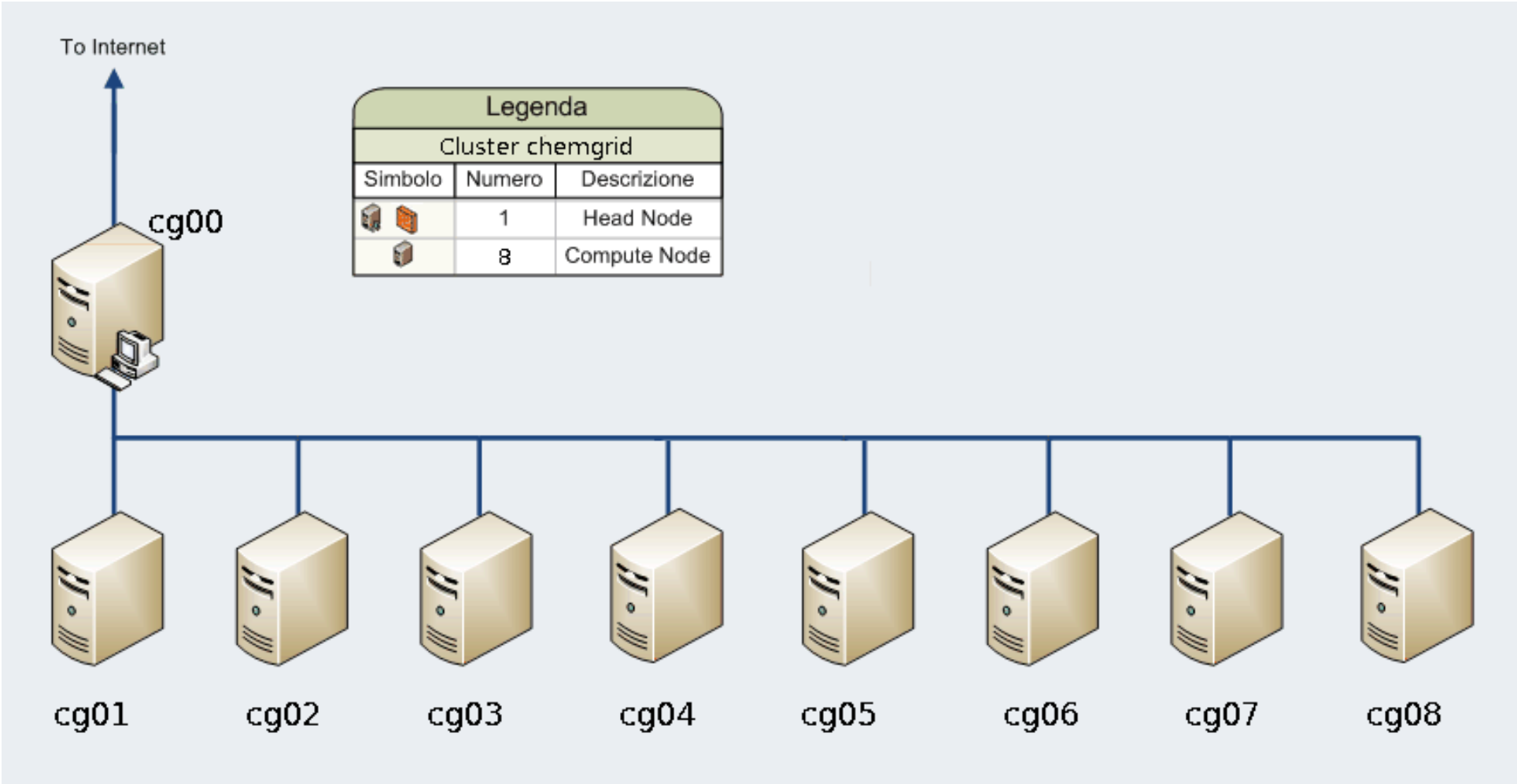
Cluster che useremo

Hostname: chemgrid.unipg.it
User id: acrXX

Vediamo se funziona l'accesso a tutti i nodi del cluster e prendiamo confidenza con la struttura stessa del cluster:

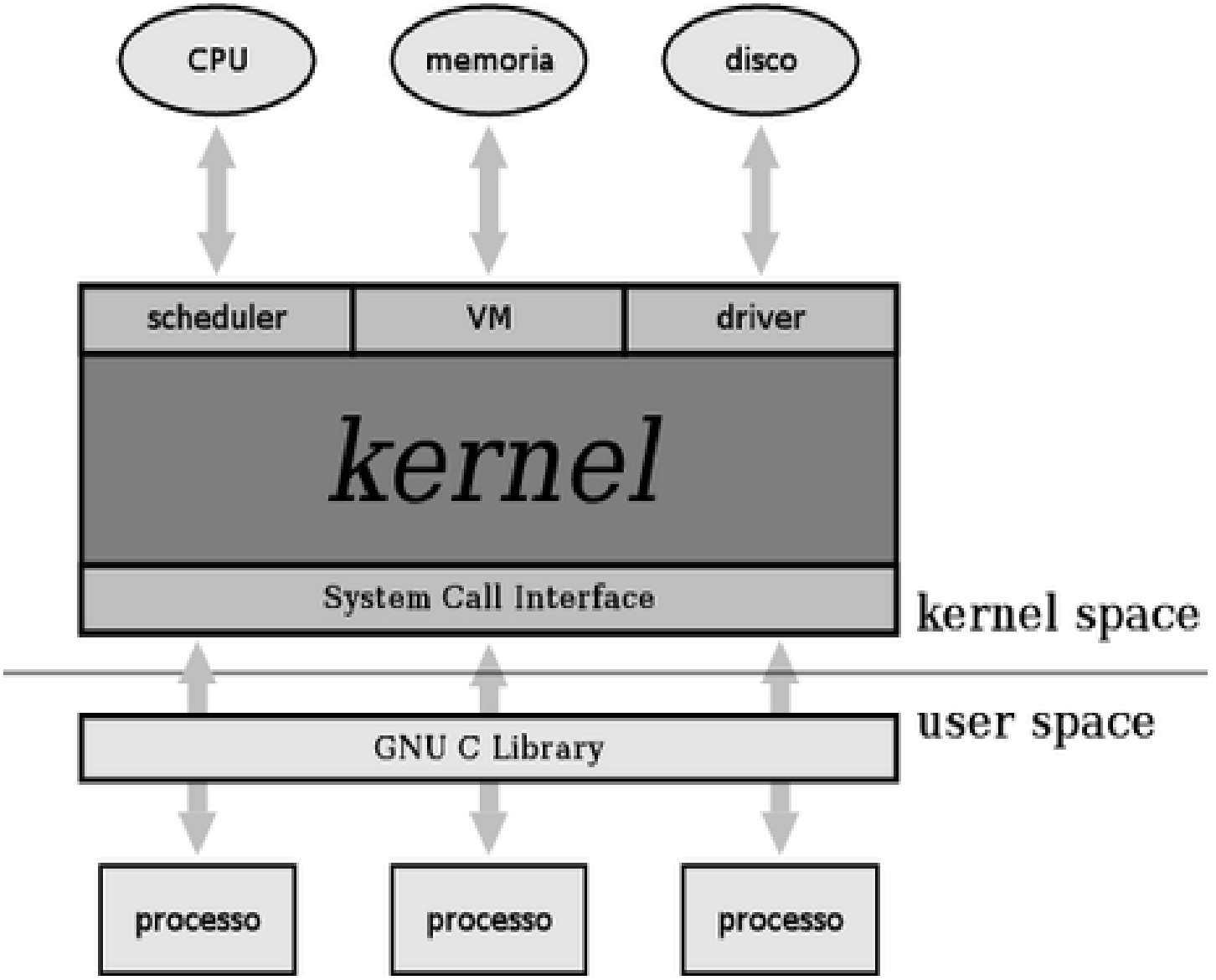
```
$ for name in cg00 cg01 cg02 cg03 cg04 cg05 cg06
   cg07 cg08; do rsh $name hostname; done
```

