Processi UNIX

(Contiene lucidi tratti dal corso del Prof. Giovanni Schmid)

Funzione Main

Un programma C inizia la sua esecuzione obbligatoriamente con la chiamata alla funzione **main**. Lo standard C stabilisce che la funzione main può non avere argomenti o prendere i due argomenti argc e argv[]:

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

argc rappresenta il numero di argomenti passati in input (incluso il nome del programma, definito dal nome del file contenente la chiamata a main);

argv è un vettore di puntatori a carattere e rappresenta la lista di argomenti (opzioni e parametri) del programma, con argv[0] che rappresenta il nome del *programma*

Lista di Ambiente

Ad ogni programma viene passata, oltre alla lista degli argomenti di input, una lista d'ambiente. Essa serve alle applicazioni per definire il contesto in cui operano (directory home, tipo di terminale, nome utente, etc.)

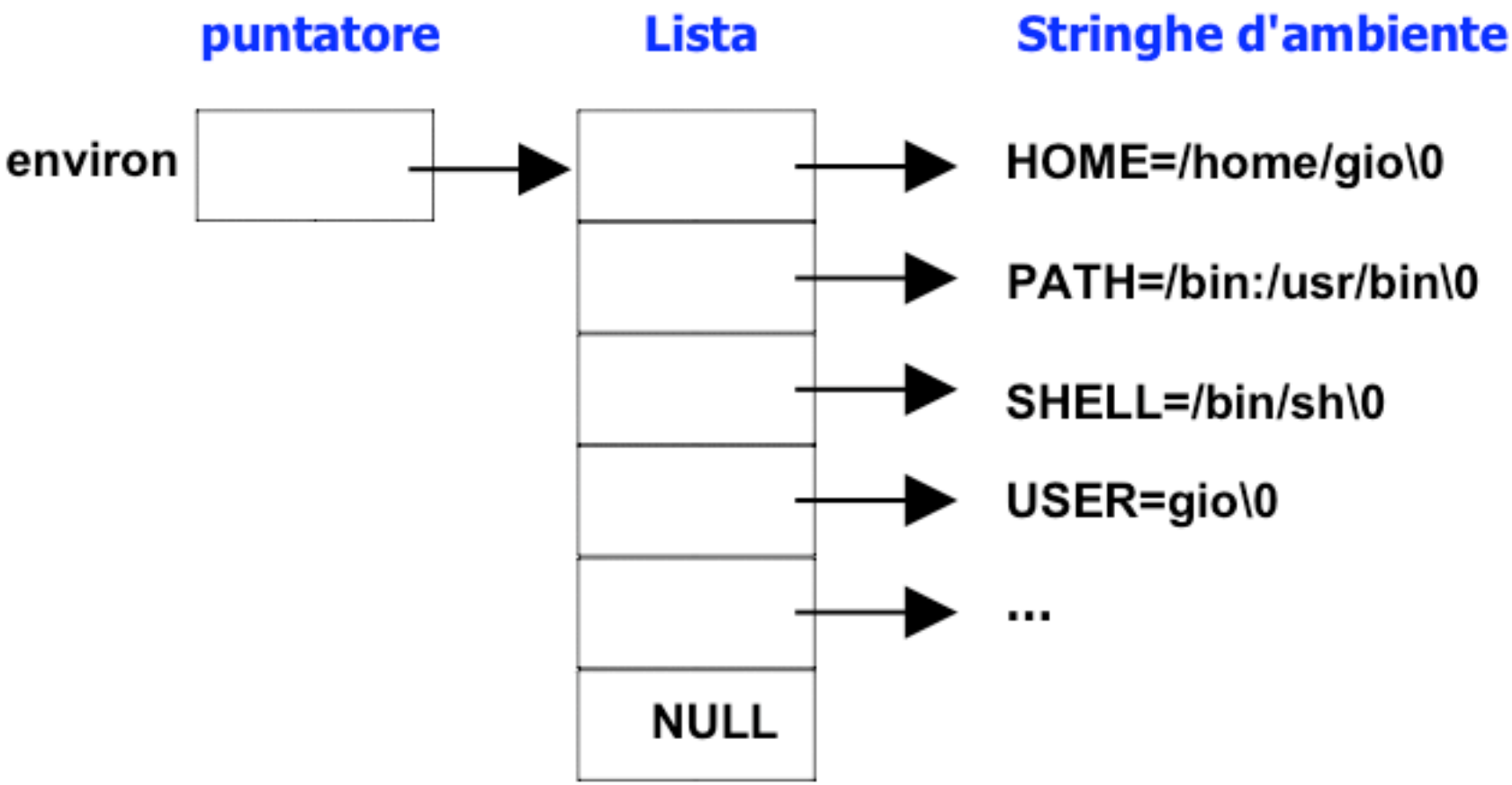
Analogamente alla lista degli argomenti, la lista d'ambiente è un array di puntatori a carattere, in cui l'ultimo puntatore punta a NULL.

Ciascun puntatore della lista contiene l'indirizzo di una stringa del tipo nome = valore, detta stringa d'ambiente.

L'indirizzo dell'array di puntatori è contenuto nella variabile globale **environ**:

extern char **environ*;

Lista di Ambiente



Variabili di Ambiente

Le variabili d'ambiente sono definite e modificate operando sulle stringhe d'ambiente, grazie ad opportune funzioni. Le due più importanti sono:

```
# include <stdlib.h>

char *getenv(const char *name);
int putenv(char *string);
```

ritorna zero se OK, non zero se ENOMEM

getenv restituisce il puntatore alla stringa valore, associata al nome della variabile d'ambiente nome nella stringa d'ambiente nome=valore. Se nome non esiste restituisce il puntatore nullo.

putenv aggiunge alla lista d'ambiente la stringa *string* della forma nome=valore. Se nome esiste ne aggiorna il valore.

Layout in memoria di un Prog.

