Esercitazioni del Corso di Sistemi Operativi 1

dr. Andrea E. Naimoli

Università degli Studi di Trento

Dipartimento di Ingegneria e Scienza dell'Informazione

via Sommarive 14

I - 38050 Trento - Povo, Italy

Ringraziamenti

Il materiale è sostanzialmente preso da quello elaborato per il precedente A.A. da

Gabriele Oligeri e Roberto Resoli

a sua volta ricavato da quello preparato da

Alessandro Russo e Nicola Manica,
Fabrizio Stefani e Fabrizio Sannicolò,

precedenti esercitatori del corso di Sistemi Operativi 1

Informazioni generali
e
Programma delle Esercitazioni

Esercitazioni del corso di Sistemi Operativi (1/4)

Ricevimento:

labso@disi.unitn.it

Aula 6 – Vecchio Polo

Label: [LABSO-2014]

Orario esercitazioni: (**aula pc-B106**)
Giovedì 09:00 – **12:00** (Gruppo 1)
Giovedì 12:00 – **15:00** (Gruppo 2)

Esercitazioni del corso di Sistemi Operativi (2/4)

12 incontri da 3h per ciascun gruppo

Febbraio: **20 (I)**, **27 (II)**
Marzo: **13 (III), 20 (IV), 27 (V)**
Aprile: **3 (VI), 10 (VII), 17 (VIII)**
Maggio: **8 (IX), 15 (X), 22 (XI), 29 (XII)**

I: Lab-1-C
II: Lab-2-QEMU-Kernel
III: Lab-3-Syscalls
IV: Lab-4-Syscalls-IO | Presentazione Progetti
V: Lab-5-Syscalls-PIPE-FIFO | Inizio Progetti
VI: Lab-6-ldd-Kernel1 | Progetti
VII: Lab-7-Kernel2 | Progetti
VIII – XII: Progetti

Importante!
Documentazioni su Cloud

- Trovate il link eventuale in bacheca su ESSE3
- E' **INDISPENSABILE** collegarsi ai documenti eventualmente predisposti su cloud (es. su Google Drive) se indicati.

Esercitazioni del corso di Sistemi Operativi (3/4)

- Tutte le slide in formato PDF e gli esercizi (codice sorgente) sono pubblicati nell'area del corso in ESSE3
- Linguaggi di riferimento: C
- Sistemi Operativi di riferimento: GNU/Linux - Debian
- Testi di riferimento sono in parte pubblicati su ESSE3 e comunque sempre in fondo alle slide presentate a lezione
- **Forum pubblico su ESSE3**

Esercitazioni del corso di Sistemi Operativi (4/4)

- Programma esercitazioni:

- ✓ Introduzione al linguaggio C, gcc, make, gdb
- ✓ Sistema operativo GNU/Linux
- ✓ Qemu e utilizzo di Debian GNU/Linux
- ✓ Configurazione e compilazione del kernel Linux
- ✓ System calls
- ✓ Scrivere un modulo per il kernel Linux
- ✓ Approfondimenti sul kernel Linux

- Presentazione e svolgimento dei progetti d'esame

Pillole di C, gcc e make

Perche' il C ?

- Struttura minimale
  - ✓ Poche parole chiave (meno del Pascal)
  - ✓ Facile da apprendere
- Unix compliant
  - ✓ Alla base di Unix (C nato per scrivere Unix)
- Portabile
  - ✓ Codice trasferibile da macchina a macchina
  - ✓ Libreria standard
- Pulito
  - ✓ Operatori potenti con accesso al bit
- Modulare
  - ✓ Stile routine: passaggio dei parametri a funzioni esterne per valore
  - ✓ No “decorators” (a la Python)
- C++
  - ✓ E' alla base del C++

C: Concetti di base (1/3)

- Compilazione ed esecuzione
  gcc hello.c
  ./a.out
- Output
  Hello word !

```
// hello.c
#include <stdio.h>
int main(void)
{
        printf("Hello word !\n");
        return 0;
}
```

- Analisi
  - ✓ Il testo del programma viene elaborato dal compilatore
  - ✓ Il compilatore legge il testo una sola volta.
  - ✓ Commenti
  - ✓ Comandi per il pre-processore
  - ✓ Un programma C eseguibile ha una funzione "main" dove inizia l'esecuzione. Le librerie non hanno il "main".

