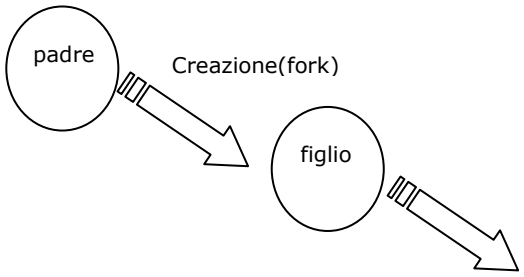
**I PROCESSI IN UNIX**

Unix è un sistema multiprogrammato e multitasking per cui processi diversi tra loro ed indipendenti possono essere eseguiti allo stesso tempo. Ogni utente può avere diversi processi attivi in un certo momento, per cui su di un sistema ci possono essere centinaia di processi in esecuzione. Infatti, su molte workstation a singolo utente, anche quando questi è assente, girano dozzine di processi in background, detti **demoni**, che partono autonomamente quando il sistema è attivato.
Un demone tipico è il demone *cron* che si attiva ogni minuto per controllare se c'è qualche lavoro da fare, nel qual caso lo svolge, ritornando a fermarsi fino al prossimo controllo.

I processi in Unix sono creati in modo particolarmente semplice tramite la chiamata alla primitiva **fork().**
Questa primitiva crea un processo figlio (child) che ha lo stesso identico codice del processo che invoca la primitiva, che viene perciò chiamato anche processo padre (father).




:::::è possibile realizzare gerarchie di processi

Subito dopo l'esecuzione della primitiva fork(), dunque, esistono due processi, ognuno con il suo spazio di memoria privato. Sia padre che figlio (in caso di corretta terminazione della syscall) continueranno la loro esecuzione da dopo la fork(): tale syscall verrà quindi chiamata una volta e ritornerà due volte (una nel processo padre ed una nel processo figlio). Il genitore ed il figlio hanno un'immagine di memoria privata, di conseguenza se il genitore cambia una delle sue variabili questa modifica non è visibile al figlio e viceversa

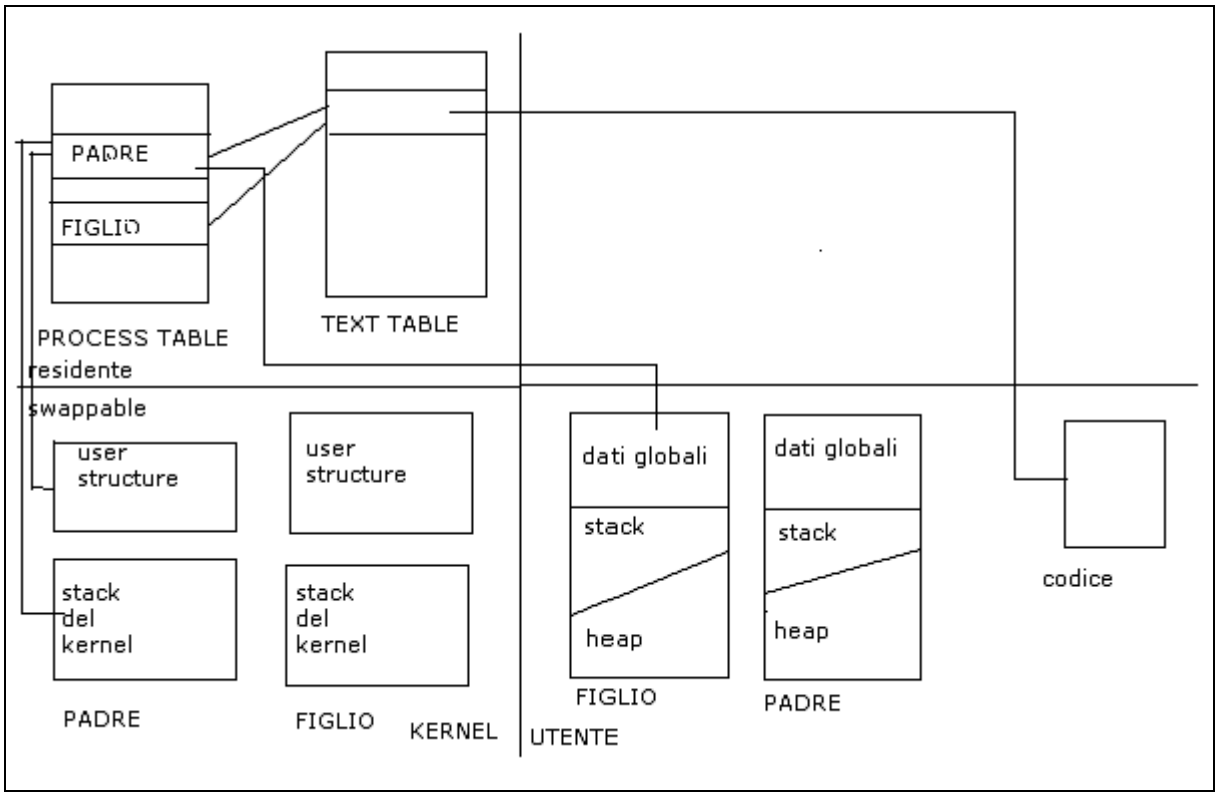
Il processo figlio è quindi una copia quasi identica del processo padre. In particolare, la fork() esegue le seguenti operazioni:

1. crea un nuovo spazio di memoria privato destinato a contenere il processo figlio;
2. crea un nuovo descrittore di processo all'interno del nucleo del sistem operativo;
3. assegna un nuovo PID al processo figlio (infatti, come detto, il PID deve identificare univocamente un nuovo processo);
4. esegue una copia quasi fedele della memoria del padre nella memoria del figlio, sia per quanto riguarda la parte dei dati, che per quanto riguarda il codice;
5. a seconda della politica di schedulazione del sistema operativo, uno dei due processi, il padre o il figlio, andrà in esecuzione, mentre l'altro resterà in attesa di essere eseguito dal processore; nei sistemi con multiprocessori, entrambi i processi possono andare in esecuzione contemporaneamente su due processori diversi;
6. ritorna due volte: nel processo padre, la primitiva **fork()** ritorna il valore del PID assegnato al processo figlio; nel processo figlio, ritorna **0.**

Nella figura si possono vedere gli effetti di una fork() sulle struttre dati associate al processo padre e figlio.

- Allocazione di una nuova process structure nella process table associata al processo figlio e sua inizializzazione
- Allocazione di una nuova user structure nella quale viene copiata quella del padre
- Allocazione dei segmenti dati e stack del figlio nei quali vengono copiati dati e stack del padre
- Aggiornamento della text structure del codice eseguito(condiviso col padre): incremento del contatore dei processi.

**Immagine di in processo ed effetto della fork()**




Dove gli elementi descritti rappresentano l'immagine di un processo:

1. **PROCESS TABLE**

CONTIENE INFORMAZIONI NECESSARIE AL SISTEMA PER LA GESTIONE DEL PROCESSO (A PRESCINDEDRE DALLO STATO DEL PROCESSO). SI TORVA NEL KERNEL ED è RESIDENTE.
CONTIENE UNA ENTRY POINT PER OGNI PROCESSO ED è DIMENSIONATE STATICEMENTE AL MOMENTO DELLA CONFIGURAZIONE DEL SISTEMA
PER OGNI PROCESSO CONTIENE LE INFORMAZIONI CHE NE PERMETTONO LA SCHEDULAZIONE E CHE DEVONO ESSERE SEMPRE RESIDENTI
GLI ELEMENTI DELLA PROCESS TABLE sono descrittori del processo chiamati ***Process Structure*** contenenti:

- √ process identifier(PID)
- √ process identifier del padre (PPID)
- √ stato del processo
- √ puntatori alle varie aree dati e stack associati al processo
- √ informazioni sullo scheduling (tempi di CPU etc)
- √ **riferimento indiretto al codice:la process structure contiene il riferimento all'elemento della text table associato al codice del processo**
- √ informazione relative alla gestione dei segnali
- √ puntatore alla user structure
- √ puntatori a processi successivi in code di scheduling(ready queue)

