SC che operano su processi

Fork, exec, etc...

Process identifier: getpid, getppid

- *pid* indice del processo all'interno della tabella dei processi
- *ppid* indice del processo padre all'interno della tabella dei processi
- si possono ottenere con le SC

```
#include <unistd.h>


pid_t getpid(void)
/* returns process ID (no errur return)*/
pid_t getppid(void)
/* returns parent process ID (no errur
   return)*/
```

PID: getpid, getppid (2)

```
/* stampa il pid del processo in esecuzione e
   quello del padre */

   …
   fprintf(stdout,"Processo %d, mio padre e' /
            %d\n.", (int)getpid(),(int)getppid());
   …
```

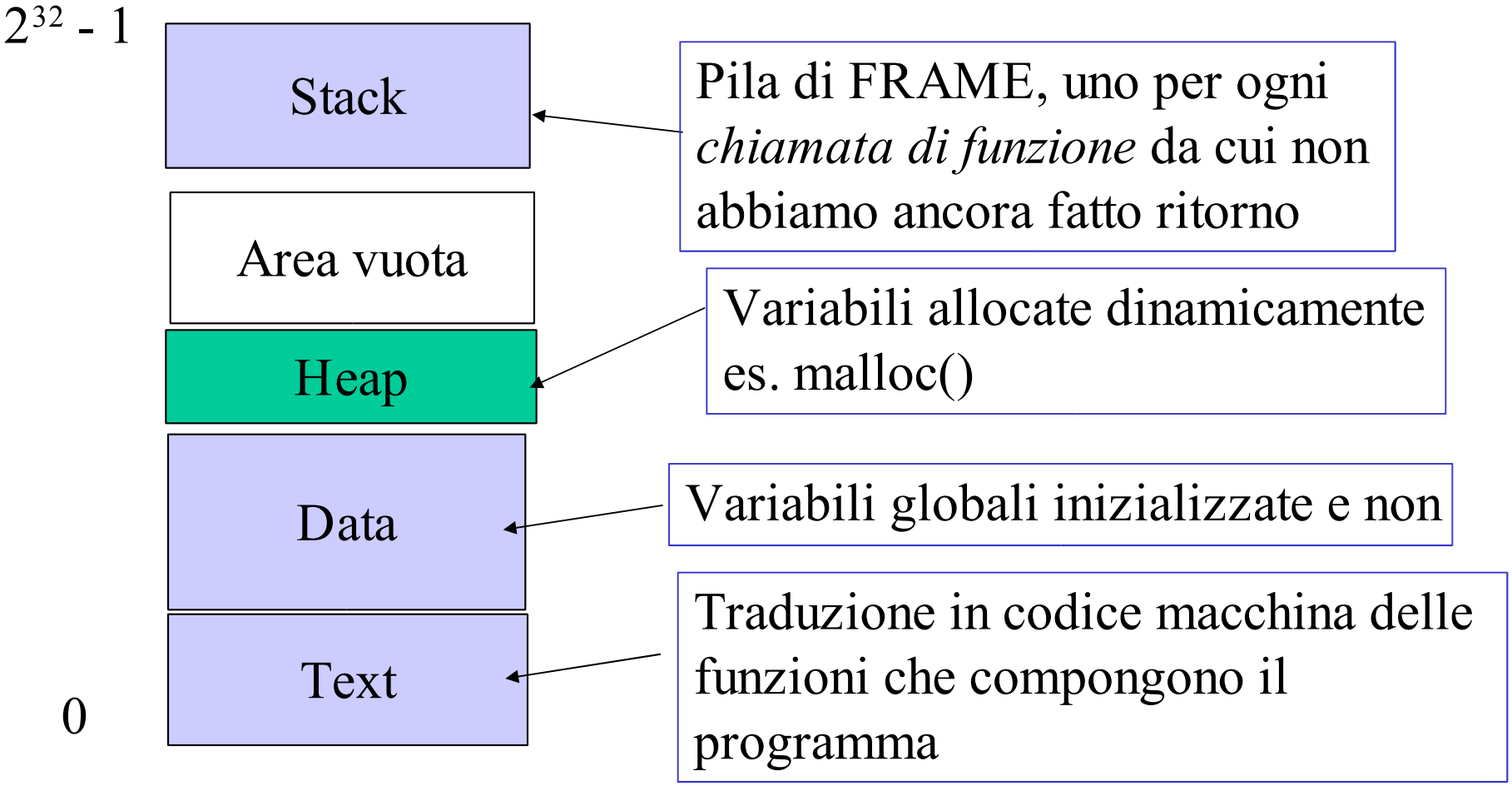
Creazione : fork()

```
#include <unistd.h>
pid_t fork(void)
/* returns process ID (father) or 0 (child)
   [on success] or -1 [on error, sets errno] */
```

- crea un nuovo processo
- lo spazio di indirizzamento del nuovo processo è un duplicato di quello del padre
- padre e figlio hanno due *tabelle dei descrittori di file* diverse (il figlio ha una copia di)
- *condividono la tabella dei file aperti*
  - e quindi anche il puntatore alla locazione corrente di ogni file