Scrivete in un file readme.txt vostro nome ed indirizzo e-mail (mailing-list)


To Internet
Legenda
Cluster chemgrid
Simbolo
Numero
Descrizione
1
Head Node
8
Compute Node
cg00
cg01
cg02
cg03
cg04
cg05
cg06
cg07
cg08

Hello World!

La compilazione



La compilazione (compila)

Per produrre un eseguibile a partire da sorgenti C (anche C++ e Fortran) sono necessari 4 passi:

1. Preprocessore: mediante l'uso di direttive produce il “sorgente” vero e proprio
2. Compilatore: converte il sorgente in linguaggio Assembly
3. Assembler: converte il codice in linguaggio assembly in linguaggio macchina
4. Linker: “collega” il codice macchina prodotto, e genera l'eseguibile vero e proprio.

La compilazione

Vediamo dunque passo per passo cosa accade:

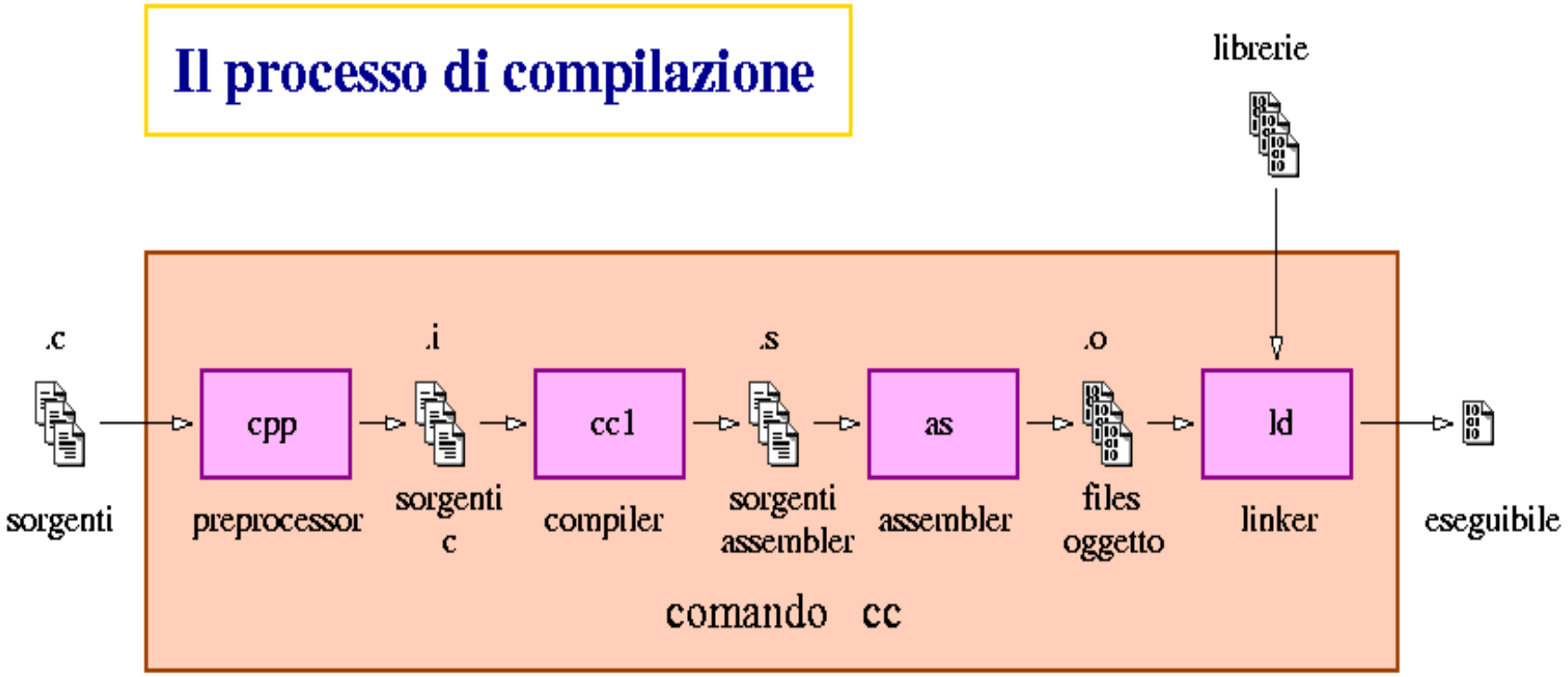
$ cpp main.c > main1.c : invocazione del preprocessore

