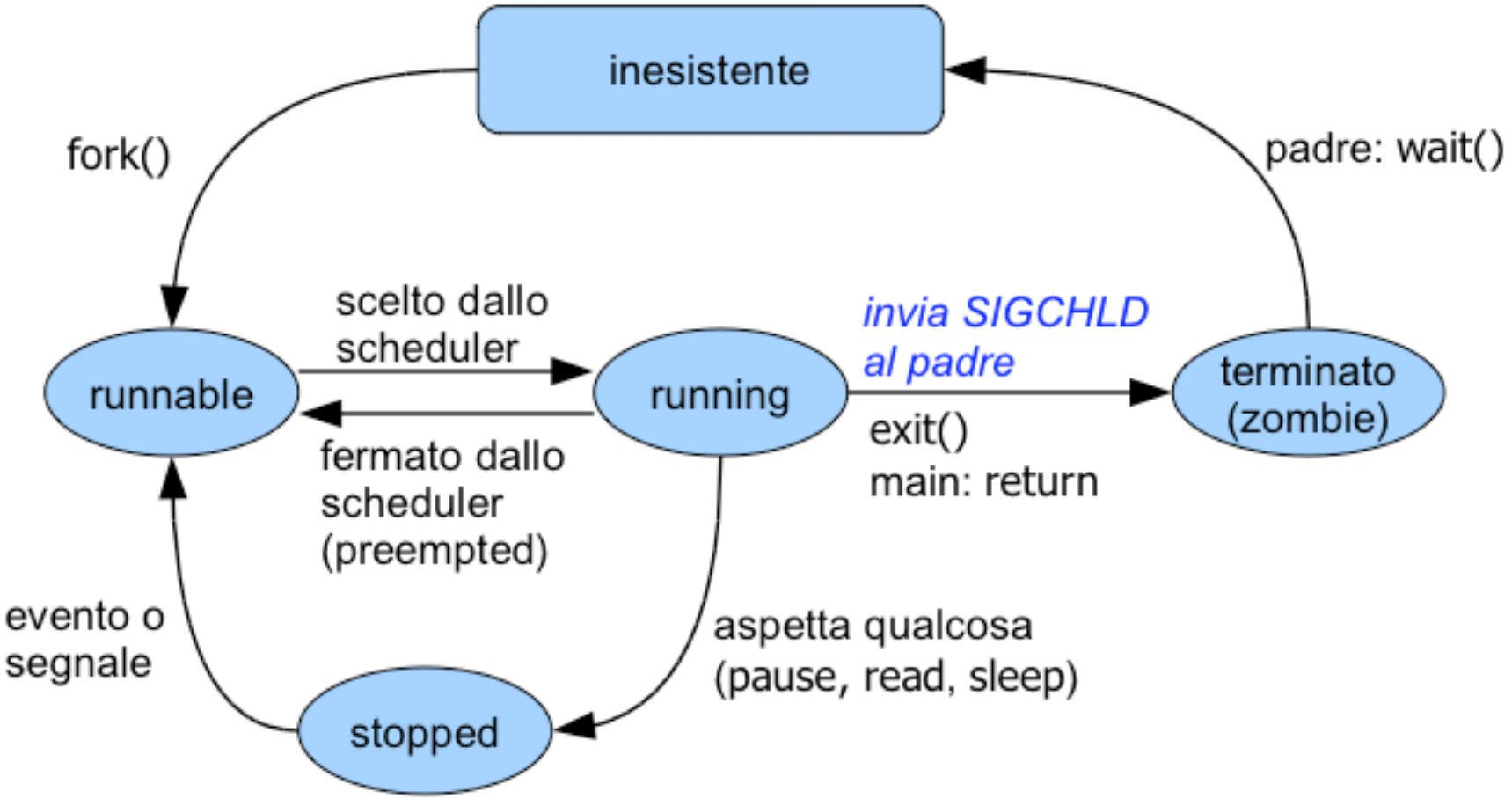
Processi: Exit, Wait, Exec

Contiene lucidi tratti da: 2005-07 Giuseppe Schmid (Univ. Federico II), 2005-2007 Francesco Pedullà, Massimo Verola (Uniroma2), 2001-2005 Renzo Davoli (Università di Bologna), Alberto Montresor (Università di Bologna)

Ciclo di vita del processo



Terminazione di processi (I)

- Esistono tre modi per terminare in modo NORMALE:
  - eseguire un return da main (è equivalente a chiamare **`exit`**)
  - chiamare la funzione **`exit:`**
    - **`void exit(int status);`**
      - invoca di tutti gli exit handlers che sono stati registrati
      - chiude di tutti gli I/O stream standard
      - è specificata in ANSI C
  - chiamare la system call **`_exit:`**
    - **`void _exit(int status);`**
      - ritorna al kernel immediatamente
      - è chiamata come ultima operazione da **`exit`**
      - **è specificata nello standard POSIX.1**

Terminazione di processi (I)

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

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

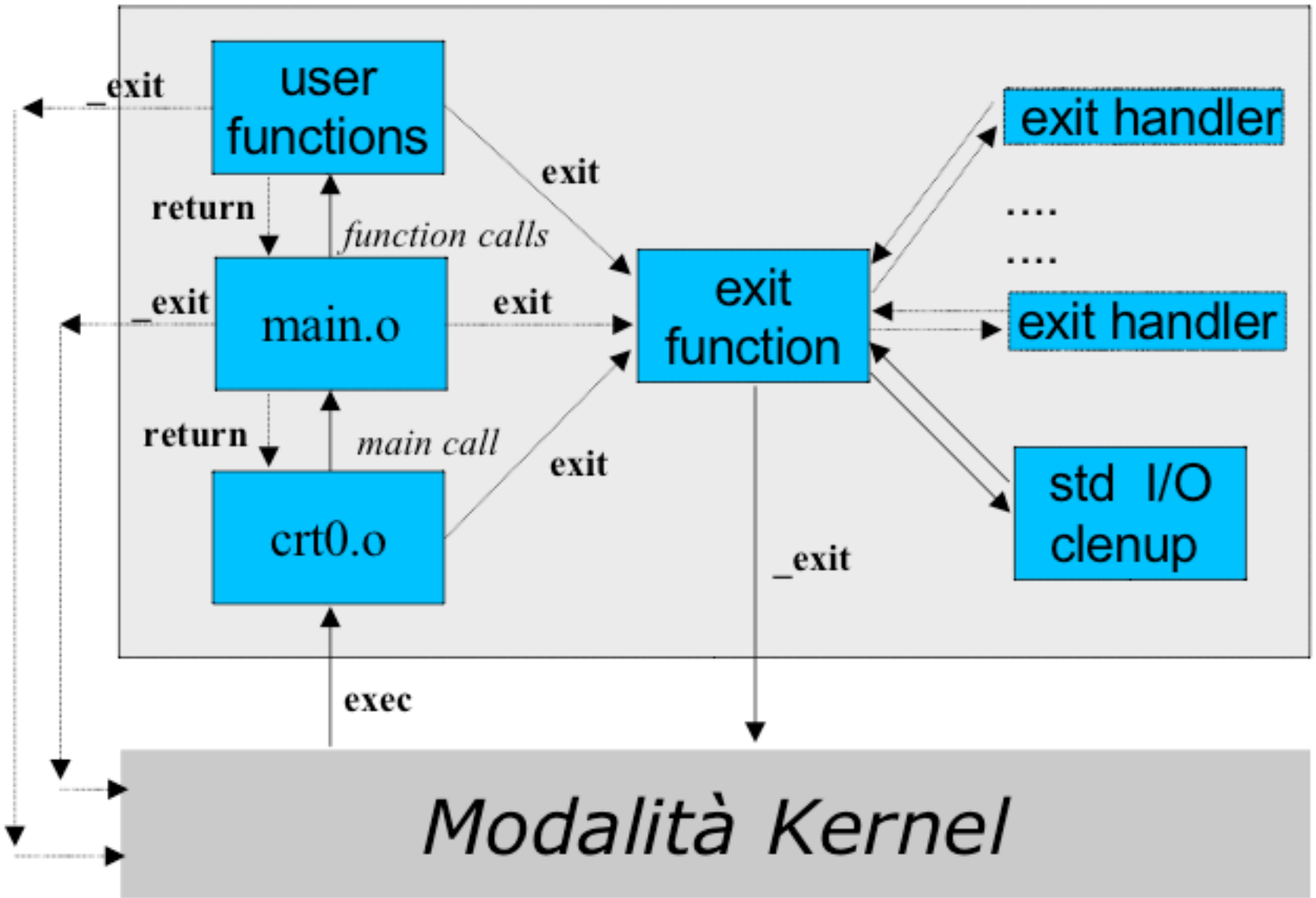
_exit restituisce il controllo al kernel immediatamente,