C: Concetti di base (2/3)

- Componenti di un compilatore:
  - ✓ Preprocessore

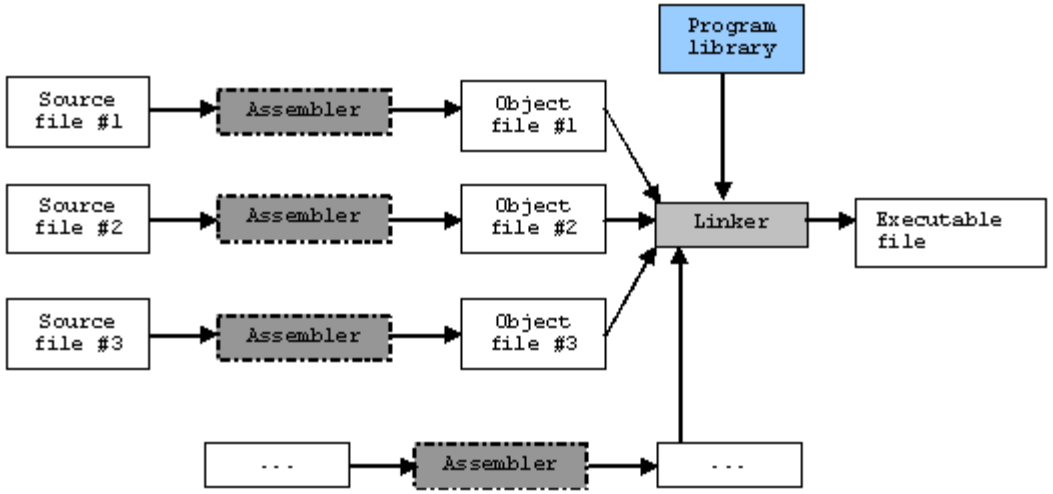
    Legge i comandi a lui riservati (#)
  - ✓ Compilatore

    Traduce da “C” ad assembler (Object files)
  - ✓ Linker

    Combina i file generati allo step precedente con le librerie

    Assegnamento degli indirizzi

```
// hello.c
#include <stdio.h>
int main(void)
{
        printf(“Hello word !\n”);
        return 0;
}
```



C: Concetti di base (3/3)

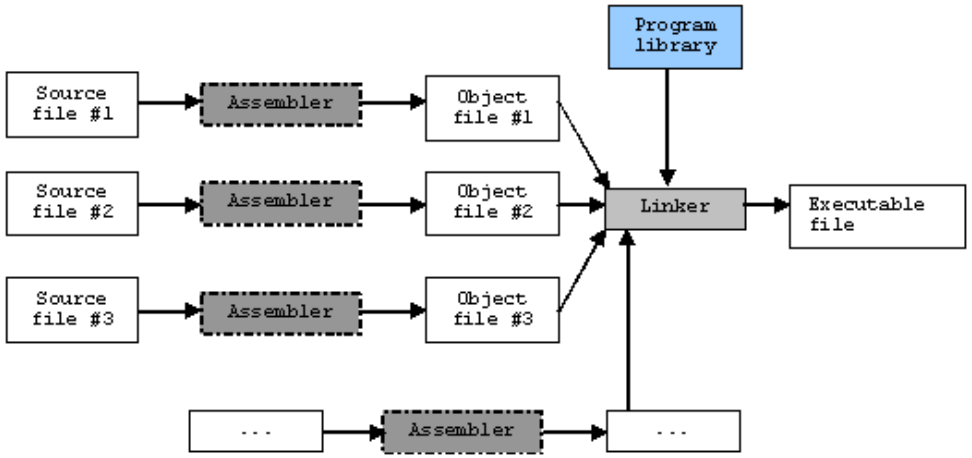
- Il compilatore legge il testo una sola volta:
  - ✓ Le dichiarazioni devono essere fatte subito.
- Ciascuna variabile:
  - ✓ Viene dichiarata una solo volta
  - ✓ Puo' essere definita o utilizzata piu' di una volta
  - ✓ Dichiarare prima di utilizzare!

```
// hello.c
#include <stdio.h>
int main(void)
{
        int a; // Declaration
        a = 5; // Definition

        if (a>0) // Use
        {
                printf("Hello word !\n");
        }
        return 0;
}
```

