Primitive C

Importa `#include <unistd>`

Importa `#include <fcntl.h>`

- `create()`

  ### `create()`

  ```
  int fd = creat(name, mode)
          char *name; //nome file
          int mode;   //attributi file
          int fd;     //file descriptor *chiave*
  //mode espresso in ottale come in chmod 110-001-001 = 611
  ```

  Crea un file di nome `name` e diritti `mode` , se il file esiste lo azzera, si deve avere diritto di scrittura.

- `open()`

  ### `open()`

  ```
  #include <fcntl.h>
  int fd = open(name, flag);
  O_RDONLY, O_WRONLY, O_RDWR //solo lettura, solo scrittura, entrambe
  ```

  Apre il file di nome `name` con modalità `flag`

  Se le primitive hanno successo ritornano un FILE DESCRIPTOR se no `-1`

  Nella `open()` sono presenti le seguenti opzioni:

  - `O_APPEND` aggiunge in fondo al file, si usa con `O_WRONLY` o `O_RDWR` .
  - `O_TRUNC` distrugge il contenuto di un file esistente
  - `O_CREAT` crea un file se non esiste, bisogna specificare il mode
  - `O_EXCL` fallisce se il file esiste, si usa con `O_CREAT`

  Esempi:

  ```
  #include <fcntl.h>

  int fd1 = open("pippo", O_CREAT, 0644);
  int fd2 = open("pippo", O_CREAT | O_EXCL, 0644);
  int fd3 = open("pippo", O_TRUNC);
  int fd4 = open("pippo", O_WRONLY | O_APPEND);
  ```

- `close()`

  ### `close()`

```
int retval = close(int fd);
        int retval;
```

Chiude il file `fd` al termine del processo

- `read()`

`read()`

```
int nread = read(fd, buffer, n);
char* buffer //è un'array di byte
int fd, n
```

Parte dalla posizione corrente e restituisce il nr di byte su cui ha lavorato, esempio: se il FP è su EOF read() restituisce 0.

Tenta di leggere `n` byte e i caratteri vengono inseriti in `buffer` e `nread` sono i byte effettivamente letti.

- `write()`

`write()`

```
int nwrite = write(fd, buffer, n);
char* buffer //è un'array di byte
int fd, n
```

Parte dalla posizione corrente e restituisce il nr di byte su cui ha lavorato. La write tenta di scrivere `n` byte presi da `buffer` e restituisce il numero di byte effettivamente scritti `nwrite` , se il tutto non è andato a buon fine `n != nwrite`

la dimensione di `buffer` deve essere ≥ di `n` .

- `lseek()`

`lseek()`

```
long int newpos = lseek(fd, offset, origin);
long int newpos, offset;
int fd, origin;
```

sposta il FP all'interno del file `fd` di `offset` byte a partire da `origin` → in `<unistd.h>` si possono usare `SEEK_SET` (inizio), `SEEK_CUR` (corrente), `SEEK_END` (fine).

`offset` può essere sia positivo che negativo, il valore di ritorno rappresenta la posizione corrente del FP.

ad esempio per tornare all'inizio:

```
long int inizio = lseek(fd, 0L, 0);
```

se ci spostiamo alla fine `newpos` rappresenta la lunghezza del file.

- `link()`

  ### `link()`

  ```
  int retval = link(name1, name2);
  char *name1, name2;
  ```

  Consente di creare un nuovo nome `name1` → link Hardware per un file esistente `name2` → viene incrementato il numero di link.

- `unlink()`

  ### `unlink()`

  ```
  int retval = unlink(name);
  char *name;
  ```

  Consente di cancellare un file `name`, ovvero cancella un link Hardware in UNIX, se il numero di link arriva a 0 allora si opera la distruzione del file e si libera spazio su disco.

- `access()`

  ### `access()`

  ```
  int retval = access(pathname, amode);
  char* pathname;
  int amode;
  ```

  Consente di verificare il tipo di accesso `amode` consentito su un file `pathname` .

  `amode` può essere:

  - `04 -> read access`
  - `02 -> write access`
  - `01 -> execute access`
  - `00 -> existence`
  - `06 -> read & write access`

  Se ha avuto esito positivo ritorna 0 se no un valore negativo.

