La Programmazione di Rete e di Sistema (iii)

A cura di:
Luca Breveglieri[*] Giacomo Buonanno[#]
Roberto Negrini[*] Giuseppe Pozzi[*] Donatella Sciuto[*]

[*] DEI, PoliMI, Milano
[#] LIUC, Castellanza (VA)

breveglieri,negrini,pozzi,sciuto@elet.polimi.it
buonanno@liuc.it
- versione del 10 marzo 2003 -

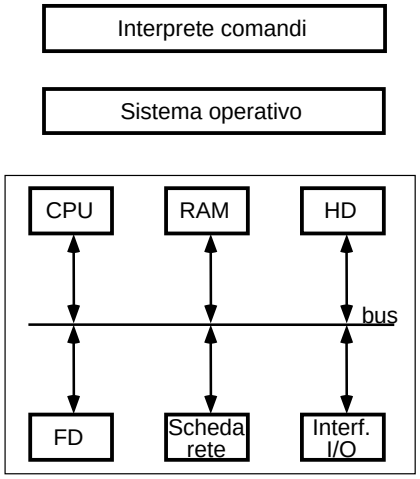
La programmazione di sistema

Introduzione al sistema operativo
ed il sistema operativo multiprocesso

Il sistema operativo

- Il sistema operativo:
  - è un insieme di moduli software;
  - controlla le risorse del sistema;
  - interagisce direttamente con i software che controllano l'hardware (*device driver*);
  - mette a disposizione dell'utente una macchina "virtuale", in grado di eseguire comandi dati dall'utente, utilizzando una macchina "reale", di livello inferiore e meno potente.

Il sistema operativo



Il processo dal punto di vista del sistema operativo

- Il processo è:
  - un'attività sequenziale (come ad es. l'istanza di un programma in esecuzione[1]),
  - con uno stato associato. Lo stato a sua volta si divide in:
    - stato interno;
    - stato esterno.

[1] il processo è anche definito come l'esecutore di un programma, creato dinamicamente.

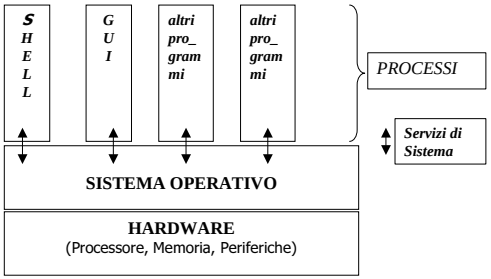
Lo stato di un processo

- Lo stato interno indica:
  - la prossima istruzione del programma che deve essere eseguita;
  - i valori delle variabili e dei registri utilizzati dal processo.
- Lo stato esterno indica se il processo è:
  - in attesa di un evento, come ad es. la lettura da disco o l'inserimento di dati da tastiera;
  - in esecuzione;
  - pronto per l'esecuzione, e quindi attende di accedere all'utilizzo della CPU.

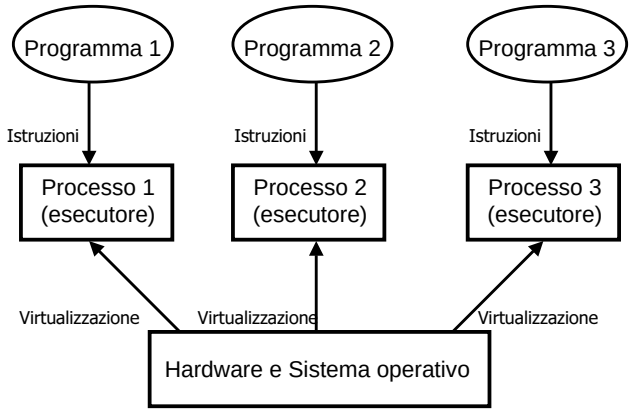
Il sistema operativo e le macchine virtuali

- Il sistema operativo:
  - controlla la macchina multiprogrammata, cioè che esegue più processi contemporaneamente, rendendo visibile ad ogni processo una macchina "virtuale" ad esso interamente dedicata e quindi con risorse proprie;

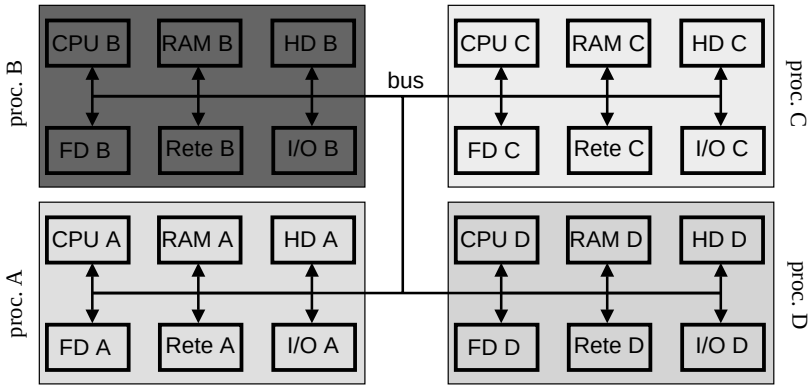
Il sistema operativo



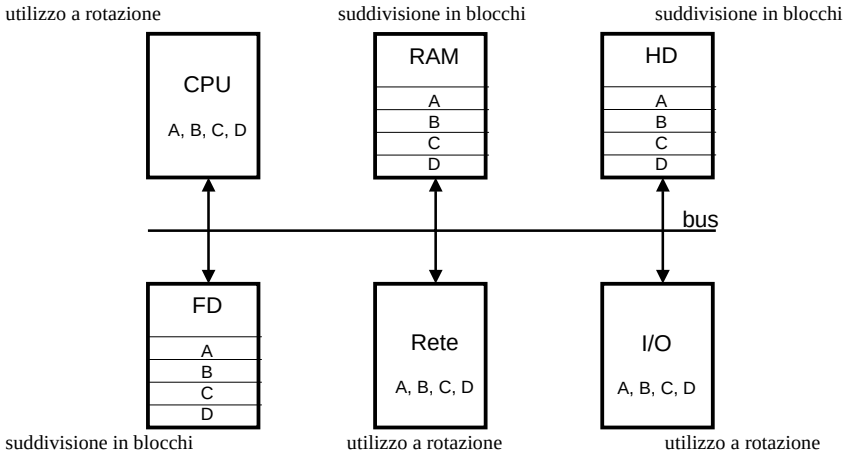
Il sistema operativo e le macchine virtuali



Il sistema operativo e le macchine virtuali



Il sistema operativo e la macchina reale



Il sistema operativo

- Il sistema operativo:
  - gestisce eventuali conflitti di accesso contemporaneo alle risorse "reali" condivise, accodando i processi richiedenti.

    Ad es. un componente, detto *scheduler*, seleziona i processi in stato di pronto ed assegna loro a rotazione l'utilizzo della CPU per un quanto di tempo di esecuzione (politica detta *round robin*).

I processi ed il sistema operativo

- Anche il sistema operativo è implementato tramite processi;
- il sistema operativo è garante che i conflitti tra i processi siano controllati e gestiti correttamente;
- il sistema operativo viene eseguito in modalità privilegiata (*kernel mode* o *supervisor*), così da poter controllare gli altri processi eseguiti in modalità *user*.

Chiamate al *supervisor*

- I processi utente per eseguire operazioni privilegiate, come ad es.,
  - accesso a file;
  - accesso ad altre risorse;
  - operazioni di I/O diretto

invocano il *supervisor* tramite chiamate di sistema (*system calls*).

Motivazioni dell'uso del *supervisor*

- Le operazioni di I/O sono operazioni riservate:
  - un processo *A* non deve poter andare a scrivere messaggi su un terminale non associato allo stesso processo *A*;
  - un processo *A* non deve poter leggere caratteri immessi da un terminale non associato allo stesso processo *A*.

Motivazioni dell'uso del *supervisor*

- Un processo non deve poter sconfinare al di fuori del proprio spazio di memoria:
  - per non accedere allo spazio di memoria associato ad un altro processo, modificando codice e dati di quest'ultimo;
  - per non occupare tutta la memoria disponibile nel sistema, bloccandolo e rendendolo così inutilizzabile da altri processi.

Motivazioni dell'uso del *supervisor*

- La condivisione di risorse, quali ad esempio dischi, CPU, schede di rete ...
  - deve essere tale da cautelare i dati di ogni utente;
  - deve evitare che un utente occupi l'intero spazio disponibile sui dischi, non lasciando spazio per altri utenti;
  - se un processo *A* entra in loop, ciò non deve bloccare l'intero sistema;
  - … ed altri esempi ancora.

