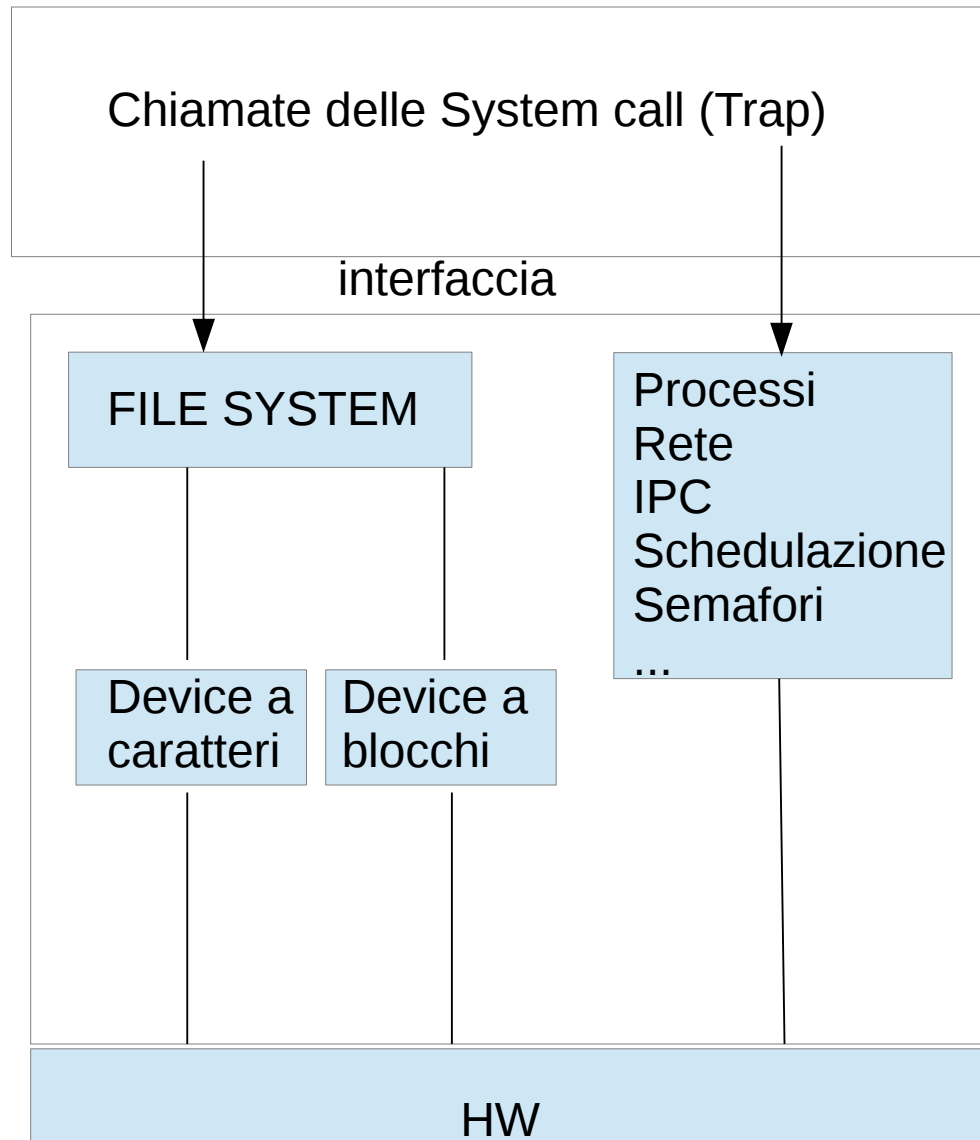


Programmazione di sistema in Linux: System Call per il controllo processi

E. Mumolo, DIA

Struttura generica di Linux



Utente
Shell comandi di linea
Compilatori
Editor, debugger, ..

System
(kernel)

Codice
delle
System call

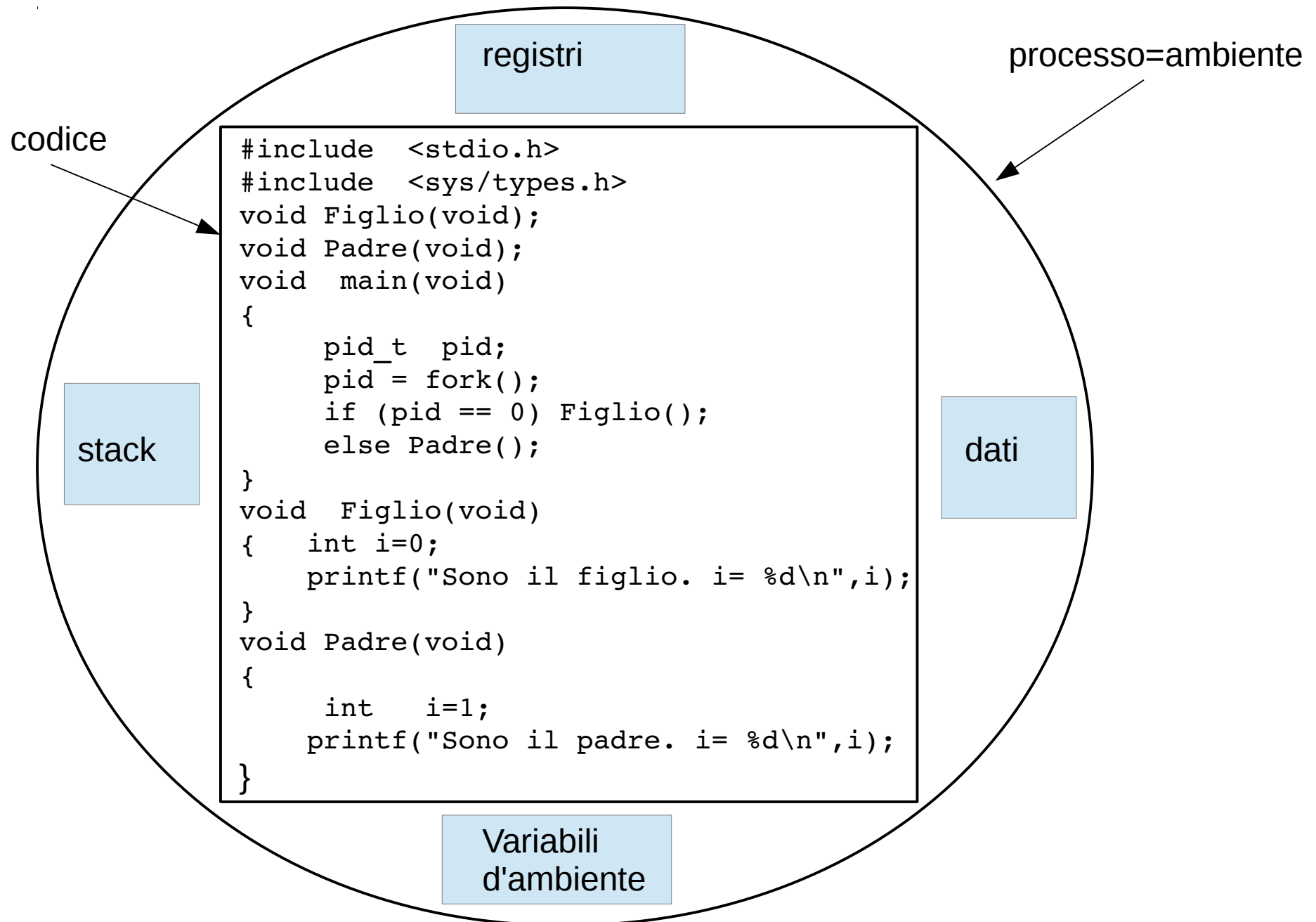
Posix

- Un cenno ai primi Standard Posix:
 - POSIX.1(1003.1a) → interfacce base, supporto per processo singolo e multiplo, controllo dei processi, segnali, file system, pipe, fifo ...
 - POSIX.1(1003.1b) → Estensioni Real Time → segnali, schedulazione prioritaria, timer, I/O asincrono, prioritario, sincrono, messaggi, semafori ...
 - POSIX.1(1003.1c) → Thread → controllo dei thread, attributi, schedulazione, variabili condizione ...
 - POSIX.2 → comandi shell
 - POSIX.6(1003.1e) → sicurezza
 - POSIX.7 → amministrazione di sistema
 - POSIX.8(1003.1f) → accesso ai file di rete
 - ...
- `<sys/types.h>` definisce tipi di dati: `dev_t`, `pid_t`, `size_t`, ...

