Teoria UNIX

Sistemi Operativi

cdl Ingegneria delle Tecnologie Informatiche
Università degli studi di Parma

4 settembre 2023

Matteo Franchini

Indice

1 Programmazione di sistema UNIX

1.1 Argomenti di programma

Un programma può accedere agli eventuali argomenti di invocazione attraverso i parametri della funzione principale **main**

```
main (int argc, char *argv[]) {
        int i;
        printf("Numero di argomenti = %d\n", argc);
        for (i = 0; i < argc; i++) {
                printf("Argomento %d (argv[%d]) = %s\n", i, i, argv[i]);
        }
}
```

si noti che `%d` si usa per indicare che in quel punto ci va un **intero**, mentre `%s` si usa per indicare che ci va una **stringa**

1.2 Compilazione

```
gcc -o mioprogramma mioprogramma.c
```

1.3 Eseguendo il programma

```
./mioprogramma 1 pippo pluto 4
```

1.4 Variabile di ambiente

```
main(int argc, char *argv[], char **envp) {
        int i;
        prinft("Numero di argomenti (argc) = %d\n", argc);
        for (i = 0; i < argc; i++) {
                printf("Argomento %d (argv[%d]) = %s\n", i, i, argv[i]);
        }
        while (*envp != NULL) { printf("%s\n", envp++); }
}
```

1.5 Perror e Streerror

`perror` e `strerror` permettono di visualizzare o di generare messaggi descritti dell'errore

```
if (syscall_N (..., ...) < 0)
{
        perror("Errore nella syscall_N");
        /*
        la descrizione dell'errore viene concatenata
        alla stringa argomento
        */
        exit(1); // terminazione del processo con errore
}
```

2 Operazioni sui file

2.1 Apertura file

Apertura ed eventuale creazione di un file

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *pathname, int flags);
```

oppure

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>

fd = int open (const char *pathname, int flags, mode_t mode);
```

- `pathname` è il nome del percorso
- `flags` contiene il modo di accesso richiesto: uno tra `O_RDONLY`, `O_WRONLY`, `O_RDWR` più altre eventuali OR
  Esempio:

  ```
  O_WRONLY|O_CREAT|O_TRUNC
  ```

  per richiedere la creazione di un nuovo file o per azzerarlo se già esiste
- `mode` indica i **diritti di accesso**

Se l'**invocazione della primitiva** `open` ha successo, viene restituito al processo un valore intero ≥ 0 che costituisce il **file descriptor (fd)** per quel file

2.2 Duplicazione file descriptor

```
#include <unistd.h>

int dup (int oldfd);
int dup2(int oldfd, int newfd);
```

2.3 Chiusura di un file descriptor

```
#include <unistd.h>

int close (int fd);
```

2.4 Lettura e scrittura di un file descriptor

```
#include <unistd.h>

int read (int fd, void *buf, size_t count);
int write (int fd, void *buf, size_t count);
```

- `read` prova a leggere dall'oggetto a cui si riferisce `fd` fino a `count` byte, memorizzandoli a partire dalla locazione `buf`
- `write` prova a scrivere sull'oggetto a cui si riferisce `fd` fino a `count` byte, letti a partire dalla locazione `buf`|

2.5 Esempio di lettura/scrittura

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFSIZ 4096

main () {
        char *f1 = "filesorg";
        char *f2 = "/tmp/filedest";
        char buffer[BUFSIZ];
        int infile, outfile; // file descriptor
        int nread;

        // apertura file sorgente

        if ((infile = open(f1, O_RONLY)) < 0) {
                perror("Apertura f1");
                exit(1);
        }

        /* open mi permette di aprire il file
         * quando scriviamo infile = ... stiamo assegnando
         * il valore restituito dalla chiamata open a infile
         * in modo da poter accedere al file utilizzando
         * direttamente infile, invece che il file descriptor
         * metto la condizione < 0 in quanto se abbiamo un
         * errore nell'apertura il fd < 0
         */

         // creazione file destinazione
         if ((outfile = open(f2, WRONLY|O_CREAT|O_TRUNC, 0644)) < 0) {
```

```
                perror("Creazione f2");
                exit(2);
        }

        /* ho messo O_WRONLY or O_CREAT or O_TRUNC
         * in quanto se esiste il file ci scrivo sopra,
         * altrimenti lo creo
         * oppure lo sovrascrivo
         */

        // ciclo di lettura/scrittura

        while ((nread = read(infile, buffer, BUFSIZ)) > 0) {
                if (write (outfile, buffer, nread) != read) {
                        perror("Errore write");
                        exit(3);
                }
                if (nread < 0) {
                        perror("Errore read");
                        exit(4);
                }
        }
        close(infile);
        close(outfile);
        exit(0);
}
```

2.6 Trasferire dati tra descrittori

