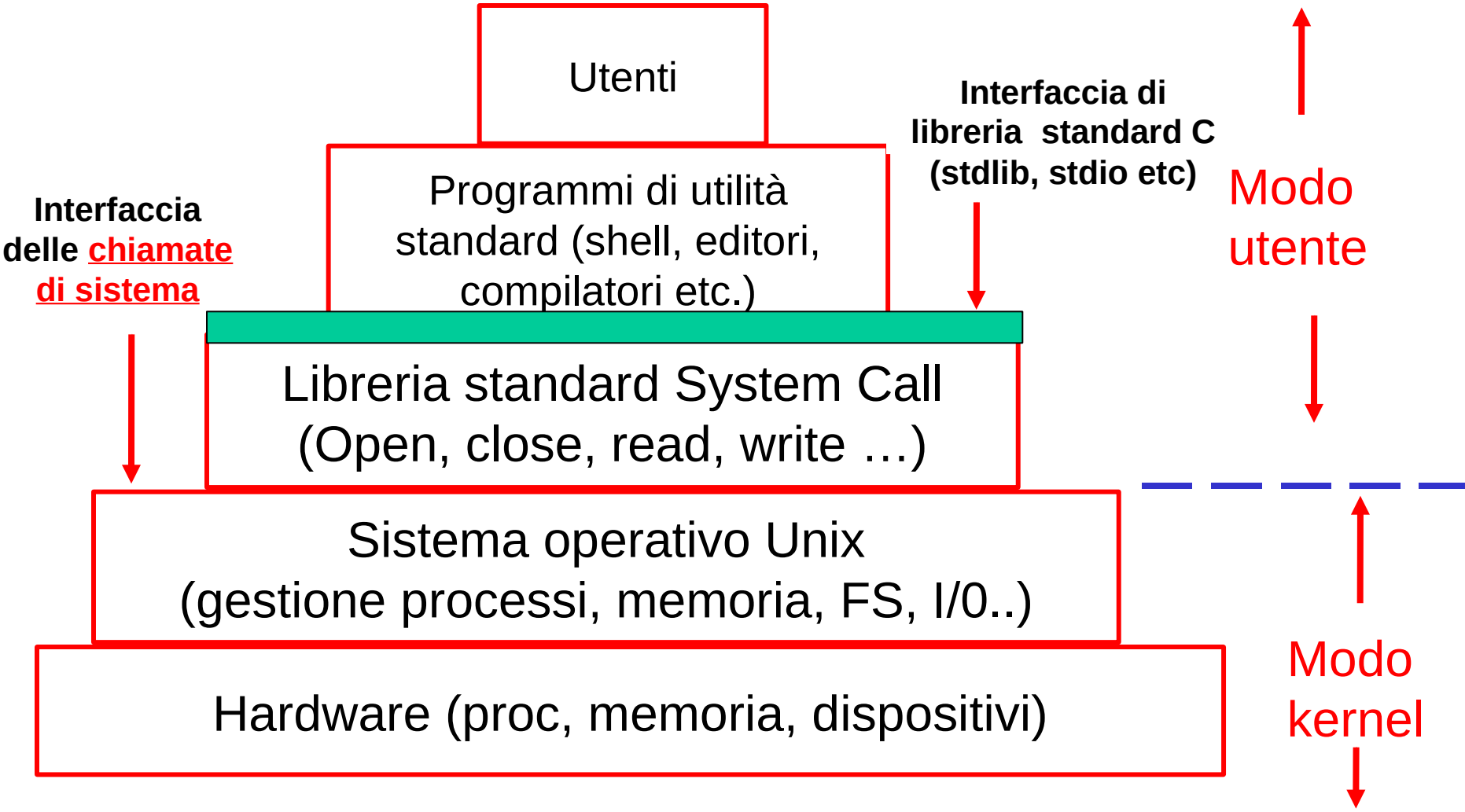
Chiamate di sistema

Introduzione

Errori : perror()

Chiamate che lavorano su file

UNIX/Linux: System calls



Come si invoca una SC da C ?

- Come una qualsiasi altra funzione di libreria:
  - es.

```
e = read(fd, buf, numbyte);
```

  - di solito è necessario includere uno o più header (vedi **man** sezione **2**, system call) es.

```
bash:~$ man 2 read
```

  - noi vedremo una piccola selezione delle call **POSIX**
  - *portabilità* : quasi tutti i sistemi aggiungono qualcosa allo standard, per la caratteristiche davvero standard
    - vedi **www.unix.org/version3/online.html** (free basta registrarsi)

Come funziona una SC

- SC e funzioni che girano in spazio utente sono MOLTO diverse
  - nella SC solo una piccola parte del lavoro viene effettuata in modo utente:
    - preparazione dei parametri etc.
  - poi il controllo viene trasferito al SO in modalità kernel con una istruzione assembler speciale (es. TRAP)
  - infine il controllo ritorna in modalità utente e si ritorna normalmente al programma chiamante
  - ci sono 2 *contex switch* (u-k-u) ogni chiamata di sistema

Come funziona una SC (2)

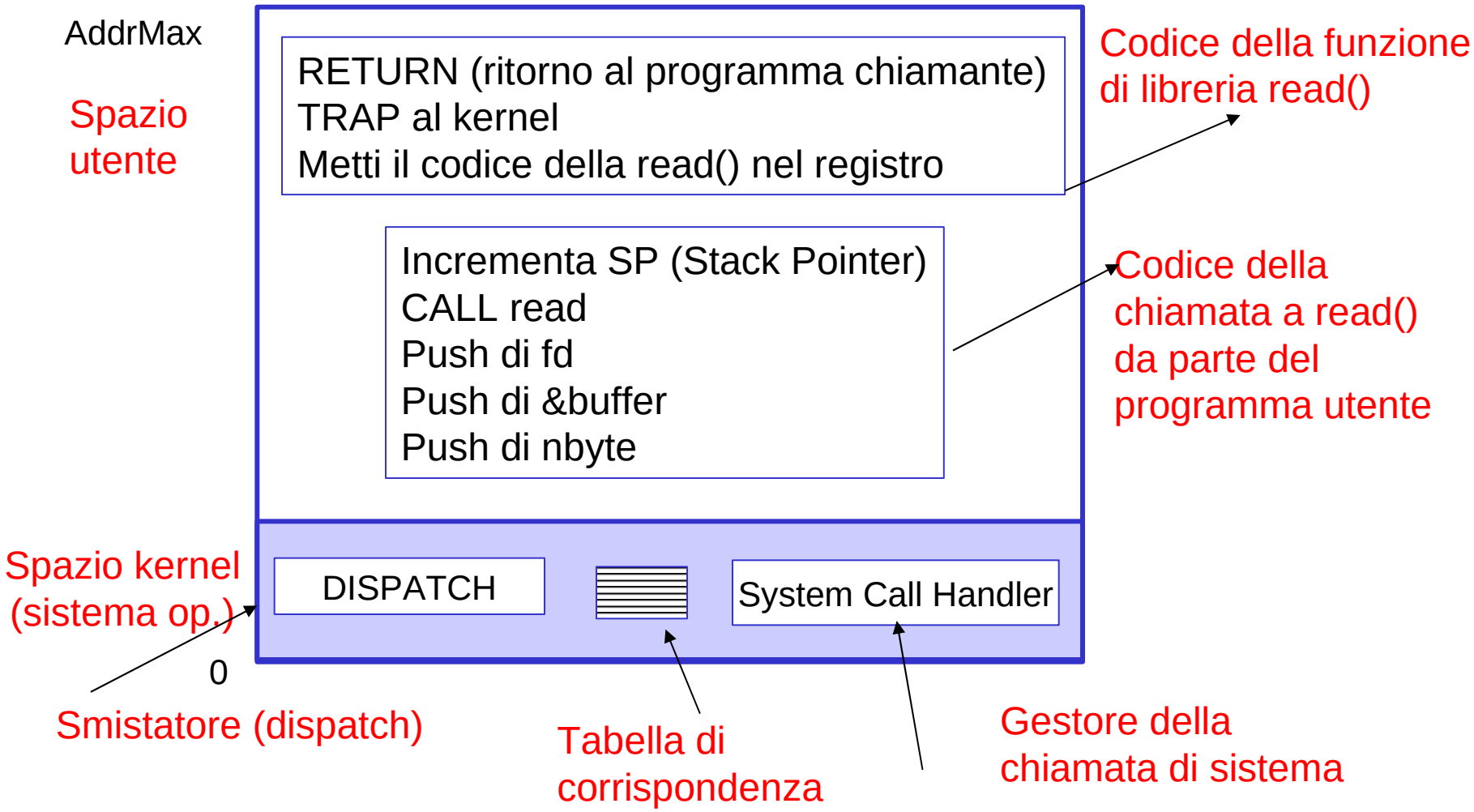
- SC e funzioni (cont) :
  - l'invocazione della TRAP non può essere generata dal compilatore C
    - quindi il codice di una SC deve contenere parti in assembler
  - la TRAP permette di
    - passare da stato utente a stato supervisore/kernel
    - saltare ad un indirizzo predefinto all'interno del sistema operativo
  - il processo rimane in esecuzione (con accesso al suo spazio di indirizzamento) solo esegue una funzione del sistema operativo in stato kernel

Un esempio: `read(...)`

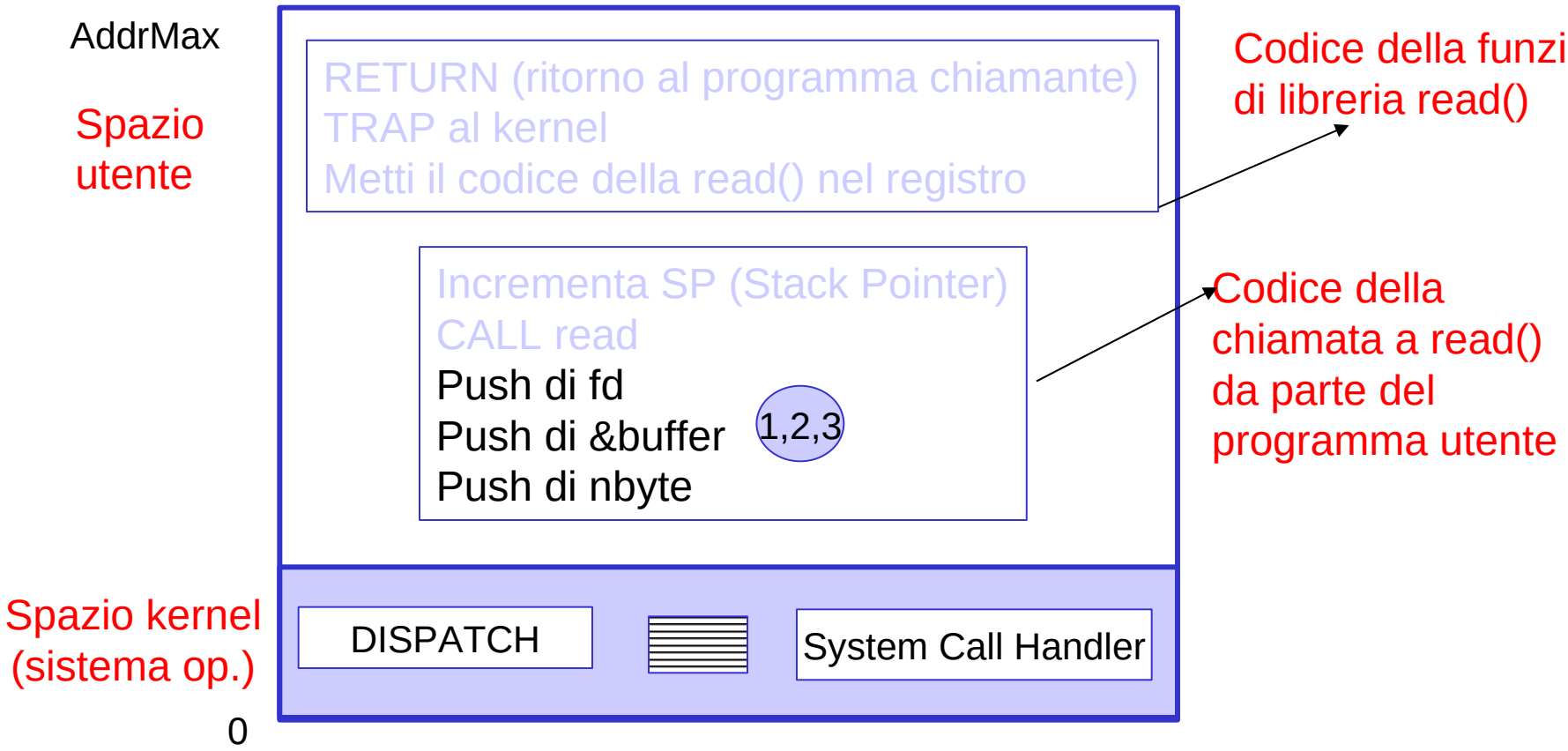
- Ma esattamente cosa accade quando viene invocata una *system call* ?
  - Vediamo in dettaglio come funziona una chiamata a

    read (fd, buffer, nbytes);

Chiamata a read(fd, buffer, nbytes)