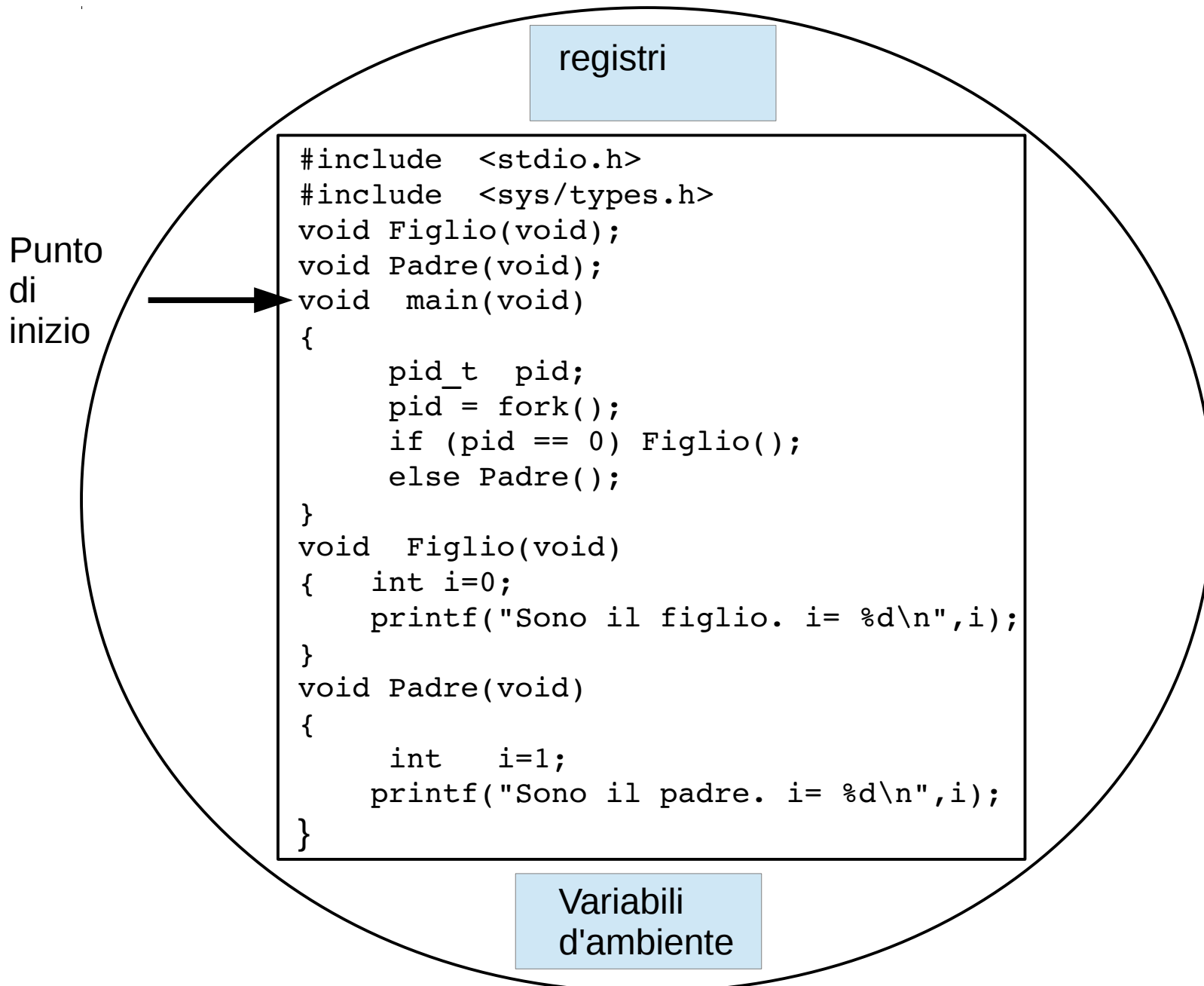
System call per la gestione dei processi

- fork crea un processo
- exec carica un codice eseguibile
- wait aspetta la terminazione del processo
- signal cattura un segnale
- kill invia un segnale

Controllo processi: chiamata di sistema fork()

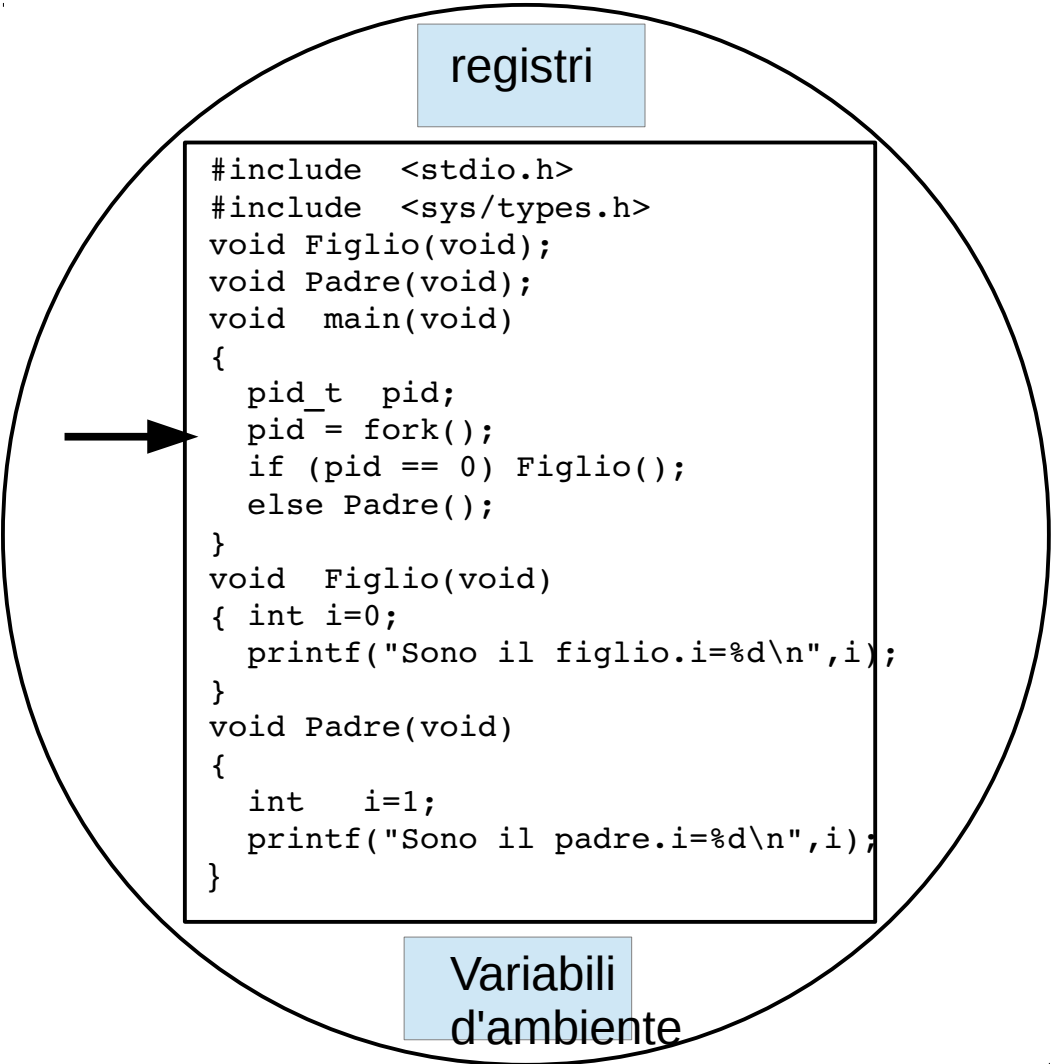


Controllo processi: chiamata di sistema fork()