Motivazioni dell'uso del *supervisor*

- Essendo un sistema multiprogrammato e/o multiutente:
  - eventuali conflitti devono essere gestiti da un "arbitro" (il sistema operativo) che funzioni secondo regole chiare e ben stabilite.

Necessità di parallelismo nei sistemi operativi

- Un utente che si collega ad una macchina (ad es. Unix tramite il programma *telnet*) dispone di un interprete comandi, eventualmente scelto dall'utente stesso tramite *chsh*.
- Nella macchina deve essere presente un processo interprete comandi per ogni sessione.

Necessità di parallelismo nei sistemi operativi

- Ad ogni nuova connessione il sistema operativo genera un processo interprete comandi specifico per quella connessione.
- Ci sono quindi più copie dello stesso programma.

Parallelismo in Unix

- Esistono due principali metodi per ottenere l'esecuzione parallela in ambiente Unix:
  - generazione di nuovi processi;
  - utilizzo di thread.
- Considereremo la generazione di nuovi processi (meccanismo più standard e meno complesso).

La programmazione di sistema

Definizioni

Definizione

- Con programmazione di sistema si intendono le tecniche utilizzate nella scrittura di programmi utente (cioè non necessariamente eseguiti in modalità *supervisor*) che interagiscono strettamente con il sistema operativo e che utilizzano i servizi (*system calls*) messi a disposizione da quest'ultimo.

Le chiamate di sistema

- Le chiamate di sistema sono divise nelle seguenti principali categorie:
  - gestione dei processi;
  - segnalazioni;
  - gestione di file;
  - gestione di direttori e di file system;
  - protezione;
  - gestione di ora e data.

da: Tanenbaum A. S., Woodhull A. W., Operating Systems - Design and Implementation, 2nd ed., Prentice Hall, 1997

Programmazione di sistema

- All'interno della programmazione di sistema, la nostra attenzione si focalizzerà sulle primitive in linguaggio C per:
  - gestione degli accessi a file;
  - gestione di processi.

Operazioni sui file

- Le principali operazioni di accesso a file consentono di:
  - aprire un file;
  - leggere un carattere, una stringa, un blocco da un file;
  - scrivere un carattere, una stringa, un blocco su un file;
  - cancellare, cambiare nome, spostare di direttorio un file;
  - chiudere un file

Operazioni sui processi

- Le principali operazioni sui processi consentono di:
  - generare un processo figlio (*child* o anche *slave*) copia del processo padre (*parent* o anche *master*) in esecuzione;
  - attendere la terminazione di un processo *figlio*;
  - terminare un processo *figlio* restituendo un codice al processo *padre*;
  - sostituire il codice di un processo in esecuzione.

La programmazione di sistema

La gestione dei file

(a esercitazione)

Il file

- E' un contenitore di informazioni:
  - permanenti;
  - su memoria di massa;
  - cui si accede tramite funzioni della standard library.
- E' anche detto "long lived object".

Il file

- E' identificato da:
  - nome (univoco nella sua directory) ed estensione;
  - directory (e volume) dove è memorizzato;
  - lunghezza (numero di byte);
  - data e ora di creazione e di ultimo accesso;
  - proprietario ed utenti autorizzati all'accesso;
  - modalità di accesso:
    - lettura;
    - scrittura.

Visualizzazione delle informazioni associate ad un file

```
/home/pozzi/temp> ls -lpa
total 4
drwx------   2    pozzi       512  Sep  1   17:55 ./
drwxr-xr-x  20    pozzi      1024 Jun 28   18:14 ../
-rw-------   1    pozzi        25  Sep  1   17:55   TemaEsame.txt
-rwxrwx---  1    pozzi     25768  Sep  2   17:55   ExecuteMe
```


Flussi e file

- Un programma (ad es. in C) che desidera utilizzare un file deve aprire un flusso (*stream*) di comunicazione, invocando il sistema operativo.
- Il sistema operativo restituisce al programma il descrittore del file (*handle*) e memorizza sempre la posizione corrente nel file.
- Ogni operazione sul file avviene tramite *handle*. Al termine il programma chiude il file, rilasciando il descrittore.

Flussi e file

- Vi sono tre flussi standard associati ad ogni processo:
  - stdin: la tastiera per leggere i messaggi in ingresso (*scanf*);
  - stdout: il video per scrivere i messaggi in uscita (*printf*);
  - stderr: il video per scrivere i messaggi di errore (*printf*).
- A tali flussi si aggiungono eventuali file aperti dal programma.

Tipi di file

- Di testo (ASCII):
  - sono sequenze di caratteri generalmente suddivise in linee separate dal carattere `\n';
  - possono essere visualizzati a video (ad es. tramite il comando cat) o stampati (ad es. tramite il comando lpr).

Tipi di file

- Binari (binary):
  - sono sequenze di byte;
  - per essere letti è necessario conoscere la modalità secondo la quale le informazioni sono state scritte;
  - generalmente non possono essere letti e stampati direttamente (se non tramite opportuni programmi).

Operazioni di base nella gestione dei file

- Apertura: *fopen* apre un file:
  - memorizza sempre la posizione corrente nel file;
- Chiusura: *fclose* chiude un file;
- Cancellazione: *remove* per cancellare un file;
- Ridenominazione: *rename* modifica il nome di un file o spostarlo di direttorio.

Modalità base di accesso

- Le modalità base di accesso sono specificate durante la fase di apertura del file e sono:
  - Lettura:
    - r (read);
    - la posizione corrente è all'inizio del file.
  - Scrittura:
    - w (write);
    - la posizione corrente è all'inizio del file.
  - Aggiunta:
    - a (append);
    - la posizione corrente è alla fine del file.

Altre modalità di accesso

- Binario:
  - b (binary);
- Lettura e scrittura:
  - + (sia scrittura che lettura).

Esempi di modalità di accesso

- w: write in modalità testo;
- w+: write e read in modalità testo;
- rb: read in modalità binaria;
- ab+: aggiunta e lettura in modalità binaria.

fopen

- Apertura del file

```
FILE *fp;
const char *fname = "MioFile";
const char *mode = "w";

fp = fopen(fname, mode);
  /* fp == NULL se errore di apertura */
```

fclose

- Chiusura del file (che deve essere stato aperto precedentemente)

```
FILE *fp;
int result;

result = fclose(fp);
  /* result != 0 se errore di chiusura */
```


remove

- Cancellazione del file

```
FILE *fp;       /* inutile per la remove */
int result;
const char *fname = "MioFile";

result = remove(fname);
  /* result != 0 se errore di cancellazione */
```


rename

- Ridenominazione del file

```
FILE *fp;       /* inutile per la rename */
const char *fnameOld = "VecchioNome";
const char *fnameNew = "NuovoNome";
int result;

result = rename(fnameOld, fnameNew);
  /* result != 0 se errore di ridenominaz. */
```


rename (ii)

- Ridenominazione del file con cambio di directory

```
FILE *fp;       /* inutile per la rename */
char *fnameOld = "MyDir1/Vecchio";
char *fnameNew = "MyDir2/Nuovo";

result = rename(fnameOld, fnameNew);
  /* result != 0 se errore di ridenominaz. */
```

Gestione degli errori nell'accesso ai file

- Raggiungimento della fine del file: *feof* indica se si è raggiunta la fine del file;
- Verifica di eventuali errori di accesso: *ferror* indica se si sono verificati errori di accesso al file;
- Cancellazione di eventuali errori di accesso: *clearerr* riporta al default i valori di *eof* e di *error* per il file.

feof

- Verifica il raggiungimento della fine del file su un file precedentemente aperto, leggendo la variabile *eof* associata a *fp*

```
FILE *fp;
int result;

result = feof(fp);
  /* result != 0 se fine file raggiunta */
```


ferror

- Controlla errori di accesso al file su un file precedentemente aperto, leggendo la variabile *error* associata a *fp*

```
FILE *fp;
int result;

result = ferror(fp);
  /* result != 0 se errore di accesso al file */
```


clearerr

- Ristabilisce i valori di default per *eof* ed *error* su file precedentemente aperto

```
FILE *fp;

void clearerr(fp);
```

Accesso diretto ai file

- Consente:
  - la modifica della posizione corrente nel file precedentemente aperto, quindi permette
  - l'accesso ad un particolare byte del quale è nota la posizione all'interno del file.
- Dispone delle funzioni:
  - fseek:
  - ftell;
  - rewind.

