Meccanismi di creazione processi comuni a Windows e Unix/Linux in C/C++

Esistono varie funzioni che generano nuovi processi, lanciando programmi esistenti, i più comuni sono:

- system
- spawn
- exec

system() restituisce -1 in caso di errore, tuttavia la restituzione di 0 non implica che comando sia stato effettivamente eseguito, infatti system() esegue il comando ricevuto come parametro, esso non fa altro che lanciare una copia dell'interprete stesso, proprio come se fosse stato digitato il comando.

Quindi system() si limita a restituire 0 nel caso in cui sia riuscita a lanciare correttamente l'interprete, non solo non è possibile conoscere il valore restituito al sistema dal child, ma non è neppure possibile sapere se questo sia stato effettivamente eseguito.

Esempio di System

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>// per errno
int main()
{if(system("/bin/ls *.c ") == -1)
 fprintf(stderr,"errore %d in system()\n", errno);
 return(0);
}
```

Le funzioni della famiglia spawn...() ocome system(), consentono di lanciare programmi esterni come se fossero subroutine, tuttavia esse non fanno ricorso all'interprete dei comandi, in quanto si basano sul servizio 4Bh dell'interupt 21h.

Di conseguenza, non è possibile utilizzarle per invocare comandi interni né file batch, ma si ha un controllo sull'esito dell'operazione.

Esse infatti restituiscono -1 se l'esecuzione del child non è riuscita; in caso contrario restituiscono il valore che il programma child ha restituito a sua volta.

Tutte le funzioni spawn...() richiedono come primo parametro un intero, di solito dichiarato nei prototipi con il nome mode, che indica la modalità di esecuzione del programma child.

In PROCESS.H sono definite le seguenti costanti dette manifest:

- P_WAIT il child è eseguito come una subroutine, quindi il processo chiamante si interrompe e aspetta la terminazione del nuovo processo
- P_OVERLAY il child è eseguito sostituendo in memoria il parent, proprio come se fosse chiamata la corrispondente funzione della famiglia exec...()
- P_NOWAIT Continua l'esecuzione del processo chiamante (asynchronous **_ spawn**)
- P_DETACH Continua l'esecuzione del processo chiamante; il nuovo processo gira in background senza accesso alla console o alla keyboard.

Il secondo parametro, di tipo char *, è il nome del programma da eseguire.

Esempio di Spawn