Controllo processi: chiamata di sistema fork()

registri



The diagram illustrates a process environment. A large circle represents the process. At the top, a light blue box labeled 'registri' (registers) is positioned. At the bottom, a light blue box labeled 'Variabili d'ambiente' (environment variables) is positioned. In the center of the circle is a rectangular box containing C code. An arrow points from the left edge of the circle to the left side of the code box.

```
#include <stdio.h>
#include <sys/types.h>
void Figlio(void);
void Padre(void);
void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0) Figlio();
    else Padre();
}
void Figlio(void)
{ int i=0;
  printf("Sono il figlio.i=%d\n",i);
}
void Padre(void)
{
    int i=1;
    printf("Sono il padre.i=%d\n",i);
}
```

Variabili
d'ambiente

Controllo processi: chiamata di sistema fork()

Nel padre, la fork() ritorna il pid del processo generato

Nel figlio, la fork() ritorna pid=0

registri

```
#include <stdio.h>
#include <sys/types.h>
void Figlio(void);
void Padre(void);
void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0) Figlio();
    else Padre();
}
void Figlio(void)
{ int i=0;
  printf("Sono il figlio.i=%d\n",i);
}
void Padre(void)
{
    int i=1;
    printf("Sono il padre.i=%d\n",i);
}
```

Variabili
d'ambiente

registri

```
#include <stdio.h>
#include <sys/types.h>
void Figlio(void);
void Padre(void);
void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0) Figlio();
    else Padre();
}
void Figlio(void)
{ int i=0;
  printf("Sono il figlio.i=%d\n",i);
}
void Padre(void)
{
    int i=1;
    printf("Sono il padre.i=%d\n",i);
}
```

Variabili
d'ambiente

Controllo processi: chiamata di sistema fork()

pid=???

registri

```
#include <stdio.h>
#include <sys/types.h>
void Figlio(void);
void Padre(void);
void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0) Figlio();
    else Padre();
}
void Figlio(void)
{ int i=0;
  printf("Sono il figlio.i=%d\n",i);
}
void Padre(void)
{
    int i=1;
    printf("Sono il padre.i=%d\n",i);
}
```

Variabili
d'ambiente

pid=0

registri

```
#include <stdio.h>
#include <sys/types.h>
void Figlio(void);
void Padre(void);
void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0) Figlio();
    else Padre();
}
void Figlio(void)
{ int i=0;
  printf("Sono il figlio.i=%d\n",i);
}
void Padre(void)
{
    int i=1;
    printf("Sono il padre.i=%d\n",i);
}
```

Variabili
d'ambiente

Controllo processi: chiamata di sistema fork()

pid=???

registri

```
#include <stdio.h>
#include <sys/types.h>
void Figlio(void);
void Padre(void);
void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0) Figlio();
    else Padre();
}
void Figlio(void)
{ int i=0;
  printf("Sono il figlio.i=%d\n",i);
}
void Padre(void)
{
    int i=1;
    printf("Sono il padre.i=%d\n",i);
}
```

Variabili
d'ambiente

pid=0

registri

```
#include <stdio.h>
#include <sys/types.h>
void Figlio(void);
void Padre(void);
void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0) Figlio();
    else Padre();
}
void Figlio(void)
{ int i=0;
  printf("Sono il figlio.i=%d\n",i);
}
void Padre(void)
{
    int i=1;
    printf("Sono il padre.i=%d\n",i);
}
```

Variabili
d'ambiente

Chiamata di sistema fork()

- System call: **pid_t fork();**
 - crea un nuovo processo figlio, copiando completamente l'immagine di memoria del processo padre (data, heap, stack)
 - La memoria è completamente INDIPENDENTE tra padre e figlio
 - il codice viene generalmente condiviso tra padre e figlio
 - codice copy-on-write (copiato quando viene modificato)
 - tutti i descrittori dei file aperti nel processo padre (UFDT) sono duplicati nel processo figlio
 - sia il processo child che il processo parent continuano ad eseguire l'istruzione successiva alla **fork**
 - processo figlio: ritorna 0
 - processo padre: ritorna il process ID del processo figlio
 - Errore della fork → pid negativo

Chiamata di sistema fork(). Esempio

```
#include <stdio.h>
#include <sys/types.h>
void Figlio(void);
void Padre(void);
void main(void)
{
    pid_t  pid;
    pid = fork();
    if (pid == 0) Figlio();
    else Padre();
}
void Figlio(void)
{
    int i=0;
    for(i=0;i<10;i++){
        usleep(200);
        printf("\tSono il figlio. i= %d\n",i);
    }
}
void Padre(void)
{
    int i=1;
    for(i=0;i<10;i++){
        usleep(250);
        printf("Sono il padre. i= %d\n",i);
    }
}
```

Chiamata di sistema fork()

- Ereditarieta' in **pid_t fork()**;
 - Non tutte le proprieta' del padre vengono ereditate dal figlio
 - Esempio PID, PPID, file locking, operazione semop, timer, ...
 - Le variabili d'ambiente sono ereditate, a parte nuove variabili
 - Nuove variabili della shell vengono ereditate dal figlio col comando `export`
 - `export -p` mostra le variabili della shell esportate

Identificativi dei processi

- **Identificativi di processo:**

- Process ID (PID)
- Parent Process ID (PPID) del processo che l'ha generato
- Process Group ID (PGID) del gruppo di processi al quale appartiene
- Process Real User : UID dell'utente che ha richiesto accesso
- Process Real Group ID: GID dell'utente che ha richiesto accesso
- Process Effective User ID: normalmente uguale al REAL USER ID,
- Process Effective Group ID:

- **Lettura identificativi**

- Process ID del processo: `pid_t getpid();`
- Process ID del processo padre: `pid_t getppid();`
- Real user ID: `uid_t getuid();`
- Real group ID: `gid_t getgid();`
- Effective user ID: `uid_t geteuid();`
- Effective group ID: `gid_t getegid();`

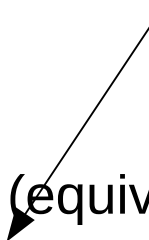
Terminazione processi

- Terminazione dei processi:

- Terminazione normale:

- eseguire un return da main (equivale a chiamare exit());
 - chiamare la funzione **exit()**: chiude gli stream I/O, chiama _exit()
 - Chiamare _exit(): torna al kernel immediatamente


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



- Terminazione anormale:

- Ricezione segnali
 - Chiamando abort()

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



- In ogni caso, l'azione del kernel e' lo stesso:

- Rimozione della memoria utilizzata dal processo
 - Chiusura dei descrittori aperti

