PROCESSI IN UNIX

In UNIX ogni attività è mappata in un processo (con le parti che abbiamo visto) e con uno spazio di indirizzamento solo locale

MODELLO DI PROCESSI IN UNIX

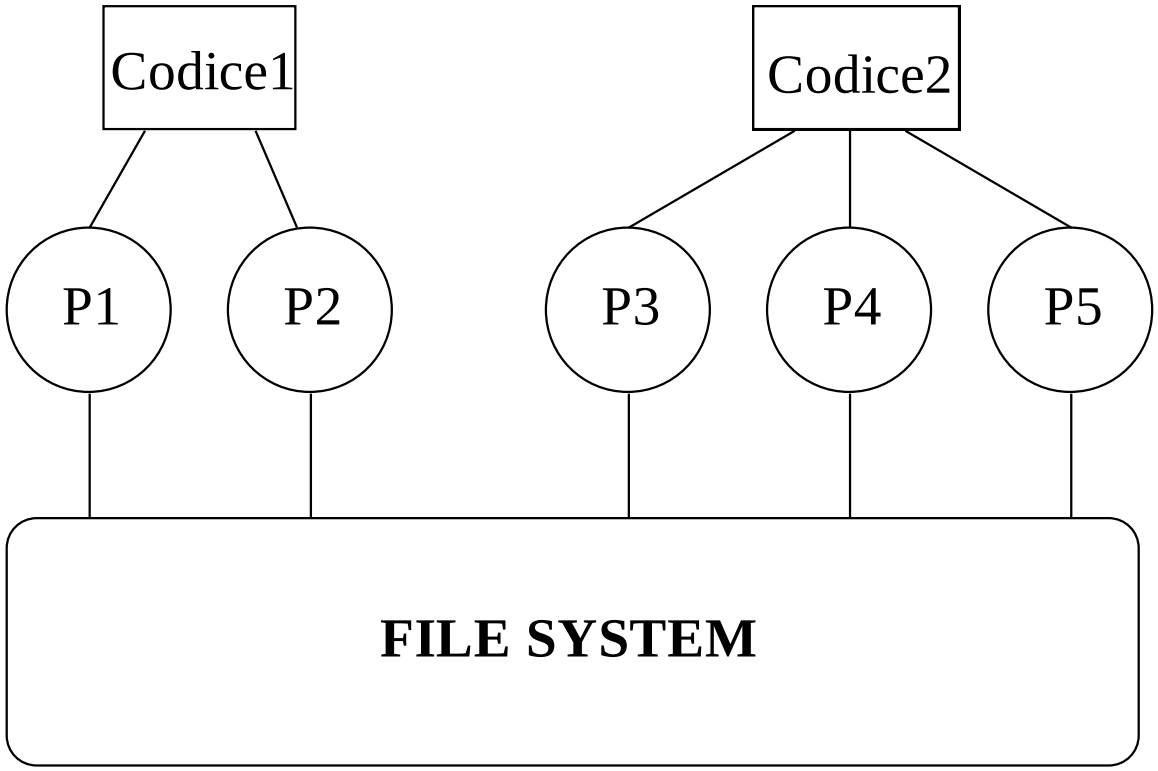
Ogni processo ha un proprio spazio di indirizzamento completamente locale e non condiviso

====> **MODELLO AD AMBIENTE LOCALE**

⇨ **PROTEZIONE** e **CORRETTEZZA**

ECCEZIONE:

* il file system è l'unica risorsa condivisa



OTTIMIZZAZIONE:

* il codice NON viene replicato nel caso di processi in parentela!

PROCESSI IN UNIX (segue)

* un **PROCESSO** per ogni **UTENTE**
* più **PROCESSI** per singolo **UTENTE**
* più **PROCESSI** di **SISTEMA**

I processi di UNIX sono tutti **rientranti**: sono rientranti più processi che eseguono insieme lo stesso codice senza causare problemi ⇨ il codice eseguito è rientrante

Processi rientranti possono condividere il codice a patto che a ciascuno sia associata un'area dati separata

Esempio di codice rientrante:
CODICE PURO ⇨ separato dai dati e che non si modifica

I processi sono gestiti a **divisione di tempo** con **più livelli di priorità variabile** (round-robin nell'ambito della stessa priorità e aging sulla priorità)

La gestione della memoria prevede l'uso di **memoria virtuale** integrata con **swapping**

⇨ lo **swapping** comporta che processi non necessari possono essere soggetti a swap (cioè vengono portati fuori dalla memoria centrale e copiati in memoria secondaria)

MODI di esecuzione di un processo

Un processo può eseguire in due **modi** diversi:

* "processo di utente" (**user process**)
  ⇨ modo di esecuzione NORMALE
* "processo di sistema" (**kernel process**)
  ⇨ modo di esecuzione con MAGGIORE VISIBILITÀ

La transizione tra i "modi" **user** e **kernel** avviene mediante l'invocazione delle primitive

⇨ *La esecuzione delle primitive avviene in stato kernel*

Kernel di Linux (versione 2.0)

Kernel

⇨ parte del S.O. che si occupa dei servizi di base

Esegue in **system mode** (modo protetto: livello 0 su processori della Famiglia x86, dal 386 in poi)

Mette a disposizione servizi
- system call
- routine di risposta a interrupt

Kernel monolitico

Processi e task

Punto di vista del kernel

task = entità con dati e codice
parte dei task esegue in user mode

Punto di vista esterno

processi parte dei task eseguita in **user mode** (livello 3 in x86)

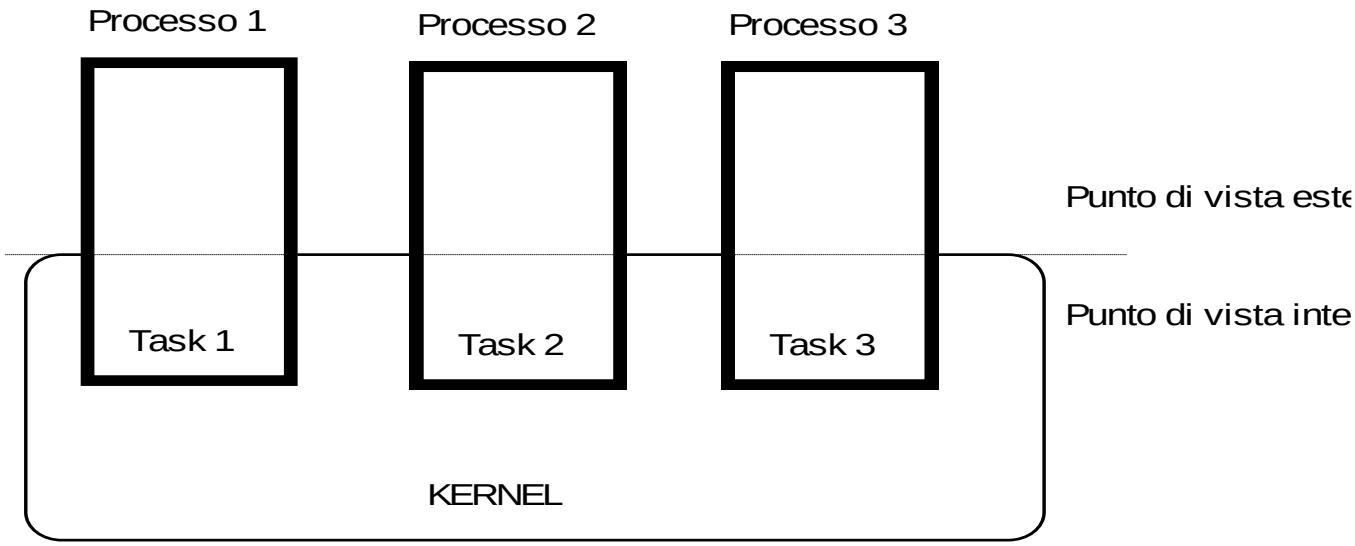
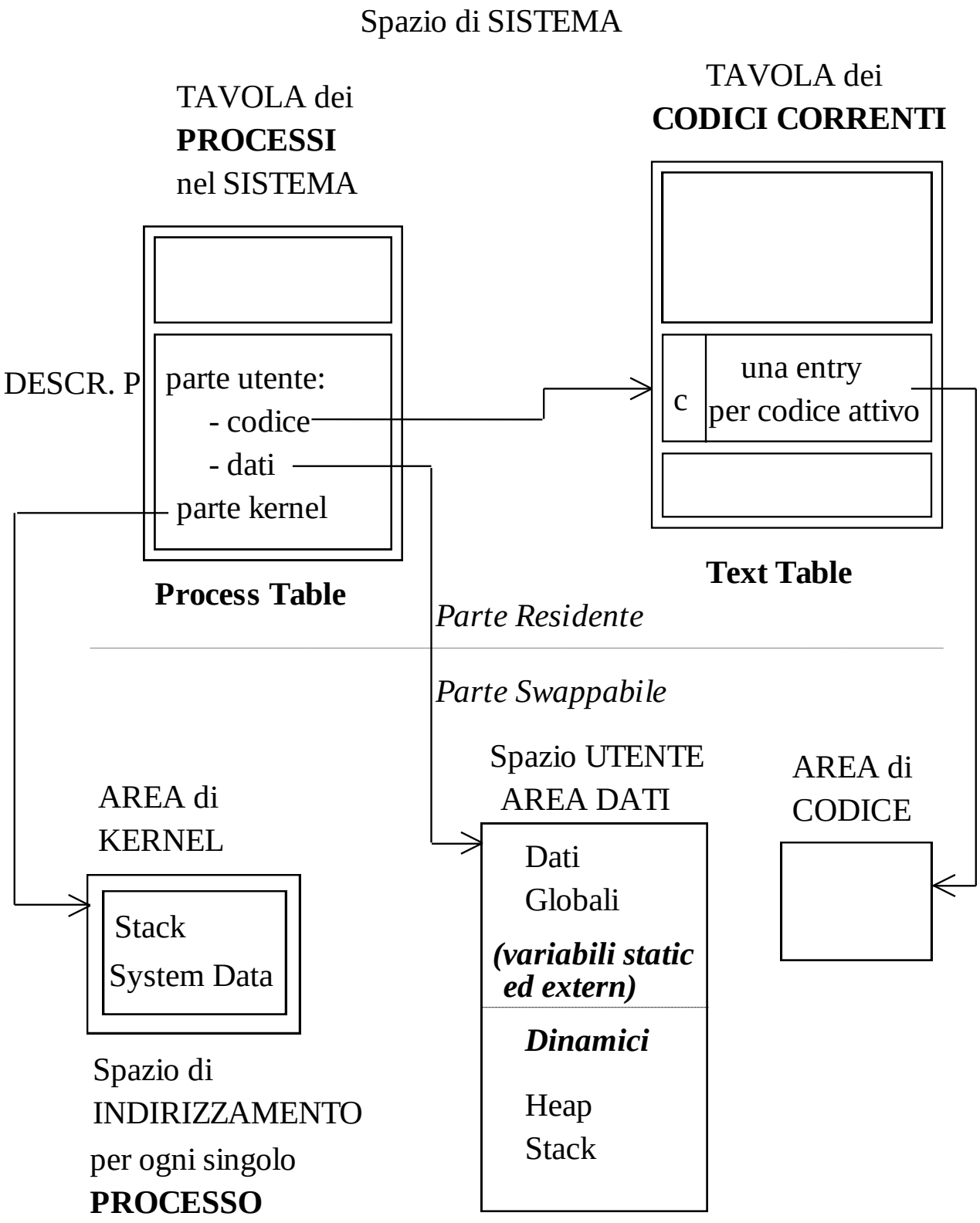