fseek

- Modifica della posizione corrente di un file precedentemente aperto

```
FILE *fp;
long offset;
int result, refpoint;

result = fseek(fp, offset, refpoint);
```


fseek

- offset:
  - indica lo scostamento e può assumere valori positivi o negativi;
- refpoint;
  - indica se lo scostamento è relativo a:
    - SEEK_SET: inizio del file;
    - SEEK_CUR: posizione corrente;
    - SEEK_END: fine del file.
- result:
  - se !=0 indica un errore nell'operazione.

ftell

- Riporta la posizione corrente di un file precedentemente aperto

```
FILE *fp;
long offset;

offset = ftell(fp);
```

rewind

- Riporta all'inizio la posizione corrente di un file precedentemente aperto

```
FILE *fp;

void rewind(fp);
  /* equivale a fseek(fp, 0, SEEK_SET) */
```


Lettura e scrittura di caratteri su file

- Lettura: legge un carattere dal file precedentemente aperto.
  - getc;
  - fgetc
- Scrittura: scrive un carattere sul file precedentemente aperto.
  - putc;
  - fputc

fgetc

- Lettura di un carattere da un file precedentemente aperto

```
FILE *fp;
char c;

c = fgetc(fp);
  /* c == EOF se fine file raggiunta */
```


fputc

- Scrittura di un carattere su un file precedentemente aperto

```
FILE *fp;
int result;
char c;

result = fputc(fp, c);
  /* result != 0 se errore di scrittura */
```

Lettura e scrittura di stringhe su file

- Lettura: legge una stringa di caratteri dal file precedentemente aperto.
  - gets;
  - fgets
- Scrittura: scrive una stringa di caratteri sul file precedentemente aperto.
  - puts;
  - fputs

fgets

- Lettura di una stringa di *n* caratteri da un file precedentemente aperto

```
FILE *fp;
char *s;
int n;

if (fgets(s, n, fp) == NULL)
                {errore di lettura};
```


fgets

- L'operazione di lettura può:
  - leggere *n* caratteri;
  - leggere meno di *n* caratteri se incontra un '\n'.
- La stringa in memoria è terminata da '\0'.
- Se si incontra eof prima di aver letto *n* caratteri, i caratteri letti sono assegnati a *s* che termina con '\0'.

fputs

- Scrittura di una stringa su un file precedentemente aperto

```
FILE *fp;
char *s;
int result;

result = fputs(s, fp);
  /* result = 0 se errore in scrittura */
```

Lettura e scrittura di *struct*

- L'operazione prevede l'accesso a file per blocchi.
- Lettura: legge un blocco di dati da un file precedentemente aperto.
  - fread
- Scrittura: scrive un blocco di dati su un file precedentemente aperto.
  - fwrite

Lettura e scrittura di *struct*

- Si ipotizza sia data una *struct* del tipo:

```
typedef struct {
        char cognome[20];
        char nome[20];
        int matricola;
        } studente;
```


fread

- Lettura di un blocco di dati da un file precedentemente aperto

```
FILE *fp;
studente stud;
int result, dimelem, numelem;

result = fread(&stud, dimelem, numelem, fp);
```


fread

- &stud:
  - indirizzo di studente;
- dimelem;
  - dimensione dell'elemento:
    sizeof(stud);
- numelem:
  - numero di elementi da leggere;
- result:
  - numero di elementi effettivamente letti.
    Se result != numelem indica un errore nell'operazione.

fwrite

- Scrittura di un blocco di dati su un file precedentemente aperto

```
FILE *fp;
studente stud;
int result, dimelem, numelem;

result = fwrite(&stud, dimelem, nunelem,
  fp);
```


fwrite

- &stud:
  - indirizzo di studente;
- dimelem;
  - dimensione dell'elemento:
    sizeof(stud);
- numelem:
  - numero di elementi da scrivere;
- result:
  - numero di elementi effettivamente scritti.
    Se result != numelem indica un errore nell'operazione.

La programmazione di sistema

La gestione dei processi

Motivazioni

- Un processo (padre) può dover mandare in esecuzione un altro processo (figlio), eventualmente sospendendo la propria esecuzione in attesa della terminazione del processo figlio.
- Esempio: interprete comandi.

Aspetti generali relativi ai processi

- Ogni processo è identificato in modo univoco da un PID (Process IDentifier).
- Tutti i processi sono creati da altri processi e quindi hanno un processo *padre* (unica eccezione: il processo *init*, primo processo creato all'avviamento del SO, che non ha un processo padre).

Aspetti generali relativi ai processi

- Dal punto di vista del programmatore di sistema, la memoria di lavoro associata ad un processo può essere vista come costituita da 3 segmenti:
  - **segmento codice** (*text segment*): contiene l'eseguibile del programma;
  - **segmento dati** (*user data segment*): contiene tutte le variabili del programma (globali o statiche, locali allocate su stack, create dinamicamente dal programma tramite **malloc()**);
  - **segmento di sistema** (*system data segment*): contiene i dati non gestiti esplicitamente dal programma in esecuzione, ma dal S.O (ad esempio, la tabella dei file aperti e i descrittori di socket).

Operazione sui processi

- Le principali operazioni sui processi consentono di:
  - generare un processo;
  - attendere la terminazione di un processo;
  - terminare un processo;
  - sostituire il codice di un processo in esecuzione.

(il presente lucido ne richiama uno precedente)

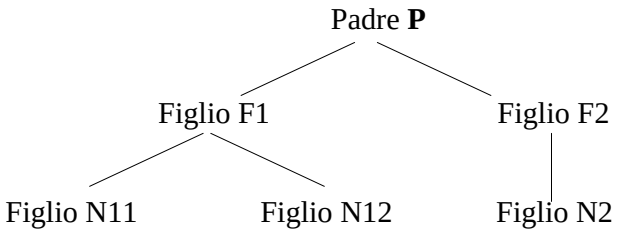
fork

- Crea un processo figlio (o child) identico al processo padre (o parent);
- il figlio è una copia identica del padre all'istante della fork. Le variabili, i file aperti, i socket utilizzati sono **duplicati** nel figlio;
- l'unica differenza tra figlio e padre è la variabile *pid* del processo figlio generato:
- il padre conosce il *pid* dello figlio, il figlio non conosce il proprio *pid*. Quindi, nel figlio, *pid*=0.

fork

- Un processo figlio *F* può a sua volta generare un ulteriore processo figlio *N*.
- Si stabilisce così una gerarchia di processi.

Gerarchia di processi



P è padre di F1 e F2
F1 è padre di N11 e N12
F2 è padre di N2

Sintassi di *fork*

```
pid_t fork ()
/* Biforca il processo in padre e figlio:
  al figlio restituisce sempre 0; al
  padre restituisce il pid > 0 del figlio
  biforcato; la funzione restituisce –1
  se la biforcazione del processo
  fallisce (restituisce solo al padre,
  ovviamente, visto che il figlio non è
  stato biforcato), per esempio per
  insufficienza di memoria per la
  creazione del processo figlio, o altro
  motivo */
```


Sintassi di *fork* (ii)

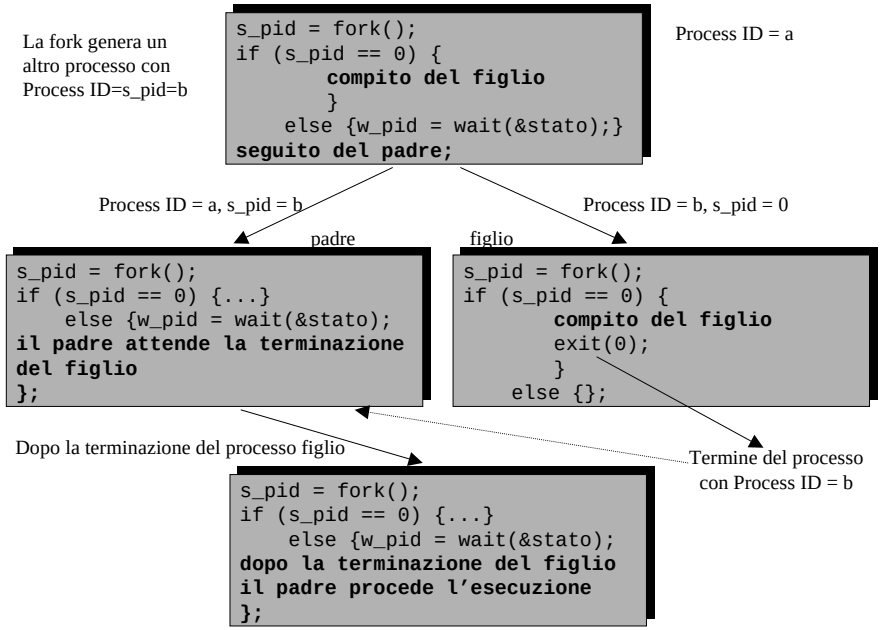
- Sintassi:

  pid_t pid = fork(); /* restituisce pid_t */

  pid assume valore:
  - 0 nel figlio;
  - -1 nel padre, se la fork non è stata eseguita (ad es. per mancanza di memoria);
  - qualsiasi altro valore nel padre indica il pid del figlio.

