Sistemi Operativi, Secondo Modulo, Canale A–L e Teledidattica
Riassunto della lezione del 28/04/2021

Igor Melatti

Il Linguaggio C: il *casting*

- Alla fine della giostra, i tipi del C possono essere:
  - predefiniti
  - strutture
  - puntatori ad uno dei 2 precedenti (include il caso degli array)
- Considerare l'esempio `cast.c` (*casting esplicito*)
  - quello implicito avviene ogni volta che si usa un tipo al posto di altro, senza fare cast espliciti
    - non sempre possibile: il compilatore dà *errore* se si prova a farlo tra un qualsiasi tipo ed una struct
    - a meno che non si tratti della stessa struct
    - due 2 struct sono diverse se hanno *nomi* diversi (anche se le strutture sembrano uguali!)
    - il compilatore dà *warning* se si fa un cast tra un puntatore con n livelli di indirezione ed un altro con $m \neq n$ (ad esempio, un cast tra `int **` e `float *`)
    - uno tra m ed n può anche essere 0...
    - dà warning anche nel caso si faccia casting tra un puntatore ad una struttura ed un puntatore ad un'altra struttura (solo se implicito; per l'esplicito, niente warning)
  - casi tipici: assegnamenti e confronti (ad es.: `int a; float b; a = b; if (a == b) ...`) e argomenti di funzione (`void f(int a) {} int main() {f(1.0);}`)
  - l'effetto è lo stesso del corrispettivo cast esplicito
  - farlo esplicito è un modo per dire "so cosa sto facendo, voglio veramente che avvenga la conversione prevista dal casting"

        * ovviamente, se il cast è vietato dalle regole di cui sopra è inutile essere consci...

- L'operazione di *cast* (esplicito) si effettua sintatticamente in questo modo `(tipo)espressione`
  - di nuovo, non sempre possibile: stessi casi di cui sopra
- Semanticamente, si sta dicendo al compilatore: l'espressione avrebbe un tipo t (e il compilatore, grazie alle sue regole interne, sa cos'è t), ma tu trasformala nel tipo `tipo` (quello dato nel cast)
  - se si fa cast tra tipi predefiniti, allora avviene anche una effettiva conversione: ad esempio, `(double)1` trasforma `1` in `1.0`
    * infatti, l'intero 1 e il double 1 hanno rappresentazioni binarie molto diverse: il compilatore fa la conversione dall'una all'altra
  - finché il cast di destinazione è in grado di memorizzare il tipo di partenza, non è un problema
  - ad esempio: da int a long, o da short a int
  - da notare che passare da tipo intero a tipo con virgola mobile può causare la perdita di precisione, ma viene mantenuto correttamente l'ordine di grandezza
  - altrimenti, tipicamente si perdono i bit in eccesso (vedere assegnamento `l = i;` in `cast.c`)
  - se invece il cast è tra puntatori (con lo stesso livello di indirezione), non avviene alcuna conversione
    * l'effetto principale è sull'aritmetica dei puntatori: `(char *)p + 1` e `(int *)p + 1` danno risultati diversi
    * a livello di referenziamento, si sta facendo un cast sulla destinazione: vedere ultima istruzione di `cast.c`
    * il cast tra puntatori a tipi diversi serve per dire al compilatore "tranquillo, so cosa sto facendo"

Il *Makefile*

- Il `Makefile`
  - file di testo che viene interpretato dal comando `make`
  - usato per creare automaticamente dei file in dipendenza di altri file
  - nel nostro caso, per creare o file oggetto o file eseguibili a partire dai sorgenti C
- Com'è fatto (minimalmente!) un `Makefile`
  - sequenza di regole di questo tipo:

```
target: lista_di_file
        comando1
        ...
        comandon
```

- lanciando il comando `make target`, viene controllato se `target` è un file con mtime più vecchio di uno dei file in `lista_di_file`
  - `lista_di_file` sono i file da cui `target` dipende
- se così è, vengono eseguiti i comandi `comando1`...`comandon` (*recipe*)
  - si suppone che l'effetto finale di tali comandi sia generare il file `target`, usando i file in `lista_di_file` come input
- altrimenti non viene eseguito nulla: il target è già up-to-date rispetto alle sue dipendenze
- se `lista_di_file` è vuota:
  - se `target` è un file esistente, viene considerato sempre up-to-date (nessun comando viene eseguito)
  - altrimenti, i comandi vengono sempre eseguiti
- se `target` non dev'essere inteso come un nome di un file, ma come una regola da invocare sempre (es. tipico: il target `clean`), allora occorre dichiarare inizialmente:

  ```
  .PHONY: lista_di_target_che_non_sono_file
  ```

  - altrimenti, se per caso viene creato un file che si chiama `target` (ad es.: `touch clean`), la corrispondente regola non viene eseguita
- il tutto è ricorsivo: se lanciando `make target` viene eseguito un comando che modifica `target`, e `target` compare a sua volta nella lista delle dipendenze di qualche regola, verrà attivata anche quest'altra regola
- `make` da solo esegue la prima regola in `Makefile`

