

SC per Inter Process Communication

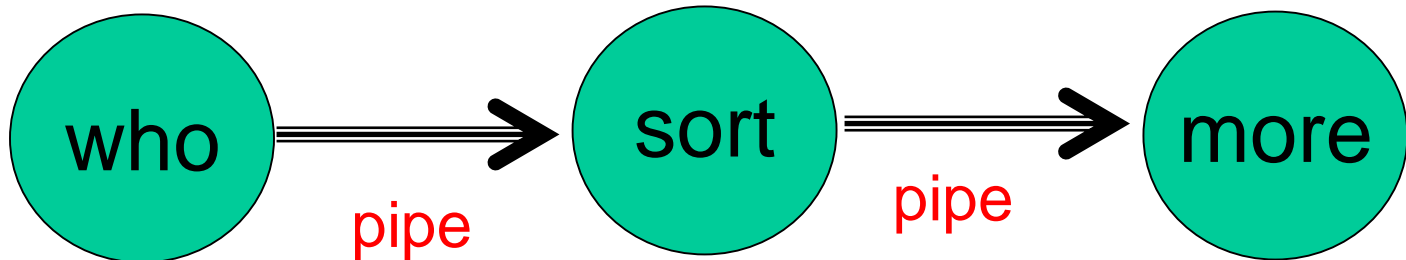
Pipe senza nome e con nome (FIFO)

Pipe

- **Pipe** : file speciali utilizzati per connettere due processi con un canale di comunicazione

- Possono essere utilizzati in modo unidirezionale da shell
es:

```
bash:~$ who | sort | more
```



Pipe (2)

- **In generale** : le pipe possono essere bidirezionali, connettere più processi fra loro etc
 - queste interazioni più complesse non sono direttamente disponibili da shell ma devono essere programmate esplicitamente usando le SC relative alle pipe
- Le pipe sono state uno dei primi meccanismi di IPC in Unix e sono disponibili su tutti i sistemi

Pipe con nome e senza nome

- pipe senza nome (unnamed) file senza un nome visibile, che viene utilizzato per comunicazioni fra processi che condividono puntatori alla tabella dei file aperti (es. padre figlio, nonno nipote etc)
- pipe con nome (named o FIFO) file speciale (**p**) con nome visibile a tutti gli altri processi sulla stessa macchina

Creazione di pipe senza nome: pipe()

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

```
int pipe(  
    int pfd[2] /*descrittori di file*/  
);  
/* (0) success (-1) error, sets errno */
```

- crea una pipe, rappresentata da due descrittori di file: `pfd[1]` con cui si può scrivere dati sulla pipe e `pfd[0]` con cui si può leggere dalla pipe

Pipe senza nome: esempio

```
int main (void) {
    int pfd[2];          /* descrittori */
    /* creazione pipe */
    if ( pipe(pfd) == -1 ) { /* errore */
        /* adesso pfd[1] puo' essere usato per la
           scrittura e pfd[0] per la lettura */
    }
}
```

Pipe senza nome

- Una volta creata la pipe è possibile leggere e scrivere dati con `read()` e `write()`
- la capacità minima della pipe è definita dalla variabile Posix `_POSIX_PIPE_BUF` ma può essere maggiore
 - vediamo come fare a vederne il valore
- le `write()` sono garantite essere atomiche solo se vengono scritti meno byte della capacità effettiva della pipe
- Alla creazione, il flag `O_NONBLOCK` non è settato, quindi sia lettura che scrittura possono bloccarsi.

Limiti di configurazione: *pathconf()

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

```
long fpathconf(
```

```
    int fd,    /*descrittore di file*/
```

```
    int name   /*opzione di configurazione*/
```

```
);
```

```
long pathconf(
```

```
    char * path,    /*path file*/
```

```
    int name        /*opzione di configurazione*/
```

```
);
```

```
/* (limite o -1) on success (-1) on error,  
   sets errno */
```


Limiti ... : *pathconf() (2)

- Come opera **fpathconf (pfd, name)** :
 - forniscono il limite corrente per l'opzione **nome**
 - attenzione: può ritornare -1 sia se non c'è limite all'opzione, sia se si è verificato un errore
 - al solito possiamo discriminare i due casi settando **errno** a 0 appena prima della chiamata
 - vediamo come leggere la capacità minima Posix e stabilire la capacità reale!