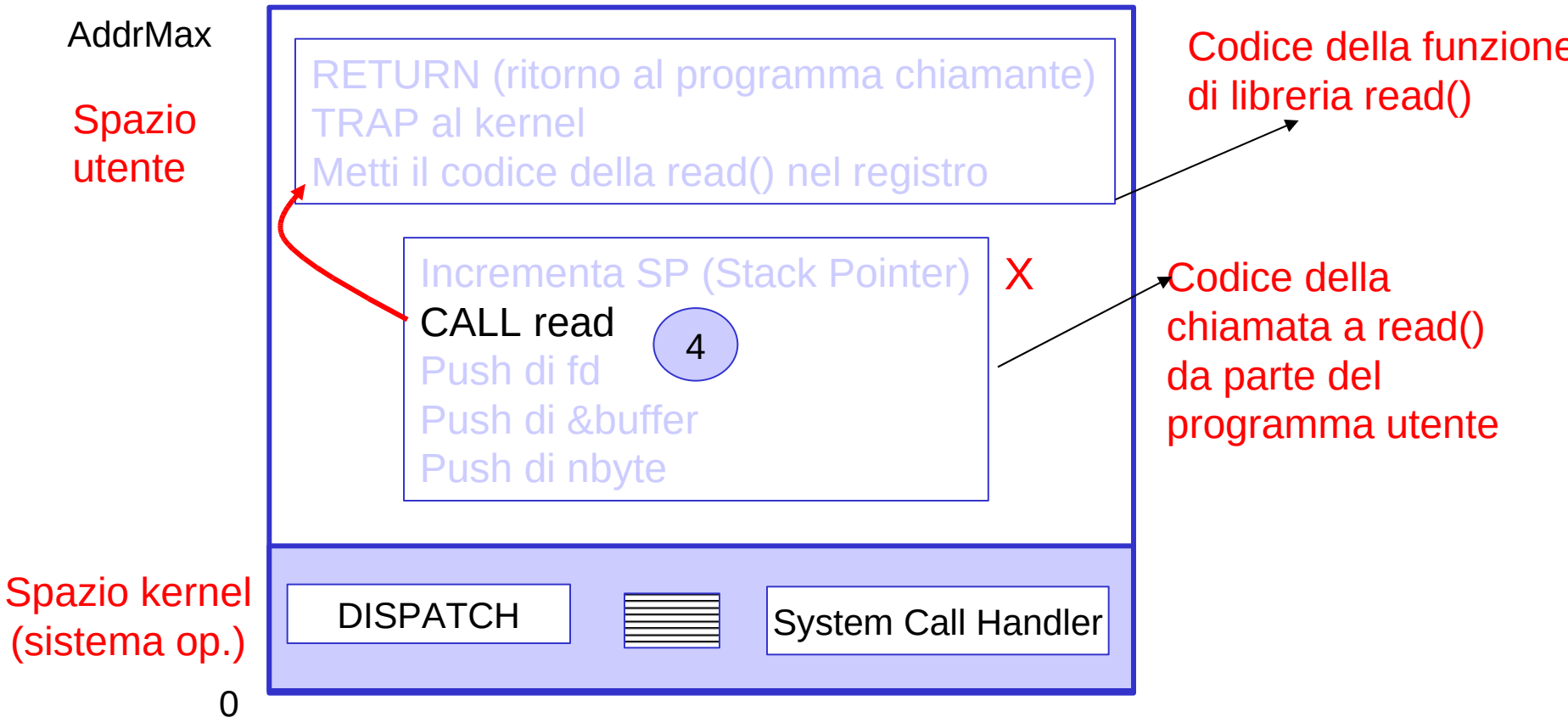


read(fd, buffer, nbytes) (2)



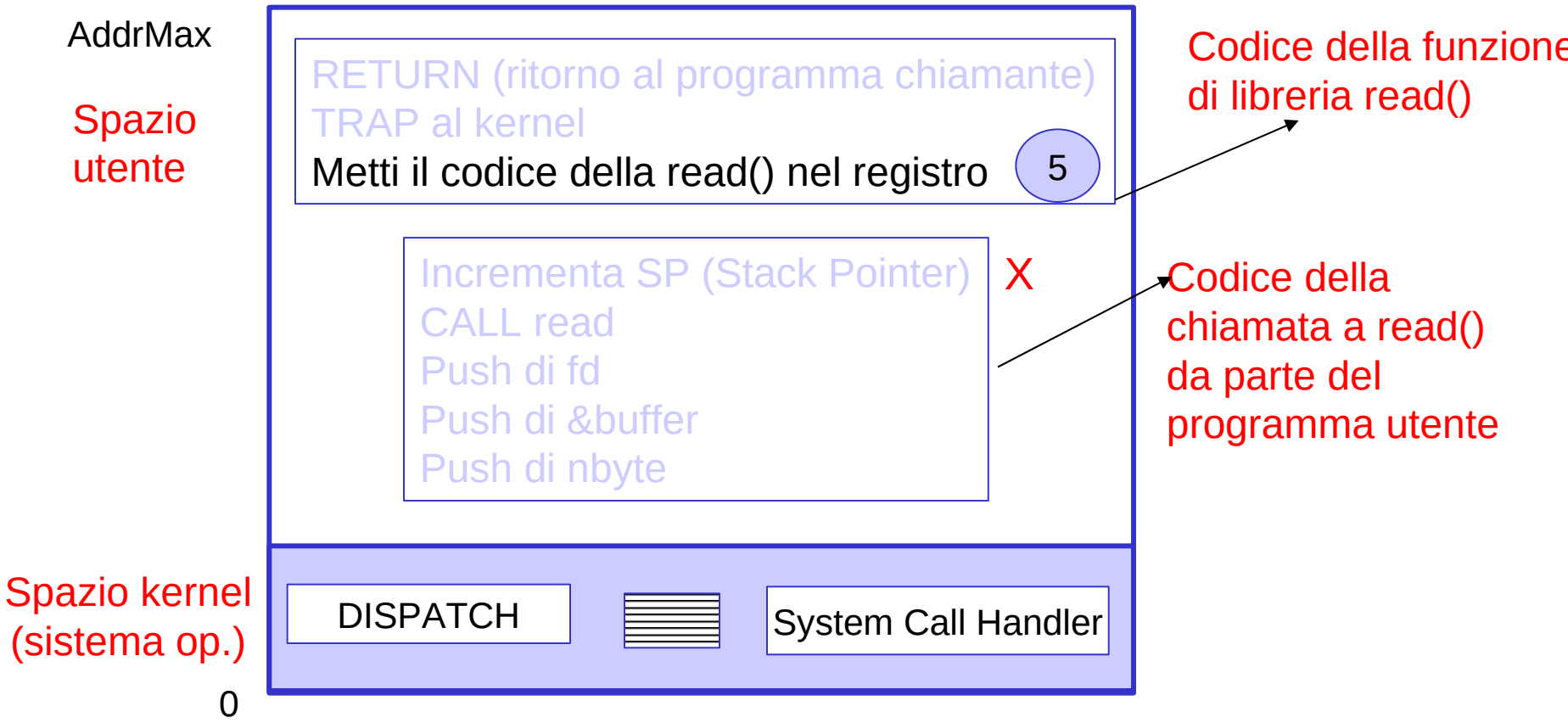
- Passi 1,2,3 :
  - si ricopia il valore dei parametri sullo stack

read(fd, buffer, nbytes) (3)



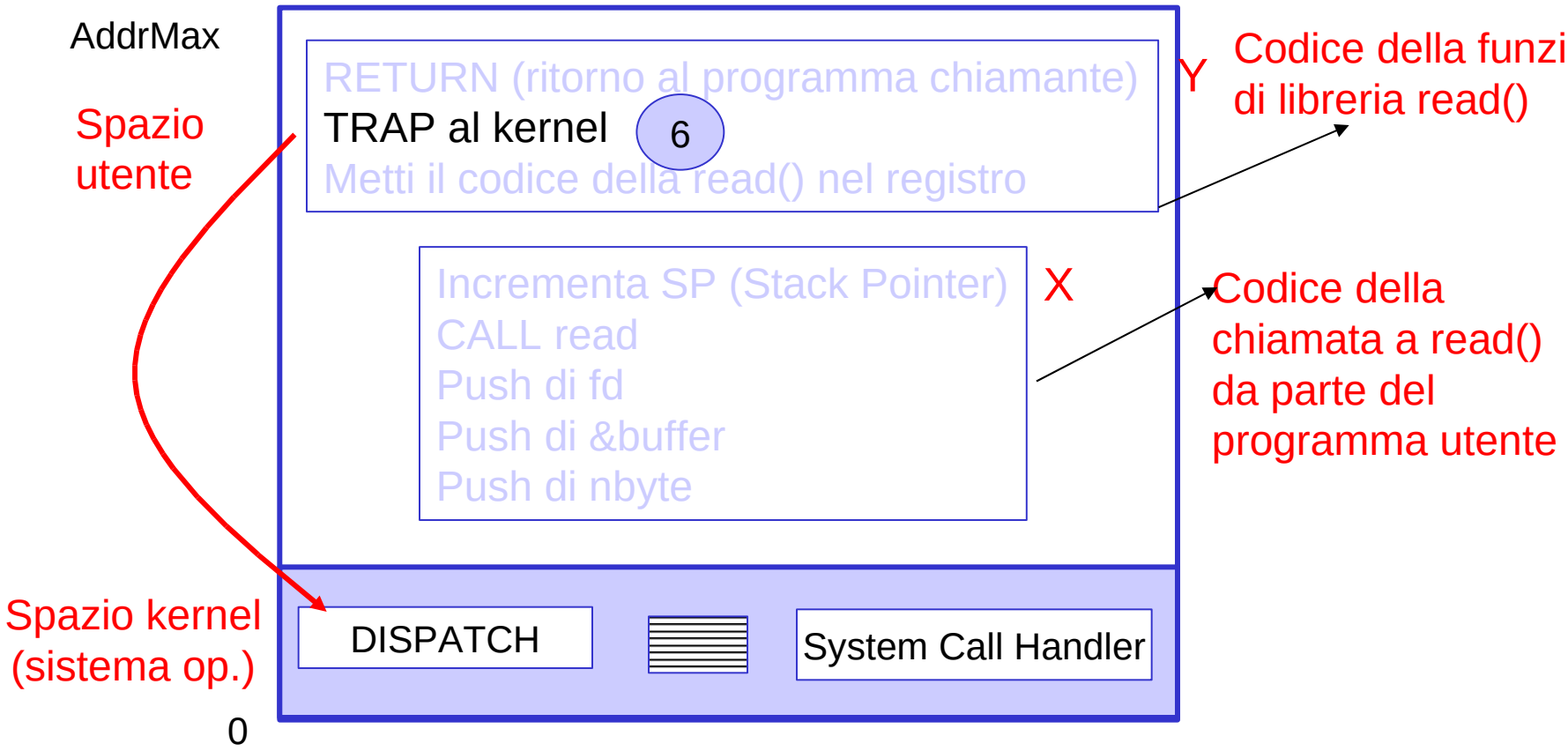
- Passo 4 : chiamata di read()
  - salto alla prima istruzione di read() + push indirizzo di ritorno (X) sullo stack

read(fd, buffer, nbytes) (4)



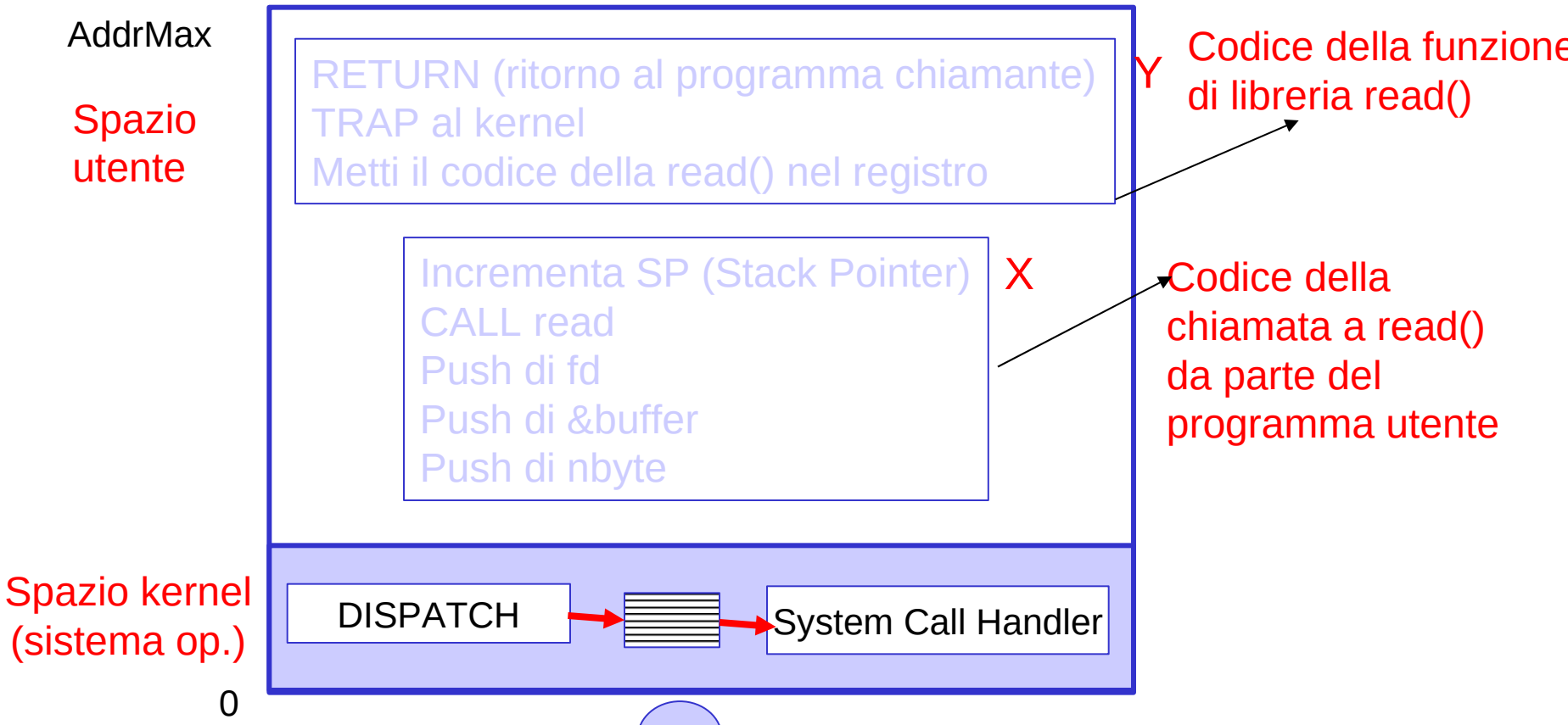
- Passo 5 : Inizia l'esecuzione della read() :
  - caricamento del codice della system call in un registro fissato Rx

read(fd, buffer, nbytes) (5)