TABELLE PER I PROCESSI IN UNIX



Si noti che le tabelle di UNIX sono di dimensioni fissate: non si può eccedere la dimensione

SPAZIO DI INDIRIZZAMENTO/IMMAGINE DI UN PROCESSO

Ogni **PROCESSO** (***pesante***) possiede:

1) una **PARTE di UTENTE** composta da
- **a) Area di CODICE rientrante**
  codice eseguibile del processo utente
  ⇨ condivisibile da più processi
- **b) Area di DATI**
  dati sia globali (variabili *extern* e *static* in C) che dinamici (variabili locali allocate sullo *stack* e variabili allocate nello *heap*)
  area privata del processo
  ⇨ invisibile agli altri processi

2) una **PARTE di KERNEL ➔ KERNEL AREA**
come estensione della parte di UTENTE
In genere accessibile SOLO durante la esecuzione delle primitive del Sistema Operativo
- **a) stack del kernel**
  usato durante l'esecuzione di primitive del kernel chiamate dal processo
- **b) system data**
  - **environment**
  ⇨ area con argomenti ed environment
  - **user area**
  informazioni per l'esecuzione del processo:
  ad esempio
  TABELLA DEI FILE APERTI
  INFORMAZIONI PER I SEGNALI

Tutte le aree (di utente e di kernel) possono essere soggette a **swap** ⇨ sono necessarie solo se il processo è in esecuzione

Il Sistema Operativo UNIX ha accesso, in ogni istante, alla **kernel area** del processo correntemente in esecuzione

TABELLE DEL KERNEL

Il **Kernel** di UNIX mantiene **due Tabelle:**

* **PROCESS TABLE ⇨ *tabella dei processi*:**
  un elemento per ogni processo attivo nel sistema

* **TEXT TABLE ⇨ *tabella dei codici*:**
  un elemento per ogni codice attivo nel sistema (poichè il codice è rientrante)

⇨ **sempre residenti in memoria e globali**

1) PROCESS TABLE

ogni elemento rappresenta il **DESCRITTORE DI PROCESSO** (detto anche Process Control Block, PCB) cioè contiene tutte le informazioni importanti per il processo (sia quando è in esecuzione e sia quando è pronto o sospeso)
⇨ ***Il descrittore esiste per tutta la vita del processo***

Ad esempio:

- **pid** (**p**rocess **id**entifier) ⇨ nome del processo
- **ppid** (**p**arent **p**rocess **id**entifier) ⇨ nome del prc. padre
- **real uid** (**u**ser **id**entifier)
- **real gid** (**g**roup **id**entifier)
  ⇨ Un processo è creato da un utente (con un UID e un GID) di cui si tiene traccia nel descrittore
- **effective uid** (**u**ser **id**entifier)
- **effective gid** (**g**roup **id**entifier)
  ⇨ Un processo può eseguire un programma che ha il SUID e/o il SGID settato che appartiene ad un utente (con UID e GID) di cui si tiene traccia nel descrittore
- **locazioni delle aree utente e di kernel**
- **stato** del processo
- **priorità** del processo
- **puntatore al prossimo processo in coda**

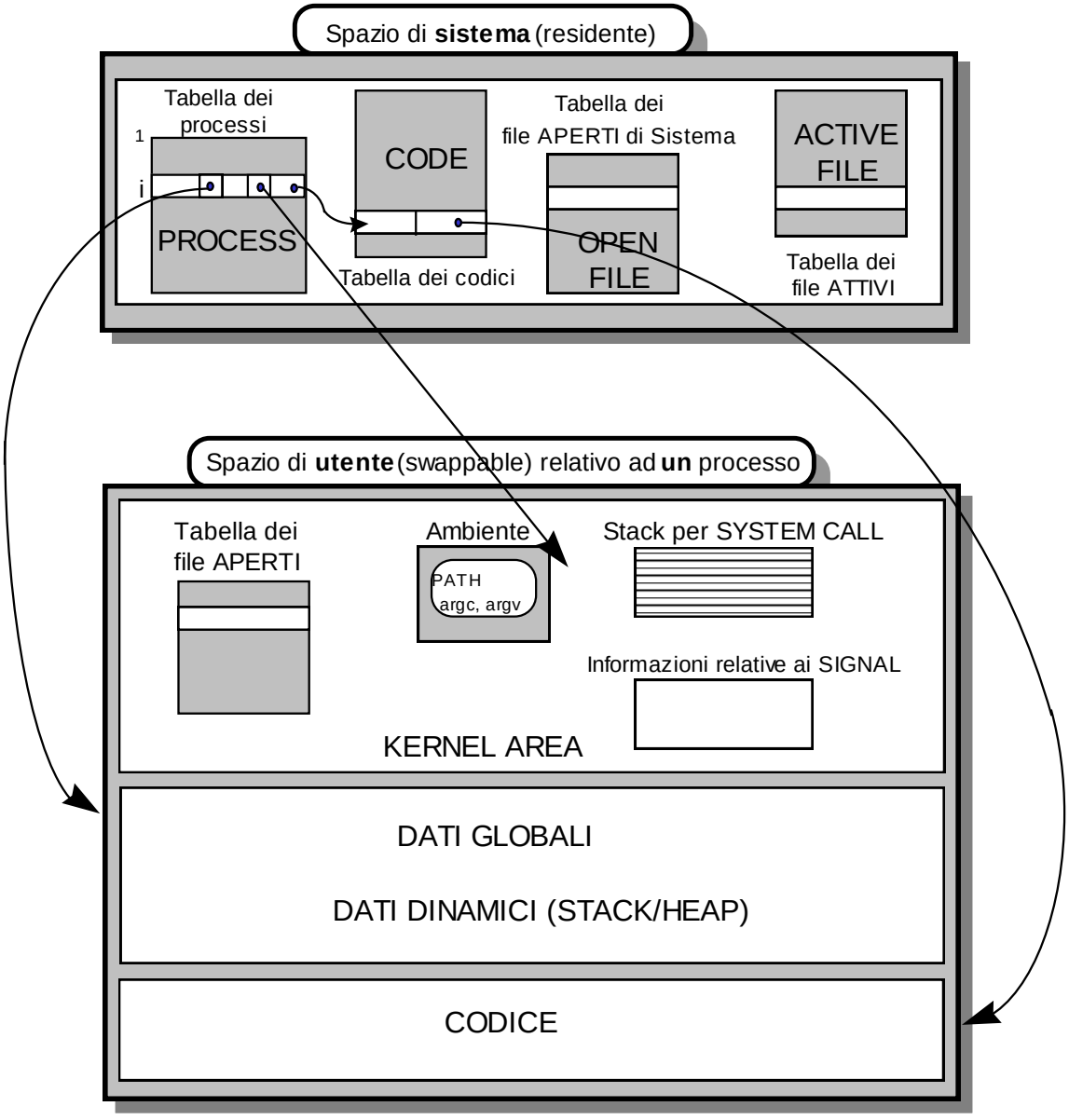
TABELLE DEL KERNEL (segue)

2) TEXT TABLE

ogni elemento rappresenta un **codice attivo** nel sistema

⇨ contiene un contatore di processi che fanno riferimento a quel codice, l'indirizzo in memoria centrale (se caricato) e l'indirizzo in memoria secondaria (in caso di swap)

OSSERVAZIONE: In caso di uso di paginazione, la text table contiene l'indirizzo della tabella delle pagine



Kernel di Linux (versione 2.0)