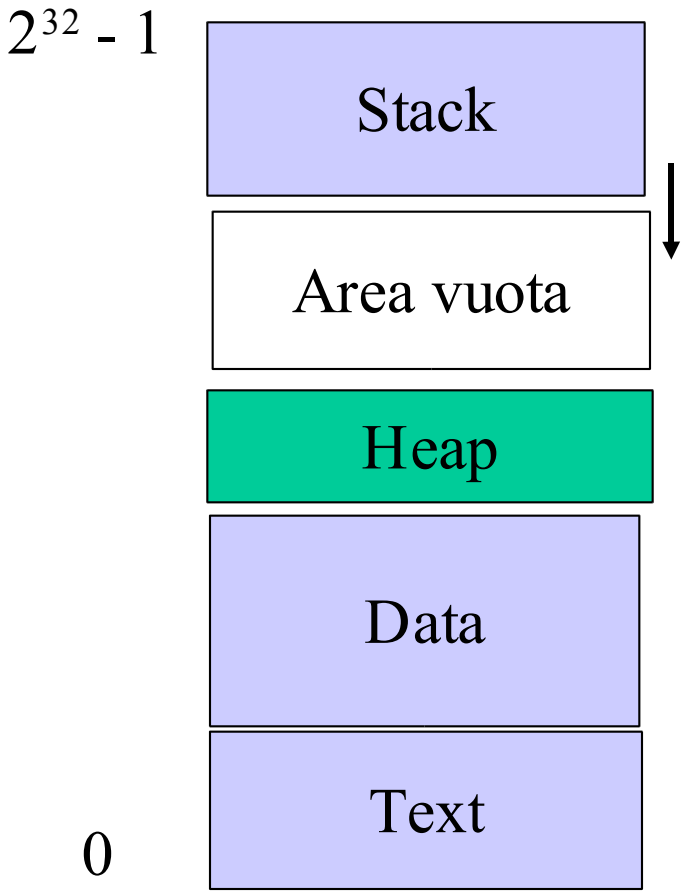
Spazio di indirizzamento

- Come vede la memoria un programma C in esecuzione



Spazio di indirizzamento (2)

- Lo stack

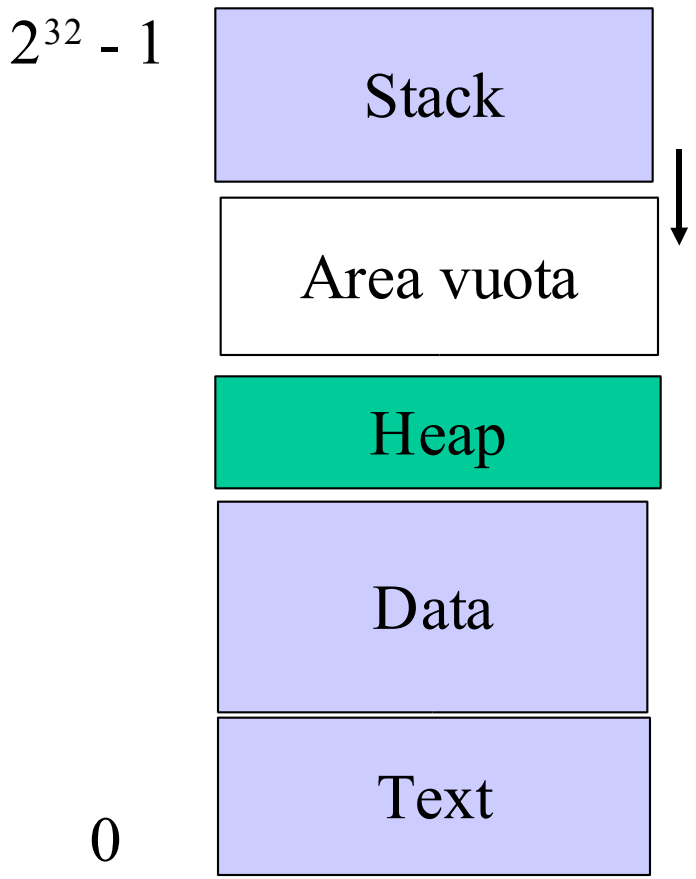


Direzione di
crescita dello stack

Contenuti tipici di un FRAME :
- variabili locali della funzione
- indirizzo di ritorno (indirizzo dell'istruzione successiva a quella che ha effettuato la chiamata alla funzione)
- copia del valore parametri attuali

Spazio di indirizzamento (3)

- Lo stack

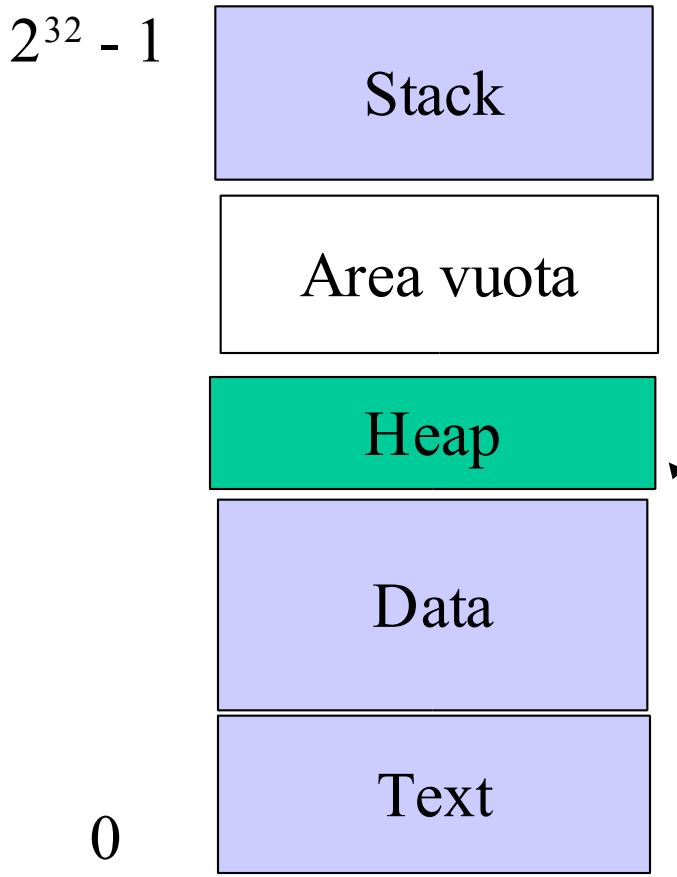


All'inizio dell'esecuzione lo Stack contiene solo il FRAME per la funzione `main`

Successivamente :

* ogni volta che viene chiamata una nuova funzione viene inserito un nuovo frame nello stack
* ogni volta che una funzione termina (`es. return 0`) viene eliminato il frame in cima dello stack e l'esecuzione viene continuata a partire dall'*indirizzo di ritorno*

Spazio di indirizzamento (4)