exit effettua le seguenti operazioni:

- chiama delle funzioni (opzionali), dette exit handler;
- chiude tutti gli stream di I/O chiamando **fclose**, che svuota i buffer di output;
- chiama **_exit**

Le due funzioni hanno per argomento un intero, lo stato di uscita ***status***, che viene passato al processo genitore. Entrambe non ritornano alcun valore.

Terminazione di processi (II)



Terminazione di processi (III)

- Esistono due modi per terminare in modo ANORMALE:
  - Quando un processo riceve certi segnali
    - generati dal processo stesso
    - generati da altri processi
    - generati dal kernel
  - Chiamando `abort`:
    - `void abort();`
    - La chiamata ad `abort` costituisce un caso speciale del primo caso dei tre sopra elencati, in quanto genera il segnale `SIGABRT`

NOTA: per informazioni sui segnali usa `man 7 signal`

Azioni del kernel a terminazione di processo

- Sia nel caso di terminazione normale che in quella anormale le azioni attuate dal kernel sono le stesse:
  - ***rimozione della memoria utilizzata dal processo***
  - ***chiusura dei descrittori aperti***

Valori di ritorno a terminazione del processo

- Exit status:
  - Valore che viene passato ad **`exit`** (e **`_exit)`** e che notifica il padre su come è terminato il processo (successo, con errore)
- Termination status:
  - Valore che viene generato dal kernel nel caso di una terminazione normale/anormale
  - Nel caso si parli di terminazione anormale, si specifica la ragione per questa terminazione anormale
  - L'*exit status* e' convertito dal kernel in *termination status* quando alla fine viene chiamata **`_exit`**
- Come ottenere questi valori?
  - Tramite le funzioni **`wait`** e **`waitpid`** (descritte in seguito)

Processi *zombie*

- ***Cosa succede se il padre termina prima del figlio?***
  - il processo figlio viene "adottato" dal processo `init` (PID=1), in quanto il kernel vuole evitare che un processo divenga "orfano" (cioè senza un PPID)
  - quando un processo termina, il kernel esamina la tabella dei processi per vedere se aveva figli; in tal caso, il PPID di ogni figlio viene posto uguale a 1
- ***Cosa succede se il figlio termina prima del padre?***
  - generalmente il padre aspetta mediante la funzione `wait` che il figlio finisca ed ottiene le varie informazioni sull'exit status
  - se il figlio termina senza che il padre lo “aspetti”, il padre non avrebbe più modo di ottenere informazioni sull'exit status del figlio
  - per questo motivo, alcune informazioni sul figlio vengono mantenute in memoria e il processo diventa uno zombie

Lo stato di *zombie*

- **Quando un processo entra nello stato di zombie, il kernel mantiene le informazioni che potrebbero essere richieste dal processo padre tramite `wait` e `waitpid`**
  - process ID
  - termination status
  - accounting information (tempo impiegato dal processo)
- **Il processo resterà uno zombie fino a quando il padre non eseguirà una delle system call `wait` o `waitpid`**
- **Per quanto riguarda i figli del processo `init`:**
  - non possono diventare zombie
  - tutte le volte che un figlio di **`init`** termina, **`init`** esegue una chiamata **`wait`** e raccoglie eventuali informazioni: questo è il modo in cui gli zombie vengono eliminati dal sistema

System call `wait` e `waitpid`

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- La `wait` e `waitpid` sono utilizzate per ottenere informazioni sulla terminazione dei processi figli
- Quando un processo chiama `wait` o `waitpid`:
  - può bloccarsi, se tutti i suoi figli sono ancora in esecuzione
  - può ritornare immediatamente con il termination status di un figlio, se un figlio ha terminato ed il suo termination status è in attesa di essere raccolto
  - può ritornare immediatamente con un errore, se il processo non ha alcun figlio
- Nota:
  - se eseguiamo una system call `wait` quando abbiamo già ricevuto `SIGCHLD`, essa termina immediatamente, altrimenti si blocca

System call **`wait`** e **`waitpid`**

- Significato degli argomenti:
  - **`status`** è un puntatore ad un intero; se diverso da **`NULL`**, il termination status viene messo in questa locazione
  - Il valore di ritorno è il process id del figlio che ha terminato
- Differenza tra `wait` e `waitpid`:
  - **`wait`** blocca il chiamante fino a quando un qualsiasi figlio non sia terminato
  - **`waitpid`** ha delle opzioni per evitare di bloccarsi
  - **`waitpid`** può mettersi in attesa di uno specifico processo
- Il contenuto del termination status dipende dall'implementazione:
  - bit per la terminazione normale, bit per l'exit status, etc.

