La Programmazione di Rete e di Sistema (iii)

Giuseppe Pozzi

Impianti di Elaborazione (allievi Gestionali – Como)
Facoltà di Ingegneria dell'Informazione
Politecnico di Milano

giuseppe.pozzi@polimi.it
- versione del 26 novembre 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.
- … ecc. ecc.

Si veda "Il sistema operativo ed i processi".

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

(già vista a esercitazione)

La programmazione di sistema

La gestione dei processi

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)

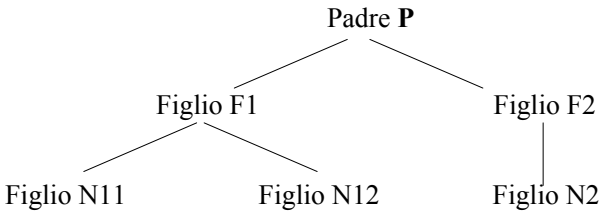
fork

- 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)

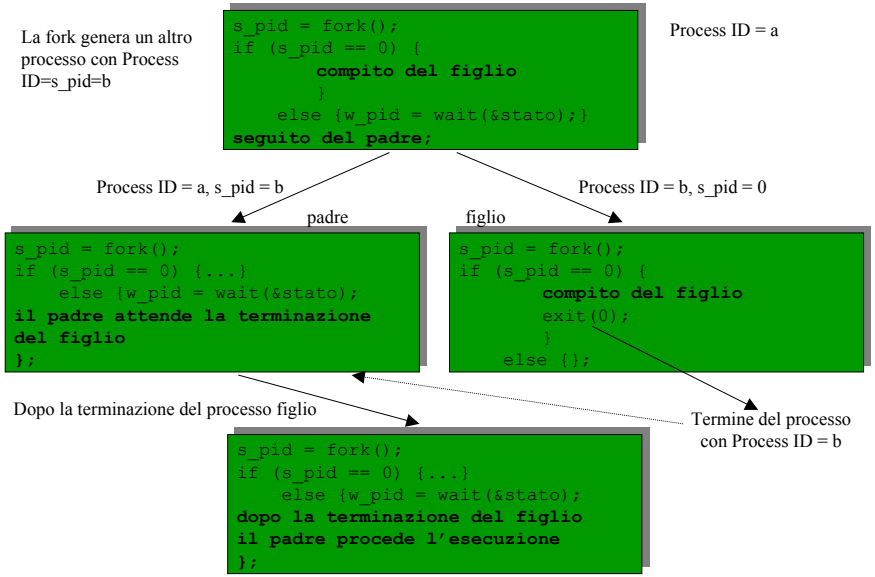
- Sintassi:

  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 meno significativi; gli 8 bit più 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);
    }
}
```


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 meno 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);
   }
}
```


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);
  /* 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

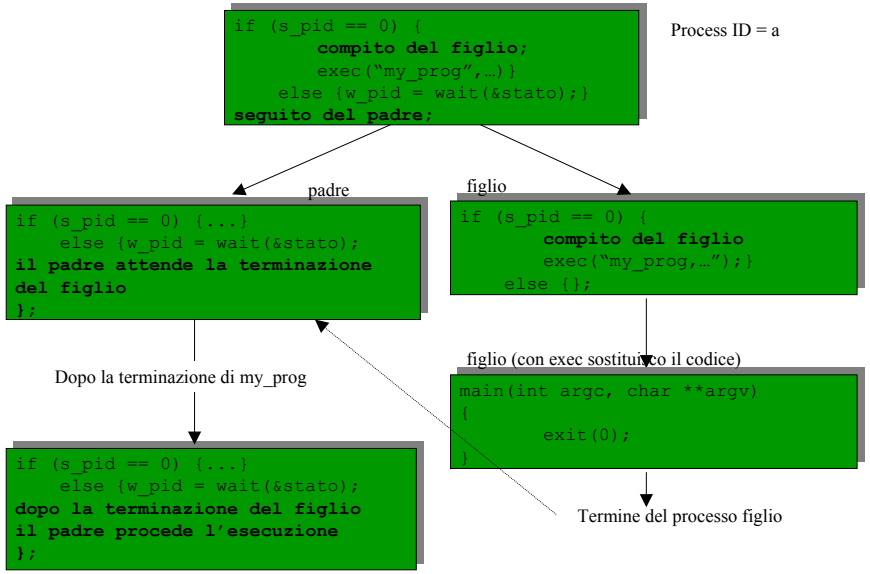
  argn deve essere un puntatore a NULL

  *void main (int argc, char *argv[ ])*

  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[])
{  pid_t pid;
   int retstatus;