2. **TEXT TABLE**

E' una struttura dati globale gestita dal S.O in cui sono contenuti i puntatori ai codici utilizzati ed eventualmente condivisi dai processi. Ogni elemento della text table si chiama text structure e contiene ad esempio il puntatore al codice(se il processo è swappato riferimento alla memoria secondaria), numero di processi che lo condividono

3. **USER STRUCTURE**

CONTIENE INFORMAZIONI NECESSARIE SOLO SE IL PROCESSO è RESIDENTE IN MEMORIA CENTRALE(struttura dati contenente dati necessari al kernel per la gestione del processo )

- √ Copia registri Cpu
- √ Informazioni sulle risorse allocate(file aperti)
- √ Informazioni sulla gestione dei segnali
- √ Ambiente del processo :direttorio corrente, utente, gruppo, argc/argv, path

***N.B. PROCESS SRUCTURE + USER STRUCTURE =PCB***

4. **STACKE DEL KERNEL**

stack di sistema associato al processo per le chiamata di sistema(es. exec)

5. **STACK E HEAP DI UTENTE**

aree dinamiche associate al programma eseguito

*6.* **AREA DATI GLOBALE UTENTE** contiene la variabili globali del programma eseguito dal processo

Il punto 4 è molto importante e merita di essere approfondito meglio con un esempio. Se il processo padre ha dichiarato una variabile intera "a" il cui valore subito prima di eseguire la fork() e' 5, subito dopo la fork() sia il padre che il figlio possiedono una variabile a = 5. Però, da questo momento in poi, le due variabili sono completamente distinte, una sta nello spazio privato del processo padre e l'altra sta nello spazio privato del processo figlio, e avranno vita diversa. Quindi, se successivamente il processo padre esegue l'istruzione a++, solo la sua variabile a verrà incrementata, mentre la variabile a che sta nel processo figlio conserverà il valore originale.

I file aperti sono condivisi tra genitore e figlio, ovvero se un certo file era aperto per il genitore prima del fork(), esso continuerà ad essere aperto per entrambi anche dopo.

Il fatto che le immagini di memoria, le variabili, i registri e qualsiasi altra cosa sono identiche per il genitore ed il figlio porta ad una piccola difficoltà: in che modo i processi possono sapere quale di essi dovrà girare con il codice del padre e quale con quello del figlio? Il segreto sta nel fatto che la chiamata di sistema fork restituisce uno **0** per il figlio ed un numero **>0**(maggiore di zero) per il padre, detto **PID**(*process identifier*); se la fork() restituisce un numero **<0** allora la fork() non è stata eseguita con successo.
I processi sono denominati con i loro PID. Un processo può conoscere il proprio PID attraverso la chiamata di sistema **GETPID()**.

***1° Esempio(a)***
Creazione di due copie dello stesso processo ed esecuzione di entrambe contemporaneamente (multitasking)

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>

 int main()
{
        cout≪("Prima del FORKING…\n";
        fork();
        cout≪"Il process ID del processo"≪ getpid()≪"\n";
}
```

La funzione getpid() individua il PID del processo di appartenenza.
*L'output sarà il seguente*:
Prima del FORKING
Il process ID del processo è 1160
Il process ID del processo è 1161

N.B.: **I processi hanno PID unici**, che risulteranno diversi ad ogni esecuzione

***1° Esempio(b)***

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        int id_processo;
        cout << "prima del forking….\n";
        id_processo=fork();
        if (id_processo==-1)
        {
                cout<<"impossibile eseguire il fork!!!!!\n";
                return 1;
        }
                if (id_processo==0)
                {
                cout<<"Sono nel processo figlio\n";
                cout<<"il mio PID è "<<getpid()<<"\n";
                }
                else
                {
                cout<<"Sono nel processo padre\n";
                cout<<"il mio PID è "<<getpid()<<"\n;
                }
}
```

