

Sistemi Operativi

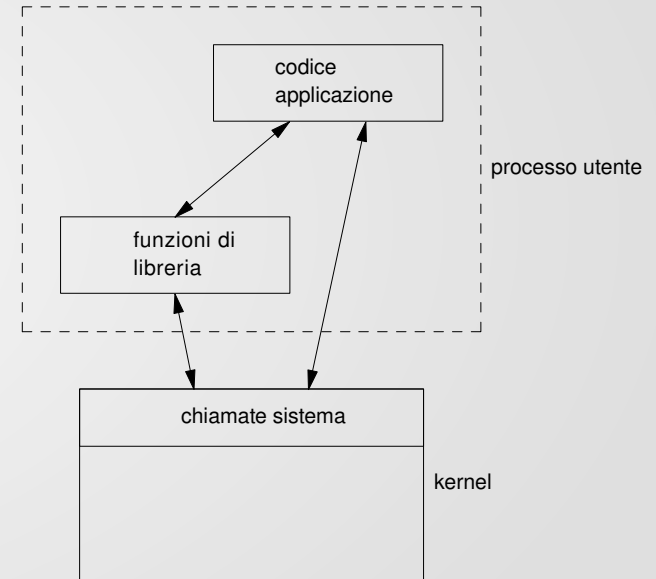
C.d.L. in Informatica (laurea triennale)
Anno Accademico 2022-2023

Laboratorio di Sistemi Operativi

Dipartimento di Matematica e Informatica – Catania

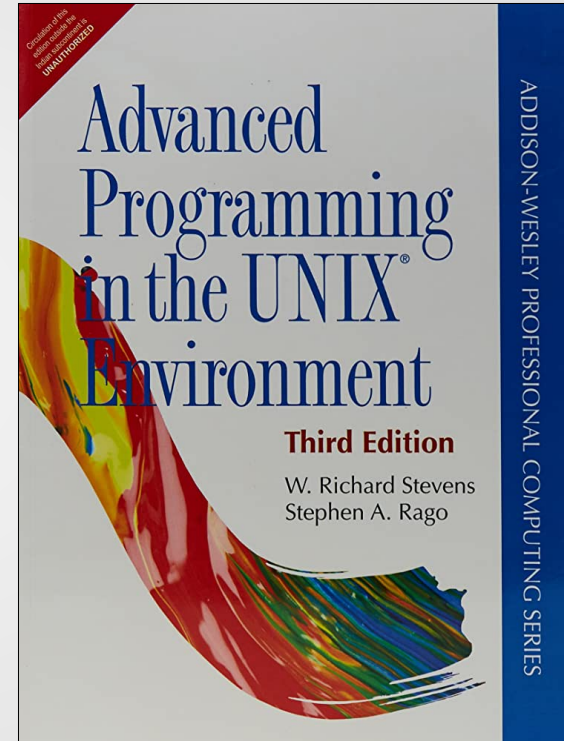
Chiamate di Sistema

- Nei nostri programmi possiamo impiegare:
 - **chiamate di sistema:** servizi offerti direttamente dal Sistema Operativo (TRAP, modalità kernel)
 - **chiamate di libreria:** funzioni incluse in libreria di sistema (modalità utente)
- nel corso ci occuperemo dei seguenti sotto-sistemi:
 - gestione dei **file system** (file e directory) e dell'**I/O**
 - gestione dei **processi**
 - gestione dei **thread**
 - **comunicazione e sincronizzazione** di processi e thread



Materiale di Riferimento

- Manuale di riferimento:
 - ***Advanced Programming in the UNIX Environment*** (terza edizione – 2013) di Stevens e Rago
- è possibile utilizzare qualunque altro testo o risorsa web che tratti l'argomento
- altre valide alternative:
 - ***Programming With POSIX Threads*** di Butenhof
 - ***PThreads Primer: A Guide to Multithreaded Programming*** di Lewis e Berg



