Sistemi Operativi
Teledidattico

Anno 2002

3a esercitazione

5/10/2002

Paolo Torroni

processi

`fork` (creazione di un figlio)

`exec` (sostituzione di codice del programma in esecuzione)

`wait` (attesa della terminazione di un figlio)

`kill -N` (*da shell*; invio di un segnale)

Esercizio: `fork`, `execl`, `wait`

Si realizzi un programma C che, utilizzando le primitive del Sistema Operativo Unix, abbia un'interfaccia del tipo:

`scrivi stringa comando`

Il programma ***`scrivi`*** prevede due argomenti, ***`stringa`*** e ***`comando`***, il primo rappresenta una qualunque stringa di testo, il secondo un comando per unix (path completo!!). Il programma deve funzionare nel modo seguente:

- il processo iniziale (padre) crea un processo figlio
- il processo figlio, una volta avviato, esegue il comando passato come 2° parametro
- il padre attende la terminazione del figlio, quindi stampa su video la stringa passata come 1° parametro (e finalmente termina).

Si verifichi l'esecuzione del programma, e si utilizzino i comando ***`ps`*** per monitorare i processi in esecuzione, ed eventualmente ***`kill –9`*** per forzare la terminazione di processi..

(1) argomenti del main

- uso di argc, argv:
- Il programma ***scrivi*** prevede due argomenti, ***stringa*** e ***comando***, il primo rappresenta una qualunque stringa di testo, il secondo un comando per unix (path completo!!).
- quanto vale argc? che tipo di dato è argv?
- In C:

```
char *stringa, *comando;
main(int argc, char*argv[]) {
  if( argc!=3 ) exit( 0 );
  stringa = argv[1];
  comando = argv[2];
}
```

(2) il processo padre crea un processo figlio

- uso della fork (**`pid=fork()`**)
- che tipo di dato è pid? quanto vale pid?
- In C:

```
main(int argc, char*argv[]) {
   int pid;
   ...
   pid=fork();
   if( pid==0 ) { // chi esegue? (A)
   }
   else { // chi esegue? (B)
   }
   // chi esegue? (C)
}
```

(3) il processo figlio esegue il comando passato come parametro

- uso della execl (**`execl( com,arg[0],…,NULL )`**)
- quali/quanti sono gli argomenti della execl? cosa succede dopo la execl?
- In C:

```
main(int argc, char*argv[]) {
  int pid;
  ...
  // codice figlio
  execl( comando, comando, NULL );
  perror( “C’e’ stato un errore nella execl” );
  }
```

(4) il processo padre attende la terminazione del figlio poi stampa la stringa (2º param.)

- uso della wait (**`pid_term=wait( &status )`**)
- che cosa sono pid_term e status? a cosa servono?
- In C:

```
main(int argc, char*argv[]) {
  int pid, pid_term, status;
  ...  pid=fork(); ...
  // codice padre
  pid_term=wait( &status ); // quanto vale status?
  // pid_term==pid?
  printf( stringa );
  exit( 0 );
  }
```

file (sequenze di byte)

`creat` (creazione di un nuovo file)

`open` (apertura di un file [esistente])

`read` (lettura di una sequenza di byte)

`write` (scrittura di una sequenza di byte)

`close` (chiusura di un file aperto)

* **`lseek`** (per spostare l'IO-pointer)

Esercizio: lettura/scrittura

Scrivere un programma C che abbia un'intefaccia del tipo:

`copia nomefile1 nomefile2`

Il primo parametro indica un file esistente, il secondo no.

Dopo il controllo dei parametri (sull'esistenza dei file), il processo iniziale P0 deve creare un processo figlio P1.

Successivamente, P1 deve leggere il contenuto del file *nomefile1* e scriverlo in *nomefile2*. Quando ha terminato, il processo padre deve visualizzare il contenuto di *nomefile2*.