***2° Esempio***

```
#include <iostream.hh>
#include <sys/types.h>
#include <unistd.h>
```

```
int main()
{
	int process_pid = 0;
	sleep(2);
	process_pid = fork();
	if (process_pid==0)
	{
		cout≪"in figlio\n";
		sleep(10);
		cout≪"fine figlio\n";
	}
	else
	{
		cout"in padre...";
		cout≪"PID figlio  "≪process_pid<"\n";
		sleep(10);
		cout≪"fine padre\n";
	}
return 0;
}
```

### *3° Esempio*

In questo esempio vedremo la generazione di più processi figli dello stesso processo padre

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>

#define MAXPROC		10

int main()
{
	int i, process_pid[MAXPROC];

	cout≪"Sono il processo PADRE, PID = " ≪getpid() ≪"\n";

	for (i=0;i<MAXPROC;i++)
	{	process_pid[i]=fork();
		if (process_pid[i] == -1)
		{
			cout≪"Fork non possibile!\n";
			return 1;
		}

		if(process_pid[i] == 0)
		{
			cout≪"Sono il processo figlio n° "≪ i+1≪" di PID "≪ getpid()≪"\n";
			break;
		}
	}
}
```

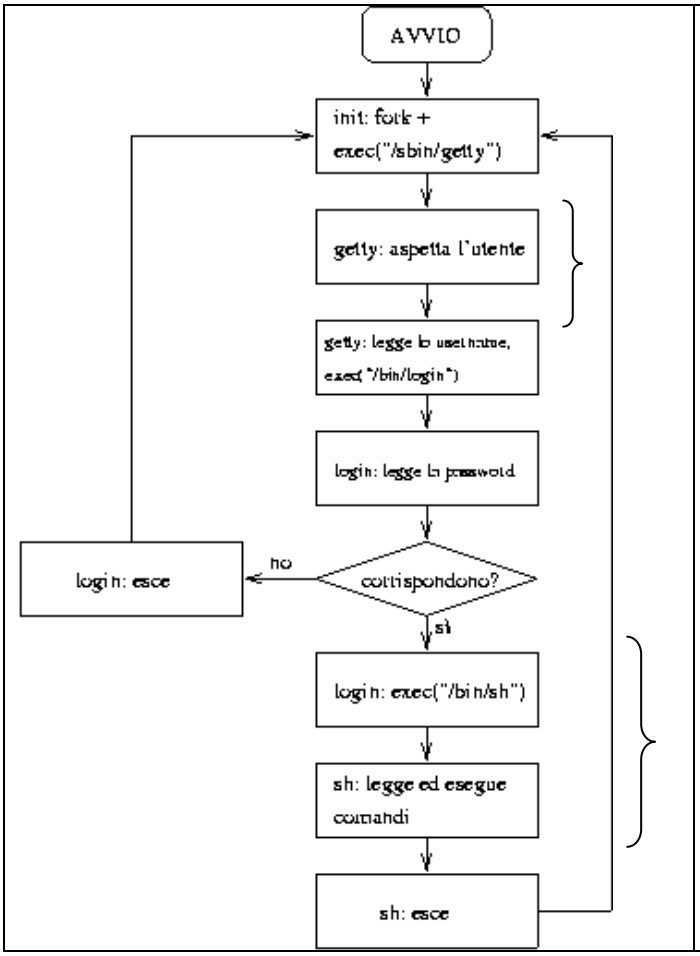
### *4°Esempio*

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        int id_processo;
        int condivisa = 0;
        cout << "prima del forking….\n";
        id_processo=fork();
        if (id_processo==-1)
        {
                cout<<"impossibile eseguire il fork!!!!!\n";
                return 1;
        }
        do{
                if (id_processo==0)
                {
                cout<<"Sono nel processo figlio\n";
                cout<<"la variabile condivisa vale "<<condivisa<<"\n";
                }
                else
                {
                cout<<"sono nel processo padre\n";
                condivisa++;
                cout<<"la variabile condivisa vale "<<condivisa<<"\n;
                }
        }while(1);
}
```