Documentazione

- Le pagine di manuale (**man pages**) UNIX rappresentano la documentazione ufficiale:
 - accessibile da:
 - ♦ **shell:** `man comando-o-funzione`
 - pacchetto `manpages-posix-dev` su Debian/Ubuntu
 - ♦ **online:** `man.cx` , `man7.org` , ...
 - **sezioni:**
 - ♦ n.1: comandi utente
 - ♦ n.2: chiamate di sistema
 - ♦ n.3: librerie di sistema
 - ♦ ...
 - ♦ n.8: comandi di amministrazione
 - casi di omonimia: `man chown` vs. `man 3 chown`

In caso di più pagine con lo stesso nome, si può specificare la sezione usando la sintassi: `man numero_sezione nome_pagina`

Standard

- L'uso degli **standard** è importante per creare codice che sia **portatile** (previa compilazione) su molteplici piattaforme e architetture.
 - **ISO C**: linguaggio e funzioni di libreria
 - **IEEE POSIX** (Portable Operating System Interface) Std 1003.1 e estensioni:
 - ♦ POSIX.1: interfacce di programmazione con sintassi C (sovrapp. con ISO C)
 - ♦ POSIX.2: comandi e utilità sulla shell UNIX
 - ♦ POSIX.4: estensioni real-time (tra cui i thread)
 - ♦ POSIX.6: sicurezza
 - ♦ POSIX.7: amministrazione di sistema
- **Supporto:**
 - **GNU/Linux**: alquanto completo (**piattaforma di riferimento** per il laboratorio)
 - ♦ con alcune estensioni **GNU** specifiche attive di default
 - **Windows**: parziale ma ampliabile con **Windows Subsystem for Linux** (WSL)
 - **MacOS**: alquanto completo

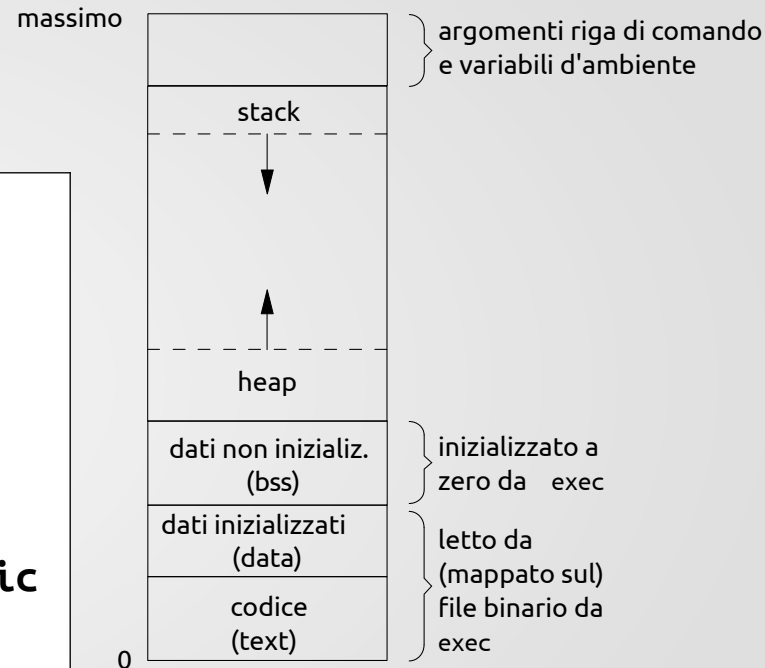
Creazione di Programmi Eseguibili

- Compilazione diretta di un solo sorgente:
 - `gcc -o nome-eseguibile sorgente.c`
- compilazione da sorgenti multipli:
 - `gcc -c file1.c ; gcc -c file2.c`
 - `gcc -o nome-eseguibile file1.o file2.o`
- linking con librerie: opzione `-l nome-libreria`
 - `gcc -l m -l pthread sorgente.c`
 - linking statico: opzione aggiuntiva `-static`
- specifica dello standard C da utilizzare: opzione `-std=`
 - `gcc -std=c99 sorgente.c`
- per progetti più articolati si può usare `make` e un progetto `makefile`
 - regole del tipo: `target ← dipendenze / comando`
 - esempio: `makefile.sample` 