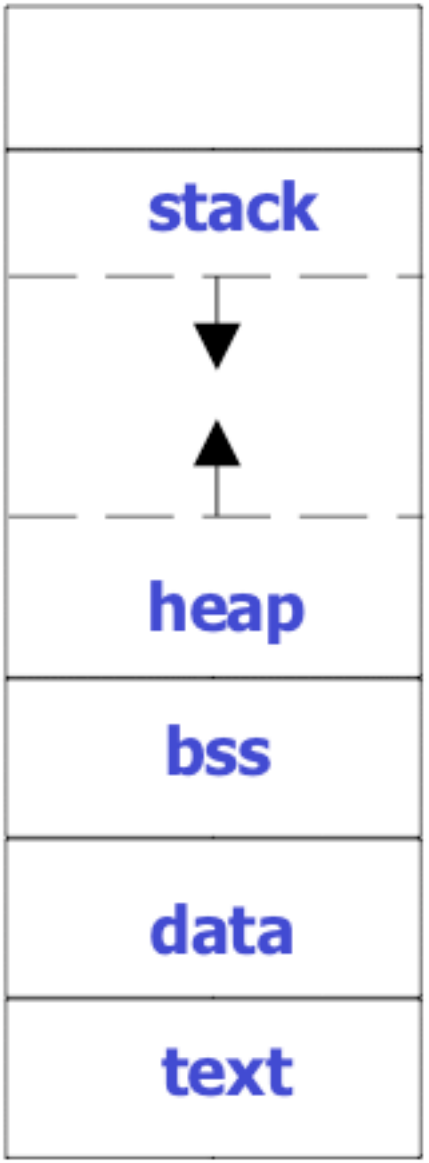
Questo segmento dell'area di memoria riservata ad un programma serve per la memorizzazione degli argomenti della linea di comando e delle variabili d'ambiente.

Lo **stack** *serve a* memorizzare le variabili locali e le informazioni relative alle chiamate di funzioni. Lo *stack* è implementato come una coda *LIFO (Last In, First Out),* per permettere l'uso annidato e ricorsivo delle funzioni.

Lo *heap è dove viene effettuata l'*allocazione dinamica di memoria (funzioni malloc, calloc, realloc, free,...). Anche quest'area è implementata come una coda LIFO, ed è tipicamente gestita a basso livello dalla syscall sbrk.

Le variabili globali non inizializzate, analogamente ai puntatori a variabili globali, vengono memorizzati nel segmento **bss**, detto anche *uninitialized data segment*. I dati in tale segmento sono inizializzati dal kernel come valori numerici pari a zero o puntatori nulli prima che il programma vada in esecuzione.
L'area **data** contiene le variabili globali ed inizializzate del programma.

L'area text contiene le istruzioni macchina relative al programma. Quest'area è in sola lettura, perchè condivisa tra tutti i processi che eseguono uno stesso codice binario.

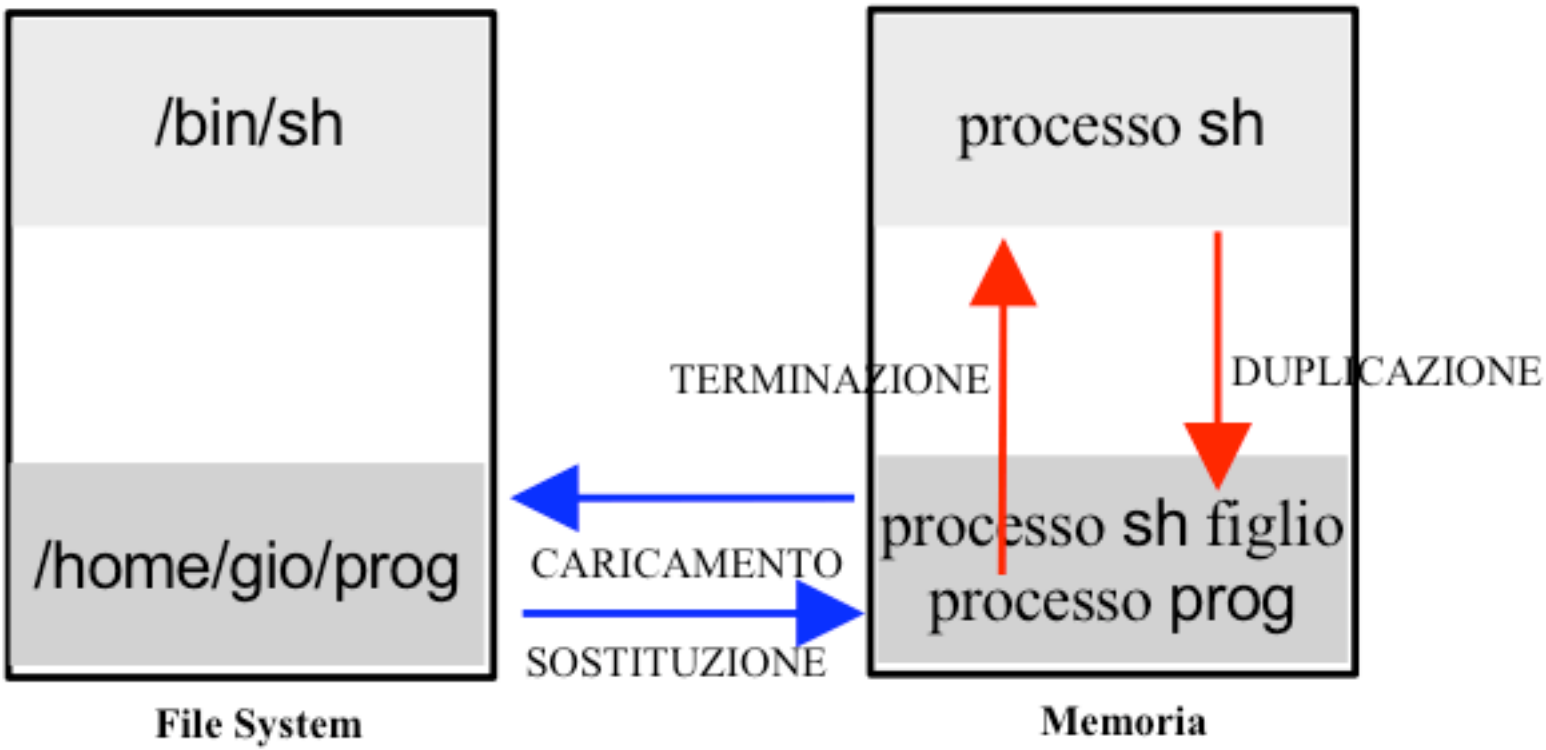


G. Schmid-Processi Unix

Esecuzione di un Prog.

