

Laboratorio di sistemi operativi – T4

Controllo dei processi

Il programma

1. Introduzione a UNIX
2. Nozioni integrative del linguaggio C
- 3. controllo dei processi;**
4. pipe e fifo;
5. code di messaggi;
6. memoria condivisa;
7. semafori;
8. segnali;
9. introduzione alla programmazione bash

Controllo dei processi

Process ID e Parent Process ID

- Ogni processo ha un ID associato (PID), un intero positivo che identifica univocamente il processo nel sistema
- la system call `getpid()` restituisce il process ID del processo chiamante

```
#include <unistd.h>

pid_t getpid(void);

//Always successfully returns process ID of caller
```

Process ID e Parent Process ID

- Ogni processo ha un genitore: il processo che lo ha creato. La system call `getppid()` permette ad un processo di conoscere il process ID del proprio padre
- Le relazioni tra processi costituiscono una struttura ad albero. Il genitore di ogni processo ha a sua volta un genitore, fino ad arrivare alla radice: il processo `init`

```
#include <unistd.h>

pid_t getppid(void);

//Always successfully returns process ID of caller
```

Lo stato corrente di un processo

STAT Code	Description
S	Sleeping. Usually waiting for an event to occur, such as a signal or input to become available.
R	Running. Strictly speaking, “runnable,” that is, on the run queue either executing or about to run.
D	Uninterruptible Sleep (Waiting). Usually waiting for input or output to complete.
T	Stopped. Usually stopped by shell job control or the process is under the control of a debugger.
Z	Defunct or “zombie” process.
N	Low priority task, “nice.”
W	Paging. (Not for Linux kernel 2.6 onwards.)
s	Process is a session leader.
+	Process is in the foreground process group.
l	Process is multithreaded.
<	High priority task.

```
ps -aux  
ps -la  
ps -axjf  
top
```

Controllo dei processi

- con controllo dei processi si indica un insieme di operazioni che include la creazione di nuovi processi, l'esecuzione di processi, e la loro terminazione.
- a queste operazioni corrispondono le system call `fork()`, `exit()`, `wait()`, `waitpid()` e `execve()`

fork()

- La syscall `fork()` permette a un processo, il padre, di creare un altro detto figlio.
 - il figlio è (quasi!) una copia esatta del padre: ottiene copie di stack, data, heap, and text segments del padre.
 - Il termine fork deriva dal fatto che ci si può raffigurare il processo padre come un processo che si suddivide in due.

exit(status)

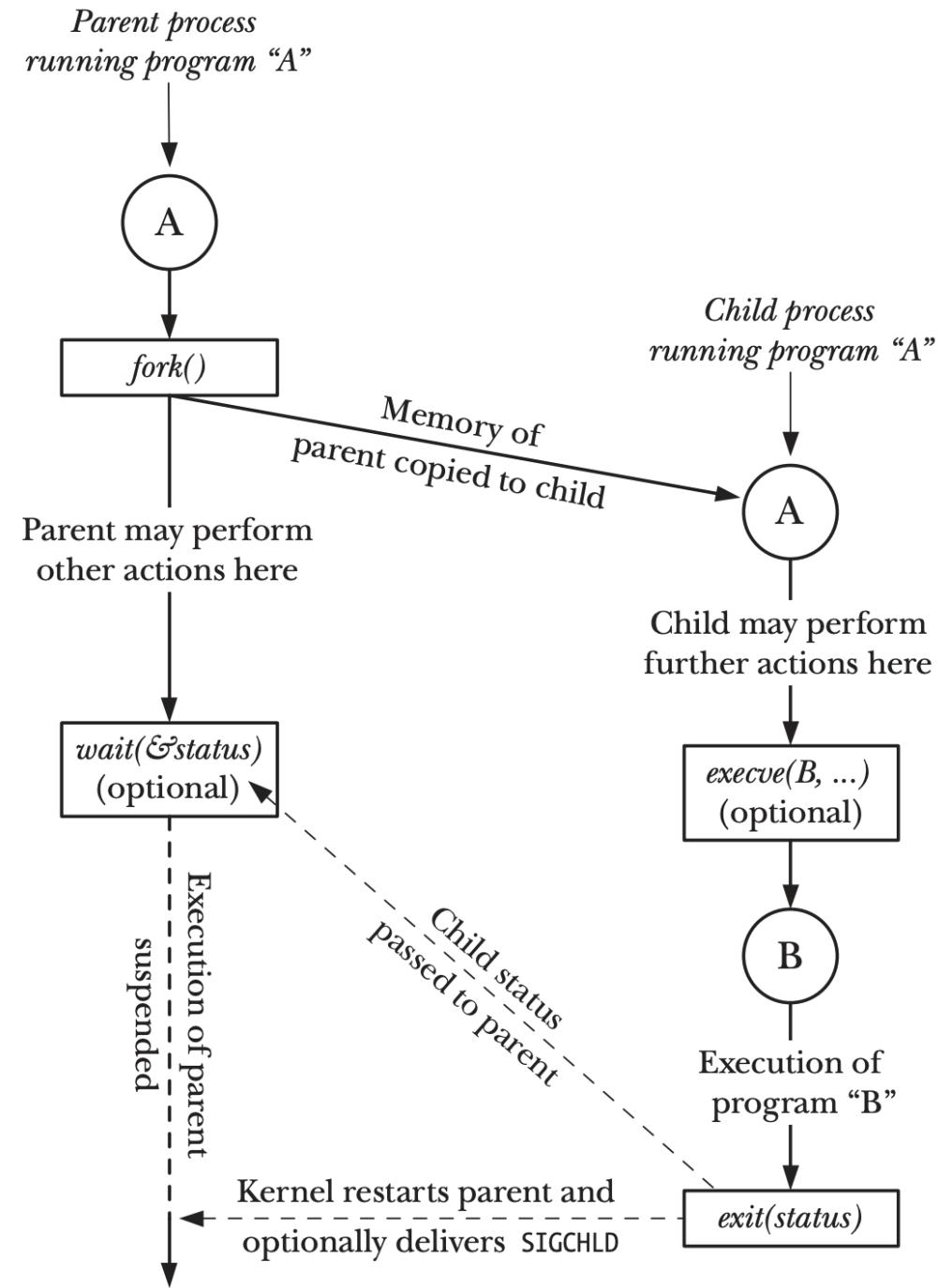
- La funzione di libreria `exit(status)` termina un processo, rendendo le risorse utilizzate dal processo (memoria, descrittori dei file aperti, etc.) nuovamente disponibili per essere allocate dal kernel.
- l'argomento `status` è un intero che descrive lo stato di terminazione del processo: utilizzando la system call `wait()` il processo padre può risalire a tale `status`.