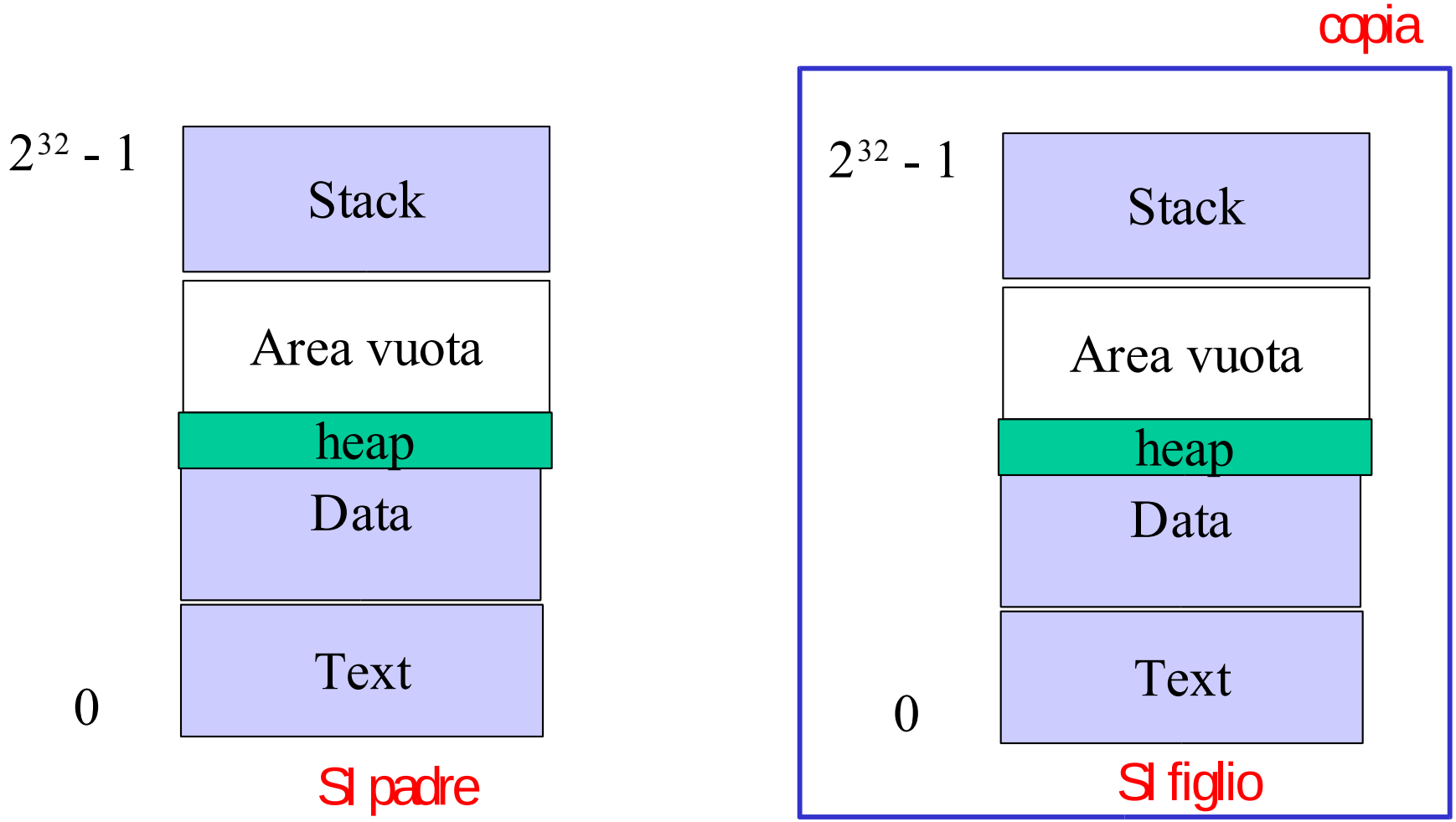


Direzione di
crescita dello *heap*

malloc() calloc() realloc() fanno crescere lo heap verso l'alto
la memoria viene deallocata solamente con chiamate alla funzione free() . Se la free() non viene invocata la memoria rimane occupata (*memory leak*)

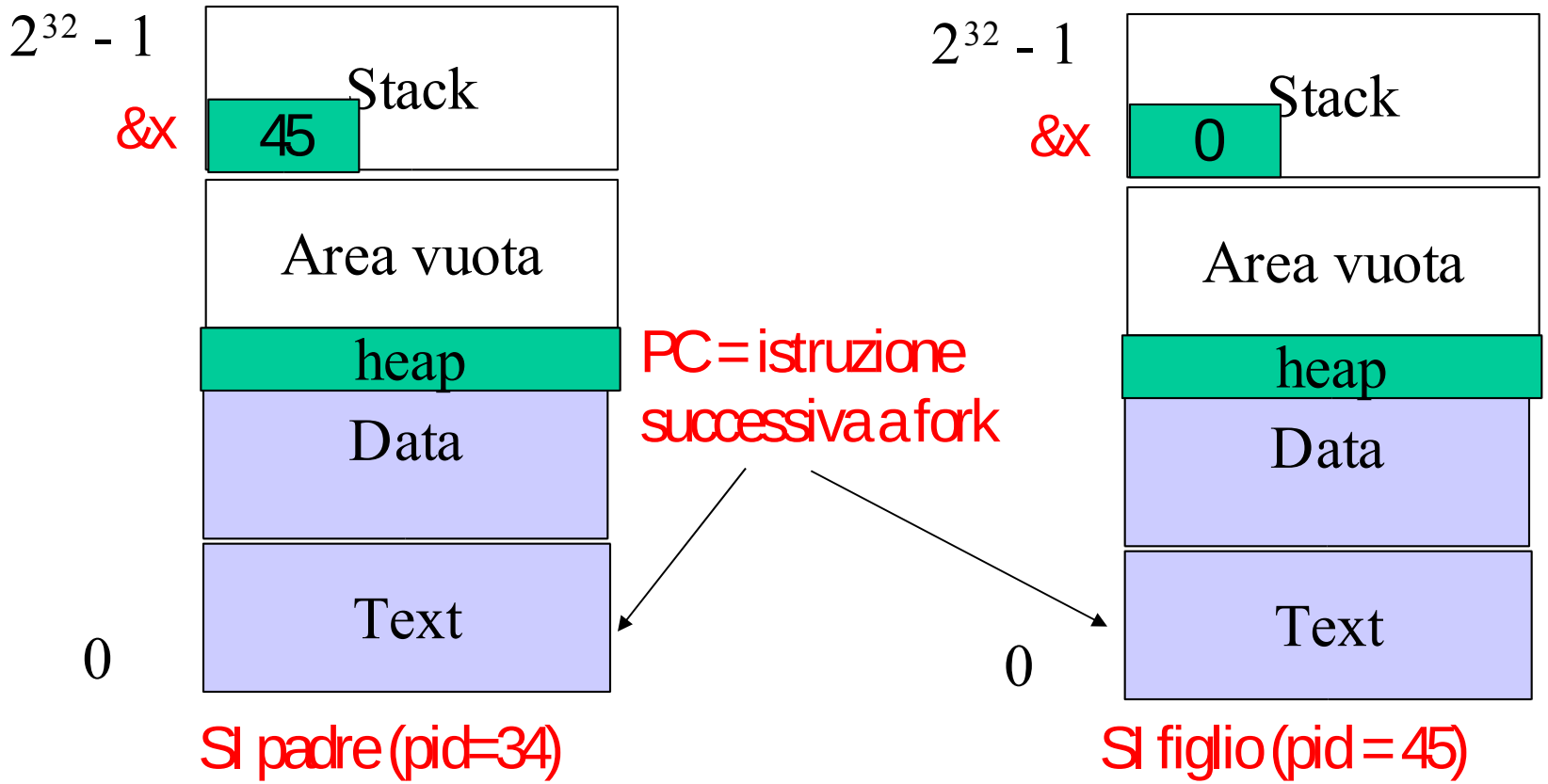
Creazione di processi (2)