Status wait waitpid

Se *statloc* non è NULL, l'intero cui esso punta rappresenta lo stato di terminazione del processo atteso, secondo una codifica che dipende dall'implementazione.
POSIX prevede che tale stato possa essere rilevato grazie ad opportune macro definite in <sys/wait.h>:

Macro	Descrizione
WIFEXITED(*status*)	vera se il figlio è terminato normalmente; in tal caso WEXITSTATUS(*status*) restituisce lo stato di uscita
WIFSIGNALED(*status*)	vera se il figlio è terminato a causa di un segnale; in tal caso WTERMSIG(*status*) ne restituisce il numer
WIFSTOPPED(*status*)	vera se il figlio è in stato di stop; allora il numero del segnale di stop è dato da WSTOPSIG(status)
WIFCONTINUED(*status*)	vera se il figlio ha ripreso l'elaborazione dopo uno stato di stop

System call **waitpid**

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Argomento **pid**:
  - **pid == -1** si comporta come **wait**
  - **pid > 0** attende la terminazione del figlio con process id uguale a **pid**
  - **pid == 0** attende la terminazione di qualsiasi figlio con process group ID uguale a quello del chiamante
  - **pid < -1** attende la terminazione di qualsiasi figlio con process group ID uguale a **–pid**
- Opzioni (parametro **options**):

  WNOHANG non si blocca se il child non ha terminato

Opzioni waitpid

waitpid dispone dell'argomento di tipo intero *options*, il cui valore può essere zero o una combinazione OR bit a bit (operatore |) con delle costanti, dipendenti dall'implementazione:

- se *options* = 0, **waitpid** attende per la terminazione di (almeno) un figlio specificato da pid;
- le costanti supportate da POSIX sono riassunte nella seguente tabella

Macro	Descrizione
WCONTINUED	ritorna lo stato di ogni figlio specificato da *pid* che ha ripreso l'elaborazione dopo uno stop
WNOHANG	waitipid ritorna immediatamente (con valore 0) se *pi* non individua alcun figlio che è terminato
WUNTRACED	ritorna lo stato di ogni figlio specificato da *pid* che è in stop ed il cui attuale stato non è stato riportato

```c
/* print_exit.c: visualizza la descrizione dello stato di un processo */

# include <sys/wait.h>      /* per le macro (eccetto WCOREDUMP) */
#include <stdio.h>          /* per la funz. printf */

void print_exit(int status)
{
   if (WIFEXITED(status))
         printf("normal termination, exit status = %d\n",
                  WEXITSTATUS(status));
   else if (WIFSIGNALED(status))
         printf("abnormal termination, signal number = %d%s\n",
                  WTERMSIG(status),
#ifdef   WCOREDUMP /* se e' definita, compila la linea seguente */
               WCOREDUMP(status) ? " (core file generated)" : "" );
#else                       /* altrimenti compila la linea seguente */
               "" );
#endif
   else if (WIFSTOPPED(status))
         printf("child stopped, signal number = %d\n",
                  WSTOPSIG(status));
}
```

```
/* status.c: Fornisce un esempio dei diversi stati di uscita
             di un processo figlio                                  */

#include<sys/types.h>  /* per il tipo pid_t  */
#include <unistd.h>    /* per la funzione fork  */
#include<sys/wait.h>   /* per la funzione waitpid */
#include <stdio.h>     /* per le funz. printf, fgets e perror */
#include <stdlib.h>    /* per la funzione exit */
#include <string.h>    /* per la funzione strlen */
#include <errno.h>     /* per la messaggistica di errore */

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

      if ( (pid = fork()) < 0)
          perror("fork"), exit(1);
      else if (pid == 0)     /* primo figlio...*/
          exit(7);           /* che termina in modo normale */

      if (wait(&status) != pid)       /* il genitore aspetta il figlio... */
          perror("wait"), exit(1);
      print_exit(status);             /* e ne mostra lo stato di terminazione*/
```

```
    if ( (pid = fork()) < 0)
        perror("fork"), exit(1);
    else if (pid == 0)          /* secondo figlio... */
        abort();                /* che genera SIGABRT */

    if (wait(&status) != pid)   /* il genitore aspetta il 2ndo figlio... */
      perror("fork"), exit(1);
    print_exit(status);         /* e ne mostra lo stato di terminazione */

    if ( (pid = fork()) < 0)
      perror("wait"), exit(1);
    else if (pid == 0)          /* terzo figlio.... */
          status /= 0;          /* che divide per 0 e genera SIGFPE */

    if (wait(&status) != pid)     /* il genitore aspetta il  terzo figlio... */
      perror("wait"), exit(1);
    print_exit(status);           /* e ne mostra lo stato di terminazione */

    exit(0);
}
```

Eseguendo l'esempio, si ottiene:

```
gio$ ./status
normal termination, exit status = 0
abnormal termination, signal num. = 6(core file generated)
abnormal termination, signal num. = 8(core file generated)
gio$
```

Nota: Come vedremo, i numeri di segnale corrispondono in genere a segnali diversi a seconda del sistema utilizzato. Per ottenere un nome descrittivo del segnale a partire dal numero del segnale, è necessario ricorrere a funzioni opportune.

Notifica della terminazione di un figlio

- **Quando un processo termina (normalmente o no), il padre viene informato dalla ricezione di un segnale `SIGCHLD`**
- **La notifica è asincrona**
- **Il padre ha la possibilità di:**
    - ignorare il segnale (default)
    - predisporre una funzione speciale, detta *signal handler*, che viene invocata automaticamente quando il segnale viene ricevuto

Esecuzione di Programmi

L'esecuzione di un nuovo programma in UNIX si ottiene con la chiamata ad una delle funzioni exec;

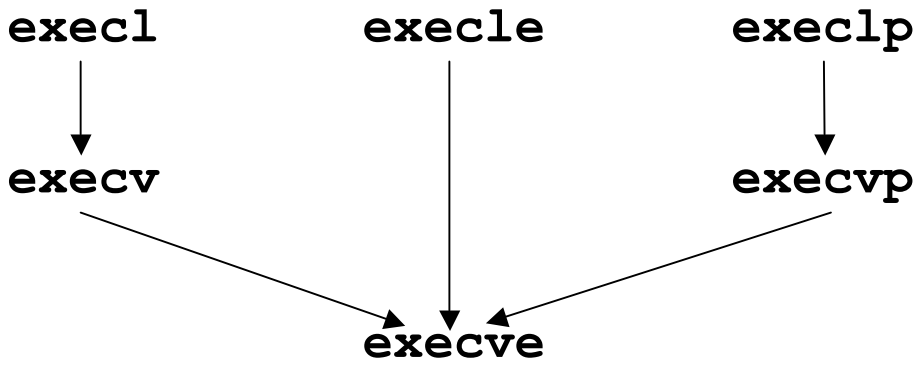
Le funzioni exec *non* generano un nuovo processo, ma modificano il layout in memoria del processo chiamante per l'esecuzione del nuovo programma;

Quando un processo chiama una delle funzioni exec, le aree di memoria text, data, heap e stack relative al processo corrente vengono sostituite in base al nuovo programma;

Il nuovo programma incomincia la propria esecuzione a partire dalla funzione main.