Utilizzo di *fork*

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

void main(int argc, char * argv[])
{  pid_t pid;
   int retstatus;

   pid = fork();
   if (pid==0) {
        /* sono nel figlio*/
        ...
        /* il figlio termina restituendo al padre lo */
        /* stato di terminazione */
        exit(retstatus);
        }
   else {
        /* pid != 0, sono nel padre */
   }
}
```

wait

- Sospende l'esecuzione del processo padre ed attende la terminazione di un qualsiasi processo figlio;
- se il figlio termina prima che il padre esegua la *wait*, l'esecuzione della *wait* nel padre termina istantaneamente.

Sintassi di *wait*

pid_t wait (int * stato)

/* Mette un processo padre in stato di attesa dell'evento di terminazione di un processo figlio qualsiasi, e restituisce il pid del figlio effettivamente terminato; la funzione restituisce –1 se il processo padre esce dalla "wait" per un errore che non abbia a che vedere con la terminazione di un figlio; l'argomento "stato" assume il valore di "exit" del processo figlio terminato.

Parametri: **stato** è il puntatore a un intero, che conterrà lo stato di uscita del figlio terminato (contano solo gli 8 bit più significativi; gli 8 bit meno significativi vengono impostati da parte del sistema operativo stesso) */

Esempio di *wait*

- Sintassi:

  my_pid = wait(&status);

  /* restituisce un tipo pid_t */

  nel padre:
  - my_pid assume il valore del pid del figlio terminato;
  - status, di tipo intero, assume il valore del codice di terminazione del processo figlio.

Utilizzo di *wait*

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

void main(int argc, char * argv[])
{  pid_t pid, my_pid;
   int status;

   pid = fork();
   if (pid==0) {             }
   else {
   printf("Ho generato il processo figlio con pid %d\n",pid);
               /* pid != 0, sono nel padre */
   ...
   my_pid = wait(&status);
   printf{"E' terminato il processo %d con esito %d\n",my_pid,
   status/256);
   }
}
```


waitpid

- Sospende l'esecuzione del processo padre ed attende la terminazione del processo figlio di cui viene **fornito** il pid (my_pid);
- se il figlio termina prima che il padre esegua la *waitpid*, l'esecuzione della *waitpid* nel padre termina istantaneamente.

Sintassi di *waitpid*

pid_t waitpid (pid_t pid, int * stato, int opzioni)

/* Mette un processo padre in stato di attesa dell'evento di terminazione del processo figlio avente pid pari a "pid", e ne restituisce il pid; la funzione è insensibile alla terminazione di eventuali altri figli aventi pid diverso da "pid"; la funzione restituisce –1 se il processo padre esce dalla "waitpid" per un errore che non abbia a che vedere con la terminazione del figlio "pid"; l'argomento "stato" assume il valore di "exit" del processo figlio terminato.

Parametri: **pid** è il "pid" del figlio della cui terminazione il padre si mette in attesa; **stato** è il puntatore a un intero, che conterrà lo stato di uscita del figlio terminato (solo gli 8 bit più significativi); **opzioni** è un intero che specializza la funzione "waitpid", e di solito vale 0 (altrimenti vedere il manuale in linea della funzione, perché le opzioni possibili sono parecchie) */

Esempio di *waitpid*

- Sintassi:

  my_pid = waitpid(pid, &status,
  options); /* restituisce un pid_t*/

  nel padre:
  - my_pid assume il valore del pid del figlio terminato;
  - status assume il valore del codice di terminazione del processo figlio;
  - options specifica ulteriori opzioni (ipotizziamo > 0).

Utilizzo di *waitpid*

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

void main(int argc, char * argv[])
{  pid_t pid, my_pid;
   int status;

   pid = fork();
   if (pid==0) {             }
   else {
        printf("Ho generato il processo figlio con pid %d\n",pid);
               /* pid != 0, sono nel padre */
   ...
   printf{"Attendo la terminazione del figlio con pid %d\n",pid)
   my_pid = waitpid(pid, &status, 1);
   printf{"E' terminato il processo %d con esito %d\n",my_pid,
   status/256);
   }
}
```


exec

- Sostituisce il codice del processo corrente con il codice del programma specificato;
- mantiene lo stesso pid;
- passa i parametri al processo con il nuovo codice attraverso la linea di comando (argc e argv - vedi appendice);
- esistono numerosi varianti: considereremo la funzione execl.

Sintassi di *execl*

- Sintassi:

  int execl(char *path_programma,
  char *arg0, char *arg1, … char *argn, NULL);
  /* restituisce int */

  path_programma: path completo del file

  arg0, arg1, ... argn: puntatori a stringhe che verranno passate come parametri al main del programma

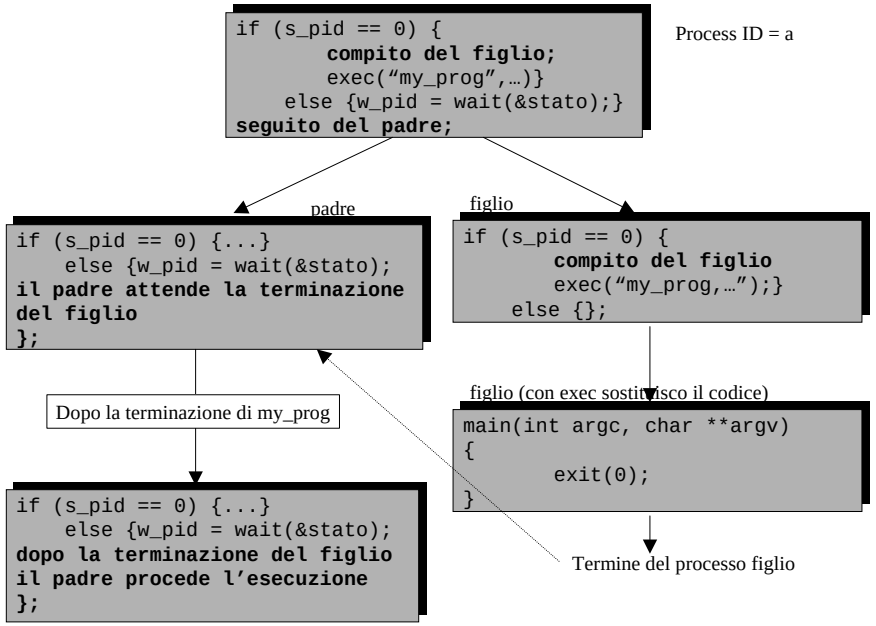
  arg0 deve essere il nome del programma

  l'ultimo elemento deve essere un puntatore a NULL

  il valore restituito è:
  - 0 se l'operazione è stata eseguita correttamente;
  - -1 se c'è stato un errore e l'operazione di sostituzione del codice è fallita.

Utilizzo di *execl*

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

void main(int argc, char * argv[])
{
   int retstatus;
   char par0[] = “my_main1”;
   char par1[] = “primo parametro”;
   char par2[] = “secondo parametro”;
   retstatus = execl(“/usr/home/pozzi/main1", par0, par1,
               par2, NULL);
}
```

exit

- Pone termine al processo corrente (figlio) restituendo un codice al processo padre.
- Se il processo che è terminato non ha un processo padre, il codice della exit viene restituito all’interprete comandi.

Sintassi di *exit*

- Sintassi:

  void exit(int status);

  /* non restituisce nulla al */

  /* processo che esegue la exit*/

  il processo padre può accedere al codice di terminazione del figlio.