`wait(&status)`

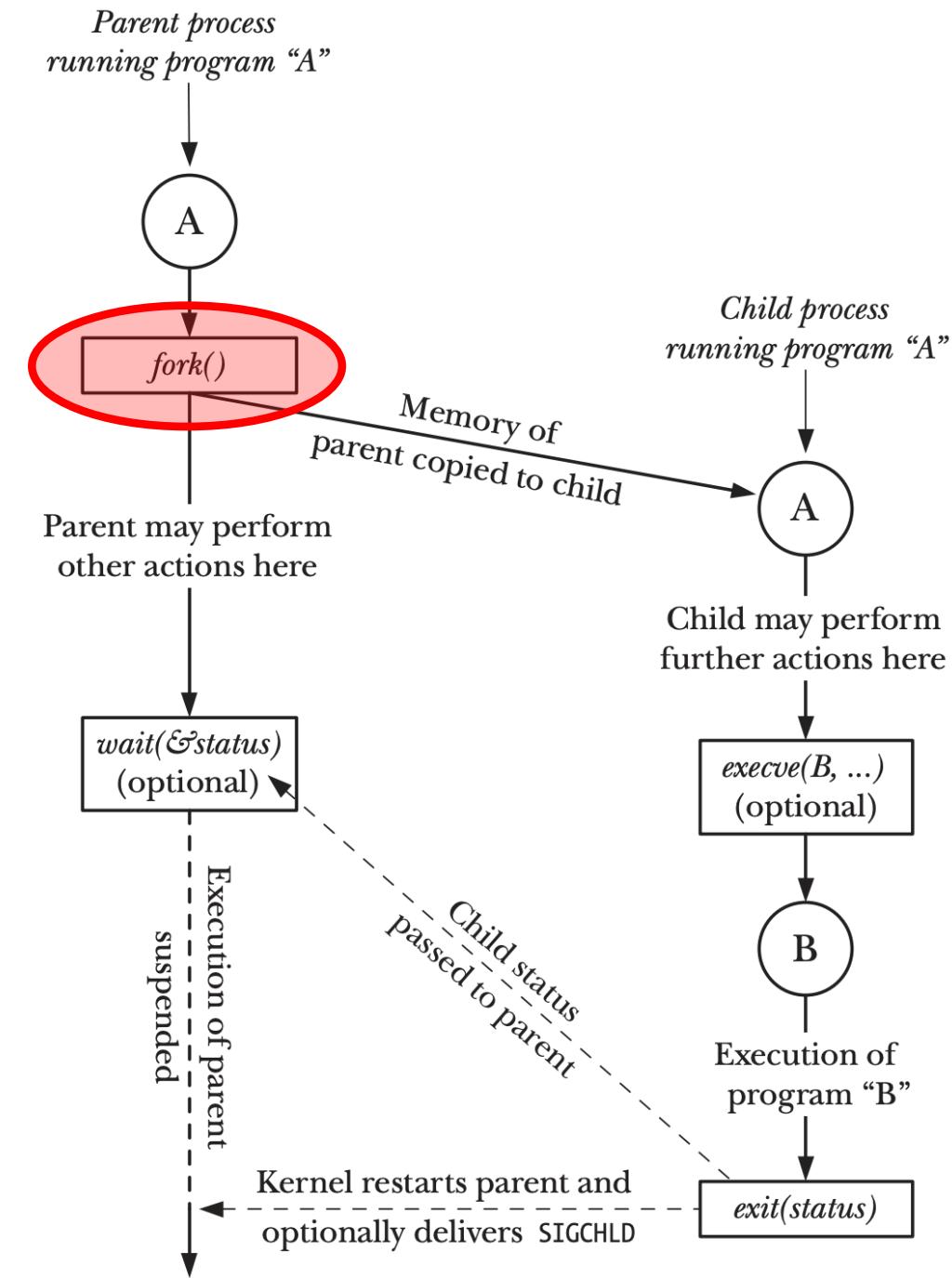
- la system call `wait (&status)` ha due fini:
 - se un figlio non ha ancora concluso la propria esecuzione chiamando una `exit ()`, la `wait ()` sospende l'esecuzione del processo chiamante finché uno dei figli non ha terminato la propria esecuzione.
 - dopo la terminazione del figlio, lo stato di terminazione del figlio è restituito nell'argomento `status` della `wait ()`.