La famiglia di system call **exec**

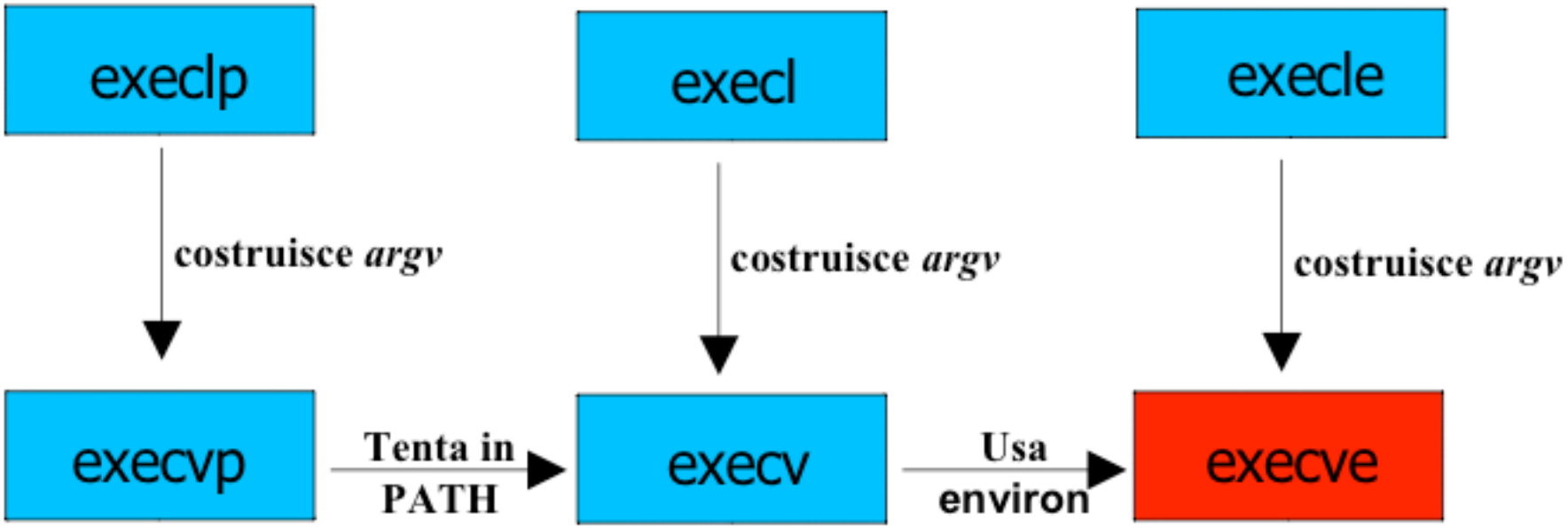
- Se fork fosse l'unica primitiva per creare nuovi processi, la programmazione in ambiente Unix sarebbe ostica, dato che si potrebbero creare soltanto copie dello stesso processo.
- La famiglia di primitive `exec` può essere utilizzata per superare tale limite in quanto le varie system call `exec` permettono di iniziare l'esecuzione di un altro programma sovrascrivendo la memoria del processo chiamante.



- In realtà tutte le funzioni chiamano in ultima analisi **execve** che è l'unica vera system call della famiglia. Le differenze tra le varianti stanno nel modo in cui vengono passati i parametri.

La famiglia di system call **exec**

In molte implementazioni UNIX, solo execve è una chiamata di sistema, le altre cinque sono semplicemente funzioni di libreria che invocano execve.

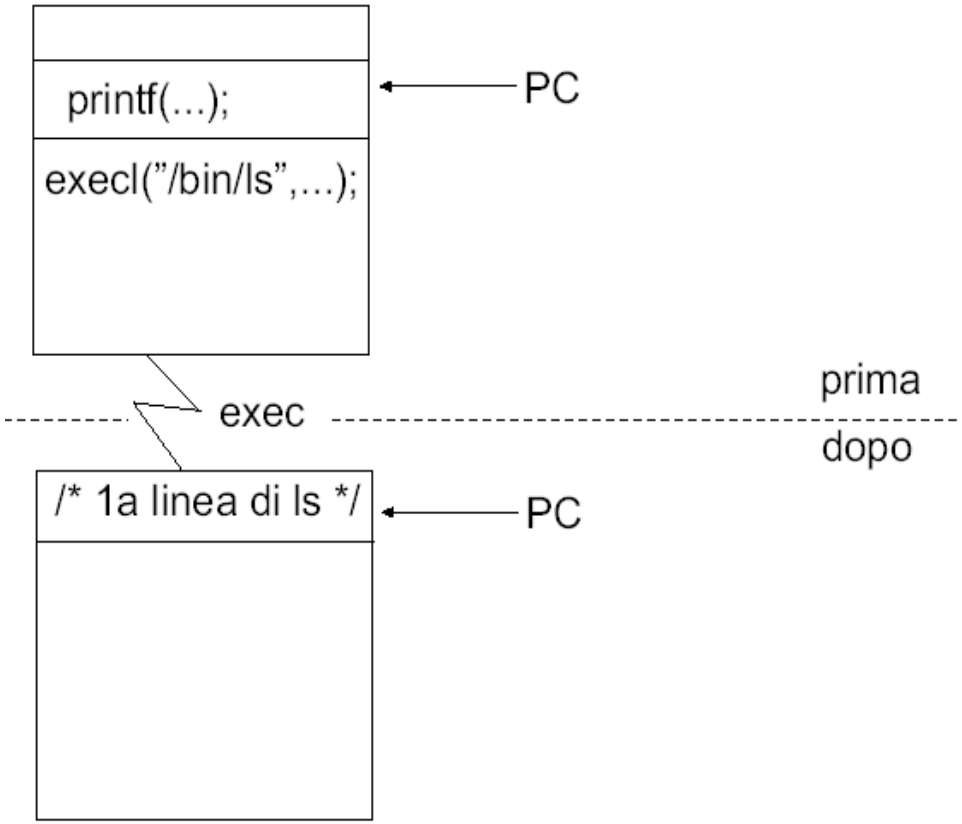


Esempio di utilizzo di **`execl`**

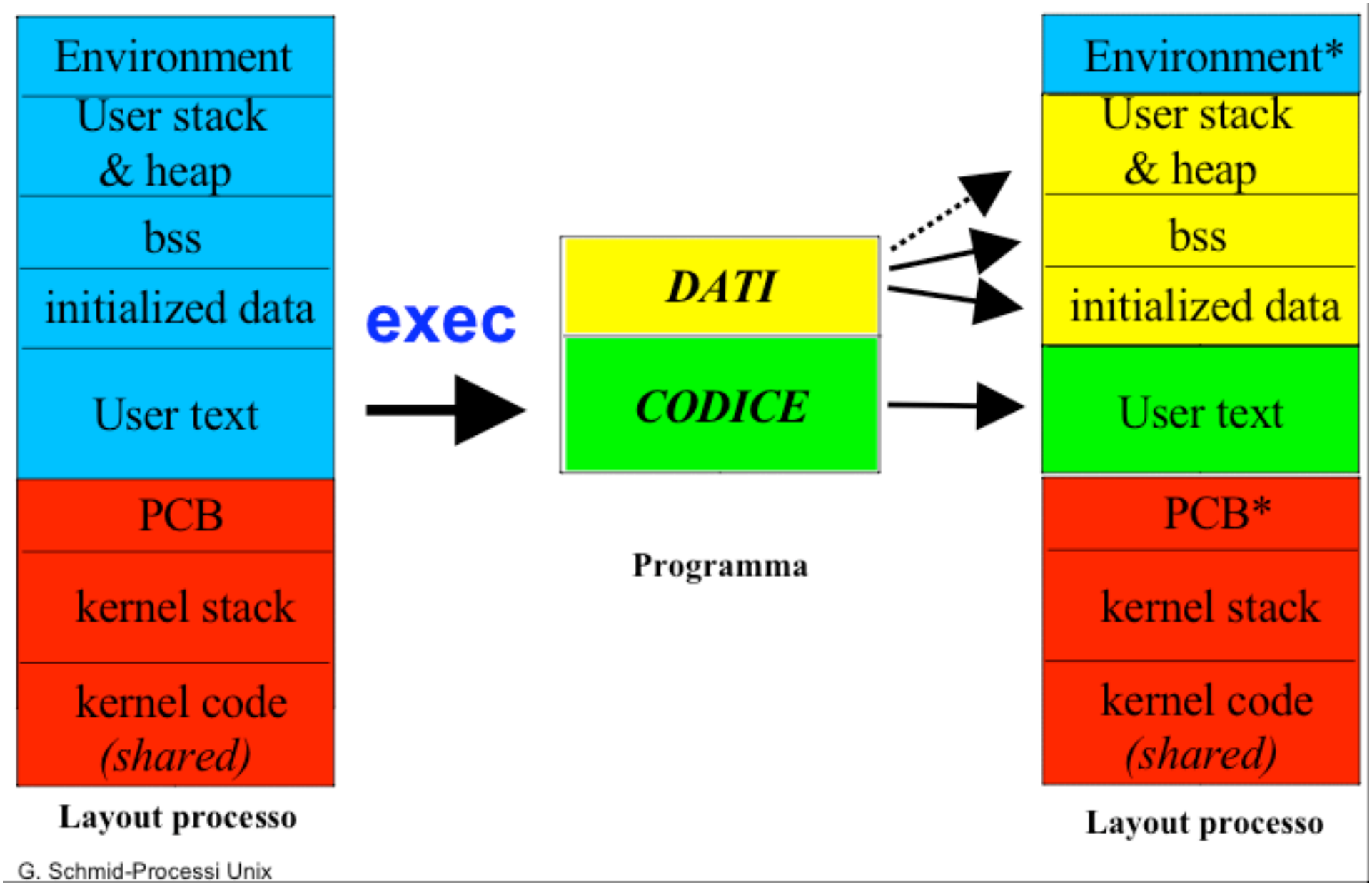
```
#include <stdio.h>
#include <unistd.h>
main()
{
    printf("Esecuzione di ls\n");
    execl("/bin/ls", "ls", "-l", (char *)0);
    perror("La chiamata di execl ha generato un
      errore\n");
    exit(1);
}
```