- Stato di un processo: raccolto con wait(), waitpid()

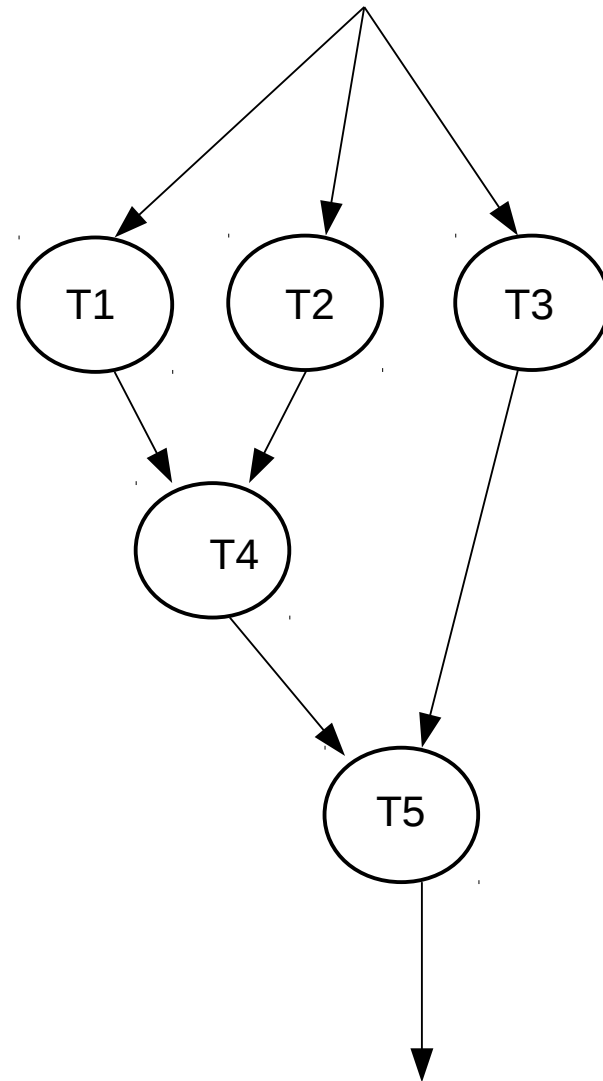
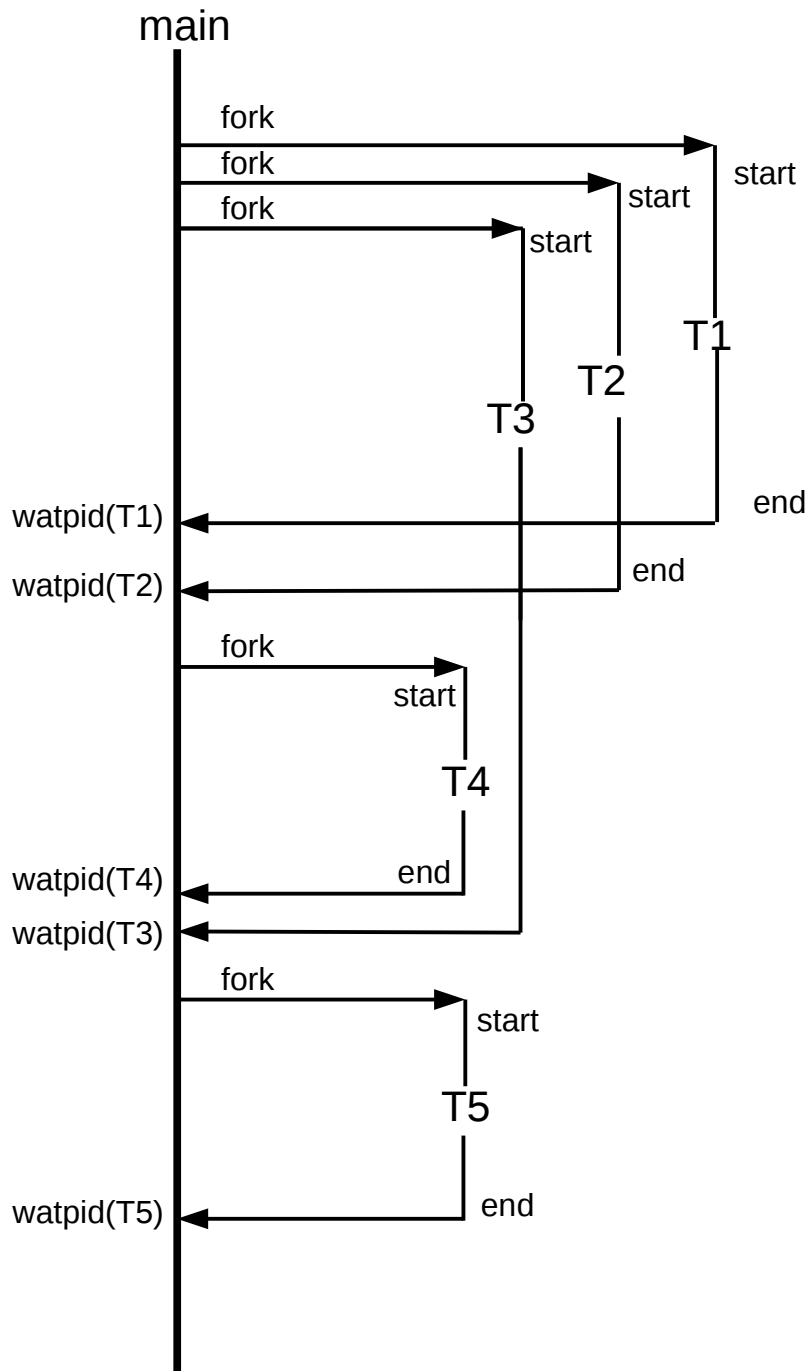
Controllo processi

- **pid_t wait(int *status);**
- **pid_t waitpid(pid_t pid, int *status, int options);**
 - Argomento **pid**:
 - **pid == -1** → attende la prima terminazione
 - **pid > 0** → attende la terminazione del processo generato con process id corrispondente
 - **pid == 0** → attende la terminazione di qualsiasi processo con process group id uguale a quello del processo chiamante
 - **pid < -1** → attende la terminazione di qualsiasi child con process group ID uguale a **-pid**
 - Argomento **status**
 - puntatore ad un intero;
 - se diverso da NULL, contiene lo stato della terminazione
 - Differenza wait/waitpid: wait aspetta il primo figlio


```

/*forkd1.c*/ #include <stdio.h>#include <unistd.h> #include <sys/types.h>
int main(int argc, char *argv[]){
    pid_t cpid, w, t1,t2,t3,t4,t5;          int status;
    printf("Sono il processwo MAIN.  PID %ld, PPID %ld\n", (long) getpid(), (long) getppid());
    t1 = fork();
    if (t1 == 0) { /* codice del figlio */
        printf("processo T1; PID %ld, PPID %ld\n", (long) getpid(), (long) getppid());
        sleep(1); _exit(0);
    } else { /* codice del padre */
        t2 = fork();
        if (t2 == 0) { /* figlio */
            printf("processo T2; PID %ld, PPID %ld\n", (long) getpid(), (long) getppid());
            sleep(1); _exit(0);
        } else {
            t3 = fork();
            if (t3 == 0) { /* figlio */
                printf("processo T3; PID %ld, PPID %ld\n", (long) getpid(), (long) getppid());
                sleep(1); _exit(0);
            } else {
                waitpid(t1, &status,0); printf("sono il padre, ho aspettato t1\n");
                waitpid(t2, &status,0); printf("sono il padre, ho aspettato t2\n");
                t4 = fork();
                if (t4 == 0) { /* figlio */
                    Printf("processo T4; PID %ld, PPID %ld\n", (long) getpid(), (long) getppid());
                    Sleep(1); _exit(0);
                } else {
                    waitpid(t4, &status,0); printf("sono il padre, ho aspettato t4\n");
                    waitpid(t3, &status,0); printf("sono il padre, ho aspettato t3\n");
                    t5 = fork();
                    if (t5 == 0) { /* figlio */
                        printf("proc. T5; PID %ld, PPID %ld\n", (long) getpid(), (long) getppid());
                        sleep(1); _exit(0);
                    } else { /* padre */
                        waitpid(t5, &status,0); printf("sono il padre, ho atteso t5\n");
                    }
                }
            }
        }
    }
}
}
}