- Passo 6 : Esecuzione TRAP
  - passa in kernel mode, disabilita le interruzioni, salva PC sullo stack, salta al codice dello smistatore

read(fd, buffer, nbytes) (6)

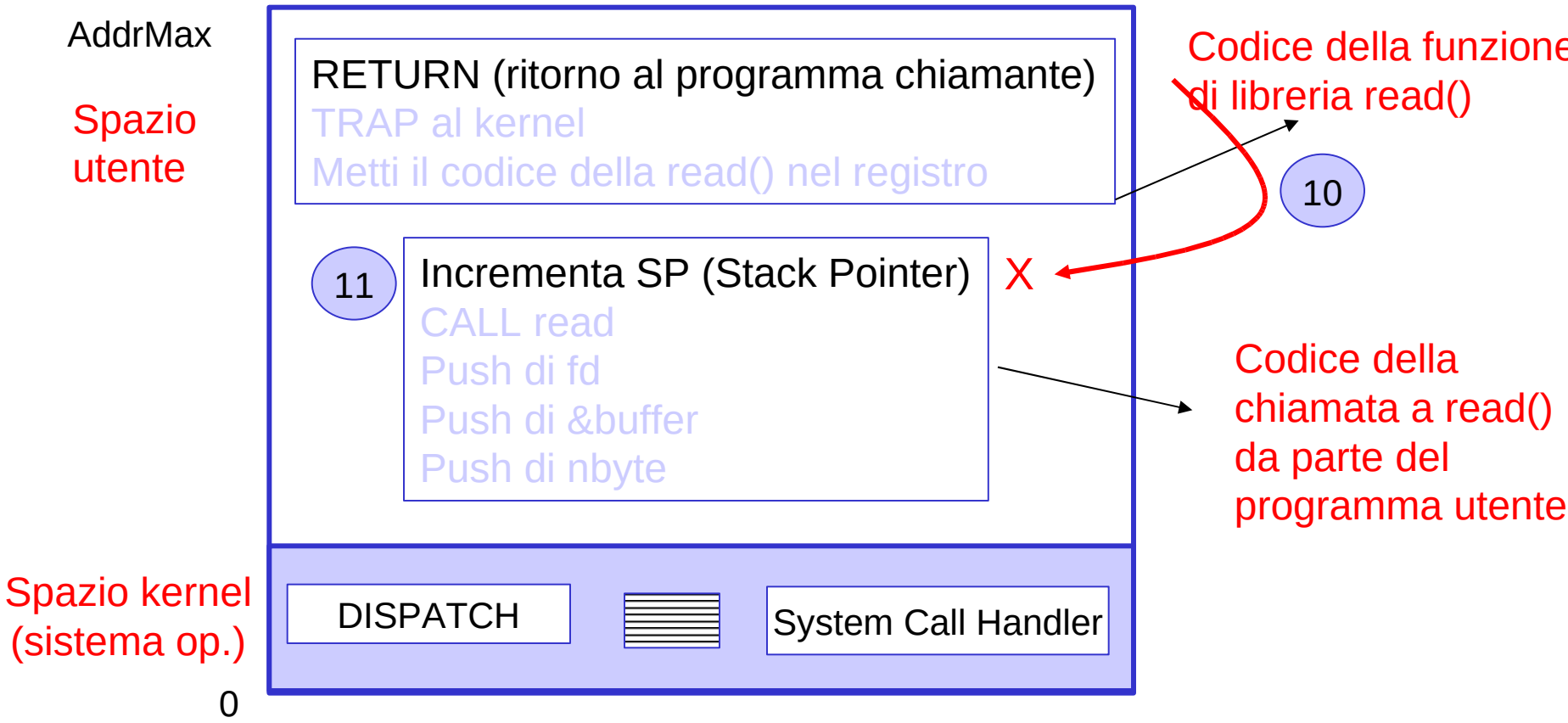


- Passo7-8: Seleziona della SC secondo il codice in Rx, salva stato processore nella *tabella dei processi, assegna a SP indirizzo della kernel stack,* riabilita interruzioni, esegui codice handler

read(fd, buffer, nbytes) (7)

- Passo 9 : Ritorno alla funzione di libreria
  - disabilita interruzioni, ripristina user mode, ripristina stato processore, carica PC con l'indirizzo dell'istruzione successiva alla TRAP (Y), riabilita interruzioni

read(fd, buffer, nbytes) (8)



- Passi 10-11 : Ritorno al codice utente (nel modo usuale)
  - PC = X, SP viene incrementato per eliminare il frame della read()

Chiamate di sistema: errori

- Le chiamate di sistema possono fallire
  - in caso di fallimento ritornano un valore diverso da 0
    - tipicamente -1, ma anche NULL, SIG_ERR o altro
    - leggere sempre accuratamente il manuale per ognuna
  - ci sono tantissime ragioni per cui una SC può fallire:
    - la maggior parte di esse inserisce il codice relativo all'errore rilevato nella variabile globale **`errno`** (in **`errno.h`**)
- E' molto importante isolare l'errore appena si verifica
  - si devono testare tutte le invocazioni a SC sistematicamente

Esempio: calling read()

```
/*error testing per una SC */
if ( (e = read (fd, buf, numbyte ) ) == -1 ) {
   /* gestione errore minimale */
   fprintf(stderr,"Read failed!\n");
   exit(EXIT_FAILURE); /* standard C stdlib.h*/
}
```

Esempio: calling read() (2)

```
#include <errno.h>
……

/*error testing per una SC */
if ( (e = read (fd, buf, numbyte ) ) == -1 ) {
   /* usare errno */
   fprintf(stderr,"Read failed! Err: %d\n",errno);
   exit(EXIT_FAILURE); /* standard C */
}
```

Esempio: calling read() (3)

-- possibile output se fd non è un file descriptor corretto

```
bash:~$ a.out
Read failed! Err: 9
bash:~$
```

Esempio: calling read() (3.1)

-- possibile output se fd non è un file descriptor corretto

```
bash:~$ a.out
Read failed! Err: 9
bash:~$
```

-- non molto significativo!

Chiamate di sistema: errori (2)

- Le chiamate di sistema possono fallire (cont)
  - i codici di errore sono definiti in vari file di include
  - **`perror()`** routine della libreria standard C che stampa i messaggi di errore relativi a diversi codici (includere **`stdio.h`**,**`errno.h`**)

Chiamate di sistema: errori (3)

- Esempi di codici di errore

```
/* no such file or directory*/
#define  ENOENT   2
/* I/O error*/
#define  EIO   5
/* Operation not permitted */
#define  EPERM   1
```

Esempio: calling read() (4)

```
#include <errno.h>
……

/*error testing per una SC */
if ( (e = read (fd, buf, numbyte ) ) == -1 ) {
   /* la funzione standard C perror
      stampa la stringa, e poi un messaggio di errore
      in base al valore di errno */
   perror(”Read”);
   exit(EXIT_FAILURE); /* standard C */
}
```

Esempio: calling read() (4.1)

-- possibile output se fd non è un file descriptor corretto (con perror())

```
bash:~$ a.out
Read: Bad file number
bash:~$
```

Chiamate di sistema: errori (4)

- Come funziona **`perror("msg")`**
  - legge il codice di errore contenuto nella globale **`errno`**
  - stampa "**`msg`**" seguito da "**`:`**" seguito dal messaggio di errore relativo al codice
  - uso tipico : **`perror("fun, descr")`** dove **`fun`** è il nome della funzione che ha rilevato l'errore, **`descr`** descrive cosa stiamo tentando di fare
  - la stampa viene effettuata sullo standard error

Chiamate di sistema: errori (5)

- Attenzione!!!!
  - **`errno`** è significativa solo se testata immediatamente dopo una chiamata di funzione che ha segnalato l'errore
  - viene sovrascritta dalle chiamate successive
  - Il programma deve controllare l'esito di ogni SC immediatamente dopo il ritorno ed agire di conseguenza
  - L'azione minima è chiamare la **`perror()`** per stampare un messaggio di errore
- come organizzare il test sistematico
  - diversi stili: macro con parametri, funzioni eventualmente **`inline`**

Esempio: test sistematico con macro

```
/* controlla -1; stampa errore e termina */
#define ec_meno1(s,m) \
  if ( (s) == -1 ) {perror(m); exit(errno);}
/* controlla NULL; stampa errore e termina (NULL) */
#define ec_null(s,m) \
  if((s)==NULL) {perror(m); exit(errno);}

/* controlla -1; stampa errore ed esegue c */
#define ec_meno1_c(s,m,c) \
  if((s)==-1) {perror(m); c;}
```

Esempio: test con macro... (2)

```
/* esempio di uso */
int main (void) {
…
    ec_null( p = malloc (sizeof(buf)), ”main” );
…
    ec_meno1( l = read(fd,buf,n), ”main” );
/* in caso di errore chiama una funzione di cleanup() */
    ec_meno1_c(l = read(fd,buf,n) ,”main”, cleanup());


…


}
```

Esempio: test con funzioni...

```
/* esempio di uso */
int main (void) {
    …
    p = Malloc (sizeof(buf));
    …
}

void* Malloc (size_t size) {
   void * tmp;
   if ( ( tmp = malloc(size) ) == NULL) {
      perror(”Malloc”);
      cleanup();
      exit(EXIT_FAILURE); }
   else
     return tmp;
}
```

SC che operano su file (1)

`open()`, `read()`, `write()`, `close()`, `unlink()`

Prologo

Implementazione del FS: i-node, file descriptor e tabella dei file aperti

Implementazione del FS di Unix

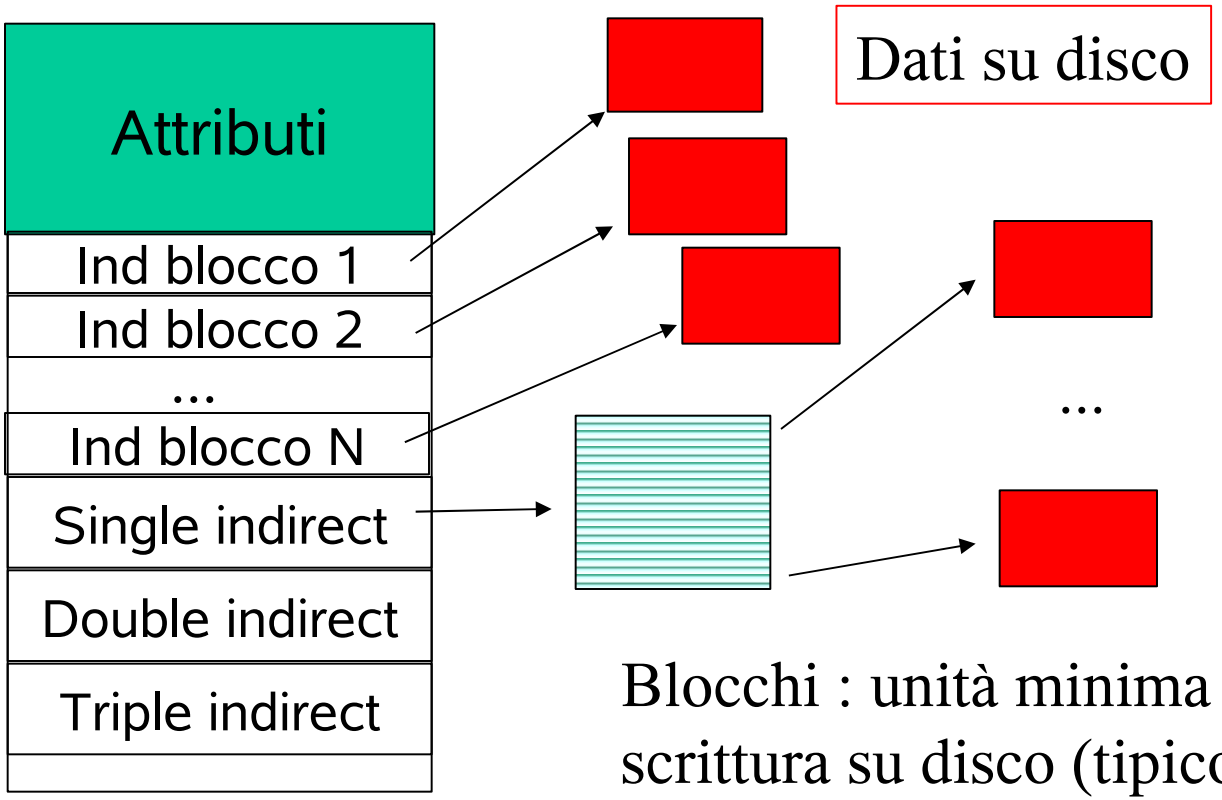
- Ogni file è rappresentato da un i-node.
- Cosa contiene un i-node:
  - tipo dl file **`-,d,l`** …
  - modo, bit di protezione (**`r-w-x`**)
  - **`uid, gid`** : identificativo utente e gruppo
  - **`size`**, tempi di creazione, modifica etc
  - campo count per i *link hard*
    - quante directory puntano a quell'i-node