- Si noti che `execl` elimina il programma originale sovrascrivendolo con quello passato come parametro.
- Quindi le istruzioni che seguono una chiamata a `execl` verranno eseguite soltanto in caso si verifichi un errore durante l'esecuzione di quest'ultima ed il controllo ritorni al chiamante.

Esempio di utilizzo di **`execl`**



Esecuzione di Programmi



Chiamata alla system call `exec`

- Quando un processo chiama una delle system call `exec`:
  - il processo viene rimpiazzato completamente da un nuovo programma (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`:
  - con/senza gestione della variabile di ambiente `PATH`
    - se viene gestita la variabile d'ambiente, un comando corrispondente ad un singolo filename verrà cercato nel `PATH`
  - con variabili di ambiente ereditate / con variabili di ambiente specificate
  - con array di argomenti / con argomenti nella chiamata (null terminated)

Famiglia system call **exec**

```
int execl(char *pathname, char *arg0, … );

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

int execle(char *pathname, char *arg0, …, char* envp[]);

int execve(char *pathname, char *argv[], char* envp[]);

int execlp(char *filename, char *arg0, … );

int execvp(char *filename, char *argv[]);
```

Suffisso e parametri di **exec**

Funzione	pathname	filename	arg list	argv[ ]	environ	envp[ ]
execl	●		●		●	
execlp		●	●		●	
execle	●		●			●
execv	●			●	●	
execvp		●		●	●	
execve	●			●		●
lettere del suffisso		p	l	v		e

Eseguire Programmi

```
int execl(const char *pathname, const char *arg0, ...);
```

- execl accetta il nome di un programma da eseguire ed un numero variabile di argomenti per il programma
- l'ultimo argomento deve essere un puntatore nullo di tipo char*
- execl("a.out", "a.out", "xxx", (char *)NULL) esegue il programma a.out, con argomenti "a.out" e "xxx"

Eseguire Programmi

int **execl**(const char *pathname, const char *arg0, ...);

- se execl ha successo, il controllo non viene mai restituito al chiamante
  - il processo chiamante *diventa* il nuovo programma
- altrimenti, restituisce -1

Eseguire Programmi

- Per default, i file aperti dal processo corrente restano aperti dopo una exec
  - questo comportamento si può cambiare usando la system call fcntl
- Questo comportamento è utile per redirigere STDIN e STDOUT

Esempio di utilizzo di **execlp**

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

- Le istruzioni che seguono una chiamata a `execlp` verranno eseguite soltanto in caso si verifichi un errore durante l'esecuzione di quest'ultima ed il controllo ritorni al chiamante.
- Rispetto ad `execl` viene specificato il nome del file (non path) come nelle applicazioni lanciate da shell

Proprietà ereditate da **`exec`**

- Cosa viene ereditato da `exec`?
  - process ID e parent process ID
  - real uid e real gid
  - supplementary gid
  - process group ID
  - session ID
  - terminale di controllo
  - current working directory
  - root directory
  - maschera creazione file (umask)
  - file locks
  - maschera dei segnali
  - segnali in attesa

Proprietà NON ereditate con `exec`

- Cosa NON viene ereditato da `exec`?
  - effective user ID e effective group ID, in quanto 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)

Eseguire Programmi

int **execl**(pathname, arg0, ...)
int **execv**(pathname, char *const argv[])
int **execlp**(filename, arg0, ...)
int **execvp**(filename, char *const argv[])

Esempi d'uso:

```
char *args[] = {"ls", "-l", NULL};
execvp("ls", args);
```

equivalenti

```
execlp("ls", "ls", "-l", NULL);
```

```
execl("ls", "ls", "-l", NULL);
```

errato: ls non si trova nella directory corrente

Famiglia Exec

```
# include <unistd.h>
int execl (const char *path, const char *arg0, ... /* (char *) 0 */);
int execlp (const char *file, const char *arg0, ... /* (char *) 0 */);
int execle (const char *path, const char *arg0, ... /* (char *) 0,
                char *const envp[ ] */);
```

non ritornano se OK, ritornano -1 se EACCESS, EINVAL, EAGAIN, EINTR, ENOENT, ...

```
execl ("/home/gio/prog", "prog", "500", "out.txt", (char *) 0);
execlp ("prog", "prog", "500", "out.txt", (char *) 0);
```

```
char *env[ ] = {"USER=unknown", "PATH=/tmp", NULL};
...
execle ("/home/gio/prog","prog","500","out.txt", NULL, env);
```

Famiglia Exec

```
# include <unistd.h>
int execv (const char *path, char *const arg[ ]);
int execvp (const char *file, char *const arg[ ]);
int execve (const char *path, char *const arg[ ], char *const
envp[ ]);
```

```
char *arg_vector[ ]= {"prog", "500", "out.txt", NULL};
...
execv ("/home/gio/prog", arg_vector);
execvp ("prog", arg_vector);
```

```
char *arg_vector[ ]= {"prog", "500", "out.txt", NULL};
char *env[ ] = {"USER=unknown", "PATH=/tmp", NULL};
...
execve ("/home/gio/prog", arg_vector, env);
```