Dal file /usr/include/linux/sched.h

Tabella dei processi (statica): variabile globale

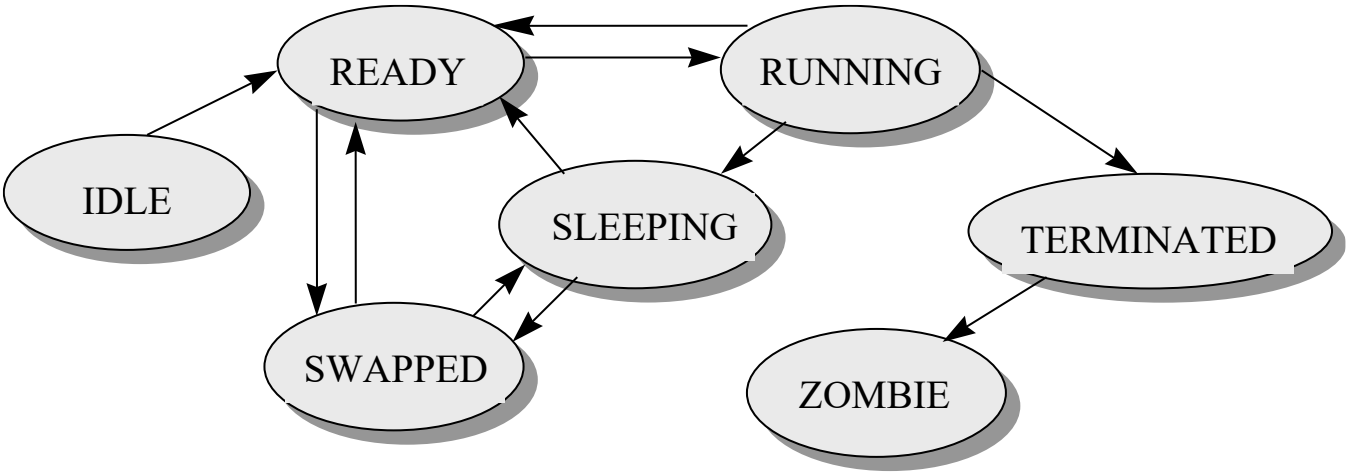
```
extern struct task_struct *task[NR_TASKS];
```

In tasks.h: `#define NR_TASKS 512`

```
struct task_struct {  /* descrittore di processo */
/* stato del processo */
    volatile long state;
/* identificatore del processo */
    pid_t pid;
/* user id, effective user id, saved user id */
    uid_t uid,euid,suid;
/* group id, effective group id, saved group id */
    gid_t gid,egid,sgid;
/* tempo di esecuzione in "thick"; priorità dinamica
    long counter;
/* priorità statica */
    long priority;
/* maschera dei segnali ricevuti non ancora serviti */
    int sigpending;
/* codice di terminazione e segnale che ha causato la terminazione */
    int exit_code, exit_signal;
/* task seguente e precedente nella lista dei task */
    struct task_struct *next_run;
    struct task_struct *prev_run;
/* padre originale */
    struct task_struct *p_opptr;
/* padre attuale */
    struct task_struct *p_pptr;
/* informazioni sui file aperti*/
    struct files_struct *files;
    ... }
```

Stati interni possibili di un processo UNIX:

IDLE	stato iniziale (caricamento in memoria del processo e inizializzazione delle strutture dati del Sistema Operativo)
READY	pronto per la esecuzione
RUNNING	in esecuzione
SLEEPING	in attesa di un evento per proseguire



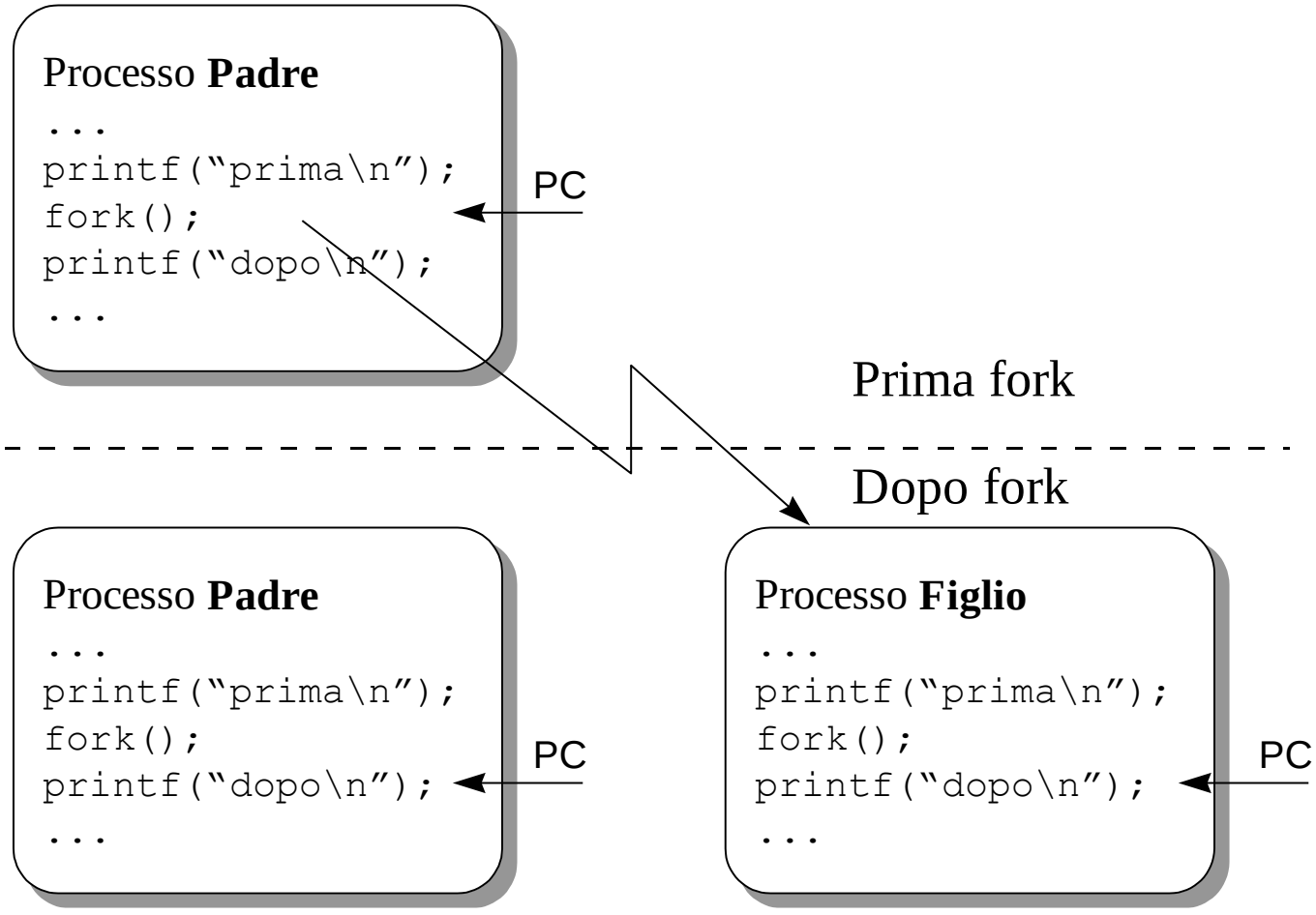
SWAPPED	immagine copiata su disco ⇨ swap out: si applica preferibilmente a processi sospesi
TERMINATED	terminato
ZOMBIE	terminato, ma presente in attesa di consegna del risultato al padre (che non ha ancora aspettato il figlio)

PRIMITIVE PER LA GESTIONE DEI PROCESSI

CREAZIONE

FORK pid = **fork** ( );
int pid;

Un processo ne genera un altro
➔ dopo la generazione si hanno 2 processi concorrenti e separati:
- il **parent** (processo padre), quello originario
- il **child** (processo figlio), quello generato.

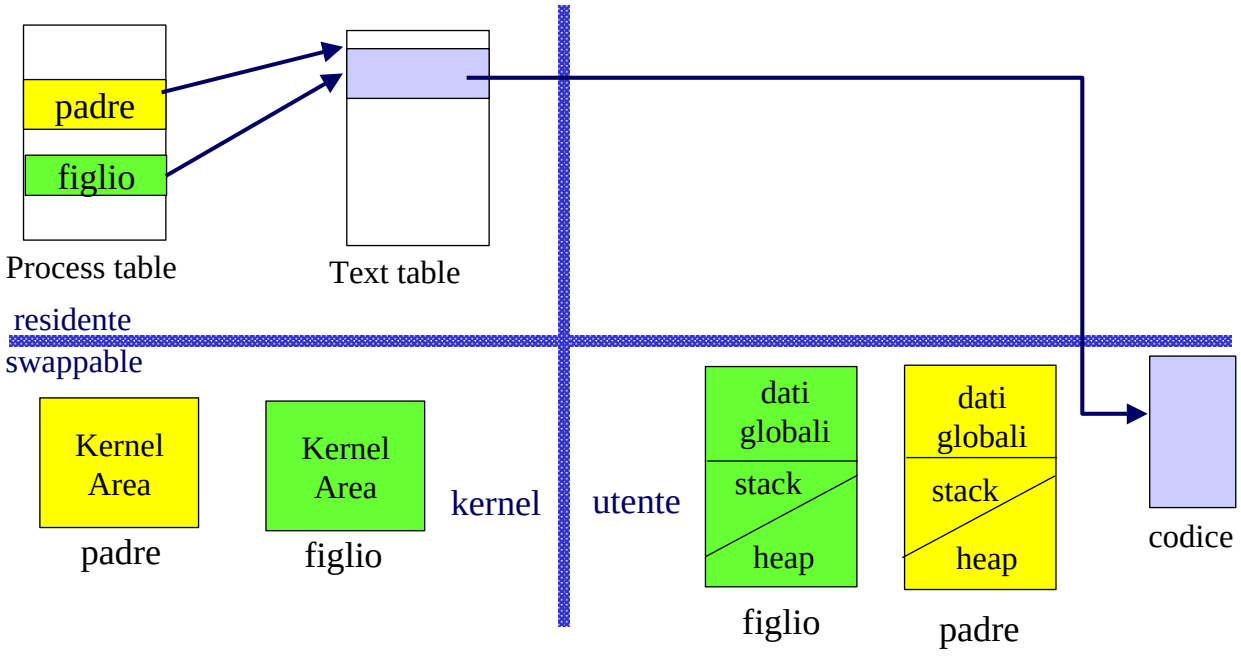


EFFETTI della FORK

Se si verifica un qualsiasi ***errore***,
la fork restituisce al processo parent il valore **-1**