```
/* SPAWN .C: This program accepts a number in the range 1-8 from the command line.
Based on the number it receives,  it executes one of the eight different procedures
that spawn the process named child. For some of these procedures, the CHILD.EXE file
must be inthe same directory; for  others, it only has to be in the same path. */
#include <stdio.h>
#include <process.h>
char *my_env[] =
{"THIS=environment will be","PASSED=to child.exe by
the","_SPAWNLE=and","_SPAWNLPE=and","_SPAWNVE=and","_SPAWNVPE=functions",NULL};
void main( int argc, char *argv[] )
{char *args[4]; /* Set up parameters to be sent: */
 args[0] = "child"; args[1] = " spawn ??"; args[2] = "two";   args[3] = NULL;
 if (argc <= 2){printf( "SYNTAX: SPAWN <1-8> <childprogram>\n" );exit( 1 );}
 switch (argv[1][0]) /* Based on first letter of argument */
 {case '1':_spawnl(_P_WAIT,argv[2],argv[2],"_spawnl","two",NULL); break;
  case '2':_spawnle(_P_WAIT,argv[2],argv[2],"_spawnle", "two", NULL, my_env );break;
  case '3': _spawnlp( _P_WAIT, argv[2], argv[2], "_spawnlp", "two", NULL ); break;
   case '4': _spawnlpe( _P_WAIT, argv[2], argv[2], "_spawnlpe", "two", NULL, my_env
); break;
  case '5': _spawnv( _P_OVERLAY, argv[2], args ); break;
  case '6': _spawnve( _P_OVERLAY, argv[2], args, my_env ); break;
  case '7': _spawnvp( _P_OVERLAY, argv[2], args ); break;
   case '8': _spawnvpe( _P_OVERLAY, argv[2], args, my_env ); break;
   default: printf( "SYNTAX: SPAWN <1-8> <childprogram>\n" ); exit( 1 );
 }
 printf( "from SPAWN !\n" );
}
```

Le funzioni della famiglia exec...(), a differenza delle spawn...(), non trattano il child come una subroutine del parent.

Al contrario, esso viene caricato in memoria ed eseguito al posto del parent, sostituendolo a tutti gli effetti. E' del tutto equivalente utilizzare

spawnv(P_OVERLAY,"myutil",childArgv) o

execv("myutil",childArgv);

Esempio di execv

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
char *my_env[] = /* Environment for exec */
{ "THIS=environment will be", "PASSED=to new process by","the EXEC
=functions", NULL };
void main()
{char *args[4], prog[80];
 int ch; printf( "Enter name of program to exec : " );
```

```
 gets( prog );
 printf( " 1. _execl 2. _execle 3. _execlp 4. _execlpe\n" );
 printf( " 5. _execv 6. _execve 7. _execvp 8. _execvpe\n" );
 printf( "Type a number from 1 to 8 (or 0 to quit): " );
 ch = _getche();
 if( (ch < '1') || (ch > '8') ) exit( 1 );
 printf( "\n\n" ); /* Arguments for _execv? */
 args[0] = prog; args[1] = " exec ??"; args[2] = "two";
 args[3] = NULL;
 switch( ch )
  {case '1': _execl( prog, prog, "_execl", "two", NULL ); break;
   case '2': _execle( prog, prog, "_execle", "two", NULL, my_env ); break;
  case '3': _execlp( prog, prog, "_execlp", "two", NULL ); break;
  case '4': _execlpe( prog, prog, "_execlpe", "two", NULL, my_env ); break;
  case '5': _execv( prog, args ); break;
  case '6': _execve( prog, args, my_env ); break;
  case '7': _execvp( prog, args ); break;
  case '8': _execvpe( prog, args, my_env );
  break;
  default: break;
  } /* This point is reached only if exec fails. */
  printf( "\nProcess was not execed ." ); exit( 0 );
}
```

Pipe su Unix/Linux

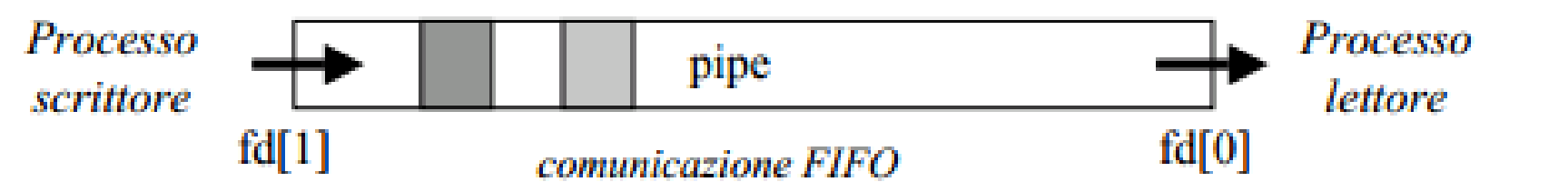
Le pipe sono canali di comunicazione unidirezionali che costituiscono un primo strumento di comunicazione (con diverse limitazioni), basato sullo scambio di messaggi, tra processi UNIX

