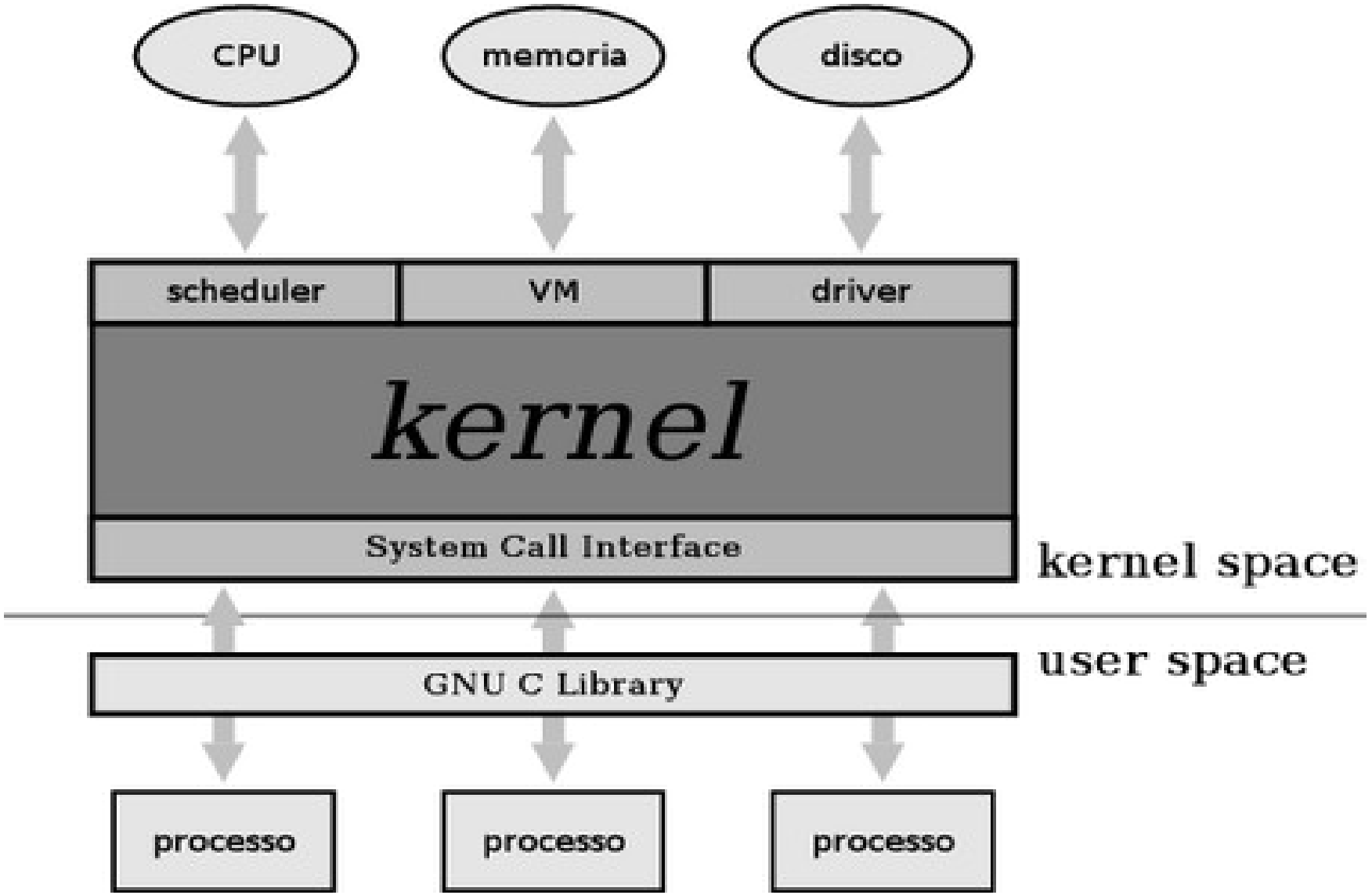
Kernel and User space



Esercizio

Esercizio: Scrivere e compilare un programma che stampi a video "Hello world" (soluzione.c)