Utilizzo combinato di `fork` e `exec`

L'utilizzo combinato di `fork` per creare un nuovo processo e di `exec` per eseguire nel processo figlio un nuovo programma costituisce un potente strumento di programmazione in ambiente Linux

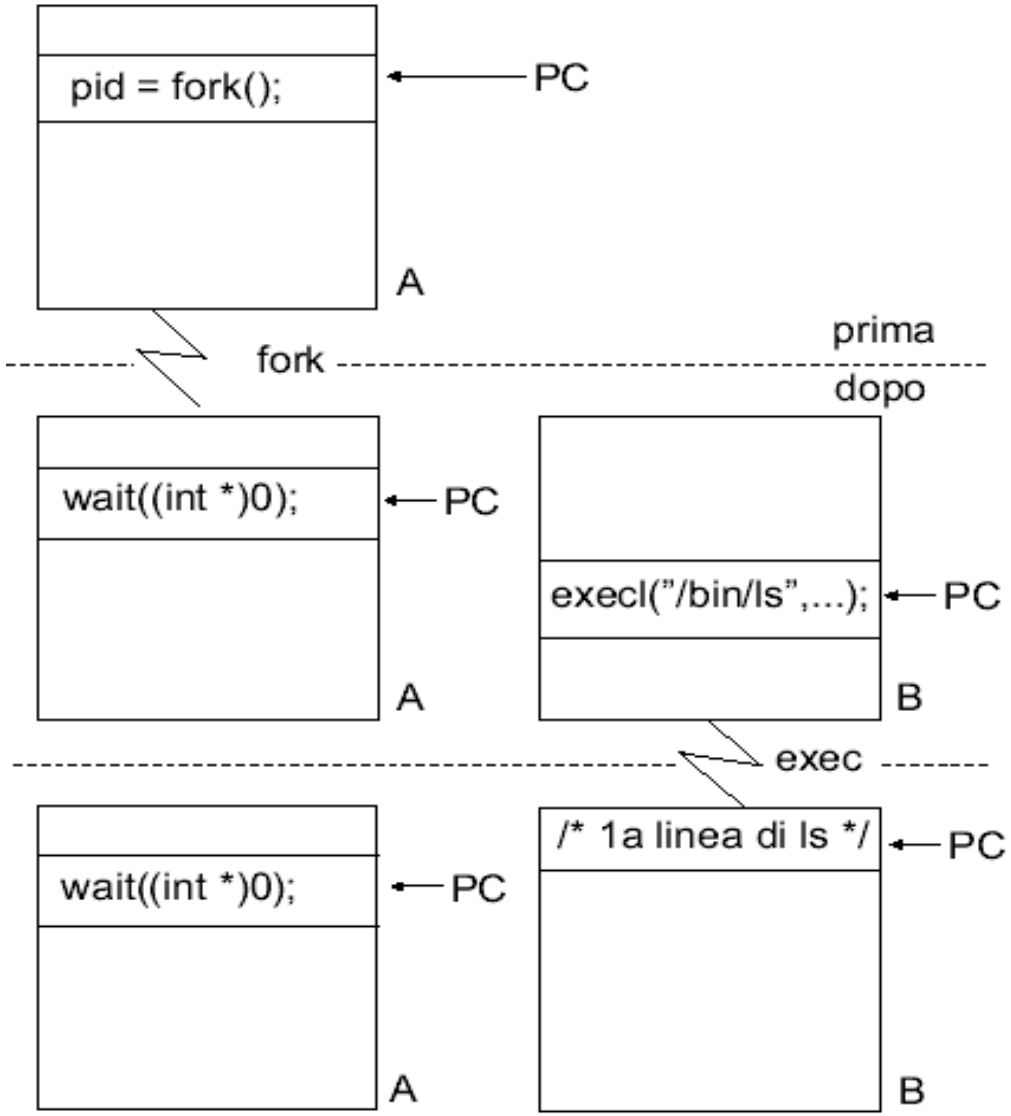
Esempio:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
void fatal(char *s)
{
    perror(s);
    exit(1);
}
/* continua ... */
```

Utilizzo combinato di **fork** e **exec**

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

Utilizzo combinato di **`fork`** e **`exec`**



Esercizio 1

Scrivere un programma che manda in esecuzione un eseguibile il cui filename e' inserito come argomento sulla linea comando e ne aspetta la terminazione.

Aggiungere al programma precedente la capacità di lanciare eseguibili residenti in qualsiasi directory memorizzata nella variabile di ambiente `$PATH`.

Esercizio (A)

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <libgen.h>

int main(int  argc, char** argv) {
  char cmd[256];
  pid_t child_pid;

  // Estrai argomento dalla linea di input per poterlo passare ad exec()
  if (argc == 2) {
    strcpy(cmd, argv[1]);
  } else {
    fprintf(stderr, "Usage: %s command\n",argv[0]);
    exit(1);
  }

  // Duplica il processo con fork
  child_pid = fork();

  if (child_pid == -1) { // Gestione dell'errore
    perror("Errore nella fork()");
    exit(2);
  }
  else if (child_pid == 0) { // Codice del processo figlio
    // Esegue il comando
    execl(cmd, basename(cmd), (char *)0);

    // Questo segmento viene eseguito solo in caso di errore nella exec
    perror("Errore nella exec()");
    exit(1); // l'exit status puo' essere rilevato dal processo padre mediante wait()
  }
```

(estrae il filename)

Esercizio1 (A)

```
  else { // Codice del processo padre
    int status;

    // Il processo attende la terminazione del figlio
    wait(&status);
    if (WIFEXITED(status) && (WEXITSTATUS(status) == 0))
      printf("Comando <%s> completato correttamente\n",cmd);
    else
      fprintf(stderr, "Errore nel comando <%s>\n",cmd);
  }

  exit(0);
}
```

Esercizio 1 (B)

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <libgen.h>

int main(int  argc, char** argv) {
  char cmd[256];
  pid_t child_pid;

  // Estrai argomento dalla linea di input per poterlo passare ad exec()
  if (argc == 2) {
    strcpy(cmd, argv[1]);
  } else {
    fprintf(stderr, "Usage: %s command\n",argv[0]);
    exit(1);
  }

  // Duplica il processo con fork
  child_pid = fork();
  if (child_pid == -1) { // Gestione dell'errore
    perror("Errore nella fork()");
    exit(2);
  }
  else if (child_pid == 0) { // Codice del processo figlio
    // Basta aggiungere un 'p' alla execl() per utilizzare PATH
    execlp(cmd, basename(cmd), (char *)0);

    // Questo segmento viene eseguito solo in caso di errore nella exec
    perror("Errore nella exec()");
    exit(1); // l'exit status puo' essere rilevato dal processo padre mediante wait()
  }
```

Esercizio2

```
  else { // Codice del processo padre
    int status;