Implementazione del FS di Unix (2)

- Cosa contiene un i-node :
  - *file regular, directory* :
    - indirizzo dei primi 10 blocchi su disco
    - indirizzo di uno o più blocchi indiretti
  - *device file* : major number, minor number (identificatore del driver e del dispositivo)
  - *link simbolico* : path del file collegato

Implementazione del FS di Unix (2)

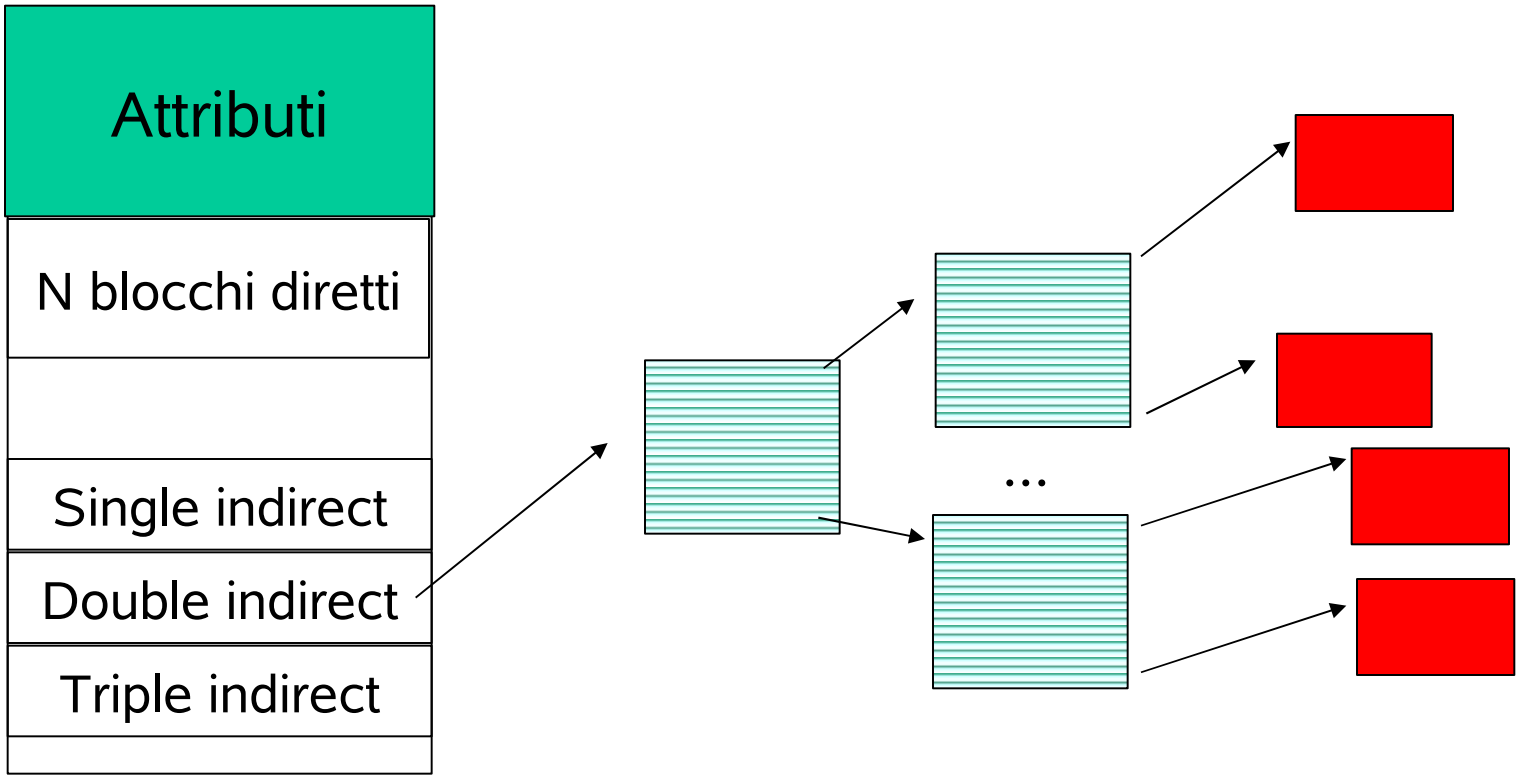
- *i-node* di un file regolare



Blocchi : unità minima di lettura scrittura su disco (tipico 1-4KB)
Ind. Blocco : tipicamente 4 byte

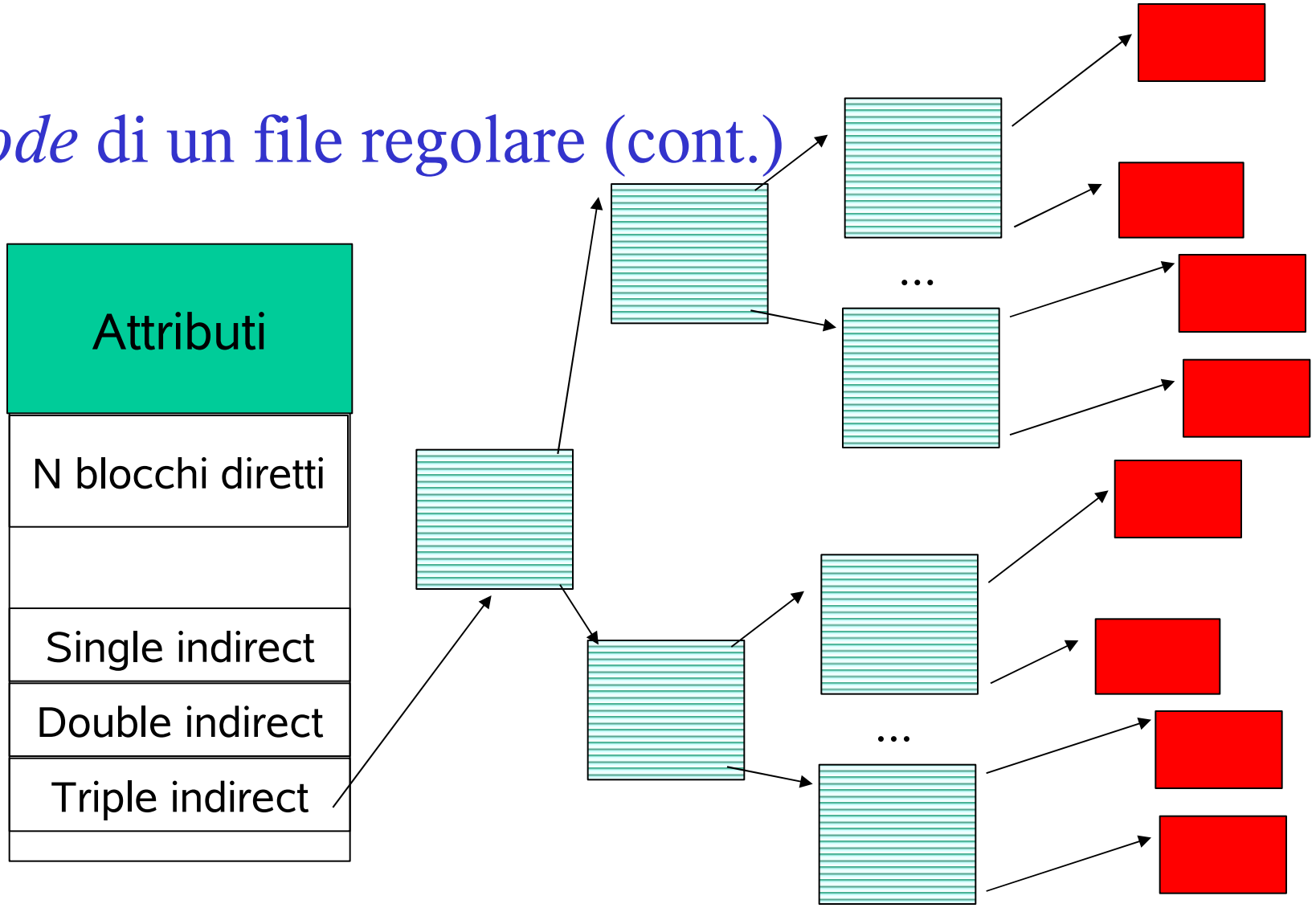
Implementazione del FS di Unix (3)

- *i-node* di un file regolare (cont.)



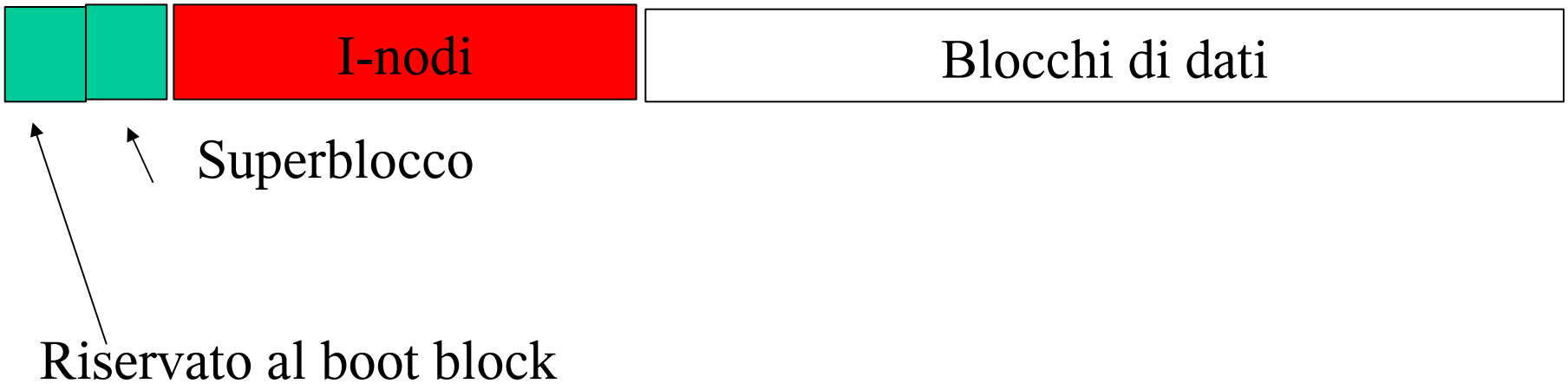
Implementazione del FS di Unix (4)

- *i-node* di un file regolare (cont.)