Spazio di Indirizzamento Virtuale

- **dati inizializzati (data):**
 - `int max = 100;`
- **dati non inizializzati (bss)**
 - `int vector[50];`

```
$ size /usr/bin/gcc /bin/bash
text    data    bss      dec      hex filename
1028295  10184    15600  1054079  10157f  /usr/bin/gcc
$ gcc -o hello hello.c
$ gcc -static -o hello-static hello.c
$ ls -l hello hello-static
-rwxr-xr-x 1 mario mario  15416 26 mag 22.17 hello
-rwxr-xr-x 1 mario mario 733560 26 mag 22.17 hello-static
$ size hello hello-static
text    data    bss      dec      hex filename
1310    584      8      1902     76e hello
619340  20816   22624   662780   a1cfc hello-static
```



Gestione Standard degli Errori

- La maggior parte delle chiamate di sistema segnalano **errori** nell'esecuzione:
 - riportando un **valore di ritorno** anomalo (in genere -1)
 - impostando una **variabile globale** prestabilita:
 - ♦ `extern int errno;`
 - ♦ dichiarata nell'header `errno.h` (generalmente automaticamente inclusa)

<code>#define EPERM</code>	<code>1</code>	<code>/* Operation not permitted */</code>
<code>#define ENOENT</code>	<code>2</code>	<code>/* No such file or directory */</code>
<code>#define ESRCH</code>	<code>3</code>	<code>/* No such process */</code>
<code>#define EINTR</code>	<code>4</code>	<code>/* Interrupted system call */</code>
...		
 - ♦ nota: in caso di successo `errno` non viene resettata!
 - **errori fatali vs. non fatali** (ad es. `EINTR`, `ENFILE`, `ENOBUFFS`, `EAGAIN`, ...)

```
char *strerror(int errnum);   
void perror(const char *s); 
```

`<string.h>`, `<stdio.h>`

Terminazione del Processo

```
void exit(int status);   
int atexit(void (*func)(void)); 
```

<stdlib.h>

- **exit** termina il processo con **exit code** pari a (status && 0xFF)
 - convenzione UNIX (quasi universale): "0" = *"tutto ok"*, ">0" = *"errore"*
 - costanti apposite per maggiore portatilità: **EXIT_SUCCESS**, **EXIT_FAILURE**
- un processo termina anche quando:
 - il main ritorna (si può pensare a qualcosa tipo: `exit(main(argc, argv))`)
 - l'ultimo thread termina
- **exit** termina in *"modo pulito"* il processo:
 - scrivendo eventuali buffer in sospeso (vedi *stream* più avanti)
 - eseguendo eventuali procedure di chiusura registrate con **atexit**
- esempio: **at-exit.c** 

Descrittori di File

- Ogni processo può aprire uno o più file ottenendo un intero non negativo detto **descrittore di file** (*file descriptor*) come riferimento
- esistono tre **canali predefiniti** a cui è associato già un descrittore:
 - ♦ **standard input** (0)
 - ♦ **standard output** (1)
 - ♦ **standard error** (2)
- in **unistd.h** sono definite apposite costanti:
 - ♦ **STDIN_FILENO**, **STDOUT_FILENO** e **STDERR_FILENO**
- in esecuzioni dirette in genere sono associati al terminale ma non sempre:
 - ♦ **./my-prog > outfile.txt < input.txt**
 - ♦ **cat input.txt | ./my-prog | sort > output.txt**

Apertura, Creazione e Chiusura di un File

```
int open(const char *path, int oflag, [ mode_t mode ] );  
int creat(const char *path, mode_t mode );  
int close(int fd)
```