Utilizzo di *exit*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void main(int argc, char * argv[])
{  pid_t pid;
   int status,retstatus;
   pid = fork();
   if (pid==0) {
        /* il figlio restituisce al padre lo stato di terminazione */
        retstatus = 250;
        exit(retstatus);}
   else {      /* pid != 0, sono nel padre */
        pid = wait(&status);
        printf("E' terminato il processo %d con esito
        %d\n",pid,status/256);
        /* stampera' il numero del pid del figlio e il numero 250 */
   }
}
```


La programmazione di rete in ambiente multiprocesso

Il parallelismo nei server

Necessità di parallelismo nei server

- Server sequenziali:
  - accettano **una sola** connessione alla volta;
  - non sono pertanto particolarmente utili. E' solo una comunicazione tra due processi per volta, eventualmente accodando altri processi che vogliano connettersi allo stesso server (*listen*);
  - sono anche detti monoconnessione.

Necessità di parallelismo nei server

- Server paralleli:
  - superano i limiti dei server sequenziali sfruttando il meccanismo di parallelismo;
  - utilizzeremo il parallelismo di Unix, trascurando quello di Windows;
  - il client non ha modo di distinguere se si è connesso ad un server sequenziale o ad un server parallelo: può solo accorgersi di una connessione rifiutata.

Funzionamento di un server parallelo

- Il server si pone in attesa di richieste di connessioni (*accept*) su un port *srvport*;
- alla ricezione di una richiesta di connessione (*connect* del client su *srvport*), il server crea tramite la *fork* un nuovo processo detto **figlio**, identico a se stesso (processo **padre**, impropriamente detto anche server principale).

All'interno del server parallelo

- Dopo la creazione del processo figlio, il padre chiude la connessione col client e rimane in attesa di nuove richieste di *connect*, che utilizzeranno sempre lo stesso *srvport*;
- Il figlio utilizza il socket aperto dal padre alla *connect*. Al termine, il figlio chiude la connessione (*close*) e termina (*exit*).

Funzionamento di base di un server parallelo

```
while (1) /* il server è in un ciclo perenne */
{
   new_sd = accept(sd, ...)
   pid = fork();
   if (pid==0) /*sono nel figlio */
        { /*esegui le richieste del client utilizzando
               il socket connesso new_sd; alla fine
               chiudi new_sd ed esegui exit  */
        }
   else /* sono nel padre */
        { /*chiudi new_sd e
        riprendi ad eseguire la accept su sd */
   }
}
```


Realizzazione di un server parallelo

Modificare Server3 così da renderlo un server parallelo (Server3par).

n.b. sui calcolatori Unix, il tipo pid_t coincide con il tipo int. Verrà quindi utilizzata la dichiarazione del tipo:

```
int p_id;
```

Listato di Server3Par (i)

```
                        /* programma Server3Par  */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define PORT 4000
#define MAXCONN 5

void addr_initialize();

void main(int argc, char * argv[])
{  int  sd,new_sd,bind_result,listen_result; int pid; char inbuf;
   struct sockaddr_in server_addr;
   struct sockaddr_in client_addr;
   int client_len=sizeof(client_addr);
   char rispostaA[]="**il Server ha eseguito il comando A**";
   char rispostaB[]="**il Server ha eseguito il comando B**";
   char errmessage[]="**comando errato**";
   char terminatore[2]={'*','\n'};
```


Listato di Server3Par (ii)

```
addr_initialize(&server_addr, PORT, INADDR_ANY);
sd=socket(AF_INET,SOCK_STREAM,0);
bind_result=bind(sd,(struct sockaddr*) &server_addr,sizeof(server_addr));
listen_result=listen(sd,MAXCONN);
while (1) {
     printf("\nMi pongo in attesa di richieste di connessione\n");
     new_sd=accept(sd,(struct sockaddr*) &client_addr,
             &client_len);
     printf("Ho accettato una connessione da: %d\n",
             ntohs(client_addr.sin_port));
     pid=fork();
```


Listato di Server3Par (iii)

```
if (pid==0){
   do{   recv(new_sd,&inbuf,1,0);
         printf("Ho ricevuto il comando %c ",inbuf);
         printf("da: %d \n", ntohs(client_addr.sin_port));
         switch (inbuf)
         { case 'e':break;
           case 'a':{send(new_sd,rispostaA, sizeof(rispostaA),0);
                         send(new_sd,terminatore,2,0);break;}
           case 'b':{printf("i parametri sono: ");
                         do{recv(new_sd,&inbuf,1,0);
                                 printf("%c",inbuf);
                         }while (inbuf!='\0');
                         printf(" ");
                         do {recv(new_sd,&inbuf,1,0);
                                 printf("%c",inbuf);
                         }while (inbuf!='\0');
                         printf(" \n");
                         send(new_sd,rispostaB,sizeof(rispostaB),0);
                         send(new_sd,terminatore,2,0);break;}
```


Listato di Server3Par (iv)

