

Sviluppo di software di rete con l'utilizzo di sistemi UNIX

Il sistema di INPUT/OUTPUT

Claudio Vicari

Sistema di I/O

- ➔ file descriptor: identificativo di un file utilizzato dal processo
- ➔ file descriptor fondamentali
 - standard input
 - standard output
 - standard error
- ➔ I/O non buffered
- ➔ I/O buffered: `stdio.h`

Funzioni di I/O di sistema

Esempio (prova_io1.c)

Primo esempio

Funzioni di I/O di sistema

open

```
int open( const char * pathname, int oflag,  
         ..., /* mode_t mode */ )
```

- ⇒ **pathname**: il percorso (completo o relativo) del file
- ⇒ Possibilità di aprire in lettura (`O_RDONLY`), in scrittura (`O_WRONLY`), lettura/scrittura (`O_RDWR`)
- ⇒ appendere, creare, troncare il file
- ⇒ restituisce il **file descriptor**, -1 se c'è un errore

Funzioni di I/O di sistema

close

```
int close( int filedes )
```

- ⇒ chiude il file, rilascia tutte le risorse associate compresi eventuali lock sul file
- ⇒ le risorse vengono comunque rilasciate al termine del programma
- ⇒ restituisce 0 in caso di successo, -1 altrimenti

Funzioni di I/O di sistema

lseek

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

- ➔ sposta l'offset di lettura/scrittura del file corrente
- ➔ offset specifica di quanti byte spostarsi
- ➔ whence può essere SEEK_SET, SEEK_CUR, SEEK_END per specificare se riferirsi all'inizio del file, alla posizione corrente o alla fine del file
- ➔ restituisce -1 in caso di errore, l'offset nel file altrimenti

Funzioni di I/O di sistema

read

```
ssize_t read( int fd, void * buff,  
              size_t nbytes )
```

- ➔ scrive il buffer specificato, per una lunghezza di `nbytes` byte
- ➔ restituisce il numero di byte letti, che può essere minore della lunghezza del buffer se:
 - il file è terminato
 - il dispositivo è atto a leggere solo una riga alla volta (terminali)
 - si sta leggendo un dispositivo di rete, e i buffer di rete vengono esauriti momentaneamente

Funzioni di I/O di sistema

write

```
int write( int filedes, const void * buff,  
          size_t nbytes )
```

- ⇒ scrive il buffer nel file specificato
- ⇒ restituisce il numero di byte scritti
- ⇒ in caso di errore, può scrivere meno byte del previsto, per esempio se il disco in cui si scrive è pieno

Funzioni di I/O di sistema

condivisione e atomicità

- ➔ i file possono essere utilizzati contemporaneamente fra più processi
- ➔ chi programma deve essere in grado di governare il flusso delle operazioni
- ➔ esempio: due processi che fanno 'append' sullo stesso file
- ➔ scrivere un file aperto con `O_APPEND` è un'operazione atomica
- ➔ aprire un file e quindi fare `lseek` alla fine del file non è atomico

Strumenti di controllo: lsof

- ➔ UNIX fornisce un comando per controllare quali file sono aperti: `lsof`
- ➔ `lsof` elenca tutti i file aperti nel sistema: file normali, a blocchi, socket locali e di rete
- ➔ ha molte opzioni per visualizzare tipi specifici di file aperti da mostrare
- ➔ esempio:
 - `lsof +D /tmp/`
 - `lsof +D ~vicari`
- ➔ file in `/dev/fd`: l'elenco di tutti i file descriptor aperti! E' utile soprattutto da linea di comando....

*La libreria **STDIO***

- ➔ Non si usano i fd ma oggetti di tipo `FILE`
- ➔ Si parla di **stream** invece che di fd, perché lo stream ha altre strutture associate
 - In particolare, vi è un buffer per i dati da leggere/scrivere nel file
- ➔ stream predefiniti: `stdin`, `stdout`, `stderr`
- ➔ Gli stream quindi possono usare i buffer: i dati vengono scritti solo quando il buffer è pieno
 - L'efficienza dell'uso aumenta notevolmente!
 - la libreria si prende cura di questi dettagli

*La libreria **STDIO***

- ⇒ funzioni per aprire stream:
 - `fopen` per `pathname`, `fdopen` per file descriptor
- ⇒ funzioni per posizionarsi:
 - `fseek`, `fsetpos`, `rewind`
- ⇒ chiusura di file, finalizzazione delle scritture:
 - `fclose`, `fflush`
- ⇒ funzioni di I/O non formattato:
 - un carattere per volta: `getc`, `fgetc`, `putc`, `fputc`
 - una linea per volta: `fgets`, `fputs`
 - diretto: `fread`, `fwrite`
- ⇒ controllo di errori o fine del file:
 - `feof`, `ferror`

La libreria `STDIO`: output formattato

➔ output:

- `printf(const char * format, ...)`
- `fprintf(FILE * fp, const char * format, ...)`
- `sprintf(char * buf, const char * format, ...)`
- `snprintf`
 - controllare sempre la validità dell'input! `snprintf` è consigliata...

- ## ➔ Consentono di scrivere dati, con estrema flessibilità riguardo al formato con cui questi dati vengono scritti

Esempio di `printf`

*La libreria **STDIO**: input formattato*

➔ input:

- `scanf(const char * format, ...)`
- `fscanf(FILE * fp, const char * format, ...)`
- `sscanf(const char * buf, const char * format, ...)`

➔ ricevono i puntatori alle variabili come argomenti

➔ restituiscono:

- EOF in caso di fine stream
- il numero di elementi letti in caso di successo o lettura parziale

➔ anche qui, è bene controllare sempre se l'operazione è andata come volevamo

*La libreria **STDIO**: file temporanei*

- ➔ `char * tmpnam(char * ptr, ...)`
 - genera un nome file adatto per essere usato come temporaneo, lo mette in `ptr`
 - se `ptr` è `NULL`, restituisce un puntatore dove possiamo leggere questo nome

- ➔ `FILE * tmpfile(void)`
 - genera file adatto per essere usato come temporaneo, cancellato in automatico quando si fa `close`

- ➔ `int mkstemp(char *template)`
 - crea un file univoco da un template: vedere man