execve(pathname, argv, envp)

- La system call execve (pathname, argv, envp) carica un nuovo programma (pathname, con il relativo argomento list argv, e l'environment envp) nella memoria del processo.
- Il testo del programma precedente è cancellato e stack, dati, e heap sono creati per il nuovo programma.
 - Questa operazione è riferita come "execing" di un nuovo programma. Varie funzioni di libreria utilizzano la syscall execve (), della quale ciascuna costituisce una variazione nell'interfaccia



Creazione dei processi



fork()

- La creazione di processi può essere uno strumento utile per suddividere un compito.
- Per esempio, un server di rete può ascoltare le richieste da parte dei client e creare un nuovo processo figlio per gestire ciascuna richiesta, e nel frattempo continuare a restare in ascolto di ulteriori contatti da parte di altri client.
- La system call `fork()` crea un nuovo processo, il figlio, che è una copia quasi esatta del processo chiamante, il padre.

fork()

```
#include <unistd.h>

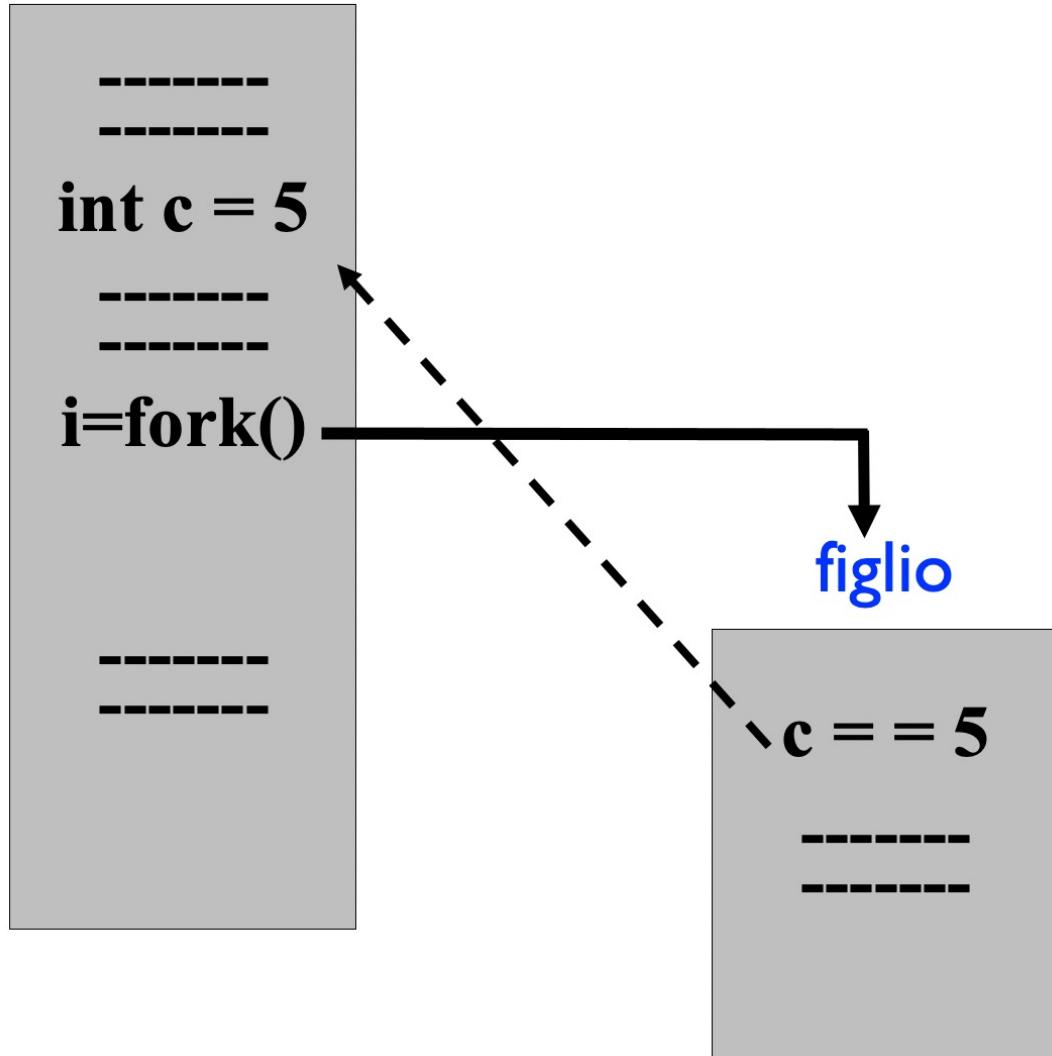
pid_t fork(void);
```