Pipe senza nome: esempio

```
int main (void) {
    int pfd[2];
    long int v;
    /* creazione pipe */
    if ( pipe(pfd) == -1 ) { /* errore */
        printf("POSIX = %ld e ", _POSIX_PIPE_BUF);
        errno = 0;
        if ((v = fpathconf(pfd[0], _PC_PIPE_BUF)) == -1) {
            if (errno != 0) { /* errore */
                else printf("reale = illimitato\n");
            }
        }
        else
            printf("reale = %ld\n", v);
    }
    ... }
```

Pipe senza nome : esempio (2)

- Compiliamo:

```
bash:~$ gcc main.c -o pipetest
```

- Eseguiamo:

```
bash:~$ ./pipetest
```

```
POSIX = 512 e reale = 4096
```

```
bash:~$
```

Uso di una pipe senza nome

- Si usano le SC che abbiamo già visto per gli altri file
 - **write**, **read**, **close** hanno però una semantica diversa da quella sui file regular, la descriveremo in dettaglio
 - **stat** permette di capire se un file è una pipe
 - **lseek** non ha senso per le pipe
- più altre che non abbiamo ancora visto
 - **dup**, **dup2**: le descriveremo dopo, sono fondamentali per implementare *pipelining* e *ridirezione* di shell

Uso di una pipe: write

- **k = write(pfd, buf, n)**
 - i dati sono scritti nella pipe in ordine di arrivo
 - se **n** è minore della capienza la write è atomica:
 - non ci sono scritture parziali, la **write** si blocca finché non sono stati letti abbastanza dati dalla pipe per far posto ai dati nuovi (comportamento normale **O_NONBLOCK** non settato -- si può settare con **fcntl**)
 - **n** è maggiore della capienza la write non è atomica, ci possono essere scritture parziali
 - se **O_NONBLOCK** è settato la write scrive subito tutto oppure ritorna -1 (**errno** settato a **EAGAIN**)

Uso di una pipe: write (2)

- **k = write(pfd, buf, n)**
 - se non esiste nessun lettore sulla pipe **pfd** (tutti i descrittori di lettura sono stati chiusi)
 - una write su **pfd** provoca l'invio di un segnale **SIGPIPE** al processo
 - **MOLTO PERICOLOSO!** : la gestione di default di **SIGPIPE** è la terminazione del processo che lo riceve
 - è necessario ridefinire la gestione ignorando **SIGPIPE** (vedremo come è possibile quando parliamo di segnali)
 - se **SIGPIPE** è stato ignorato, una scrittura su una pipe senza lettori ritorna -1 errore **EPIPE** (equivale a EOF)

Uso di una pipe: read

- **`k=read(pfd, buf, n)`**
 - i dati sono letti dalla pipe in ordine di arrivo
 - i dati letti non possono essere rimessi nella pipe
 - comportamento normale **`O_NONBLOCK`** non settato
 - se la pipe è vuota e i descrittori di scrittura non sono ancora stati chiusi la read si blocca in attesa
 - se i descrittori di scrittura sono tutti chiusi ritorna subito 0 (*end_of_file*)
 - altrimenti legge al più **`n`** byte, se al momento dell'invocazione la pipe contiene meno di **`n`** byte legge tutti i byte presenti e ritorna il numero dei byte letti **`k`**

Uso di una pipe: read (2)

- `k=read(pfd,buf,n)`
 - se `O_NONBLOCK` è settato
 - se la pipe è vuota la read ritorna subito -1 con errore settato a **EAGAIN**

Uso di una pipe: close

- `k=close (pfd)`
 - libera il descrittore di file (come file regolari)
 - quando l'ultimo descrittore di scrittura è chiuso genera l'*end_of_file* per i lettori
 - facendo ritornare 0 ad eventuali read in attesa
 - se viene effettuata una write su una pipe in cui tutti i descrittori di lettura sono stati chiusi il processo riceve un segnale fatale (**SIGPIPE**)