Se la fork **ha avuto successo** viene creato un nuovo processo figlio:

1- si inserisce una nuova entry nella **tabella dei processi**
   ⇨ descrittore per il nuovo processo con attributi ereditati dal padre: ad esempio, stesso UID e GID (reali ed effettivi!) e stesso PC[1] ➔ processo figlio in stato IDLE
2- si crea per il nuovo processo una area **dati utente** come copia dell'area **dati utente** del processo padre
3- il nuovo processo esegue lo stesso codice del padre
   ⇨ aggiornamento contatore nell'elemento della **text table**, perchè padre e figlio *condividono* lo stesso codice
4- si crea per il nuovo processo una **kernel area** come copia della **kernel area** del processo padre
   ⇨ tabella dei file aperti e situazione nello stato dei segnali uguale a quella del processo padre
5- si inserisce il descrittore del figlio nella coda dei processi pronti ➔ processo figlio in stato READY



[1] **PC** = PROGRAM COUNTER

SCHEMA di GENERAZIONE

Se la **fork** ha avuto **successo** restituisce un valore differente ai due processi padre e figlio:

* nel figlio (*child)* tale valore è zero (**0**)
* nel padre (*parent) è* un intero diverso da zero
  ===> l'**identificatore del processo creato (PID)**

Questo consente ai due processi (PADRE e FIGLIO) di eseguire sezioni diverse dello stesso codice

PRIMA POSSIBILITÀ

```
   ...
if (fork()==0)
   { ...  /* codice eseguito dal figlio */
     ...
    }
else
    { /* codice eseguito dal padre */
       ...
     }
```

SECONDA POSSIBILITÀ ← useremo di norma questa!

```
   ...
if (fork()==0)
   { ...  /* codice eseguito dal figlio */
     ...
     exit(valore); /* il figlio termina con
uno specifico valore che verra' ritornato al
padre!
   }
/* NON SERVE PIU' l'else! */
/* codice eseguito dal padre */
       ...
```

OSSERVAZIONI:

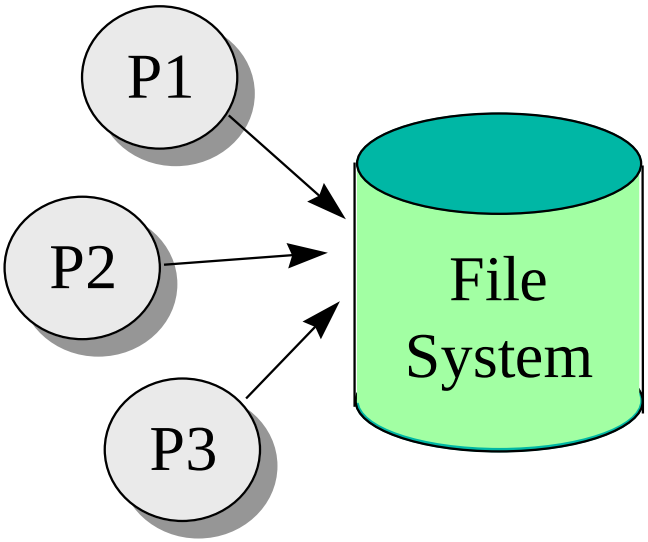
* Un processo PADRE può generare un numero qualunque di processi figli e ogni processo figlio può diventare PARENT di altri processi
  ===> generazione di una **FAMIGLIA di PROCESSI**

* Le variabili e i puntatori del padre sono *copiati* e quindi *non* vengono *condivisi* da parent e child ma *duplicati*

* Per effetto della copia della Kernel Area vengono copiate le informazioni sui segnali
  ⇨ lo stato dei segnali è *mantenuto* dal padre al figlio

* I file aperti dal processo risultano *condivisi* perchè sempre per effetto della copia della Kernel Area viene copiata, in particolare, la tabella dei file aperti dal processo padre al processo figlio e quindi viene condiviso sia l'elemento della Tabella dei file aperti di sistema (*condivisione* dei file pointer dei file usati) sia l'elemento della Tabella dei File Attivi
  ⇨ l'I/O pointer *si sposta* per tutti in seguito a letture o scritture eseguite da una famiglia di processi

CONDIVISIONE FILE SYSTEM



Vediamo le implicazioni a livello di programmazione

CONDIVISIONE di FILE: caso padre e figlio

Processo P1 (padre)

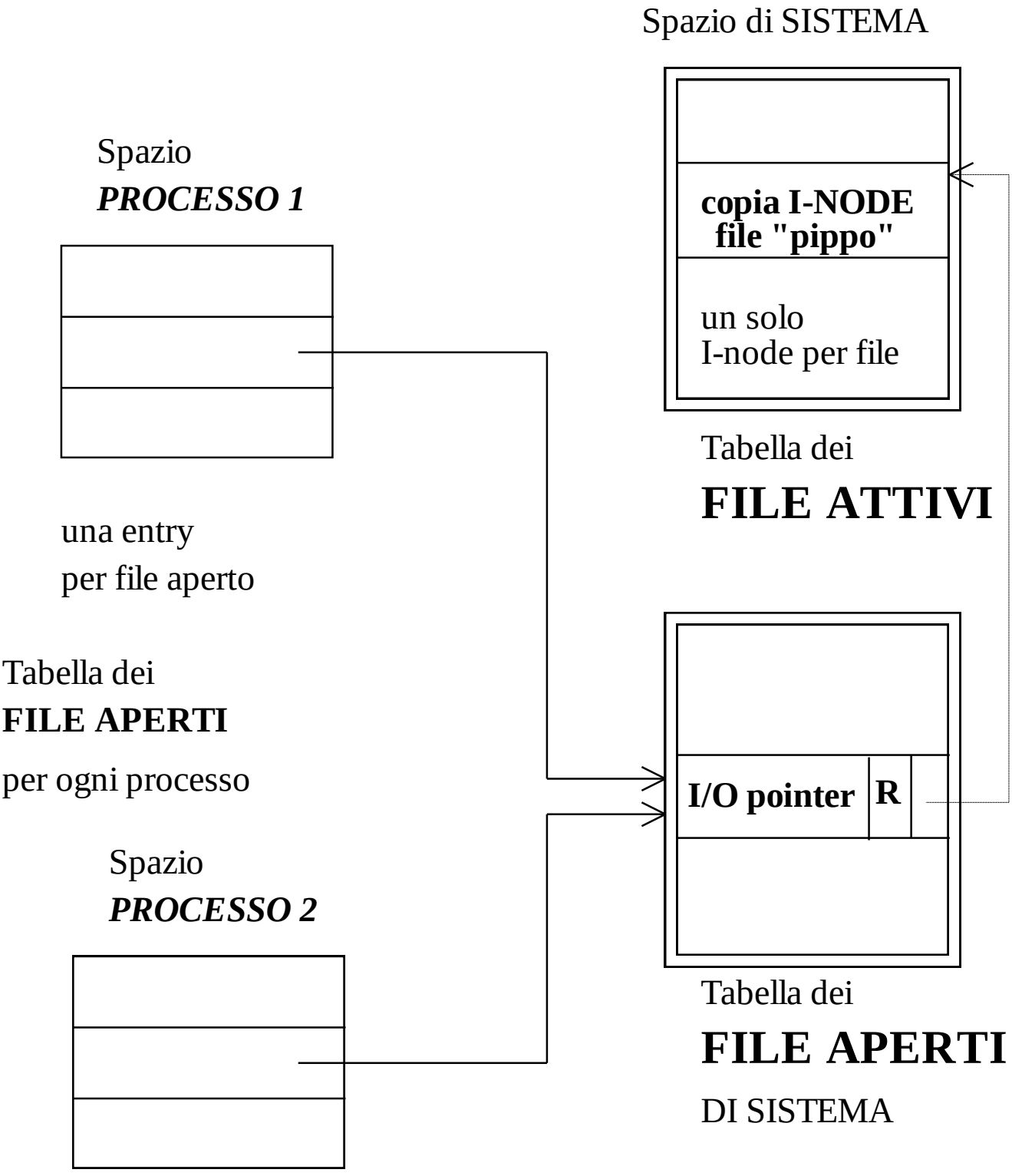
```
main()
{
int fd;

fd = open("pippo", O_RDONLY);

if    (fork()==0)
    { ...
    /* codice eseguito dal figlio */
    ...
     }
else
     { /* codice eseguito dal padre */
          ...
      }
}
```

Processo P2 (figlio)

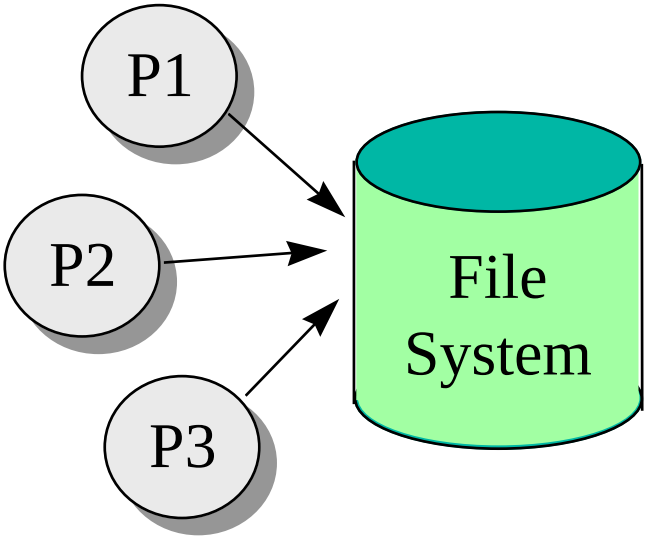
Il processo P1 (padre) apre il file"pippo": il processo P2 (figlio) può accedere al file e condivide il file pointer