(1) controllo dei parametri

- esistenza di file:
- ***nomefile1*** (primo parametro) deve rappresentare il nome di un file esistente, mentre ***nomefile2*** (secondo parametro) deve rappresentare il nome di un file che non esiste
- utilizzo della ***open***: che cosa restituisce in uscita?
- In C:

```
int fd1,fd2;
fd1=open( argv[1], O_RDONLY );
if( fd1<0 ) exit( -1 );
fd2=open( argv[2], O_WRONLY ); // quali sono i file aperti?
if( fd2>0 ) exit( -2 );
fd2=creat( argv[2], 0644 ); // quali sono i file aperti?
```

(2) copia da fd1 a fd2

- uso delle read/write (**`read( FD, buffer, n )`**)
- quali diritti devo avere per la read (write)?
- quanti caratteri leggo per volta?
- cosa succede quando arrivo alla fine del file?
- In C:

```
... // codice del FIGLIO
char buffer[10];
int n;
do {
 n=read( fd1, buffer, 10 );
 write( fd2, buffer, n ); // perché n e non 10?
} while( n==10 )
// a che punto sta il file pointer?
 // se il file è stato aperto prima della fork, a che
 // punto sta il file pointer del figlio?
```

(3) uso degli standard input/output

- scrittura su video: **`write( 1, buffer, n )`**
- lettura da terminale: **`read( 0, buffer, n )`**
- CTRL+D serve per indicare *fine-file* (EOF) da terminale
- In C:

```
char buffer[10]; // codice PADRE
  int n;
  do {
   n=read( fd2, buffer, 10 );
   write( 1, buffer, n );
  } while( n==10 )
```

pipe (canali di comunicazione)

`pipe` (creazione di una pipe)

* **`dup`** (duplicazione di un file-pointer)

Esercizio: comunicazione

Scrivere un programma C che abbia un'intefaccia del tipo:

`FILTRO nomefile car`

Il primo parametro indica un file esistente, il secondo un carattere.

Dopo il controllo dei parametri (sull’esistenza dei file), il processo iniziale P0 deve creare un processo figlio P1.

Successivamente, P1 deve leggere il contenuto del file, carattere per carattere, e passare a P0 solo i caratteri diversi da ***`car`***. Il padre termina quando il figlio ha finito.

(1) creazione della pipe

- la pipe va creata PRIMA della fork:
- che tipo di dato restituisce? quante ‘entry’ usa, nella tabella dei file aperti?
- In C:

```
int p[2], pid;
pipe( p );
pid=fork();
if( pid==0 ) { // figlio: deve scrivere su pipe
  close( p[0] ); // chiudo il lato di lettura
  ...
}
else { //padre: deve leggere da pipe
  close( p[1] ); // chiudo il lato di scrittura
  ...
}
```

(2) lettura da file, scrittura su pipe

- uso delle read/write (**`read( FD, buffer, n )`**)
- quando finisco di scrivere?
- come faccio a essere sicuro di non andare a sovrascrivere dei caratteri non ancora letti (sulla pipe)?
- In C:

```
... // codice del FIGLIO
char c, car=argv[2][0];
while( read( fd, &c, 1 ) ) { // finché ho da leggere
 if( c!=car )
  write( p[1], &c, 1 );
} // se c non fosse un carattere ma un intero?
```

(3) lettura da pipe, scrittura su video

- uso delle read/write (**`read( FD, buffer, n )`**)
- come faccio a essere sicuro che non leggo un carattere prima che sia stato scritto?
- come faccio a capire che non c'è più niente da leggere?
- che cosa restituisce la read da pipe? è 'bloccante'?
- In C:

```
... // codice del PADRE
char c;
while( read( p[0], &c, 1 ) ) { // finché ho da
leggere
  write( 1, &c, 1 );
}
 // come faccio a uscire da questo ciclo?
```

Da fare in laboratorio (oggi)