Usi di una pipe senza nome

- Solo per processi discendenti
 - si passano i file *descriptor* attraverso la condivisione dei puntatori alla tabella dei file aperti
- Tipicamente per comunicazione unidirezionale
 - uno scrittore ed un lettore
 - più scrittori ed un lettore
 - negli altri usi bisogna fare molta attenzione a possibili situazioni di deadlock
- Non esiste il concetto di messaggio
 - byte non strutturati, i processi comunicanti devono stabilire il protocollo di scambio dati

Uso di una pipe senza nome (2)

- Sequenza tipica di utilizzo di una pipe senza nome:
 - il padre crea la pipe
 - il padre si duplica con una `fork()`
 - i file descriptor del padre sono copiati nella tabella dei file descriptor del figlio
 - il processo scrittore (padre o figlio) chiude `pfid[0]` mentre il processo lettore chiude `pfid[1]`
 - i processi comunicano con `read/write`
 - quando la comunicazione è finita ognuno chiude la propria estremità

Pipe senza nome: esempio

```
/* msg lunghezza fissa N, manca gest. errori */
int main (void) {
    int pfd[2], pid, l;
    char msg[N];
    if ( pipe(pfd) == -1 ) {/* errore */}
    if ( (pid=fork()) == -1 ) {/* errore */}
    if ( pid ) { /* siamo nel padre */
        close(pfd[1]); /* chiude scrittura */
        l=read(pfd[0],msg,N);
        /* controlla esito read ..... elabora il msg */
        close(pfd[0]); /* chiude lettura */
    } .....
```

Pipe senza nome: esempio (2)

```
int main (void){
    int pfd[2], pid, l;
    char msg[N];
    .....
    else { /* siamo nel figlio */
        close(pfd[0]); /* chiude lettura */
        /* ... .. prepara il msg */
        l = write(pfd[1],msg,N);
        /* ... .. controlla esito */
        close(pfd[1]); /* chiude scrittura */
    }
    return 0;
}
```

Es: msg di lunghezza variabile

- Possibile protocollo:
 - ogni messaggio logico è implementato da due messaggi fisici sul canale
 - il primo messaggio (di lunghezza nota), specifica la lunghezza **lung**
 - il secondo messaggio (di lunghezza **lung**) codifica il messaggio vero e proprio
- vediamo un frammento di possibile implementazione

Msg di lunghezza variabile (2)

```
} else { /* siamo nel figlio */
    int lung; /* per la lunghezza*/
    close(pfd[0]); /* chiude lettura */
    snprintf(msg,N,"Da %d: Hi!!\n",getpid());
    lung = strlen(msg) + 1;
    /* primo messaggio */
    k = write(pfd[1], &lung, sizeof(int));
    /* esito ... secondo messaggio */
    k = write(pfd[1],msg,lung);
    /* esito ... */
    close(pfd[1]);
}
/* NB: non funziona con più di 1 scrittore*/
```

Msg di lunghezza variabile (3)

```
/* manca totalmente gest errori ed EOF read */
if ( pid ) { /* siamo nel padre */
    int l, lung;
    char* pmsg;
    close( pfd[1] );
    l=read( pfd[0], &lung, sizeof( int ) );
    pmsg = malloc( lung );
    l=read( pfd[0], &pmsg, lung );
    printf( "%s", pmsg );
    free( pmsg );
    close( pfd[0] );
}
/* NB: non funziona con più di 1 scrittore*/
```


Pipelining e ridirezione

- più comandi possono essere combinati assieme dalla shell usando il pipelining
- il pipelining viene implementato attraverso le pipe
- la ‘connessione’ fra stdout (1) dello stadio I e stdin (0) dello stadio I+1 viene implementata usando le pipe e sfruttando due chiamate di sistema per la duplicazione dei descrittori: **dup** e **dup2**
- le stesse SC permettono di implementare anche la ridirezione

Duplicazione di fd: dup, dup2()

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

```
int dup(
```

```
    int fd /*descr di file da duplicare*/
```