```
                  default:{send(new_sd,errmessage, sizeof(errmessage),0);
                                send(new_sd,terminatore,2,0);break;}
                }
           } while (inbuf!='e');
           printf("\nChiudo la connessione con: %d\n\n",
                ntohs(client_addr.sin_port));
           close(new_sd);
           exit();
         }                /* chiude if (pid == 0) */
         close(new_sd);   /* il padre chiude il socket che */
                          /* viene utilizzato dal figlio */
   }                      /* chiude while (1) */
}                         /* chiude main */
```

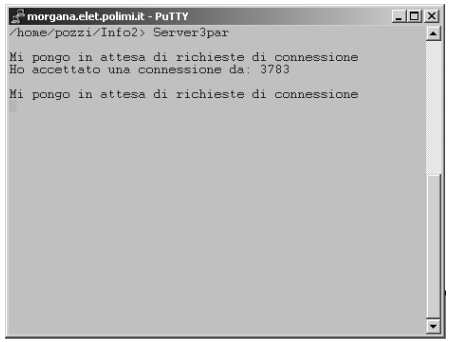
Esecuzione di Server3Par

- Consideriamo ora un esempio di esecuzione di Server3Par che accetta connessioni da 3 distinti processi, rispettivamente C1, C2 e C3.
- C1, C2 e C3 sono esecuzioni del programma Client3 (o WClient3).
- Client3 non ha modo di riconoscere se il server cui si collega è sequenziale o parallelo.

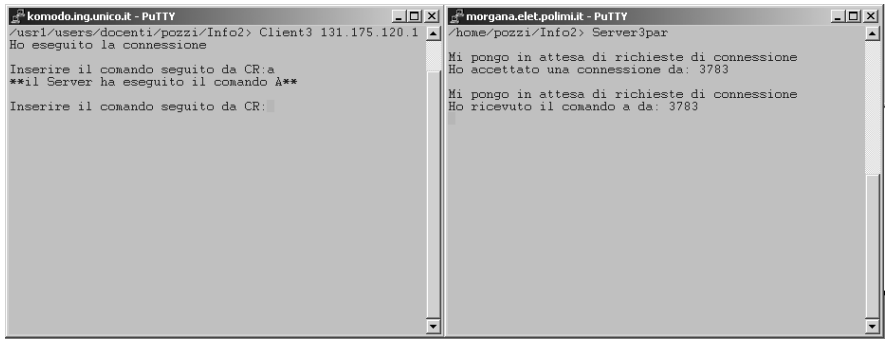
Esecuzione di Server3Par

- I processi figli di Server3Par utilizzano la stessa sessione di terminale del processo padre.
- I messaggi del padre e dei figli saranno alternati su una unica finestra.

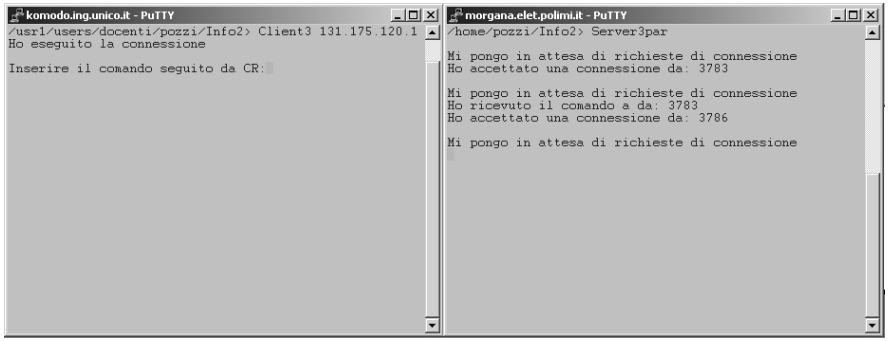
Avvio di Server3Par



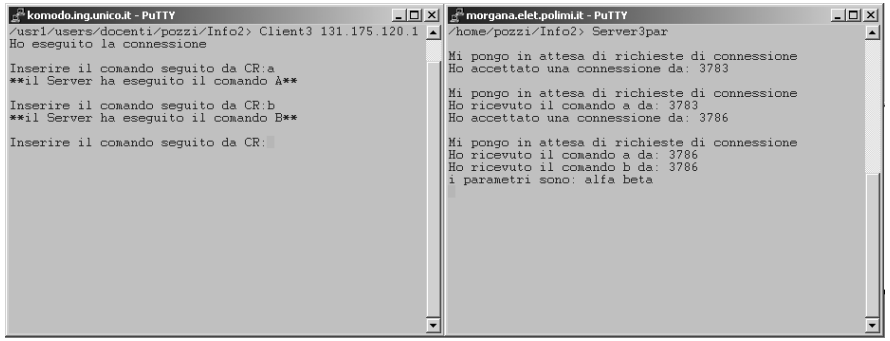
C1 esegue il comando a



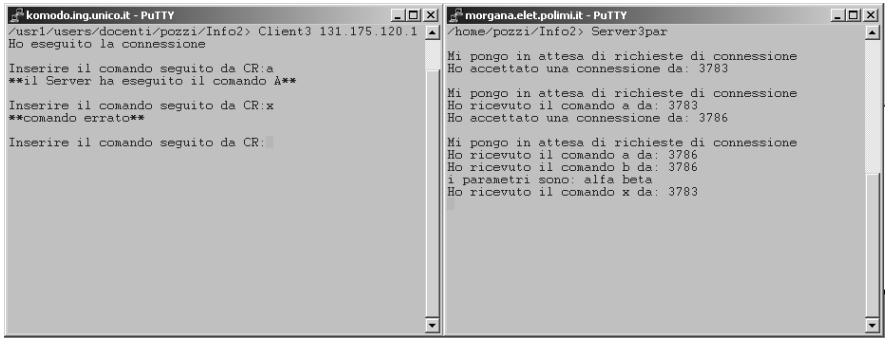
Avvio di C2



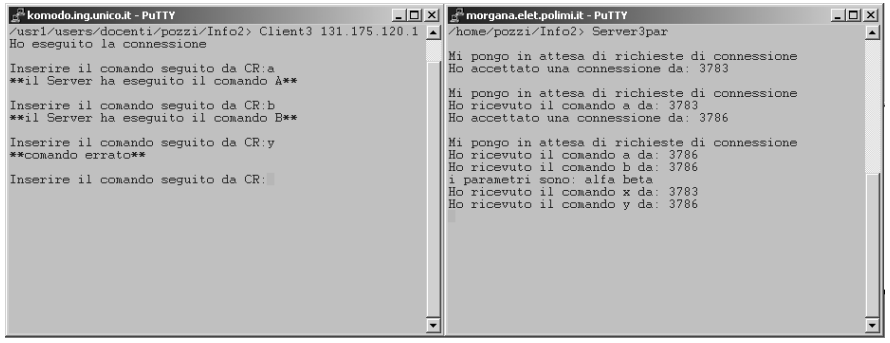
C2 esegue i comandi a e b



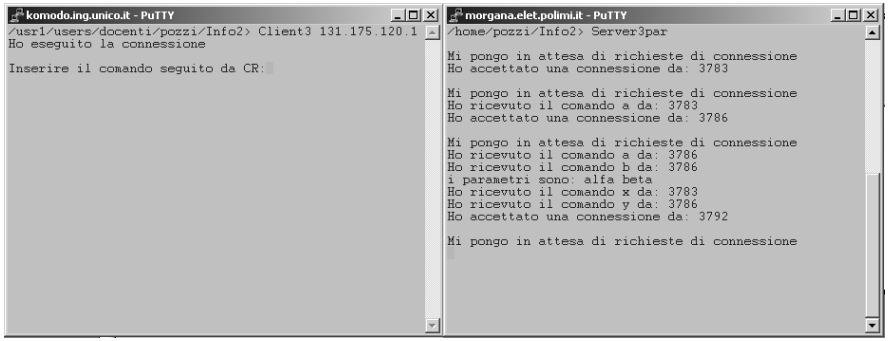
C1 esegue il comando x



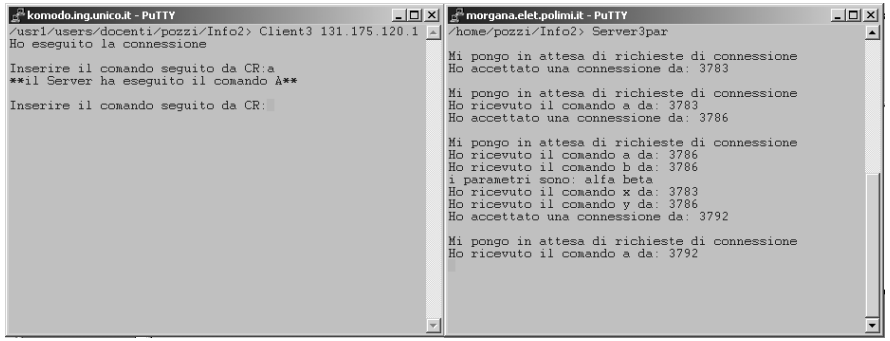
C2 esegue il comando y



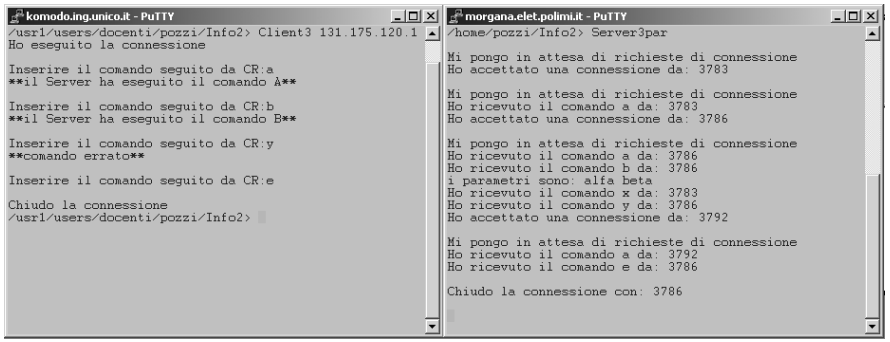
Avvio di C3



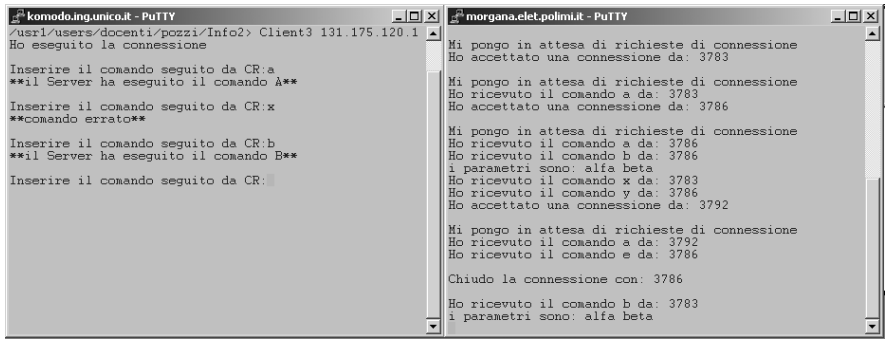
C3 esegue il comando a



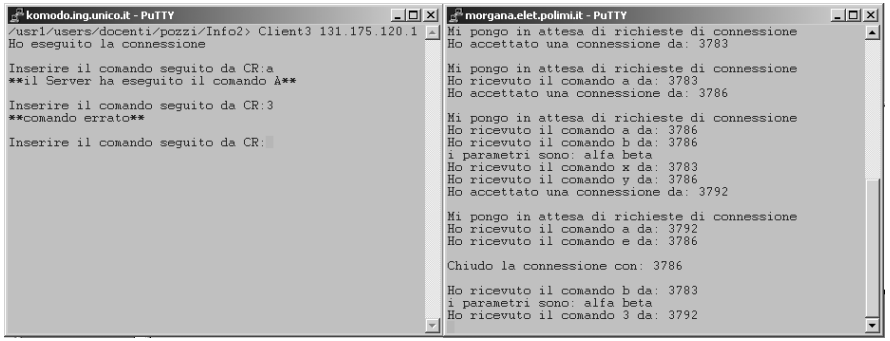
C2 chiude la connessione



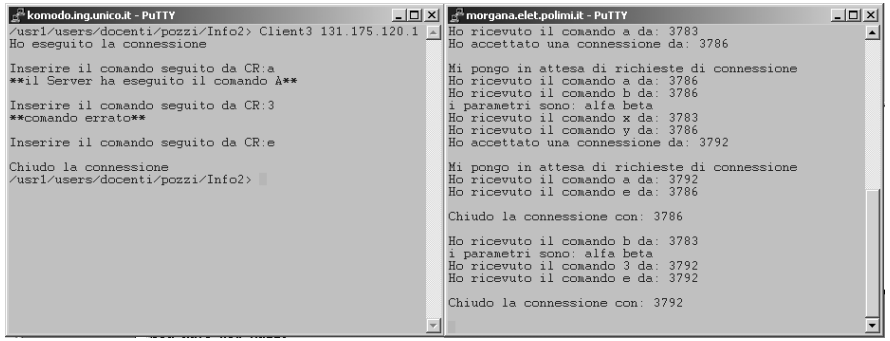
C1 esegue il comando b



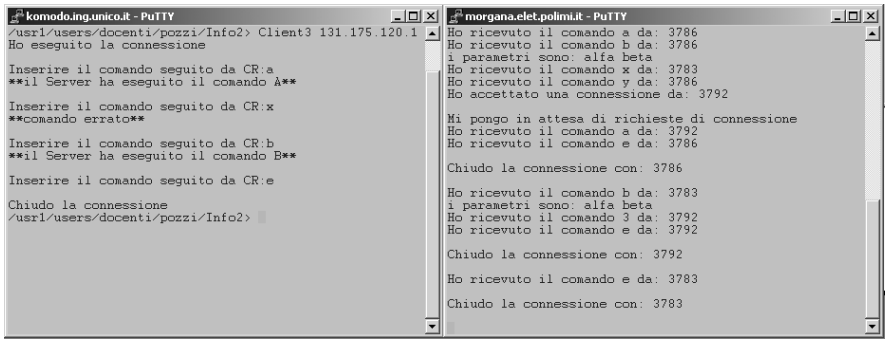
C3 esegue il comando 3



C3 chiude la connessione



C1 chiude la connessione



Invocazione di programmi di servizio

- Un server presenta un'interfaccia attraverso cui differenti client richiedono a quel server l'esecuzione di programmi già esistenti.
- Il server parallelo (*fork* e generazione di figlio per ogni connessione ricevuta) deve:
  - eseguire il programma, specificato dal client, tramite una *exec* passandogli il descrittore del socket;
  - attendere il termine del programma (*wait*).

Server3Exec

- Rispetto a Server3Par:
  - è sempre multiprocesso, quindi per ogni connessione esegue una fork;
  - ad ogni richiesta dal client, genera un ulteriore processo (tramite una fork) che esegue il comando indicato dal client (tramite la execl);
  - passa al programma invocato (MainA o MainB) il descrittore del socket per comunicare col client.

Listato di Server3Exec (i)

