Modulo 4 - Sezione A
Introduzione alle system call

Laboratorio di Sistemi Operativi I
Anno Accademico 2008-2009

La programmazione di sistema

- Il *kernel* è la parte del sistema operativo che gestisce le risorse HW (memoria, dischi, canali di I/O) e ne arbitra e coordina l'utilizzo tra le varie applicazioni e i vari utenti
- Il codice del kernel è sempre residente in memoria
- Il kernel mette a disposizione del programmatore una serie di funzioni chiamate *system call* (possono essere considerate le *primitive* del sistema operativo)
- L'insieme delle system call costituisce l'interfaccia del kernel verso il programmatore (*API=Application Programming Interface*)
- La programmazione di sistema consiste nell'utilizzare le system calls del kernel

Le system call

- Tutti i programmi che girano sotto Linux in ultima analisi fanno uso di tale interfaccia; anche le librerie di sistema si appoggiano sulle system call.
- In Linux (kernel 2.6) sono circa 280 (**`/usr/include/asm/unistd.h`**)
- Spesso ad una system call corrisponde (a più alto livello) una funzione di libreria standard C (e.g.: **`open`** → **`fopen`**)
- L'invocazione della system call segue la sintassi di chiamata di una normale funzione C
- Ogni system call ha un prototipo, definito negli include file di sistema (nella directory **`/usr/include`** e sue subdirectory): ad esempio
  `pid_t fork(void);`
- Alcune system call possono essere invocate con successo soltanto se il processo chiamante gira con i privilegi dell'utenza root
- **Nota**: utilizzare **`man`** ***`nome_system_call`*** per visualizzare il prototipo della system call e gli include file necessari

La libreria standard C

- La libreria standard del C (`libc`) contiene funzioni di utilità che forniscono servizi *general purpose* al programmatore
- Queste funzioni **NON** sono chiamate dirette a servizi del kernel, sebbene alcune di esse possono fare uso delle system call per realizzare i propri servizi
- Le funzioni di libreria standard C risiedono generalmente in una libreria dinamica
- **Esempi:**
  - la funzione `fopen` invoca la system call `open` per accedere ad un file