    // Il processo attende la terminazione del figlio
    wait(&status);
    if (WIFEXITED(status) && (WEXITSTATUS(status) == 0))
      printf("Comando <%s> completato correttamente\n",cmd);
    else
      fprintf(stderr, "Errore nel comando <%s>\n",cmd);
  }

  exit(0);
}
```

Esempio 6

```
/* shell1.c: Un esempio di programma shell-like */

#include <sys/types.h>      /* per il tipo pid_t  */
#include <unistd.h>         /* per la funzione fork  */
#include <sys/wait.h>       /* per la funzione waitpid */
#include <stdio.h>          /* per le funz. printf, fgets e perror */
#include <stdlib.h>         /* per la funzione exit */
#include <string.h>         /* per la funzione strlen */
#include <errno.h>          /* per la messaggistica di errore */
#define MAXLINE 4096        /* max. num. di caratteri in buf */

int main(void)
{
      char      buf[MAXLINE];
      pid_t     pid;
      int       status;

      printf("%% ");       /*  "%"  e' il prompt di questa shell */
      while (fgets(buf, MAXLINE, stdin) != NULL) {
            buf[strlen(buf) - 1] = 0;  /* sost. newln con 0 per conformita' con la funz. execlp */

            if ( (pid = fork()) < 0)
                  perror("fork"), exit(1);        /* errore di fork */
```

Process IDentifier

DUPLICAZIONE

```
        else if (pid == 0) {        /* figlio */

                execlp(buf, buf, NULL);
                perror("execlp"), _exit(1);   /* errore di execlp */
        }

        /* pid >0, genitore */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
                perror("waitpid"), exit(1);    /* errore di waitpid  */
        printf("%% ");
    }
    exit(0);
}
```

CARICAMENTO e SOSTITUZIONE

ATTESA DEL FIGLIO

Ambiente di un processo

- L'ambiente di un processo è un insieme di stringhe (terminate da \0).
- 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 sufficiente 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;
```

L'ambiente di un processo

- L’ambiente di default di un processo coincide con quello del processo padre.
- Per specificare un nuovo ambiente è necessario usare una delle due varianti seguenti della famiglia `exec`, memorizzando in `envp` l’ambiente desiderato:

```
execle(path, arg0, arg1, ..., argn, (char *)0, envp);
execve(path, argv, envp);
```

L'ambiente di un processo

```
/* setmyenv */
#include <unistd.h>
#include <stdio.h>
main()
{ char *argv[2], *envp[3];
    argv[0] = "setmyenv";
    argv[1] = (char *)0;
    envp[0] = "var1=valore1";
    envp[1] = "var2=valore2";
    envp[2] = (char *)0;
    execve("./showmyenv", argv, envp);
    perror("execve fallita");
}
```

- ***Eseguendo il programma precedente si ottiene quanto segue:***

```
$ ./setmyenv
var1=valore1
var2=valore2
```

L'ambiente di un processo

- Esiste una funzione della libreria standard che consente di cercare in `environ` il valore corrispondente ad una specifica variabile d'ambiente:

```
#include <stdlib.h>
char *getenv(const char *name);
```

- getenv prende come argomento il nome della variabile da cercare e restituisce il puntatore al valore (ciò che sta a destra del simbolo =) o NULL se non lo trova:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    printf("PATH=%s\n", getenv("PATH"));
}
```

- Dualmente, `putenv` consente di modificare o estendere l'ambiente:

```
putenv("variabile=valore");
```

Current working directory e root directory

- Ad ogni processo è associata una current working directory che viene ereditata dal processo padre.
- La chiamata di sistema seguente consente di cambiarla:

```
#include <unistd.h>
int chdir(const char *path);
```

- Ad ogni processo inoltre è associata una root directory che specifica il punto di inizio del file system visibile dal processo stesso.
- Per cambiare la root directory si può usare la chiamata di sistema seguente:

```
#include <unistd.h>
int chroot(const char *path);
```

Esempio 7

```
/* exec.c: Dimostra il funzionamento di due funzioni exec */
#include <sys/types.h>    /* per il tipo pid_t  */
#include <unistd.h>       /* per le funzioni fork  execle ed execlp */
#include <sys/wait.h>     /* per la funzione waitpid */
#include <stdio.h>        /* per la funz. printf */
#include <stdlib.h>       /* per la funzione exit */

char *ambiente[ ] = { "USER = nobody", "PATH=/tmp", NULL };

int main(void)
{
 pid_t   pid;

 if ( (pid = fork()) < 0) printf("errore fork"), exit(1);

 else if (pid == 0) {          /* 1mo figlio */
   if (execle("/home/gio/echoall", "echoall", "1mo arg",
              "2ndo e ultimo", NULL, ambiente) < 0)
               printf("errore di execle"), _exit(1);
 }
```

Esecuzione del programma echoall con specifica del pathname e dell'ambiente

Esempio 7 (cont.)

```
  /* pid >0, genitore */
  if ( (pid = waitpid(pid, NULL, 0)) < 0)
    printf("errore di waitpid"), exit(1);

  if ( (pid = fork()) < 0)
    printf("errore di fork"), exit(1);

  else if (pid == 0) {          /* 2ndo figlio */
     if (execlp("echoall", "echoall", "1mo e ultimo", NULL) < 0)
                printf("errore di execlp"), _exit(1);
  }

  /* pid >0, genitore */
  if ( wait(NULL) != pid)
     printf("errore di wait"), exit(1);
  exit(0);
}
```

Esecuzione del programma echoall senza specifica del pathname e dell'ambiente

Esempio 7 (cont.)

```
/* echoall.c: Il programma eseguito da exec.c  */

#include <stdlib.h>      /* per la funzione exit */
#include <stdio.h>       /* per la funzione printf */

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

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

   for (ptr=environ; *ptr != 0; ptr++)   /* e di tutte le stringhe
                                            d'ambiente */
         printf("%s\n", *ptr);

        exit(0);
}
```

Esempio 7 (cont.)

Compilando ed eseguendo, si ottiene:

```
gio$ pwd
/home/gio
gio$ cc -o echoall echoall.c
gio$ cc -o exec exec.c
gio$ echo $PATH
/usr/bin:/usr/local/bin:/home/gio
gio$ ./exec
argv[0] : echoall
argv[1] : 1mo arg
argv[2] : secondo e ultimo
USER=nobody
PATH=/tmp
argv[0] : echoall
argv[1] : 1mo e ultimo
USER=gio
HOME=/home/gio
...
```

Race Condition

Una race condition (condizione di tempificazione) si verifica quando più processi elaborano dati condivisi e l'effetto dell'insieme di siffatte elaborazioni dipende dall'ordine in cui i processi sono eseguiti.

L'ordine in cui vengono elaborate le istruzioni per un genitore ed un figlio dopo un fork (vfork) in generale dipende dallo scheduler e dal carico del sistema.

Una chiamata fork (vfork) può causare una race condition, se le istruzioni relative al genitore ed al figlio operano su dati condivisi;

Il rilevamento in base a run di una race condition del tipo suddetto può essere molto difficile.

Esempio 8

```
/* racecond.c: Fornisce un esempio di condizione di
tempificazione  */
#include <sys/types.h>  /* per il tipo pid_t  */
#include <unistd.h>       /* per  fork  ed  _exit */
#include <stdio.h>        /* per  printf e putc */
#include <stdlib.h>       /* per  exit */

static void char_char(char *);  /* dich. di char_char */

void main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        printf("errore di fork"), exit(1);
    else if (pid == 0)  {
       char_char("io sono il figlio, e intendo scrivere prima del
padre\n");
        _exit(0);
    }
```

Esempio 8 (cont.)