- Spazio di indirizzamento di padre e figlio dopo una **`fork`** terminata con successo



Creazione di processi (3)

- Come prosegue l'esecuzione nei processi padre e figlio



Creazione di processi (4)

```
/* un piccolo esempio: forktest */
  int main (void) {
     int pid;
     printf(”Inizio\n”);
     pid = fork();
     printf (”%d: Ho ricevuto: %d\n”, getpid(), /
   pid);
     return 0;
  }
```

Creazione di processi (5)

```
/* esecuzione */
  bash:~$ forktest
  Inizio
  45643: Ho ricevuto 0     -- stampato dal figlio
  45642: Ho ricevuto 45643  -- stampato dal padre
/* non si può assumere nessun ordine di
  esecuzione! */
  bash:~$ forktest
  Inizio
  45653: Ho ricevuto 45654  -- stampato dal padre
  45654: Ho ricevuto 0     -- stampato dal figlio
  bash:~$
```

Creazione di processi (6)

```
/* esecuzione */
  bash:~$ forktest > out
  bash:~$ more out
  Inizio
  45653: Ho ricevuto 45654   -- stampato dal padre
  Inizio
  45654: Ho ricevuto 0       -- stampato dal figlio
  bash:~$
```

Perché due Inizio ??????????

fork() : alcuni commenti

- È costosa!
  - Il segmento TEXT non può essere modificato e quindi può essere condiviso
  - DATI, HEAP, STACK devono essere duplicati
- praticamente sempre implem. con *copy-on-write*
  - è necessaria memoria virtuale paginata
  - nel figlio si copia solo la tabella delle pagine
    - le pagine sono marcate in sola lettura
    - se il figlio tenta la scrittura sono duplicate al volo dal kernel
  - efficace perché spesso il figlio butta via tutto lo spazio di indirizzamento specializzandosi con un altro eseguibile (vedi **`exec`**)

fork() : alcuni commenti (2)

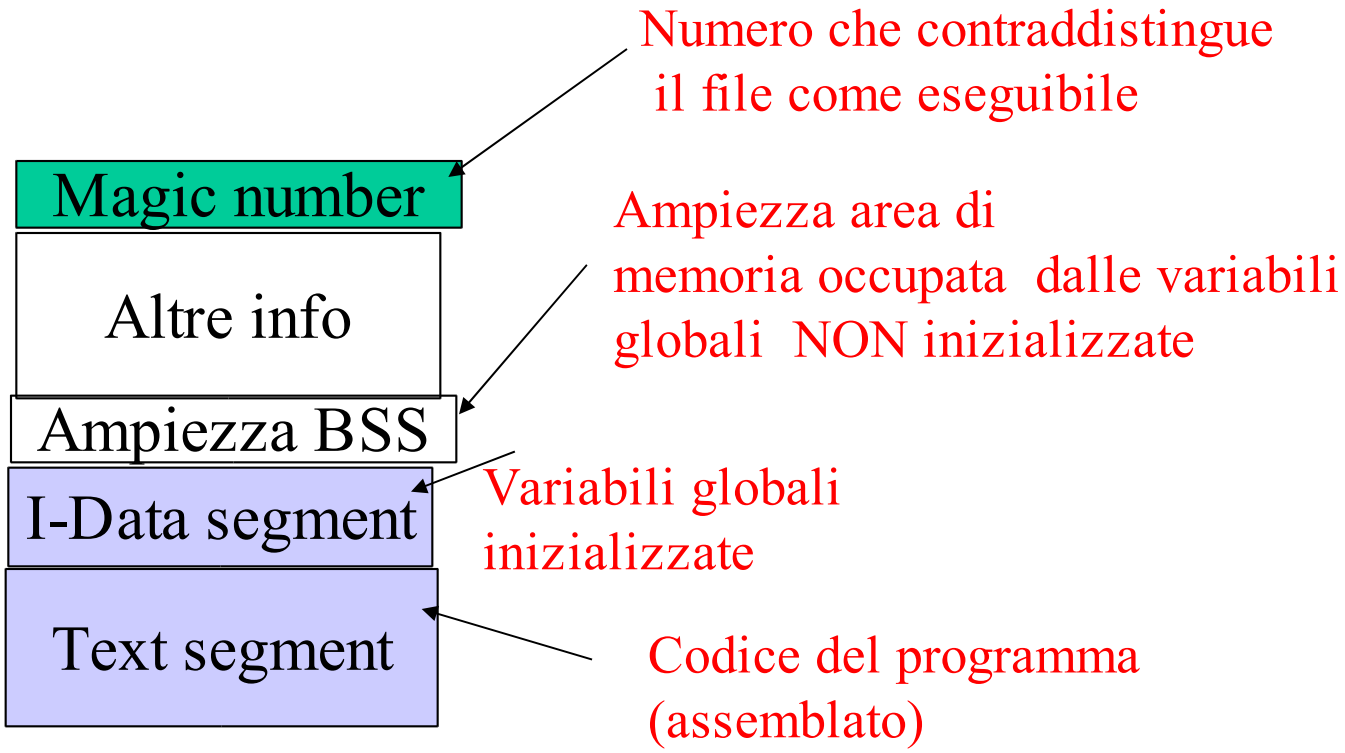
- **`vfork()`**
  - fa condividere la memoria a padre e figlio!
  - Era popolare prima della *copy-on-write*
  - Pericolosa, anche se più veloce di *copy-on-write*
  - da NON usare mai, ci possono essere race condition o effetti collaterali imprevisti