Implementazione del FS di Unix (5)

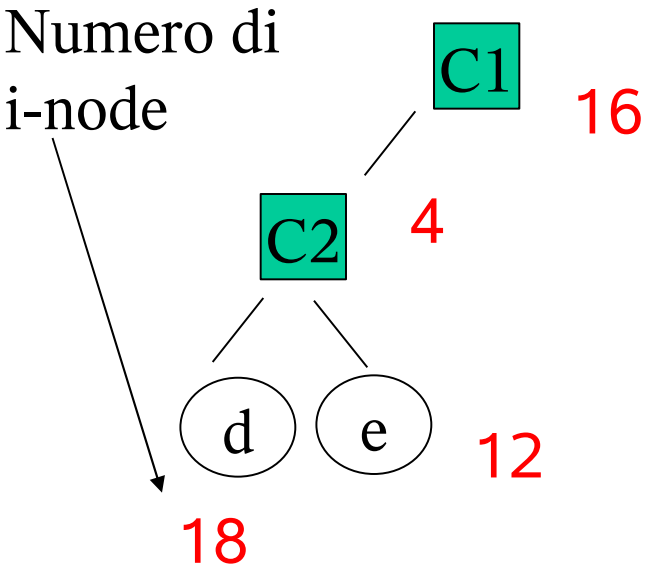
Organizzazione di una partizione in un file system tipico UNIX



Implementazione del FS di Unix (6)

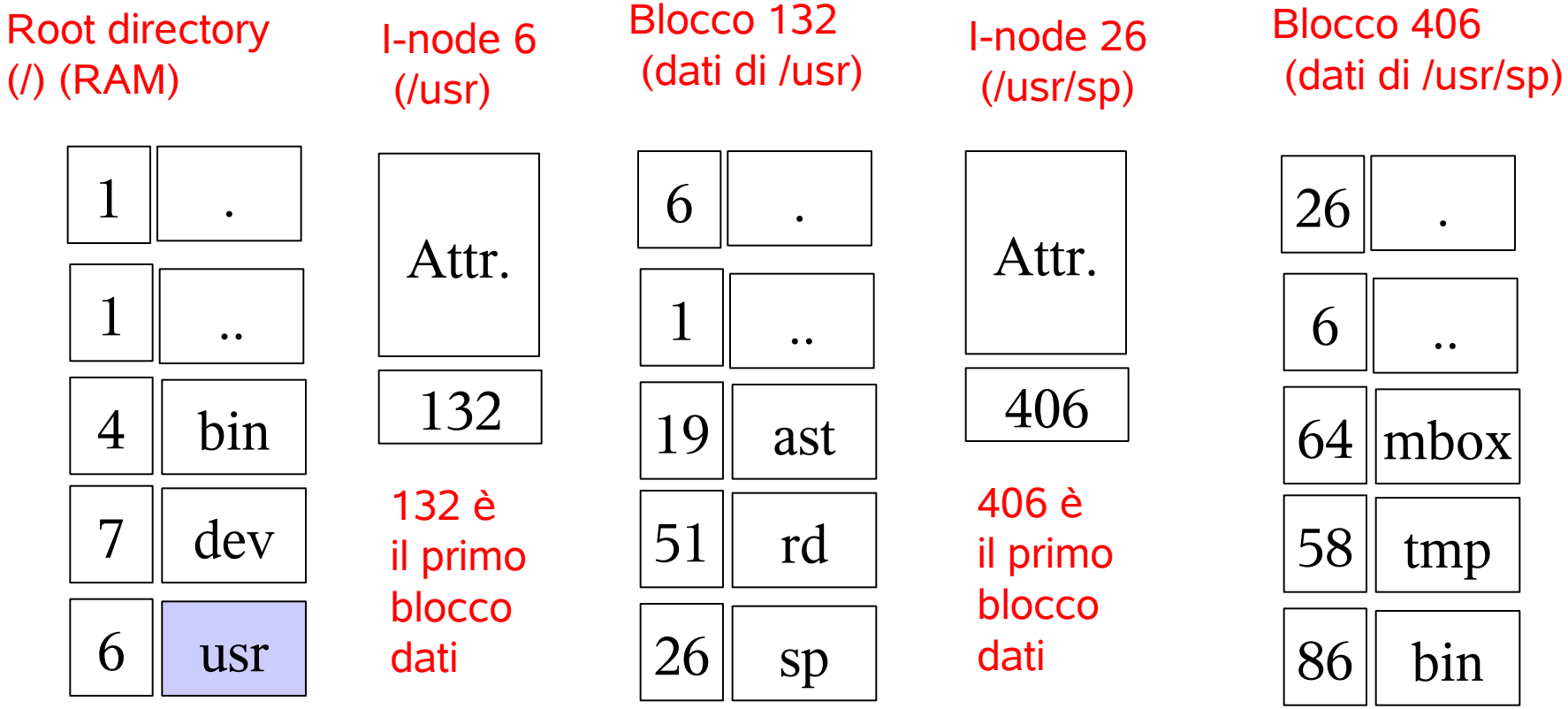
Organizzazione dei blocchi dati di una directory (Unix V7)

4	.(punto)
16	..(punto punto)
12	e
18	d



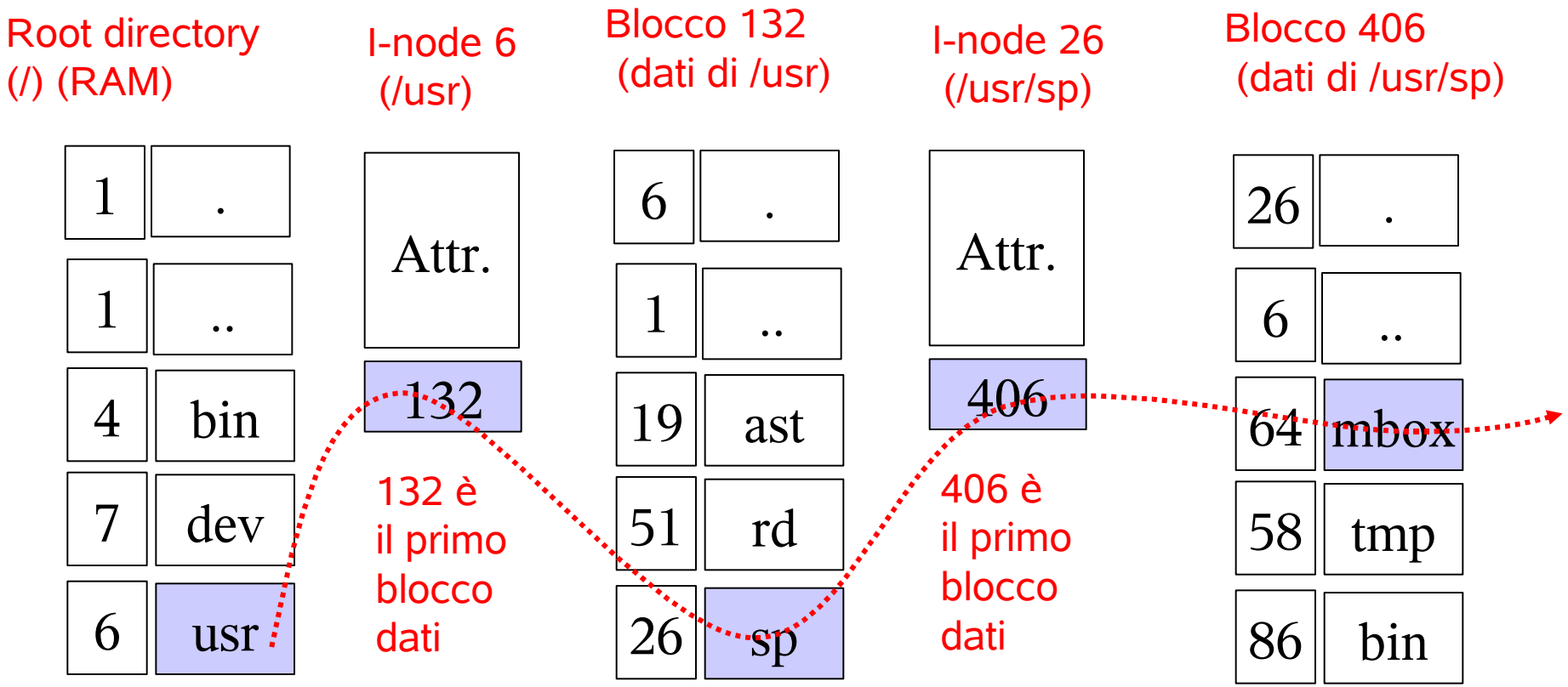
Blocco dati relativo alla directory C2

Implementazione del FS di Unix (7)



I passi necessari per aprire (open) */usr/sp/mbox*

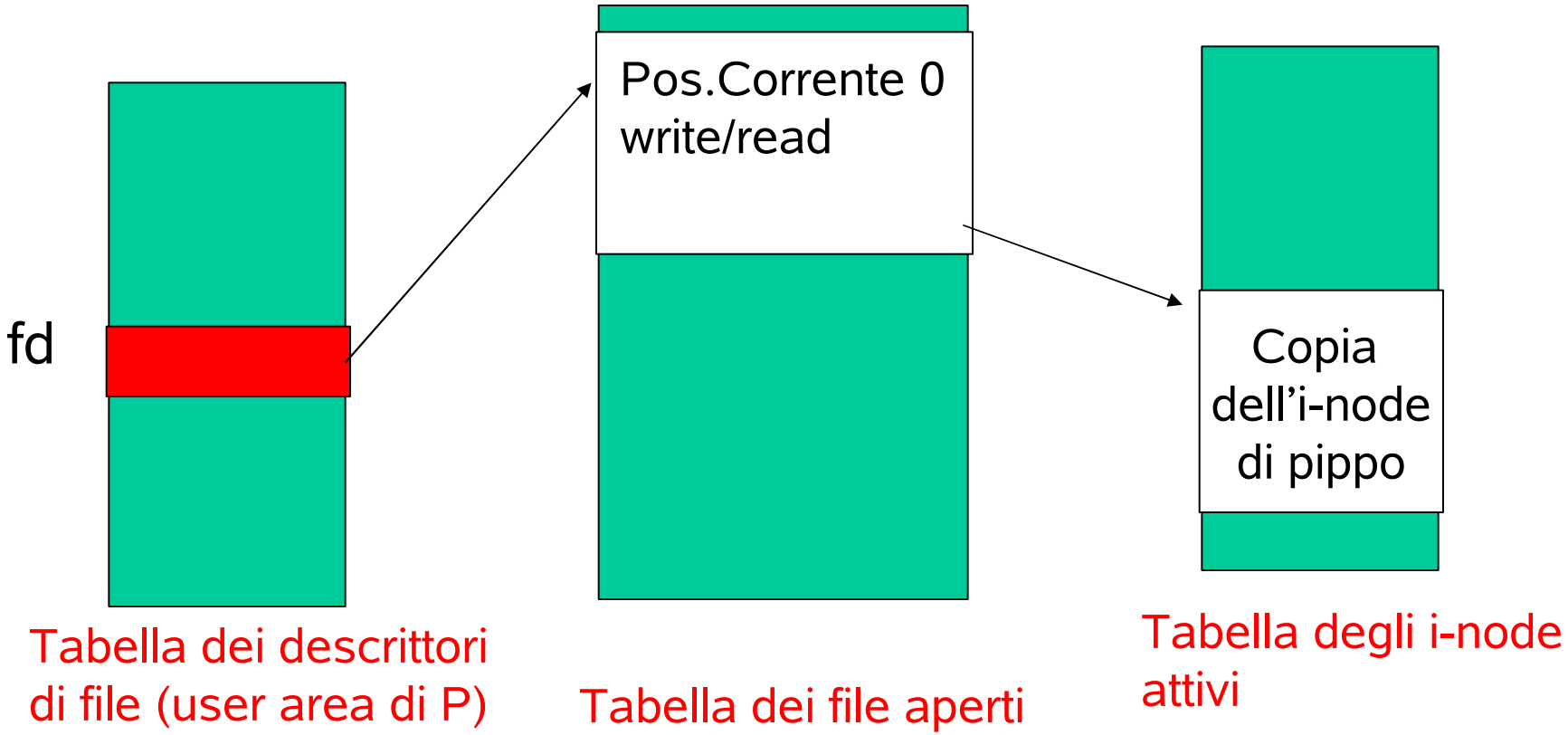
Implementazione del FS di Unix (8)



I passi necessari per leggere */usr/sp/mbox*

Tabelle di nucleo relative ai file

- Rappresentazione di un file aperto
  - subito dopo la **`open("pippo",O_RDWR)`** terminata con successo da parte del processo P



Apertura di un file : SC open()