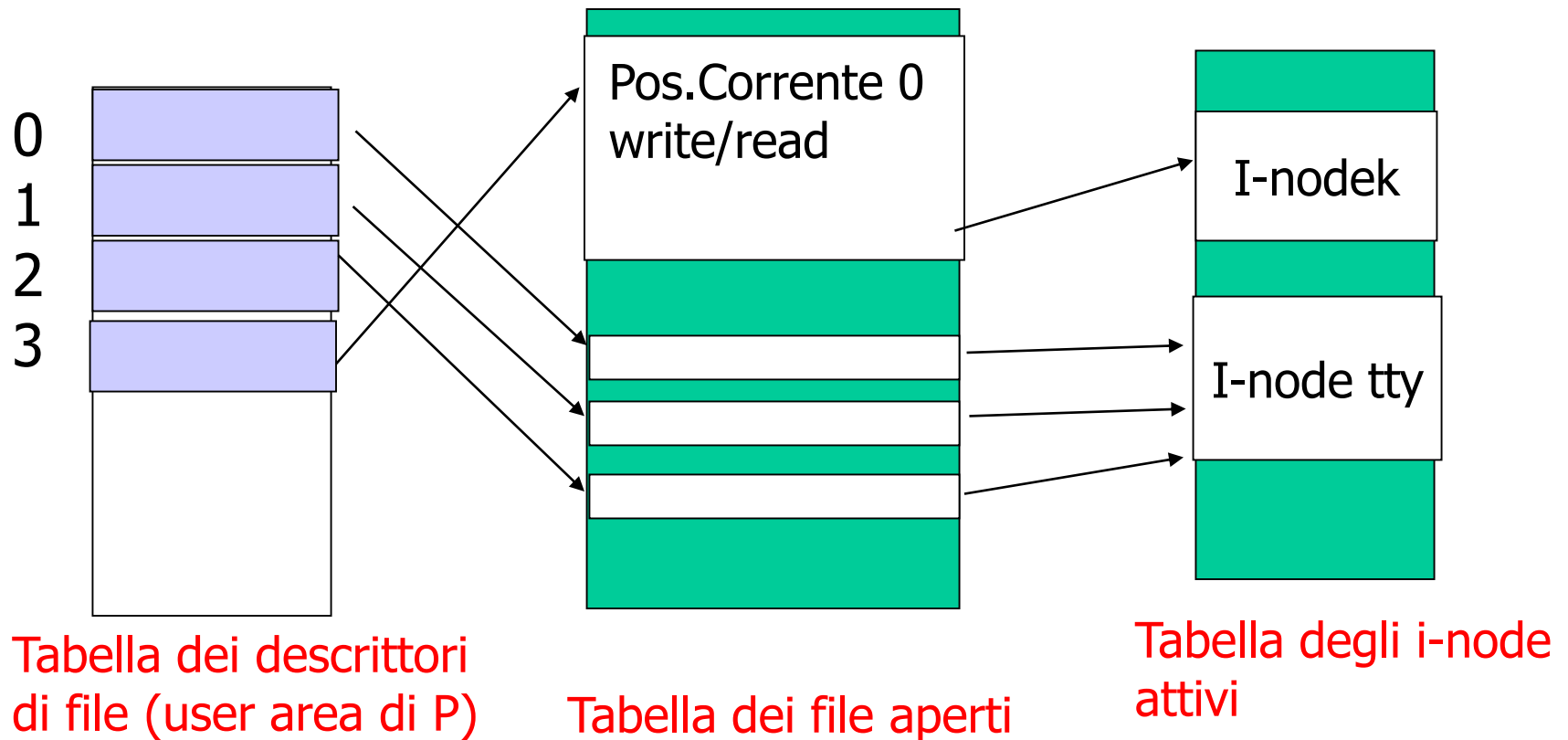
```
);
```

```
/* (fd1) success (-1) error, sets errno */
```

- duplica il descrittore **fd** e ritorna la posizione (**fd1**) della tabella dei descrittori in cui ha scritto la copia,
- **fd1** è la prima posizione libera nella tabella dei descrittori

