Lezione 7: Processi(Ambiente di un processo)

7.1 Processo
7.2 Avvio di un processo
7.3 Terminazione di un processo
7.4 Funzione di uscita
7.5 Funzione atexit
    7.5.1 Esempio
    7.5.2 Inizio e fine di un programma C
7.6 Argomenti dalla linea di comando
7.7 Enviroment list
7.8 Struttura della memoria per un programma C
7.9 Allocazione della memoria
7.10 Unix e i processi
    7.10.1 Caratteristiche del processo unix
    7.10.2 Stati di un processo unix
    7.10.3 Processi swapped
    7.10.4 Rappresentazione dei processi
    7.10.5 Immagine di un processo
    7.10.6 Immagine di un processo unix
7.11 Processi
7.12 Chiamata di sistema Fork()
7.13 Creazione dei processi
7.14 Creazione di un processo figlio: `fork()`
7.15 Effetti della fork()
7.16 Ottenere il PID: getpid() e getppid()
    7.16.1 Esempio: padre e figlio eseguono x = 1 dopo il ritorno dalla fork
    7.16.2 Esempio: creazione di una catena di n processi
7.17 Identificativo di processo
    7.17.1 Esempio
    7.17.2 Esempio: myfork.c
7.18 Ulteriori informazioni sulla fork()
7.19 La chiamata di sistema vfork()

7.19.1 Esempiio vfork()
7.19.2 Ulteriori informazioni sulla vfork()

7.1 Processo

Un programma é costituito da istruzioni e dati ed è memorizzato in un file.

Un processo é un programma in esecuzione.

7.2 Avvio di un processo

Chiamato da una shell o da un altro programma in esecuzione, quando si esegue un programma si esegue prima una routine di start-up speciale, specificata come indirizzo di partenza del programma eseguibile, che prende valori passati dal kernel in argv[] dalla linea di comando

Successivamente è chiamata la funzione main, un programma C inizia l'esecuzione con una funzione chiamata main, il cui prototipo è :

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

argc é il numero di argomenti, argv è un array di puntatore agli argomenti

7.3 Terminazione di un processo

Esistono otto modi per terminare un processo:

Terminazione normale

- Ritorno dal main
- Chiamata di exit
- Chiamata di _exit
- Ritorno dell'ultimo thread dalla sua routine di avvio
- Chiamata di pthread_exit dall'ultimo thread

Terminazione anomala

- Chiamata di abort
- Ricezione di un segnale
- Risposta dell'ultimo thread ad una richiesta di cancellazione

La routine di avvio fa in modo che quando la funzione main ritorna venga chiamata exit.

7.4 Funzione di uscita

Sono tre le funzioni che terminano un programma normalmente...

- _exit (chiamata di sistema) ed _Exit (Libreria standard) che ritornano al kernel immediatamente.
- exit(libreria standard) che prima esegue una procedura di "Pulizia" e poi ritorna al kernel.

```
#include <stdlib.h>
void exit (int status)
void _Exit(int status)
```

```
#include <unistd.h>
void _exit(int status)
```

La funzione exit esegue sempre una terminazione pulita della libreria I/O

Tutti gli stream aperti sono chiusi con fclose.

Tutte e tre le funzioni exit riceevono un argomento intero.

Le shell dei sistemi unix forniscono un modo per esaminare lo stato di uscita di un processo.

Lo stato di uscita é indefinito se, le funzioni di uscita sonòchiamate senza alcun codice di uscita, main fa un return senza valore di ritorno, il main non é dichiarato per restituire un intero. Se main è dichiarato per restituire un intero e si ha un ritorno implicito, allora lo stato di uscita è 0.

Esempio

```
#include <stdio.h>
main (){
    printf(“Hello, World\n”);
}
```

Compilando ed eseguendo il programma osserviamo un codice di uscita casuale. Compilando lo stesso programma su sistemi differenti otteniamo codici di uscita differenti a seconda del contenuto dello stack e dei registri al momento in cui la funzione main restituisce il controllo.

Restituire un valore intero dalla funzione main equivale a chiamare exit con lo stesso valore

Richiamare return(0) é equivalente a richiamare exit(0)

7.5 Funzione atexit

```
#include<stdlib.h>
int atexit (void(*func)(void));
//Resistuisce 0 se OK, <>0 in caso di errore
```

Si passa l'indirizzo di una funzione come argomento, la funzione non riceve alcun argomentoe non restituisce nulla. Quando si invoca la exit questa chiama le funzioni nell'ordine inverso rispetto a quello di registrazione.