CONDIVISIONE di FILE: caso padre e figlio (segue)

Sottoprocessi dello stesso processo condividono la entry nella tabella dei file aperti (e quindi lo ***stesso I/O pointer***) e anche la corrispondente entry nella tabella degli i-node attivi



CONDIVISIONE FILE SYSTEM



Vediamo le implicazioni a livello di programmazione

CONDIVISIONE di FILE: caso più generale

Processo P1

```
main()
{
int fd;

fd = open("pippo",O_RDONLY);

}
```

Processo P2

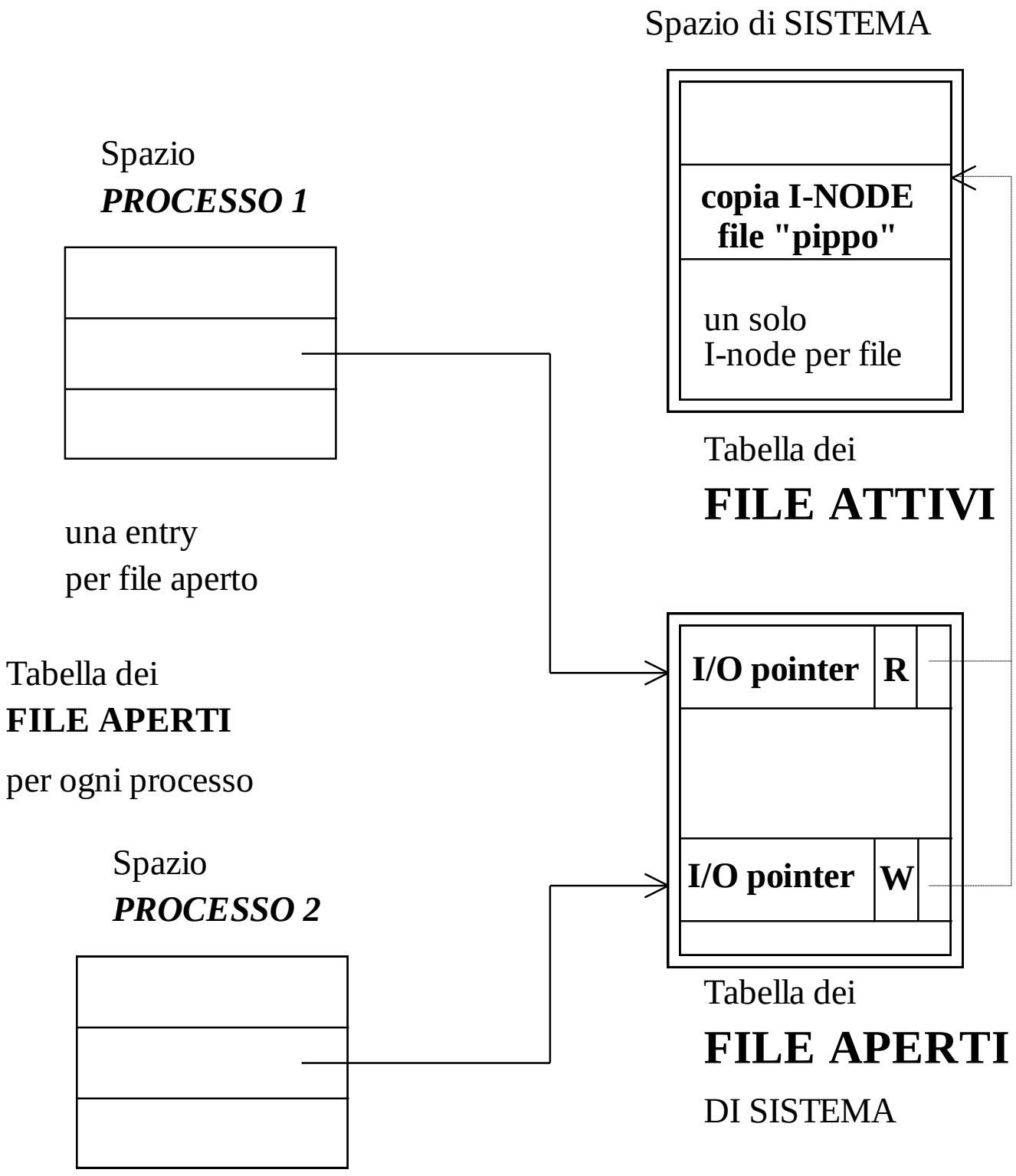
```
main()
{
int fd;

fd=open("pippo",O_WRONLY);

}
```

I processi P1 e P2 aprono lo stesso file"pippo" (supponendo che abbiano lo stesso direttorio corrente e che abbiano i diritti giusti)

NOTA BENE:
P1 e P2 potrebbero anche essere in relazione di parentela

CONDIVISIONE di FILE: caso più generale (segue)

Aperture separate di uno stesso file portano condividere solo la stessa entry della tabella degli i-node attivi, ma si avranno distinte entry nella tabella dei file aperti di sistema



Kernel di Linux (versione 2.0)

System call `fork()`:

- crea un nuovo descrittore di processo (`task_struct`)
- cerca un elemento libero nella tabella dei processi
- copia le informazioni del processo padre nel nuovo processo
- imposta correttamente i puntatori del processo padre e del processo figlio

E le pagine di dati?

⇒ **copy-on-write**:
area dati duplicata solo se un processo la modifica

- area dati marcata *write-protected*
- una scrittura genera un *page-fault*
- il kernel fa una copia dell'area dati e dà al processo scrittore il permesso di modificarla

⇨ il tutto in modo trasparente al processo

PROCESS IDENTIFIER

Fra gli attributi di un processo, uno dei più importanti è il process identifier
⇨ identificatore UNICO per il processo
⇨ il PID che viene ritornato al processo padre dopo una FORK

Ogni processo può conoscere il proprio PID mediante la primitiva GETPID

GETPID
```
pid = getpid ();
int pid;
```

Ogni processo può, inoltre, conoscere il PID del PADRE mediante la primitiva GETPPID

GETPPID
```
ppid = getppid ();
int ppid;
```

USER IDENTIFIER E GROUP IDENTIFIER

Altri attributi importanti di un processo sono l'**UID** (User IDentifier) e il **GID** (Group IDentifier) sia reali che effettivi
⇨ identificatori di utente e di gruppo

Ogni processo può conoscere i propri UID e GID reali mediante le primitive GETUID e GETGID

GETUID
```
uid = getuid ();
int uid;
```

GETGID
```
gid = getgid ();
int gid;
```

Le versioni **GETEUID** e **GETEGID** restituiscono rispettivamente UID e GID effettivi

SOSPENSIONE

Dopo la generazione del child il parent può decidere
se operare **contemporaneamente** ad esso
oppure
se **attendere** la sua terminazione usando la primitiva

WAIT pid = **wait(&status)**;
int status;
int pid;

Normalmente, nel valore di ritorno **status** (supposto a 16 bit) si ha:
* nel *byte alto*, il valore restituito da un figlio con la **exit**
* nel *byte basso*, zero

Invece, in caso di terminazione di un figlio in seguito alla ricezione di un segnale, nel valore di ritorno **status** si ha:
* nel *byte alto*, zero
* nel *byte basso*, il numero del segnale che ha provocato la terminazione del figlio

La primitiva wait **ritorna -1** in caso il processo invocante non abbia figli da attendere, altrimenti il PID del figlio terminato

EFFETTO della primitiva WAIT:

La **wait** sospende un processo padre in attesa della terminazione di uno dei processi figli

Quindi la wait è:
* *sospensiva* se ci sono processi figli attivi (non terminati)
* *non sospensiva* se tutti i processi figli sono terminati

SCHEMA di sincronizzazione tra padre e figlio

1 caso)

```
int pid, status, pidfiglio;
if ((pid = fork()) == 0)
   { ...          /* codice eseguito dal child */
      ...
   }
else
   {           /* codice eseguito dal parent */
      ...
      pidfiglio = wait(&status);
      ...
   }
```

2 caso) nel caso non interessi il valore restituito con la exit

```
if ((pid = fork()) == 0)
      {           /* codice eseguito dal child */
         ...
   }
else
   {           /* codice eseguito dal parent */
      ...
      pidfiglio = wait((int *) 0);
      ...
   }
```

*Si può anche ignorare il valore di ritorno (PID) della **wait***

In caso di più figli, se si vuole attendere un figlio con un certo pid: **while ((rid = wait (&status) != pid);**

Normalmente vengono definite le seguenti **macro** (in sys/wait.h):

- **WIFEXITED(status)** e **WEXITSTATUS(status)**
- **WIFSIGNALED(status)** e **WTERMSIG(status)**

Un processo può terminare:

modo *involontario*
modo *volontario*

INVOLONTARIO
a seguito di:
- **azioni non consentite** (come riferimenti a indirizzi scorretti o tentativi di eseguire codici di operazioni non definite)
- **segnali generati dall'utente** da tastiera e ricevuti dal processo
- **segnali spediti da un altro processo** tramite la system call kill

VOLONTARIO
- invocazione della **exit**
o
- alla conclusione del programma main

TERMINAZIONE

EXIT

```
void exit (status);
int status;
```

La exit **chiude tutti i file aperti**, per il processo che termina

Il valore del parametro **status** viene passato al **processo padre**, quando questo invoca la primitiva wait

