Processi in Unix

Capitolo 8 -- Stevens

Controllo dei processi

- Creazione di nuovi processi
- Esecuzione di programmi
- Terminazione del Processo
- Altro…

Identificatori di processi

- Ogni processo ha un identificatore unico non negativo
- Process ID = 1 → il cui file di programma è contenuto in **/sbin/init**
  - invocato dal kernel alla fine del boot, legge il file di configurazione **/etc/inittab** dove ci sono elencati i file di inizializzazione del sistema (rc files) e dopo legge questi rc file portando il sistema in uno stato predefinito (multi user)
  - non muore mai.
  - è un processo utente (cioé non fa parte del kernel) di proprietà di root e ha quindi i privilegi del superuser

Identificatori di processi

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

pid_t getpid (void);        process ID del processo chiamante
pid_t getppid (void);    process ID del padre del processo chiamante

uid_t getuid (void);     real user ID del processo chiamante
uid_t geteuid (void);       effective user ID del processo chiamante
gid_t getgid (void);     real group ID del processo chiamante
gid_t getegid (void);       effective group ID del processo chiamante
```

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>



int main(void)
{
   printf("pid del processo = %d\n", getpid() );
   return (0);
}
```

Funzione fork

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

pid_t fork(void);
```

Restituisce: 0 nel figlio,

pid del figlio nel padre

-1 in caso di errore

Creazione di nuovi processi

- L'unico modo per creare nuovi processi è attraverso la chiamata della funzione fork da parte di un processo già esistente
- quando viene chiamata, la fork genera un nuovo processo detto figlio
- Il valore di ritorno della fork **non è UNIVOCO**
  - il valore restituito al processo figlio è 0
  - il valore restituito al padre è il **pid** del figlio
    - un processo può avere più figli e non c'è nessuna funzione che può dare al padre il pid dei suoi figli
- figlio e padre continuano ad eseguire le istruzioni che seguono la chiamata di fork

Creazione di nuovi processi

- il figlio è una copia del padre
- condividono *dati, stack* e *heap*
  - il kernel li protegge settando i permessi *read-only*
- solo se uno dei processi tenta di modificare una di queste regioni, allora essa viene copiata (*Copy On Write)*
- in generale non si sa se il figlio è eseguito prima del padre, questo dipende dall'algoritmo di scheduling usato dal kernel

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

int glob=10;        /* dati inizializzati globalmente */

int main(void)
{
    int var=100;   /* vbl locale */
    pid_t pippo;
     printf(“prima della fork\n”);
     pippo=fork();
      if( (pippo == 0) {glob++; var++;}
        else sleep(2);
    printf(“pid=%d, glob=%d, var=%d\n”,getpid(),glob,var);
   exit(0);
}
```

```
$ a.out
prima della fork
pid=227, glob=11, var=101
pid=226, glob=10, var=100
```

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

int glob=10;        /* dati inizializzati globalmente */

int main(void)
{
    int var=100;   /* vbl locale */
    pid_t pippo;
     printf(“prima della fork”);
     pippo=fork();
      if( (pippo == 0) {glob++; var++;}
        else sleep(2);
    printf(“pid=%d, glob=%d, var=%d\n”,getpid(),glob,var);
   exit(0);
}
```