C : Preprocessore

- Manipola tipograficamente il testo del programma
- Trasforma le direttive in linguaggio C
- Direttive principali:
- Inclusione dei file header:
  #include <stdio.h>
  #include ”persona.h”
- Definizione di nomi simbolici costanti:
  #define DEBUG
  #define NOERROR 0



C : Librerie e Linker

- Librerie: Contengono funzioni globali e variabili globali utilizzabili nei nostri programmi
- Molte sono standardizzate e rese disponibili automaticamente dal compilatore:

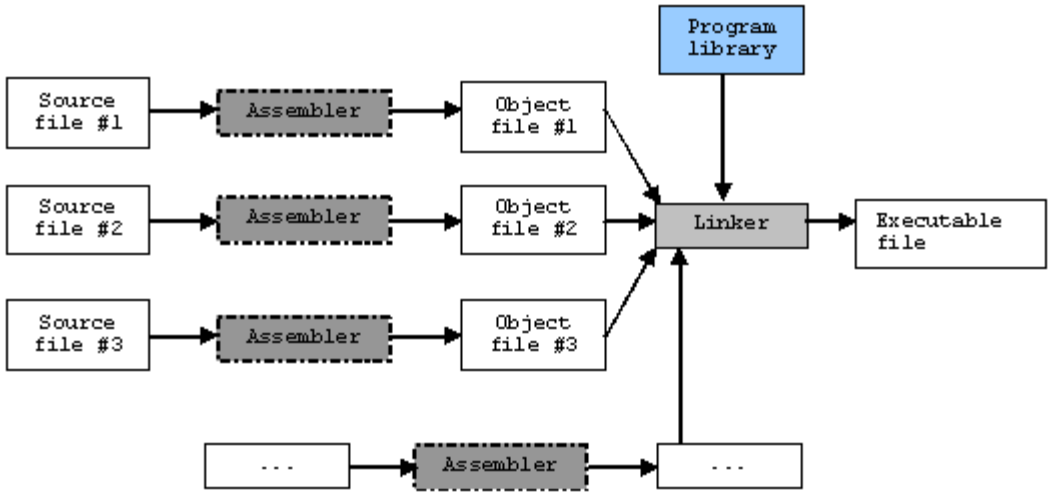
```
#include <stdio.h>
#include <string.h>
```

- Altre librerie aggiuntive devo essere indicate espressamente al compilatore (preprocessore e linker):

```
#include <jpeglib.h>

#include “my_lib.h”
```

- Linker: associa indirizzi di memoria a simboli non definiti nel file oggetto e crea il file eseguibile

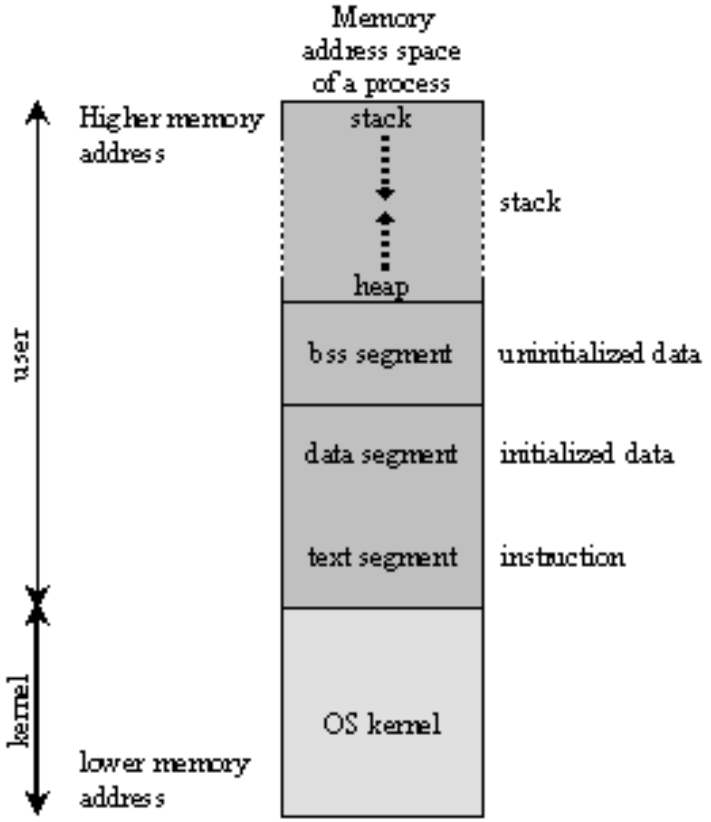


Allocazione della memoria (1/3)

