argomenti, tra cui eventualmente un argomento costituito da un solo simbolo `'@'`.

   Il programma deve leggere il file riga per riga, eseguendo il comando specificato nella riga di comando una volta per ogni riga, sostituendo ad ogni occorrenza di `'@'` nella linea di comando, il contenuto della riga corrente del file.

   Esempio:

```
$ cat example.txt
file1
file2
file3
$ ./file_exec example.txt cat @
contenuto dei file...
```

Corso di Laboratorio di Sistemi Operativi

Lezione 15

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Comunicazione tra processi: pipe

Comunicazione tra processi: pipe

Affinché due processi possano cooperare, è spesso necessario che **comunichino** fra loro dei dati.

- Una prima possibile soluzione a questo problema consiste nell'utilizzo condiviso dei file (e.g., leggendo e scrivendo in un file comune). Tuttavia tale approccio risulta inefficiente; inoltre vi è la possibilità che si verifichino dei problemi di contesa della risorsa condivisa.
- UNIX, per risolvere il problema, mette a disposizione una primitiva, detta **pipe**, che consiste in un canale **unidirezionale** di comunicazione che collega un processo ad un altro, generalizzando il concetto di file.
- È possibile inviare dati in una pipe attraverso la system call `write` e leggere dalla pipe attraverso la system call `read`, e in generale quasi tutte le funzioni di I/O su file.

Il **pipelining** di due comandi nella shell:

```
$ ls -l | less
```

è implementato tramite questo meccanismo.

Creare una pipe

Per creare una pipe, esiste l'apposita system call:

```
#include <unistd.h>
int pipe(int *filedes);
```

dove `filedes` deve puntare ad un array di due interi, che conterranno i file descriptor dei due capi della pipe:

- Il primo, `filedes[0]`, serve a leggere dalla pipe.
- Il secondo, `filedes[1]`, serve a scrivervi.
- I dati scritti da un capo vengono letti (in modo FIFO) dall'altro capo.

Passare i capi della pipe ai processi figli

Perché una pipe sia utile, è necessario in qualche modo passare uno dei due capi ad un processo figlio, in modo da poter comunicarci:

- I file descriptor restano aperti dopo la chiamata `fork()`: padre e figlio possono comunicare direttamente.
- È possibile **redirigere** un file descriptor aperto, ad esempio lo standard input, su un altro, ad esempio un capo della pipe. La redirezione resta in piedi dopo una chiamata a `execv()`.

Esempio

Creazione di una pipe

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

#define MSGSIZE 14

int main() {
  int pipes[2] = { };
  if(pipe(pipes) == -1) {
    perror("pipe call");
    return 1;
  }

  char msg[MSGSIZE] = { };
  pid_t pid = fork();
  switch(pid) {
    case -1:
      perror("fork call");
      return 2;
    case 0:            // processo figlio
      close(pipes[0]); // chiude l'altro capo
      write(pipes[1], "Hello, world!", MSGSIZE);
      break;
    default:           // processo padre
      close(pipes[1]); // chiude l'altro capo
      read(pipes[0], msg, MSGSIZE);
      printf("%s\n", msg);
      wait(NULL);
  }
  return 0;
}
```

Redirezione di un file descriptor

Per **redirigere** un file descriptor verso un altro è disponibile la chiamata di sistema `dup2()`:

```
#include <unistd.h>
int dup2(int old_fd, int new_fd);
```

Dopo la chiamata, il file descriptor `new_fd` punterà alla stessa risorsa puntata dal file descriptor `old_fd` (che doveva essere già aperto). Se `new_fd` è già in uso al momento della chiamata, viene chiuso e riaperto prima del suo riutilizzo.

Quindi questo codice:

```
int fds[2] = { };
pipe(fds);
dup2(fds[1], 1);
```

crea una pipe e ne collega il capo di scrittura al file descriptor dello standard output.

Esempi

Due esempi sono allegati alle slide:

- Il programma `lsgrep.c` esegue l'equivalente del comando:

```
$ ls -l | grep ⟨pattern⟩
```

  dove ⟨pattern⟩ viene dato da riga di comando.

- Il programma `guardieladri.c` implementa un semplice gioco in cui i due giocatori sono gestiti da due processi differenti, che comunicano tramite pipe con un terzo processo che gestisce lo stato del gioco.

Gioco guardie e ladri

Il programma `guardieladri.c` è un esempio di utilizzo delle pipe per coordinare e far comunicare più processi.

- Il gioco è chiamato "guardie e ladri":
  - il ladro (rappresentato dal carattere `$`) verrà mosso in modo casuale sullo schermo del terminale (80×24) dal computer
  - la guardia (rappresentata dal carattere #) verrà mossa dall'utente tramite i tasti freccia;
  - il gioco terminerà quando la guardia ed il ladro si incontrano.
- Nell'implementazione usiamo tre processi:
  - un processo principale, responsabile della visualizzazione e del controllo dell'evento in cui la guardia ed il ladro si incontrano
  - un processo figlio che aggiorna la posizione del ladro
  - un processo figlio che aggiorna la posizione della guardia
- Il programma utilizza la libreria **ncurses** (opzione di compilazione per il linker: `-lcurses`) per la gestione del terminale, usata da tutti i programmi testuali, ma interattivi (es. editor di testo).
  - per installare la libreria nel sistema:
    `sudo apt install libncurses-dev`

Esercizi

Esercizi

1. Scrivere un programma che, dato il nome di un file sulla riga di comando, ne mostri il contenuto lanciando il comando 'cat' in modo equivalente al seguente comando da terminale:

```
$ cat < ⟨file⟩
```

   cioè redirigendo il file sullo standard input del processo figlio.
2. Scrivere un programma che, dato sulla riga di comando il nome di un file, esegua quanto segue:
   - Ogni riga del file sarà del formato:

```
comando1 arg1 ... argn | comando2 arg2 ... argn
```

   - Per ogni tale riga, il programma deve eseguire `comando1` e `comando2`, con relativi argomenti, collegando con una pipe lo standard output del primo allo standard input del secondo.

Corso di Laboratorio di Sistemi Operativi

Lezione 16

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Comunicazione tra processi: segnali

Segnali

I segnali in Unix sono un meccanismo semplice per inviare degli interrupt software ai processi.

- Solitamente sono utilizzati per gestire errori e condizioni anomale, piuttosto che per trasmettere dati.
- Un processo può ricevere segnali inviati da altri processi, o provenienti dal sistema operativo, e può:
  1. eseguire un'opportuna funzione per trattare l'errore (signal handling);
  2. bloccare il segnale;
  3. inviare il segnale ad un altro processo.
- Il comportamento di default alla ricezione della maggior parte dei segnali, è quello di terminare il processo.

Segnali

Nella tabella a fianco sono elencati i diversi tipi di segnale.

- In rosso quelli che causano una **terminazione anomala** del processo.
- Gli altri, per la maggior parte, causano una **terminazione normale**.

La differenza è che i primi non possono essere ignorati, mentre gli altri possono essere ignorati o gestiti in modo particolare.

N°	Nome	Causa
1	SIGHUP	terminal line hangup
2	SIGINT	interrupt program
3	**SIGQUIT**	quit program
4	**SIGILL**	illegal instruction
5	**SIGTRAP**	trace trap
6	**SIGABRT**	abort program (formerly SIGIOT)
7	**SIGEMT**	emulate instruction executed
8	**SIGFPE**	floating-point exception
9	**SIGKILL**	kill program
10	**SIGBUS**	bus error
11	**SIGSEGV**	segmentation violation
12	**SIGSYS**	non-existent system call invoked
13	SIGPIPE	write on a pipe with no reader
14	SIGALRM	real-time timer expired
15	SIGTERM	software termination signal
16	SIGURG	urgent condition present on socket
17	**SIGSTOP**	stop (cannot be caught or ignored)
18	SIGTSTP	stop signal generated from keyboard
19	SIGCONT	continue after stop
20	SIGCHLD	child status has changed
21	SIGTTIN	background read from control terminal
22	SIGTTOU	background write to control terminal
23	SIGIO	I/O is possible on a descriptor
24	**SIGXCPU**	cpu time limit exceeded
25	**SIGXFSZ**	file size limit exceeded
26	SIGVTALRM	virtual time alarm
27	SIGPROF	profiling timer alarm
28	SIGWINCH	Window size change
29	SIGINFO	status request from keyboard
30	SIGUSR1	User defined signal 1
31	SIGUSR2	User defined signal 2

Inviare segnali

La system call principale per inviare segnali è `kill()`:

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

int kill(pid_t pid, int sig);
```