   pid = fork();
   if (pid==0) { /* sono nel figlio */
        execl("ls", {"ls", "-lag", "/home", (char *) NULL});
                /* non restituisce status */
        retstatus = 0;
        exit(retstatus);
        }
   else {
        /* pid != 0, sono nel padre */
   }
}
```



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 = 1024;
        exit(retstatus);}
   else {        /* pid != 0, sono nel padre */
        pid = wait(&status);
        printf("E' terminato il processo %d con esito %d\n",pid,status);
        /* stampera' il numero del pid del figlio e il numero 1024 */
   }
}
```

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**, impropria-mente 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 UServer3 così da renderlo un server parallelo (UServer4).

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 UServer4 (i)

```
                        /* programma USERVER4  */
#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 UServer4 (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 UServer4 (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 UServer4 (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 UServer4

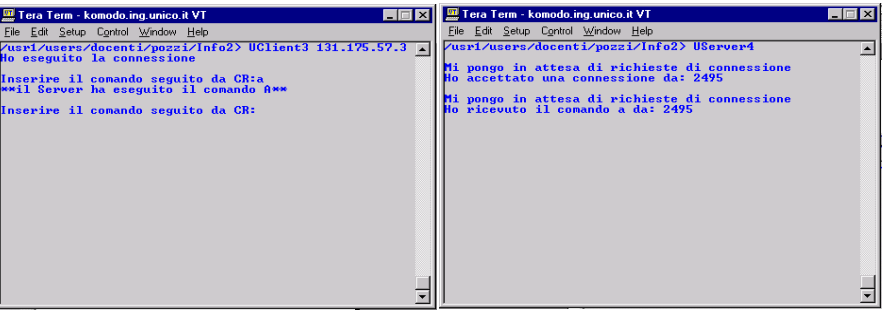
- Consideriamo ora un esempio di esecuzione di UServer4 che accetta connessioni da 3 distinti processi, rispettivamente C1, C2 e C3.
- C1, C2 e C3 sono esecuzioni del programma UClient3 (o WClient3).
- UClient3 non ha modo di riconoscere se il server cui si collega è sequenziale o parallelo.

Esecuzione di UServer4

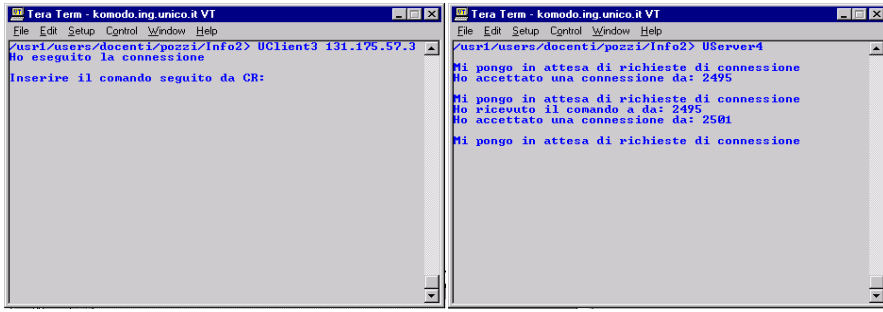
- I processi figli di UServer4 utilizzano la stessa sessione di terminale del processo padre.
- I messaggi del padre e dei figli saranno alternati su una unica finestra.

Avvio di UServer4

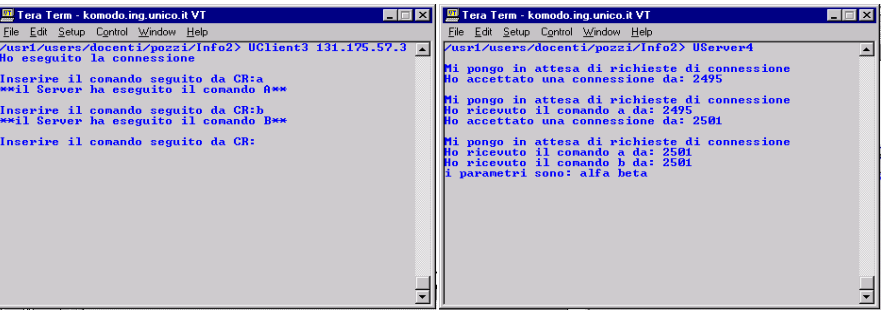