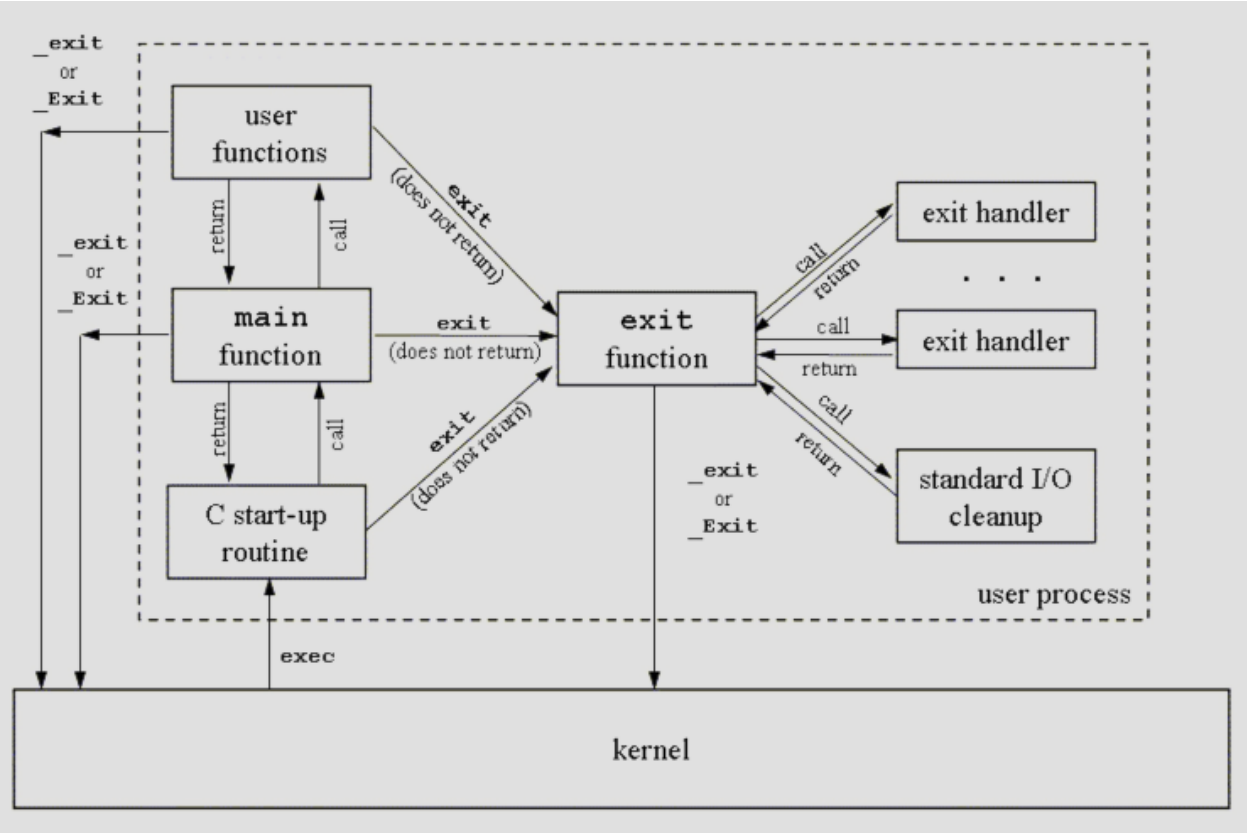
exit prima chiude gli exit handler e poi chiude tutti gli stream aperti.

7.5.1 Esempio

```
#include "apue.h"
static void my_exit1(void);
static void my_exit2(void);

int main (void){
    if (atexit(my_exit2)!=0)
        err_sys("Non posso registrare my_exit2");
    if (atexit(my_exit1)!=0)
        err_sys("Non posso registrare my_exit1");
    if (atexit(my_exit1)!=0)
        err_sys("Non posso registrare my_exit1");
    print("Main ha completato\n");
    return(0);
}

static void my_exit1(void){
    printf("Primo exit handler"\n);
}

static void my_exit(void)2{
```

```
    printf(“Secondo exit handler”\n);
}
```

7.5.2 Inizio e fine di un programma C



7.6 Argomenti dalla linea di comando

Quando é eseguio un programma il processo che esegue `exec` può passare argomenti da linea di comando al nuovo programma.

```
#include “apue.h”
int main (int argc, char *argv[]){
```

```
    int i;
    for (i=0; i<argc; i++)
        printf("argv[%d]: %s\n",i,argv[i]);
    exit(0);
}
```

7.7 Enviroment list

ad ogni programma è passata una lista dell'ambiente.