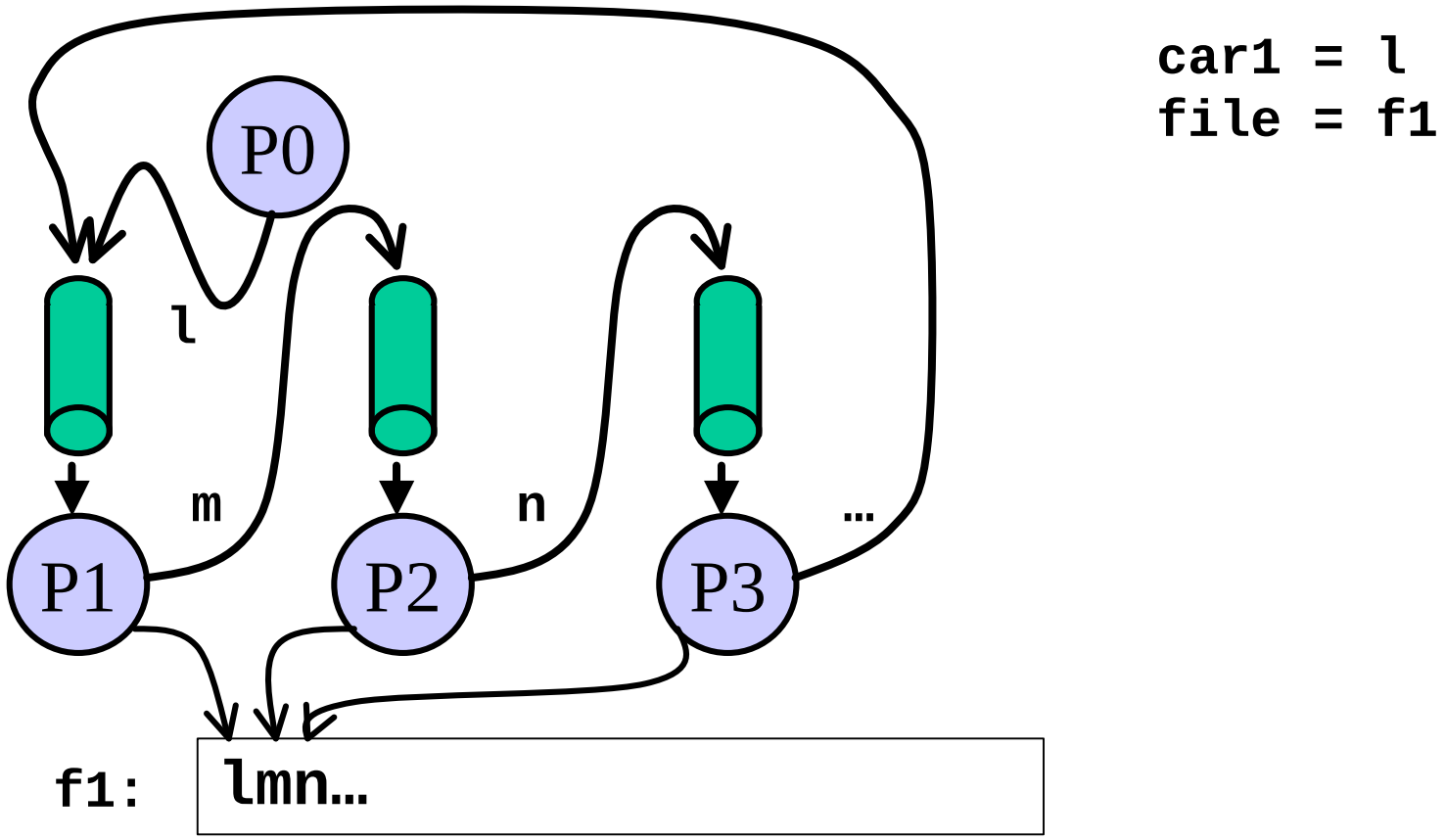
- Svolgere i 3 esercizi visti in aula
- Ulteriore esercizio (da compito):

sintassi: **`esame c1 c2 file`** [**`c1`**, **`c2`**: caratteri alfabetici; **`file`**: nome di file non esistente]

1. controllo parametri:
   - **`c1`** e **`c2`** devono essere caratteri alfabetici, da '**`a`**' a '**`z`**',
   - **`c1`** deve venire prima di **`c2`**;
   - **`file`** deve **non** essere il nome di un file già esistente
2. 4 processi: 1 padre (P0) e 3 figli (P1,…, P3);
3. il padre (P0) comunica il carattere **`c1`** a P1, poi si sospende;
4. P1 scrive in carattere ricevuto su **`file`** e comunica P2 il carattere successivo (in ordine alfabetico), P2 fa lo stesso con P3, P3 con P1, e così via
5. quando un figlio riceve il carattere **`c2`**, tutti i processi devono terminare.

schema di soluzione

- 3 pipe (una da cui legge P1, una da cui legge P2, etc.)
- Lo stesso I/O-pointer condiviso da tutti i figli




```
car1 = l
file = f1
```