C1 esegue il comando a



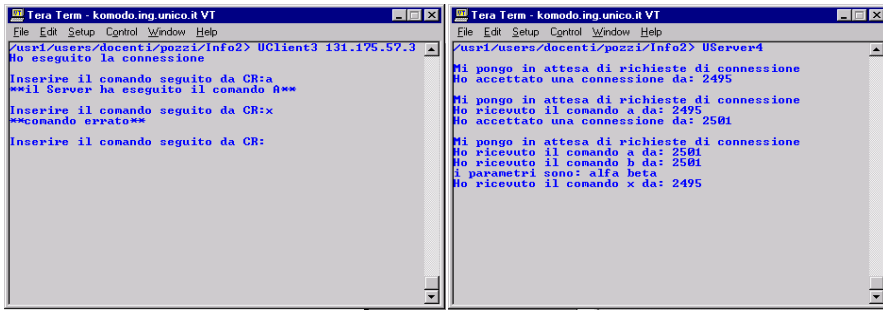
Avvio di C2



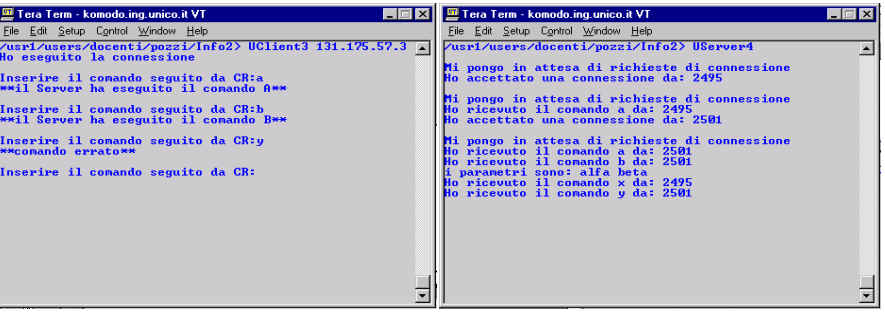
C2 esegue i comandi a e b



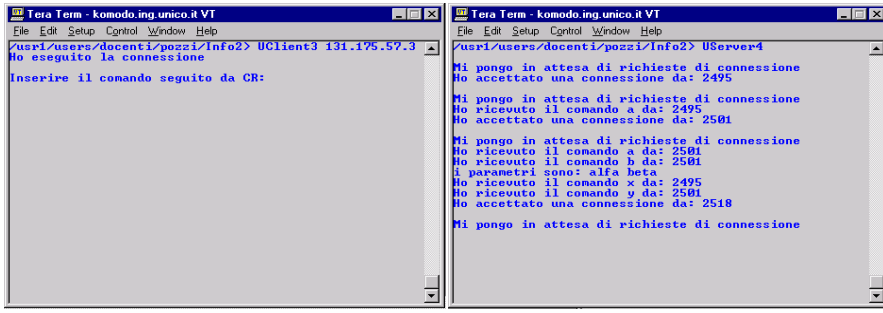
C1 esegue il comando x



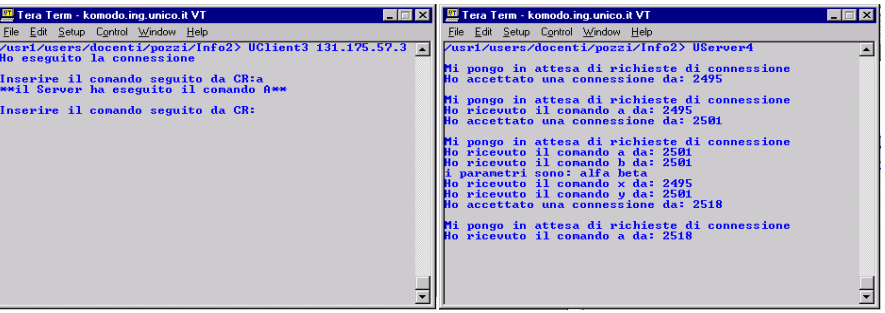
C2 esegue il comando y



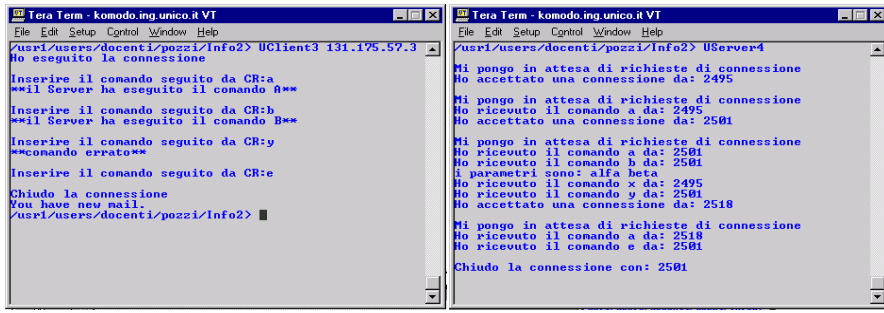
Avvio di C3



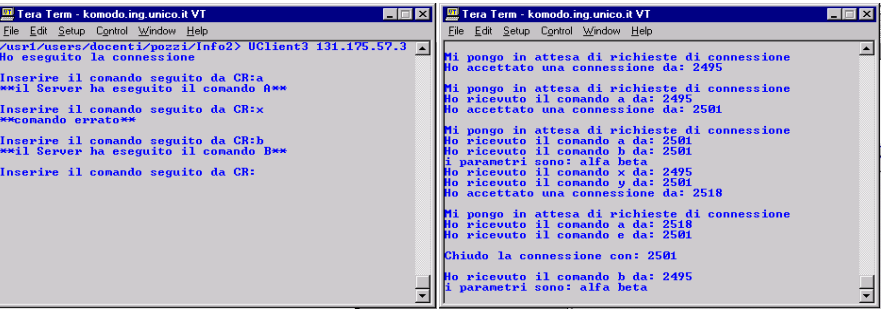
C3 esegue il comando a



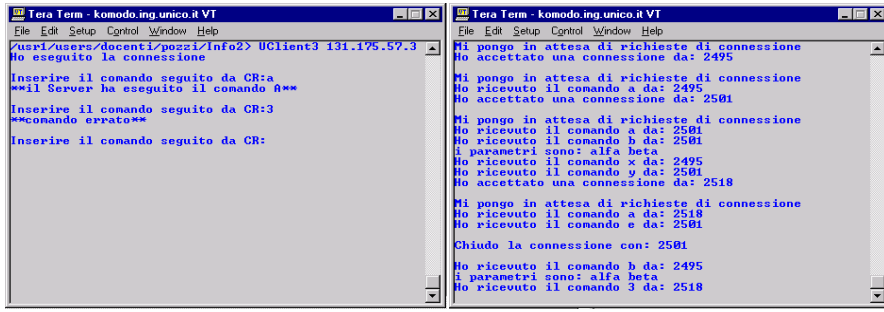
C2 chiude la connessione



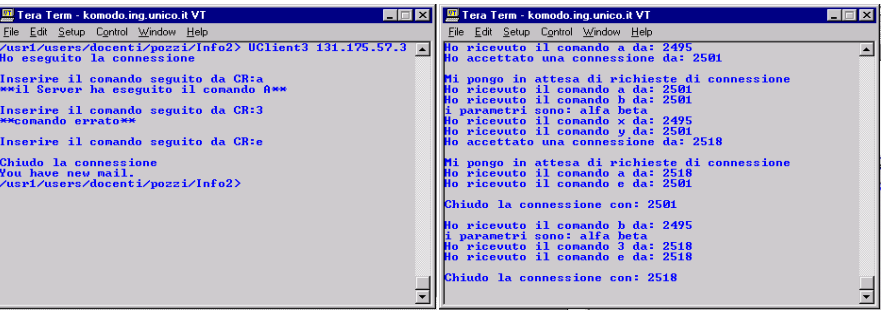
C1 esegue il comando b



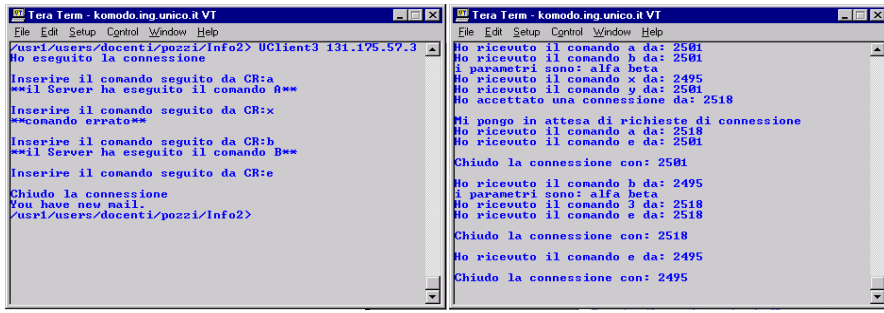
C3 esegue il comando 3



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*).

UServer5

- Rispetto a Userver4:
  - 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 execve);
  - passa al programma invocato (MainA o MainB) il descrittore del socket per comunicare col client.

Listato di UServer5 (i)

