# Esercitazione [1]

# **Debugging e Processi in C**

Leonardo Aniello <aniello@dis.uniroma1.it>
Daniele Cono D'Elia <delia@dis.uniroma1.it>
Federico Lombardi <lombardi@dis.uniroma1.it>

Sistemi di Calcolo - Secondo modulo (SC2)
Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2015-2016

# Sommario

1. Debugging
   - gdb
   - valgrind
2. Utilizzo delle struct
3. Creazione di processi

# 1. Errori

*Esempio di codice errato*

- sorgente: `nullPtr.c`
- compilazione: `gcc -o nullPtr nullPtr.c`
- esecuzione: `./nullPtr ciao!`
- output: `Segmentation fault`

Un *segfault* si verifica quando un programma tenta di accedere ad una posizione di memoria alla quale non gli è permesso accedere o in un modo che non gli è concesso (e.g., scrittura su una zona di memoria read-only).

- Come risalire alla causa di un errore? Debugging!

# 1. Debugging con gdb (1/2)

- Per ottimizzare il debug, durante la compilazione:
  - inserire i simboli di debugging: `gcc` **`-g`**
  - non abilitare le ottimizzazioni! (default: `-O0`)

- Ricompiliamo il nostro esempio:

```
gcc -g -o nullPtr nullPtr.c
```

- Debugging in gdb:

```
> gdb ./nullPtr
> run ciao!  // passo argomenti al programma
> bt         // mostra stack trace dopo errore
> frame 5    // vai a stack frame del main (#5 da bt)
> print p    // stampa il valore della variabile p
```

# 1. Debugging con gdb (2/2)

- I breakpoint consentono di fermare l'esecuzione ed ispezionare lo stato di un programma prima di un errore!
  1. Inseriamo uno o più breakpoint e lancio il programma

     ```
     > gdb ./nullPtr
     > break nullPtr.c:7
     > run ciao!
     ```

  2. Raggiunto un breakpoint, possiamo inserirne ulteriori

     ```
     > break nullPtr.c:10
     ```

  3. Riprendiamo l'esecuzione

     ```
     > cont
     ```

# 1. Debugging con valgrind

*Spesso si ricorre ad un memory error detector…*

- sorgente: `arrayBounds.c`
- compilazione: `gcc –g –o arrayBounds arrayBounds.c`
- esecuzione: `./arrayBounds 10`
- output: Nessun segfault, ma valore calcolato errato!

- Debugging con memcheck (valgrind):

  `valgrind ./arrayBounds 10`

  – Invalid write of size 1 at […] main (arrayBounds.c:16)
  – Invalid read of size 1 at […] main (arrayBounds.c:21)

# 1. Compilazione con ottimizzazioni

*Le ottimizzazioni possono nascondere errori del programmatore…*

- sorgente: `infiniteRec.c`
- compilazione: `gcc –o infiniteRec infiniteRec.c`
- esecuzione: `./infiniteRec`
- output: `Segmentation fault`

Dopo un certo numero di iterazioni, otteniamo un segfault come conseguenza di un pattern di ricorsione infinito.

- Ricompiliamo abilitando le ottimizzazioni:

  `gcc -O2 –o infiniteRec infiniteRec.c`

  (oppure con `–O1 –foptimize-sibling-calls`)

- Perché il programma ora non crasha???

# 1. Materiale di approfondimento

- gdb:
  - https://www.cs.cmu.edu/~gilpin/tutorial/
  - http://www.unknownroad.com/rtfm/gdbtut/gdbsegfault.html
  - http://sourceware.org/gdb/current/onlinedocs/gdb/
- valgrind:
  - http://valgrind.org/docs/manual/quick-start.html
- misc:
  - https://en.wikipedia.org/wiki/Heisenbug
  - http://www.go4expert.com/articles/reasons-segmentation-fault-c-t27220/
  - http://ismail.badawi.io/blog/2015/09/07/when-optimizations-hide-bugs/
  - http://stackoverflow.com/questions/89603/how-does-the-debugging-option-g-change-the-binary-executable
  - dispensa UNIX System Programming (dal sito del corso)

# 2. struct e typedef

```
struct book {
  char[30]  author;
  char[100] title;
  unsigned  year;
};

// static
struct book book1;
my_book.author = […]

// dynamic
struct book *book2_p =
     malloc(sizeof(
     struct book));
book2_p->author = […]
```

```
typedef struct book_s {
  char[30]  author;
  char[100] title;
  unsigned  year;
} book_t;

/* Usage examples */
book_t  book1;
printf("year %d\n",
     book1.year);

book_t* p = &book1;
// p->x same as (*p).x
printf("year %d\n",
     p->year);
```

# 2. Esercizio proposto

- Correggere il codice fornito per il calcolo della distanza euclidea tra un punto del piano cartesiano e l'origine degli assi

- sorgente: `distance.c`
- In fase di compilazione linkare la libreria math (`-lm`):
  `gcc -o distance distance.c -lm`
- esecuzione: `./distance 1.0 2.5`

# 3. Processi in C - fork e wait

```
pid_t pid;
pid = fork();
if (pid == -1) {
      // gestione errore
      exit(EXIT_FAILURE);
} else if (pid == 0) {
      // codice processo figlio
      exit(EXIT_SUCCESS);
} else {
      // codice processo padre
      int status; // per ottenere return code figlio
      wait(&status); // attende terminazione figlio
      exit(EXIT_SUCCESS);
}
```

# 3. Esercizio sui processi

- Il codice fornito in `processes.c` calcola sequenzialmente il fattoriale e il numero di fibonacci per i primi N numeri
- Sfruttare `fork()` per far calcolare:
  - processo figlio: fattoriale dei primi N numeri
  - processo padre: sequenza di Fibonacci (primi N numeri)
- Ulteriori requisiti:
  - i processi devono stampare il loro PID in ogni riga stampata in output (suggerimento: usare `getpid()` per ottenere il `pid_t` di un processo)
  - il padre deve:
    - calcolare i propri risultati (in «parallelo» al figlio)
    - attendere la terminazione del figlio
    - stampare i propri risultati

➢ sorgente: `processes.c`