Panoramica su UNIX

Laboratorio Sistemi Operativi

Giuseppe Salvi
Email: giuseppe.salvi@uniparthenope.it

Informazioni

- Orari del corso
  - Martedì (Aula 4)
    - 11:00 – 13:00
  - Giovedì (Aula 4)
    - 11:00 – 13:00

Testi di riferimento

- Il libro di testo del corso è:
  - W.R. Stevens, S.A. Rago, Advanced Programming in the UNIX Environment, 3° Edition, Addison-Wesley
- OK, una qualsiasi delle precedenti versioni
  - Le diverse versioni sono del tutto equivalenti

Materiale didattico

- Lucidi delle lezioni disponibili sulla piattaforma e-learning

https://elearning.uniparthenope.it/

- File in formato pdf
- Tracce (alcune) prove di laboratorio sessioni di esami precedenti

Panoramica su UNIX

- Introduzione
  - Storia
  - Standard
- Autenticazione
  - Logging in
  - Identificazione degli utenti
  - Shell
- File e directory
- Input ed output
- Programmi e processi
- Gestione degli errori
- Segnali
- Chiamate di sistema e funzioni di libreria

Introduzione

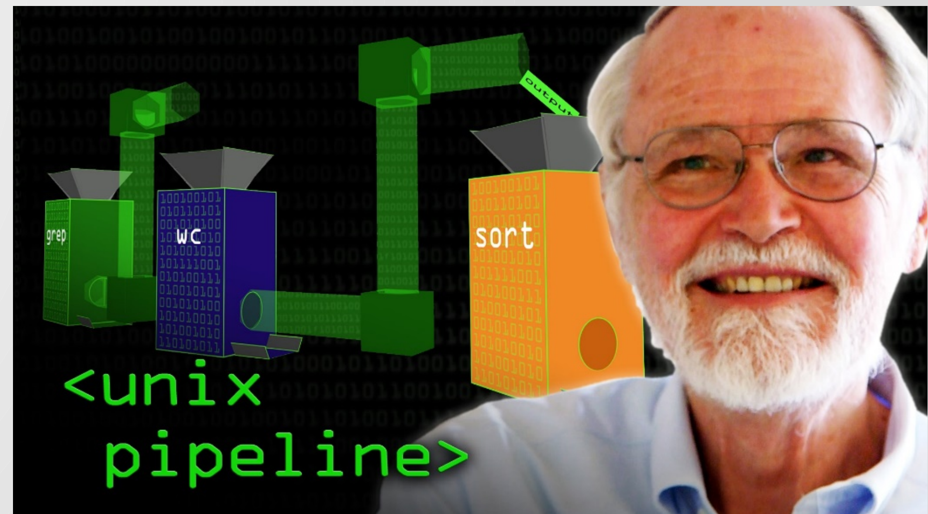
- Tutti i sistemi operativi forniscono servizi
- Eseguire un nuovo programma, aprire un file, allocare la memoria sono esempi di servizi forniti da un sistema operativo
- Descriveremo alcuni servizi forniti da varie versioni del sistema operativo Unix, ma prima ...

Il mondo UNIX

- UNIX è un sistema operativo multiutente e multitasking
  - Più utenti possono avere vari task/processi che sono eseguiti contemporaneamente
- UNIX è anche un ambiente di sviluppo software
  - È nato ed è stato progettato per questo scopo (ad esempio fornisce utilità di sistema, editor, compilatori, assemblatori, interpreti, . . .)
- UNIX è stato appositamente progettato per essere generale e indipendente dall’hardware (facilmente portabile essendo scritto nel linguaggio ad alto livello C)

Storia (1)

- 1964: La AT&T, il MIT e la General Electric si uniscono nel progetto di un SO innovativo, che sia multiutente, multitasking, multiprocessore, con file system gerarchico: MULTICS
- 1969: Il progetto MULTICS “fallisce”
- Presso i Bell Laboratories della AT&T, dalle ceneri di MULTICS, nasce UNIX, sviluppato da Ken Thompson e Dennis Ritchie del gruppo Multics per un PDP-7
  - Anche sulla base dell’esperienza MULTICS, incorpora alcune caratteristiche e supera i limiti di sistemi operativi preesistenti, supportando, in modo efficiente, multiutenza e multitasking
- Il nome UNIX (inizialmente Unics) è scelto da Kernighan in opposizione a MULTICS



Storia (2)

- 1973: La terza versione viene scritta nel linguaggio di programmazione ad alto livello C (anziché in assembler), sviluppato ai Bell Labs da Dennis Ritchie (come evoluzione di B) proprio per supportare UNIX
- È un passo importante dato che esisteva la convinzione che solo il linguaggio Assembly permettesse di ottenere un livello di efficienza accettabile
- Fine anni ’70: AT&T rende UNIX largamente disponibile, offrendolo alle Università a basso costo e distribuisce i sorgenti di varie versioni (costretta da un decreto dell’antitrust che le impone di scegliere tra telecomunicazioni e informatica)