La creazione di una pipe avviene mediante la funzione

int pipe(int fd[2])

che inizializza in fd due descrittori,

fd[0] per la lettura , fd[1] per la scrittura

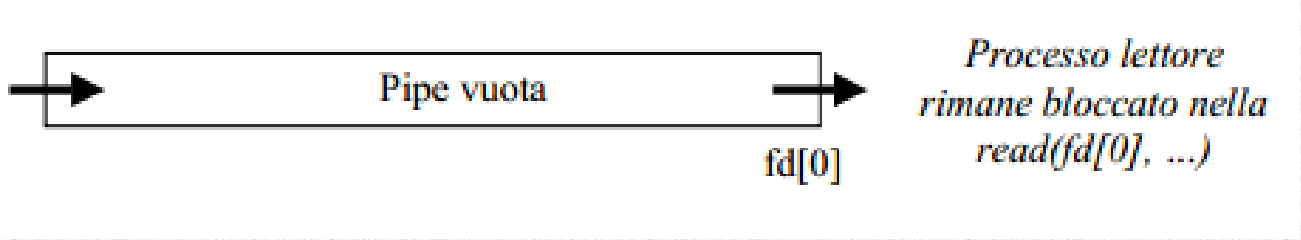


Lettura e scrittura sulla pipe sono ottenute mediante le normali primitive read e write.

I valori di ritorno delle funzione pipe sono 0 se tutto va bene, -1 se si ha un errore ed errno avrà uno dei seguenti valori:EMFILE Troppi file descriptors in uso, ENFILE La system file table è piena, EFAULT filedes non valido.

L'accesso alle pipe è regolato da un meccanismo di sincronizzazione:

- la lettura da una pipe da parte di un processo è bloccante se la pipe è vuota (in attesa che arrivino i dati)



- la scrittura su una pipe da parte di un processo è bloccante se la pipe è piena

Le pipe sono anche dette pipe anonime (unnamed pipe) perché non sono associate ad alcun nome nel File System, e quindi solo i processi che possiedono i descrittori possono comunicare attraverso una pipe.

La tabella dei file aperti di un processo (contenente i descrittori della pipe) viene duplicata per i processi figli, quindi la comunicazione attraverso una pipe anonima è quindi possibile solo per processi in relazione di parentela che condividono un progenitore.

Esempio di pipe pipe0.c

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define N_MESSAGGI 10
int main()
{int pid, j,k, piped[2];
/*Apre la pipe creando due file descriptor, uno per la lettura e l'altro
per la scrittura, vengono memorizzati nei due elementi dell'array piped*/
 if (pipe(piped) < 0) exit(1);
 if ((pid = fork()) < 0) exit(2);
 else
 if (pid == 0)  /* Il figlio eredita una copia di piped[] */
  { /* Il figlio e' il lettore dalla pipe: piped[1] non gli serve */
   close(piped[1]);
   for (j = 1; j <= N_MESSAGGI; j++)
   {read(piped[0],&k, sizeof (int));
    printf("Figlio: ho letto dalla pipe il numero %d\n", k);}
    exit(0);}
 else/* Processo padre */
  {/* Il padre e' scrittore sulla pipe: piped[0] non gli serve */
   close(piped[0]);
   for (j = 1; j <= N_MESSAGGI; j++)
    {write(piped[1], &j, sizeof (int));}
  wait(NULL);exit(0);}
}
```

Consideriamo adesso questo codice che ha come scopo generare una pipe, utilizzata dal processo figlio per inviare dati al processo padre, che poi viene chiusa quando lo scambio di dati è concluso.

bad_pipe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>
void put_into_tube(int file);
void look_into_tube(int file);
int main(int argc, char *argv[]);
// funzione per leggere dalla pipe e mostrate l'output su schermo
void look_into_tube(int file)
{FILE *fstream;
 int c;
 // converte da descrittore file a file stream per utilizzare fgetc
 fstream=fdopen(file,"r");
 /* finquando c'e' input dalla pipe mostralo sullo schermo, si blocca in
attesa se la pipe e' vuota ma aperta, in caso contrario da' EOF
 while ((c=fgetc(fstream))!=EOF) putchar(c);
 //chiude l'uscita della pipe (non raggiungera' mai questa linea di codice)
 fclose(fstream);}
```

```c
// funzione per scrivere nella pipe
void put_into_tube(int file)
{FILE *fstream;
 int i;
 time_t seed;// seme
 // converte da descrittore file a file stream per utilizzare fgetc
 fstream=fdopen(file,"w");
 // scrive qualcosa nella pipe
 fprintf(fstream,"writing\n");fprintf(fstream,"something:\n");
 seed=time(NULL);// usa la data corrente come seme
 srand(seed); // imposta il seme
 // genera un carattere random e lo immette nella pipe
 for (i=0;i<26;i++) fputc('a'+(char)(rand()%26),fstream);
 fprintf(fstream,"\n");
 fclose(fstream);// chiude l'ingresso della pipe
}

int main(int argc, char *argv[])
{int w;
 int tube[2];// tube[0] e' in lettura, mentre tube[1] in scrittura
 pid_t ok_fork;// genera la pipe
 if (pipe(tube))
  {printf("Error!File %s,line%d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
 if ((ok_fork=fork())<0)  // genera il figlio che eredita la pipe
  {printf("Error!File %s,line%d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
 printf("Pid: %d\n",getpid());
 if (!ok_fork)
   {close(tube[0]);//il figlio chiude tube[0], deve solo scrivere
```

```
    put_into_tube(tube[1]);// e scrive nella pipe}
  else
   {// il padre si dimentica di chiudere il canale in scrittura
    // (tube[1]) e inizia a leggere dalla pipe
    look_into_tube(tube[0]);
    // anche se non ci arrivera' mai, qui dovrebbe rilasciare
    // le risorse sul vettore di status del figlio
    while (1)
     {wait(&w);
      if (WIFEXITED(w)) break; }
        // e attendere in input da tastiera
    getchar();}
  exit(EXIT_SUCCESS); }
```