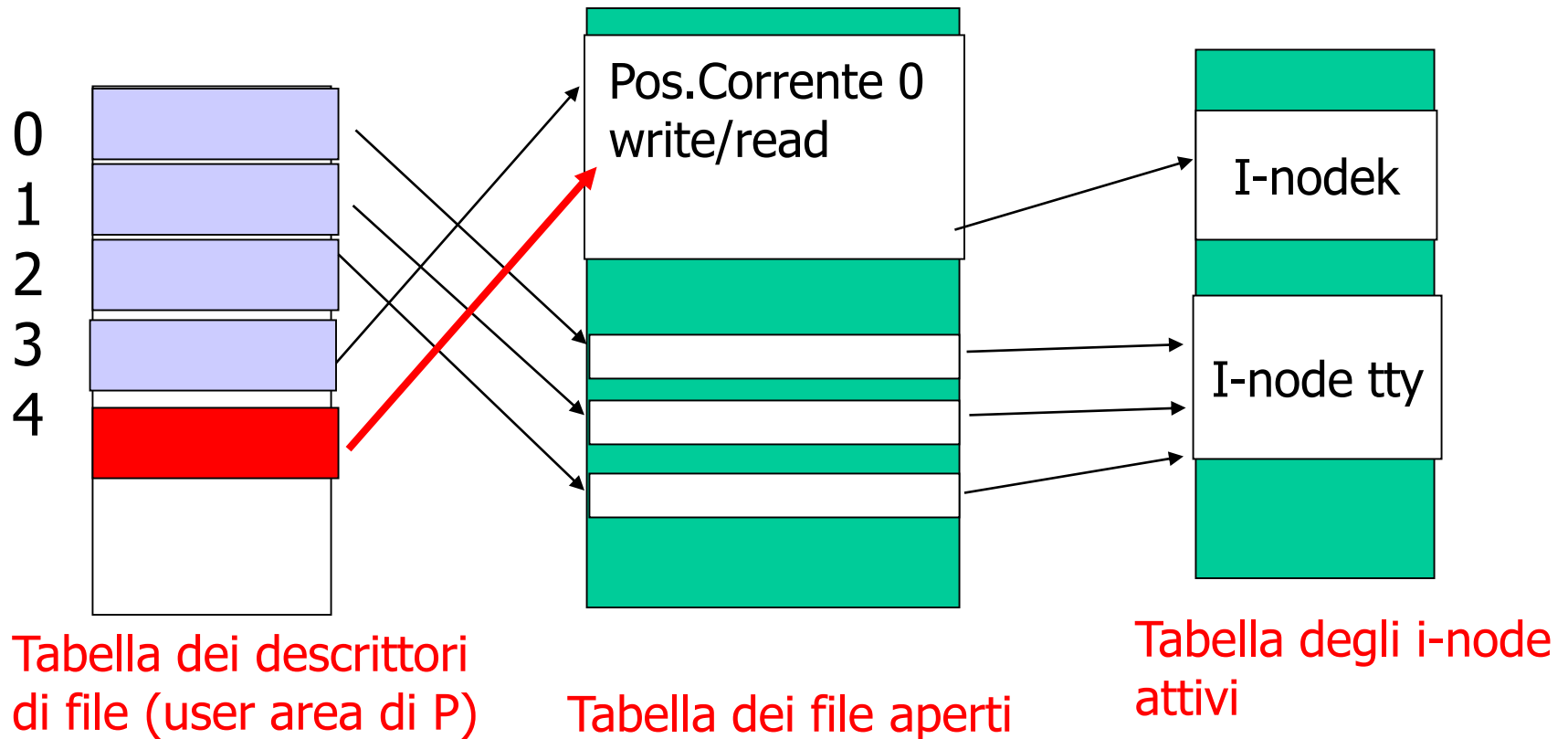
Situazione tabella descrittori

- Prima di invocare la **dup (3)**



Situazione tabella descrittori (2)

- Dopo la invocazione di **dup (3)** terminata con successo da parte del processo P. Il valore ritornato è **4**



Duplicazione di fd: dup, dup2 (2)

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

```
int dup2 (
```

```
    int fd, /*descr di file da duplicare*/
```

```
    int fd2, /*fd da usare per la copia*/
```