Storia (3)

- 1984: presso l'Università di Berkeley (California) nasce la Berkeley Software Distribution (BSD)
  - versione 4BSD: sviluppata con una sovvenzione del DARPA (Defense Advanced Research Project Agency) al fine di progettare uno standard UNIX per uso governativo
  - versione 4.3BSD: sviluppata originariamente per il VAX, è una delle versioni più influenti. Verrà "portata" su molte altre piattaforme
- 1984: molte caratteristiche della BSD vengono incorporate nella nuova versione di AT&T: la System V
- Esistono oggi diverse implementazioni di UNIX supportate da molte case costruttrici di computer: SunOS/Solaris (Sun Microsystems), Ultrix (DEC), XENIX (Microsoft), AIX (IBM), . . .

Sun Solaris e OpenSolaris

- Solaris è la versione UNIX sviluppata da Sun Microsystems
  - E’ basata sulla System V Release 4, con più di 10 anni di evoluzioni da parte degli ingegneri Sun
  - E’ l’unica versione commerciale di successo che discende da SVR4 ed è certificata formalmente quale sistema UNIX
  - L’ultima versione è la Solaris 10
- Sebbene Solaris sia storicamente software proprietario, a partire dal 31 gennaio 2005 alcuni suoi componenti sono stati rilasciati sotto licenza open source CDDL (Common Development and Distribution License) all'interno del progetto definito OpenSolaris. Le future versioni di Solaris saranno basate su OpenSolaris

Storia di “free” UNIX

- 1983: Richard Stallmann (MIT) inizia una re-implementazione “free” di UNIX, detta GNU (acronimo ricorsivo di “GNU is not UNIX”) che porterà a un editor (emacs), un compilatore (gcc), un debugger (gdb) e numerosi altri tool
- 1991: Linus Torvald (studente ad Helsinki) inizia l’implementazione di un kernel (POSIX compatibile) che prenderà il nome di Linux
- Arricchito con i tool GNU esistenti ed altri sviluppati da volontari, diviene un sostituto completo di UNIX
- 1991: L’ Università di Berkeley rilascia una versione “free” di UNIX, rimuovendo il codice proprietario rimanente della AT&T. Progetti volontari continueranno poi il suo sviluppo (FreeBSD, NetBSD, OpenBSD)

Filosofia di UNIX

- Influenzata da problematiche relative all'HW disponibile all'epoca della progettazione di UNIX: i dispositivi di I/O erano telescriventi e più tardi terminali video con 80 caratteri per linea (ASCII fixed width)
  - Comandi, sintassi di input e output succinti (per ridurre il tempo di battitura)
  - Ogni componente/programma/tool realizza una sola funzione in modo semplice ed efficiente. Es. who e sort
  - L'output di un programma può diventare l'input di un altro
    - Programmi elementari possono essere combinati per eseguire compiti complessi (Es. who | sort)
  - Il principale software di interfaccia (la shell) è un normale programma, sostituibile, senza privilegi speciali
  - Supporto per automatizzare attività di routine

Il successo di UNIX

- Ha dimensioni relativamente ridotte, un progetto modulare e “pulito”
- Indipendenza dall’hardware
  - Il codice del SO è scritto in C anziché in uno specifico linguaggio assembler
  - UNIX e le applicazioni UNIX possono essere “facilmente” portate da un sistema hardware ad un altro. Il porting usualmente consiste nel trasferire il codice sorgente e ricompilarlo
- È un ambiente produttivo per lo sviluppo del software
  - Ricco insieme di tool disponibili
  - Linguaggio di comandi versatile

Standard

- La proliferazione di diverse versioni di UNIX durante gli anni ‘80 rese necessario l’inizio di una standardizzazione da parte del governo USA
  - C (American National Standard Institute nel 1989, poi ISO – International Organization for Standardization)
    - Ha lo scopo di fornire la portabilità dei programmi conformi in C ad un’ampia gamma di sistemi operativi, non solo di tipo UNIX. Definisce anche la libreria standard
  - POSIX (IEEE dal 1988, poi ISO) Portable Operating System Interface for Unix poi: Portable Operating System Interface for Computer Environments
  - XPG (X/Open, dal 1989) “X/Open Portability Guide”
  - SVID (AT&T, 1989) “System V Interface Definition”