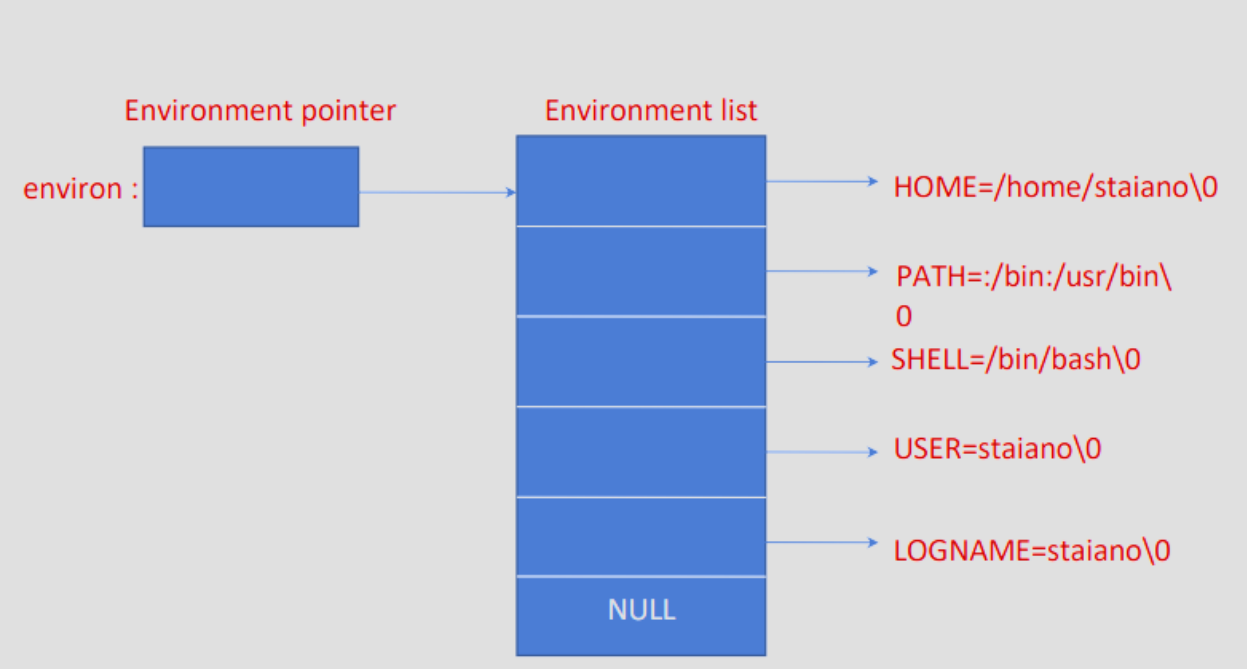
array di puntatori a stringhe, ogni puntatore contiene l'indirizzo di una stringa C terminata con null. L'indirizzo dell'array di puntatori è contenuto nella variabile globale environ:

`axtern char **environ`

Per convenzione l'ambiente consiste delle stringhe:

`nome = valore (ad esempio, HOME = /home/senneca\n)`

Ambiente di 5 stringhe di caratteri in C



Storicamente, molti sistemi unix forniscono un terzo argomento alla funzione main, cioé l'indirizzo della lista dell'ambiente

```
int main(int argc, char *argv[], char *envp[]);
```

- ISO C specifica che la funzione main sia scritta con due argomenti
- POSIX.1 specifica che si debba usare environ anziché il terzo argomento, poiché il terzo argomento non comporta alcun vantaggio rispetto alla variabile globale environ