Invia il segnale `sig` al processo con PID `pid`.

Nota: Un segnale può essere lanciato solo a processi dello stesso utente, a meno che non si abbiano privilegi di **root**.

Inviare segnali

Alternative

Altrimenti si può inviare un segnale a se stessi con `raise()`:

```
int raise(int sig);
```

La funzione `alarm()` causa la ricezione di un segnale `SIGALRM` dopo l'intervallo di tempo specificato:

```
unsigned int alarm(unsigned int secs);
```

Inviare segnali

Il comando `kill`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

int main(int argc, char **argv)
{
  if(argc < 2) {
    fprintf(stderr, "Specificare il PID di un processo\n");
    return 1;
  }

  char *endptr = NULL;
  pid_t pid = strtoll(argv[1], &endptr, 10);
  if(*endptr != 0) {
    fprintf(stderr, "Specificare il PID di un processo\n");
    return 1;
  }

  if(kill(pid, SIGKILL) == -1) {
    fprintf(stderr, "Impossibile uccidere il processo %d: %s\n", pid,
            strerror(errno));
    return 2;
  }

  return 0;
}
```

Signal handling

Un segnale si può **gestire**, eseguendo una funzione ogni volta che viene ricevuto, in modo simile agli interrupt handler hardware.

```
#include <signal.h>
typedef void (*sighandler_t)(int); // typedef per un puntatore a funzione
sighandler_t signal(int sig, sighandler_t handler);
```

La funzione `signal()` registra la funzione puntata da `handler` come gestore del segnale `sig`.

La funzione `signal()` è una versione semplificata di `sigaction()`, funzione più flessibile che permette di:

- Impostare delle maschere per bloccare determinati segnali
- Passare informazioni aggiuntive all'handler
- Impostare varie opzioni sull'interazione del segnale con le system call di I/O.

Esempio di signal handling

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

void ahah(int x) {
  printf("ahah you cannot terminate me!\n");
}

int main()
{
  signal(SIGINT, ahah);
  signal(SIGTERM, ahah);
  signal(SIGKILL, ahah); // Funzionera'?

  while(1) {
    printf("Hey apple!\n");
    sleep(1);
  }

  return 0;
}
```

Comunicazione tra processi: socket

Socket

I **socket** (presa/spinotto) sono un meccanismo di comunicazione interprocesso **bidirezionale**, a differenza delle pipe.

- Come le pipe, una volta configurato il meccanismo si ottengono dei **file descriptor** su cui si può scrivere e leggere normalmente.
- I socket adottano una filosofia **client**/**server**.
- Il processo server **ascolta** su un indirizzo, mentre il/i processo/i client si **connettono** a tale indirizzo.
- È lo stesso modello di funzionamento delle **comunicazioni in rete**. Infatti, oltre che la comunicazione tra processi in esecuzione sullo stesso sistema, l'interfaccia a socket è quella usata per la comunicazione via rete.
- Esistono quindi più **domini**:
  - i socket UNIX-domain, per la comunicazione locale.
  - i socket Internet-domain, per la comunicazione IPv4/IPv6.
  - altri (Novell, AppleTalk, ...) caduti in disuso.