Per convenzione in UNIX:
* il valore zero rappresenta una terminazione NORMALE
* un valore diverso da zero rappresenta un problema

ESEMPIO di uso di WAIT ed EXIT

```
main ()
{ int pid, status, exit_s;

if ((pid = fork()) < 0)
   { /* fork fallita */
      printf("Errore in fork\n"); exit(1);
   }

if (pid == 0)
   { /* figlio */
      printf("Esecuzione del figlio\n");
      sleep(4);
      exit(5);
   }

/* padre */
if (wait (&status) < 0)
   { printf("Errore in wait\n");
      exit(1);
   }
if ((status & 0xFF) != 0)
      printf("Errore in status\n");
else
   { exit_s = status >> 8;
/* selezione degli 8 bit piu' significativi */
      exit_s &= 0xFF;
      printf("Per il figlio %d lo stato di EXIT e %d\n",
            pid, exit_s);
   }
exit(0);
}
```

Esercizio:
Scrivere una versione che fa uso delle macro opportune

Altro esempio di uso della WAIT e della EXIT: CONDIVISIONE FILE E I/O POINTER

```
/*  Il  figlio  scrive  su  un  file;  il  padre  torna
all'inizio e legge */
#include <stdio.h>
#include <fcntl.h>
#include <sys/wait.h>

int procfile (char *f1);
{int nread, nwrite, atteso, status, fileh, pid;
 char st1[80], st2 [80];

if ((fileh = creat (f1, 0777)) < 0 ) return 1;
close(fileh);
fileh = open (f1, O_RDWR);
/* apertura in lettura/scrittura */

if ((pid = fork()) < 0)   {close (fileh); return 2; }
else
  if (pid == 0)    /* FIGLIO */
{  scanf  ("%s", st1);
   nwrite = write (fileh, st1, strlen(st1)+1);
      exit (nwrite); }
else /* PADRE */
{  atteso = wait (&status);  /* attesa del figlio */
   lseek (fileh, 0L, 0);
   nwrite = WEXITSTATUS(status);
   nread = read (fileh, st2, nwrite);
   printf("Il figlio ha avuto la stringa %s\n", st2);
   close (fileh);
   return (0); }
   }

main  (int argc, char **argv)
{ int integi;
   integi = procfile (argv[1]);
   exit (integi); }
```

OSSERVAZIONE:

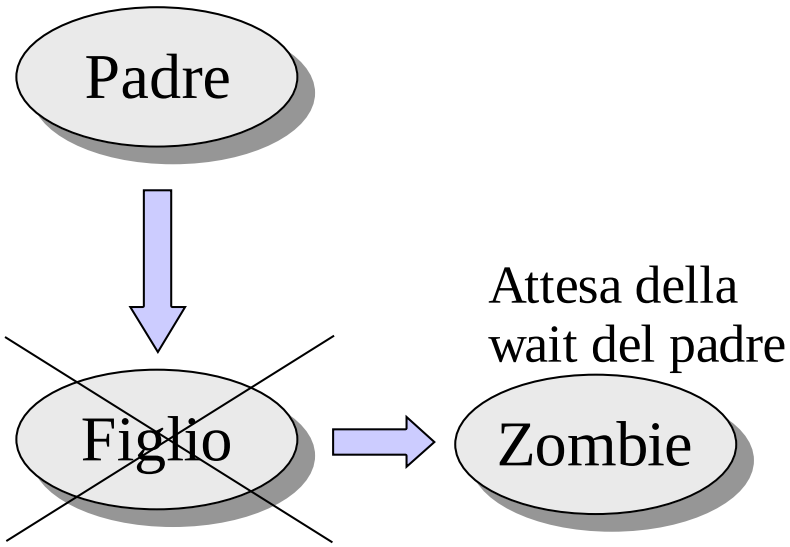
Usando exit e wait possono presentarsi due possibilità:

a) **un processo figlio può terminare mentre il padre non ha (ancora) eseguito la primitiva wait**

===> il processo figlio è messo in una sorta di LIMBO e passa nello stato **ZOMBIE**

Quando il processo padre eseguirà la **wait**, il processo figlio ***zombie*** verrà rimosso dal sistema

Terminazione del Figlio: Processo Zombie

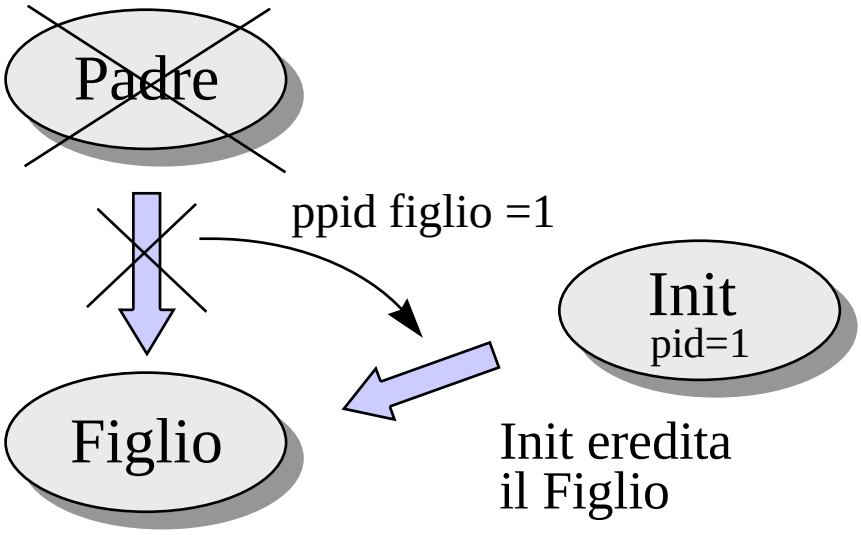


segue OSSERVAZIONE

b) **un processo padre può terminare prima dei suoi figli**

===> processi figli (inclusi quelli zombie) che rimangono senza padre vengono adottati dal **processo INIT**

Terminazione del Padre



NOTA BENE: Quando termina un processo adottato da INIT, il processo INIT rileva automaticamente il suo stato di terminazione e quindi i processi adottati da INIT non diventano mai ZOMBIE

ESECUZIONE DI UN PROGRAMMA (famiglia di primitive EXEC)

UNIX consente di cambiare il programma che un processo sta eseguendo usando una delle primitive della famiglia exec

OSSERVAZIONI:

- **NON** si prevede di tornare al **programma chiamante**

- La primitiva exec **non produce un nuovo processo**, ma solo il cambiamento dell'area di UTENTE del processo interessato, sia come codice e sia come dati

- NON si ha un ritorno al codice che ha invocato la EXEC
  ⇨ corrisponde ad un GOTO
  si torna all'istruzione successiva **SOLO** in caso di **ERRORE**

Alcune delle **primitive** della famiglia EXEC sono:

execv (pathname, argv);
execl (pathname, arg0, argv1, ..., argvn, (char *) 0);
execvp (name, argv);
execlp (name, arg0, argv1, ..., argvn, (char *) 0);

char *pathname, *name, *argv[], *arg0, *arg1, ... *argn;

Il nuovo programma deve essere un file eseguibile

NOTA:
Per convenzione, il primo parametro (argv[0] o argv0) deve essere sempre presente ed essere il nome del programma da eseguire

ULTERIORI EFFETTI della EXEC

- i descrittori dei file aperti rimangono tali dopo la exec

  NON è necessario riaprire i file:
  i FILE DESCRIPTOR sono *ereditati*

- i segnali sono *alterati*
  se essi venivano ignorati rimangono tali
  ma, se erano collegati a funzioni vengono riportati allo stato di default
  ⇨ **PERCHÉ???**

- l'identificatore *effettivo* del processo viene cambiato in quello del proprietario del file corrispondente al programma eseguito nel caso di esecuzione di un programma con **suid** settato, mentre l'identificatore *reale* rimane inalterato (analogo discorso per il sgid)

Si ereditano:
direttorio corrente, file aperti, maschera dei segnali, terminale di controllo ed altre caratteristiche

ESEMPI di uso delle PRIMITIVE EXEC:

a) EXECV (Argomenti in vettore)

execv(pathname, argv);
char *pathname, *argv[];

```
main ()
{    char *av[3];
     av[0]="ls";
     av[1]="-l";
     av[2]= (char *)0;

     printf("Esecuzione di ls\n");
     execv("/bin/ls", av);

     printf("Errore in execv\n");
     exit(1);
}
```

2) EXECL (Argomenti in lista)

execl(pathname, argv0, ... argvn, (char *)0);
char *pathname, *argv0, ... *argvn;

```
main ()
{
     printf("Esecuzione di ls\n");
     execl("/bin/ls", "ls", "-l", (char *)0);

     printf("Errore in execl\n");
     exit(1);
}
```

3) EXECVP (Argomenti in vettore e path)

execvp (name, argv);
char *name, *argv[];

/* file callecho.c */

```
main ()
{    char *argin[4];

     argin[0]="myecho";
     argin[1]="hello";
     argin[2]="world!";
     argin[3]=(char *)0;

     printf("Esecuzione di myecho\n");
     execvp(argin[0], argin);

     printf("Errore in execvp\n");
     exit(1);
}
```

/* file myecho.c */

```
main (int argc, char **argv)
{    int i;
     printf("Sono myecho\n");
     for (i=0; i < argc; i++)
          printf("Argomento argv[%d]=%s\n",i,argv[i]);
}
```

ESERCIZIO:
Verificare cosa succede se viene eseguito un programma con SUID (o SGID) settato

OSSERVAZIONI:

* La differenze fra **execl** ed **execlp**
  e fra **execv** ed **execvp**

  è che nelle prime il nome del programma da eseguire deve essere determinato (pathname), mentre nelle seconde è un nome relativo semplice e la ricerca del file eseguibile avviene secondo il valore della variabile di ambiente PATH