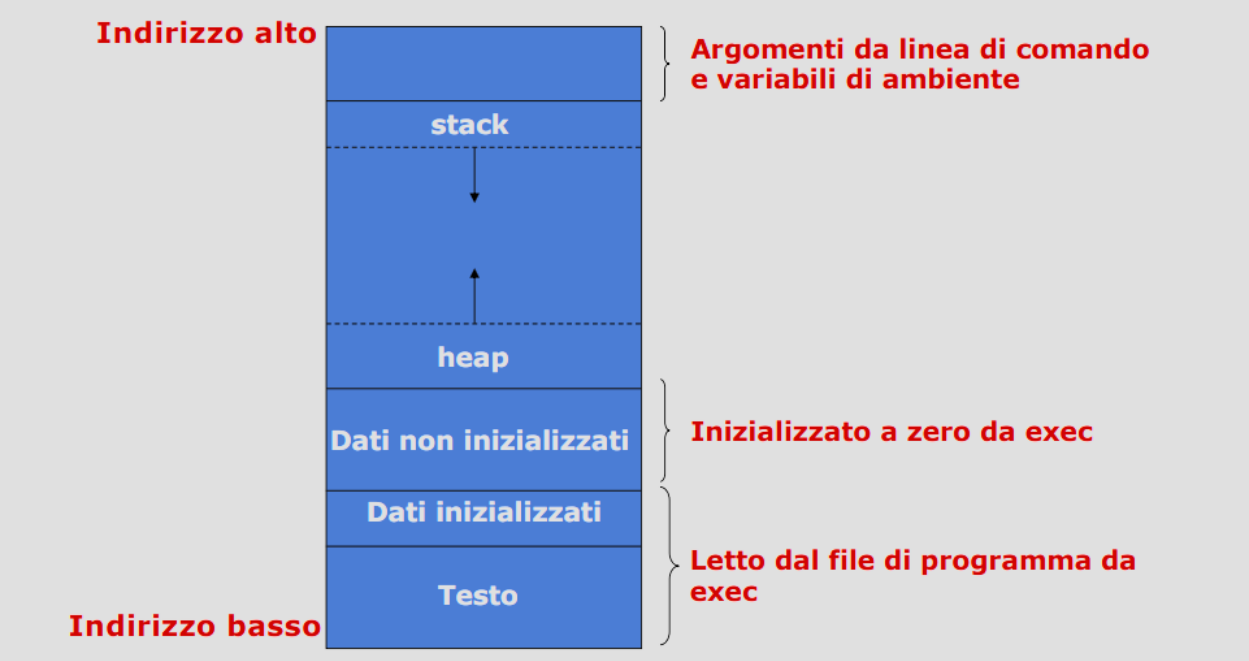
7.8 Struttura della memoria per un programma C

Un programma C é composto dai seguenti pezzi:

- segmento di testo: le istruzioni macchina eseguite dalla CPU, sono condivisibile e a sola lettura.
- Segmento di dati inizializzati: contiene variabili globali e statiche inizializzate nel programma.
- Segmento di dati non inizializzati: le variabili globali e statiche sono inizializzati dal kernel a 0 o al puntatore nullo prima dell'esecuzione.

Lo **stack** contiene le variabili automatiche con le informazioni salvate ogniqualvolta é chiamata una funzione. Indirizzo di ritorno, registri. La funzione chiamata alloca spazio per le sue variabili automatiche e temporanee.

L'**heap** luogo in cui avviene l'allocazione dinamica della memoria.



Il comando `size` riporta la dimensione dei segmenti di testo, dati e bss

7.9 Allocazione della memoria

ISO C specifica tre funzioni:

malloc: alloca un numero specificato di byte di memoria.

calloc: alloca spazio per uno specifico numero di oggetti di dimensione specificata.