Utilizzo dei socket

L'utilizzo dell'interfaccia a socket è più complesso di quello delle pipe.
Molte funzioni e chiamate di sistema entrano in gioco.

`Funzione`	Scopo	Usato da
`socket()`	Crea il file descriptor di un capo della connessione	entrambi
`bind()`	Lega il socket ad un indirizzo specifico	server
`listen()`	Marca il socket come passivo (per accettare connessioni)	server
`accept()`	Accetta una connessione in arrivo	server
`connect()`	Connette un socket ad un altro socket in ascolto	client

Utilizzo dei socket

Funzione `socket()`

La funzione `socket()` va chiamata sia dal client che dal server per aprire un file descriptor che verrà usato nelle operazioni successive.

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

I valori degli argomenti specificano il dominio (locale, internet, ...) e il protocollo di comunicazione utilizzato (nel caso di internet).

Per i socket di dominio UNIX, l'uso della funzione è il seguente:

```
int fd = socket(AF_LOCAL, SOCK_STREAM, 0);
```

Utilizzo dei socket

Funzione `bind()` e `listen()`

La funzione `bind()` lega un socket ad un indirizzo, per il successivo ascolto, che viene abilitato dalla funzione `listen()`:

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

struct sockaddr_un {
    short sa_family;     // = AF_LOCAL
    char  sun_path[108]; // Indirizzo del socket
};

int bind(int sockfd, const struct sockaddr *addr, size_t addr_len);
int listen(int sockfd, int queue_size);
```

Il parametro `queue_size` di `listen()` è il numero massimo di client che possono restare in attesa di una connessione.

Nei socket locali l'indirizzo è il nome di un **file**. Il file viene creato da `bind()`, ma non si tratta di un file regolare, bensì di un **socket file**.

Uso dei socket

Funzioni `connect()` e `accept()`

Dopo la chiamata a `socket()`, un processo può connettersi come client ad un socket su cui esista un processo in ascolto.

```
int connect(int sockfd, const struct sockaddr *address, size_t addr_len);
```

La comunicazione si instaura effettivamente quando il server chiama `accept()`:

```
int accept(int sockfd, struct sockaddr *address, size_t *addr_len);
```

`accept()` blocca il processo finché un client non si connette:

- restituisce un **nuovo** file descriptor collegato all'altro capo della comunicazione.
- Il vecchio file descriptor può contemporaneamente essere utilizzato per accettare un'altra connessione (solitamente forkando il processo)
- Da questo momento i due processi possono comunicare leggendo e scrivendo dati sui file descriptor a loro disposizione.

Leggere e scrivere da un socket

Una volta ottenuti i file descriptor dei due capi del socket, ci si può leggere e scrivere come su qualsiasi altro file. Esistono però delle funzioni più flessibili, specializzate ad operare su socket:

```
ssize_t send(int fd, const void *buffer, size_t length, int flags);
ssize_t recv(int fd, void *buffer, size_t length, int flags);
```

Le due funzioni operano come rispettivamente `write()` e `read()`, ma forniscono il parametro aggiuntivo `flags`, con il quale si possono specificare opzioni aggiuntive.

Esempio

Un programma di esempio di uso dei socket è allegato alle slide.

- Il progetto è costituito da due programmi: un client e un server
- Il server resta in ascolto di connessioni, e rispedisce al client qualsiasi dato ricevuto, ma trasformato in maiuscolo.
- Il client legge righe di testo dallo standard input e le invia al server, stampando la risposta.
- Il server gestisce connessioni multiple creando un processo per ogni client, in modo che il processo padre possa tornare ad ascoltare mentre il figlio gestisce il singolo client.
- In allegato trovate il server, il client è per esercizio.

Test del funzionamento del server

1. Compilazione: `make`
2. Esecuzione del server: `./upperserver`
3. Connessione al server (2 alternative):
   - 3.1 usando il comando `nc` (premere Ctrl-C per uscire):