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

int open(
  const char * pathname,
   int flags,
   mode_p permission
)
```

- **pathname** : PN relativo o assoluto del file
- **flags** : indicano come voglio accedere al file (segue)

Apertura di un file : SC open() (2)

- **`flags`** : indicano come voglio accedere al file
  - **`O_RDONLY`** sola lettura, **`O_WRONLY`** sola scrittura, **`O_RDWR`** entrambe
  - *eventualmente messe in or bit a bit una o più delle seguenti maschere* : **`O_APPEND`** scrittura in coda al file, **`O_CREAT`** se il file non esiste deve essere creato (solo file regolari), **`O_TRUNC`** in fase di creazione, se il file esiste viene sovrascritto, **`O_EXCL`** in fase di creazione, se il file esiste si da errore
- **`permission`** : indicano i diritti richiesti (se creiamo il file se no si può omettere)

Apertura di un file : SC open() (3)

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

- **`returns`** : un intero, il descrittore di file (fd) o (-1) in caso di errore (setta **`errno`**)

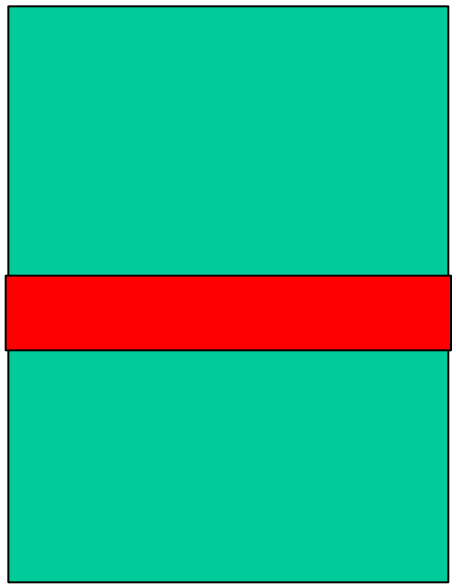


Tabella dei descrittori di file
(nella user area)
-- Array di strutture, una per ogni file aperto
-- Di ampiezza fissa (dipende dal sistema, almeno 20 **`_POSIX_OPEN_MAX`**) vedi: **`sysconf(_SC_OPEN_MAX)`**

Il **fd** è l'indice del descrittore assegnato al file appena aperto

Apertura di un file : SC open() (4)

- Tipico codice di apertura di un file :

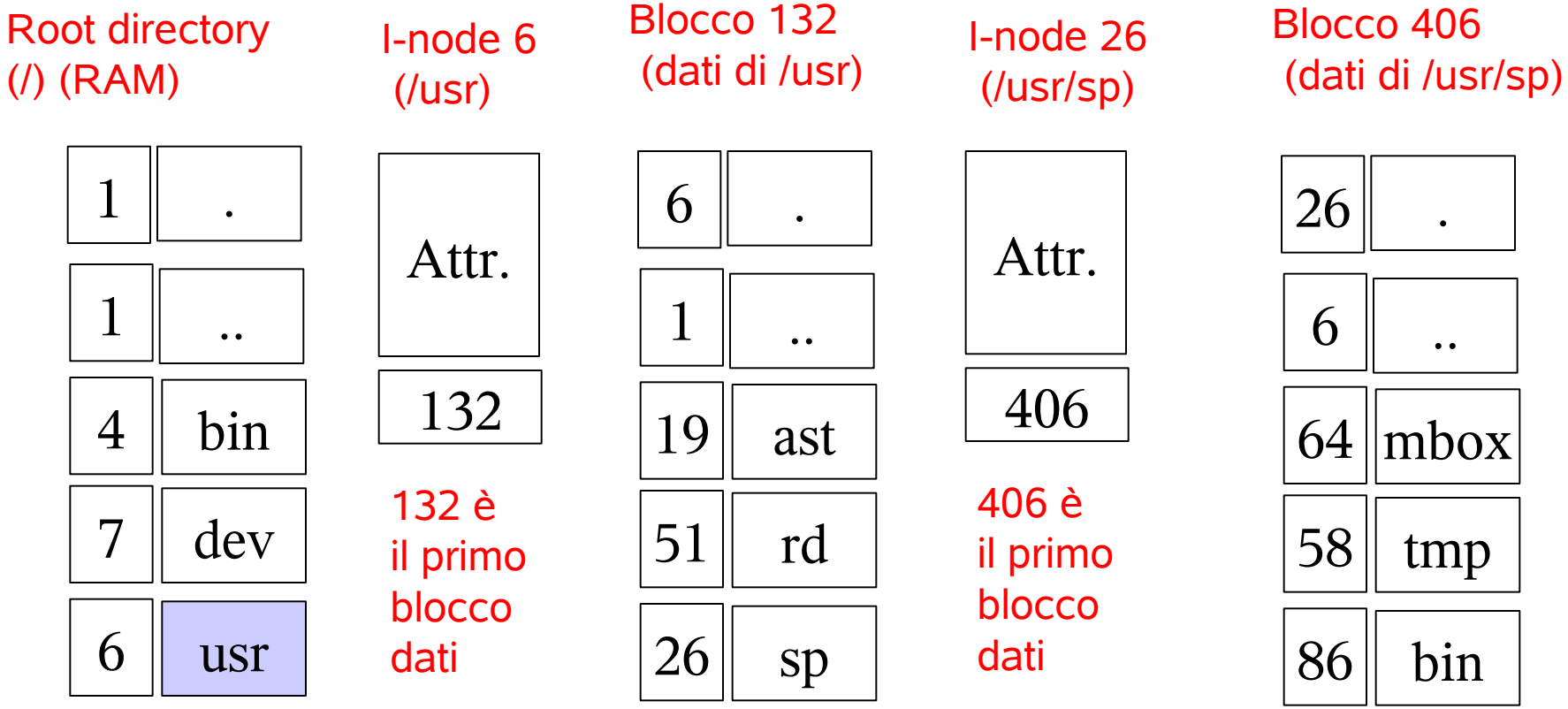
```
int fd;      /*file descriptor */

/* tento di aprire in sola lettura*/
if(( fd = open("s.c", O_RDONLY)) == -1) {
   perror("s.c, in apertura");
   exit(errno);  /* termina */
}
```

Apertura di un file : SC open() (5)

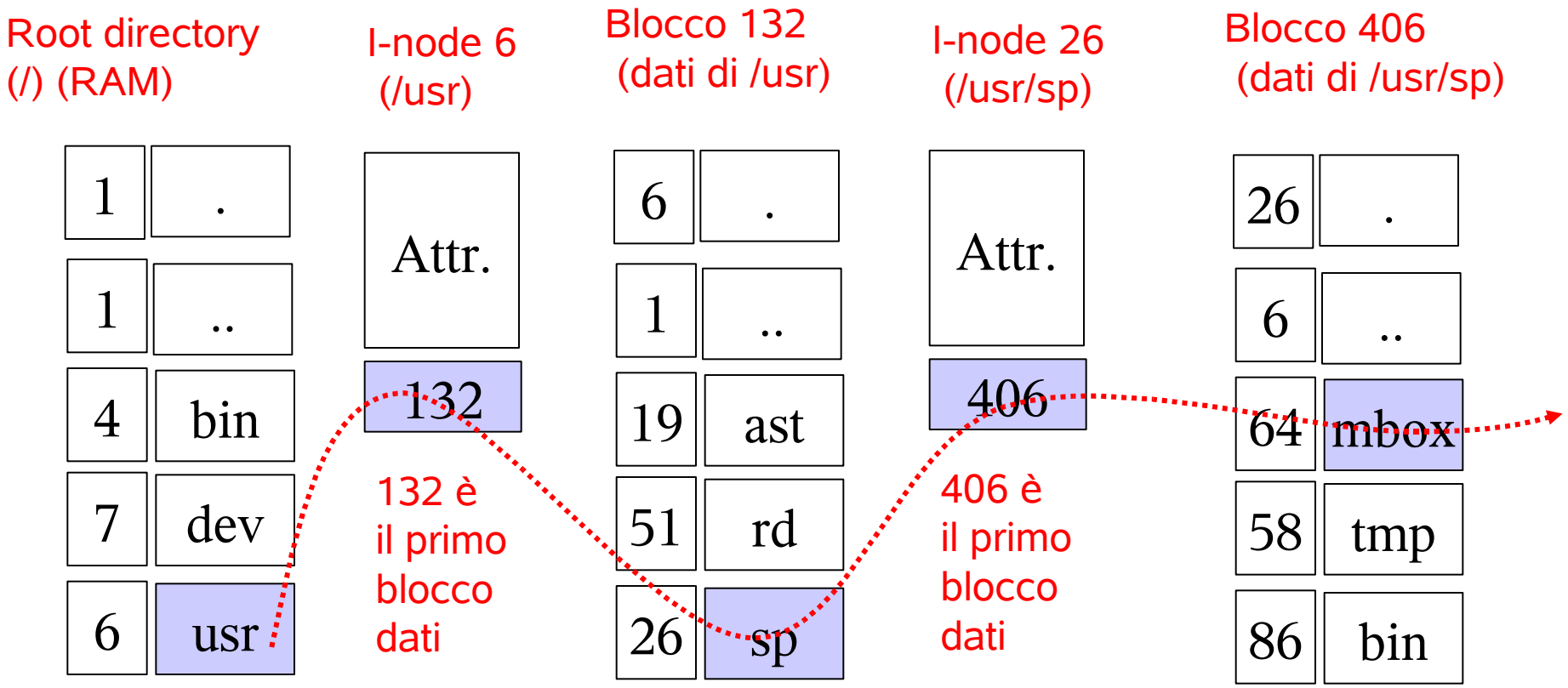
- Cosa fa la open :
  - segue il **`path`** del file per recuparare l'i-node corrispondente
  - controlla i diritti i accesso (li confronta con le richieste in **`flags`**)
  - se l'accesso è consentito assegna al file l'indice di una posizione libera nella tabella dei descr. (**`fd`**)
    - aggiorna le strutture dati interne al nucleo …
  - se si è verificato un errore ritorna -1 (**`errno`**)
  - altrimenti ritorna **`fd`**, che deve essere usato come parametro per tutti gli accessi successivi

Implementazione del FS di Unix (7)



I passi necessari per aprire (open) */usr/sp/mbox*

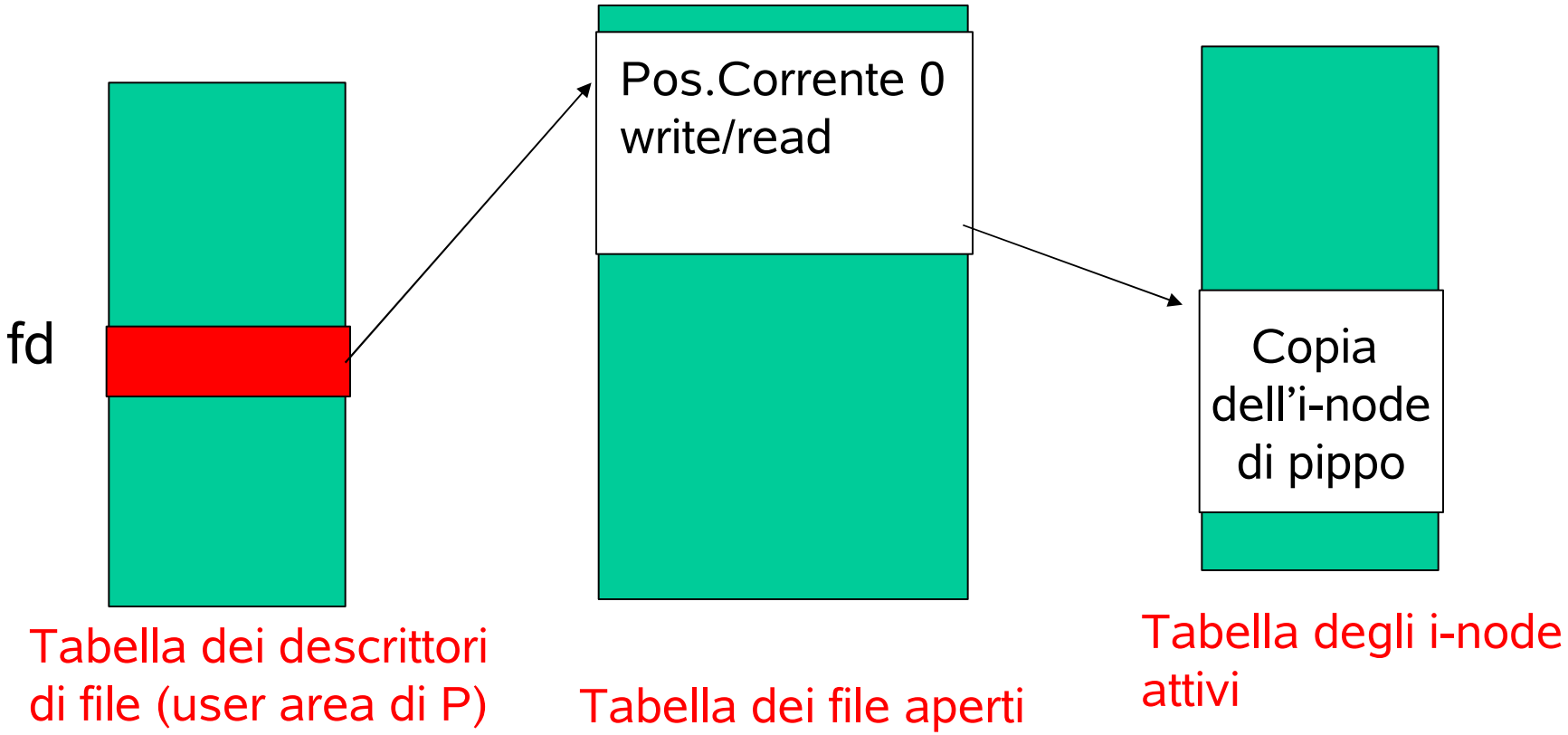
Implementazione del FS di Unix (8)



I passi necessari per leggere */usr/sp/mbox*

Tabelle di nucleo relative ai file

- Rappresentazione di un file aperto
  - subito dopo la **`open("pippo",O_RDWR)`** terminata con successo da parte del processo P



Lettura : SC read ()

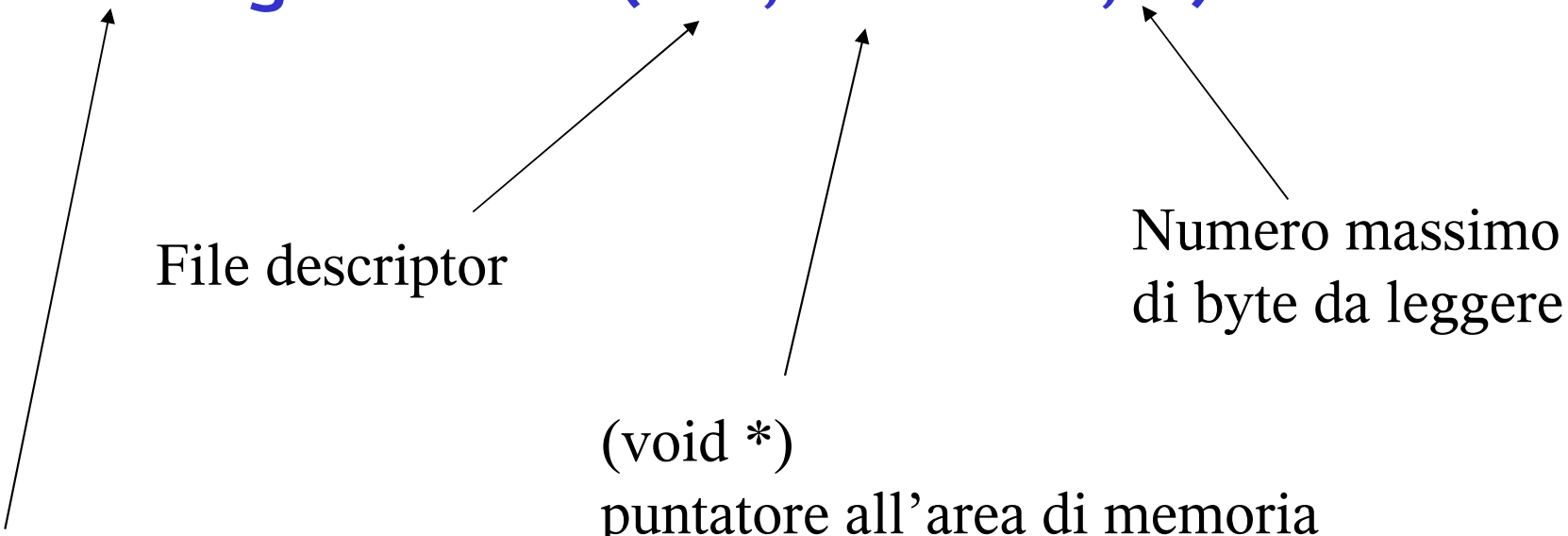
```
#include <unistd.h>