realloc: incrementa o decrementa la dimensione di un'area di memoria allocata in precedenza.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc (size_t nobj, size_t size);
void *realloc (void *ptr, size_t newsize);
void free (void *ptr);
```

Le routine di allocazione sono implementate con la SC `sbrk` espande o contrae l'heap del processo.

Molte implementazioni allocanòun poco più spazio di quanto richiesto per memorizzare varie informazioni, tra cui:

Dimensione del blocco allocato.

Puntatore al successivo blocco allocato.

Sorgenti di errore

Scrivere prima della fine di un'area allocata può sovrascrivere le informazioni relative ad un blocco successivo.

Liberare un blocco già liberato da una chiamata a free.

Chiamare free con un puntatore non ottenuto da una delle tre funzioni di allocazione.

Se un processore chiama malloc e dimentica di chiamare free l'uso di memoria continua a crescere.

7.10 Unix e i processi

unix é una famiglia di sistemi multiprogrammati bassati su processi, un pocesso consiste nell'insieme di eventi che scaturiscono durante l'esecuzione di un programma.

È un entità dinamica a cui é associato un inseime di informazioni necessarie per la corretta esecuzione e gestione del processo da parte del sistma operativo.

Il processo unix mantiene spazi di indirizzamento separati per i dati e per il codice, ogni processo ha uno spazio di indirizzamento dei dati privati, non è possibile condividere variabili tra processi diversi. È necessaria un'interazione basata su scambi di messaggi.

A differenza dello spazio di indirizzamento dati, lo spazio di indirizzamento del codicé condivisibile.

Più processi possono eseguire lo stesso programma facendo riferimento alla stessa area di codice nella
memoria centrale

7.10.1 Caratteristiche del processo unix

Processo pesante con codice rientrante, dati non condivisi e codice condivisibile con altri processi.

Funzionamento in doppia modalità, processi utente e processi di sistema.

7.10.2 Stati di un processo unix

Come nel caso generale

- init
- ready
- running
- sleeping
- terminated

Inoltre

- zombie, il processo è terminato ma é in attesa che il padre ne rilevi lo stato di terminazione
- swapped, il processo è temporanemanete trasferiuto in memoria secondaria.

7.10.3 Processi swapped

Lo scheduler a medio termine gestisce i trasferimenti dei processi, da memoria centrale a secondaria **swapped out,** si applica preferibilmente ai processi bloccati prendendo in considerazione tempo di attesa, di permanenza in memoria e dimensione del processo.

Da memoria secondaria a centrale **swapped in** si applica prederibilmente ai processi corti.