Differenziazione : le `exec*()`

- Avere due processi uguali non sembra una buona idea!
  - Infatti **`fork()`** viene quasi sempre combinata con una chiamata della famiglia **`exec()`** per reinizializzare lo spazio di indirizzamento del processo corrente in base a un *eseguibile*
- tutte le **`exec*()`**
  - sono 6 funzioni di libreria con differenze sul tipo di parametri
  - alla fine invocano la **`execve`**

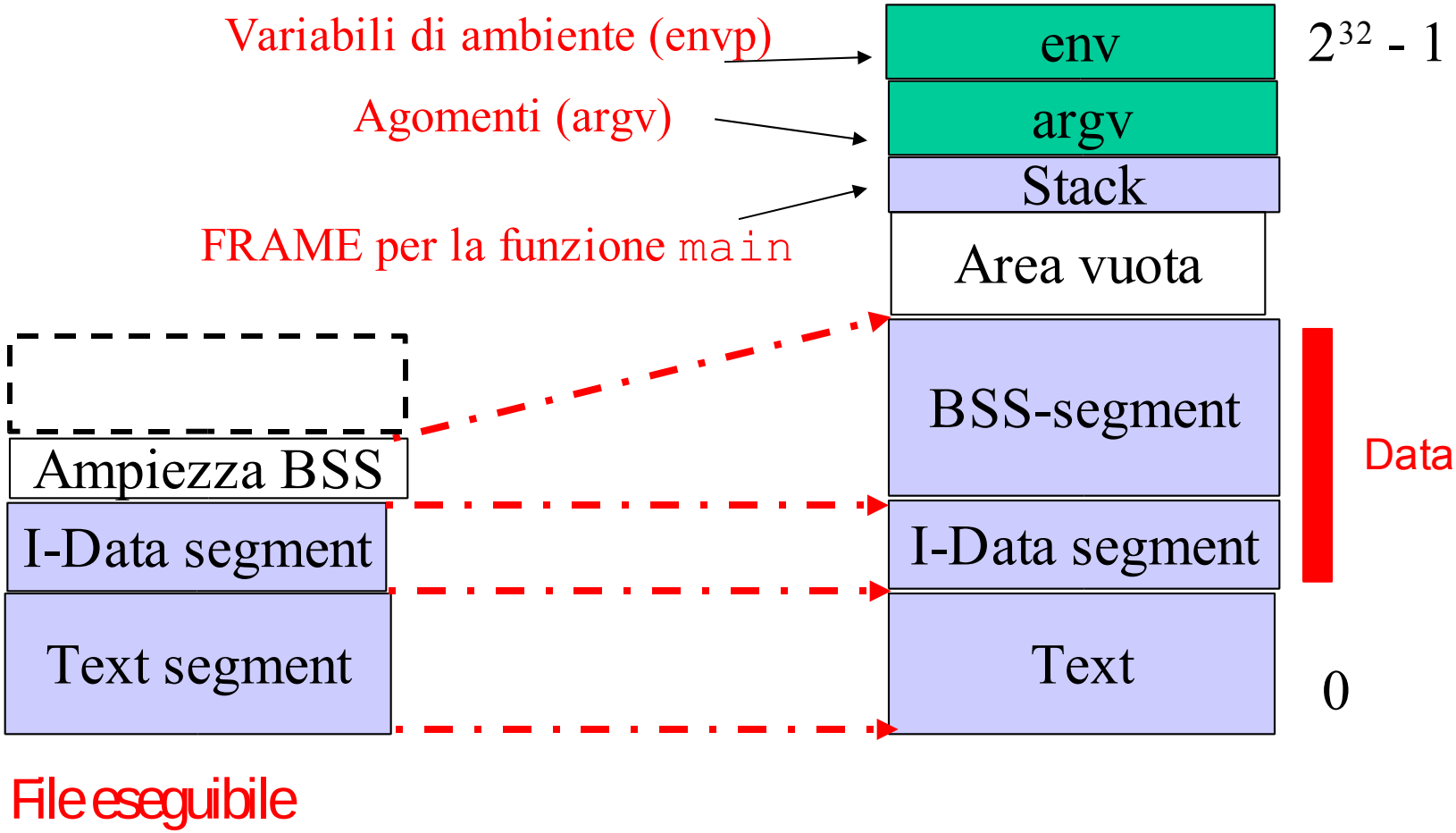
Differenziazione : le `exec*()` (2)