```
#include <sys/sendfile.h>

ssize_t sendfile (int out_fd, int in_fd, off_t *offset, size_t count);
```

`sendfile` copia dati da un file descriptor all'altro **rimanendo all'interno del kernel**, quindi è più efficiente dell'uso combinato di read e write che trasferiscono dati tra spazio utente e kernel

- Se `offset` non è NULL, indica l'indirizzo di una variabile contenente lo spiazzamento da cui iniziare la lettura da `in_fd` che sarà modificata all'offset successivo all'ultimo byte letto; `count` è il numero di byte da copia

- Se `offset` non è NULL, allora `sendfile()` non modifica il file offset di `in_fd` altrimenti esso viene aggiustato per riflettere il numero di byte letti da `in_fd`

2.7 Copia di file con sendfile

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[]) {
        int read_fd, write_fd;
        struct stat stat_buf;
        off_t offset = 0;

        // apertura file input

        read_fd = open(argv[1], O_RDONLY);

        /* con fstat otteniamo le informazioni
         * del file che viene aperto e le salviamo
         * all'interno di stat_buf
         * in particolare in questo caso lo facciamo
         * per ottenere la dimensione del file
         */

        fstat(read_fd, &stat_buf);

        /* apriamo il file di lettura con gli stessi
         * permessi del file sorgente "stat_buf.st_mode"
         */

         write_fd = open (argv[2], O_WRONLY | O_CREAT, stat_buf.st_mode);

         sendfile (write_fd, read_fd, &offset, stat_buf.st_size);
```

```
    close (read_fd);
    close (write_fd);

    return 0;
}
```

2.8 Informazioni su file (ordinari, speciali, direttori)

```
#include <sys/stat.h>
#include <unistd.h>

int stat (const char *filename, struct stat *buf);
int fstat (int fd, struct stat *buf);
```

```
struct stat
   {
   dev_t         st_dev;     /* device */
   ino_t         st_ino;     /* inode */
   mode_t        st_mode;    /* protection */
   nlink_t       st_nlink;   /* number of hard links */
   uid_t         st_uid;     /* user ID of owner */
   gid_t         st_gid;     /* group ID of owner */
   dev_t         st_rdev;    /* device type (if inode device) */
   off_t         st_size;    /* total size, in bytes */
   unsigned long st_blksize; /* blocksize for filesystem I/O */
   unsigned long st_blocks;  /* number of blocks allocated */
   time_t        st_atime;   /* time of last access */
   time_t        st_mtime;   /* time of last modification */
   time_t        st_ctime;   /* time of last status change */
   };
```

Figura 1: Struttura buf

2.9 Cancellazione di file

```
#include <unistd.h>

int unlink (const char *filename);
```

Il file viene cancellato solo se: si tratta dell'ultimo link al file, non vi sono altri processi che lo hanno aperto

3 Primitive per la gestione degli accetti

3.1 Creazione di un nuovo processo

```
#include <unistd.h>

int fork (void);
```

Viene creato un nuovo processo (figlio) identico al processo padre che ha invocato `fork()`.
Solo il valore di uscita della `fork` è diverso per i due processi

```
pid = fork();
```

per il padre `pid` vale il pid del figlio, per il figlio `pid = 0`

3.2 Sistema di generazione

```
#include <unistd.h>

if (fork() == 0) {
        // codice del FIGLIO
}
else {
        // codice del PADRE
}
```

il padre può decidere se continuare la propria esecuzione concorrentemente a quella del figlio, oppure attendere che il figlio termini (**primitiva `wait`**)

3.3 Identificazione dei processi

```
#include <unistd.h>

pid_t getpid (void);
pid_t getppid (void);
```

La `getpid` ritorna al processo chiamante il suo PID, mentre `getppid` ritorna al processo chiamante l'identificatore di processo di suo padre (PID del PADRE)

3.4 Sincronizzazione tra padre e figlio

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

pid_t wait (int *status);
```

Il processo chiamante rimane bloccato in attesa della terminazione di uno tra i suoi figli.
Se la `wait` ha successo il valore di ritorno è il PID del processo figlio che è terminato

3.5 Uso della wait

```
int status;

if (fork () == 0) {
        // Codice del figlio
}
else {
        // Codice del padre
        ...
        // Attende che il figlio termini
        wait(&status);
}
```

Nel caso di più figli in esecuzione può essere necessario attenere **la terminazione di uno specifico figlio**

```
while (pid = wait(&status) != pidfiglio);
```

oppure direttamente

```
waitpid(pidfiglio, &status, NULL)
```

3.6 Terminazione volontaria di un processo

```
#include <stdlib.h>
void exit(int status);