```
                    /* programma USERVER5  */
#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 UServer4 mancano */
   char terminatore[2]={'*','\n'};         /* rispostaA e rispostaB */
   char sdstring[7];                       /* nuovo rispetto a UServer4 */
   char *envp[3] = {NULL, NULL, NULL};
   int pid,my_pid,status;
```


Listato di UServer5 (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 UServer5 (iii)

```
switch (inbuf)
    {
    case 'e':break;
    case 'a':{sprintf(sdstring,"%d",new_sd);
             argv[1]=sdstring;
             pid=fork();
             if (pid==0) {execve("MainA", argv, envp);}
             else {my_pid = wait(&status); break;}}
    case 'b':{sprintf(sdstring,"%d",new_sd);
              argv[1]=sdstring;
              pid=fork();
              if (pid==0) {execve("MainB", argv, envp);}
              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 UServer5 (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);
}
```

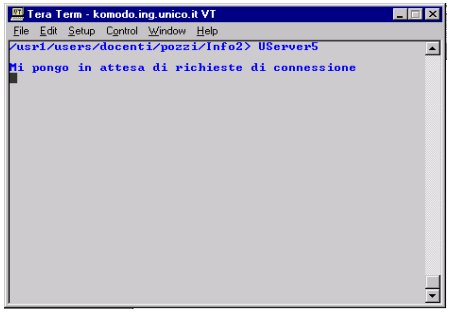

Comunicazione UClient3-MainB

- La comunicazione tra UClient3 e MainB avviene attraverso il descrittore del socket.
- UClient3 non si accorge di parlare con MainB: è convinto di parlare con UServer5.
- MainB riceve sulla linea di comando il descrittore del socket per colloquiare con UClient3.

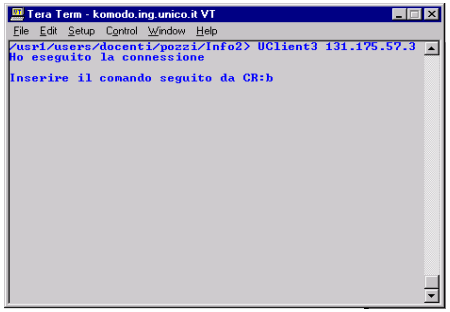
Listato di MainB