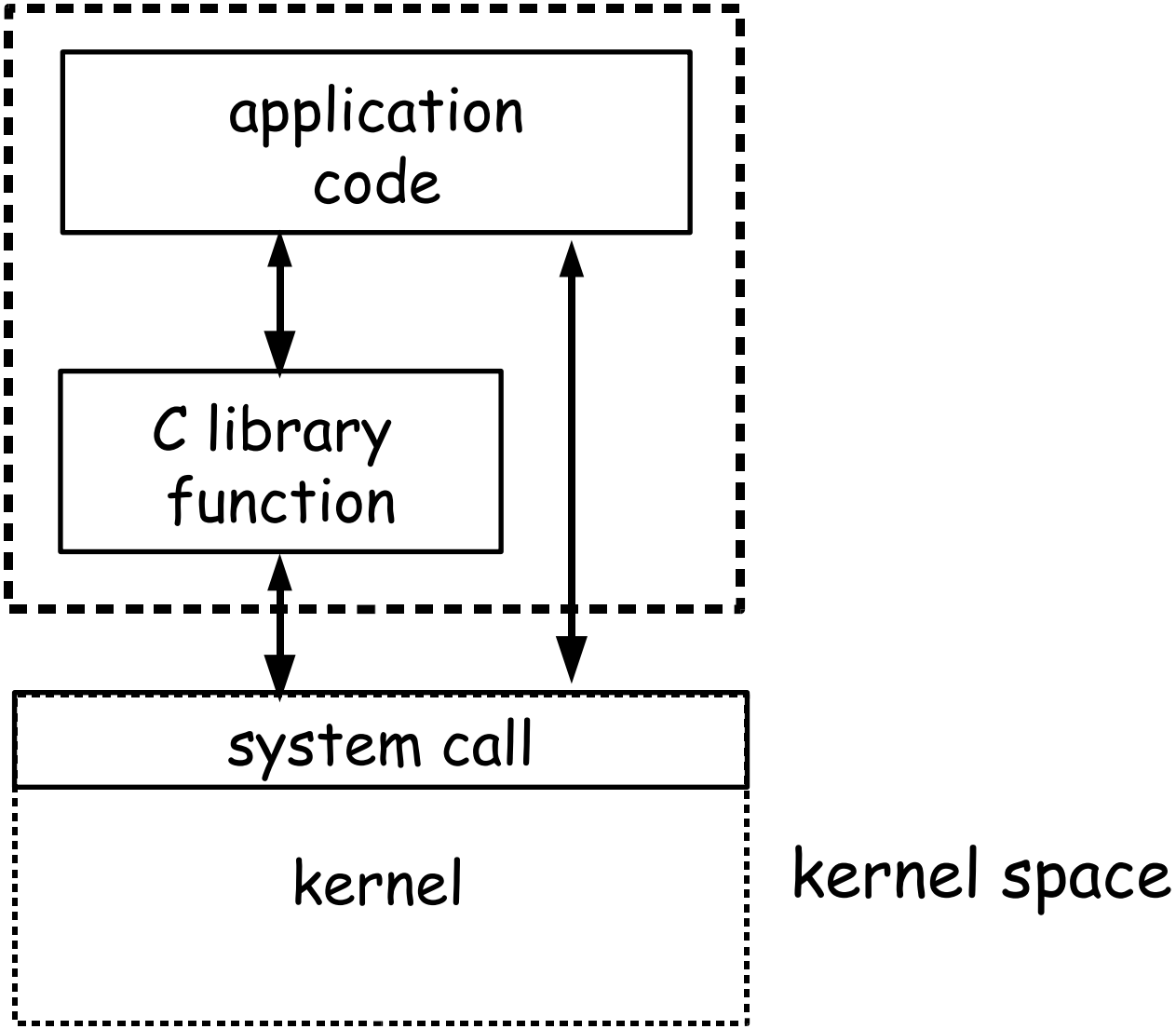
  invece

  - la funzione `strcpy` (string copy) e la funzione `atoi` (convert ASCII to integer) non coinvolgono alcuna system call

Systema call vs funzione di libreria (I)

- **Distinzione:**
  - Dal punto di vista del programmatore, non vi sono grosse distinzioni fra le funzioni di libreria e le system call: entrambe sono funzioni
  - Dal punto di vista di chi implementa un sistema operativo, la differenza è notevole
- **Nota:**
  - In generale è possibile sostituire le funzioni di libreria con altre funzioni che realizzino lo stesso compito, magari in modo diverso (ad esempio potremmo riscrivere una funzione tipo **`atoi`**)
  - Mentre non è possibile sostituire le system call, che dipendono dal sistema operativo e sono eseguite da codice all'interno del kernel

System call vs funzione di libreria (II)



Cause di errore nelle system call

- Le system call possono fallire per vari motivi:
  - Il sistema ha esaurito la disponibilità di una certa risorsa (o la richiesta ha superato il limite consentito per singolo processo)
  - Il kernel blocca l'esecuzione in quanto il processo non ha i permessi per richiedere l'esecuzione di una certa operazione
  - Gli argomenti passati ad una system call non sono validi
  - Un dispositivo HW non è funzionante o è mancante
  - Una interruzione (*interrupt*) causata da un evento esterno, come la ricezione di un segnale, fa terminare prematuramente la system call
- **Per tutti questi motivi è importante sempre rilevare e, se necessario, gestire un eventuale errore di ritorno dalla system call**

Codici di errore delle system call

- La maggior parte delle system call restituisce il valore -1 in caso di errore (0 in caso di successo) ed assegna un codice di errore più specifico alla variabile globale:

```
extern int errno;
```

Nota: se la system call ha successo, **`errno`** non viene resettato

- L'include file **`errno.h`** contiene la definizione dei nomi simbolici dei codici di errore:

```
#define EPERM 1      /* Operation not permitted */
#define ENOENT 2     /* No such file or directory */
#define ESRCH 3      /* No such process */
#define EINTR 4      /* Interrupted system call */
#define EIO   5      /* I/O error */
#define ENXIO 6      /* No such device or address */
.......................................................
```

Gestione dell'errore

- L'azione da programmare in risposta ad un errore può consistere in una o più delle seguenti possibilità:
  - visualizzazione del messaggio di errore
  - cancellazione della procedura in corso
  - terminazione del programma
  - tentativo di nuova esecuzione della system call
  - ignorare l'errore e proseguire
- La funzione **`perror`** può essere utilizzata per visualizzare una descrizione dell'ultimo errore verificatosi
- **`void perror(const char *str)`** converte il codice in **`errno`** in un messaggio, e lo stampa anteponendogli il messaggio di errore **`str`**
- **Esempio:**

```
fd=open("nonexist.txt", O_RDONLY);
if (fd==-1) perror ("main");
-->  main: No such file or directory
```

Tipi di system call

- Esistono vari tipi di system call relative a:
  - controllo di processi
  - gestione della memoria
  - gestione dei file e dei file system
  - segnali
  - comunicazione tra processi (IPC=Inter Process Communication)
  - networking
- In questo corso studieremo in modo completo i primi 4 gruppi di system call e alcuni meccanismi base di IPC

NOTA: **`man 2 intro`** e **`man 2 syscalls`** danno alcune informazioni sulle system calls disponibili sotto Linux.

Standardizzazione delle system call

- Al fine di facilitare il porting di applicazioni tra sistemi operativi differenti si è definito uno standard di riferimento per le system call che un kernel dovrebbe offrire al programmatore
- *POSIX* (Portable Operating System Interface) è una famiglia di standard sviluppato dall'IEEE:
  - IEEE 1003.1 (POSIX.1)
    Definizione delle system call
  - IEEE 1003.2 (POSIX.2)
    Shell e utility
  - IEEE 1003.7 (POSIX.7)
    System administration