$ gcc -S main1.c : produzione del sorgente assembly

$ gcc -c -o main.o man1.s : assemblatore produce il file oggetto main.o

$ gcc -v -o main main.c : l'opzione -v ci permette di vedere esplicitamente l'invocazione del linker

La compilazione



Mai fidarsi troppo del compilatore ?

Vediamo l'esempio: **compilaerror**

```
$ g++ -W -Wall -o main main.cpp
$ ./main
Here
```

la resposta e' data dai due comandi seguenti, oltre che dal “sacro” libro dello standard:

```
$ man index
$ cpp main.cpp
```

Librerie

Una **libreria statica** (.a) e' semplicemente una collezione di file oggetto:

```
$ nm libutil.a
  alloca.o:
    00000000 T dalloca_
             U fprintf
             U malloc
             U stdout
    free.o:
             U free
```

Librerie

Durante la fase di linking il linker estrae dalla libreria le funzioni richieste (non il sorgente) e le rende parte del file eseguibile.

Nel caso di una **libreria dinamica** (.so) le cose cambiano considerevolmente. In questo caso le "funzioni" non vengono rese parte dell'eseguibile, ma rimangono nella libreria che dovra' essere caricata dal loader al momento dell'esecuzione dell'eseguibile (l**dd** permette di stampare a video le dipendenze)

Makefile (makefile)

Un makefile consiste di alcune regole cosi' descritte:

```
TARGET: DIPENDENZE
        COMANDO
```

Se le regole sono memorizzate in un file chiamato Makefile o makefile e' sufficiente digitare il comando make seguito dal target che si vuole aggiornare (altrimenti si deve usare l'opzione -f per specificare il file corretto).Se non si specifica alcun target, viene eseguito automaticamente il primo.

Makefile

Di solito TARGET e' il nome dell'eseguibile o del file oggetto che si vuole generare, ma puo' anche essere un'azione ad esempio clean, una sorta di identificativo dell'azione da eseguire, in tal caso alla chiamata:

$ make clean

verra' eseguito il target “clean”

Makefile

Dipendenze, quando eseguire il comando ?

```
eseguibile: object1.o main.o
        comando
```

a sinistra dei due punti troviamo quindi il target, a destra troviamo la lista delle dipendeze che sono necessarie in qualche modo al target. Quando si esegue

```
$ make eseguibile
```

make controlla la data di ciascun oggetto, se questa e' piu' recente di eseguibile, allora viene eseguito il comando.

Makefile

Una caratteristica del make e' che le dipendenze (o “sorgenti”) del target sono “costruite” prima del confronto dei “timestamps”. In pratica la linea:

eseguibile: main.o

implica un “make main.o”:

main.o: main.c

comando1

che ha come “sorgente” main.c. Se main.c e' piu' recente di main.o, quest'ultimo viene ricostruito (cioe' viene seguito comando1). A questo punto main.o sara' piu' recente di eseguibile e quindi ?

Makefile

Vediamo l'esempio: **makefile**

Esercizio makefile: Dati i files main1.c, sub1.c, sub2.c e sub1.h scrivere il Makefile corrispondente.