- Memory layout
  - ✓ OS kernel: lowest addresses
  - ✓ Text segment: Istruzioni eseguite dalla CPU
  - ✓ Data segment: “int a=5;” (initialized)
  - ✓ BSS segment: “int b;”, “int b[100];” (uninitialized)
  - ✓ Heap: dynamic memory allocation
  - ✓ Stack: automatic variable allocation

```
// hello.c
#include <stdio.h>
int main(void)
{
        int a =5; //Data segment
        int b; //BSS segment
        int *p = malloc(sizeof(int)); //Heap
        /* … */
        b= f(a);
}

int f(int input) // Stack
{ return input;}
```



Allocazione della memoria (2/3)

- Non ci sono primitive di alto livello (niente costruttori e distruttori ecc.)
- Non c'è garbage collection
- La memoria usata dai programmi è di 3 tipi:
  - **statica** (dichiarata in compilazione, il linker assegna un indirizzo immutabile, es. “int a;”)
  - **dinamica** (allocata durante il funzionamento del programma, es. da malloc, accessibile tramite puntatore)
  - **automatica** (locale, allocata nello stack, scompare quando termina il blocco in cui e' stata dichiarata, es. “{int b;}”)

Allocazione della memoria (3/3)

Riflessioni:

- Variabili statiche:
  - quelle inizializzate finiscono nel segmento dati e nel file eseguibile
  - quelle non inizializzate finiscono nel segmento dati chiamato BSS che viene azzerato all'inizio dell'esecuzione
- Variabili dinamiche
  - sono create a run-time
  - se non inizializzate hanno valore casuale
  - ad ogni malloc deve corrispondere una free
- Variabili automatiche:
  - se non inizializzate hanno valore casuale

gcc : Introduzione (1/3)

- gcc = GNU C Compiler
  - ✓ e' software libero, parte del progetto GNU di Free Software Foundation
  - ✓ e' presente su pressoche' tutte le architetture e sistemi operativi
- Riceve opzioni dalla riga di comando, tra cui nomi di file:
  - ✓ .c : sorgenti C da compilare
  - ✓ .s : sorgenti assembly per l'assembler
  - ✓ .o : object file per il linker

gcc : Comandi di Base (2/3)

- Opzioni piu' frequenti:
- *gcc <foo.c>* : compila <foo.c> e genera l'eseguibile a.out
- *gcc <opzioni> -o <foo>* : scrive il risultato della compilazione in <foo> invece che a.out
- *gcc -c <foo.c>* : compila <foo.c> in <foo.o> senza eseguire il linker
  - ✓ Codice macchina con indirizzi relativi (0 e 1)
- *gcc -E <foo.c>* : esegue solo il preprocessore e scrive il risultato su stdout (a schermo)
- *gcc -S <foo.c>* : compila fermandosi prima dell'assemblatore ottenendo <foo.s> (codice assembler)
  - ✓ File assembly

gcc : Comandi di Base (3/3)

- *gcc -I<directory> <opzioni>* : cerca anche in <directory> gli header files da includere
- *gcc -L<directory> <opzioni>* : cerca anche in <directory> le librerie necessarie
- *gcc -l<libreria> <opzioni>* : aggiungi anche <libreria> nella lista da passare al linker

C : Tipi di Dati Fondamentali

- Caratteri
  - char
  - char c='a'; char c[30];
- Interi
  - short, int, long;
  - int a=5; int a1[30];
- Reali
  - float, double, long double
  - double d=5; double d[20];
- Puntatori
  - int *ap = a1;
  - Int *ap = &a;

Operatore sizeof() e esercizio sizeof.c

Stringhe e Puntatori

- Una stringa e' un vettore di caratteri che termina con '\0'
- Esercizi dal file string.c
- Ogni volta che il compilatore incontra una stringa esplicita (es. "abc") la memorizza nel segmento dati e la rappresenta come un puntatore al primo carattere
- *(a+i) e' equivalente a a[i] e i[a]