#include <unistd.h>
void _exit(int status)
```

Un processo **termina volontariamente** invocando la primitiva `_exit` oppure la funzione `exit` della libreria standard I/O di C

3.7 Esecuzione di un programma

```
#include <unistd.h>

int execve (const char *pathname, char *const argv[], char *const envp[]);
```

Il processo chiamante passa ad eseguire il programma `filename` La fork **crea un nuovo processo identico al padre**, la exec permette **di modificare l'ambiente di esecuzione di un processo**

3.8 Esempio di uso della execve

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

int main () {
        int status;
        pid_t pid;
        char *env[] = {
                "TERM=vt100",
                "PATH=/bin:/usr/bin",
                (char *) 0
        };

        char *args[] = {
                "cat",
                "f1",
                "f2",
                (char *) 0
        };

        if ((pid=fork()) == 0) {
                // codice del figlio
                execve("bin/cat", args, env);

                /* si torna a questo punto solo
                 * nel caso in cui si verifichi un
                 * errore
                 */
                perror("excve")
                exit(1);
```

```
        }
        else {
                // codice del padre
                wait(&status);
                printf("Il processo %d e' terminato con %d\n",
                pid, WEXITSTATUS(status));
        }
        exit(0);
}
```

4 Primitive per la gestione dei segnali

Quali sono le possibilità di gestione di un segnale per un processo?

1. può decidere di **ignorarlo**
2. può contare su un'azione di **default**
3. può far eseguire un'**azione** specifica dall'utente

4.1 Elenco dei segnali Linux

```
#define   SIGHUP      1       /* Hangup (POSIX).  */
#define   SIGINT      2       /* Interrupt (ANSI).  */
#define   SIGQUIT     3       /* Quit (POSIX).  */
#define   SIGILL      4       /* Illegal instruction (ANSI).  */
#define   SIGTRAP     5       /* Trace trap (POSIX).  */
#define   SIGABRT     6       /* Abort (ANSI).  */
#define   SIGIOT      6       /* IOT trap (4.2 BSD).  */
#define   SIGBUS      7       /* BUS error (4.2 BSD).  */
#define   SIGFPE      8       /* Floating-point exception (ANSI).  */
#define   SIGKILL     9       /* Kill, unblockable (POSIX).  */
#define   SIGUSR1     10      /* User-defined signal 1 (POSIX).  */
#define   SIGSEGV     11      /* Segmentation violation (ANSI).  */
#define   SIGUSR2     12      /* User-defined signal 2 (POSIX).  */
#define   SIGPIPE     13      /* Broken pipe (POSIX).  */
#define   SIGALRM     14      /* Alarm clock (POSIX).  */
#define   SIGTERM     15      /* Termination (ANSI).  */
#define   SIGSTKFLT   16      /* Stack fault.  */
#define   SIGCLD      SIGCHLD         /* Same as SIGCHLD (System V).  */
```

Figura 2: Elenco segnali - prima parte

```
#define   SIGCHLD     17      /* Child status has changed (POSIX).  */
#define   SIGCONT     18      /* Continue (POSIX).  */
#define   SIGSTOP     19      /* Stop, unblockable (POSIX).  */
#define   SIGTSTP     20      /* Keyboard stop (POSIX).  */
#define   SIGTTIN     21      /* Background read from tty (POSIX).  */
#define   SIGTTOU     22      /* Background write to tty (POSIX).  */
#define   SIGURG      23      /* Urgent condition on socket (4.2 BSD).  */
#define   SIGXCPU     24      /* CPU limit exceeded (4.2 BSD).  */
#define   SIGXFSZ     25      /* File size limit exceeded (4.2 BSD).  */
#define   SIGVTALRM   26      /* Virtual alarm clock (4.2 BSD).  */
#define   SIGPROF     27      /* Profiling alarm clock (4.2 BSD).  */
#define   SIGWINCH    28      /* Window size change (4.3 BSD, Sun).  */
#define   SIGPOLL     SIGIO   /* Pollable event occurred (System V).  */
#define   SIGIO       29      /* I/O now possible (4.2 BSD).  */
#define   SIGPWR      30      /* Power failure restart (System V).  */
#define   SIGSYS      31      /* Bad system call.  */
#define   SIGUNUSED   31
```

Figura 3: Elenco segnali - seconda parte

4.2 Interfaccia signal

```
#include <signal.h>

void (*signal(int signo, void (*func) (int))) (int);
```

si specifica quale segnale (`signo`) e come deve essere trattato `func`

```
void catchint(int);

main () {
int i;

/* la notifica di un segnale SIGINT
 * deve avviare il gestore catchint:
 * si dice comunemente che il
 * processo "intercetta" o "aggancia"
 * il segnale
 */

 signal(SIGINT, catchint);

 while(1) {
        for (i = 0; i < 100; i++) {
                printf("i vale %d\n", i);
        }
        sleep(1);
 }
}

void catchint (int signo) {
        printf("catchint: signo=%d\n", signo);
}
```