int read(
   int fd,                    /*file descriptor*/
   void * buffer, /* address to receive data*/
   size_t nbytes  /*amount(bytes) to read*/
)
/*returns (n) number of bytes read
(-1) on error sets errno */
```

- file regolari, ne riparliamo per file speciali

Lettura: SC read() (2)

- Es: **`lung = read(fd,buffer,N)`**



Effetto: Legge al più N byte a partire da Pos.Corrente, Pos.Corrente += lung

Lettura: SC read() (3)

- Tipico ciclo di lettura da file regolare:

```
int fd, lung;      /* fd, n byte letti */
char buf[N];       /* dove salvare i dati */
/* apertura file */
if ( (fd = open("s.c", O_RDONLY)) == -1)
 { perror("s.c"); exit(errno); }
/* file aperto OK */
while ((lung = read(fd,buf,N))>0){
    …
}
if ( lung == -1)
{ perror("s.c: lettura"); exit(errno); }
```

Scrittura : SC write ()

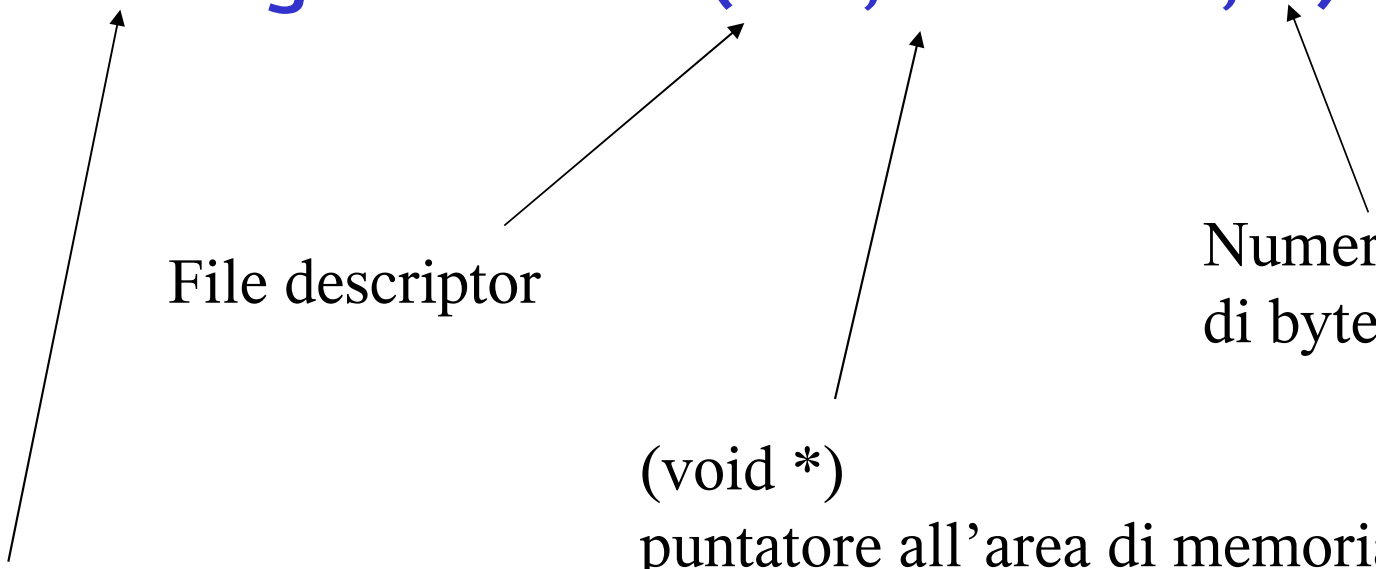
```
#include <unistd.h>

int write(
  int fd,                    /*file descriptor*/
  const void * buffer, /*data to write*/
  size_t nbytes  /*amount(bytes) to write*/
)
/*returns (n) number of bytes written
(-1) on error sets errno */
```

- file regolari, ne riparliamo per file speciali

Scrittura: SC write() (2)

- Es: `lung = write(fd,buffer,N)`



File descriptor

Numero massimo
di byte da scrivere

(void *)
puntatore all'area di memoria
dove andare a prendere i dati

-1 : errore
n => 0 : numero
byte scritti

Effetto: Scrive al più N byte a partire da Pos.Corrente, Pos.Corrente += lung

Scrittura: SC write() (3)

- Es. scrittura sullo *stdout* (fd 1) di un file regolare

```
int fd, lung;
char buf[N];
/*… apertura file etc …*/

while ((lung = read(fd,buf,N))>0){
if ( write(1, buf, lung) == -1)
  { perror("s.c: scrittura");
  exit(errno); }
}

if ( lung == -1)
   { perror("s.c: lettura"); exit(errno); }
```

Chiusura file : SC close ()

```
#include <unistd.h>

int close(
   int fd                    /*file descriptor*/
)
/* returns (0) success (-1) error (sets
   errno) */
```

- libera il file descriptor (che può essere riutilizzato), la memoria nelle tabelle di nucleo ed eventualmente l'i-node
- NON fa il 'fflush' del *buffer cache* nel kernel
  - la write reale può avvenire dopo! (**fsync**()…)

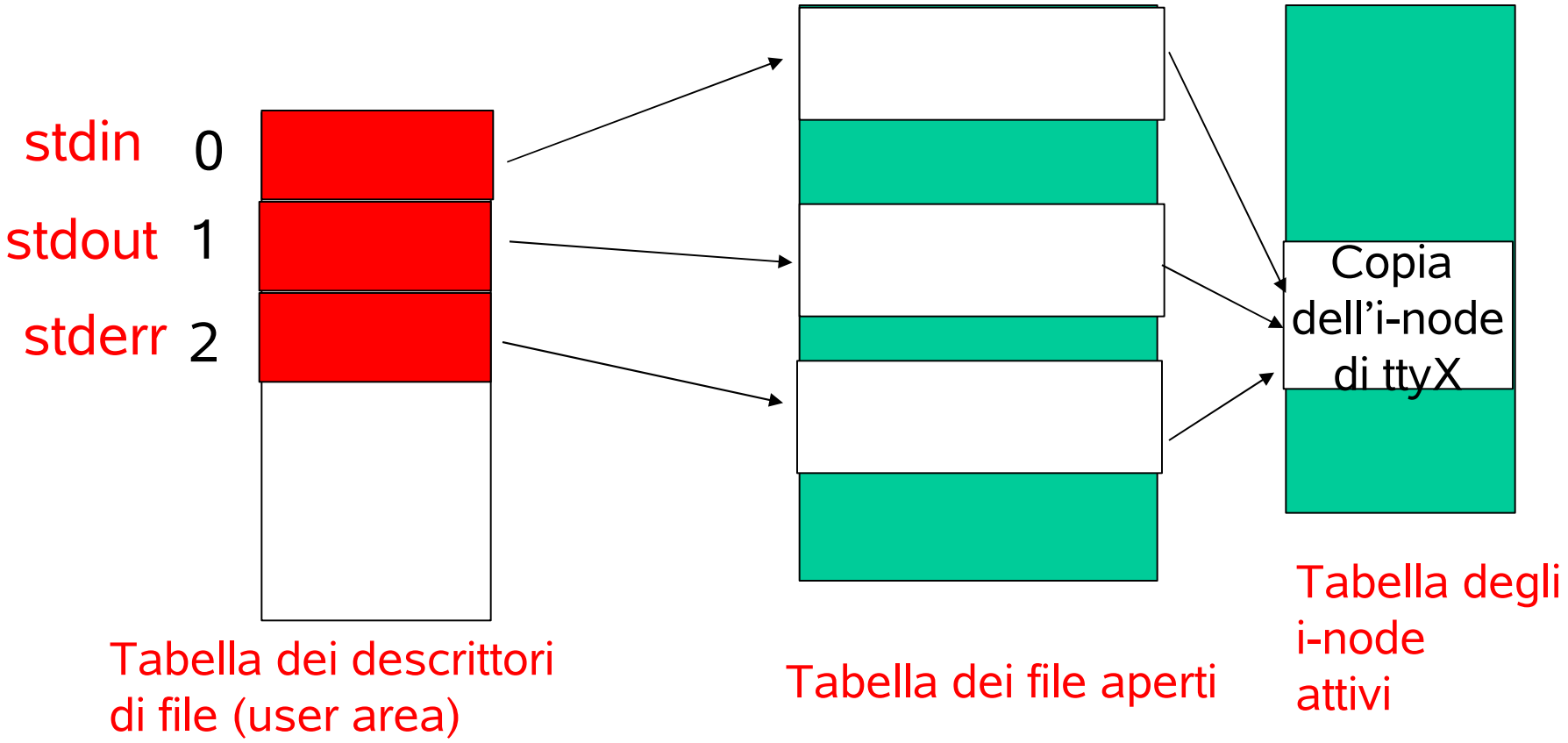
Chiusura: SC close() (2)

- Es. chiusura di un file ….

```
int fd, lung;
char buf[N];
/*… apertura file etc …*/
while ((lung = read(fd,buf,N))>0){
if ( write(1, buf, lung) == -1)
  { perror("s.c: scrittura");
  exit(errno);}
}
if ( lung == -1)
   { perror("s.c: lettura"); exit(errno);}
if ( close(fd) == -1)
{ perror("s.c: close"); exit(errno);}
```

Standard input, output and error

- Situazione tipica



SC vs standard I/O library

- **`open()`, `read()`, `write()`, `close()`** fanno parte della libreria standard POSIX per i file e corrispondono a System Calls
  - permettono di effettuare I/O su file regolari in blocchi di ampiezza arbitraria, non sono bufferizzate in spazio utente
  - non bufferizzano stdin, out ed error
- **`fopen()`, `fread()`, `fwrite()`, `fclose()`, `printf()`** fanno parte della libreria standard di I/O (**`stdio.h`**) definito dal comitato ANSI
  - forniscono I/O bufferizzato in spazio utente (ampiezza fissata dalla macro **`BUFSIZ`**)
  - tipicamente più efficienti e veloci

SC vs standard I/O library (2)

- La standard I/O library bufferizza anche stdout
  - se il programma termina in modo anomalo i buffer possono non essere svuotati in tempo
  - **`fflush()`** permette di svuotare i buffer
- mischiare chiamate ad I/O bufferizzato e non sullo stesso stream può portare a risultati impredicibili
  - usate o le SC (non bufferizzate) o le chiamate alla lib standard (bufferizzate) ma non entrambe

Open: ancora su creazione file…..

- Se ho specificato **`O_CREAT`** e il file non esiste
  - crea il nuovo file
  - calcola i diritti di accesso mettendo in AND il valore di **`permissions`** con il complemento della *file mode creation mask* (**`umask`**) del processo (si eredita dal padre)
  - esempio:

    `open("ff",O_CREAT|O_RDWR,0666)`

  -

Open : umask

– es. (cont) **open("ff",O_CREAT|O_RDWR,0666)**

```
bash:~$ umask
0022                    /* 000 010 010 ottale */
bash:~$ ls -l ff
-rw-r--r-- 1 … … susanna users  … …  ff
bash:~$
```

mentre il terzo parametro della open specificava:

```
 r  w  -  r  w  -  r  w  -
 1  1  0  1  1  0  1  1  0
    6        6        6
```

?????????

Open : umask (2)

- umask
  - fornisce una restrizione ai diritti di accesso di un file al momento della creazione
  - il modo del file viene calcolato come

```
perm & ~ (umask)
```

- Tipicamente umask = 0022 quindi :