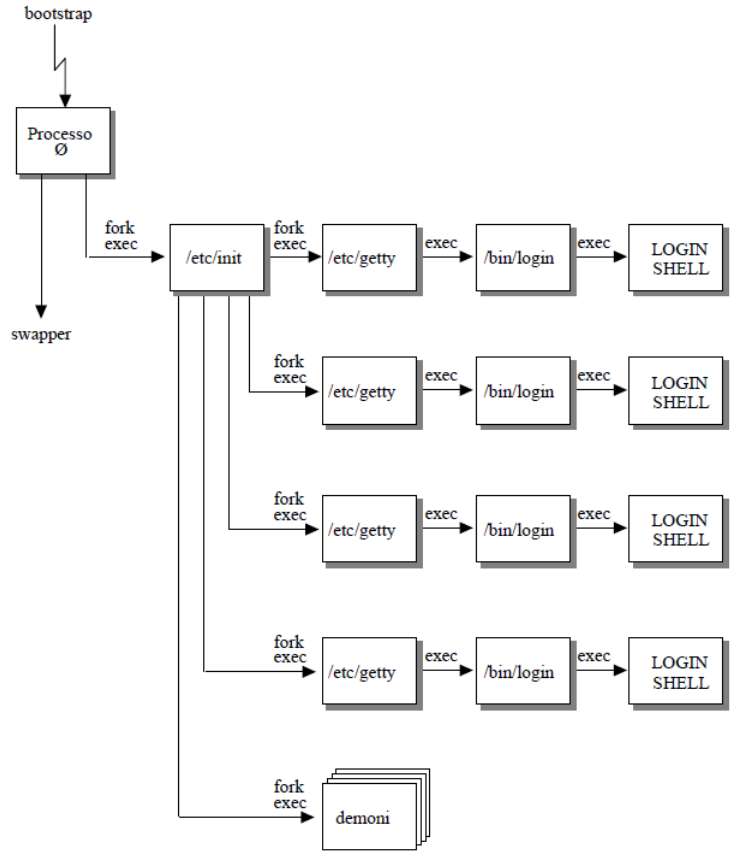
**Esercizio.** - Scrivere un programma che dato un numero N>1 forka un figlio, il quale a sua volta forka un altro figlio, e così via, fino ad ottenere N processi in tutto.

Un figlio può generare a sua volta dei figli che a loro volta creeranno figli generando un albero di figli nipoti e discendenti futuri. Questa capacità di formare un albero di processi è la chiave per capire come funziona il timesharing di Unix.
Quando il kernel si è avviato(cioè quando è stato caricato in memoria, si è inizializzato e ha inizializzato tutti i driver dei device, le strutture dati e cose del genere) finisce la sua parte di boot facendo partire un programma a livello utente:**init**. Init è il primo processo (il suo PID è 1).
Quando **init** parte completa il processo di **boot** eseguendo una **getty** per ogni terminale a cui un utente può collegarsi




Getty ascolta al terminale ed aspetta che l'utente sia pronto a fare il login premendo un tasto

Avvia la shell opportunamente configurata



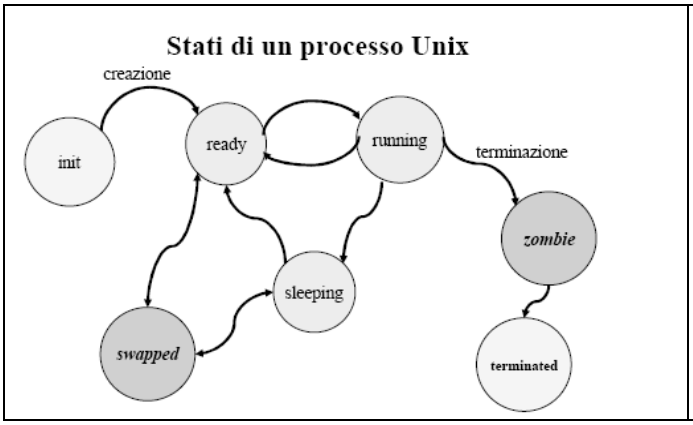
## Terminazione di un processo

Una volta che il processo figlio è in esecuzione questo procede e termina indipendentemente dal padre. Il sistema Unix prevede due situazioni:

- il processo figlio termina prima del processo padre(zombie)
- Il processo padre termina prima del processo figlio(orfano)