4.3 Kill

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Invia il segnale `sig` al processo `pid`

- il processo mittente e quello destinatario del segnale devono appartenere allo stesso utente
- solo **root** può inviare segnali a processi di altri utenti
- `kill (0, sig)` invia il segnale sig a tutti i processi del gruppo del processo chiamante (padre, figli, nipoti, ecc)

4.4 Invio temporizzato di segnali

```
#include <unistd.h>

unsigned int alarm (unsigned int nseconds);
```

dopo *nseconds* secondi il processo chiamante riceve un segnale SIGALARM inviato dal SO

4.5 Attesa di un segnale

```
#include <unistd.h>

int pause(void);
```

4.6 Gestione affidabile dei segnali

```
#include <signal.h>

// azzera/rende vuoto un sigset

int sigemptyset (sigset_t *set);

// mette tutti i segnali nel sigset

int sigfillset (sigset_t *set);

// aggiunge un segnale al siget

int sigaddset (sigset_t *set, int signo);

// rimuove un segnale dal siget

int sigdelset (sigset_t *set, int signo);
```

```
// valuta se quel segnale è presente nel sigset

int sigismember (sigset_t *set, int signo);
```

4.7 Signal mask

Un processo può esaminare e/o modificare la propria *signal mask* che è **l'insieme dei segnali che sta attualmente bloccando**

```
int sigprocmask (int how, const sigset_t *set, sigset_t *oset);
```

dove *how* può valere:

- `SIG_BLOCK`: la nuova signal mask diventa l'OR binario di quella corrente con quella specificata dal *set*
- `SIG_UNBLOCK`: i segnali indicati da set sono rimossi dalla signal mask
- `SIG_SETMASK`: la nuova signal mask diventa quella specifica da `set`

```
int sigpending (sigset_t *set);
```

restituisce in *set* il sottoinsieme dei segnali bloccati che sono attualmente pendenti ovvero inviati ma non ancora notificati perché bloccati.
La `sigaction` è la primitiva fondamentale per la gestione dei segnali affidabili

```
#include <signal.h>

int sigaction (int signo, const struct sigaction *act,
const struct sigaction *oact);
```

permette di esaminare e/o modificare l'azione associata ad un segnale

- `signo` identifica il segnale del quale si vuole esaminare e/o modificare l'azione
- `act` non è NULL si modifica l'azione
- `oact` non è NULL viene restituia l'azione precedente

Definizione della struttura sigaction

```
struct sigaction {
// indirizzo del gestore o SIG_IGN o SIG_DFL
void (*sa_handler) ();

/*
 * indirizzo del gestore che riceve informazioni addizionali
 * sul segnale ricevuto
 */
void (*sa_sigaction)(int, siginfo_t *, void *);

// segnali addizionali da bloccare prima dell'esecuzione del gestore
sigset_t sa_mask;

// opzioni addizionali
int sa_flags;
}
```

Attesa di un segnale con la gestione affidabile

```
int sigsuspend(const sigset_t *sigmask)
```

permette l'attivazione della signal mask specificata e l'attesa di un qualunque segnale in modo atomico