Il programma non funziona perché:

- il padre genera una pipe e successivamente un figlio che la eredita.
- Il padre legge dalla pipe mentre il figlio ci scrive.
- Quando non ci sono più dati il padre dovrebbe terminare, ma poiché il padre stesso non ha chiuso l'estremità in scrittura della pipe crederà che il figlio sia ancora attivo e attenderà invano (o quantomeno finché darete energia al pc).

Invece dello file stream (FILE *) vengono impiegati i descrittori a file (int) nella pipe. I dati vengono generati in maniera random, e il seme impostato in base alla data corrente per evitare che ad ogni esecuzione i dati forniti dal figlio siano identici.

Il padre non raggiungerà mai la parte relativa al rilascio delle risorse sullo status del figlio.

Il codice corretto è il seguente

pipe.c

```
int main(int argc, char *argv[])
{int w;
 // tube[0] e' in lettura, mentre tube[1] in scrittura
 int tube[2];
 pid_t ok_fork;// genera la pipe
 if (pipe(tube))
  {printf("Error !!! File %s, line %d\n",__FILE__,__LINE__);
   exit(EXIT_FAILURE);}
 // genera il figlio che eredita la pipe
 if ((ok_fork=fork())<0)
```

```
  {printf("Error !!! File %s, line %d\n",__FILE__,__LINE__);
    exit(EXIT_FAILURE);}
  printf("Pid: %d\n",getpid());
  if (!ok_fork)
   {close(tube[0]);//il figlio chiude tube[0], deve solo scrivere
    put_into_tube(tube[1]);// e scrive nella pipe}
  else
   {close (tube[1]);
    look_into_tube(tube[0]);
    while (1)
     {wait(&w);
      if (WIFEXITED(w)) break; }
        // e attendere in input da tastiera
    getchar();}
  exit(EXIT_SUCCESS);
 }
```

Cosa succede quando condivido una pipe con più di un figlio?
Nel seguente codice un padre genera una pipe e successivamente i due figli la ereditano.
Il padre legge dalla pipe mentre i due figli scrivono.
Quando non ci sono più dati il padre termina.