```
nc -U /tmp/upperserver.socket
 Benvenuto all'UpperServer 1.0!
prova...
PROVA...
```

   - 3.2 usando il comando `socat` (premere Ctrl-D per uscire):

```
socat - UNIX-CONNECT:/tmp/upperserver.socket
 Benvenuto all'UpperServer 1.0!
prova 2 ...
PROVA 2 ...
```

Esercizi

Esercizio

1. Scrivere il programma client per connettersi al programma `upperserver`. Il programma deve:
   - Connettersi al server e stampare il messaggio di benvenuto che viene ricevuto. Il server spedisce un dato di tipo `int` con la lunghezza del messaggio (compreso il terminatore nullo), seguito dal messaggio stesso.
   - Leggere righe di testo dallo standard input, inviarle al server, e ricevere la risposta, che sarà della stessa lunghezza, stampandola in output.

Corso di Laboratorio di Sistemi Operativi

Lezione 17

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Multithreading

Parte 1

I thread

Nei sistemi operativi moderni, ogni processo può contenere uno o più flussi di esecuzione, chiamati **thread**.

- Ogni thread viene eseguito in modo indipendente, ma condivide la maggior parte delle risorse del processo con gli altri thread:
  - Codice
  - Spazio degli indirizzi
  - File aperti, handler dei segnali, ...
- I singoli thread mantengono invece separate le risorse legate al proprio flusso di esecuzione:
  - Program counter
  - Registri della CPU
  - Stack

I thread

Usi e motivazioni

I thread sono la primitiva di base per ottenere **concorrenza** e **parallelismo** all'interno di un singolo processo.

- Oggigiorno il multithreading è un paradigma fondamentale data l'evoluzione degli attuali sistemi multicore.
- Avendo un address space comune, la creazione dei thread ed il cambio di contesto sono notevolmente meno costosi rispetto ai corrispondenti meccanismi che riguardano i processi.
- Inoltre, la comunicazione tra thread è molto efficiente, senza la necessità di meccanismi di comunicazione ad-hoc.
- Anche nei sistemi con un'unica CPU si hanno dei benefici. Infatti, è possibile sfruttare i tempi di latenza delle operazioni di I/O di un thread per eseguirne nel frattempo un altro.

I thread

Problematiche

La programmazione multithread presenta delle problematiche che la rendono particolarmente difficoltosa:

- La concorrenza spesso richiede che i thread accedano a dati **condivisi**, ma ciò deve avvenire evitando sempre **race condition**, ovvero scritture contemporanee sulle stesse locazioni di memoria.
- L'interazione tra thread va quindi sempre sincronizzata tramite opportune primitive, che vanno usate in modo corretto.
- Il **nondeterminismo** dato dall'imprevedibilità dello scheduler rende molto difficile l'individuazione e la risoluzione di bug dovuti all'errata sincronizzazione di thread. Occorre quindi molta disciplina nella scrittura del codice.

L'interfaccia POSIX per i thread

La libreria `pthread` (POSIX thread), è l'interfaccia standard nei sistemi UNIX per la creazione e la manipolazione di thread.

Per utilizzare le funzioni `pthread` è necessario linkare l'apposita libreria:

```
$ clang -lpthread programma.c -o programma
```

Creare un thread

La funzione principale per creare un thread è `pthread_create()`:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void * (*start_routine)(void *), void *arg);
```

Significato degli argomenti, in ordine:

1. Un puntatore ad una variabile di tipo `pthread_t` che funge da **handler** per rappresentare il thread creato.
2. Un puntatore ad una struttura contenente opzioni e configurazioni aggiuntive per il comportamento del thread.
3. Un puntatore ad una funzione che verrà eseguita dal nuovo thread. La funzione deve accettare e restituire un `void*`.
4. Un puntatore a `void` che verrà passato come argomento alla funzione lanciata dal nuovo thread.
5. Il valore di ritorno in assenza di errori è `0`.