4.8 Esempio di interazione tra processi mediante segnali affidabili

In questo esempio vediamo due processi, un padre e un figlio, che comunicano attraverso un segnale SIGUSR1

```
#include <signal.h>
#include <unistd.h>

/*
 * Questa funzione viene chiamata quando
 * il processo riceve il segnale SIGUSR1
 * e conta quante volte viene ricevuto il
 * segnale
 */

void catcher(int signo)
{
        static int ntimes = 0;
        printf("Processo %d: SIGUSR1 ricevuto #%d volte\n",
        getpid(), ++ntimes);
}

int main ()
{
        int pid, ppid;

        /*
         * inizializzazione della struccura
         * sig per impostare la gestione
         * del segnale
         */
        struct sigaction sig, osig;

        /*
         * creazione di alcune maschere dei segnali
         * in particolare "sigmask" viene utilizzata
         * per contenere un insieme di segnali
         * "oldmask" per salvare l'insieme di segnali
         * precedente e "zeromask" e' un insieme
         * vuoto di segnali
```

```
 */
sigset_t sigmask, oldmask, zeromask;

/*
 * assegnazione della funzione catcher
 * come gestore del segnale SIGSR1.
 * Quindi quando arriva il segnale, viene
 * chiamata la funzione "catcher"
 */
sig.sa_handler = catcher;

/*
 * questo vuol dire che nessun
 * altro segnale deve essere bloccato
 * durante l'esecuzione di catcher
 */
sigemptyset (&sig.sa_mask);

// non ci sono flag speciali
sig.sa_flags = 0;

sigemptyset (&zeromask);
sigemptyset (&sigmask);

/*
 * aggiungo all'insieme sigmask
 * il segnale SIGUSR1
 */
sigaddset(&sigmask, SIGUSR1);

/*
 * questa funzione blocca temporaneamente
 * il segnale SIGUSR1, quindi viene messo
 * in attesa fino a che non e' bloccato
 * esplicitamente
 */
sigprocmask(SIG_BLOCK, &sigmask, &oldmask);

 /*
  * utilizziamo sigaction per impostare la gestione
  * del segnale SIGUSR1 con la configurazione
```

```
  * definita sig e salviamo la precedente configurazione
  * in osig
  */
sigaction(SIGUSR1, &sig, &osig)


/*
 * IN SINTESI: quello che succede in questo
 * blocco e' che stiamo configurando la gestione
 * del segnale in modo da chiamare la funzione "catcher"
 * e quando il segnale viene ricevuto temporaneamente
 * blocca il segnale per evitare interruzioni durante l'esecuzione
 * della funzione "catcher"
 */


if ((pid=fork()) < 0) {
        perror("fork error");
        exit(1);
}
else
        if (pid == 0) {
                // figlio
                ppid = getppid();
                printf("figlio: mio padre e' %d\n", ppid);
                while(1) {
                        sleep(1);
                        kill(ppid, SIGUSR1);
                        // sblocca il segnale SIGUSR1 e lo attende

                        sigsuspend(&zeromask);
                }
        }
        else {
                // padre

                printf("padre: mio figlio e' %d\n", pid);
                while(1) {
                        // sblocca il segnale SIGUR1 e lo attende
                        sigsuspend(&zeromask);
                        sleep(1);
```

```
                kill(pid, SIGUSR1);
            }
        }
}
```

5 Primitive per la gestione della comunicazione via pipe e FIFO

5.1 Pipe

```
int pipe(int fd[2]);
```

le pipe sono **canali di comunicazione** unidirezionali che costituiscono un primo strumento di comunicazione, basato sullo **scambio di messaggi**, tra processi UNIX
La creazione di una *pipe* mediante la primitiva omonima restituisce in *fd* due descrittori: `fd[0]` per la lettura, `fd[1]` per la scrittura

5.2 Esempio di comunicazione su pipe

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define N_MESSAGGI 10

int main ()
{
        int pid, j, k, piped[2];

        /*
         * apre le pipe creando due file descriptor,
         * uno per la lettura e l'altro per la
         * scrittura
         */

        if (pipe(piped) < 0) { exit(1); }

        if ((pid = fork()) < 0 ) { exit(2); }
```

```
        else if (pid == 0) {
                // il figlio eredita una copia di piped[]

                /*
                 * il figlio e' il lettore della pipe
                 * e quindi piped[1] non gli serve
                 */

                close(piped[1]);

                for (j = 1; j <= N_MESSAGGI; j++) {
                        read(piped[0], &k, sizeof (int));
                        printf("Figlio: ho letto dalla pipe
                        il numero %d\n", k);
                }
                exit(0);
        }
        else {
                // Processo padre

                /*
                 * il padre e' lo scritto e quindi piped[0]
                 * non gli serve
                 */
                close(piped[0]);
                for (j = 1; j <= N_MESSAGGI; j++) {
                        write(piped[1], &j, sizeof (int));
                }
                wait(NULL);
                exit(0);
        }
}
```

5.3 FIFO

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo (const char *pathname, mode_t mode);
```

Le FIFO sono adatte per applicazione client-server in locale

- il server apre una FIFO con un nome noto (ad es. `/tmp/server`)
- i client aprono la FIFO e scrivono le proprie richieste

6 La suite di protocolli di rete TPC/IP

6.1 Struttura a livelli

Per ridurre la complessità progettuale, tutte le reti sono progettate a livelli. Il numero di livelli, i loro nomi, il contenuto di ciascun livello differisce da rete a rete.
I livelli più alti sono vicini all'uomo, quelli più bassi all'hardware.