```



Controllo processi

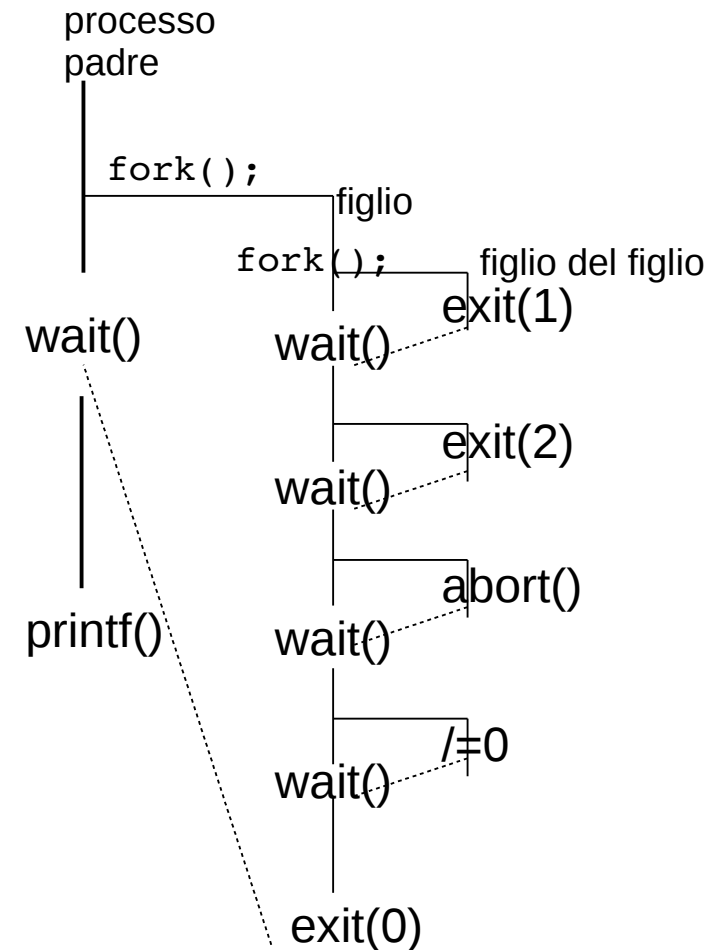
- **Cosa succede se il padre termina prima del figlio?**
 - il processo child viene "adottato" dal processo **init** (pid 1): quando un processo termina, il kernel esamina la tabella dei processi per vedere se aveva figli; in caso affermativo, il PPID viene posto uguale a 1
- **Cosa succede se il figlio termina prima del padre?**
 - il processo va in **Stato 'zombie'**
 - vengono mantenute le informazioni che sono richieste dal padre tramite **wait** e **waitpid**
 - il processo resterà uno zombie fino a quando il parent non eseguirà una delle system call **wait**
- **I processi figli del processo init non possono diventare zombie**
 - tutte le volte che un child di **init** termina, **init** esegue una chiamata **wait** e raccoglie eventuali informazioni
 - in questo modo gli zombie vengono eliminati

Controllo processi

```
#include <sys/types.h>    //pid_t
#include <unistd.h>        //fork()
#include <sys/wait.h>      //waitpid
#include <stdio.h>         //printf...
#include <stdlib.h>        //exit()

int main(void)
{
    pid_t pid;
    int status;

    pid=fork();
    if(pid!=0) {
processo padre { printf("pid1=%d\n",pid);
                 wait(&status);printf("status1=%d\n",status);
                 exit(0);
    }else{
processo figlio { pid=fork();      if(pid==0) exit(1);
                  printf("\tpid=%d ppid=%d\n",pid,getpid());
                  wait(&status);printf("\tstatus=%d\n",status);
                  pid=fork();     if(pid==0) exit(2);
                  printf("\tpid2=%d\n",pid);
                  wait(&status);printf("\tstatus2=%d\n",status);
                  pid=fork();     if(pid==0) abort();
                  printf("\tpid3=%d\n",pid);
                  wait(&status);printf("\tstatus3=%d\n",status);
                  pid=fork();     if(pid==0) status /= 0;
                  printf("\tpid4=%d\n",pid);
                  wait(&status);printf("\tstatus4=%d\n",status);
                  exit(0);
    }
}
```



NB:

exit(n) ritorna il valore $n*256$ al processo padre

abort() e /=0 inviano un segnale al processo figlio

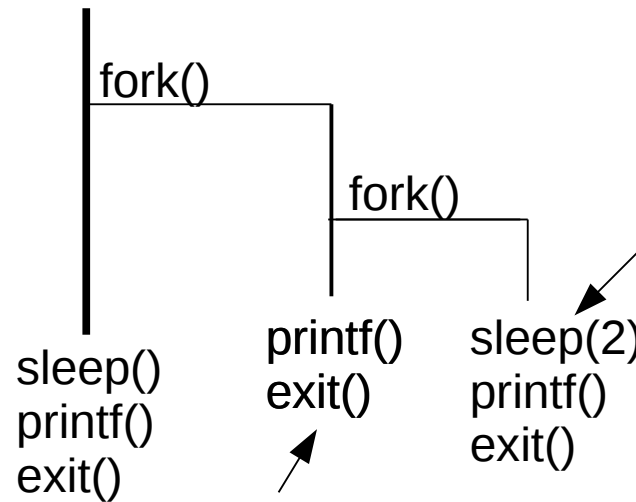
Controllo processi

```
/* processi ZOMBIE */

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

int main(void)
{
    pid_t    pid;

    printf("ID del main=%d\n", getpid());
    pid = fork();
    if (pid == 0) {          /* primo processo figlio*/
        pid = fork();
        if (pid > 0)
            {printf("\tsono il 1o figlio. ID=%d PPID=%d\n", getpid(), getppid());
             exit(0);
            }
        /* il primo figlio termina */
        else{
            sleep(2);
            printf("\t\tsono il 2o figlio. L'ID di mio padre e' = %d\n", getppid());
            exit(0);
        }
    }
    /* A questo punto sono tornato al processo originale cioe' il padre dei due figli*/
    sleep(1);
    printf("sono il processo %d\n", getpid());
    exit(0);
}
```



Qui siamo all'interno del secondo processo figlio; non appena suo padre (il primo processo figlio) chiama exit(), viene adottato. Il secondo processo figlio continua ...

Riepilogo...