Creare un thread

Esempio

```c
#include <stdio.h>
#include <pthread.h>

void *print_msg(void *ptr);

int main()
{
  char msg1[] = "Thread 1";
  char msg2[] = "Thread 2";

  pthread_t thread1, thread2;

  pthread_create(&thread1, NULL, print_msg, (void *)msg1);
  pthread_create(&thread2 ,NULL, print_msg, (void *)msg2);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}

void *print_msg(void *ptr)
{
  char *arg = (char *) ptr;

  while(1)
    printf("%s\n", arg);

  return NULL;
}
```

Attributi di un thread

Il parametro di tipo `struct pthread_attr_t` di `pthread_create()` è detto **thread attribute object**, e specifica diversi parametri che influenzano la vita del thread.

- L'oggetto va inizializzato e distrutto con le apposite funzioni:

```
struct pthread_attr_t attr;
pthread_attr_init(&attr); // inizializza a valori di default
// impostare i parametri in attr con le apposite funzioni
pthread_create(&thread, &attr, func, arg);
pthread_attr_destroy(&attr); // dismettere l'attributes object
```

- Il valore di ogni singolo attributo X si imposta con l'apposita funzione `pthread_attr_setX()`.

Attributi di un thread

Tra gli attributi che è possibile impostare troviamo:

Attributo	`pthread_attr_setX`	Significato
Scheduling policy	`schedpolicy`	Le politiche di scheduling relative al thread. Può essere lasciato il valore di default `SCHED_OTHER` oppure richiesto uno scheduling **round robin** o **FIFO**.
Sched. Inheritance	`inheritsched`	Specifica se il thread eredita le policy di scheduling del thread padre.
Contention Scope	`scope`	Specifica se il thread compete, per l'uso della CPU, con tutti gli altri thread/processi del sistema o solo quelli del processo (`PTHREAD_SCOPE_SYSTEM` oppure `PTHREAD_SCOPE_PROCESS`).
Scheduling priority	`schedparam`	Specifica la priorità associata al thread.

Terminazione di un thread

L'esecuzione di un thread termina quando la sua funzione principale ritorna, o quando viene chiamata la funzione `pthread_exit()`:

```
void pthread_exit(void *retval);
```

La funzione `pthread_join()` permette ad un thread di aspettare la fine di un altro di cui abbia l'handler:

```
int pthread_join(pthread_t th, void **value_ptr);
```

La funzione blocca il thread corrente in attesa della terminazione del thread identificato da `th`. Il valore di ritorno del thread viene scritto in `*value_ptr`.

Parallelismo tramite threads

Esempio `parallel_max.c`

Nel file di esempio in allegato trovate un esempio d'uso del multithreading per effettuare una computazione in **parallelo** su più threads.

- Il programma `parallel_max.c` legge il contenuto di un file in modo binario, lo interpreta come un array di numeri interi, e stampa il numero più alto tra quelli letti.
- Il calcolo del numero più alto avviene in parallelo su due thread the operano su due diverse metà dell'array.

Nota: Non è necessario sincronizzare l'accesso dei diversi thread alla memoria condivisa perché tali accessi avvengono sempre esclusivamente in lettura (e comunque su parti separate dell'array).

Documentazione

- Nel libro di testo consigliato per la parte del corso sulla programmazione di sistema non c'è una trattazione dei thread.
- È possibile reperire dei tutorial sul Web in modo da integrare le informazioni disponibili nelle man page.
- Alcuni tutorial (in Inglese) sono disponibili agli indirizzi seguenti:
  1. `http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`
  2. `https://hpc-tutorials.llnl.gov/posix/`
- Libri per approfondimenti:
  1. David R. Butenhof. *Programming with POSIX Threads*, 1997, ISBN: 978-0201633924, Addison-Wesley Professional.
  2. Dick Buttlar, Jacqueline Proulx Farrell, Bradford Nichols. *PThreads Programming: A POSIX Standard for Better Multiprocessing (A Nutshell handbook)*, 1996, ISBN: 978-1565921153, O'Reilly Media.