La compilazione

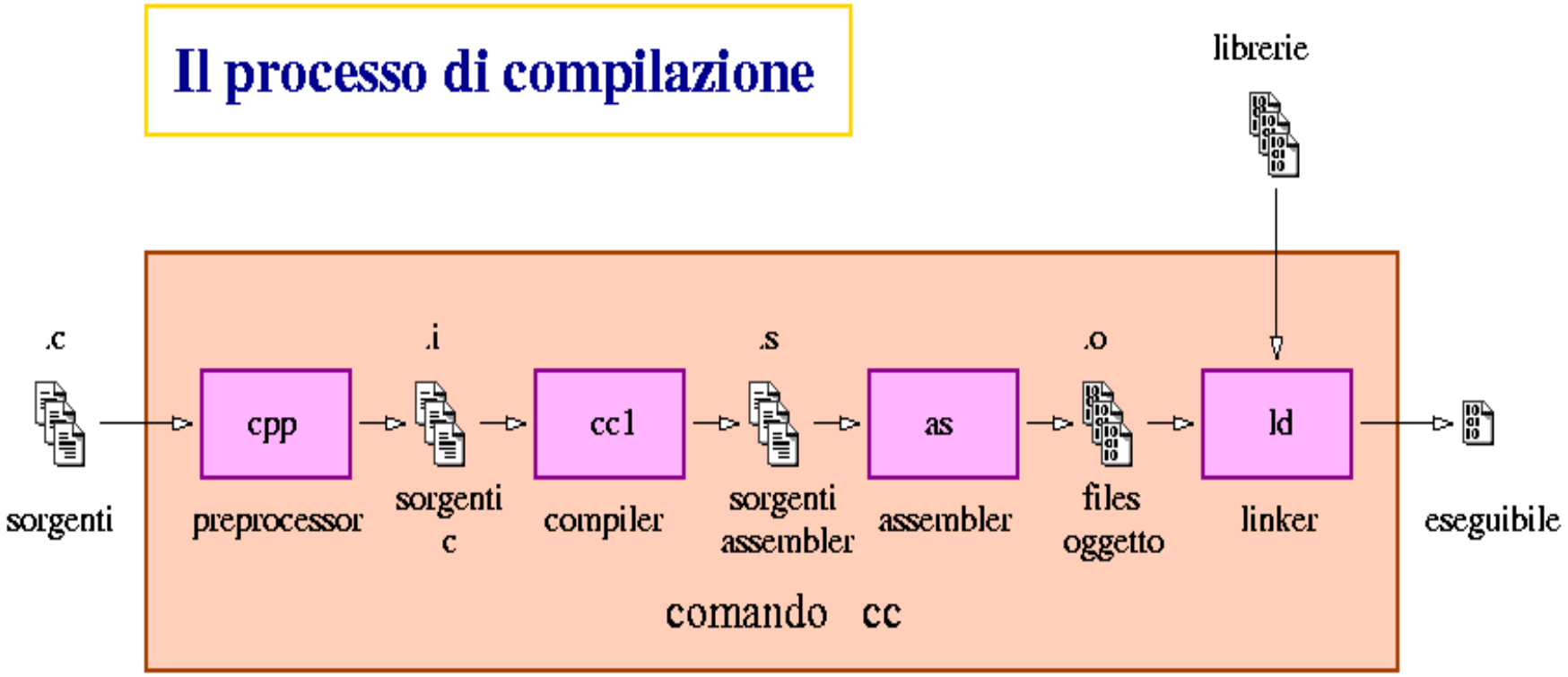
Compilare un sorgente C:

```
$ gcc -o main main.c
$ file main
main: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.6.9, dynamically linked (uses shared libs), for GNU/Linux
2.6.9, not stripped
```

Ci sono almeno tre tipi di file ELF (Executable and Linking Format, come dice il nome stesso un formato per file eseguibili) : file oggetto *.o (questo come vedremo comprende anche librerie statiche), file eseguibili veri e propri e librerie a link dinamico. Anche se servono per scopi differenti sono di per se tre tipi di file simili. Meglio vedremo quando parleremo di eseguibili e processi. Ad esempio come creare un file oggetto ?

```
$ gcc -c -o main.o main.c
$ file main.o
main.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not
stripped
$ readelf -a main.o
```

La compilazione



Il preprocessore

```
$ cpp main.c

...cut
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;


typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;

cut...
```

Il preprocessore

E' fondamentale capire che il codice che viene passato al compilatore e' il risultato del codice preprocessato. Questo puo' portare a qualche incoveniente, ad esempio:

```
#include <iostream>

int main ()
{
  if (index)
     std::cout << "here I am" << std::endl;
  else
    std::cout << "There" << std::endl;

  return 0;
}
```

Il codice contiene un errore evidente, ma se proviamo a compilarlo ed eseguirlo cosa succede ? E perche' ?