- `stat()` & `fstat()`

  ### `stat()` & `fstat()`

  ```
  int retval = stat(pathname, &buff);
  char* pathname;
  struct stat buff; //struttura che rappresenta descrittore file
  ```

```
int retval = fstat(fd, &buff);
int fd;
```

Verificano lo stato di un file `pathname` / `fd`

`fstat()` si può usare solo se un file è già stato aperto.

Tornano 0 in caso di successo se no un valore negativo.

Campi `struct stat`:

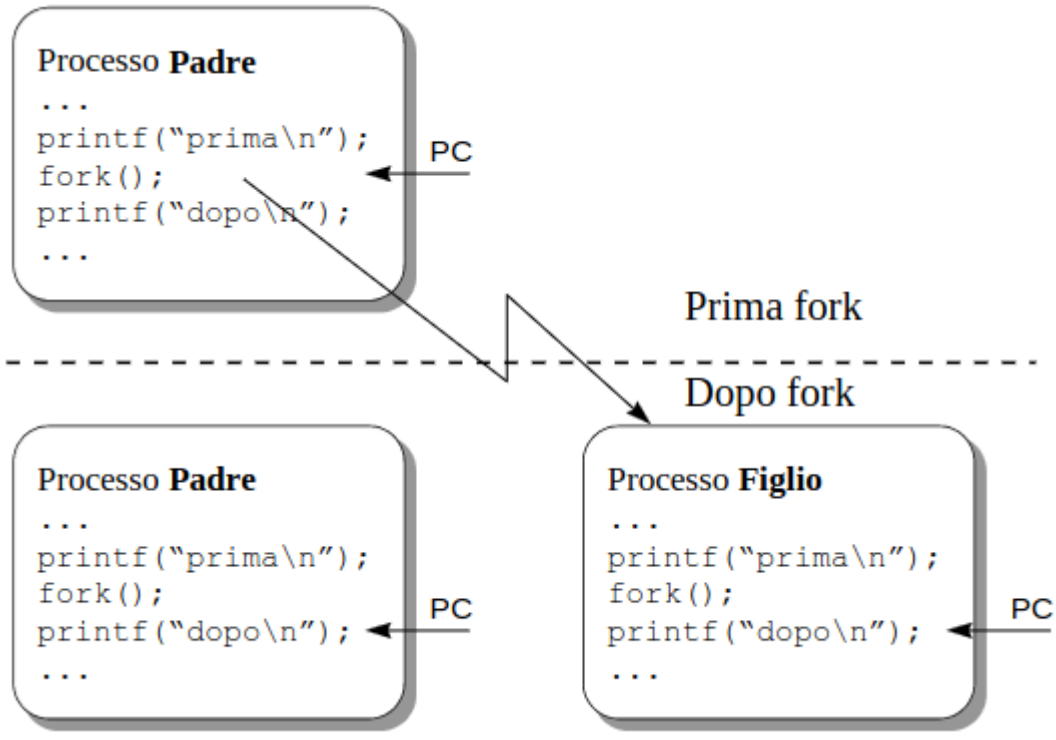
```
struct stat  {  ushort  st_mode;       /* modo del file */
    ino_t   st_ino;         /* I_node  number */
    dev_t   st_dev;         /* ID del dispositivo */
    dev_t   st_rdev;        /* solo per file speciali */
    short   st_nlink;       /* numero di link */
    ushort  st_uid;         /* User ID del proprietario */
    ushort  st_gid;         /* Group ID del proprietario */
    off_t   st_size;        /* Lunghezza del file in byte */
    time_t  st_atime;       /* tempo dell'ultimo accesso */
    time_t  st_mtime;       /* tempo dell'ultima modifica*/
    time_t  st_ctime; /* tempo dell'ultimo cambiamento di stato */}
```

- `fork()`

`fork()`

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

Un processo ne genera un'altro, dopo si hanno 2 processi concorrenti e separati: il parent (padre) e il child (figlio).



Effetti del `fork()`:

Se si verifica qualsiasi errore la `fork()` restituisce -1 se no viene creato un nuovo processo figlio.