* Un file comandi per un interprete shell può essere eseguito solo usando le primitive **execlp** e **execvp**

ALTRE PRIMITIVE della FAMIGLIA EXEC:

execve (pathname, argv, envp);
execle (pathname, arg0, argv1, ..., argvn, (char *) 0, envp);

char *pathname, *argv[], *arg0, *arg1, ... *argn, *envp[];

Queste primitive sono simili alle execv ed execl, ma in più possono specificare anche un **ambiente DIVERSO**

RIASSUMENDO:
exec**l**, exec**le**, exec**lp**, exec**v**, exec**ve**, exec**vp**

l ➔ la funzione riceve una lista di argomenti (terminata da NULL);
v ➔ la funzione riceve un vettore di argomenti argv[];
p ➔ la funzione prende un nome di file come argomento e lo cerca nei direttori specificati in PATH;
e ➔ la funzione riceve anche un vettore envp[] che rimpiazza l'environment corrente

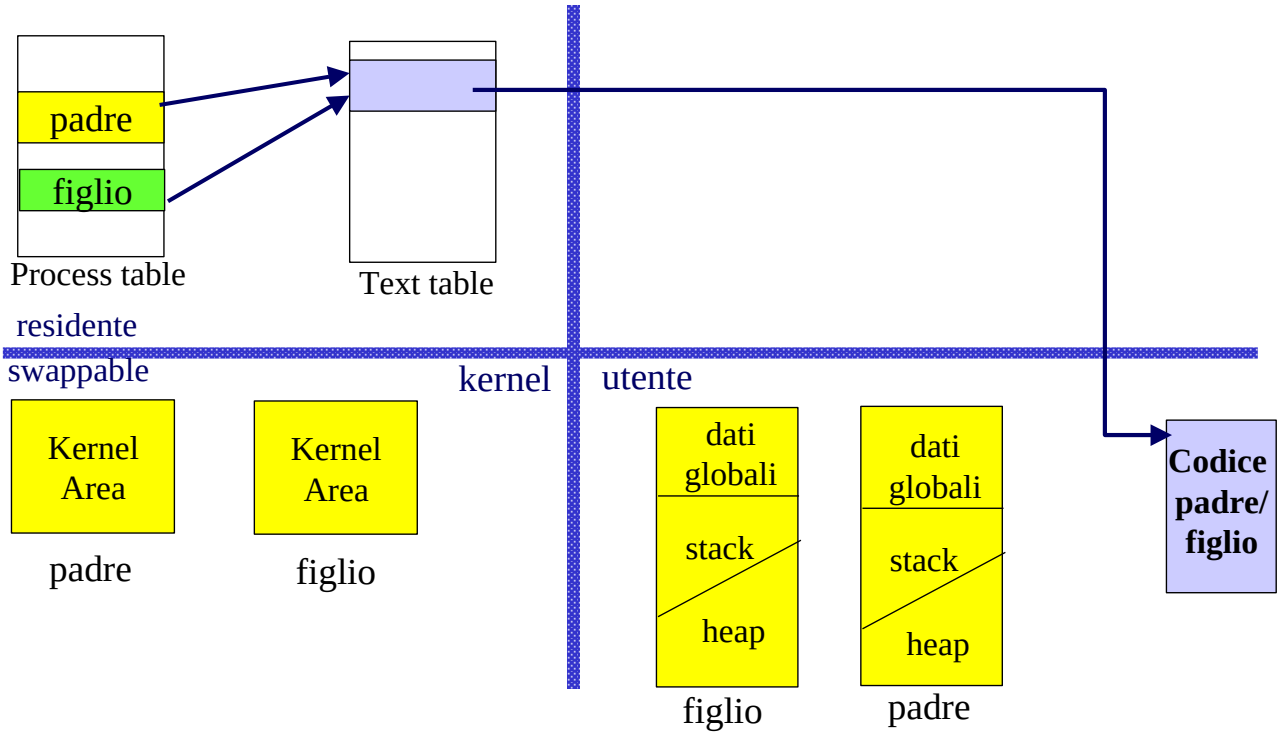
USO DI FORK ed EXEC

Esecuzioni differenziate per padre e figlio

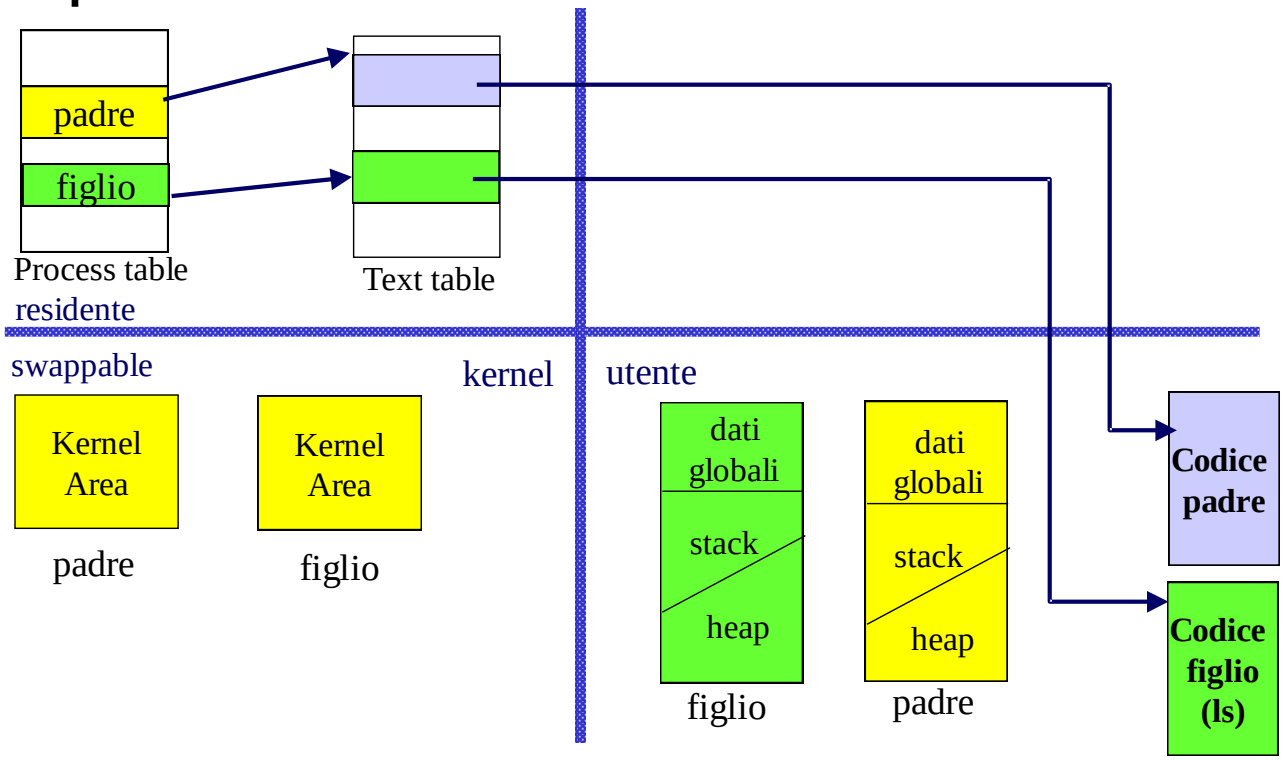
```
main ()
{   int pid;

    pid = fork();

if (pid == 0){ /* figlio */
               printf("Esecuzione di ls\n");
               execl("/bin/ls", "ls", "-l", (char *)0);
               printf("Errore in execl\n");
               exit(1);
             }

if (pid > 0)  { /* padre */
               wait ((int *)0);
               printf("Terminato ls\n");
               exit (0);
             }

if (pid < 0)  { /* fork fallita */
               printf("Errore in fork\n");
               exit(1);
             }

}
```

Il figlio esegue una **exec** per **eseguire** un **altro programma**

Prima della exec



Dopo la exec



INIZIALIZZAZIONE

In UNIX alla accensione, il processo iniziale (il processo **init**) esegue una procedura di inizializzazione:

* crea tutti i processi di sistema (ad esempio quello che gestisce la memoria virtuale)

* per la gestione dei terminali (e quindi per gli utenti interattivi), crea un processo per ognuno, che esegue il comando **getty** (*exec getty*)

PROCESSO GETTY:
Questo processo rimane in attesa di un collegamento presentando all'utente la richiesta di login

Quando un utente si collega viene eseguito il comando **login** (*exec login*) che controlla lo username e la password

Se tutto è giusto, questo processo esegue il comando specificato nel file /etc/passwd, ad esempio, **/bin/sh** (*exec shell)*