Printf

*int printf(char *format, arg list ...)*

- stampa su *stdout* la lista di argomenti conformemente alla stringa di formato specificata
- Ritorna il numero di caratteri stampati

- Conversioni importanti e osservazioni:

  %i : interi

  %f : double

  %s : stringhe

  %x : esadecimali

  %3.2f : stampa un reale con 2 cifre decimali riservando 3 spazi

Scanf

*int scanf(char *format, args ...)*

- Legge dallo *stdin* e mette l'input nelle variabili i cui indirizzi sono specificati nella lista di args
- Ritorna il numero di caratteri letti
- La stringa di controllo *format* e' simile a quella di printf
- La funzione scanf richiede di specificare l'indirizzo di ogni variabile, oppure un puntatore ad essa:
  - ✓ `int i;  scanf("%d",&i);`
- E' anche possibile indicare il nome di una stringa:
  - ✓ `char string[80];  scanf("%s",string);`

Strutture e typedef

Una dichiarazione *struct* consiste di una lista di campi, ciascuno di un tipo a scelta.

```
struct country
{
  char capital[30];
  int population;
  double area;
};
```

Dichiarazione di una variabile "struct country":

```
struct country italy = {"Rome", 61000001, 301308.2};
```

Lo scopo di *typedef* è di assegnare nomi alternativi (*alias*) a tipi esistenti:

una definizione *struct* dopo *typedef* definisce un nuovo tipo:

```
typedef struct
{
  char capital[30];
  int population;
  double area;
} country;
```

```
country italy = {"Rome", 61000001, 301308.2};
```

Memoria dinamica

- *malloc* alloca *size* bytes di memoria
- *calloc* alloca lo spazio per *nobj* oggetti di dimensione *size* bytes
- *realloc* incrementa lo spazio precedentemente allocato
- *free* libera lo spazio

```
#include<stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);

void free(void *ptr);
```

getopt

The getopt function automates some of the chore involved in parsing typical unix command line options

Intestazione: `#include <unistd.h>`

The options argument is a string that specifies the option characters that are valid for this program. An option character in this string can be followed by a colon (':') to indicate that it takes a required argument. If an option character is followed by two colons ('::'), its argument is optional; this is a GNU extension.

- Variable: *int optopt*

When getopt encounters an unknown option character or an option with a missing required argument, it stores that option character in this variable.

- Variable: *int optind*

This variable is set by getopt to the index of the next element of the argv array to be processed. Once getopt has found all of the option arguments, you can use this variable to determine where the remaining non-option arguments begin.

- Variable: *char * optarg*

This variable is set by getopt to point at the value of the option argument, for those options that accept arguments.

Visibilita’

- Globale: visibile in tutto il programma
- Globale statica: Visibile all’interno del file
- Locali: variabile automatica
- Locale statica: persistente

```
#include<stdio.h>

int i=5;
static int j;

static int invert(int i)
{
      int j;
      j = -i;
      return j;
}

int count(void)
{
      static int i;
      return ++i;
}

int main() {
  printf("%d\n",count());
  printf("%d\n",count());
  printf("%d\n",invert(count()));
  printf("%d\n",invert(i));
}
```

Automazione: make (1/5)

- *make* esegue operazioni automatizzate seguendo una descrizione di regole e dipendenze (Makefile)
- Analogia: ricetta
- Elemento base di Makefile (regola):

  target : dipendenze

  <TAB> azione

```
foo : foo.o
      gcc -o foo foo.o

foo.o : foo.c foo.h
      gcc -c foo.c
```

Automazione: make (2/5) target multipli

- make esegue la prima regola presente in Makefile (e tutte le sottoregole necessarie)
- posso eseguire una regola specifica indicandone il nome a make

  clean:

  */bin/rm -f *~ .*~ *.BAK .*.BAK *.o*
- E' opportuno indicare nel Makefile i target che non producono file direttamente