La compilazione separata

- Uso di `static` ed `extern`
  - possono essere usati sia per variabili che per funzioni
  - utili nei casi in cui il programma è diviso su più file da compilare separatamente e da linkare solo alla fine
    - vedere esempi allegati alla lezione 17: `extern.c` e `static.c` sono compilati separatamente nei rispettivi file oggetto, e poi linkati assieme (vedere il file `Makefile`)
  - ci limitiamo ai seguenti casi

    - variabili globali statiche, come `var_glob_static` di `static.c`
        - dichiararle statiche è un modo per dire: "se questo file verrà compilato e il corrispondente file oggetto usato in una fase di linkaggio con altri file oggetto, queste variabili non potranno essere accedute né in lettura né in scrittura dagli altri file oggetto"
        - vengono proprio nascoste, quindi gli altri file possono anche dichiarare delle variabili globali con lo stesso nome (come `var_glob_static2` di `extern.c`)
    - variabili globali esterne, come `var_glob_extern` di `extern.c`
        - dichiararle esterne è un modo per dire: "non allocare la memoria per questa variabile: verrà fatto in un altro file che verrà linkato con questo"
        - non possono quindi essere inizializzate, visto che la memoria non c'è
        - solo lette/modificate nei corpi delle funzioni
        - dovrà esserci un file usato in fase di link nel quale la variabile non sia dichiarata `extern`
        - `extern` può essere omesso, ma *solo se non c'è l'inizializzazione*
        - altrimenti, in fase di linking verrà trovata una variabile dichiarata 2 volte, e ci sarà un errore da parte di `gcc`
    - variabili locali esterne
        - come le globali, ma possono essere usate solo dalla funzione nella quale si trovano
    - variabili locali statiche, come `times_called` di `static.c`
        - la variabile non verrà allocata sullo stack delle chiamate, ma ne esisterà una sola copia che verrà acceduta da ogni chiamata alla funzione
    - funzioni statiche, come `f_only_here` di `static.c`
        - dichiararle statiche è un modo per dire: "se questo file verrà compilato e il corrispondente file oggetto usato in una fase di linkaggio con altri file oggetto, queste funzioni non potranno essere chiamate dagli altri file oggetto"
    - funzioni esterne: come le funzioni "normali"

- Posizionamento delle variabili in memoria: vedere Figura 1

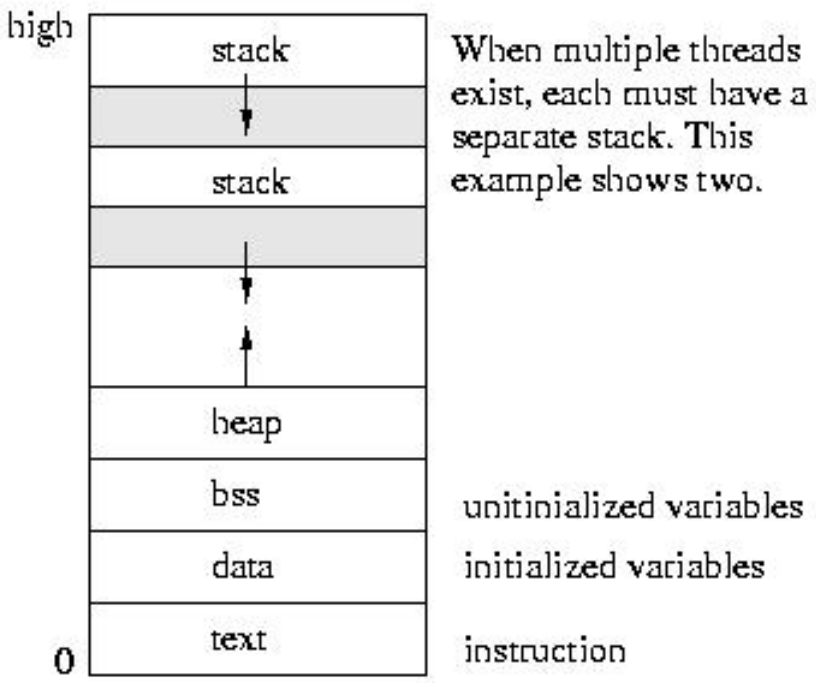

Figure 1: Composizione di un processo in memoria. Le variabili `var_glob_extern`, `times_called`, `var_glob_static` e `var_glob_static2` sono in `data`; `var_glob_extern2` è in `bss`; la `var_glob_extern` alla riga 30 di `extern.c` è sullo stack (ma solo quando viene eseguito il corrispondente pezzo di codice). Rispetto a `puntatori.c`, le variabili `p`, `p1` ed `n` sono sullo stack (quest'ultima, solo quando viene chiamata `incr` oppure `incr2`); non essendoci nessuna `malloc/calloc/realloc`, lo `heap` è sempre vuoto