7.10.4 Rappresentazione dei processi

Il codice dei processi è rientrante, vale a dire più processi possono condividere lo stesso codice, Codice e dati sono separati. Il SO gestisce una struttura dati globale in cui sono contenuti i puntatori ai segmenti di testo dai processi.

- L'elemento della text table si chiama text structure e contiene tra gli altri Puntatore al segmento di testo (se il processo è in stato di swap, il riferimento alla
  memoria secondaria), Numero dei processi che lo condividono

IL PCB é rappresentato da due strutture dati:

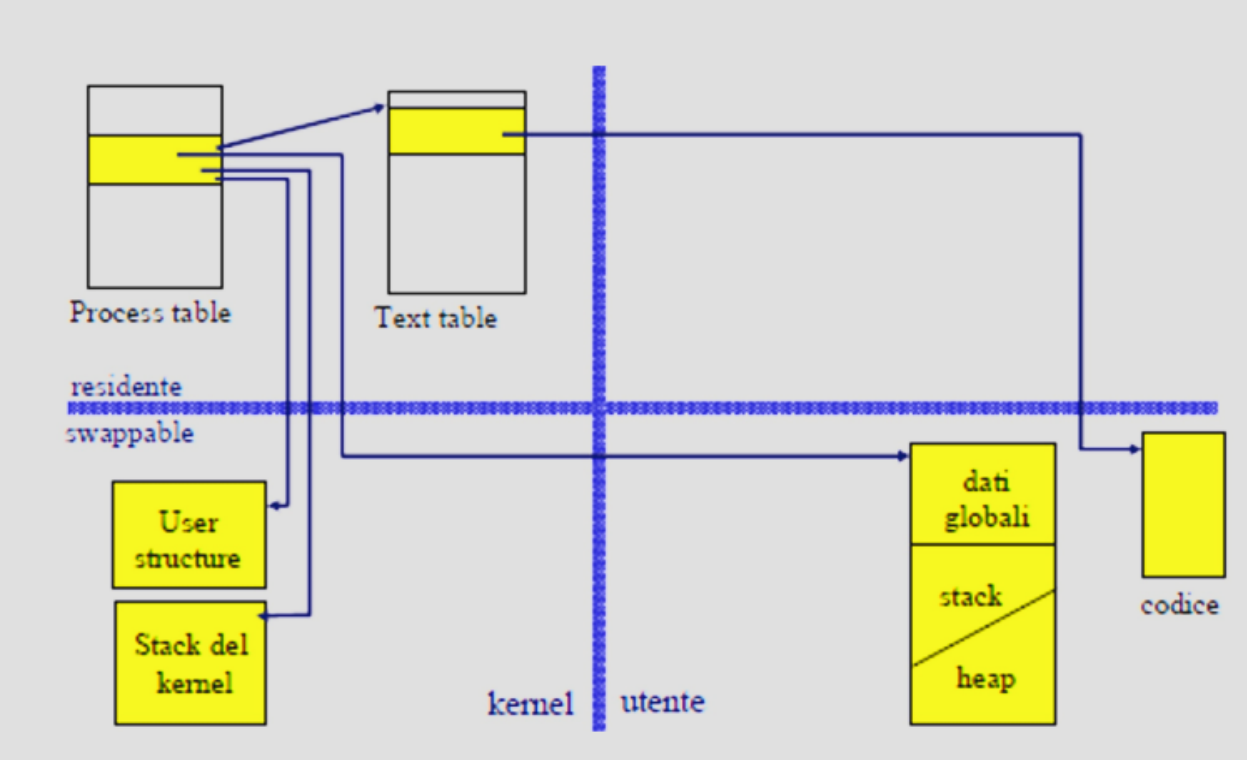
1. Process structure: informazioni necessarie al sistema per la gestione del processo
2. User struscture: informazione necessaria solo se il processo è residente in memoria centrale.

7.10.5 Immagine di un processo

L'immagine di un processo é l'insieme delle aree di memoria e delle strutture dati associate al processo

Non tutta l'immagine è accessibile in modo user. PArte di kernel e parte di utente

Ogni processo può essere soggetto a swapping. Non tutta l'immagine può essere trasferita in memoria.



7.10.6 Immagine di un processo unix

- **Process structure** (kernel, residente), è l'elemento della process table associato al processo
- **Text structure** (kernel, residente), elemento della text table associato al codice del processo
- **Area dati globali utente** (user, swappable),
  - Segmento dati inizializzati
  - Segmento dati non inizializzati
- **Stack, heap utente** (user, swappable), aree dinamiche associate al programma eseguit
- **Stack del kernel**(kernel, swappable), stack di sistema associato al processo per le chiamate a system call
- **U-area**(kernel, swappable), struttura dati contenente i dati necessari al kernel per la gestione del processo quando è

residente

7.11 Processi

All'avvio del SO c'é un solo processo utente visibile chiamato `init()` il cui identificativo numerico unico é sempre 1, é invocato dal kernel alla fine della procedura di bootstrap.

È quindìl'antenato comune di tutti i processi utenti esistenti in un dato momento nel sistema.

Esempio

`init()` crea i processi `getty()` responsabili di gestire i login degli utenti.

Il processo con ID 0 é lo scheduler noto anche come swapper, a tale processo non corrisponde alcun programma su disco poiché è parte del kernel ed é dunque un processo di sistema.

Ogni implementazione di unix ha i propri processi kernel; che forniscono i servizi del sistema operativo.

Ad esempio il processo con ID 2 é il pagedaemon che é responsabile della paginazione del sistema della memoria virtuale

7.12 Chiamata di sistema Fork()