```
/* 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);
}
```

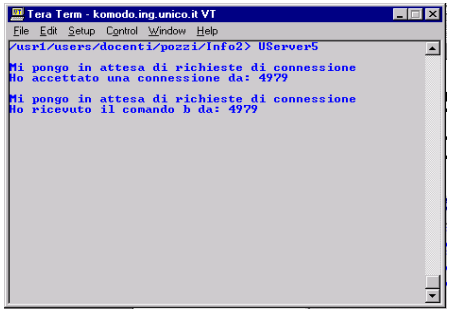

Avvio di UServer5



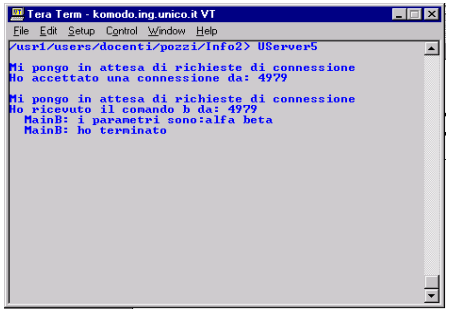
UClient3 richiede il comando e lo invia al server



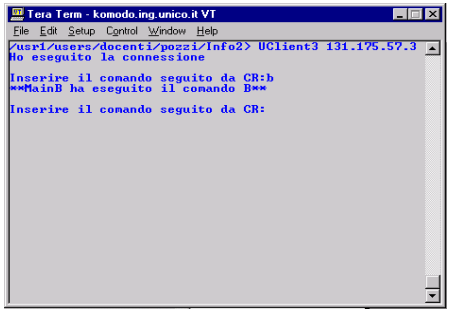
UServer5 riceve il comando b



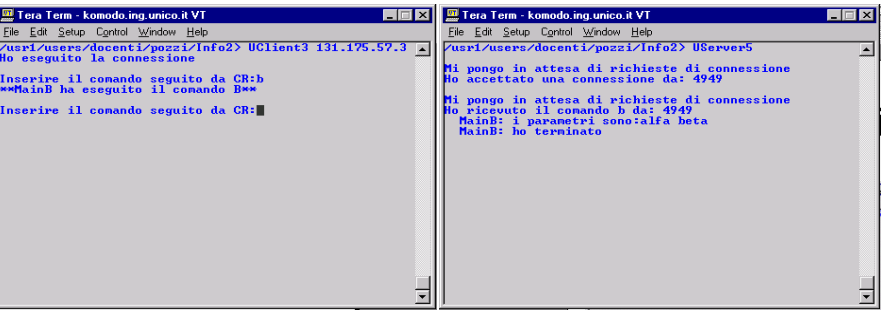
UServer5 lancia il programma MainB



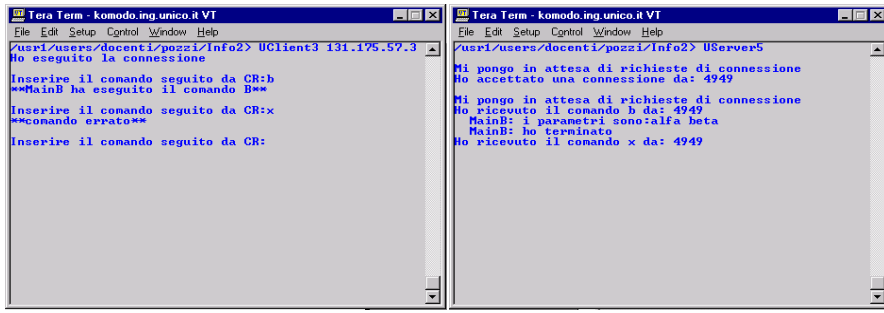
UClient3 riceve i messaggi da MainB



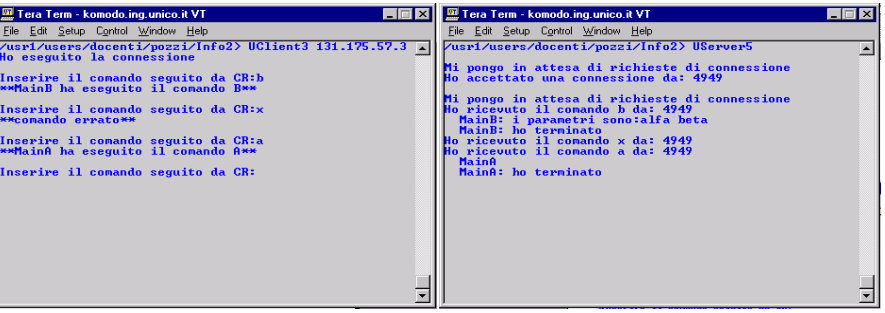
UClient3 esegue il comando b



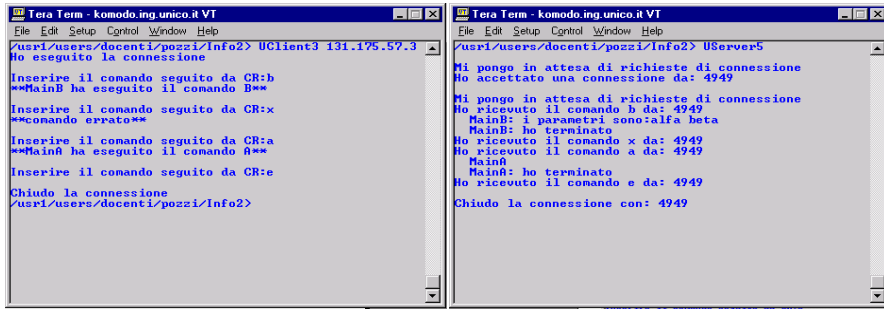
UClient3 esegue il comando x



UClient3 esegue il comando a



UClient3 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:

  ```
  $ UServer4 &
  ```

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

Fine della parte di programmazione di rete e di sistema