POSIX: gli standard principali

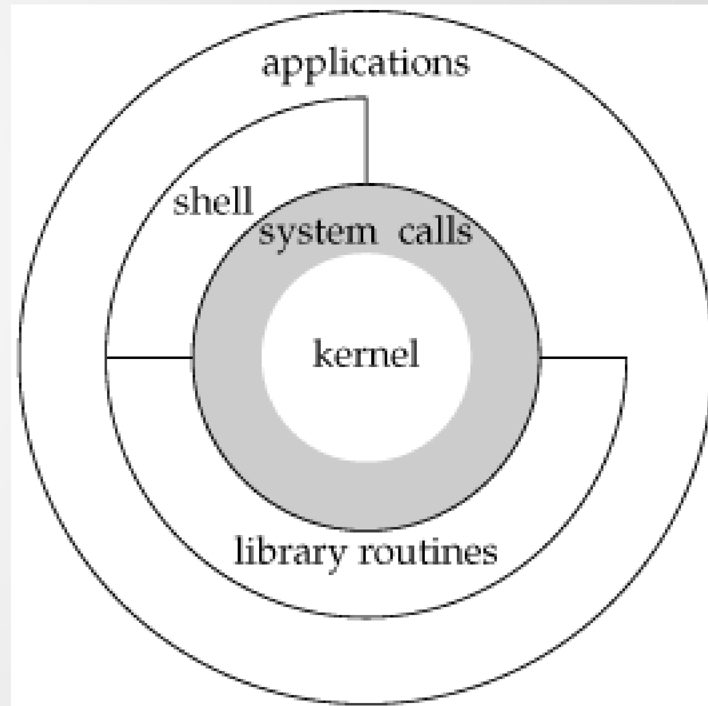
- 1003.1(POSIX.1): System Application Programming Interface (API)
  - dal 1990
  - definisce l'interfaccia fra programmi applicativi e S.O., nei termini di una libreria di funzioni
  - obiettivo: portabilità dei programmi fra sistemi conformi a POSIX (a livello sorgente)
- 1003.2 (POSIX.2): Shell and Utility Application Interface
  - dal 1992
  - definisce il linguaggio di shell e i principali comandi
  - obiettivo: portabilità degli script di shell fra sistemi conformi a POSIX

PER COMINCIARE …

Architettura di UNIX

- Un SO può essere definito come un software che controlla le risorse hardware del computer e fornisce un ambiente nel quale eseguire i programmi
  - Tale software è chiamato **kernel**
    - Relativamente piccolo
- L'interfaccia al kernel è uno strato di software chiamato **system call** (chiamate di sistema)
  - Le librerie di funzioni comuni sono costruite in cima all'interfaccia delle chiamate di sistema (le applicazioni, volendo, possono usare entrambe)
  - La **shell** è un'applicazione speciale che fornisce l'interfaccia con il kernel e che consente l'esecuzione di altre applicazioni

Architettura di UNIX (cont.)



AUTENTICAZIONE

Autenticazione

- Quando accediamo ad un sistema Unix forniamo la nostra login e password
- La nostra login viene cercata nel file di password del sistema (es. /etc/passwd)
  - All'interno del file notiamo che ciascuna entrata è costituita da sette campi: *nome di login*, *password* criptata, *ID utente* numerico, *ID gruppo* numerico, un campo *commento*, la *home directory* ed il programma di *shell*

```
$ grep giusal /etc/passwd
giusal:x:1000:1000: Giuseppe Salvi:/home/giusal:/bin/bash
$
```

Identificazione degli Utenti

- Ciascun utente appartiene ad un gruppo il cui identificativo viene detto group ID
- Più utenti possono appartenere allo stesso gruppo (es. studenti, staff, ospiti,...)
- Lo scopo dei gruppi è di raggruppare utenti per far condividere le risorse
- La corrispondenza tra identificativi numerici e gruppi si trova di solito nel file /etc/group

Identificazione degli utenti

- Lo user ID è il numero che identifica univocamente ciascun utente nel sistema
- Il superutente ha user ID uguale a 0
- Esempio

E’ necessario includere un file header (contenuto nei sorgenti che scaricherete). Esso contiene alcuni header di sistema standard e definisce costanti e prototipi di funzioni.

```
#include "apue.h"
int main(void)
{
  printf("uid = %d, gid = %d\n", getuid(), getgid());
  exit(0);
}
```

```
$ ./a.out
uid = 205, gid =105
```

Logging in

- Una volta eseguito il login, possiamo dare comandi al sistema, utilizzando un interprete dei comandi (shell)
- Una shell è un interprete che accetta l'input dell'utente ed esegue i comandi
- Ciascun utente ha una shell che viene attivata di default
- Tra le shell più utilizzate ci sono:
  - La Bourne shell, /bin/sh (Steve Bourne, Laboratori Bell)
  - La C shell, /bin/csh (Bill Joy, Berkeley)
  - La Korn shell, /bin/ksh (David Korn, Laboratori Bell)
  - La Bourne Again shell, /bin/bash (shell GNU)

FILE E DIRECTORY

File e Directory

- Il filesystem di Unix è un insieme di file e directory organizzato in maniera gerarchica
- La radice (root) del filesystem è una directory rappresentata da /
- Una directory è una tabella che contiene un nome e un puntatore ad una struttura di informazioni per ciascun file o directory in essa contenuto
- Gli attributi in tale struttura riguardano:
  - tipo di file,
  - dimensione,
  - proprietario,
  - permessi,
  - ...