1. Si inserisce un nuovo elemento nella TP, inserito il descrittore con attributi ereditati dal padre → stato processo child IDLE

2. Il nuovo processo esegue lo stesso codice del padre → aggiornamento contatore TT.
3. Viene creato un area dati utente per il nuovo processo come copia di quella del padre.
4. Viene creato una kernel area per il nuovo processo come copia di quella del padre.
5. Il descrittore del figlio viene inserito nella coda dei processi pronti → processo figlio in stato READY

Se la `fork()` ha avuto successo restituisce un valore differente ai 2 processi padre e figlio:

- CHILD → 0
- PARENT → ≠ 0 → restituisce il PID creato

Questo consente ai 2 processi di eseguire azioni diverse sullo stesso codice

es:

```
...
if (fork() == 0) {
        ... //figlio
        exit(value); //il valore verrà ritornato al padre
}
//padre
exit(0);
```

- `getpid()`

  ### `getpid()`

  ```
  int pid = getpid();
  ```

  Usata per conoscere il PID del processo corrente

- `getppid()`

  ### `getppid()`

  ```
  int ppid = getppid();
  ```

  Usata per conoscere il PID del padre

- `getuid()`

  ### `getuid()`

  ```
  int uid = getuid();
  ```

  Usata per conoscere lo UID (User ID) del processo corrente.

  Mediante la versione `geteuid()` si può conoscere lo UID effettivo.

- `getgid()`

`getgid()`

```
int gid = getgid();
```

Usata per conoscere il GID (Group ID) del processo corrente.

Mediante la versione `getegid()` si può conoscere il GID effettivo.

- `wait()`

`wait()`

```
int pid = wait(&status);
int status;
```

Dopo la creazione del processo figlio il padre può decidere se operare contemporaneamente o attendere usando `wait()`.

Normalmente nel valore di ritorno `status` (a 16 bit):

- nel byte alto il valore restituito da un figlio con la exit
- nel byte basso 0

Invece, in caso di terminazione di un figlio in seguito alla ricezione di un segnale si ha:

- nel byte alto 0
- nel byte basso il numero di segnale che ha provocato la terminazione del figlio.

Ritorna -1 in caso il processo invocante non abbia figli altrimenti il PID del figlio terminato.

Effetto della `wait()`:

Sospende un processo padre in attesa della risposta del figlio:

- sospensiva se ci sono processi figli attivi
- non sospensiva se tutti i processi figli sono terminati

Se si vuole aspettare un processo figlio con un certo PID:

```
while (pid1 = wait(&status) != pid2){ ... }
```

Vengono definite anche le seguenti macro:

- `wifexited(status)`
- `wexitstatus(status)`
- `wifsignaled(status)`
- `wtermsig(status)`

- `exit()`

```

```

```
void exit(status);
int status;
```

La exit chiude i file aperti, per il processo che termina, il valore di status viene passato al padre quando questo invoca la wait. Per convenzione 0 terminazione normale ≠ 0 rappresenta un problema.

- `exec()`

Famiglia di `exec()`

Di base eseguono un nuovo programma all'interno di un processo esistente.

```
execv(pathname, argv);
execl(pathname, arg0, argv1, ..., argvn, (char *) 0);
execvp(name, argv);
execlp(name, arg0, argv1, ..., argvn, (char *) 0);
char *pathname, *name, *argv[], *arg0, ..., *argn;
```

`execv()`: sostituisce programma corrente con uno nuovo specificato.

`execl()`: uguale a `execv()` ma prende gli argomenti come un array di stringhe.

`execvp()`: cerca il programma nei percorsi definiti in `PATH`

`eseclp()`: uguale a `execvp()` ma prende gli aromenti come un array di stringhe.

I File Descriptor FD rimangono tali anche dopo la exec e vengono ereditati infatti non è necessario riaprire il file, i segnali vengono alterati poichè l'identificatore effettivo del processo viene cambiato in quello del proprietario del file corrispondente al programma eseguito.

Si ereditano: directrory corrente, file aperti, maschera dei segnali, terminale di controllo e altre caratteristiche.

RIASSUMENDO:
execl, execle, execlp, execv, execve, execvp
l ➔ la funzione riceve una lista di argomenti (terminata da NULL);
v ➔ la funzione riceve un vettore di argomenti argv[];
p ➔ la funzione prende un nome di file come argomento e lo cerca nei direttori specificati in PATH;
e ➔ la funzione riceve anche un vettore envp[] che rimpiazza l'environment corrente