- Formato di un file eseguibile
  - risultato di compilazione, linking etc …



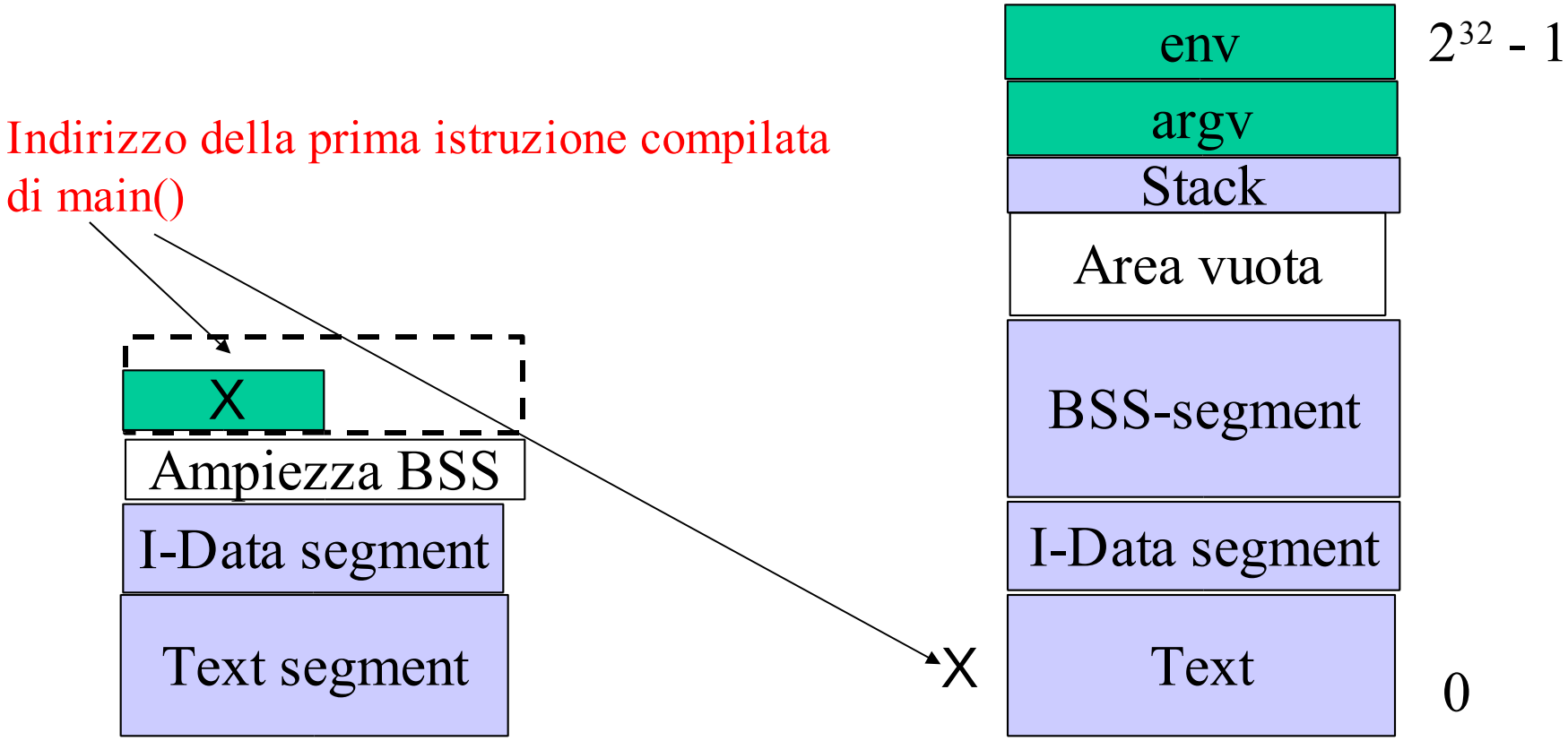
Differenziazione : le `exec*()` (3)

- (1) il contenuto del file eseguibile viene usato per sovrascrivere lo spazio di indirizzamento del processo che la invoca



Differenziazione : le `exec*()` (4)

- (2) si carica in PC l'indirizzo iniziale X
- la DIFFERENZIAZIONE è terminata!



Differenziazione : `execl()`

```
#include <unistd.h>

int execl(
   char* path,     /* eseguibile */
  char* arg0,     /* primo arg (nome file) */
  char* arg1,     /* secondo arg (se c'è) */
  …, /* altri arg (se ci sono) */
  (char *) NULL /* termina la lista */
)
/* [on success] DOES NOT RETURN
    [on error] returns -1, sets errno */
```

Differenziazione : `execl()` (2)

- **`execl()`**
  - il path specificato deve essere un eseguibile, e deve avere i permessi di esecuzione per l'effective-user-ID del processo che esegui la **`execl()`**
  - una **`execl()`** che ha successo non può ritornare perché il contenuto del vecchio indirizzo di ritorno è stato sovrascritto nella reinizializzazione dello spazio di indirizzamento
  - non è necessario testare il valore ritornato da exec() perché se ritorna c'è stato sicuramente errore!
  - Gli argomenti arg0…argN, saranno accessibili dal **`main ()`** del programma appena attivato

Differenziazione : `execl()` (3)

- **`execl()`**
  - l'ambiente (**`environ`**) viene preservato (vedi poi)
  - il processo rimane lo stesso, quindi quasi tutti gli attributi rimangono gli stessi
    - es. pid, ppid,session, controlling terminal, real user and groupID, cwd, open file descriptors
  - gli attributi possono cambiare sono:
    - gestione dei segnali (viene resettata a default)
    - *effective-userID* e/o *effective-groupID* (se l'eseguibile ha settato i bit set-user-ID e/o set-group-ID)
    - funzioni registrate at_exit(), segmenti di memoria condivisa (unmapped), semafori POSIX (resettati)

Differenziazione : `execl()` (4)

```
/* un piccolo esempio: exectest */
  int main (void) {

    printf("The quick brown fox jumped over");
    execl("/bin/echo", "echo", "the", "lazy",
          "dogs", (char*)NULL);
/* se execl ritorna si è verificato un errore */
    perror("execl");
    return 1;
 }
```

Differenziazione : `execl()`(5)

```
/* esecuzione */
  bash:~$ exectest
  the lazy dogs        -- what about fox???
  bash:~$
```

Differenziazione : `execl()` (5.1)

```
/* esecuzione */
  bash:~$ exectest
  the lazy dogs      -- what about fox???
  bash:~$
  -- stdout not flushed as we did not exit!!!!!
```

Differenziazione : `execl()` (6)

```
/* un piccolo esempio: exectest */
  int main (void) {

    printf("The quick brown fox jumped over ");
    fflush(stdout);
    execl("/bin/echo", "echo", "the", "lazy",
          "dogs", (char*)NULL);
/* se execl ritorna si è verificato un errore */
    perror("execl");
    return 1;
 }
```