Qual'è il meccanismo per l'esecuzione di un nuovo programma in UNIX?

```
$ prog  <invio>
<eventuale output di prog>
$
```



Esecuzione del Prog.

- Quando il prog. viene eseguito (una funzione exec), una routine start-up prende i command-line args e l'env prima del lancio della funzione main
- La fine dell'esecuzione avviene in diversi modi: return, exit, _exit, pthread_exit, abort, segnale, etc.

Controllo processi

Per il controllo dei processi occorrono primitive per gestire:

- Creazione di Processi
- Esecuzione di Programmi
- Terminazione di Processi
- Identificazione e proprietà dei processi

Controllo di Processi

Il controllo dei processi in UNIX si esplica mediante:

- L'identificazione dei processi:
  Identificatore di processo (pid);
  Identificatore di gruppo-processi (pgid)
  Identificatori di utente (famiglia uid);
  Identificatori di gruppo-utenti (famiglia gid);

- L'impiego di funzioni (primitive di controllo di processo) per:
  Duplicare un processo esistente (**fork**, **vfork**) ;
  Caricare un nuovo programma (famiglia **exec**);
  Attendere la terminazione di un processo (**wait**, **waitpid**);
  Terminare un processo (**exit**, **_exit**).

Identificazione di Processi

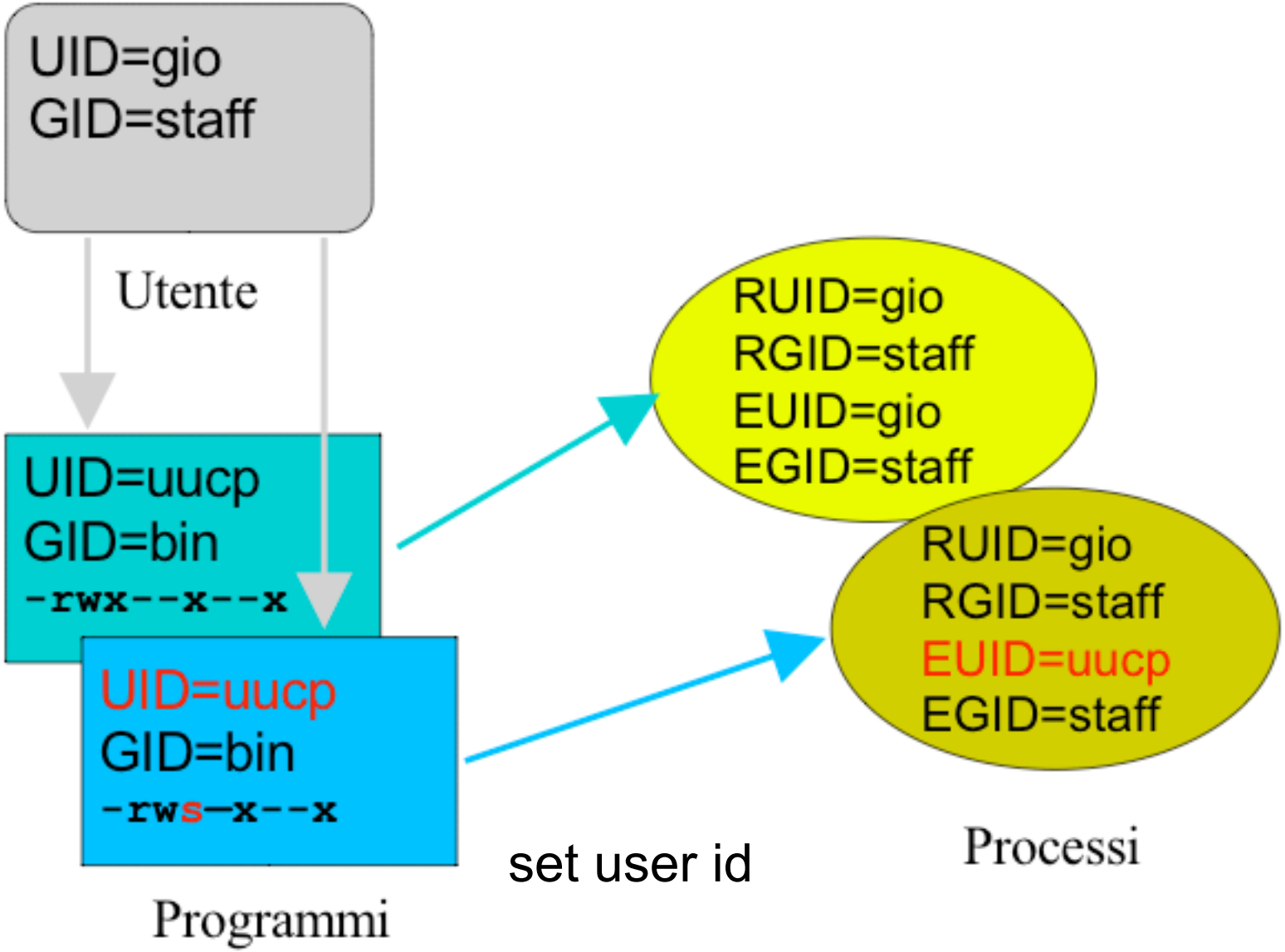
I processi in UNIX si dividono in processi di sistema e di utente;