```
                    /* programma Server3Exec  */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define PORT 4000
#define MAXCONN 5

void addr_initialize();
void main(int argc, char * argv[])
{
   int  sd,new_sd,bind_result,listen_result; char inbuf;
   struct sockaddr_in server_addr;
   struct sockaddr_in client_addr;
   int client_len=sizeof(client_addr);
   char errmessage[]="**comando errato**"; /* rispetto a Server3Pa mancano */
   char terminatore[2]={'*','\n'};         /* rispostaA e rispostaB */
   char sdstring[7];                       /* nuovo rispetto a Server3Par */
   int pid,my_pid,status;
```


Listato di Server3Exec (ii)

```
addr_initialize(&server_addr, PORT, INADDR_ANY);
sd=socket(AF_INET,SOCK_STREAM,0);
bind_result=bind(sd,(struct sockaddr*) &server_addr,sizeof(server_addr));
listen_result=listen(sd,MAXCONN);
while (1)
   {
   printf("\nMi pongo in attesa di richieste di connessione\n");
   new_sd=accept(sd,(struct sockaddr*) &client_addr, &client_len);
   printf("Ho accettato una connessione da: %d\n",
   ntohs(client_addr.sin_port));
   pid=fork();

   if (pid==0)
      {
      do
         {
         recv(new_sd,&inbuf,1,0);
         printf("Ho ricevuto il comando %c ",inbuf);
         printf("da: %d \n", ntohs(client_addr.sin_port));
```


Listato di Server3Exec (iii)

```
switch (inbuf)
    {
    case 'e':break;
    case 'a':{sprintf(sdstring,"%d",new_sd);
             pid=fork();
             if (pid==0) {
                    execlp("./MainA", "MainA", sdstring, NULL);
                    printf("errore di execl"); exit(255)}
             else {my_pid = wait(&status); break;}}
    case 'b':{sprintf(sdstring,"%d",new_sd);
              pid=fork();
              if (pid==0) {
                    execlp("./MainB", "MainB", sdstring, NULL);
                    printf("errore di execl"); exit(255)}
              else {my_pid = wait(&status); break;}}
    default: {send(new_sd,errmessage, sizeof(errmessage),0);
             send(new_sd,terminatore,2,0);break;}
    }                       /* chiude switch */
 } while (inbuf!='e');    /* chiude do */
```

Listato di Server3Exec (iv)

```
        printf("\nChiudo la connessione con: %d\n\n",
                  ntohs(client_addr.sin_port));
        close(new_sd);
        exit();
      }               /* chiude if (pid == 0) */
      close(new_sd);  /* il padre chiude il socket che */
                      /* viene utilizzato dal figlio */
   }                  /* chiude while (1) */
}                     /* chiude main */
```


Listato di MainA

```
                    /* programma MainA  */

void main(int argc, char * argv[])
{
   int sd;
   char terminatore[2] = {'*','\n'};
   char rispostaA[] = "**MainA ha eseguito il comando A**";

   printf("  MainA\n");
   sd = atoi(argv[1]);
   send(sd, rispostaA, sizeof(rispostaA),0);
   send(sd, terminatore,2,0);
   printf("  MainA: ho terminato\n");
   exit(0);
}
```


Comunicazione Client3-MainB

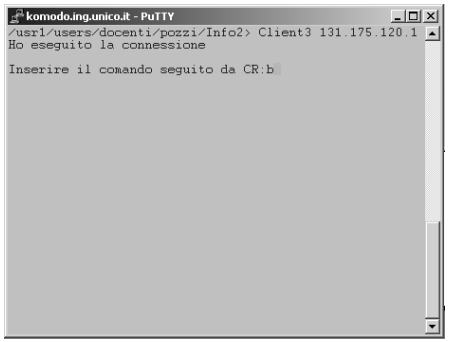
- La comunicazione tra Client3 e MainB avviene attraverso il descrittore del socket.
- Client3 non si accorge di parlare con MainB: è convinto di parlare con Server3Exec.
- MainB riceve sulla linea di comando il descrittore del socket per colloquiare con Client3.

Listato di MainB

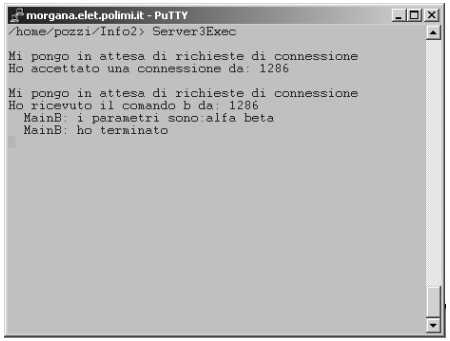
```
                    /* programma MainB*/
void main(int argc, char * argv[])
{  int  sd; char inbuf;
   char rispostaB[] = "**MainB ha eseguito il comando B**";
   char terminatore[2] = {'*','\n'};