Figura 4: Livelli nel modello ISO/OSI

6.1.1 Livello 1: Fisico

Si occupa di trasmettere sequenze binarie sul canale di comunicazione.
A questo livello si specificano:

- tensioni dello 0 e dell'1
- tipi, dimensioni, impedenze dei cavi
- tipi di connettori

6.1.2 Livello 2: Data Link

Ha lo scopo di trasmmissione affidabile di pacchetti di dati (frame), accetta come input i frame e li trasmette sequenzialmente.
Verifica la presenza di errori aggiungendo delle FCS

6.1.3 Livello 3: Rete

Questo livello gestisce l'instradamento dei messaggi e determina quali sistemi intermedi devono essere attraversati da un messaggio per giungere a destinazione

6.1.4 Livello 4: Trasporto

Fornisce servizi per il trasferimento dei dati *end-to-end*.
In particolare il livello 4 può:

- frammentare i pacchetti in modo che abbiano dimensioni idonee al livello 3
- rilevare/correggere gli errori
- controllare il flusso
- controllare le congestioni

6.1.5 Livello 5: Sessione

Il livello 5 è responsabile dell'organizzazione dei dialogo e della sincronizzazione tra due programmi applicativi e del conseguente scambio di dati

6.1.6 Livello 6: Presentazione

Il livello di presentazione gestisce la sintassi dell'informazione da trasferire. L'informazione è infatti rappresentata in modi diversi su elaboratori diversi.

6.1.7 Livello 7: Applicazione

È il livello dei programmi applicativi, cioè di quei programmi appartenenti al SO o scritti dagli utenti, attraverso i quali l'utente finale utilizza la rete

7 Internet Protocol (IP)

Il protocollo IP esegue le seguenti principali funzioni:

- Indirizzamento
- Instradamento
- Controllo dell'errore sull'intestazione IP
- Se necessario esegue frammentazione e ri-assemblaggio dei pacchetti

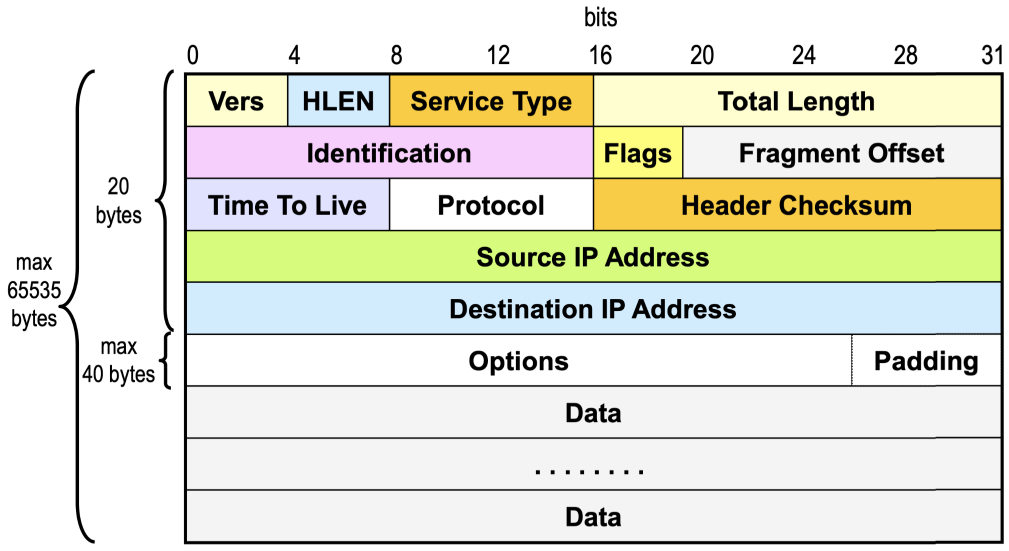

Figura 5: Header IPv4

IP consente ad ogni nodo (IP) connesso alla rete di comunicare con ogni altro nodo (IP), a tal fine utilizza un metodo globale di identificazione e

indirizzamento di tutti i nodi (host e router) connessi alla stessa rete IP. IPv4 utilizza indirizzi di 32 bit.
Un indirizzo IP è formato da: $IP\_ADRESS = network\_prefix + host\_id$

7.1 Configurazione di un nodo Linux a riga di comando

- `ifconfig`: mostra/configura le interfacce di rete e indirizzi
- `route`: mostra/manipola la tabella di istradamento IP
- `ip`: tramite appositi sottocomandi permette di mostrare, modificare interfacce, indirizzi e tabelle di istradamento
- `arp`: manipola la cache ARP
- `nslookup`: effettua le interrogazioni al DNS
- `netstat`: mostra connessioni di rete, tabelle di routing, statistiche

8 Protocollo ICMP

ICMP (Internet Control Message Protocol) è utilizzato per la trasmissione dei messaggi di errore e di controllo relativi al protocollo IP, i messaggi vengono manipolari dal software IP, non dagli applicativi utente.
ICMP può quindi essere considerato un sub-strato di IP ma è funzionalmente al di sopra di IP.
ICMP è una parte integrante di IP e deve essere incluso in ogni implementazione IP, un messaggio ICMP è incapsulato nella parte dati di un datagramma IP