```
1 1 0 1 1 0 1 1 0    (perm 0666)
0 0 0 0 1 0 0 1 0    (umask)
1 1 1 1 0 1 1 0 1    (~umask)
1 1 0 1 0 0 1 0 0    (perm & (~umask))
r w - r - - r - -
```

Open : umask (3)

- Si può modificare il valore di umask con il comando *umask* o la SC *umask()*

```
bash:~$ umask
   -- fornisce il valore corrente della
    maschera
bash:~$ umask valore_ottale
   -- setta umask al valore_ottale
```

  - Il valore di umask viene ereditato dal padre e vale fino alla prossima modifica
  - *ATTENZIONE: I file creati con la ridirezione usano la open() con modo 0666, e quindi sono sensibili al valore di umask*

Open : ancora qualcosa ...

- **`open()` e `creat()`:**

```
creat(path,perms)
open(path,O_WRONLY|O_CREAT|O_TRUNC,perms)
   -- sono equivalenti
   -- nel corso useremo sempre la open
```

- owner e gruppo del file creato
  - l'owner è l'*effective-user-id* del processo
  - il gruppo è *effective-group-id* del processo o il *group-id* della directory dove il file viene creato
- altri flag sono disponibili
  - li spiegheremo quando servono

Cancellare : SC unlink()

```
#include <unistd.h>

int unlink (
   const char * pathname
)
```

- **`pathname`** : PN relativo o assoluto del file
- elimina un link riducendo il contatore degli hard link nell'i-node, se il contatore va a 0 il FS elimina il file (*blocchi e i-node inseriti fra i liberi*)
  - funziona con tutti i tipi di file eccetto directory (**`rmdir`**())
- **`returns`** : (0) se OK o (-1) in caso di errore e setta **`errno`**

Cancellare : SC unlink() (2)

- se qualche processo ha il file ancora aperto l'eliminazione viene ritardata finchè tutti hanno chiamato la **`close()`**
- si può sfruttare per lasciare l'ambiente pulito in caso di file temporanei. es:

…

```
fd = Open (”temp”, O_RDWR|O_CREAT|O_TRUNC,0);
Unlink(temp);
```

…

```
/* in questo modo se il processo termina per
   qualsiasi ragione il file ‘temp’ viene
   automaticamente eliminato senza bisogno di
   fare altro */
```

SC che operano su file (2)

`lseek()`, `stat()`

Posizionamento : lseek ()

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(
   int fd,                 /*file descriptor*/
   off_t offset,              /*position*/
   size_t whence           /*from where?*/
)
/*returns (n>=0) new file offset (bytes)
  (-1) on error (sets errno) */
```

- **whence** può essere **SEEK_SET** (inizio file) , **SEEK_CUR** (posizione corrente) o **SEEK_END** (fine file)
- **offset** di quanti byte voglio spostarmi (anche negativo)

Posizionamento : lseek() (2)

- Esempi:

```
/* inizio e fine file */
lseek(fd, 0, SEEK _SET);
lseek(fd, 0, SEEK _END); /* (*) */
/*conoscere la posizione corrente*/
pos = lseek(fd, 0, SEEK _CUR);
/*indietro di un byte*/
lseek(fd, -1, SEEK _CUR);
/* esattamente in posizione k */
lseek(fd, k, SEEK _SET);
```

NOTA: in un file aperto con **`O_APPEND`** *(*) precede ogni* **`write()`** *(atomico!)*

Attributi : stat(),fstat()

```
#include <sys/stat.h>
int stat(
   const char *path,    /*pathname*/
   struct stat *buf     /*informazioni
                          restutuite da stat*/
)
int fstat(
   int fd,              /*file descriptor*/
   struct stat *buf     /*informazioni ..*/
)
/* return (0) success (-1) on error
   (set errno) */
```

Attributi : stat(),fstat() (2)

```
/* struttura tipica: può variare in diverse
   implementazioni */
struct stat {
   …
  ino_t     st_ino;    /* # i-nodo*/
  mode_t    st_mode;   /* diritti protezione*/
  nlink_t   st_nlink;  /* # hard link */
  uid_t     st_uid;    /* ID owner */
  off_t     st_size;   /* lung totale (byte)*/
  time_t    st_atime;  /* ultimo accesso*/
  time_t    st_mtime;  /* ultima modifica */
  time_t    st_ctime;  /* ultima var i-node */
}
```

Attributi : stat(), fstat() (3)

```
struct stat info;
if ( stat(“./dati”,&info)== -1) ){
    /* gestione errore */ }

if (S_ISLNK(info.st_mode)){ /* link simbolico*/ }
if (S_ISREG(info.st_mode)){ /* file regolare*/ }
if (S_ISDIR(info.st_mode)){ /* directory */ }
if (S_ISCHR(info.st_mode)){ /* sp caratteri */ }
if (S_ISBLK(info.st_mode)){ /* sp blocchi */ }

if (info.st_mode & S_IRUSR){ /* r owner */ }
if (info.st_mode & S_IWGRP){ /* w group */ }
```

Esempio: stampare gli attributi

```
void printattr(char * path) {
struct stat info;
if ( stat(“./dati”,&info)== -1) ){/* gestione errore */}
else {printf(“Attributi %s:\n”,path); /* nome file */
  printf(“tipo: ”); /* stampa il tipo */
  if (IF_ISREG(info.st_mode)) printf(“regular”);
  else if (IF_ISDIR(info.st_mode)) printf(“directory”);
  else if (IF_ISLNK(info.st_mode)) printf(“link simb”);
  else if (IF_ISCHR(info.st_mode)) printf(“character \
  special file”);
  else if (IF_ISBLK(info.st_mode)) printf(“block \
  special file”);
  else if (IF_ISFIFO(info.st_mode)) printf(“pipe”);
  else if (IF_ISSOCK(info.st_mode)) printf(“socket”);
  printf(“non riconosciuto”);
/* continua……  */
```

Esempio: stampare gli attributi (2)

```
/* stampa il numero di i-node */
  printf("\n i node number %ld", (long)info.st_ino);
/* stampa il modo (formato rw---x--x) */
 /* user */
  if (S_IRUSR & info.st_mode) putchar('r');
  else putchar('-');
  if (S_IWUSR & info.st_mode) putchar('w');
  else putchar('-');
  if (S_IXUSR & info.st_mode) putchar('x');
  else putchar('-');
/* group */
  if (S_IRGRP & info.st_mode) putchar('r');
  else putchar('-');
  if (S_IWGRP & info.st_mode) putchar('w');
  else putchar('-'); /* continua.....  */
```

Esempio: stampare gli attributi (3)

```
/* continua group */
  if (S_IXGRP & info.st_mode) putchar('x');
  else putchar('-');
/* others */
  if (S_IROTH & info.st_mode) putchar('r');
  else putchar('-');
  if (S_IWOTH & info.st_mode) putchar('w');
  else putchar('-');
  if (S_IXOTH & info.st_mode) putchar('x');
  else putchar('-');
/* ultimo accesso */
  printf("ultima modifica: %s",ctime(&info.st_mtime));

/* continua …. */
```

Esempio: stampare gli attributi (4)

```
/* stampa uid, gid numerico */
 printf("uid %d\n", info.st_uid);
 printf("gid %d\n", info.st_uid);
}
/* per la stampa formato stringa di uid e gid si possono
   utilizzare le funzioni di libreria getpwuid() e
   getpwgrp() vedi man */
```

Alcune SC che operano su directory

`opendir`, `closedir`, `readdir`, `rewinddir`, `getcwd`

Directory

- Il formato delle directory varia nei vari FS utilizzati in ambito Unix
- Quando una directory viene aperta viene restituito un puntatore a un oggetto di tipo **`DIR`** (definto in **`dirent.h`**)
  - es. **`DIR* mydir;`**
- Per leggere le informazioni sui file contenuti esiste la chiamata di sistema POSIX **`getdents()`**
  - non la useremo direttamente

Directory (2)

- Useremo invece:
  - funzioni di libreria standard C conformi a POSIX che lavorano sul puntatore in modo trasparente e chiamano **`getdents`** quando necessario
  - *readdir, rewinddir, opendir, closedir, getcwd* (sez 3 manuali)
  - attenzione! : esiste anche una **`readdir`** chiamata di sistema (sez 2) di nuovo a basso livello

Directory: **opendir**

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

DIR* opendir(
  const char* path    /*directory name*/
)
/*returns (p) DIR pointer (NULL) on error
   (sets errno) */
```

- funziona in modo analogo all'apertura di un file con una **fopen()** (**DIR** ==> **FILE**)
- il puntatore ritornato va passato a tutte le altre funzioni

Directory: `closedir`

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

int closedir(
  DIR* dirp    /*directory pointer*/
)
/*returns (0)success (-1) error (sets errno)*/
```

Directory: opendir, closedir

```
DIR * d;
/* esempio di apertura directory */
if ((d = opendir(".")) == NULL){
  perror(”opening cwd");
  exit(errno);
}

/* lavoro sulla directory */

/* chiusura directory */
if (( closedir(d) == -1) ){
  perror(”closing cwd"); exit(errno);}
```

Directory: `readdir`

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

struct dirent* readdir(
  DIR* dirp    /*directory pointer*/
)
/*returns (p) structure pointer or
  (NULL) on EOF or error (sets errno)*/
```

– va chiamata ripetutamente in un ciclo, ogni volta ritorna il puntatore ad una struttura che descrive il prossimo file nella directory

Directory: `readdir` (2)

- ATTENZIONE: **`readdir`**() restutuisce NULL in due casi diversi:
  - sia quando non ci sono più file (siamo arrivati alla fine della directory), ovvero EOF
  - sia quando si verifica un errore
- l'unico modo per distinguere correttamente i due casi è utilizzare la variabile **`errno`**, che viene settata solo se si è verificato un errore
  - conviene settare errno a 0 prima di ogni invocazione e testarlo subito dopo per discriminare correttamente i due casi

Directory: readdir (3)

```
/* POSIX fields di struct dirent … gli altri
   dipendono dall'implementazione  */
struct dirent {
  …
 /* # di i-node */
 ino_t d_ino;
/* nome del file (con terminatore)*/
 char d_name[];
}
```

Directory: readdir (4)

```
DIR * d;
struct dirent* file;
if ((d = opendir(".")) == NULL){ perror(”opening cwd");
   exit(errno); }
/* lettura di tutte le entry della directory */
/* settiamo ogni volta errno a 0 per evitare
   sovrascritture in printattr() */
while ( (errno = 0, file = readdir(d))!=NULL) {
      printattr(file->d_name); /* stampa info file */
   }
if (errno != 0) { /* trattamento errore */ }
else { /* trattamento caso OK */ }
/* chiusura directory */
if (( closedir(d) == -1) ){ perror(”closing cwd");
   exit(errno);}
```

Directory: **`readdir`** (4)

- PROBLEMA: il codice appena visto funziona solo per la directory corrente ('.')
  - la printattr() chiama la **`stat`** che ha bisogno del path completo
  - file.nome è solo il nome del file e non il suo pathname relativo
  - es: pippo viene interpretato come ./pippo e tutto funziona perché sono nella directory giusta
- per farlo funzionare semplicemente con directory diverse bisogna essere in grado di cambiare directory
  - vediamo subito alcune SC e funzioni relative alla working directory

Directory corrente? **getcwd**

```
#include <unistd.h>

char* getcwd (
   char* buf,          /*where write path*/
   size_t bufsize      /*size of buffer*/
)
/*returns (pathname) success (NULL) error
    (sets errno)*/
```

- attenzione: se il buffer non è abbastanza lungo getcwd() ritorna NULL con errore **ERANGE**
  - in questo caso è possibile allocare un buffer più lungo e ritentare

Cambiare la directory corrente ...

```
#include <unistd.h>

int chdir(
  const char* path /* path new cwd*/
)
int fchdir(
  int fd       /* file descriptor new cwd*/
)
/*return (0) success (-1) error (set
  errno)*/
```

Directory readdir: esempio rivisto

```
/* stampa gli attributi di tutti i file di cwd (.) */
void processdir (void) {
  DIR * d;
  struct dirent* file;
  if ((d = opendir(".")) == NULL)
    { perror(”opening cwd");  exit(errno); }
  while ( (errno = 0, file = readdir(d))!=NULL) {
      printattr(file->d_name); /* stampa info file */
  }
 if (errno != 0) { /* trattamento errore */ }
 else { /* trattamento caso OK */ }
/* chiusura directory */
if (( closedir(d) == -1) ){ perror(”closing cwd");
  exit(errno);}
 }
```

Directory readdir: esempio rivisto (2)

```
/* nel primo argomento ho il nome della directory*/
void main (int argc, char** argv) {
  char buf[N];
   ……
  if (getcwd(buf,N)=NULL) { /* errore */
     perror(”getcwd");  exit(errno); }
  printf(”directory %s”,argv[1] );
/* mi sposto nella directory argv[1] */
  if ( chdir(argv[1]) == -1 ) { /* errore */
     perror(”chdir");  exit(errno); }
  processdir();
/* ritorno nella directory corrente */
  if ( chdir(buf) == -1 ) { /* errore */
     perror(”chdir");  exit(errno); }
…… } /* end main  */
```