Differenziazione : `execl()`(7)

```
/* esecuzione */
  bash:~$ exectest
  The quick brown fox jumped over the lazy dogs
  bash:~$
```

Differenziazione : le `exec*()`

- execl, execlp, execle, execv, execvp, execve
  - è possibile richiedere che la exec() cerchi il file nelle directory specificate dalla variabile di ambiente PATH (**p** nel nome)
    - prova anche ad eseguire il file come script chiamando una shell come interprete **(es #!/bin/bash**)
  - è possibile passare un array di argomenti secondo il formato di **argv[]** (**v** nel nome)
  - è possibile passare un array di stringhe che descrivono l'environment (**e** nel nome)
  - è possibile passare gli argomenti o l'environment come lista (terminato da **NULL**) (**l** nel nome)

Esempio : una *shell* semplificata

```
int main (void) {
  char * argv [MAXARG];
  int argc      , fine=FALSE;
  inizializza();
  while (TRUE) {              /*ciclo infinito*/
    type_prompt();             /* stampa prompt*/
    if (read_cmd_line(&argc,argv,MAXARG)!= -1) {
       execute(argc,argv);
    else
       fprintf(stderr,"invalid command!\n");
    if (fine) exit(EXIT_SUCCESS);
  } /*end while */
  return 0; } /*end main */
```

Esempio : una *shell* semplificata (2)

```
int main (void) {
  char * argv [MAXARG];
  int argc      , fine=FALSE;
  inizializza();
  while (TRUE) {              /*ciclo infinito*/
    type_prompt();             /* stampa prompt*/
    if (read_cmd_line(&argc,argv,MAXARG)!= -1) {
       execute(argc,argv);
    else
       fprintf(stderr,"invalid command!\n");
    if (fine) exit(EXIT_SUCCESS);
  } /*end while */
  return 0; } /*end main */
```

Esempio : una *shell* semplificata (3)

```
static void execute (int argc, char* argv []) {
  pid_t pid;
  switch( pid = fork() ) {
  case -1: /* padre errore */ {
    perror(”Cannot fork”);
    break; }
  case 0: /* figlio */ {
    execvp (argv[0],argv);
    perror(”Cannot exec”);
    break; }
  default: /* padre */ {
   /*attende il figlio o ritorna*/ }
}
```

Terminazione

- Un processo può terminare solo in 4 modi:
  - Chiamando **`exit()`**
  - Chiamando **`_exit()`** (Unix) o **`_Exit()`** (standard C)
  - Ricevendo un segnale
  - per System crash: staccare la spina, bug nel SO etc

- Tratteremo i primi due (!) e il terzo più avanti, quando si parla dei segnali

Terminazione : _exit() _Exit()

```
#include <unistd.h>
void _exit(
   int status,     /* exit status */
)
/* DOES NOT RETURN */

#include <stdlib.h>
void _Exit(
   int status,     /* exit status */
)
/* DOES NOT RETURN */
```

Terminazione : `exit()`

```
#include <stdlib.h>


void exit(
   int status,     /* exit status */
)
/* DOES NOT RETURN */
```

- fa tutto quello cha fa la **`_exit()`** più
  - chiama la funzione/i registrata/i con **`atexit()`** (se c'è)
  - esegue il fflush dei buffer di I/O (con fflush() o fclose())

Intermezzo : `atexit()`

```
#include <stdlib.h>
int atexit(
   void *function (void),  /* exit status */
)
/* (0) success (!=0) fallimento (NON setta
   errno) */
```

- registra la funzione function in modo che sia chiamata quando il programma termina con **`exit()`** (o **`return`** dal main)
  - tipicamente usata per codice di pulizia (cancellare file temporanei, pipe,stampare messaggi sullàesito della computazione etc)

Intermezzo : `atexit()` (2)

- Esempio:

```
#include <stdlib.h>
static void cleanup (void) {
    Unlink(tempfile);
     fprintf(stderr,"closing ...");
}
int main (void) {
   …
   if (!atexit(cleanup)) { /* gest err */ }
   /* resto del codice */
}
```

Intermezzo : `atexit()` (3)

- Si possono registrare più funzioni
  - verranno chiamate in ordine inverso

Terminazione : `_exit()`

- **`_exit(status);`**
  - termina il processo
  - chiude tutti i file descriptor
  - libera lo spazio di indirizzamento,
  - invia un segnale **`SIGCHLD`** al padre
  - salva il byte meno significativo (0-255) di **`status`** nella tabella dei processi in attesa che il padre lo accetti (con la **`wait()`, `waitpid()`**)
  - i figli, diventati 'orfani' vengono adottati da **`init`** (cioè **`ppid`** viene settato a **`1`**)
  - se eseguita nel main è equivalente ad una **`return`**

Attesa del figlio : `waitpid()`

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

int waitpid(
   pid_t pid,     /* pid or proces group id */
   int* statusp,     /* punt status (o NULL) */
   int options,     /* opzioni (vedi sotto) */
)
/* [on success] PID o 0
    [on error] returns -1, sets errno */
```

Attesa del figlio: `waitpid()` (2)

- **`waitpid(pid,&status,options)`**
  - attende che un figlio cambi di stato (terminato, sospeso/stopped, riattivato/continued)
  - *si possono attendere solo i figli direttamente attivati con fork()*
  - **`pid>0`** : attende il figlio di PID '**`pid`**'
  - **`pid=-1`** : attende un qualsiasi figlio (ritorna il **`pid`** di quello che ha cambiato stato)
  - **`pid=0`** : attende un qualsiasi processo figlio nello stesso *process group*
  - **`pid<-1`** : attende un qualsiasi processo figlio nel gruppo **`-pid`**

Attesa del figlio: `waitpid()` (3)

- **`waitpid(pid,&status,options)` `(cont)`**
  - **`status`**, prende il codice di ritorno (1 byte, specificato nella `_exit()` o nella `exit()`) più un insieme di altre informazioni recuperabili tramite maschere. Es:

    --true se terminato con *exit()

    `WIFEXITED(status)`

    --if WIFEXITED lo stato si recupera con

    `WEXITSTATUS(status)`

    --true se terminato con segnale

    `WIFSIGNALED(status)`

    --if WIFSIGNALED il segnale si recupera con

    `WTERMSIG(status)`

Attesa del figlio: `waitpid()` (4)

- **`waitpid(pid,&status,options)` (cont)**
  - se almeno un figlio è già terminato, e il suo stato non è stato ancora letto con una **`wait *()`, `waitpid()`** termina subito altrimenti si blocca in attesa
  - options: uno o più flag combinati con OR (**`|`**) es:
    - **`WNOHANG`**: se lo stato non è disponibile non si blocca ma ritorna subito 0

Attesa del figlio: `waitpid()` (5)

- **Esempi:**