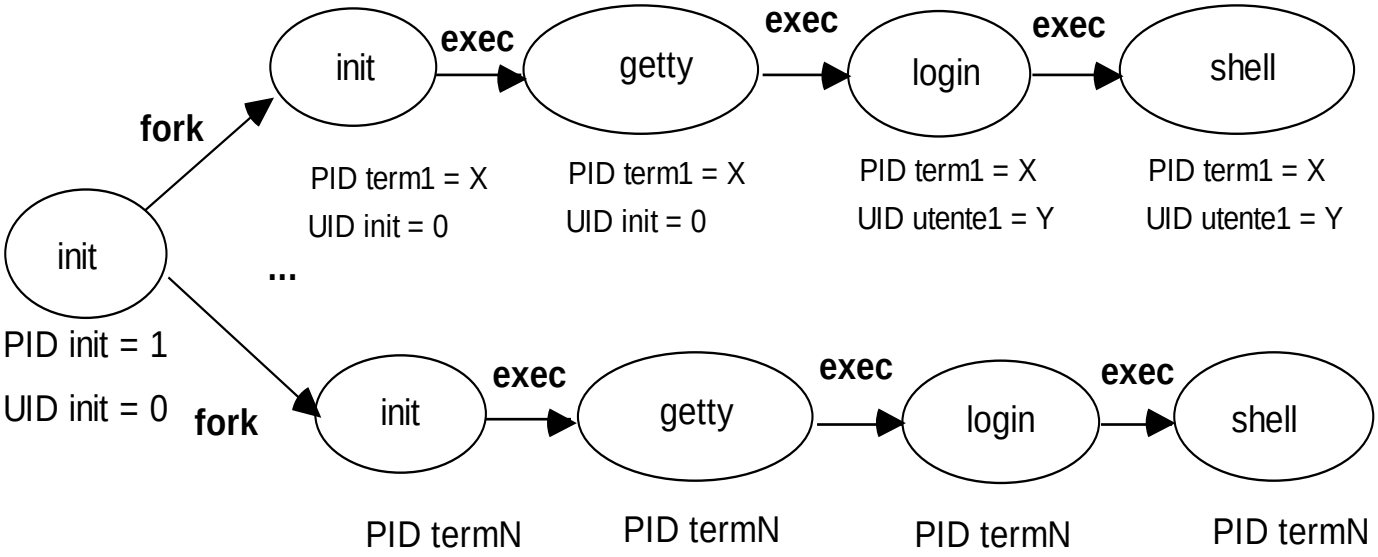
Questo processo, a questo punto, diventa il PADRE di tutta la famiglia di processi generati nella sessione di lavoro
⇨ ogni comando (non di ambiente) genera (*fork*) un nuovo processo shell figlio

SHELL PADRE:
se in FOREGROUND ➔ lo shell PADRE si pone in attesa del suo completamento usando la primitiva WAIT
se in BACKGROUND ➔ il PADRE NON FA LA WAIT

SHELL FIGLIO:
esegue le eventuali espansioni e ridirezioni e poi passa ad eseguire il comando richiesto (*exec comando*)

Evoluzione dei processi in UNIX

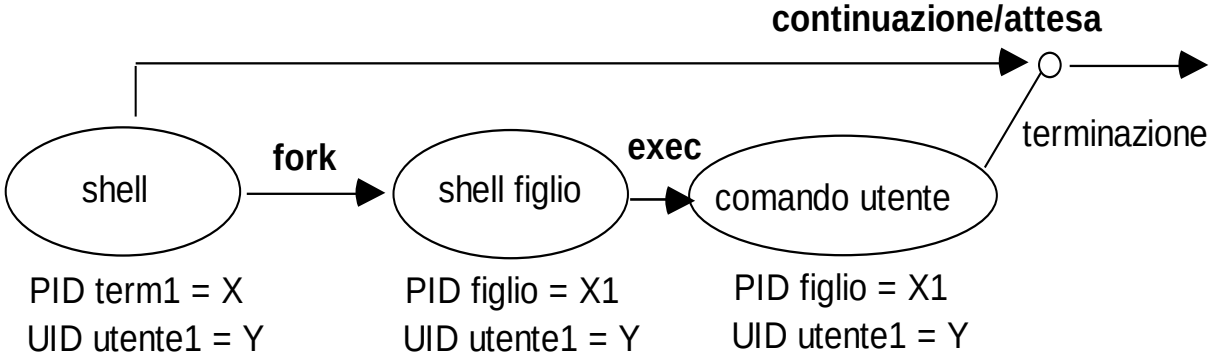
Inizializzazione:
il processo iniziale cura la fase iniziale e
crea un processo shell per ogni terminale



Schema di funzionamendo del programma di **login**:

- chdir sulla home directory (6° campo linea di /etc/passwd)
- cambio UID e GID del processo (3° e 4° campi linea di /etc/passwd)
- exec del comando (ultimo campo linea di /etc/passwd)

Esecuzione in shell:
il processo di utente associato al terminale attende una linea di comando e crea un processo per ogni comando



ESERCIZIO (esecuzione di comandi)

```
#include <stdio.h>

main  (int argc, char **argv)
{ int stato, atteso, pid;
  char st[80];

for (;;)   /* ciclo infinito */
{
  if ((pid = fork()) < 0)
              { perror("fork"); exit(1);}

  if (pid == 0)
    {  /*  FIGLIO: esegue i comandi */
    printf("inserire il comando da eseguire:\n");
    scanf("%s", st);
    execlp(st, st,(char *)0);
    perror("errore");
    exit (0);
    }
  else {      /* PADRE */
      atteso=wait (&stato);
/*attesa figlio: sincronizzazione */
      printf("eseguire altro comando? (si/no) \n");
      scanf ("%s", st);
      if (strcmp(st, "si")) exit(0);
    }
  }
}
```

Gestione degli errori

In caso di fallimento, le System Call ritornano **-1**

In più, UNIX assegna alla variabile globale **`errno`** il codice di errore occorso alla system call

Vedere
`errno.h`
per le corrispondenze codici di errori e loro descrizione (dove è definita `extern int errno`)

perror()

funzione utilizzata nella gestione degli errori, stampa (su **standard error**) una stringa definita dall'utente, seguita dalla descrizione dell'`errno` avvenuto

Esempio:

```
perror("stringa descrittiva");
```

può stampare

```
stringa descrittiva: No such file or directory
```

SISTEMA OPERATIVO UNIX

Ambienti

processore comandi (**shell**)
linguaggio di sistema (**C**)

UNIX livelli

utente	programmatore
processore comandi (shell)	linguaggio di sistema (C)
nucleo (primitive di sistema)	

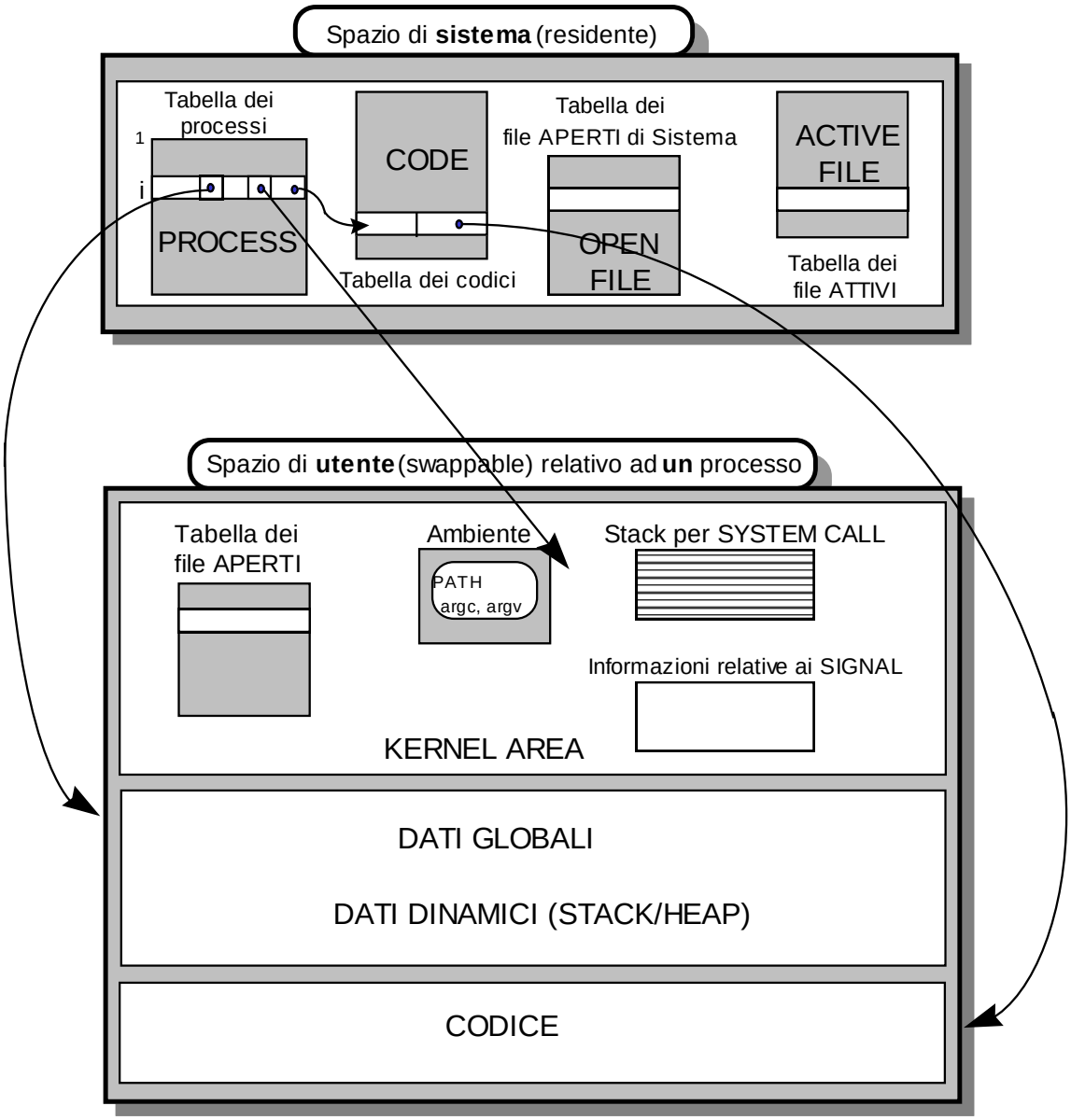
Passaggio da un ambiente all'altro

Uso di shell (**fork**)
che crea un altro processo che fa una **exec** ed esegue in C

fork *exec*
wait *exit*

*passaggio argomenti e ambiente in **avanti***
*solo intero di **ritorno***

Processi in UNIX



Le tabelle sono vettori di dimensioni fissate

Un processo può trovarsi in **due modi** (stati) di esecuzione: **user** o **kernel**

La esecuzione delle primitive avviene in stato kernel