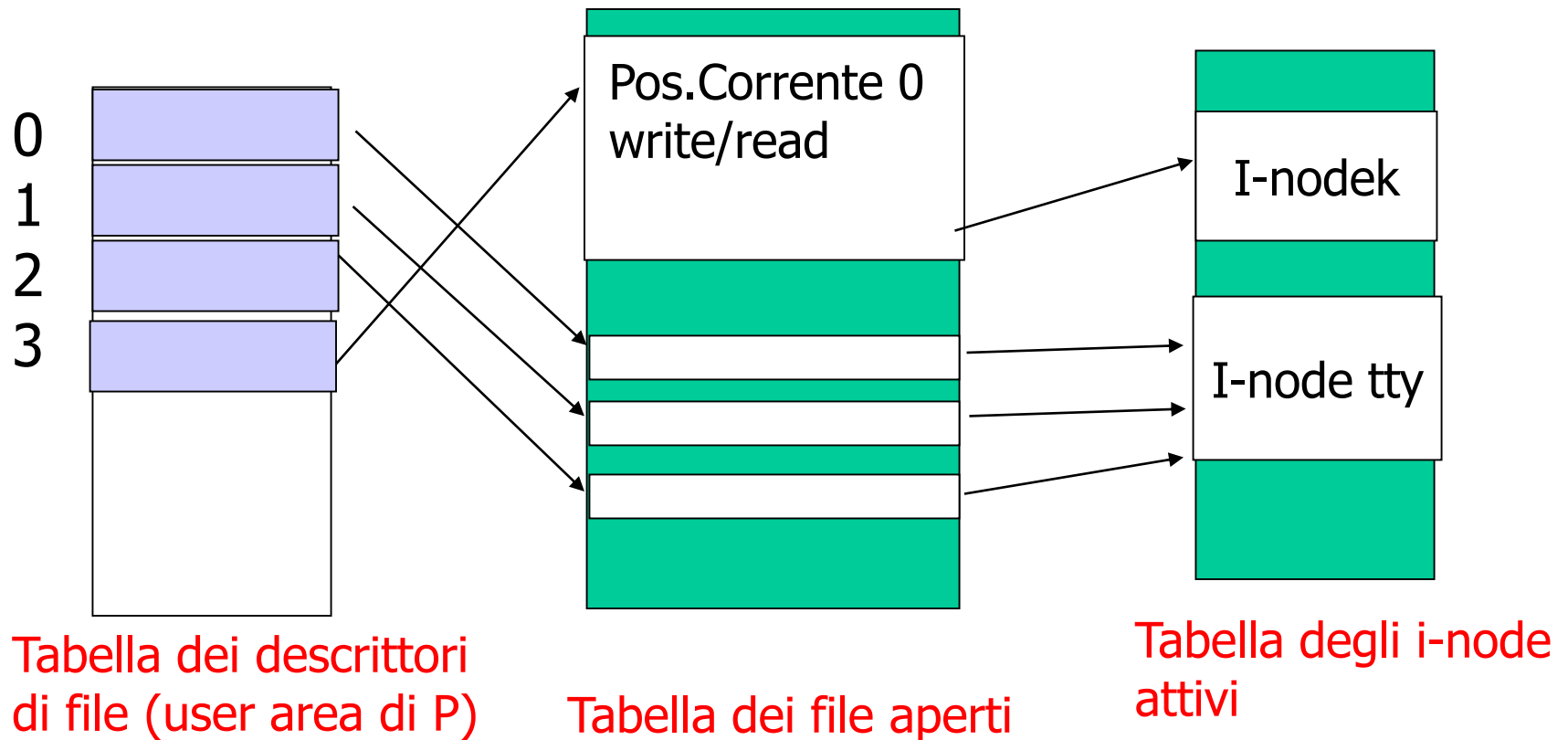
```
);
```

```
/* (fd2) success (-1) error, sets errno */
```

- duplica **fd** nella posizione specificata da **fd2**
- se **fd2** è in uso viene chiuso prima della copia
- la **dup2** è atomica, se qualcosa va male **fd2** non viene chiuso
- se **fd=fd2** la **dup2** non fa niente