   sd = atoi(argv[1]);
   printf("  MainB: i parametri sono:");
        do {   recv(sd, &inbuf, 1, 0);
               printf("%c", inbuf);
        } while (inbuf != '\0');
        prinf(" ");
        do {   recv(sd, &inbuf, 1, 0);
               printf("%c",inbuf);
        } while (inbuf != '\0');
        printf("\n" );
   send(sd, rispostaB, sizeof(rispostaB),0);
   send(sd, terminatore, 2, 0);
   printf("  MainB: ho terminato\n");
   exit(0);
}
```

Avvio di Server3Exec

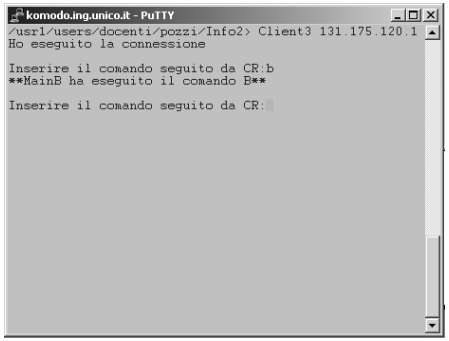
Client3 richiede il comando e lo invia al server



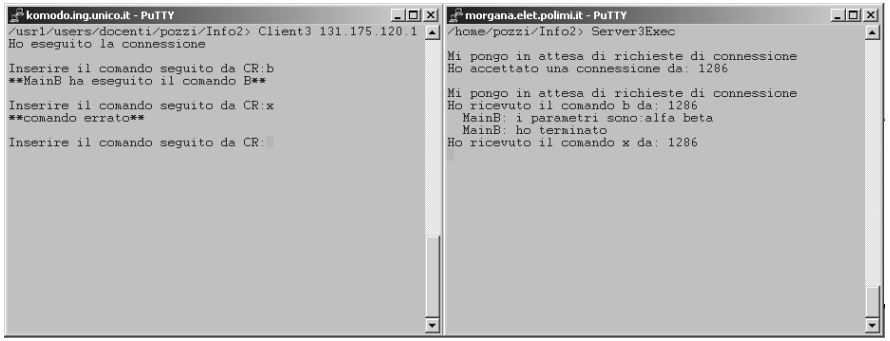
Server3Exec riceve il comando b e lancia l'escuzione di MainB



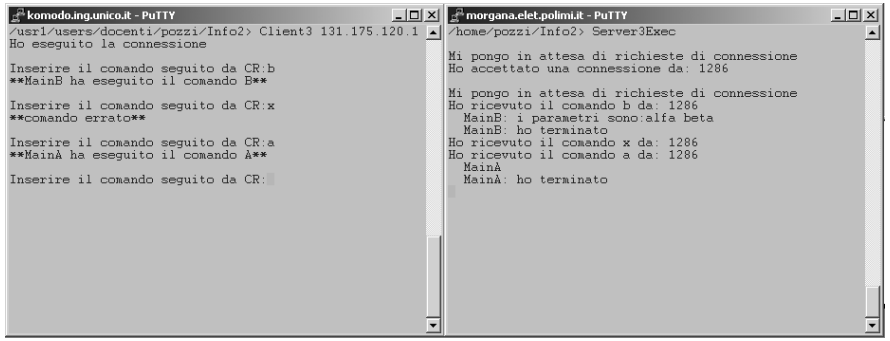
Client3 riceve i messaggi da MainB



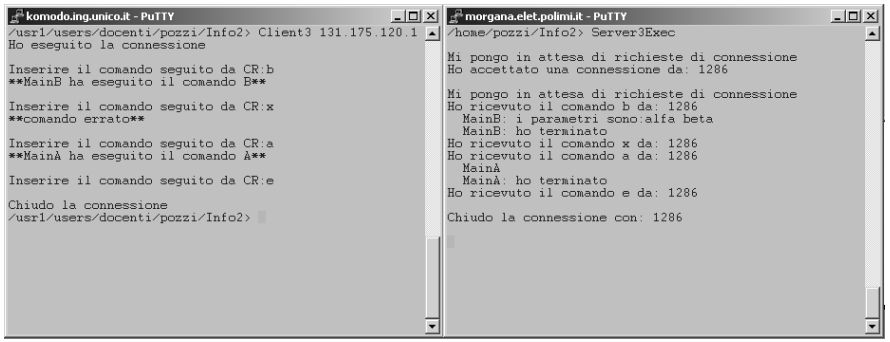
Client3 esegue il comando x



Client3 esegue il comando a



Client3 termina



Considerazione conclusive sui server

- Molti server sono attivati quando la macchina viene accesa e devono rimanere sempre attivi (ad es. server Web). In Unix sono detti *daemons*.
- Spesso i daemons non devono disporre di una sessione di interprete comandi e di terminale associati (esecuzione in *background*). Non dispongono di un terminale associato al processo.

Considerazioni conclusive sui server

- I daemons utilizzano file di log, sui quali scrivono eventuali messaggi di errore.
- Per alcuni servizi vengono utilizzati port riservati (con numero compreso tra 0 e 1023). Per avviare un server che utilizzi un port riservato è necessario avere i privilegi di gestore del sistema.

Daemons

- La creazione di un processo in background avviene specificando il comando e terminando la linea di comando con il carattere '&'. Ad es:

  $ Server3Par &

- Esiste un file di configurazione (*inetd*) nel quale è possibile specificare quali siano i daemons che devono essere attivati alla accensione della macchina.

Appendice A

Il passaggio di parametri ad un programma attraverso la linea di comando, nel linguaggio C

Specifica di parametri sulla linea di comando

- Motivazioni:
  - specifica dei valori per alcuni parametri del programma così da velocizzarne la lettura.
- Il metodo deve essere:
  - standard per l'utente;
  - funzionante su ogni compilatore e con ogni sistema operativo per avere portabilità del codice.

I parametri sulla linea di comando

- L'utente deve conoscere:
  - il formato secondo il quale i parametri sono specificati;
  - l'ordine secondo il quale i parametri devono essere forniti.
- Il programma deve:
  - verificare la correttezza del tipo e dei valori forniti per i parametri.

Passaggio di parametri via linea di comando

```
#include <stdio.h>
void main(int argc, char *argv[])
{
   int i;

   for (i=0; i<argc; i++)
        printf("parametro %i = %s\n", i, argv[i]);

   /* n.b. per leggere un intero: valore = atoi(argv[i]) */
}
```


Appendice B

Le operazioni di I/O su terminale nel linguaggio C

I/O su terminale

- Scrittura, su terminale, di testo formattato:
  - printf
    (analoga all'istruzione cout << del linguaggio C++)
- Lettura, da terminale, di testo formattato:
  - scanf
    (analoga all'istruzione cin >> del linguaggio C++)

printf

- Scrive del testo formattato;
  - specifica comandi di controllo come ritorno a capo, tabulazione e nuova linea;
  - richiede di specificare il formato e le variabili che devono essere utilizzate;
  - per le variabili deve essere indicato il numero di caratteri che devono essere utilizzati per la stampa.

printf

- Principali comandi di controllo:
  - \b inserisce un backspace;
  - \f inserisce un salto pagina;
  - \n va a linea successiva;
  - \r inserisce un ritorno carrello;
  - \t inserisce una tabulazione
  - \ddd inserisce il carattere di cui si fornisce il codice Ascii ottale;
  - \0xYY inserisce il carattere di cui si fornisce il codice ASCII esadecimale.

printf

- Formati per la scrittura di variabili:
  - %c 1 carattere;
  - %5c 5 caratteri;
  - %d intero decimale con segno;
  - %i equivale a %d;
  - %8d intero decimale con segno a 8 cifre incluso il segno;
  - %u intero decimale senza segno;
  - %8u intero decimale senza segno a 8 cifre.

printf

- Formati per la scrittura di variabili:
  - %s stringa terminata da '\0';
  - %40s i primi 40 caratteri della stringa (o meno se si incontra '\0');
  - %x intero esadecimale (0..9abcdef) senza segno;
  - %X intero esadecimale (0..9ABCDEF) senza segno;
  - %o intero ottale senza segno;

printf

- Formati per la scrittura di variabili:
  - %f un float o un double con segno;
  - %f8.3 un float o un double con segno, su 8 caratteri totali e 3 caratteri decimali;
  - %e un float o un double con segno in formato esponenziale indicato con 'e' e con segno;
  - %E un float o un double con segno in formato esponenziale indicato con 'E' e con segno;

printf

- Allineamento a sinistra:
  - e' necessario utilizzare il simbolo '-' tra il carattere '%' ed il carattere che specifica il formato.
  - Es: %-5d intero di 5 caratteri allineati a sinistra

printf

- Ipotizziamo di disporre delle seguenti variabili:
  - my_string = "la mia stringa";
  - my_int = 27;

Uso di *printf*

```
printf("una stringa: %s\n ed un intero: %d\n",
        my_string, my_int);
```

scrivera':

```
una stringa: la mia stringa
 ed un intero: 27
```

```
printf("un intero: %-8d\ned un reale: %f9.7\n",
        my_int, atan(1)*4);
```

scrivera':

```
un intero:      27
ed un reale: 3.1415926
```

scanf

- Legge del testo formattato.
- Richiede di specificare:
  - le variabili che devono essere lette;
  - il formato ed il numero di caratteri che devono essere utilizzati;
  - le variabili lette vengono restituite per indirizzo (quindi &my_string, &my_int).
- La *scanf* utilizza gli stessi formati utilizzati dalla *printf*.

scanf

- Formati per la lettura di variabili:
  - %c 1 carattere;
  - %d intero decimale con segno;
  - %i equivale a %d;
  - %u intero decimale senza segno;
  - %x intero esadecimale;
  - %o intero ottale;
  - %f un float o un double con segno;
  - %e un float o un double con segno;

scanf

- Formati per la lettura di variabili:
  - %s stringa che non include il carattere spazio.

Estensioni di *printf* e *scanf*

- Le operazioni di *printf* e *scanf* possono essere applicate senza variazioni anche a:
  - file:
    - fprintf(fp, "…", …); /* scrittura su file */
    - fscanf(fp, …); /* lettura da file */
  - stringhe:
    - sprintf(string_ptr, "…", …); /* scrittura su stringa */
    - sscanf(string_ptr, …); /* lettura da stringa */