<fcntl.h>, <unistd.h>

- **open** apre (ed eventualmente crea) un file con percorso **path**
 - **oflag**: intero che può combinare alcuni flag per l'apertura:
 - ♦ **O_RDONLY** / **O_WRONLY** / **O_RDWR** (in modo mutuamente esclusivo)
 - ♦ **O_APPEND**: ogni scrittura avverrà alla fine del file
 - ♦ **O_CREAT**: crea il file se non esiste usando i permessi indicati in **mode**
 - ♦ **O_EXCL**: usato con **O_CREAT**, genera un errore se il file esiste già
 - ♦ **O_TRUNC**: se il file esiste, viene troncato ad una lunghezza pari a 0
 - ritorna: -1 in caso di errore o il descrittore del file appena aperto (≥ 0)
- **creat** equivale a: `open(path, O_RDWR | O_CREAT | O_TRUNC, mode)`
- **close** chiude un file aperto

NOTA: più costanti
predefinite possono
essere combinate
utilizzando l'operatore |

Permessi sugli Oggetti del File-System UNIX

- I permessi sono di triplice natura: **lettura** (R) / **scrittura** (W) / **esecuzione** (X)
- il tipo `mode_t` è un intero che codifica una **maschera con permessi** per:
 - **utente proprietario** (USR)
 - **gruppo proprietario** (GRP)
 - **tutti gli altri utenti** (OTH)
- la maschera si può ottenere da costanti definite in `sys/stat.h`:

<code>S_IRUSR</code>	<code>S_IWUSR</code>	<code>S_IXUSR</code>
<code>S_IRGRP</code>	<code>S_IWGRP</code>	<code>S_IXGRP</code>
<code>S_IROTH</code>	<code>S_IWOTH</code>	<code>S_IXOTH</code>

- è anche prassi, non raccomandata, utilizzare direttamente la **rappresentazione numerica ottale**: ad esempio: `0640` $\approx$ `S_IRUSR` | `S_IWUSR` | `S_IRGRP`
- per le **directory**: X rappresenta il **diritto di attraversamento**

Maschera di Creazione per i Permessi

- Quando un file (o una cartella) viene creato, la maschera specificata viene combinata con una **maschera di creazione** che inibisce globalmente alcuni permessi per ragioni di **sicurezza**
 - $maschera\text{-}effettiva = maschera\text{-}specificata \&\& (\sim maschera\text{-}creazione)$
- ogni processo ha la propria maschera di creazione che viene ereditata dai figli

```
mode_t umask(mode_t cmask);
```

<sys/stat.h>

- anche la **shell** ha propria maschera di creazione che può essere cambiata con l'omonimo comando (`umask`)
- esempio: `creation-mask.c`

Posizionamento

```
off_t lseek(int fd, off_t offset, int whence);
```

<unistd.h>

- ogni file aperto ha un **file offset** che simula l'accesso sequenziale
 - posto a 0 in apertura se non si è usato O_APPEND
 - aggiornato ad ogni operazione
- **lseek** posiziona effettua uno spostamento di **offset** byte rispetto a **whence**:
 - ♦ **SEEK_SET**: rispetto all'inizio del file
 - ♦ **SEEK_CUR**: rispetto alla posizione attuale (offset può essere negativo)
 - ♦ **SEEK_END**: rispetto alla fine del file (offset può essere negativo)
 - ritorna: -1 in caso di errore o la nuova posizione rispetto all'inizio del file (≥ 0)
 - non comporta alcuna operazione di I/O e valido solo su file
- ottenere la posizione attuale: `pos = lseek(fd, 0, SEEK_CUR);`
- esempio: **test-seek-on-stdin.c** 

Lettura e Scrittura

```
ssize_t read(int fd, void *buf, size_t nbytes);  
ssize_t write(int fd, const void *buf, size_t nbytes);
```

<unistd.h>

- **read** legge **nbytes** byte dal descrittore **fd** mettendoli su buffer **buf**
 - ritorna: -1 in caso di errore, 0 se siamo alla fine del file, altrimenti il numero di byte effettivamente letti (>0)
 - ♦ può leggere meno dati se il file sta finendo, se leggendo da terminali, pipe, socket di rete, a causa dei segnali, ...
- **write** legge **nbytes** byte dal buffer **buf** e li scrive sul descrittore **fd**
 - ritorna: -1 in caso di errore o il numero di byte trasferiti (≥0)
- esempi: **count.c**, **hole.c**, **copy.c**