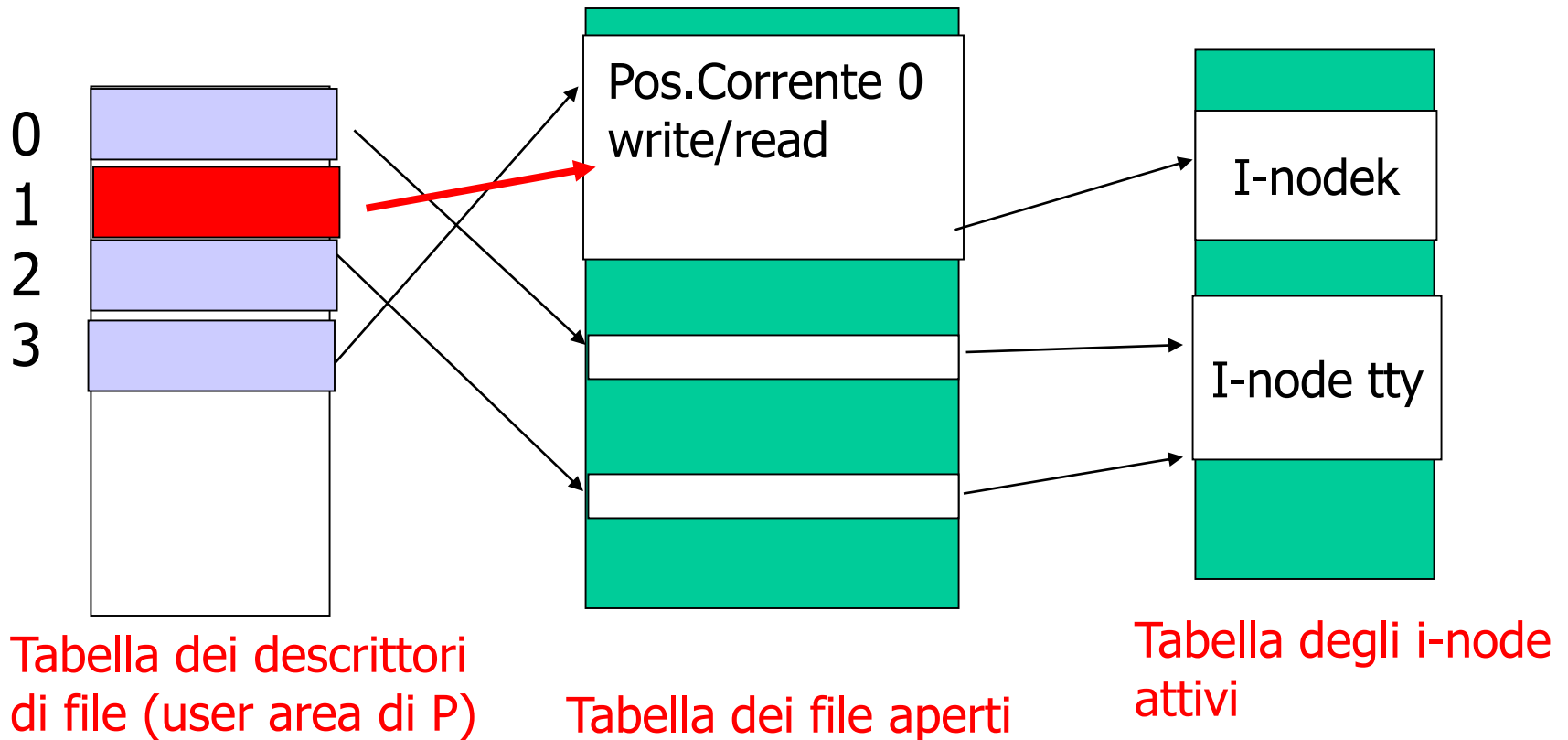
Situazione tabella descrittori

- Prima di invocare la **dup2 (3, 1)**



Situazione tabella descrittori (2)

- Dopo la invocazione di `dup2 (3, 1)` terminata con successo da parte del processo P. Il valore ritornato è **1**



Es: redirectione con dup() e dup2()

- Es. ridirigere lo standard output su un file **pippo**

```
int fd;
...
fd=open("pippo",O_WRONLY|O_TRUNC|O_CREAT,0666);
/* duplica fd su stdout */
if ( dup2(fd,1)== -1) {/* errore */ }
else {
    close(fd);           /* fd non serve piu' */
    printf("Questo viene scritto in pippo!");
}...
```


Ridirezione nella shell...