Nel primo caso il processo figlio non viene tolto dalla tabella dei processi ma esso manda un segnale al processo padre per riconoscere il completamento. Solo allora viene tolto dalla tabella dei processi. Nel periodo che intercorre tra il termine dell'esecuzione del processo figlio e la sua eliminazione si dice che il processo figlio è nello stato **"zombie**". Se un processo padre non è scritto correttamente il figlio può essere "zombie" per sempre.( Vedere comandi Unix ps e kill). Nel secondo caso si dice che il processo figlio è "orfano". In entrambi i casi il processo figlio rimane nalla tabella dei processi fino a che il processo init non lo prende in esame. Il processo figlio diventa figlio del processo **init** che lo monitora e lo toglie dalla tabella dei processi.

In Unix gli stati di un processo possono essere così definiti:




**Stati di un processo Unix**

**Come nel caso generale:**

- Init: caricamento in memoria del processo e inizializzazione delle strutture dati del S.O.
- Ready: processo pronto
- Running: il processo usa la CPU
- *Sleeping*: il processo è sospeso in attesa di un evento
- Terminated: deallocazione del processo dalla memoria.

**In aggiunta:**

- ***Zombie***: il processo è terminato, ma è in attesa che il padre ne rilevi lo stato di terminazione.
- ***Swapped***: il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria.

Per evitare queste anomalie di funzionamento è opportuno sincronizzare il processo figlio con il processo padre attraverso le due system calls:

1. **void exit(int status);**
   - √ la funzione exit prevede un parametro (status) mediante il quale il processo che termina può comunicare al padre informazioni sul suo stato di terminazione (es. l'esito della sua terminazione)
   - √ è sempre una chiamata senza ritorno
   - √ gli effetti di una exit sono la chiusura dei file aperti non condivisi e la terminazione del processo
     - se il processo che termina ha figli in esecuzione il processo INIT adotta i figli dopo la terminazione del padre(nella process structure del figlio al pid del processo padre viene assegnato 1 (pid di init))
     - se il processo termina prima che il padre ne rilevi lo stato di terminazione (con la system call call wait) il processo passa nello stato zombie

2. **int wait(int *status);**
   - √ lo stato di terminazione del processo può essere rilevato dal processo padre mediante la system call wait
   - √ il parametro status è l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio
   - √ il risultato prodotto dalla wait è il pid del processo terminato oppure un codice di errore(<0)
   - √ il processo che chiama la wait può avere figli in esecuzione con gli effetti di :
     - se tutti i figli non sono ancora terminati il processo si sospende in attesa che termini uno di essi
     - se almeno un figli è già terminato ed il suo stato non è ancora stato rilevato (cioè è in stato di zombie)la wait ritorna immediatamente con il suo stati di terminazione(nella variabile status)
     - se non esiste neanche un figlio la wait non è sospensiva e ritorna un codice di errore(codice<0).

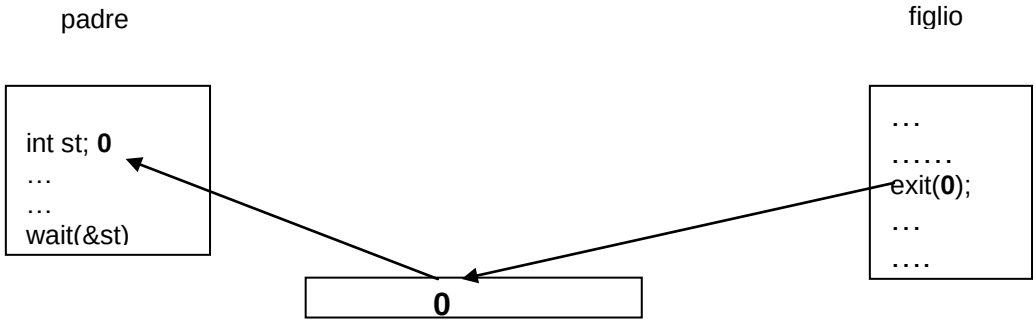


Tabella dei processi del figlio

St è l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio

**Per evitare di dover scrivere il corpo dei due processi nello stesso file, Unix mette a disposizione la famiglia di funzioni exec().**