Un codice d'esempio di un processo

```
void main(int argc, char *argv[])
{
→   int i,fd, numero_righe,newlen,oldlen;
→   int size=100, n;
    char **righe, *str="ieri";
    Char testo[100], *p;
    FILE *fp;

    righe=(char**)malloc(100*sizeof(char*));
    numero_righe=atoi(argv[2]);
    fd=open(argv[1],O_WRONLY);
    fp=fopen(argv[1],"r");
→   oldlen=0;
→   fork();
→   fgets(testo, size, fp);
    newlen=strlen(testo);
    righe[i]=malloc(strlen(testo));
    strcpy(righe[i],testo);
    p = strstr(righe[i], "oggi");
    n = p-righe[i]+oldlen;
    lseek(fd,n,SEEK_SET);
    write(fd,str,4);
    oldlen+=newlen;
    close (fp);
    close(fd);
}
```

Nuovo processo creato per duplicazione

```
void main(int argc, char *argv[])
{
    int i,fd, numero_righe,newlen,oldlen;
    int size=100, n;
    char **righe, *str="ieri";
    Char testo[100], *p;
    FILE *fp;

    righe=(char**)malloc(100*sizeof(char*));
    numero_righe=atoi(argv[2]);
    fd=open(argv[1],O_WRONLY);
    fp=fopen(argv[1],"r");
    oldlen=0;
    fork();
→   fgets(testo, size, fp);
    newlen=strlen(testo);
    righe[i]=malloc(strlen(testo));
    strcpy(righe[i],testo);
    p = strstr(righe[i], "oggi");
    n = p-righe[i]+oldlen;
    lseek(fd,n,SEEK_SET);
    write(fd,str,4);
    oldlen+=newlen;
    close (fp);
    close(fd);
}
```

Riepilogo...

Il codice d'esempio

Nuovo processo creato

```
void main(int argc, char *argv[])
→ {
    int i, fd, numero_righe, newlen, oldlen;
    int size=100, n;

→    fork();
    wait();
```

...Attesa...

```
void main(int argc, char *argv[])
{
    int i, fd, numero_righe, newlen, oldlen;
    int size=100, n;

    fork();
    wait();

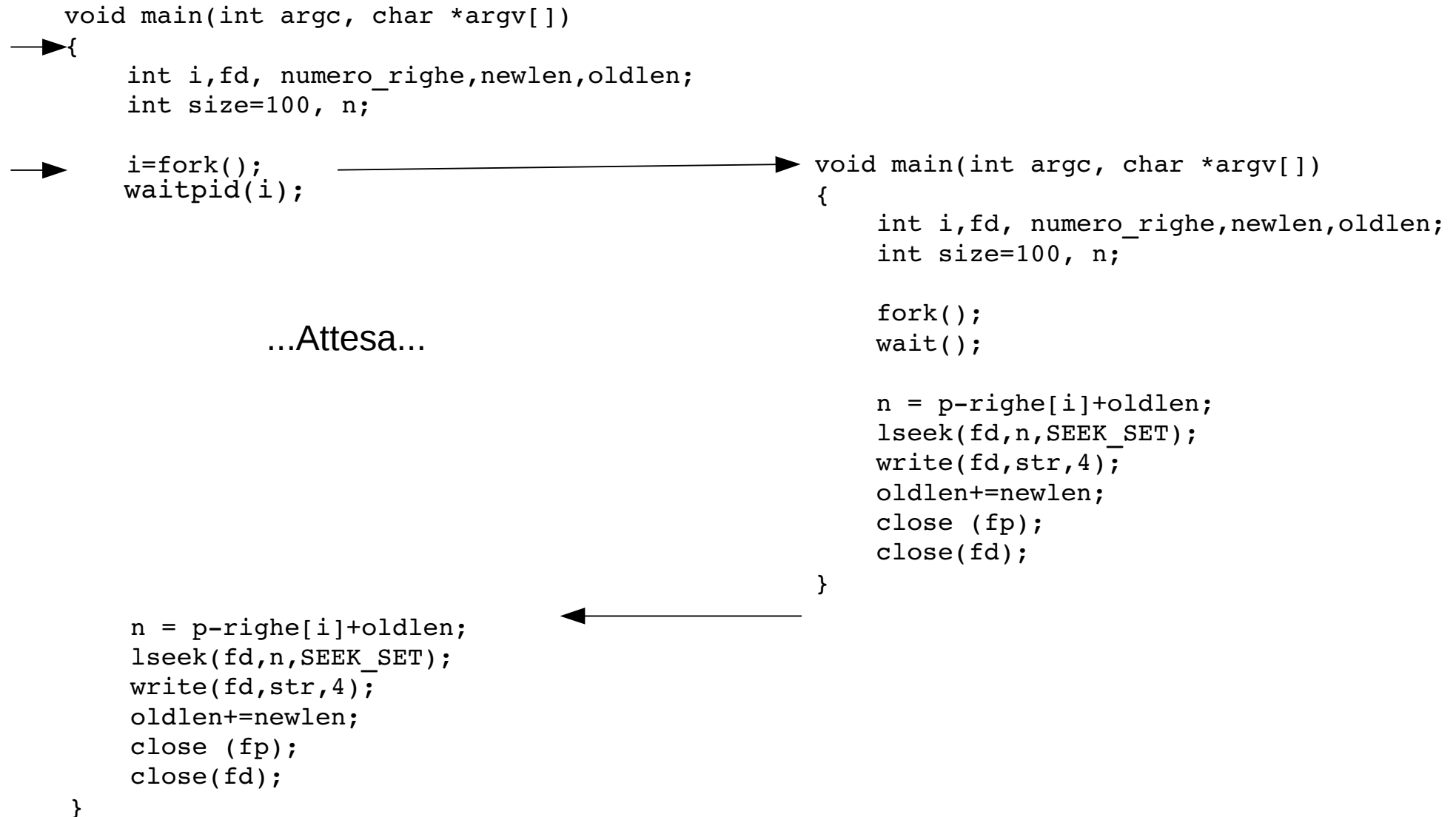
    n = p-righe[i]+oldlen;
    lseek(fd, n, SEEK_SET);
    write(fd, str, 4);
    oldlen+=newlen;
    close (fp);
    close(fd);

}
```

```
    n = p-righe[i]+oldlen;
    lseek(fd, n, SEEK_SET);
    write(fd, str, 4);
    oldlen+=newlen;
    close (fp);
    close(fd);
}
```

Riepilogo...

usando waitpid:



Riepilogo...

- Fork() ritorna il pid del nuovo processo al processo chiamante fork()
- Fork() ritorna 0 al processo generato
- Il nuovo processo eredita dal processo che ha eseguito la fork():
 - Variabili d'ambiente
 - La UFDT (file aperti)
 - Process Real User ID
 - Process Real Group ID
 - Process Effective User ID
 - Process Effective Group ID
- I processi creati vengono rimossi da Linux se terminano mentre sono attesi da wait()
- Se termina senza essere atteso da wait diventa Zombie
- Se il processo chiamante la fork termina prima del figlio, il processo creato viene adottato dal processo 1

Chiamata di sistema exec()

- **Quando un processo chiama una delle system call exec :**
 - Il processo viene rimpiazzato COMPLETAMENTE dal codice contenuto nel file specificato (text, data, heap, stack vengono sostituiti)
 - il nuovo programma inizia a partire dalla sua funzione main
 - il process id non cambia
- **Esistono sei versioni di exec:**
 - `int execl(char *pathname, char *arg0, ...);`
 - `int execv(char *pathname, char *argv[]);`
 - `int execlp(char *pathname, char *arg0, ..., char* envp[]);`
 - `int execvp(char *pathname, char *argv[] , char* envp[]);`
 - `int execlp(char *filename, char *arg0, ...);`
 - `int execvp(char *filename, char *argv[]);`

Chiamata di sistema exec()

- **Cosa viene ereditato da exec?**

- UFDT (file aperti)
- Variabili d'ambiente
- PID e PPID
- real uid e real gid
- current working directory (variabile PWD)
- root e home directory
- maschera creazione file (umask)
- segnali in attesa