Esercizi

Esercizi

1. Modificare l'esempio `parallel_max.c` in modo da permettere all'utente di specificare il numero di thread in cui si vuole suddividere il lavoro.
2. Modificare l'esempio `upperserver.c` della lezione 16 per fare in modo che le diverse connessioni in entrata vengano gestite da diversi thread dello stesso processo invece che da più processi.

Corso di Laboratorio di Sistemi Operativi

Lezione 18

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Multithreading

Parte 2

Sincronizzazione di thread

In applicazioni con più thread concorrenti è spesso necessario che i diversi thread **comunichino** tra loro:

- La comunicazione può avvenire direttamente (senza pipe, socket, ecc.) perché i thread condividono lo stesso spazio di indirizzi.
- Tuttavia, l'accesso a risorse condivise (anche semplici variabili) va sincronizzato per evitare **race condition**.
- Il sistema operativo fornisce una serie di primitive per la sincronizzazione:
  - Mutex
  - Condition variables
  - Semafori
- La sincronizzazione è necessaria spesso anche in assenza di multithreading, nella gestione di **segnali**.

Mutex

Il Mutex (da **mut**ual **ex**clusion) è forse il meccanismo di sincronizzazione più comune:

- È un meccanismo utile per proteggere strutture dati condivise da modifiche concorrenti (es., in modo da implementare sezioni critiche).
- Un mutex può essere **bloccato** (locked), oppure **libero** (unlocked).
- Un thread viene **sospeso** se prova a bloccare un mutex già bloccato, e riprende l'esecuzione quando il mutex si libera.

Mutex

Utilizzo

Per utilizzare un mutex va dichiarata una variabile di tipo `pthread_mutex_t` e inizializzata in modo particolare come segue:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Successivamente il codice compreso fra le chiamate `pthread_mutex_lock()` e `pthread_mutex_unlock()` potrà essere eseguito soltanto da un thread alla volta:

```
pthread_mutex_lock(&mutex);
// sezione critica
pthread_mutex_unlock(&mutex);
```

Se un thread non riesce a bloccare un mutex (perché già bloccato da un altro thread), viene sospeso fintanto che il thread che lo sta bloccando non lo rilascia.

Mutex

Problematiche

L'uso di mutex può avere conseguenze di cui bisogna tener conto:

- L'utilizzo errato può portare a **deadlock**: situazione in cui due o più thread rimangono bloccati indefinitamente aspettando l'uno un'azione dell'altro.
- L'abuso di mutex può degradare le performance del programma:
  - Bloccare e sbloccare i mutex comporta due chiamate di sistema, per cui almeno due context switch.
  - Se le sezioni critiche sono troppo numerose o troppo pesanti il grado di parallelismo del programma diminuisce, e non si riesce a sfruttare il numero di CPU a disposizione.
  - Occorre quindi progettare il codice in modo che l'esecuzione di sezioni critiche sia ridotto al minimo.

Mutex

Esempio: sincronizzazione dell'accesso ad un contatore

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

int n = 0;
pthread_mutex_t mutex =
  PTHREAD_MUTEX_INITIALIZER;

void *count(void *);

int main() {
  pthread_t th1, th2;
  pthread_create(&th1, NULL, count, NULL);
  pthread_create(&th2, NULL, count, NULL);

  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  printf("n: %d\n", n);

  return 0;
}
```

```c
void *count(void *arg) {
  int local_n = 0;
  do {
    usleep(500000);
    pthread_mutex_lock(&mutex);
    n += 1;
    local_n = n;
    pthread_mutex_unlock(&mutex);
    printf("n: %d\n", local_n);
  } while(local_n < 42);