```
/*aspetta il figlio di PID pid */
if ( ( waitpid(pid,&status,0) ) == -1)
   { /* gestione errore */ }

/*aspetta la terminazione di un qualsiasi
   figlio, senza memorizzare lo stato*/
if ( ( pid=waitpid(-1,NULL,0) ) == -1)
   { /* gestione errore */ }

/* come sopra senza mettersi in attesa */
if ( ( pid=waitpid(-1,NULL,WNOHANG) ) == 0)
   { /* stato non disponibile */ }
```

Esempio : wait() ed exit()

```
int status ; /* conterra' lo stato */
if ( ( pid = fork() ) == -1) {
   perror("main: fork"); exit(errno); }
if ( pid ) {  /* padre */
  sleep(20);
  pid = waitpid(pid,&status,0);
  if (WIFEXITED(status)) {
  /* il figlio terminato con exit o return */
  printf("stato %d\n", WEXITSTATUS(status)); }
else { /* figlio */
     printf("Processo %d, figlio.\n",getpid());
          exit(17); /* termina con stato 17
*/ }
```

Esempio : wait() ed exit() (2)

- cosa accade se eseguiamo un main contenente il codice dell'esempio :

```
bash:~$ a.out & -- avvio l'esecuzione in bg
Processo 1246, figlio. -- stampato dal figlio
```

Esempio : wait() ed exit() (3)

prima che i 20 secoondi siano trascorsi …

```
bash:~$ a.out  &
Processo 1246, figlio.
bash:~$ ps -l
  …    S  UID  PID   PPID  ……    CMD
  …    Z  501  1246  1245  ……    a.out
```

-- il figlio e' un processo zombie (Z)

-- è terminato ma nessuno ha fatto la

-- waitpit

-- occupa ancora la tabella dei processi

Esempio : wait() ed exit() (4)

prima che i 20 secoondi siano trascorsi …

```
bash:~$ a.out  &
Processo 1246, figlio.
bash:~$ ps -l
  …    S  UID  PID   PPID  …….    CMD
  …    Z  501  1246  1245  ……..   a.out
bash:~$
Stato 17.          -- stampato dal padre
bash:~$
```

Ancora attesa : `wait()`

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

int wait(
  int* statusp,     /* punt status (o NULL) */
)
/* [on success] PID
   [on error] returns -1, sets errno */
```

- corrisponde a **`waitpid`** con **`pid=-1`** e nessuna opzione
- quindi aspetta un qualsiasi figlio, bloccandosi se necessario

Ancora attesa : `wait()` (2)

- In realtà un figlio non può ritornare il suo stato più di una volta, quindi usare la wait() o **`waitpid`** con **`pid=-1`** può generare confusione e malfunzionamenti in programmi complessi
  - ottengo lo stato del figlio sbagliato!
  - Lo stato non è più disponibile in altre parti del programma, che ne avrebbero bisogno!
- Inoltre ci sono casi in cui non è banale capire quale figlio aspettare prima (senza andare in deadlock!)
- Nuovi standard introducono **`waitid()`**, che permette di attendere ed ottenere lo stato senza distruggerlo
  - non è ancora disponibile in Linux, non ne parleremo

Esempio : una *shell* semplificata (4)

```
static void execute2 (int argc, char* argv []) {
  pid_t pid;
  int status;
  switch( pid = fork() ) {
  case -1: /* padre errore */ {……}
  case 0: /* figlio */ { execvp (argv[0],argv);
    perror(”Cannot exec”); break; }
  default: /* padre */ {
  if ( waitpid(pid,&status,0) = -1 ) {
   perror(”waitpid:”); exit(errno)}
    print_status(pid,status);
 } /* end switch */
}
```

Esempio : una *shell* semplificata (5)

```
static void print_status (pid_t pid, int status){
{
  if ( pid != 0) printf("Process %d",(int)pid);
  if (WIFEXITED(status)) /* term normale */
   printf("Exit value: %d",WEXITSTATUS(status));
  if (WIFSIGNALED(status)) /* segnale */
    printf("Killed signal: %d",WTERMSIG(status));
  if (WCOREDUMP(status)) /* core file */
    printf("-- core dumped");
  …
  if (WIFSTOPPED(status))  printf("stopped");
  printf("\n");
}
```

User and group IDS

```
#include <unistd.h>

uid_t getuid(void);
/* [real UID] successo (no error return) */
uid_t geteuid(void);
/* [effective UID] succ (no error return) */
uid_t getgid(void);
/* [real GID] successo (no error return) */
uid_t getegid(void);
/* [effective GID] succ (no error return) */
```

Esercizio: myshell (5)

Estendere la shell con l'algoritmo per la verifica dei diritti del comando da eseguire

Attenzione : processi *zombie*

- Quando un processo fa la exit rilascia tutte le risorse eccetto la tabella dei processi, in cui viene mantenuto l'exit status finchè qualcuno non lo recupera con una wait()
- Quando nessuno ha ancora fatto la wait in processo non è *terminato* ma *zombified*
- se il padre muore, il processo zombie viene automaticamente adottato da init, che esegue la wait
- se il padre è vivo, ma non esegue la wait il processo rimane in stato di zombie, occupando la tabella dei processi, e la sua presenza inutile può far fallire fork successive!

Stati dei processi in UNIX