```
$ a.out
prima della forkpid=227, glob=11, var=101
prima della forkpid=226, glob=10, var=100
```

Condivisione file

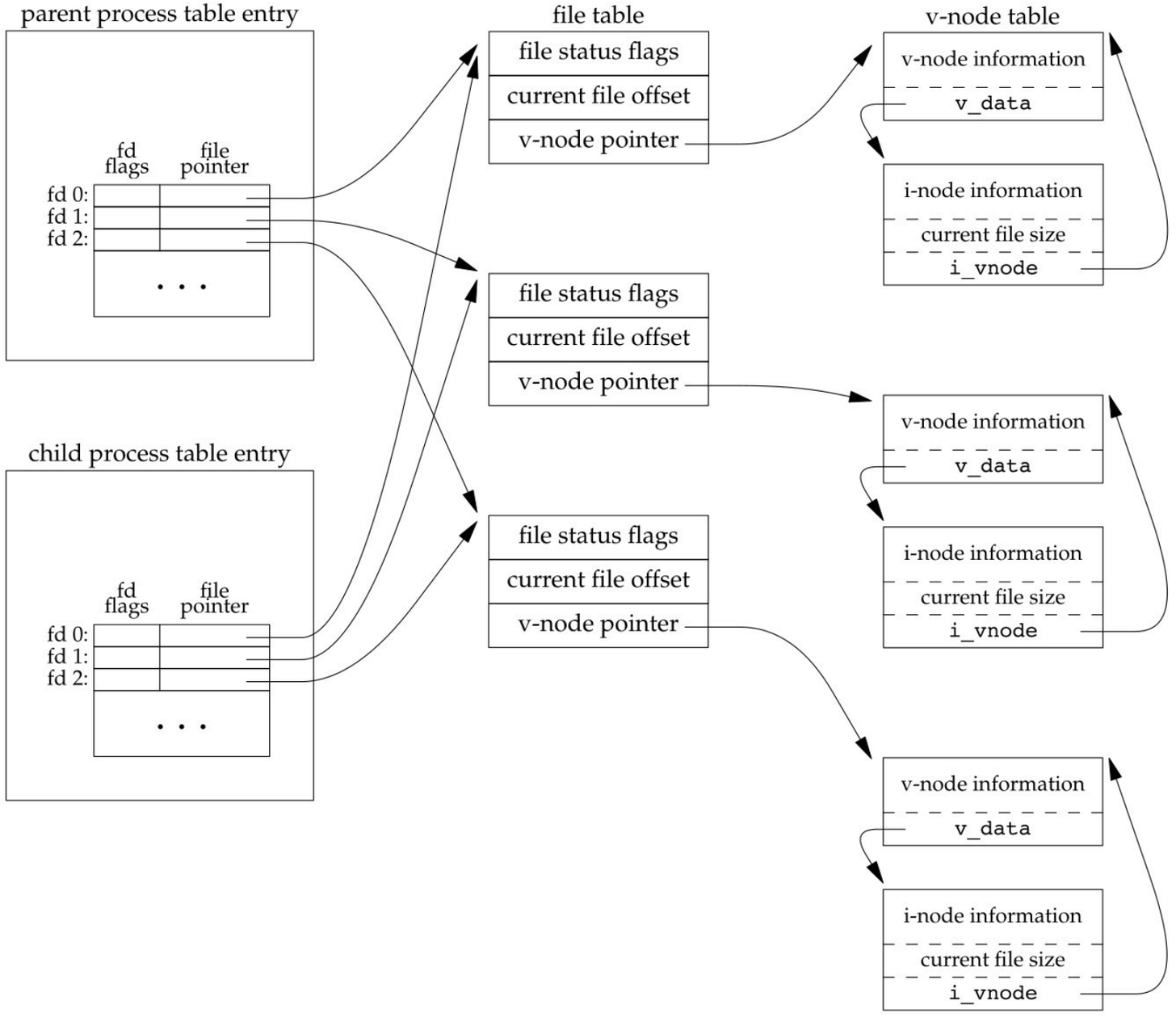

Figure 8.2 Sharing of open files between parent and child after `fork`

Condivisione file

- nel programma precedente lo standard output del figlio è condiviso con il padre
- **tutti** i file aperti del padre condivisi dai figli
- problema della sincronizzazione:
  - chi scrive per prima? oppure è intermixed
  - nel programma abbiamo messo *sleep(2)*
    - ma non siamo sicuri che sia sufficiente...ci ritorniamo

Problema della sincronizzazione

- il padre aspetta che il figlio termini
  - La posizione nei file condivisi viene eventualmente aggiornata dal figlio ed il padre si adegua
- o viceversa
- tutti e due chiudono i file che non gli servono, così non si danno fastidio
- ci sono altre proprietà ereditate da un figlio:
  - uid, gid, euid, egid
  - suid, sgid
  - cwd
  - file mode creation mask
  - ...

Ci ritorneremo

Esercizi

a) Si supponga di mandare in esecuzione il seguente programma:

```
int main(void)
{
pid_t pid1, pid2;
pid1 = fork();
pid2 = fork();
exit(0);
}
```

Dire quanti processi vengono generati. Giustificare la risposta.

b) Si supponga di mandare in esecuzione il seguente programma:

```
int main(void)
{
pid_t pid1, pid2;
pid1 = fork();
if (pid1>0)
    pid2 = fork();
exit(0);
}
```

Dire quanti processi vengono generati. Giustificare la risposta

conclusioni: quando si usa fork?

un processo attende richieste (p.e. da parte di client) allora si duplica

- il figlio tratta (handle) la richiesta
- il padre si mette in attesa di nuove richieste

un processo vuole eseguire un nuovo programma (p.e. la shell si comporta così) allora si duplica e il figlio lo esegue

Ambiente di un processo

Capitolo 7 -- Stevens

Avvio di un processo

Quando parte un processo:

- si esegue prima una routine di start-up speciale che prende
  - valori passati dal kernel dalla linea di comando
    - in *argv*[ ] se il processo si riferisce ad un programma C
  - variabili d'**ambiente**

- successivamente viene chiamata la funzione principale da eseguire
  - int main(int *argc*, char **argv* [ ]);

Tipica occupazione di memoria

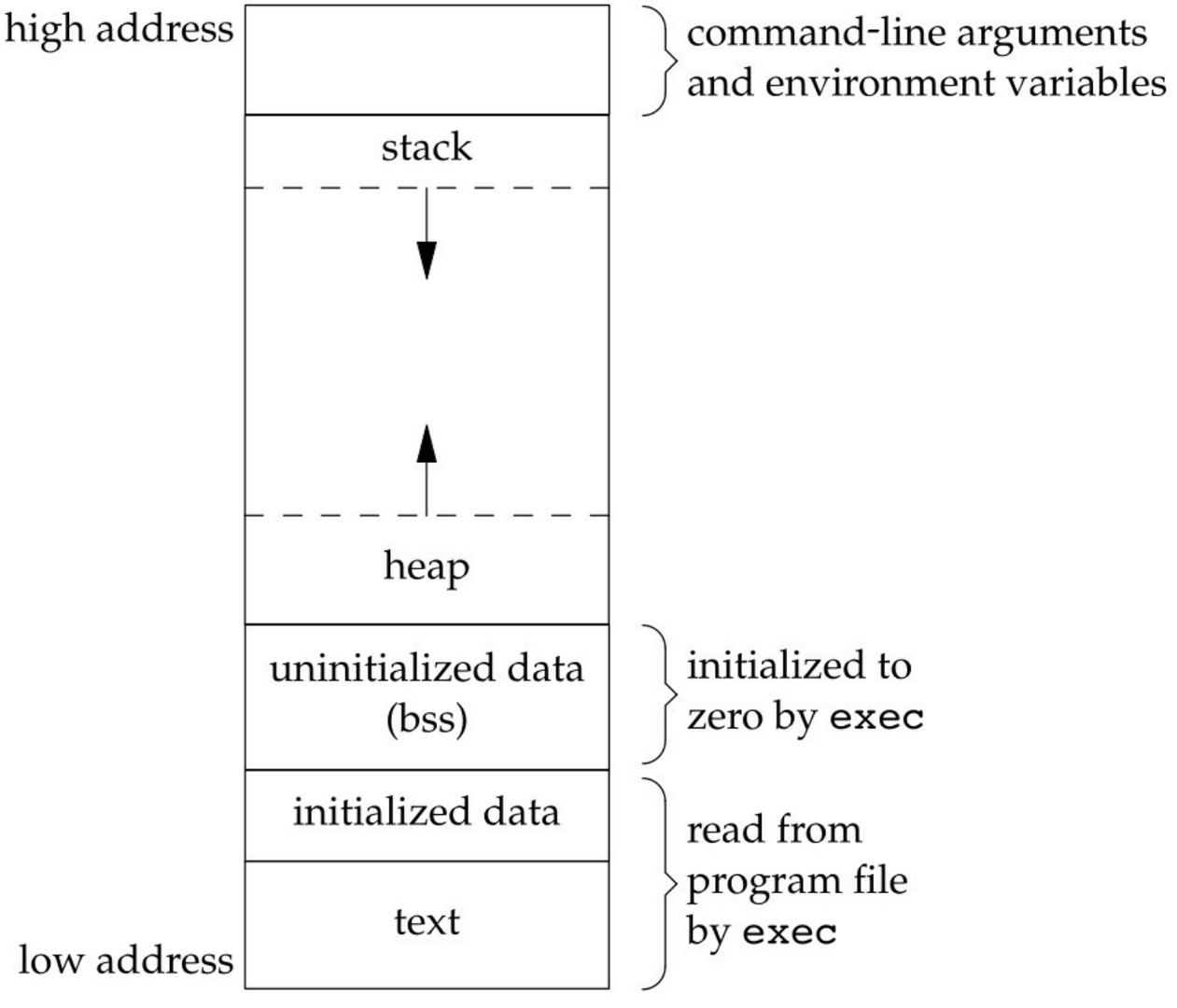

Figure 7.6 Typical memory arrangement

Environment List

ad ogni processo è passato anche una lista di variabili di ambiente individuata dalla variabile

extern char **environ

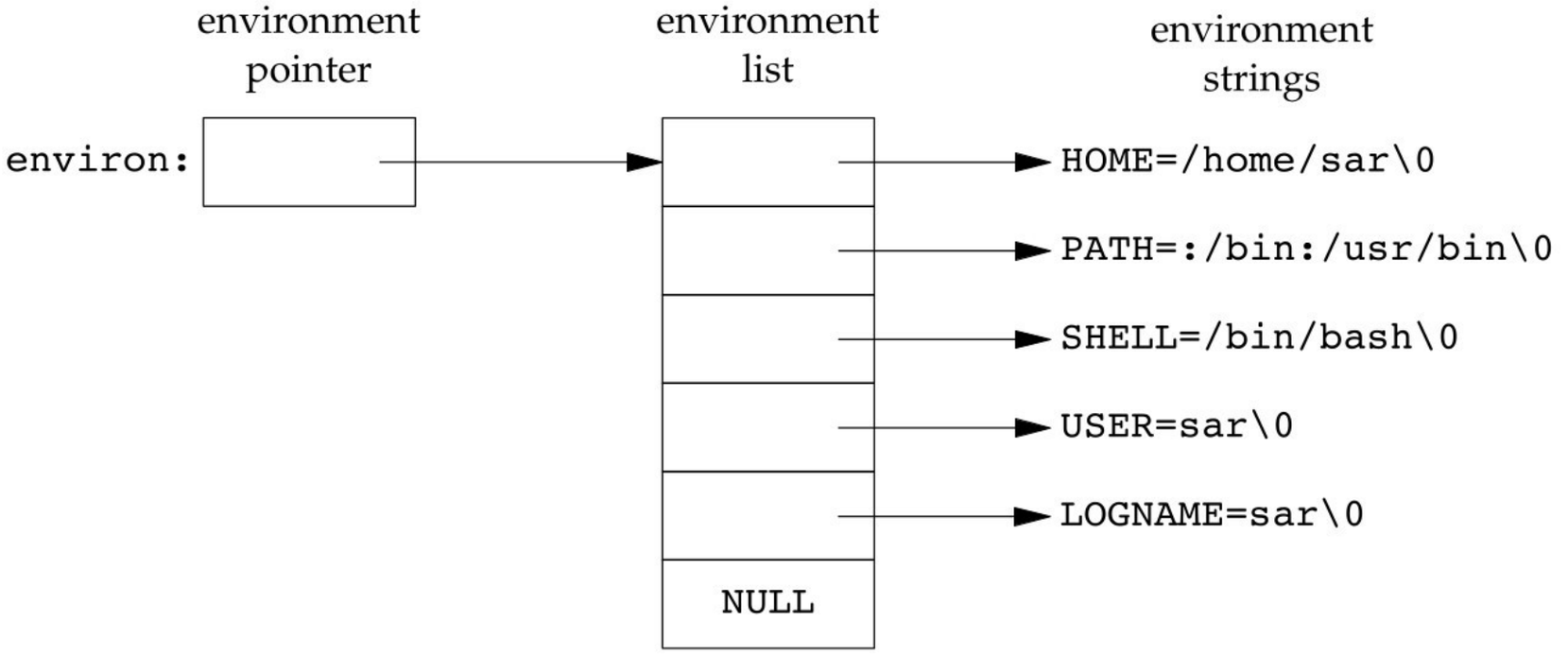

Figure 7.5 Environment consisting of five C character strings

Terminazione di un processo

- Terminazione normale
  - ritorno dal main
  - chiamata a exit
  - chiamata a _exit
- Terminazione anormale
  - chiamata abort
  - arrivo di un segnale

Funzioni exit

#include <stdlib.h>

void exit (int *status*);

Descrizione: restituisce *status* al processo che chiama il programma includente exit ; effettua prima una *pulizia* e poi ritorna al kernel

- effettua lo shoutdown delle funzioni di libreria standard di I/O (fclose di tutti gli stream lasciati aperti) => tutto l'output è flushed

#include <unistd.h>

void _exit (int *status*);

Descrizione: ritorna immediatamente al kernel