Esempio: elencare i file di una directory (comando ls)

```
#include <sys/types.h>
#include <dirent.h>
#include "apue.h"
int main(int argc, char *argv[ ])
{
  DIR *dp;
  struct dirent *dirp;
  if (argc != 2)
       err_quit("a single argument (the dir name) is required");
  if ( (dp = opendir(argv[1])) == NULL)
     err_sys("can't open %s", argv[1]);
  while ( (dirp = readdir(dp)) != NULL)
     printf("%s\n", dirp->d_name);
  closedir(dp);
  exit(0);
}
```

Esempio (cont.)

- Eseguendo il programma si ottiene, ad esempio:

```
$ ./a.out /dev
.
..
console
tty
mem
kmem
null
mouse
stdin
stdout
stderr
…..
```

- Un'altra possibile esecuzione:

```
$ ./a.out /dev/tty
Can't open /dev/tty:
Not a directory
```

INPUT/OUTPUT

File I/O

- Quando un programma apre un file o lo crea, il kernel ritorna un descrittore utilizzato per tutte le successive operazioni di lettura e scrittura
- I descrittori di file sono degli interi non negativi che identificano i file utilizzati da un particolare programma
- Quando un programma viene mandato in esecuzione, la shell apre tre descrittori:
- 0 *std input*, 1 *std output*, 2 *std error*
- Di norma questi descrittori sono collegati con il terminale
  - Tuttavia è possibile redirigerli tutti verso un file
    - `ls > file.list`

I/O non bufferizzato

- L'I/O non bufferizzato è fornito dalle funzioni
  - `open`,
  - `read`,
  - `write`,
  - `lseek`,
  - `close`
- Tutte lavorano con i descrittori di file

Esempio: leggere dallo standard input e scrivere sullo standard output

```
#include "apue.h"
#define BUFFSIZE 8192

int main(void){
 int n;
 char buf[BUFFSIZE];

 while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");

 if (n < 0)
   err_sys("read error");

 exit(0);
}
```

Esempio (cont.)

- Se compiliamo il programma e lo eseguiamo:

```
./a.out > data
```

- Lo standard input è il terminale, lo standard output è rediretto nel file data, e lo standard error è il terminale
- Il programma copia le linee di caratteri che inseriamo nello standard output fino a che inseriamo il carattere di fine file (solitamente Control-D)
- Se eseguiano

```
./a.out < infile > outfile
```

Il file chiamato `infile` è copiato nel file chiamato `outfile`

Standard I/O

- Le funzioni di I/O standard forniscono un'interfaccia bufferizzata alle funzioni di I/O non bufferizzato
- Usare le funzioni di I/O standard (i cui prototipi sono nello header `<stdio.h>`) consente di non preoccuparsi della scelta ottimale della dimensione del buffer (come ad esempio la costante `BUFFSIZE`)
- Un ulteriore vantaggio consiste nel rendere più semplice il trattamento delle linee di input
  - La funzione `fgets`, ad esempio, legge un'intera linea
  - Mentre la funzione `read` legge un numero di byte specificato

Esempio: leggere dallo standard input e scrivere sullo standard output

```
#include<stdio.h>

int main(void){
  int c;

  while ((c = getc(stdin)) != EOF)
   if (putc(c,stdout) == EOF){
     printf(“errore di output\n”);
     exit(1);
    }

  if (ferror(stdin)){
    printf(“errore di input\n”);
    exit(1);
  }

  exit(0);
}
```

PROGRAMMI E PROCESSI

Programmi e Processi

- Un programma è un file eseguibile che risiede nel filesystem
- Esso viene caricato in memoria ed eseguito dal kernel quando viene eseguita una chiamata ad una delle funzioni di exec
- Un programma in esecuzione viene detto processo
- Ciascun processo è identificato univocamente da un intero non negativo

Esempio

```
#include "apue.h"

int main(void)
{
  printf("hello world from process ID %d\n", getpid());
  exit(0);
}
```

```
$ ./a.out
hello world from process ID 851
$ ./a.out
Hello world from process ID 854
```

Controllo dei processi

- Ci sono tre funzioni principali per il controllo dei processi:
  - `fork`,
  - `exec`,
  - `waitpid`
- Vediamo un esempio di programma che legge i comandi dallo standard input e li esegue

Esempio

```
#include <sys/types.h>
#include <sys/wait.h>
#include "apue.h"

int main(void){
  char buf[MAXLINE];
   pid_t pid;
   int status;

  printf("%% "); /* print prompt (printf requires %% to print %) */

  while (fgets(buf, MAXLINE, stdin) != NULL) {
     buf[strlen(buf) - 1] = 0; /* replace newline with null */

     if ( (pid = fork()) < 0)
       err_sys("fork error");
     else if (pid == 0) { /* child */
       execlp(buf, buf, (char *) 0);
       err_ret("couldn't execute: %s", buf);
       exit(127);}
     if ( (pid = waitpid(pid, &status, 0)) < 0)
       err_sys("waitpid error");
       printf("%% ");
   } exit(0);}
```