  return NULL;
}
```

Vedere `mutexnoglobals.c` per una versione senza var. globali.

Mutex

Esempio: gestione di un segnale

Questa è la versione corretta dell'esempio `annoying.c` della lezione 16.

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>

int signaled = 0;
pthread_mutex_t m =
  PTHREAD_MUTEX_INITIALIZER;

void handler(int x) {
  pthread_mutex_lock(&m);
  signaled = 1;
  pthread_mutex_unlock(&m);
}
```

```c
int main() {
  signal(SIGINT,  handler);
  signal(SIGTERM, handler);

  while(1) {
    pthread_mutex_lock(&m);
    if(signaled)
      printf("You cannot terminate me!\n");
    signaled = 0;
    pthread_mutex_unlock(&m);

    printf("Hey apple!\n");

    sleep(1);
  }
  return 0;
}
```

Condition variables

Una **condition variable** è una primitiva di sincronizzazione che viene utilizzata per sospendere l'esecuzione di un thread in attesa che si verifichi un certo evento.

- Un thread può **sospendersi** su una condition variable, per aspettare che avvenga qualcosa.
- Un altro thread può **risvegliare** uno o più dei thread in attesa per segnalare.

Condition variables

Utilizzo

Per utilizzare una condition variable va dichiarata una variabile di tipo `pthread_cond_t` e inizializzata in modo particolare come segue:

```
pthread_cond_t var = PTHREAD_COND_INITIALIZER;
```

Successivamente è possibile:

- attendere la segnalazione della variabile con la funzione `pthread_cond_wait()`.
- risvegliare uno o tutti i thread in attesa su una variable con `pthread_cond_signal()` o `pthread_cond_broadcast()`, rispettivamente.

Condition variables

Perché il mutex?

Per bloccarsi su una condition variable, si usa la funzione:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Come si può vedere, è necessario fornire anche un mutex, che va precedentemente **bloccato**. Perché?

- La condition variable è un meccanismo per segnalare un evento, ma non c'è un meccanismo per comunicare dati oltre al solo evento.
- Servirà un mutex per sincronizzare l'accesso ai dati condivisi utilizzati per tale comunicazione.
- Il mutex assicura inoltre che un thread non "perda" una segnalazione mentre ne sta gestendo un'altra.

Condition variables

Perché il mutex?

Per bloccarsi su una condition variable, si usa la funzione:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Come funziona?

- La funzione `pthread_cond_wait()` si aspetta un mutex già precedentemente **bloccato**.
- Atomicamente **sblocca** il mutex e si mette in attesa.
- Quando la condition variable viene segnalata, la funzione al risveglio torna a **bloccare** il mutex. Il thread non si risveglia finché non trova il mutex libero.
- Anche il thread segnalante deve bloccare e sbloccare lo stesso mutex prima e dopo la segnalazione.
- **Attenzione**: Potrebbero verificarsi **risvegli spuri**: occorre controllare che l'evento segnalato sia successo davvero.

Condition variables

Comune schema di utilizzo

Thread segnalante:

```
pthread_mutex_lock(&mutex);

write_data();
event = 1;
pthread_cond_signal(&cond);

pthread_mutex_unlock(&mutex);
```

Thread in attesa:

```
pthread_mutex_lock(&mutex);

while(!event)
  pthread_cond_wait(&cond, &mutex);
read_data();

pthread_mutex_unlock(&mutex);
```

Esempio

Guardie e ladri multithread

Il file `guardieladri_th.c` allegato contiene un esempio completo di utilizzo di thread in un'applicazione **concorrente**.

- Equivalente all'esempio `guardieladri.c` visto nella Lezione 15, ma con l'utilizzo di thread multipli, comunicanti tramite una variabile condivisa, invece di processi multipli comunicanti tramite pipe.
  - Un thread gestisce l'input dell'utente spostando la guardia.
  - Un altro gestisce il ladro, scegliendo casualmente le sue mosse.
  - Il thread principale riceve le mosse da ognuno e le mostra a schermo.
- Un mutex viene utilizzato per proteggere l'accesso alla variabile condivisa utilizzata per comunicare le mosse, e una condition variable per segnalare al thread di controllo la presenza di una nuova mossa.