Es.

```
bash:$ ls -l > pippo
```

- Il processo shell si duplica con una `fork()` e si mette in attesa della terminazione del figlio
- Il figlio apre in scrittura il file `pippo` (creandolo o troncandolo)
- Il figlio duplica il descrittore di `pippo` con la `dup2` sullo `stdout` (fd 1) e chiude il descrittore originario
- Il figlio invoca una `exec` di `ls -l`, la quale conserva i descrittori dei file, e quindi va a scrivere in `pippo` ogni volta che usa il file descriptor 1

Ridirezione nella shell... (2)

Es. (segue)

```
bash:$ ls -l > pippo
```

- Quando il figlio termina il padre continua l'esecuzione con i suoi descrittori invariati
- Con un meccanismo simile la shell implementa il pipelining, vediamo un esempio:

Es: pipelining con dup() e dup2()

- Es. una funzione che esegue `who | wc -l`

```
void who_wc (void) {
    int pfd[2];
    pid_t pid1, pid2;
    if (pipe(pfd) == -1) { /* creazione pipe */ }
    switch ( pid1=fork()) {
    case -1: { /* errore */ }
    case 0 : /* figliol1: eseguirà who */
        if ( dup2(pfd[1], 1) == -1) { /* errore */ }
        else { close(pfd[0]);
                close(pfd[1]);
                execlp("who", "who", (char*) NULL);
        } } /* fine creazione figlio 'who' */
}
```

Es: pipelining con dup() e dup2() (2)

```
/* creazione figlio2*/
switch ( pid2=fork() ) {
    case -1: { /* errore */ }
    case 0 : /* figlio2: eseguirà ' wc -l */
        if ( dup2(pfd[0],0)== -1) { /* errore */ }
        else { close(pfd[0]);
                close(pfd[1]);
                execlp("wc", "wc", "-l", (char*) NULL);
        } /* ancora il padre*/
    close(pfd[0]); close(pfd[1]);
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0); }
```

Pipe con nome o FIFO

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(
    const char * path, /*path fifo*/
    mode_t perms, /*permission*/
);
/* (0) success (-1) error, sets errno */
```

- crea una pipe con nome **path** con diritti **perms**, interpretati esattamente come in **open** (usa **umask**)
- il nome è visibile ai processi sulla stessa macchina e quindi anche processi scorrelati possono aprirla in lettura e scrittura e comunicare

Creazione di pipe con nome

```
/* frammento che crea un nuovo pipe con nome  
umask 022*/  
  
...  
  
/* non fallisce se la pipe c'è già */  
if ((mkfifo("serverreq", 0664) == -1)  
    && errno != EEXIST) {/* gest errore*/}  
  
/* diritti di scrittura per il gruppo */  
if (chmod("serverreq", 0664) == -1)  
    {/* gestione errore*/}  
  
... ..  
  
/* uso pipe */
```

Uso di una pipe con nome

- Prima di iniziare a scrivere/leggere la pipe deve essere aperta contemporaneamente da un lettore e da uno scrittore:
- Apertura da parte di un processo scrittore :
 - usa le **open ()**
 - se nessun lettore ha ancora invocato la **open ()** si blocca in attesa del primo lettore
 - usando le opzioni **O_NONBLOCK**, **O_NDELAY** se non ci sono lettori la open termina subito con fallimento

Uso di una pipe con nome (2)

- Apertura da parte di un processo lettore :
 - usa le `open ()`
 - se nessun scrittore ha ancora invocato la `open ()` si blocca in attesa dello scrittore
 - usando le opzioni `O_NONBLOCK`, `O_NDELAY` se non ci sono scrittori la `open` termina subito con successo

Uso di una pipe con nome (3)

• Tipico uso : più client e un server

- il server crea la pipe e la apre in lettura e scrittura
 - perché non solo in lettura? Perché altrimenti alla chiusura dell'ultimo processo client scrittore il server legge un *end_of_file*! Il server logicamente deve rimanere sempre attivo, e mantenere aperto un descrittore di scrittura anche nel server rende la read sempre bloccante anche se non ci sono client connessi
- i client aprono in scrittura la stessa pipe
- le risposte del server in genere sono fornite su pipe private dei client (una per ogni client)
 - per evitare di essere terminato da una write il server deve ignorare SIGPIPE

Frammento di server

```
/* creazione pipe server */
if ((mkfifo("serverreq", 0664) == -1)
    && errno!=EEXIST) {/* gest errore*/}
if (chmod("serverreq", 0664) == -1) {/* gest
errore*/}

/* apertura in lettura e scrittura */
if (pfd=open("serverreq", O_RDWR) == -1)
    {/* gest errore*/ }

/* uso pipe: la read è bloccante anche senza
client*/
while (true) {
    if ( (l=read(pfd,buf,N)) == -1) ... ..
}
}
```