Esempio (cont.)

```
$ ./a.out
% date
Sun Mar 1 03:04:47 EDT 2009
% pwd
/home/giusal/prova
% ls
Makefile
a.Out
Prova.c
% ^D
$
```

GESTIONE ERRORI

Gestione degli Errori

- Per segnalare una situazione di errore, le funzioni di Unix spesso ritornano un valore negativo e l'intero **`errno`** è inizializzato ad un valore che fornisce informazioni ulteriori

```
extern int errno;
```

- Nell'header `<errno.h>` si trova la corrispondenza tra i valori che `errno` può assumere e le costanti numeriche ad essi associate

Gestione degli Errori

- Le funzioni del C per gestire gli errori sono:

```
#include <string.h>

char *strerror(int errnum);
```

  - che associa ad `errnum` un messaggio di errore, e

```
#include <stdio.h>

void perror(const char *msg);
```

  - che stampa la stringa puntata da `msg`, seguita da due punti, uno spazio e il messaggio di errore, seguiti da un carattere di nuova linea

Esempio

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

int main(int argc, char *argv[]){
    fprintf(stderr, “EACCES: %s\n”, strerror(EACCES));
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}

$ ./a.out
EACCES: Permission denied
./a.out: No such file or directory
```

SEGNALI

Segnali

- Una tecnica per notificare ad un processo l'occorrenza di una certa situazione è quella di utilizzare i segnali

- Se, ad esempio, comunichiamo ad un processo un *Ctrl+C* da tastiera, esso riceve il segnale SIGINT

- Un processo che riceve un segnale può:
  - ignorare il segnale
  - lasciare che venga eseguita l'azione di default
  - fornire una funzione da eseguire all'atto della ricezione del segnale

CHIAMATE DI SISTEMA

Chiamate di sistema

- Tutti i sistemi operativi forniscono interfacce attraverso cui i programmi richiedono servizi
  - Per utilizzare i servizi offerti da UNIX, quali creazione di file, duplicazione di processi e comunicazione tra processi, i programmi applicativi devono interagire con il sistema operativo
- Tutte le implementazioni di Unix forniscono un insieme definito e limitato di punti di accesso al kernel, detti chiamate di sistema

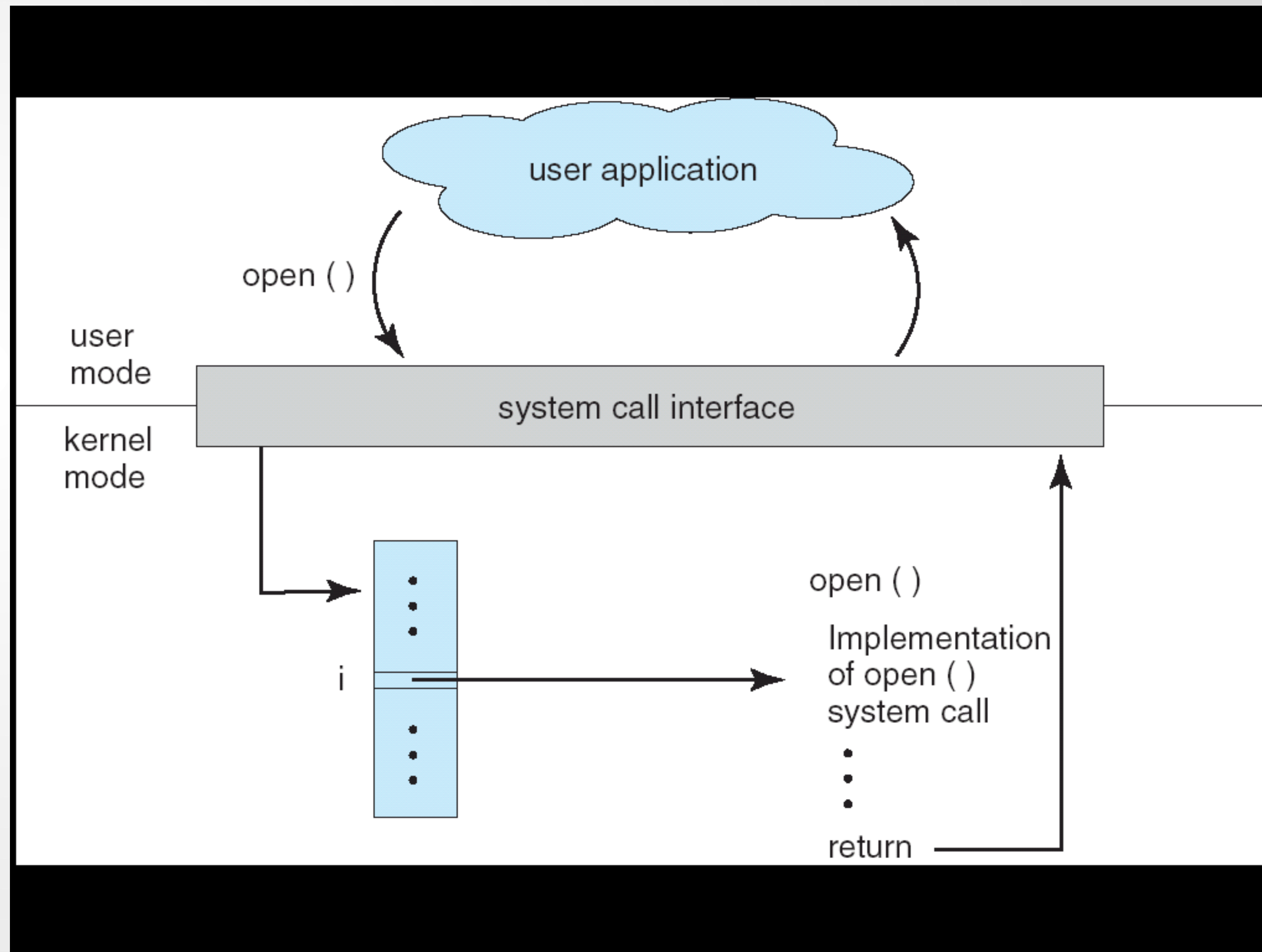
Chiamate di Sistema e Funzioni di Libreria

- Queste funzioni mettono a disposizione del programmatore di sistema dei servizi che, una volta invocati, vengono eseguiti dal kernel