Chiamata di sistema exec()

- **Cosa non viene ereditato da exec?**
 - effective user id e effective group id
 - vengono settati in base ai valori dei bit di protezione
- **Cosa succede ai file aperti?**
 - Dipende dal flag close-on-exec che è associato ad ogni descrittore
 - Se close-on-exec è true, vengono chiusi
 - Altrimenti, vengono lasciati aperti (comportamento di default)

Esempio d'uso di exec()

- Questo processo sovrascrive il suo ambiente col codice eseguibile `all`

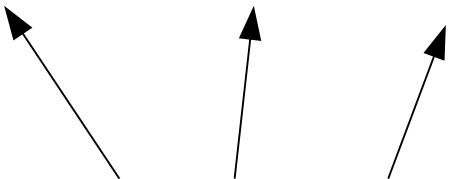
```
#include <stdlib.h>      /* 'errno' */
#include <unistd.h>      /* 'execle' */
#include <stdio.h>       /* 'printf' and 'NULL' */
#include <errno.h>       /* 'ENOENT' and 'ENOMEM' */

int main(void)
{
    pid_t    pid;

    if (execlp("/home/mumolo/all", "all", "primo_arg", "secondo_arg", "terzo_arg", NULL) < 0)
        err_sys("execlp error");

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    exit(0);
}
```



Gli argomenti passati sono sempre stringhe!

- Sorgente di `all`

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    printf("Sono il processo all. I miei argomenti passati da exec sono:\n");
    for (i = 1; i < argc; i++) /* fa l'eco di tutti gli argomenti */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Esempio d'uso di exec()

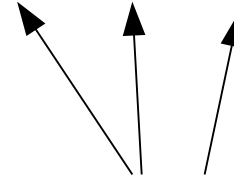
- Questo processo sovrascrive il suo ambiente col codice eseguibile `all1`

```
#include <stdlib.h>      /* 'errno' */
#include <unistd.h>      /* 'execle' */
#include <stdio.h>       /* 'printf' and 'NULL' */
#include <errno.h>       /* 'ENOENT' and 'ENOMEM' */

int main(void)
{
    pid_t    pid;
    if (execlp("/home/mumolo/all1", "all1", "5", "10", "5700", NULL) < 0)
        err_sys("execlp error");

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    exit(0);
}
```

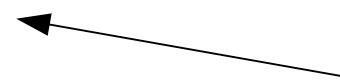


Gli argomenti passati sono sempre stringhe!

- Sorgente di `all1`

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i, buf[10];
    printf("Sono il processo all. I miei argomenti passati da exec sono numeri:\n");
    for (i = 1; i < argc; i++) /* trasforma gli argomenti in int */
        buf[i]=atoi(argv[i]);

    for (i = 1; i < argc; i++)
        printf("buf[%d]: %d\n", i, buf[i]);
    exit(0);
}
```



Le stringhe passate vengono trasformate in int!

Esempio d'uso di exec()

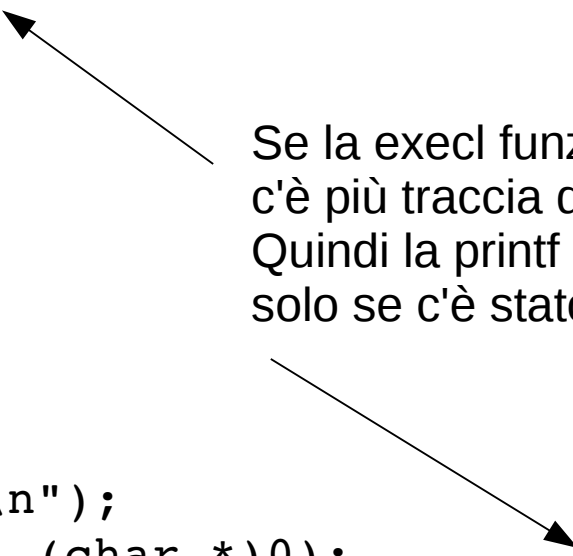
- Esecuzione processo ls

```
#include <stdio.h>
#include <unistd.h>
main()
{
    printf("Esecuzione di ls\n");
    execl("/bin/ls", "ls", "-l", (char *)0);
    printf("La chiamata di execl ha generato un errore\n");
    exit(1);
}
```

- Esecuzione processo ps

```
#include <stdio.h>
#include <unistd.h>
main()
{
    printf("Esecuzione di ps -f\n");
    execl("/bin/ps", "ps", "-f", (char *)0);
    printf("La chiamata di execl ha generato un errore\n");
    exit(1);
}
```

Se la execl funziona, non c'è più traccia della printf!
Quindi la printf viene eseguita solo se c'è stato un problema.



Uso di exec() e fork()

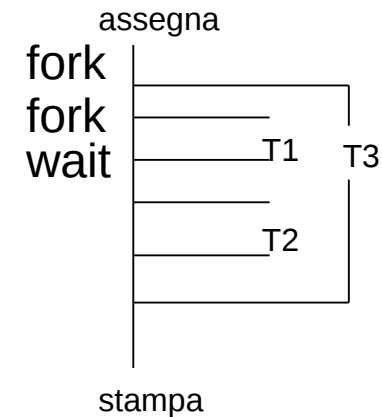
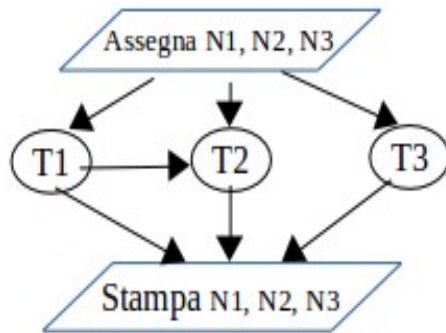
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
```

```
main()
{
    pid_t pid;
    int status;

    switch(pid = fork()) {
        case -1:
            fatal("fork failed"); break;
        case 0:
            execl("/bin/ls", "ls", "-l", (char *)0);
            sys_err("exec failed");
            break;
        default:
            wait(&status);
            printf("ls completato\n");
            exit(0);
    }
}
```

Che succede se non aspetto?

Implementazione di grafi delle precedenze



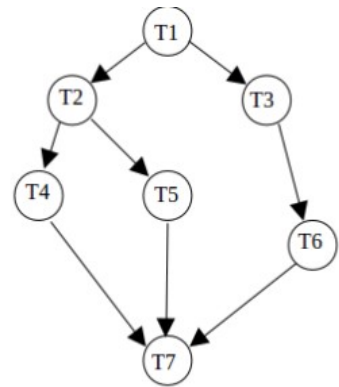
```

int main(int argc, char *argv[]){
    pid_t cpid, w, t1,t2,t3,t4,t5;

    t3 = fork();
    if (t3 == 0) { /* codice del figlio */
        execl("/home/T3", (char *)0);
    } else { /* codice del padre */
        t1 = fork();
        if (t1 == 0) { /* codice del figlio */
            execl("/home/T1", (char *)0);
        } else { /* codice del padre */
            waitpid(t1, NULL, 0)

            t2 = fork();
            if (t2 == 0) { /* codice del figlio */
                execl("/home/T2", (char *)0);
            } else { /* codice del padre */
                waitpid(t2, NULL, 0);
                waitpid(t3, NULL, 0);
            }
        }
    }
}
  