9 Address Resolution Protocol (ARP)

Ogni quale volta si debba rilanciare un pacchetto da un nodo ad un altro bisogna conoscere l'indirizzo si sottorete del noto next hop identificato dal suo indirizzo IP, è necessaria quindi una funzione di mappaggio da indirizzi IP ad indirizzi di sottorete.
Il protocollo ARP fornisce un meccanismo dinamico di associazione tra indirizzi MAC ed indirizzi IP.
Viene utilizzato ogni qual volta un nodo di una LAN debba inviare un pacchetto ad un altro nodo della stessa LAN di cui però conosca solo l'indirizzo IP

10 Strato di trasporto

L'obbiettivo è fornisce una comunicazione *end-to-end* ai processi applicativi. Per l'architettura internet sono stati definiti due protocolli di trasporto

- User Datagram Protocol
- Transmission Control Protocol

11 User Datagram Protocol (UDP)

Consente alle applicazioni di scambiare messaggi singoli e fornisce un livello di servizio minimo:

- È un protocollo senza connessione
- Non supporta meccanismi di riscontro e recupero d'errore

12 Transmission Control Protocol (TPC)

È un protocollo *end-to-end* con connessione e offre un servizio stream-oriented affidabile.
Trasferisce un flusso informativo bi-direzionale non strutturato tra due host ed effettua operazioni di multiplazione e de-multiplazione.
Non prevede nodi intermedi (TPC) e quindi non implementa la funzione di commutazione.

13 Network Adress Translation (NAT)

Un nodo NAT si usa normalmente quando si vuole interconnettere una rete IP con indirizzamento privato (intranet) alla rete pubblica (intenet). Il nodo che effettua l'interconnessione si chiama router NAT.
La rete interna vede questo nodo come router per instradare i pacchetti verso nodi della rete esterna, la rete esterna vede i pacchetti provenienti dalla rete interna come inviati dal nodo NAT

14 Domain Name System (DNS)

Oltre alla notazione dotted viene spesso utilizzata anche un'altra forma di notazione (mnemonica), per esempio "160.78.48.141" è uguale a "www.unipr.it".
È necessaria la **funzionalità di traduzione di nomi mnemonici** in indirizzo e viceversa.
I nomi sono organizzati gerarchicamente in **domini**:

- in nomi sono costituiti da stringe separate da "."
- la parte più significativa è a destra

15 Primitive per la gestione della comunicazione via socket

Una **socket** è un punto estremo di un canale di comunicazione accessibile mediante un file descriptor, le socket costituiscono un fondamentale strumento di comunicazione, basato sullo scambio di messaggi, tra processi locali e/o remoti (sia UNIX che di altri sistemi operativi).
Una socket è un oggetto con un tipo, determinato dal sottoinsieme delle seguenti proprietà che quel tipo di socket garantisce

1. **Consegna ordinata dei messaggi**
2. **Consegna non duplicata**
3. **Consegna affidabile** (i messaggi non possono andare persi)
4. **Preservamento dei confini dei messaggi** (i messaggi inviati non vengono frazionati nella comunicazione)
5. **Supporto per i messaggi** *out-of-band*
6. **Comunicazione orientata alla connessione**

Le pipe (che non sono socket) garantiscono le proprietà 1, 2 e 3.
Alcuni tipi predefiniti di socket

- **SOCK_STREAM**: orientata alla connessione, trasferisce byte con proprietà 1, 2, 3, 5, 6
- **SOCK_DGRAM**: trasferisce datagram con proprietà 4 ma non altre

- **SOCK_SEQPACKET**: trasferisce datagram con proprietà 1, 2, 3, 4, 5, 6
- **SOCK_RAW**: permette l'accesso diretto ai protocolli di rete sottostanti

15.1 Creazione della socket

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Una socket viene creata nel dominio di comunicazione `domain`
Domini principali

- **PF_UNIX**: dominio per una comunicazione locale
- **PF_INET**: dominio per una comunicazione su TCP/IP (IPv4)
- **PF_INET6**: dominio per una comunicazione su TCP/IP (IPv6)

`type` indica il tipo di socket che si vuole creare(es. `SOCK_STREAM`); `protocol` indica lo specifico protocollo utilizzato tra quelli disponibili nel dominio

15.2 Assegnazione nome alla socket

```
#include <sys/types.h>
#include <sys/socket.h>

int bind (int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

`bind` assegna un nome ad una socket per renderla designabile (indirizzabile) da parte di un processo intenzionato a comunicare con il processo che creato la socket.
L'interfaccia è generica (`my_addr, addrlen`) in quanto i diversi domini di comunicazione prevedono indirizzi di forma diversa

15.3 Strutture dati utilizzate da bind in AF_INET