```
#include<unistd.h>
pid_t fork (void)
```

Crea una copia del processo che esegue la fork(), l'area dati viene duolicata e l'area del codice viene condivisa.

Il processo creato riceve esito = 0

Il processo creante riceve esito > 0 che corrisponde all'identificatore di processo del processo creato

Se fallisce allora è -1

Il processo figlio è una copia del genitore cioé essi non condividono parti di memoria, PID E PPID nei processi padre e figlio sono differenti

Una volta invocata una fork non si può sapere se il figlio andrà in esecuzione prima del genitore o dopo

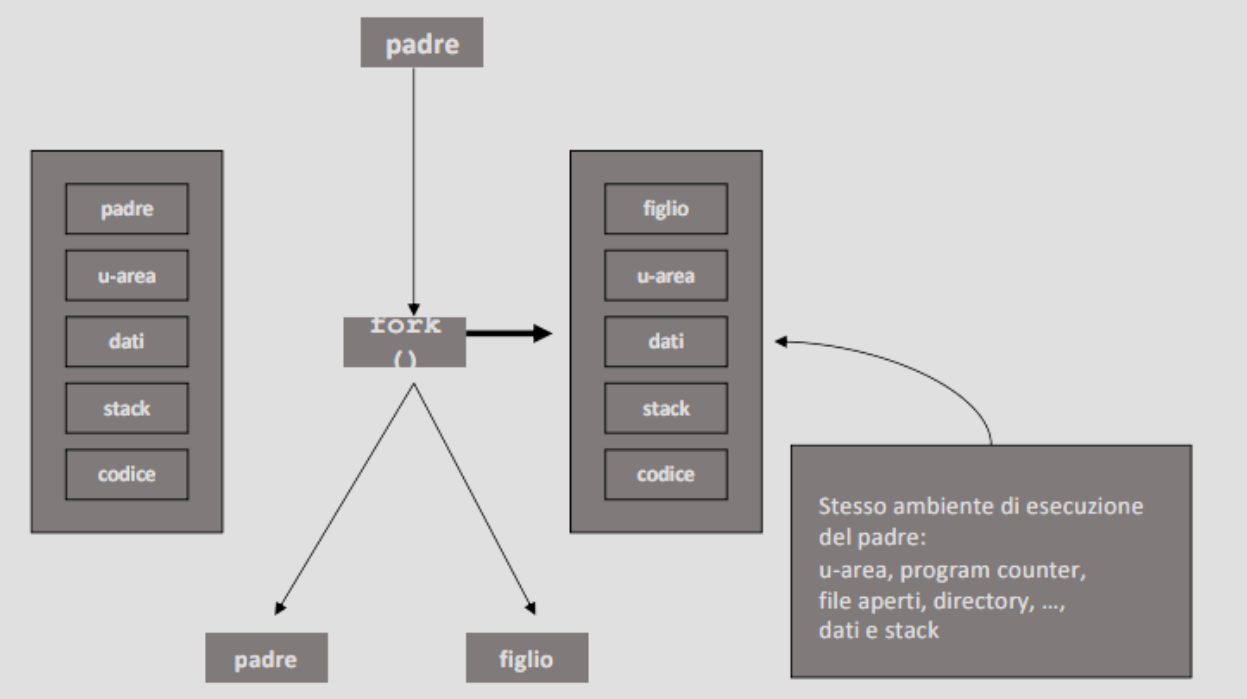
Tutti i descrittori aperti nel genitore sono duplicati nel figlio. Nella tabella dei file, il padre ed il figlio condividono lo stesso elemento per ogni descrittore aperto, condividono cioè lo stesso offset

7.13 Creazione dei processi

Quando un processo è duplicato, il processo padre ed il processo figlio sono virtualmente identici, il codice, i dati e lo stack del figlio sono una copia di quelli del padre ed il
processo figlio continua ad eseguire lo stesso codice del padre e differiscono per alcuni aspetti quali PID, PPID e risorse a run-time (es. segnali pendenti)

Quando un processo figlio termina (tramite una `exit()`), la sua terminazione è comunicata al padre (tramite un segnale) e questi si comporta di conseguenza.

7.14 Creazione di un processo figlio: `fork()`



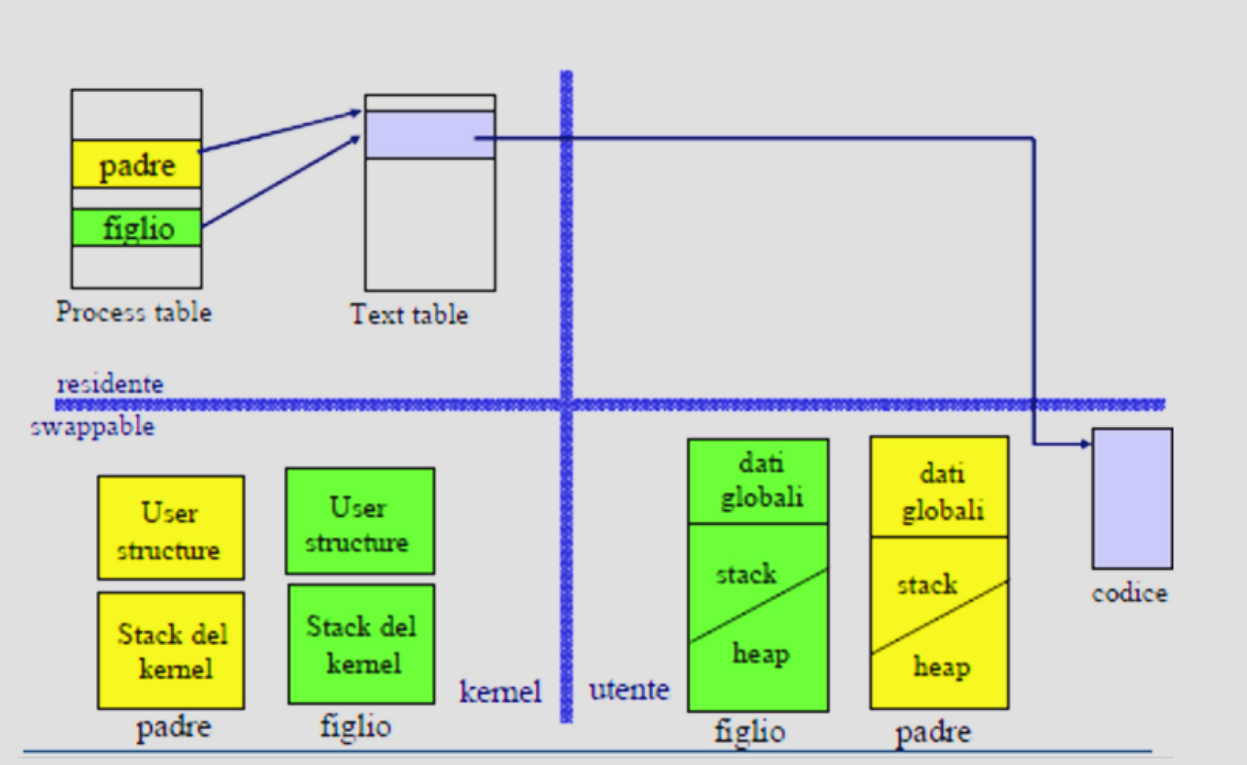
7.15 Effetti della fork()

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 la user structure del padre.

Allocazione dei **segmenti di dati e stack** del figlio nei quali vengono copiati dati e stack del padre.

Aggiornamento della **text structure** del codice eseguito: incremento del contatore dei processi...



7.16 Ottenere il PID: getpid() e getppid()