```

Implementazione di grafi delle precedenze



```
int main(int argc, char *argv[]){
    pid_t cpid, w, t1,t2,t3,t4,t5;
```

```
    t1 = fork();
```

```
    if (t1 == 0) { /* codice del figlio */
        execl("/home/T1", (char *)0);
```

```
    } else { /* codice del padre */
        waitpid(t1, NULL, 0);
```

```
        t3 = fork();
```

```
        if (t3 == 0) { /* codice del figlio */
            execl("/home/T3", (char *)0);
```

```
        } else {
```

```
            t2 = fork();
```

```
            if (t2 == 0) { /* codice del figlio */
                execl("/home/T2", (char *)0);
```

```
            } else {
```

```
                waitpid(t2, NULL, 0);
```

```
                t4 = fork();
```

```
                if (t4 == 0) { /* codice del figlio */
                    execl("/home/T4", (char *)0);
```

```
                } else { /* codice del padre */
```

```
                    t5 = fork();
```

```
                    if (t5 == 0) { /* codice del figlio */
                        execl("/home/T5", (char *)0);
```

```
                    } else { /* codice del padre */
```

```
                        t6 = fork();
```

```
                        if (t6 == 0) { /* codice del figlio */
                            execl("/home/T6", (char *)0);
```

```
                        } else { /* codice del padre */
```

```
                            waitpid(t5, NULL, 0);
```

```
                            waitpid(t4, NULL, 0);
```

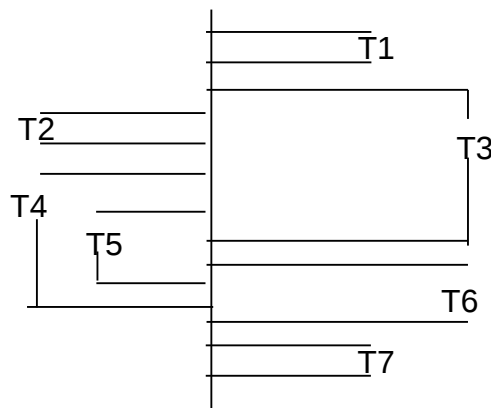
```
                            waitpid(t6, NULL, 0);
```

```
                    }
```

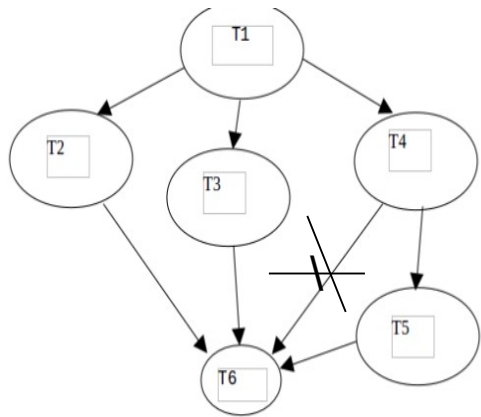
```
                }
```

```
            }
```

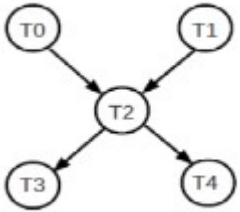
```
    }
```



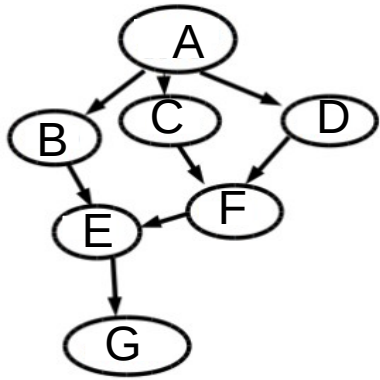
Implementazione di grafi delle precedenze



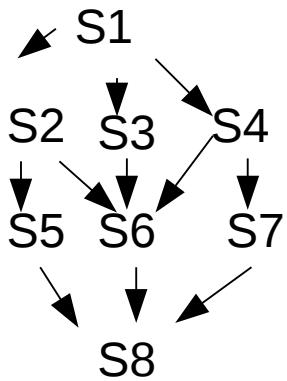
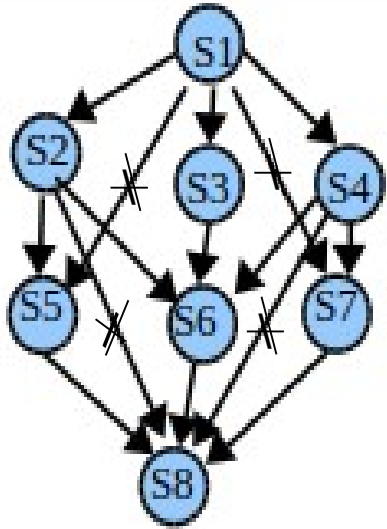
Implementazione di grafi delle precedenze



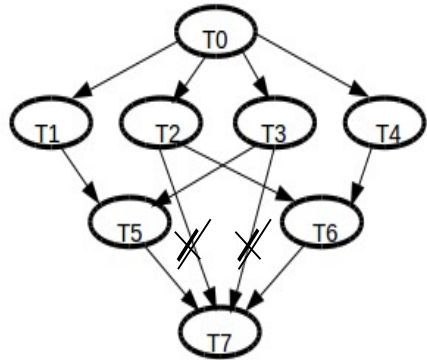
Implementazione di grafi delle precedenze



Implementazione di grafi delle precedenze



Implementazione di grafi delle precedenze



L'ambiente di un processo

- Un ambiente è rappresentato da un vettore di puntatori a caratteri terminato da un puntatore nullo.
- Ogni puntatore (che non sia quello nullo) punta ad una stringa della forma: identificatore = valore
- Per accedere all'ambiente da un programma C, è possibile:

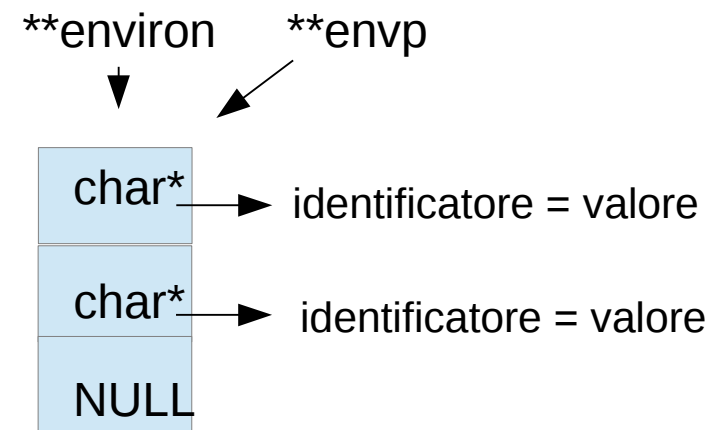
- aggiungere il parametro envp a quelli del main:

```
/* showmyenv */  
#include <stdio.h>  
main(int argc, char **argv, char **envp)  
{  
    while(*envp) printf("%s\n", *envp++);  
}
```

- oppure usare la variabile globale seguente:

```
extern char **environ;
```

- Usare il comando `env`



Stampa dell'ambiente

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

int main(int argc, char *argv[])
{
    int      i;
    char     **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* fa l'eco di tutti gli argomenti */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++) /* e di tutto l'ambiente */
        printf("%s\n", *ptr);

    exit(0);
}
```

Primo programma di shell

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define MAXLINE 128
int main(void)
{
    char    buf[MAXLINE];
    pid_t   pid;
    int     status;
    printf("%s "); /* prompt */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n') buf[strlen(buf) - 1] = 0;
        if ((pid = fork()) < 0) {
            printf("errore di fork "); exit(1);
        } else if (pid == 0) { /*figlio */
            execlp(buf, buf, NULL);
            printf("non posso eseguire: %s", buf);
            exit(127);
        } else
            if ((pid = waitpid(pid, &status, 0)) < 0) /* padre */
                {printf("errore di waitpid"); exit(1);}
        printf("%s ");
    }
    exit(0);
}
```