Ogni processo è identificato da un numero non negativo (PID); unico PID, pero' reciclato quando termina;
PID=0 scheduler, PID=1 init, PID = 2 pagedaemon

Ogni processo utente eredita una serie di altri identificativi:

- Il gruppo di processi cui esso appartiene (PGID);
- Lo UID e il GID dell'utente che lo ha mandato in esecuzione (RUID e RGID);
- Lo UID e i GID (gruppi primario e supplementari) di utente con i quali esso ha accesso ai file (EUID, EGID, suppl. EGID);

Identificazione dei Processi



Identificazione di Processi

- Ogni processo ha un pid ed un Parent's pid (ppid)
  - i processi formano un albero
- “init” e' la radice dell'albero
  - viene lanciato direttamente dal kernel
  - non ha padre
  - ha pid=1
- quando un processo termina, i suoi figli diventano figli di “init”

Ottenere pid e ppid

```
pid_t getpid(void);
pid_t getppid(void);
```

Restituiscono pid e ppid, rispettivamente

Non possono fallire

Di solito pid_t e' sinonimo di int pid_t

Identificazione di Processi

Le seguenti funzioni restituiscono gli identificativi e le credenziali di processo:

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

pid_t getpid(void);      identificativo del processo      PID
pid_t getppid(void);     identificativo del genitore      PPID

uid_t getuid(void);      credenziale utente reale         RUID
uid_t geteuid(void);     credenziale utente effettivo     EUID

gid_t getgid(void);      credenziale gruppo reale         RGID
gid_t getegid(void);     credenziale gruppo effettivo     EGID
```

Creazioni di Processi

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

pid_t fork(void);
pid_t vfork(void);
```

(vfork da usare con exec, lavora nello spazio del genitore e aspetta il figlio)

ritornano il PID se OK, -1 se EAGAIN, ENOMEM o EPERM

L'unico modo per istruire il kernel a creare un nuovo processo è di chiamare la funzione **fork** (**vfork**) da un processo esistente. Il processo creato viene detto figlio. Il processo che chiama **fork** (**vfork**) viene detto genitore.

Ogni chiamata a **fork** (**vfork**) ha due ritorni:

- al genitore viene restituito l'identificativo del figlio;
- al figlio viene restituito l'identificativo 0.

Creazione di Processo

Esempio tipico:

```
pid_t pid;


if ( (pid=fork()) < 0 )
    perror("fork"), exit(1);
else if (pid != 0) {
    // codice del padre
} else {
    // codice del figlio
}
```

Padri e Figli

- Il figlio procede indipendentemente dal padre
- **memoria**: il figlio ottiene una copia nuova della memoria del padre (variabili globali e locali)
- **file aperti**: i descrittori vengono copiati come con dup; i processi condividono l'offset!
- **segnali**: per ogni segnale, il figlio continua ad avere la stessa reazione del padre

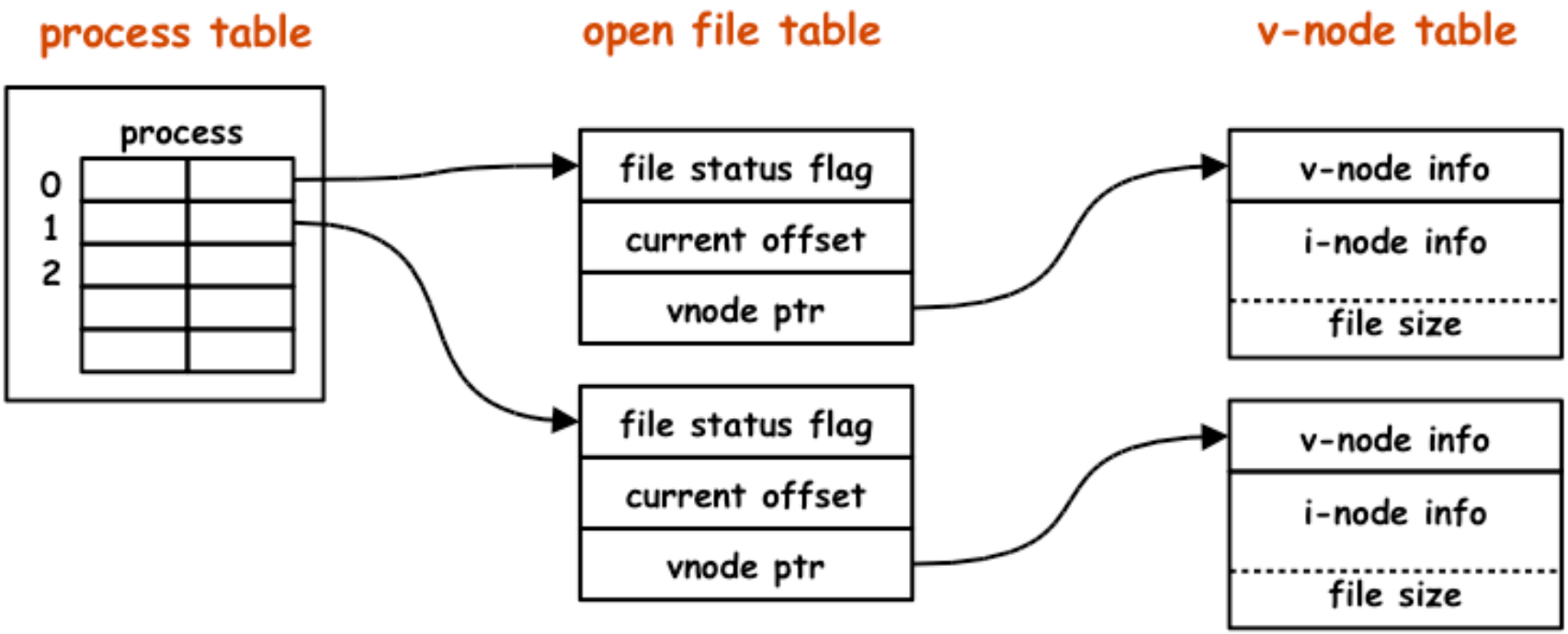
Creazione di Processi-fork

Ad una chiamata **fork** il kernel esegue le operazioni seguenti:

- Alloca uno spazio nella tabella dei processi per il figlio;
- Assegna un PID al figlio, unico nel sistema;
- Fa una copia dell'immagine del genitore, ad eccezione dei segmenti di memoria condivisi;
- Incrementa i contatori dei file del genitore, per registrare che anche il figlio possiede tali file;
- Assegna al processo figlio lo stato READY;
- Restituisce il PID del figlio al genitore e il PID 0 al figlio;
- A seconda della routine di allocazione, può:
  - rimanere nel genitore;
  - trasferire il controllo al figlio;
  - trasferire il controllo ad un altro processo.

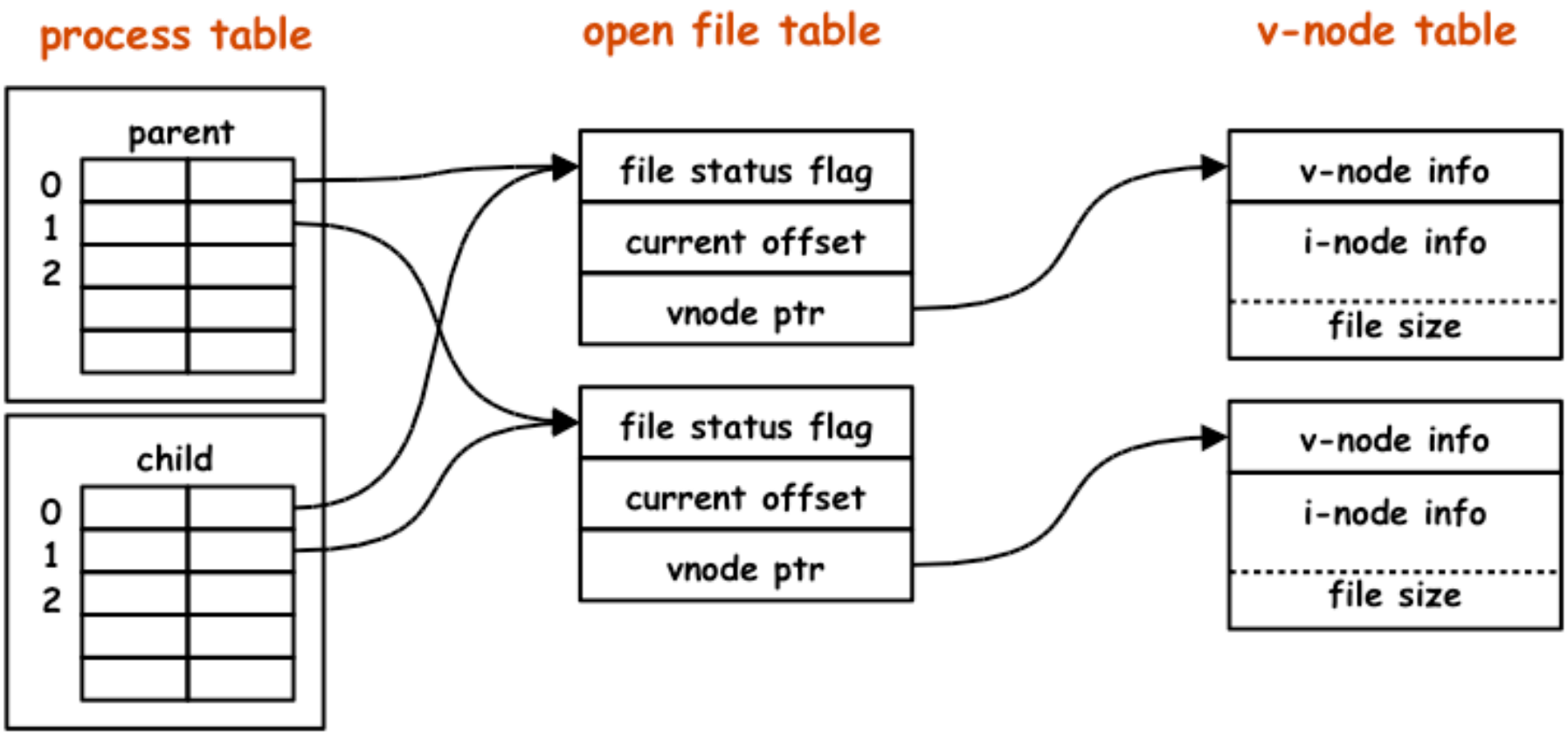
Fork e File

- Una caratteristica della chiamata **`fork`** è che tutti i descrittori che sono aperti nel processo parent sono duplicati nel processo child
- Prima della **`fork`**:



Fork e File

- Dopo la **`fork`**:



Fork e File

- E' importante notare che padre e figlio condividono lo stesso *file offset*
- Consideriamo il seguente caso:
  - un processo esegue **`fork`** e poi attende che il processo figlio termini (system call **`wait`**)
  - supponiamo che lo **`stdout`** sia rediretto ad un file, e che entrambi i processi scrivano su **`stdout`**
  - se padre e figlio non condividessero lo stesso offset, avremmo un problema:
    - il figlio scrive su **`stdout`** e aggiorna il proprio current offset
    - il padre sovrascrive **`stdout`** e aggiorna il proprio current offset

Esercizio

- Scrivere un programma C che apre un file, effettua una **`fork`** e scrive messaggi diversi sul file a seconda che sia padre o figlio. Come si alternano i messaggi nel file?
- Nel programma prima descritto spostare la open dopo la **`fork`** e verificare se il contenuto del file è cambiato rispetto al programma precendete e spiegarne il perchè.