Il compilatore

```
$ gcc -S main.c
$ cat main.s

        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello! \n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $8, %esp
        pushl   $.LC0
cut...
```

Assemblatore

```
$gcc -c -o main.o main.s
```

Visualizzare i simboli all'interno di una file oggetto:

```
$ nm main.o
         U fwrite
00000000 T main
         U stdout
$ file main.o
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

Il compilatore genera un oggetto che contiene un riferimento ad fwrite. fwrite non e' definito all'interno del file quindi nm lo segna come U, e' un riferimento esterno, cioe' all'interno dell'oggetto non si conosce ancora la locazione di fwrite (L'eseguibile finale conterra' 1 istruzione per la chiamata ad fwrite). L'assemblatore si accorge che la funzione fwrite e' esterna e genera una rilocazione e sara' in fine il linker che andra' a trovare e sostituire l'indirizzo corretto per la chiamata alla funzione fwrite. Senza scendere in dettagli ulteriori possiamo dire che, la struttura di un binario ELF si puo' dire che semplifica il compito del linker e del kernel loader, rispetto al vecchio a.out (Per vedere le differenze tra gli header di un binario a.out ed elf vedi /usr/include/linux/a.out.h e /usr/include/linux/elf.h. Il confronto diretto dei due file da solo 1 idea della differente complessita' in gioco nei due casi.)

Linking

```
$gcc -v -o main main.o
```

collect2: usando collect2 -v e' possibile vedere l'invocazione vera e propria del linker.

ld: e' il linker vero e proprio, ad esempio per generare l'eseguibile main a partire dall'oggetto main.o

```
/usr/bin/ld --eh-frame-hdr -m elf_i386 -dynamic-linker
     /lib/ld-linux.so.2 -o main
     /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crt1.o
     /usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crti.o
     /usr/lib/gcc/i386-redhat-linux/4.1.1/crtbegin.o
     -L/usr/lib/gcc/i386-redhat-linux/4.1.1 -L/usr/lib/gcc/i386-redhat
     -linux/4.1.1 -L/usr/lib/gcc/i386-redhat-linux/4.1.1/../../..
     main.o -lgcc –as-needed -lgcc_s --no-as-needed -lc -lgcc
     --as-needed -lgcc_s –no-as-needed /usr/lib/gcc/i386-redhat-
     linux/4.1.1/crtend.o /usr/lib/gcc/i386-redhat
     linux/4.1.1/../../../crtn.o
```

Stripped or not

```
$ strip main.o
$ nm main.o
nm: main.o: no symbols

$ gcc -o main main.o
/usr/lib/gcc/i386-redhat-linux/4.1.1/../../../crt1.o: In function `_start':
(.text+0x18): undefined reference to `main'
collect2: ld returned 1 exit status
```

Possiamo invece eliminare tutti i simboli che non sono necessari con strip --strip-unneeded main.o, o comunnque possiamo invece eliminare i simboli dall'eseguibile:

```
$ gcc -o main main.c
$ nm main
080495b0 A __bss_start
08048324 t call_gmon_start
080495b4 b completed.5757
080494b4 d __CTOR_END__
```

...cut

Stripped or not

```
$ ls -l main
-rwxr-xr-x 1 redo thch 6820 Mar  4 10:31 main
$ strip main
```

A questo punto non vedo piu' i simboli:

```
$ ls -l main
-rwxr-xr-x 1 redo thch 4448 Mar  4 10:31 main
$ nm main
nm: main: no symbols
$ file main
```

Ed anche i file se ne accorge:

```
$ file main
main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.9, stripped
```

Dynamically linked

Per visualizzare la lista delle librerie dinamiche richieste semplicemente e' possibile usare:

```
$ ldd main
        libc.so.6 => /lib64/libc.so.6 (0x000000316cc00000)
        /lib64/ld-linux-x86-64.so.2 (0x000000316ba00000)
```

File di include ed errori del linker

Proviamo adesso ad usare qualche funzione matematica:

```
$ gcc -o main1 main1.c
/tmp/ccksTQPe.o: In function `main':
main1.c:(.text+0x57): undefined reference to `sqrt'
collect2: ld returned 1 exit status
```

l'errore come e' evidente e' un errore di linker , e' necessario infatti "linkare" le librerie matematiche:

```
$ gcc -o main1 main1.c -lm
```

Eliminando l'include di math.h cosa succede ?

```
$ gcc -o main2 main2.c -lm
$ gcc -W -Wall -o main2 main2.c -lm
```

Vediamo cosa contine il file di include math.h

```
$ less /usr/include/math.h
```