pipe2.c

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

void put_into_tube(int file);
void look_into_tube(int file);

// funzione per leggere dalla pipe e mostrate l'output su schermo
void look_into_tube(int file)
{FILE *fstream;
 int c;
 fstream=fdopen(file,"r");
 while ((c=fgetc(fstream))!=EOF)
   putchar(c);
 fclose(fstream);
}

// funzione per scrivere nella pipe
void put_into_tube(int file)
{FILE *fstream;
 int i;
 time_t seed; // seme
 fstream=fdopen(file,"w");
 // scrive qualcosa nella pipe
```

```
 fprintf(fstream,"writing\n");
 fprintf(fstream,"something:\n");
 // usa la data corrente piu' l'id univoco del prcesso come seme
 // altrimenti visto che i figli vengono  creati nell'arco dello
 // stesso secondo in genere, entrambi i figli scriverebbero gli
 // stessi dati.
 seed=time(NULL)+getpid();
 srand(seed);   // imposta il seme
 // genera un carattere random e lo immette nella pipe
 for (i=0;i<26;i++) fputc('a'+(char)(rand()%26),fstream);
 fprintf(fstream,"\n");
 fclose(fstream); // chiude l'ingresso della pipe
}

int main(int argc, char *argv[])
{int i,w;
 // pipe tube[0] ingresso in lettura, tube[1] uscita, in scrittura
 int tube[2];
 pid_t ok_fork;
 if (pipe(tube))
 {printf("Error!!!File %s, line %d\n",__FILE__,__LINE__); exit(EXIT_FAILURE);}
 // genera il primo figlio che eredita la pipe
 if ((ok_fork=fork())<0)
 {printf("Error!!File %s,line %d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
 // se sono il padre genero il secondo figlio
 if (ok_fork)
  if ((ok_fork=fork())<0)
   {printf("Error!!File %s,line %d\n",__FILE__,__LINE__);exit(EXIT_FAILURE);}
```

```
  printf("Pid: %d\n",getpid());
  if (!ok_fork)
    {close(tube[0]); //il figlio chiude il canale lettura (deve solo scrivere)
     put_into_tube(tube[1]); // e scrive nella pipe
    }
   else
    {close (tube[1]); //il padre chiude il canale scrittura(deve solo leggere)
     look_into_tube(tube[0]); // e legge dalla pipe
     // qui rilascia le risorse sul vettore di status del figlio
     i=0;
     while (i!=2)
     {wait(&w);
      if (WIFEXITED(w)) i++;
     }
     getchar();// e attende in input da tastiera
    }
  exit(EXIT_SUCCESS);
}
```

In teoria vista la concorrenza dei figli nello scrivere nella pipe ci si aspetta che i dati si mescolino.

Ciò non avviene per la bufferizzazione che fa sì che i dati vengano scritti sulla pipe quando il buffer è pieno o alla chiusura

del processo (nel caso di scrittura da terminale anche dopo aver pressato invio).
Per "OVVIARE" a tale problema vediamo di forzata la scrittura dentro la pipe (l'output varia da esecuzione ad esecuzione) modificando la funzione **`put_into_tube(int file)`**

pipe2_messy.c

```
// funzione per scrivere nella pipe
void put_into_tube(int file)
{FILE *fstream;
 int i;
 time_t seed; // seme
 fstream=fdopen(file,"w");
 fprintf(fstream,"writing\n");
 fprintf(fstream,"something:\n");// scrive qualcosa nella pipe
 seed=time(NULL)+getpid();
 srand(seed); // imposta il seme
 for (i=0;i<26;i++) // genera un carattere random e lo immette nella pipe
  {fflush(fstream); // forza la scrittura nella pipe ora
   fputc('a'+(char)(rand()%26),fstream);}
 fprintf(fstream,"\n");fclose(fstream); // chiude l'ingresso della pipe
}
```