```
else   {
        char_char("io sono il padre, e non ci sto\n");
        exit(0);
    }
}
/* definizione di char_char */

static void char_char(char *str)
{
    char *ptr;
    int    c;

    setbuf(stdout, NULL);    /* disattiva il buffer per stdout  */
    for (ptr = str; c= *ptr++; )
        putc(c, stdout);
 }
```

Esempio 8 (cont.)

Eseguendo l'esempio, si ottiene:

```
gio$ io sono il padre, ei non  oc i sosnto
o il figlio, e intendo scrivere prima delpgio$ adre
```

Osservazione: In questo esempio, il padre termina prima che il figlio abbia finito di scrivere sullo stdout, come si evince dalla comparsa prematura del prompt. Il prompt alla terminazione del figlio non appare perchè quest'ultimo per uscire richiama _exit, che non restituisce il controllo al chiamante (la shell).

Funzione vfork()

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

Crea nuovo processo come fork, ma non copia spazio inirizzamento, eseguito nello spazio del genitore finche' figlio non esegue exec o exit.

Finche' non eseguito exec o exit, figlio eseguito per primo

vfork utilizzato con exec per l'esecuzione di un programma

Esempio

```
#include <systype.h>
#include “ourhdir.h” /* tutti header necessari*/

int glob = 6;
int main(void) {
  int var; pid_t pid;
  var = 88;
  printf(“before vfork\n”);
  if ((pid =vfork())) perrror(“vfork”), exit(0);
  else if (pid==0) {
    glob++; var++; /* cambia variabili del padre */
   _exit(0);}
printf(“pid = %d glob = %d var = %d”,getpid(), glob,var);
exit(0);
}
```

Esempio

Eseguendo:

$./a.out

before vfork

pid = 2624, glob = 7, var = 89

Cioe', l'incremento della var del figlio cambia il valore nel genitore.

File di Interprete

Un file (di) interprete è un file ASCII che incomincia con una linea del tipo:

#! *pathname [argomenti-opzionali]*

dove *pathname* è il path assoluto di un programma, e *argomenti-opzionali* sono le eventuali opzioni con cui va eseguito il programma.

- Generalmente tali file sono creati allo scopo di far eseguire le istruzioni rappresentate dalle linee successive al programma interprete (ad es. sh, csh, perl,..) specificato tramite *pathname* ;

- Un exec di tali file non provoca la loro esecuzione, bensì quella di *pathname*.

La funzione **`system`**

```
int system(char* command);
```

- Esegue un comando, aspettando la sua terminazione
- E' una funzione della libreria standard definita in ANSI C (quindi non è una system call, anche se svolge una funzione analoga alla `fork+exec`)
- Il suo modo di operare è fortemente dipendente dal sistema; in genere chiama `/bin/sh -c command`
- Non è definita in POSIX.1, perché non è un'interfaccia al sistema operativo, ma è definita in POSIX.2

Funzione system

E' comodo poter eseguire un comando UNIX da un programma C. Per tale ragione l'ANSI C definisce la funzione system, che rappresenta un interfaccia alla Bourne shell implementata mediante le primitive fork, exec e waitpid:

```
#include<stdlib.h>
int system(const char *cmdstring);
```

```
...
system("date >miofile");
...
```

```c
/*
* A simple implementation of system() (without signal handling)
* Written by W. Richard Stevens
*/

#include    <sys/types.h>
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int system(const char *cmdstring)   /* version without signal handling */
{
    pid_t   pid;
    int status;

    if (cmdstring == NULL)
        return(1);  /* always a command processor with Unix */

    if ( (pid = fork()) < 0) {
        status = -1;    /* probably out of processes */

    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127); /* execl error */
    } else {                        /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    }

    return(status);
}
```

Esercizio

- Programma che genera due processi, il primo scrive una stringa su di un file, il secondo la legge.

Esercizio

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#define BUFFSIZE 4096 /* max. num. di caratteri in buf */

int main(int argc, char* argv[]){
  pid_t pid;  int status;  int fd;  int n;
  char buffer[BUFFSIZE];  /* controlla i parametri input */
  if (argc != 3)
      printf("usage: trasfile <file> <stringa>"), exit(1);
/* genera il processo figlio */
  if ( (pid = fork()) < 0)
    perror("fork"), exit(1);
```

```
/* errore di fork */
  else if (pid == 0) { /* figlio */
    /* apre il file da scrivere */
    if ((fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU))<0)
      perror("create file error"), exit(1);
      /* scrive la stringa */
    if ((n = write(fd, argv[2], strlen(argv[2]))) < 0)
      printf("write error"),exit(1);
    /* chiude il file */
    close(fd);
  } else { /* padre */
    /* attende la terminazione del figlio */
    if ( (pid = waitpid(pid, &status, 0)) < 0)
      perror("waitpid error"), exit(1);
    /* apre il file */
    if ((fd = open(argv[1], O_RDONLY, S_IRWXU)) < 0)
      perror("open file error"), exit(1);
        /* legge la stringa */
    if ((n = read(fd, buffer, BUFFSIZE)) < 0)
      printf("read error"),exit(1);
    /* stampa la stringa */
printf("trovato: %s", buffer);
        /* chiude il file */
    close(fd);
  } exit(0);}
```

Esercizio

- Trasformare il precedente programma utilizzando exec.

Esercizio

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#define BUFFSIZE 4096 /* max. num. di caratteri in buf */

int main(int argc, char* argv[])
{
  pid_t pid;
  int status;
  int fd;
  int n;
  char buffer[BUFFSIZE];

  /* controlla i parametri input */
  if (argc != 3)
      printf("usage: trasfile <file> <stringa>\n"), exit(1);

  /* genera il processo figlio */
  if ( (pid = fork()) < 0)
    perror("fork"), exit(1); /* errore di fork */
  else if (pid == 0) { /* figlio */
    /* apre il file da scrivere */
           printf("sono processo figlio e mando execlp \n");
           execlp("./scriviStringa","scriviStringa",argv[1],argv[2],(char *)0);
           perror("execlp"), exit(1);
    /*
    if ((fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU))<0)
      perror("create file error"), exit(1);*/
```

Esercizio

```
 } else { /* padre */
   /* attende la terminazione del figlio */
   if ( (pid = waitpid(pid, &status, 0)) < 0)
     perror("waitpid error"), exit(1);

   printf("sono il padre e leggo \n");
   /* apre il file */
   if ((fd = open(argv[1], O_RDONLY, S_IRWXU)) < 0)
     perror("open file error"), exit(1);

   /* legge la stringa */
    if ((n = read(fd, buffer, BUFFSIZE)) < 0)
      printf("read error"),exit(1);

    /* stampa la stringa */
    printf("trovato: %s", buffer);

    /* chiude il file */
    close(fd);
  }

  exit(0);
}
```

Esercizio

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

int main(int argc, char* argv[]){
int n;
int fd;

printf("sono il figlio e scrivo %s su \n",argv[2],argv[1]);
if ((fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU))<0)
      perror("create file error"), exit(1);

    /* scrive la stringa */
    if ((n = write(fd, argv[2], strlen(argv[2]))) < 0)
      printf("write error"),exit(1);

    /* chiude il file */
    close(fd);
 return 0;
}
```