- In particolare, il codice delle chiamate di sistema è parte del kernel stesso: ogni processo che ne richiede l'esecuzione, trasferisce il controllo al kernel, che effettua il servizio
- Sopra il livello delle chiamate di sistema è collocata una collezione di librerie C, il cui ruolo fondamentale è quello di fornire un'interfaccia C per le chiamate di sistema:
  - ogni applicazione C può quindi richiedere l'esecuzione di una chiamata di sistema, attraverso una chiamata alla specifica funzione C di libreria che la rappresenta

Chiamate di sistema

- Le chiamate di sistema sono una sorta di “entry point” per il kernel
- Il programmatore chiama la funzione utilizzando la sintassi usuale delle funzioni C
  - int open(const char *path, int mode)
- La funzione invoca, nel modo opportuno, il servizio del sistema operativo
- “salva” gli argomenti della system call ed un numero identificativo della system call stessa (in registri)
- esegue una istruzione macchina trap …

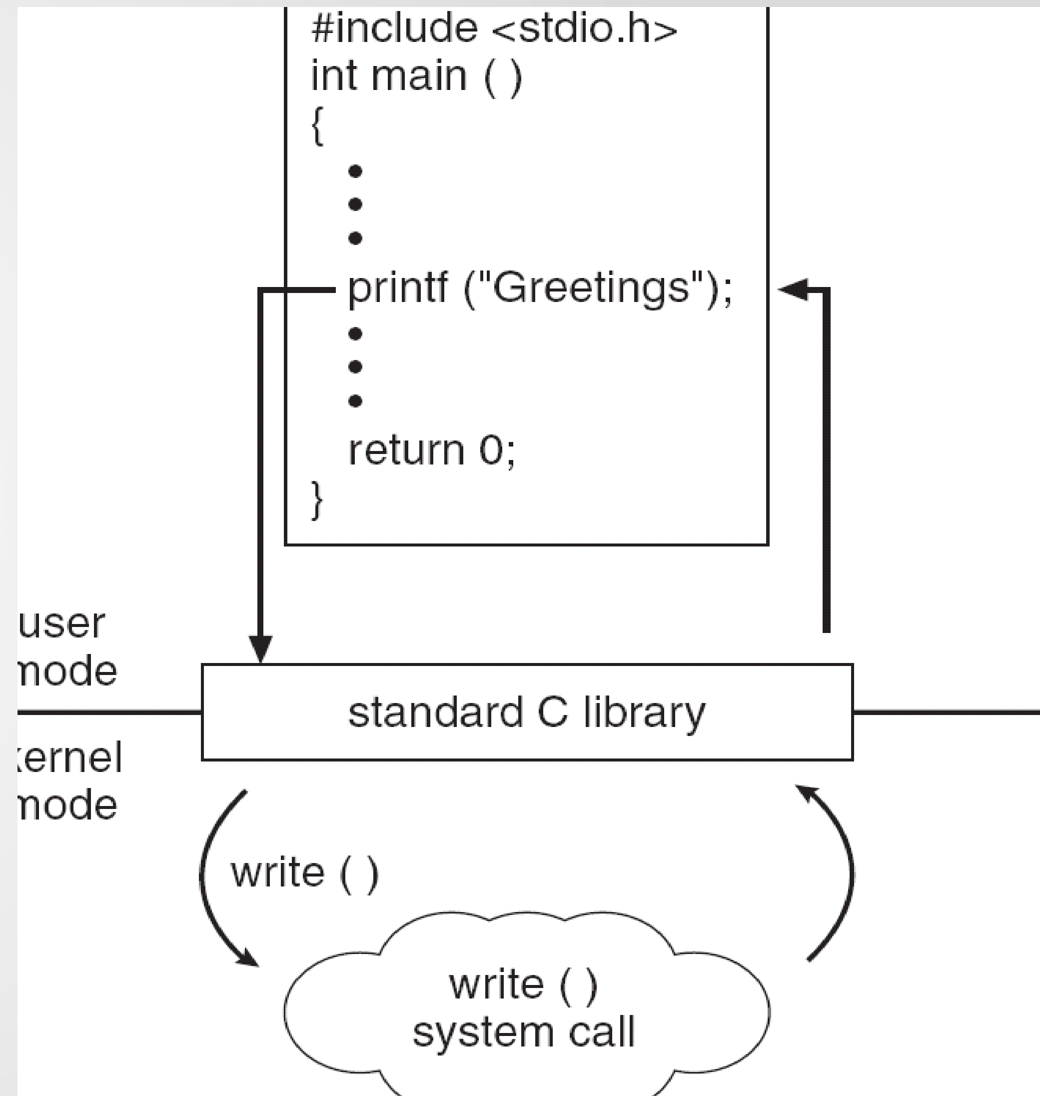
Relazione tra API – chiamate di sistema e sistema operativo



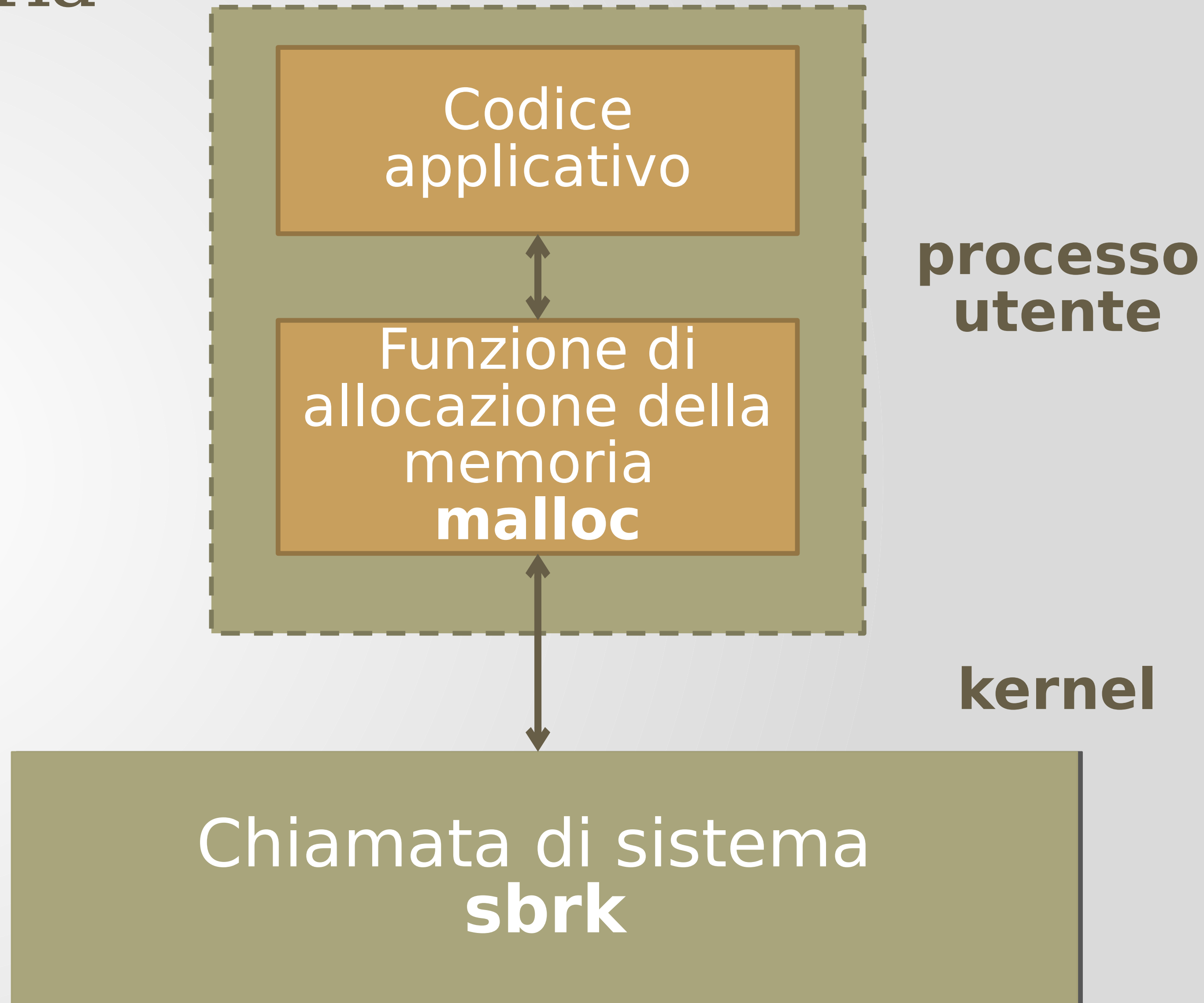
Funzioni di libreria