```
#include<unistd.h>
pid_t getpid (void)
pid_t getppid (void)
```

`getpid()` restituisce il PID del processo invocante

`getppid()` restituisce il PPID del processo invocante

hanno sempre successo, il PPID di init é ancora 1

7.16.1 Esempio: padre e figlio eseguono x = 1 dopo il ritorno dalla fork

```
#include <stdio.h>
#include <unistd.h>
```

```
int main(void) {
    int x;
    x = 0;
    fork();
    x = 1;
    printf("process %d, x = %d\n", getpid(), x);
    return 0;
}
```

7.16.2 Esempio: creazione di una catena di n processi

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    if (argc != 2){ /* controllo argomenti */
        fprintf(stderr, "Uso: %s processi\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if (childpid = fork())
            break;
    printf("i:%d processo ID:%d padre ID:%d figlio ID:%d\n", i,
    exit(0);
}
```

7.17 Identificativo di processo

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void)    //process ID
pid_t getppid(void)   //parent process ID
uid_t getuid(void)    //real user ID
uid_t geteuid(void)   //effective user ID
gid_t getgid(void)    //real group ID
gid_t getegid(void)   //effective group ID
```

Questi identificativi sono interi non negativi

7.17.1 Esempio

```
/* Stampa vari user e group ID per un processo */
#include <stdio.h>
#include <unistd.h>

int main(void) {
    printf(“Mio real user ID: %5d\n", (uid_t)getuid());
    printf("Mio effective user ID:%5d\n", (uid_t)geteuid());
    printf("Mio real group ID:%5d\n", (gid_t)getgid());
    printf("Mio effective group ID:%5d\n", (gid_t)getegid());
    return 0;
}
```

7.17.2 Esempio: myfork.c

```
/* Un programma che si sdoppia e mostra il PID e PPID dei due pr
#include <stdio.h>
```

```
int main (int argc, char *argv[]) {
    int pid;
    printf (“Sono il processo di partenza con PID %d e PPID %d.\
    pid = fork (); /* Duplicazione. Figlio e genitore continuano

    if (pid != 0) { /* pid diverso da 0, sono il padre */
        printf (“Sono il processo padre con PID %d e PPID %d.\n"
        printf (“Il PID di mio figlio e\’ %d.\n", pid); /* non a
    }
    else { /* il pid è 0, quindi sono il figlio */
        printf (“Sono il processo figlio con PID %d e PPID %d.\n
    }
    printf ("PID %d termina.\n",getpid());
    /* Entrambi i processi eseguono questa parte */
    return 0;
}
```

Nell'esempio, il padre non aspetta la terminazione del figlio per terminare a sua volta

Se un padre termina prima di un suo figlio, il figlio diventa orfano e viene automaticamente adottato dal processo `init()`

7.18 Ulteriori informazioni sulla fork()

Il segmento di testo dei processi padre e figlio è condiviso e tenuto in modalità sola lettura per il padre ed i suoi figli.

Per gli altri segmenti linux utilizza la tecnica del copy on write:

Viene effettivamente copiata una pagina di memoria per il nuovo processo solo quando vi viene effettuata sopra una scrittura.

Il meccanismo di creazione di un nuovo processo è molto più efficiente, non è necessaria la copia di tutto lo spazio degli indirizzi virtuali del padre, ma solo delle pagine di memoria che sono state modificate e solo al momento della modifica stessa.

7.19 La chiamata di sistema vfork()

```
#include <unistd.h>
pid_t vfork(void);
```

vfork crea un nuovo processo, esattamente come fork, ma senza copiare lo spazio di indirizzamento.

Fino a che il figlio non esegue una exec o exit, esso viene eseguito nello spazio di indirizzamento del genitore.

La vfork assicura che il figlio venga eseguito per primo, fino a quando questi non chiama exec o exit.

7.19.1 Esempiio vfork()

```
#include “apue.h”
int glob=6; /* variabile esterna (blocco dati inizializzati) */
int main(void)
  {
    int var; /* variabile automatica sullo stack */
        pid_t pid;
        var = 88;
        printf(“prima della fork\n”);
        if ((pid = vfork())<0) {
            err_sys(“errore della vfork”);
```

```
        }
        else if (pid ==0){ /* figlio */
                glob++; /* modifica le variabili */
                var++;
                _exit(0); /* il figlio finisce */
        }
        /* il padre continua qui */
        printf("pid = %d, glob = %d, var = %d\n",getpid(),glob,v
        exit(0);
    }
```

7.19.2 Ulteriori informazioni sulla vfork()

Non viene creata la tabella delle pagine ne la struttura dei task per il nuovo processo. Il processo padre è posto in attesa fintanto non ha eseguito una execve o non è uscito con una _exit.

Il figlio condivide la memoria del padre e non deve ritornare o uscira con una exti ma usare esplicitamente _exit.

Introdotta in BSD per migliorare le prestazioni poichè fork comportava la copia completa del segmento dati del processo padre.