Esempio

```
int main(void) {
  int i;

  for (i=0; i<2 ;i++)
    if (fork()>0) {
      printf("Padre! %d\n", i);
    } else {
      printf("Figlio! %d\n", i);
    }

  sleep(10);
  return 0;
}
```

- Qual è l'output di questo programma?
- Quanti processi vengono creati?
- Di chi è figlio ciascun processo creato?

Esempio

```
$./a.out
padre 0
padre 1
figlio! 0
padre 1
figlio! 1
figlio! 1
```

padre 0

padre1 figlio! 1

figlio! 0

padre1 figlio! 1

Esempio 2

```
/* fork1.c: fork e le immagini in memoria di genitore e figlio */
#include <sys/types.h>   /* per il tipo pid_t */
#include <unistd.h>      /* per le funzioni fork e sleep */
#include <stdlib.h>      /* per la funzione exit */
#include <stdio.h>       /* per la funzione printf */

int glob_ini=1;    /* var. globale inizializzata --> data segment */
int glob;          /* var. globale non inizializzata --> bss segment */

int main(void)
{
   /* variabili locali --> stack */
   int local=10;
   pid_t pid;
   printf("Prima di chiamare fork\n"); /* Output con buffer */
   /* chiamata a fork */
   if ( ( pid=fork( ) ) < 0 ) /* Si e' verificato un errore di fork */
      {
       printf(" errore di fork");
       return 1;
      }
```

Esempio 2 (cont.)

```
else if ( pid == 0 )
    {
      glob_ini++;
      glob=5;
      local++;
    }

  else        /* Il genitore aspetta nullafacente per 2 secondi */
    sleep(2);

  /* entrambi i processi scrivono sullo std output */
  printf("pid= %d, glob_ini= %d, glob= %d, local= %d\n", getpid( ),
glob_ini, glob, local);
  return 0;
}
```

Esempio 2 (cont.)

Eseguendo fork1, si ottiene:

```
gio$ ./fork1
Prima di chiamare fork
pid= 7183, glob_ini= 2, glob= 5, local= 11
pid= 7182, glob_ini= 1, glob= 0, local= 10
gio$ ./fork1 >fork1.out
gio$ cat fork1.out
Prima di chiamare fork
pid= 7189, glob_ini= 2, glob= 5, local= 11
Prima di chiamare fork
pid= 7188, glob_ini= 1, glob= 0, local= 10
```