```
struct sockaddr_in {
   sa_family_t     sin_family; /* address family: AF_INET */
   u_int16_t       sin_port;   /* port in network byte order */
   struct in_addr  sin_addr;  /* internet address */
   };
```

Figura 6: Strutture dati utilizzate da bind in AF_INET

15.4 Chiusura di una socket

Chiusura di una socket e successiva `bind` per lo stesso indirizzo

```
close(sock);
```

Si utilizza la `setsockopt` per forzare il riuso dell'indirizzo nel bind che quindi non fallirà anche se esiste già una socket **in fase in attesa** con lo stesso indirizzo

```
int on = 1;
ret = setsockopt (sock, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on));
...
bind(...);
```

15.5 Creazione della connessione

Un **client** inizia una connessione sulla propria socket specificando l'indirizzo della socket del server

```
connect (int cli_sockfd, ...);
/* bloccante */
```

Il **server** dichiara al SO la sua disponibilità a ricevere connessioni sulla propria socket

```
listen (int serv_sockfd, ...);
/* non bloccante */
```

Il **server** attende richieste di connessioni sulla propria socket e riceve un nuovo descrittore per ogni nuova connessione

```
conn_sockfd = accept (int serv_sockfd,...);
/* bloccante */
```

È sincronizzato con il codice sopra.
Su socket connesse

```
write (cli_sockfd, ...);
read (cli_sockfd, ...);
```

e

```
read (conn_sockfd, ...);
write (conn_sockfd, ...);
```

15.6 Connect, listen e accept

Connect

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr, socket_t addrlen);
```

Listen

```
#include <sys/socket.h>

int listen (int s, int backlog);
```

`backlog` specificava la dimensione massima della coda delle richieste di connessione pendenti (non ancora accettate).

Accept

```
#include <sys/types.h>
#include <sys/socket.h>

int accept (int s, struct sockaddr *addr, socklen_t *addrlen);
```

`s` è il descrittore della socket di controllo, in `addr` (se non è NULL viene memorizzato l'indirizzo del cliente che si è connesso)

15.7 Server concorrente

Normalmente un **server su SOCK_STREAM crea un nuovo figlio dedicato a gestire una nuova connessione da un cliente** mentre il server padre continua ad attendere nuove connessioni.
In questo codice i client in coda vengono serviti al più presto da un server dedicato, ed eventuali ritardi di un client non hanno effetto sul servizio agli altri

```
do
{
        // Attesa di una connessione
        if (msgsock = accept(sock, (struct sockaddr *) &client,
                (socklen_t *) &len) < 0) {
                        perror("accept");
                        exit(-1);
        }
        else {
                if (fork() == 0) {
                        // Server figlio
                        printf("Serving connection from %s, port %d\n",
                                inet_ntoa(client.sin_addr),
                                ntohs(client.sin_port));

                                close(sock);
                                // non interessa la socket di controllo

                                myservice(msgsock);
                                /* servizio specifico del server
                                attraverso il server connesso */

                                close(msgsock);
                                /* la socket connessa
                                puo' essere rimossa */
                                exit(0);
                }
                else {
                        close(msgsock);
                        /* non interessa la socket
                        connessa: si ritorna in
                        accept */
                }
        }
}
while(1);
```

15.8 Datagram

Non vi è alcuno stato di connessione (protocollo UDP di TCP/IP)

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto (int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
```

Invio di un messaggio con designazione esplicita del destinatario

15.9 Ricezione di un messaggio

```
int recvfrom (int s, void *buf, size_t len, int flags,
struct sockaddr *from, socklen_t *fromlen);
```

L'indirizzo del mittente del messaggio viene posto in `from` (se diverso da NULL)

16 Select

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select (int n, fd_set *readfds, fd_set *writefds,
fd_set *execptfds, struct timevale *timeout)
```

La primitiva `select` permette di attendere una variazione di stato per i file descriptor all'interno di tre distinti insiemi di file descriptor

- si attende la disponibilità di dati di lettura per i fd contenenti in `readfds`
- si attende la possibilità di scrittura immediata sui fd contenuti in `writefds`
- si attende la presenza di eccezioni per i fd contenuti in `exceptfds`

Il valore di uscita è il numero di descrittori che sono stati variati di stato.
Gli `fd_set` sono **modificati** in uscita dalla `select` in modo che contengano i soli fd che hanno variato di stato
Macro utili per la manipolazione di variabili *fd_ set*

- `FD_ZERO(fd_set *set)`: azzera un fd_set
- `FD_CLR(int fd, fd_set *set)`: rimuove un fd da un fd_set
- `FD_SET(int fd, fd_set *set)`: inserisce un fd in un fd_set
- `FD_ISSET(int fd, fd_set *set)`: predicato che verifica se un certo fd è membro di un fd_set

Elenco delle figure