  .PHONY: clean all test
- Esercizio: scrivere e provare il Makefile per sample1

  Aggiungere i target clean e test

  Aggiungere all'inizio il target all che esegue tutti gli altri

Automazione: make (3/5)
Esercizio – sample1

- Abbiamo 3 file di codice con i relative headers:
  - bar1.c, bar2.c, bar3.c
- L'eseguibile sara' funzione dei 3 oggetto:
  - bar: bar1.o bar2.o bar3.o
- bar1.o da chi dipende ?
  - bar1.c, bar1.h bar2.h bar3.h

```
bar1.o: bar1.c bar1.h bar2.h bar3.h
        gcc -c bar
```

- bar2.o da chi dipende ?
  - bar2.c, bar2.h bar3.h

```
bar2.o: bar2.c bar2.h bar3.h
        gcc -c bar2.c
```

- bar3.o da chi dipende ?
  - bar3.c, bar3.h

```
bar3.o: bar3.c bar3.h
        gcc -c bar3.c
```

```
.PHONY: clean
all: bar1.o bar2.o bar3.o
bar1.o: bar1.c bar1.h bar2.h bar3.h
        gcc -c bar1.c
bar2.o: bar2.c bar2.h bar3.h
        gcc -c bar2.c
bar3.o: bar3.c bar3.h
        gcc -c bar3.c
clean:
        /bin/rm -f *~ .*~ *.BAK .*.BAK *.o
```

Automazione: make (4/5) parametrizzazione

- Posso migliorare Makefile introducendo variabili.

```
CC          = gcc
CFLAGS   = -Wall -Wextra // extra warnings
foo : foo.o
    $(CC) -o foo foo.o $(CFLAGS)
foo.o : foo.c foo.h
    $(CC) -c foo.c $(CFLAGS)
```

- Esercizio: modificare il Makefile precedente e introdurre variabili ovunque sia appropriato
  - Makefile in exercises
  - Makefile sample 1

Automazione: make (5/5) parametrizzazione – sample 1

```
CC          = gcc
CFLAGS      = -Wall -ansi -pedantic
CDEBU       = -g
CPPFLAGS =
LDFLAGS     =
LDLIBS      =

FILE        = bar
OBJS        = $(FILE)1.o $(FILE)2.o $(FILE)3.o

all: $(OBJS)

$(FILE)1.o: $(FILE)1.c $(FILE)1.h $(FILE)2.h $(FILE)3.h
	$(CC) $(CFLAGS) $(CDEBU) -c $(FILE).c

$(FILE)2.o: $(FILE)2.c $(FILE)3.c $(FILE)3.h
	$(CC) $(CFLAGS) $(CDEBU) -c $(FILE)1.c

$(FILE)3.o: $(FILE)3.c $(FILE)3.h
	$(CC) $(CDEBU) -c $(FILE)3.c
```

Gdb – The GNU debugger (1/3)

- Permette di “vedere” cosa sta accadendo in un programma mentre viene esegue
- E' compatibile con diversi linguaggi
  - ✓ es. C, Pascal, ...
- Associa indirizzi a link simbolici a patto che i simboli siano inseriti all'interno del sorgente:
  - ✓ gcc -g program_to_debug.c
- Esistono interfacce grafiche
  - ✓ xxgdb, tgdb, ddd
- E' lo strumento piu' utilizzato per fare il debug al kernel

Gdb (2/3) - Esecuzione

- Iniziare una nuova sessione di debug
  - ✓ gdb <eseguibile>
- Eseguire il programma
  - ✓ run <parametri>
- Impostare un break point
  - ✓ break <nome funzione, file:linea, ...>
- Eseguire passo dopo passo
  - ✓ step (ed eventualmente il numero di passi)
- Eseguire istruzione successiva nella funzione corrente
  - ✓ next (ed eventualmente il numero di passi)
- Continuare fino a break point (se attivato)
  - ✓ continue

Gdb (3/3) - Information

- Avere informazioni generiche
  - ✓ Info (breakpoints, file, ...)
- Mostrare la linea su cui il task sta eseguendo
  - ✓ list
- Mostrare valore di una variabile
  - ✓ print <nome_variabile>
- Vedere il backtrace di esecuzione
  - ✓ backtrace
- Vedere codice assembly
  - ✓ disassemble
- Uscire dalla sessione di debug
  - ✓ Quit

Riferimenti

- *Linguaggio C: guida alla programmazione*, di A. Bellini – A. Guidi
- GDB: The GNU Project Debugger, http://www.gnu.org/software/gdb/