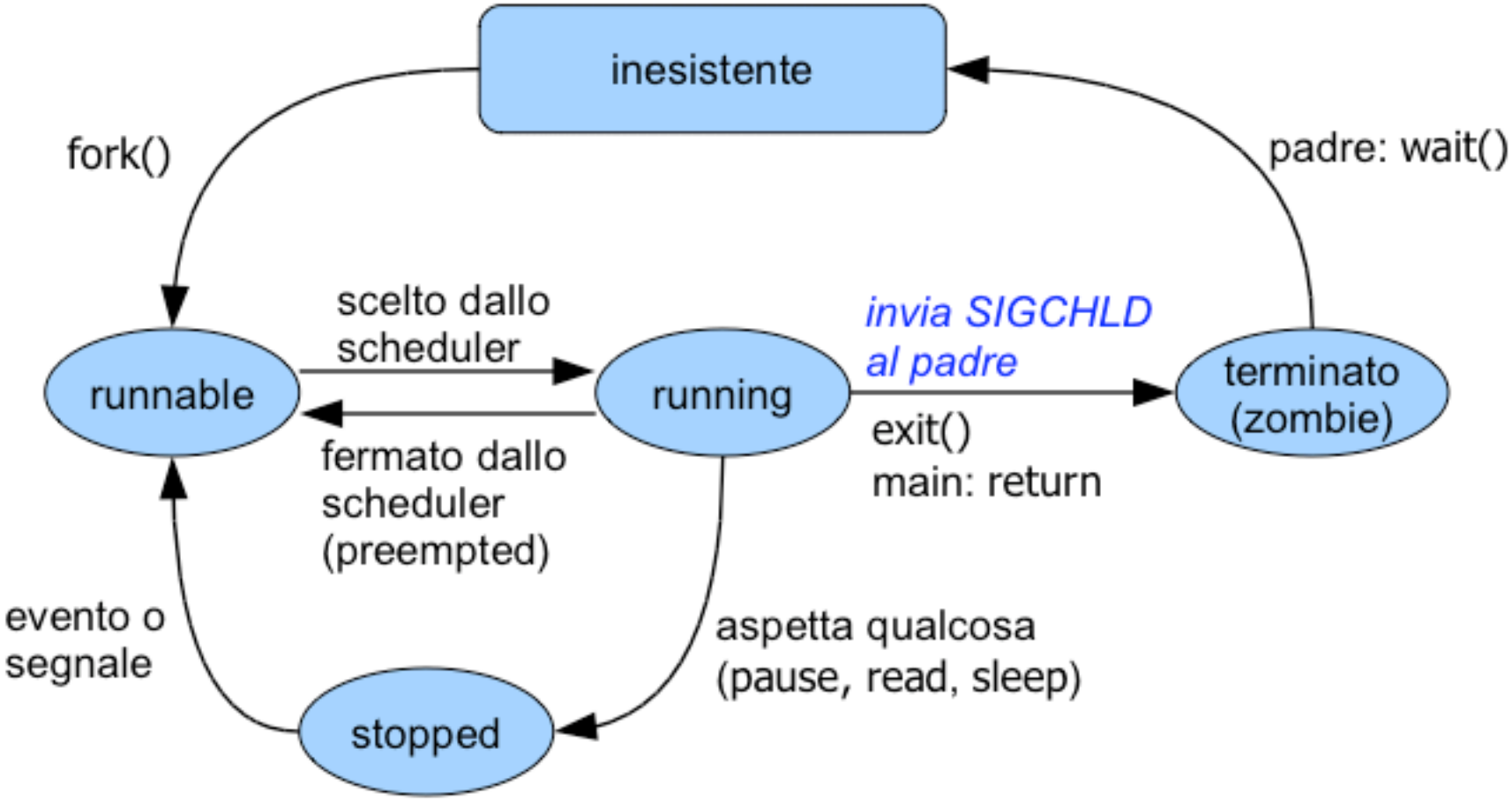
figlio

genitore

Quando un processo termina

- viene inviato il segnale **SIGCHLD** al padre
- il processo diventa uno “zombie” finchè il padre non chiama una **wait/waitpid**

Ciclo di vita del processo



Terminazione Processo

- **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 Processo

- **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`**

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*

Aspettare un figlio

```
pid_t wait(int *status);
```

- Blocca il processo finché un figlio termina
  - non blocca se c'è un figlio zombie
- Restituisce il pid del processo terminato
  - -1 in caso di errore
  - ad esempio, se un processo non ha figli
- “status” contiene il valore di uscita del figlio
  - se non ci interessa, passiamo NULL

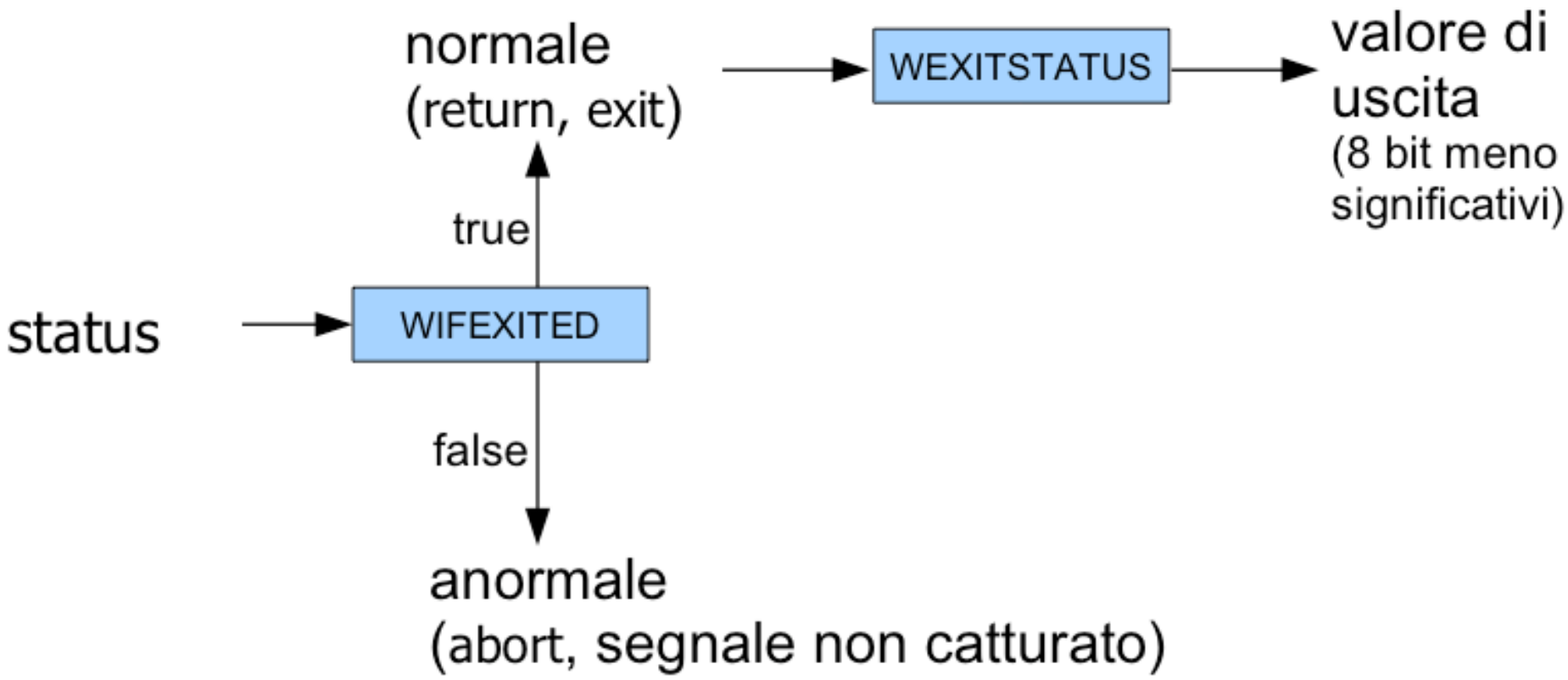
Stato di uscita del Figlio

Esempio tipico:

```
int status;


wait(&status);
if ( WIFEXITED(status) )
    printf("valore di uscita: %d\n", WEXITSTATUS(status));
else
    printf("terminazione anomala\n");
```