Come detto, la primitiva fork() permette di creare un nuovo processo, copiando nel suo spazio di memoria lo spazio di memoria del processo creante, e questo obbliga a inserire il codice ed i dati di tutti i processi nello stesso eseguibile, con conseguente aumento delle dimensioni degli eseguibili, complicazione della struttura del programma e perdita di flessibilità.
Unix permatte di differenziare il codice di due processi madiante una system call della famiglia exec().
Ognuna delle funzione della famiglia, sebbene con modalità leggermente differenti si occupa di sostituire i dati del processo che chiama la system call con codice e dati di un programma specificato come parametro della system call.
È da notare che una funzione della famiglia exec() non crea un nuovo processo, ma semplicemente modifica lo spazio di memoria del processo che la chiama; per questo motivo, **in caso di corretto funzionamento la exec() non ritornerà**. È quindi chiaro che se la syscall ritorna un valore al chiamante, tale valore è necessariamente minore di 0 (tipicamente -1). A livello intuitivo, una exec() è paragonabile ad una jmp nel codice di un altro file oggetto, che rimpiazza il primo nella memoria privata del processo.
La famiglia di funzioni exec è composta da execl(), execlp(), execle(), execv(), execvp() ed execve().
Con la funzione execl(), execute and leave,un processo sara' eseguito e poi terminato dalla stessa execl.
Viene definita con:
**execl(char *path, char *arg0, ..., char *argn, 0);**

L'ultimo parametro deve sempre essere 0. E' un termine NULL. Poiche' la lista degli argomenti e' variabile, e' necessario avere un modo per dire al C dove termina; il termine nullo serve a questo.
**"path**" punta al nome di un file contenente un comando che deve essere eseguito, **"argo"** punta ad una stringa che e' la stessa di "path" (o almeno il suo ultimo componente).
**"arg1, ..., argn"** sono puntatori agli argomenti per il comando, e 0 segna semplicemente la fine dell'elenco (di lunghezza variabile) degli argomenti.

***1° Esempio uso execl()***

```
#include <iostream.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void)
{
  execl("/usr/bin/ddate","ddate",0);
  sleep(2);
  for(int k=-32768;k<=32767;k++)
       cout<<"k vale "<<k<<"\n";
}
```

***2° Esempio uso execl()***

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

int main(void)
{
  cout<<"Attendi 10 secondi...\n";
  sleep(10);
  execl("/…/piu_figli.exe","piu_figli.exe","4",0);
  //
  // La parte che segue viene eseguita SOLO se il comando execl fallisce.
    // Questo perchè la execl "passa" lancia il comando specificato assegnandogli il
  // proprio PID
  //
  // Per provarlo, lanciare questo programma e, in un'altra finestra, usare
  //
  //  ps-ef ed individuare il PID di questo processo.
  //
  // Quando pi_figli.exe viene lanciato, riprovare il comando ps -ef e verificare
  // che il PID precedentemente assegnato a questo programma è ora associato
  // a piu_figli.exe.
   //
  for(int k=0;k<=10;k++)
      {
       cout<<"k vale "<<k<<"\n";
       sleep(1);
```

***programma più_figli.exe***

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
int res,n,count,argomento,caso,rita;
if(argc!=2)
  {
    cout<<"Parametri errati:\n";
    cout<<" piu_figli <numerodeifigli>\n";
  }
else
  {
    argomento=atoi(argv[1]);
    for(count=1;count<=argomento;count++)
    {
     n=fork();
     if(n==0)
       {
         cout<<"Sono il figlio "<<getpid()<<"\n";
         srandom(getpid());
         caso=random()%20;
         cout<<"attendero' "<<caso<<" secondi, poi terminero'\n";
         for(rita=caso;rita>0;rita--)
              {
               sleep(1);
               cout<<"Figlio "<<getpid()<<" ancora "<<rita<<" secondi di vita\n";
              }
        exit(0);
       }
    }
    for(count=1;count<=argomento;count++)
    {
     res=wait(0);
     cout<<"\nSono il padre, e' terminato "<<res<<"\n\n";
    }
  }
```