```
#include <stdio.h>


int main(void)

{

    printf(“Ciao a tutti”);


    _exit(0);
     exit(0);

}
```

Exit handler

```
#include <stdlib.h>

int atexit (void (*funzione) (void));
```

Restituisce: 0 se O.K.

diverso da 0 su errore

funzione = punta ad una funzione che è chiamata per effettuare operazioni di cleanup per il processo alla sua normale terminazione.

Il numero di exit handlers che possono essere specificate con atexit è limitato dalla quantità di memoria virtuale.

Vengono inserite in uno **stack**!

```
int main(void)
{
   atexit(my_exit2);      atexit(my_exit1);
   printf("ho finito il main\n");
   return(0);
}

static void my_exit1(void)
{
   printf("sono il primo handler\n");
}

static void my_exit2(void)
{
   printf("sono il secondo handler\n");
}
```

Sostituiamo return(0) con _exit(0).

Che cosa succede mandando in esecuzione

```
$ a.out
ho finito il main
sono il primo handler
sono il secondo handler
```

Avvio e terminazione di un processo

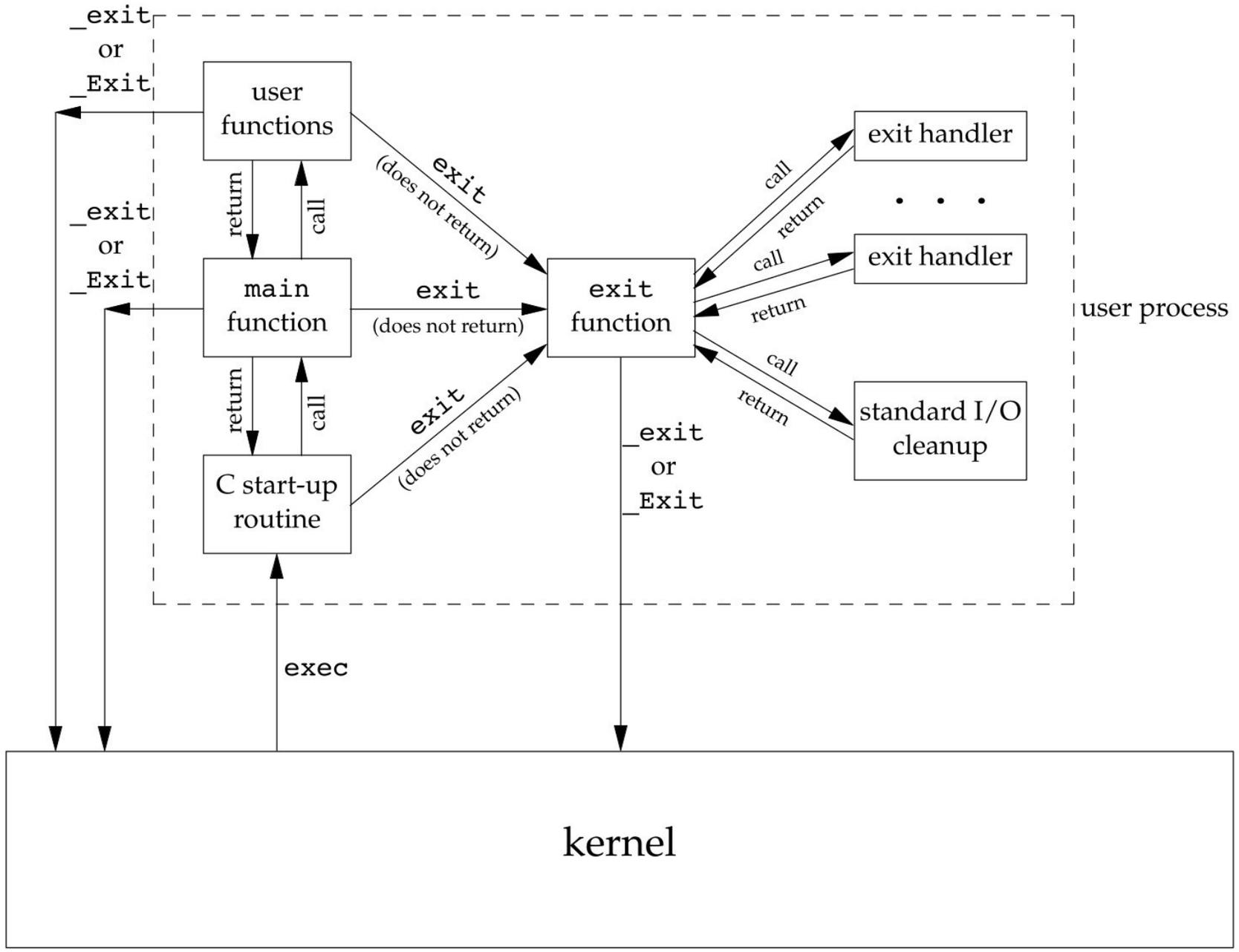

Figure 7.2 How a C program is started and how it terminates

Esercizio 6_1

Scrivere un programma che effettui la copia di un file utilizzando 2 figli: uno specializzato nella copia delle vocali ed uno nella copia delle consonanti.

Per la sincronizzazione tra i processi figli utilizzare un semaforo implementato tramite file.