Stato del figlio



Aspettare un figlio

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

- Come wait, ma aspetta un figlio specifico
- options può essere lasciato a zero

Esecuzione di Programmi

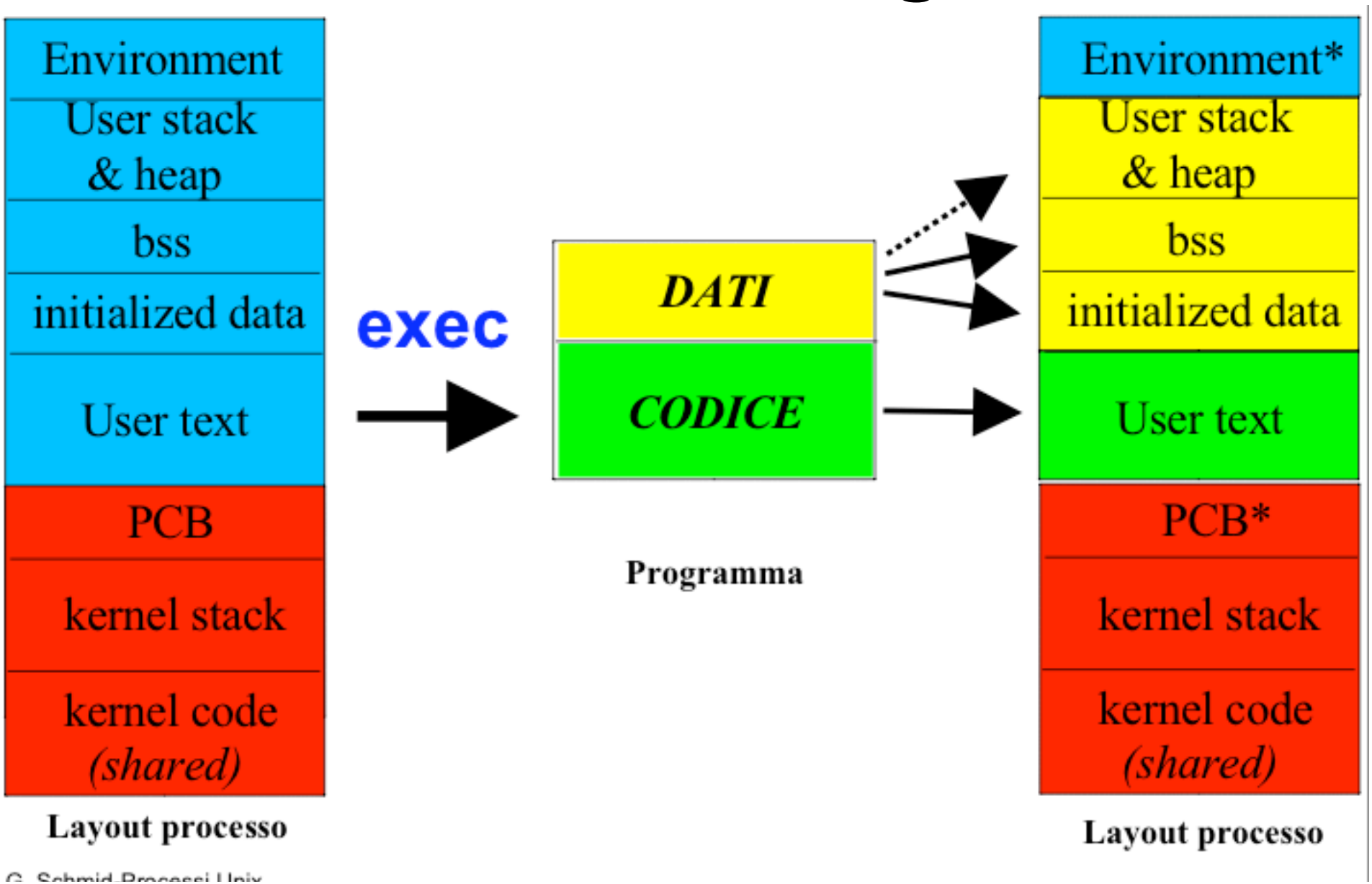
L'esecuzione di un nuovo programma in UNIX si ottiene con la chiamata ad una delle funzioni exec;

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.

Esecuzione di Programmi



Esecuzione di Programmi

Dopo un exec il processo chiamante mantiene inalterate le seguenti proprietà:

- Identificativo di processo (PID);
- Identificativo di gruppo-processi (PGID)
- Credenziali di processo (eccetto, eventualmente, EUID e/o EGID)
- Terminale di controllo
- Directory corrente di lavoro
- Directory root (radice del file system)
- Maschera di creazione dei file
- Lock sui file
- Segnali mandati al processo ma non ancora gestiti
- ...

Famiglia Exec

- La famiglia exec è costituita da sei differenti funzioni, i cui nomi si ottengono dal prefisso exec e da opportune desinenze, che indicano che tipo di argomenti accetta la funzione;
- Le funzioni exec si distinguono per le modalità di individuazione del programma da eseguire (colonna **B**) e di passaggio degli argomenti a tale programma (colonna **A**), nonchè per la possibilità o meno di definire per esso un nuovo ambiente (colonna **C**).

Prefisso	A	B	C	Nome
exec	l			execl
	l	p		execlp
	l		e	execle
	v			execv
	v	p		execvp
	v		e	execve

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

Eseguire Programmi

- Due modi di specificare il programma:
  - pathname: nome esatto, completo di percorso
  - filename: il file sara' cercato nel PATH (**P**ath)

- Due modi di specificare gli argomenti
  - come elenco di parametri formali (**L**ista)
    - terminati da un puntatore nullo (char *) NULL
  - in un array di puntatori a caratteri (**V**ettore)
    - terminato da un puntatore nullo (char *) NULL

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);
```

Esercizio

- Supponiamo che il seguente programma si chiami “a.out”. Qual è il suo output?

```
int main(void)
{
	printf(“ciao!\n);
	execl(“a.out”, “a.out”, NULL);
	return 0;
}
```

Combinazione fork+exec

- Succede spesso che il programma A voglia eseguire il programma B e poi continuare ad elaborare
- Il programma A può creare un figlio che esegue B, mentre il padre aspetta che il figlio termini

Combinazioni fork+exec

```
// codice del programma A

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

if ( pid == 0 )
     // il figlio cerca di eseguire "ls -l"
     if (execl("ls", "ls", "-l", NULL))
          perror("execl"), exit(1);

// il padre aspetta che il figlio termini
wait();
// il padre continua ad elaborare...
```

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

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
...
```