- Le funzioni di libreria forniscono servizi di utilità generale al programmatore
- Non sono entry point del kernel, anche se possono fare uso di system call per realizzare il proprio servizio
- Es:
  - **printf** può utilizzare la system call **write** per stampare
  - **strcpy** (string copy) e **atoi** (convert ASCII to integer) non coinvolgono il sistema operativo
- Possono essere sostituite con altre funzioni che realizzano lo stesso compito (in generale non possibile per le system call)

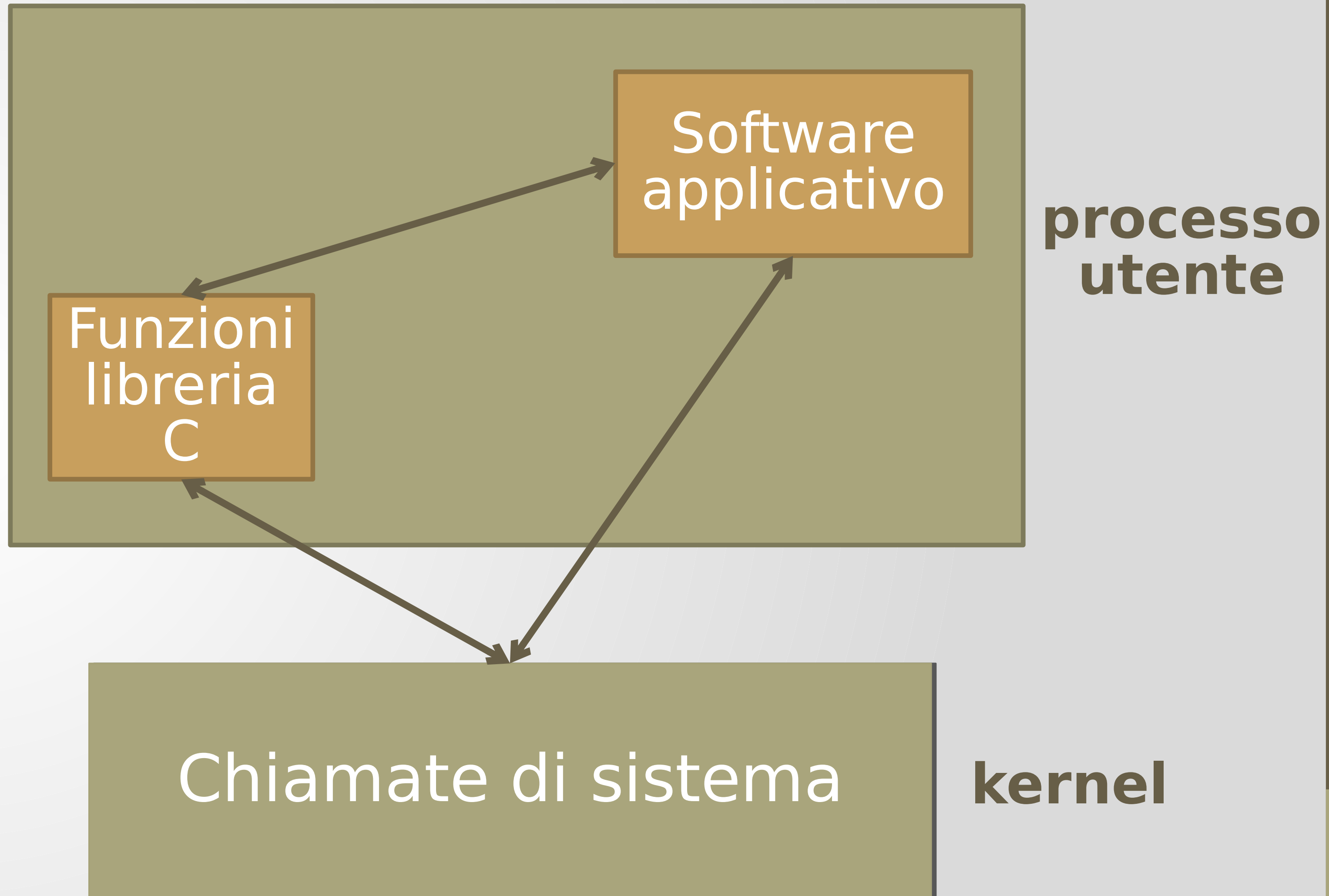
Esempio Standard C Library



Chiamate di Sistema e Funzioni di Libreria



Chiamate di Sistema e Funzioni di Libreria



Tassonomia delle system call

- Le chiamate di sistema coinvolgono attività che possono essere raggruppate in tre categorie principali:
  - gestione dei file,
  - gestione degli errori,
  - gestione dei processi
- La comunicazione tra processi (IPC – interprocess communication) rientra nella gestione dei file poiché UNIX tratta i meccanismi per IPC come file speciali
- Per utilizzare le system call sono forniti opportuni file di intestazione. Ad es.
  - sys/file.h
  - errno.h