In parent: returns process ID of child on success, or
-1 on error;
in successfully created child: always returns 0

- dopo l'esecuzione della `fork()`, esistono 2 processi e in ciascuno l'esecuzione riprende dal punto in cui la `fork()` restituisce

fork()

padre



- Nello spazio di indirizzamento del processo figlio viene creata una copia delle variabili del padre, col valore loro assegnato al momento della fork.
- Il nuovo processo incomincia l'esecuzione a partire dalla prima istruzione successiva alla fork che lo ha creato.

fork()

- I due processi eseguono lo stesso testo, ma mantenendo **copie distinte** di stack, data, heap, buffer I/O.
 - stack, dati, heap, buffer I/O del figlio sono inizialmente **esatti duplicati** delle corrispondenti parti della memoria del padre.
- Dopo la `fork()`, ogni processo può modificare le variabili in tali segmenti senza influenzare l'altro processo.

Distinguere i processi

- All'interno del codice di un programma possiamo **distinguere** i due processi per mezzo del **valore restituito** dalla `fork()`.
 - Nell'ambiente del padre, `fork()` restituisce il process ID del figlio appena creato. È utile perché il padre può creare -e tenere traccia di- vari figli. Per attenderne la terminazione può usare la `wait()` o altra syscall della stessa famiglia.
 - Nell'ambiente del figlio la `fork()` restituisce 0. Se necessario il figlio può ottenere il proprio process ID con la `getpid()`, e il process ID del padre con la `getppid()`.

Schema di utilizzo della fork()

```
pid_t f; // pid_t è una rinomina del tipo
          // int: signed integer type; da
          // usare con <sys/types.h>

f = fork();
if ( f == -1 ) exit(1);
if ( f ) { // equivale a "if ( f != 0 )"
    ...           // - - - - codice del padre
} else {        // if( f == 0 )
    ...           // - - - - codice del figlio
}
```

Schema tipico di utilizzo della fork()

```
pid_t childPid;

switch (childPid = fork()) {

case -1: /* fork() failed */
    /* --- Handle error --- */

case 0: /* Child of successful fork() comes here */
    /* --- Perform actions specific to child --- */

default: /* Parent comes here after successful fork() */
    /* --- Perform actions specific to parent --- */

}
```

Quale processo sarà eseguito?

- dopo una `fork()`, è indeterminato quale dei due processi sarà scelto per ottenere la CPU.
 - In programmi scritti male, questa indeterminatezza può causare errori
 - Se invece abbiamo bisogno di garantire un particolare ordine di esecuzione, è necessario utilizzare una qualche tecnica di sincronizzazione (ad esempio: i semafori)

03-test-fork-for.c
04-test_fork.c
05a-test-fork-buf.c

```
... // headers
static int idata = 111; // allocata nel segmento dati

int main(int argc, char *argv[]) {
    int istack = 222; // allocata nello stack
    pid_t childPid;

    switch (childPid = fork()) {
        case -1:
            errExit("fork"); // gestione dell'errore
        case 0:
            idata *= 3;
            istack *= 3;
            break;

        default:
            sleep(3); // lasciamo che venga eseguito il figlio
            break;
    }
    // sia il padre sia il figlio eseguono la printf
    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
           (childPid == 0) ? "(child)" : "(parent)", idata,
           istack);
    exit(EXIT_SUCCESS);
}
```

```
$ ./t_fork
```

```
PID=28557 (child) idata=333 istack=666  
PID=28556 (parent) idata=111 istack=222
```

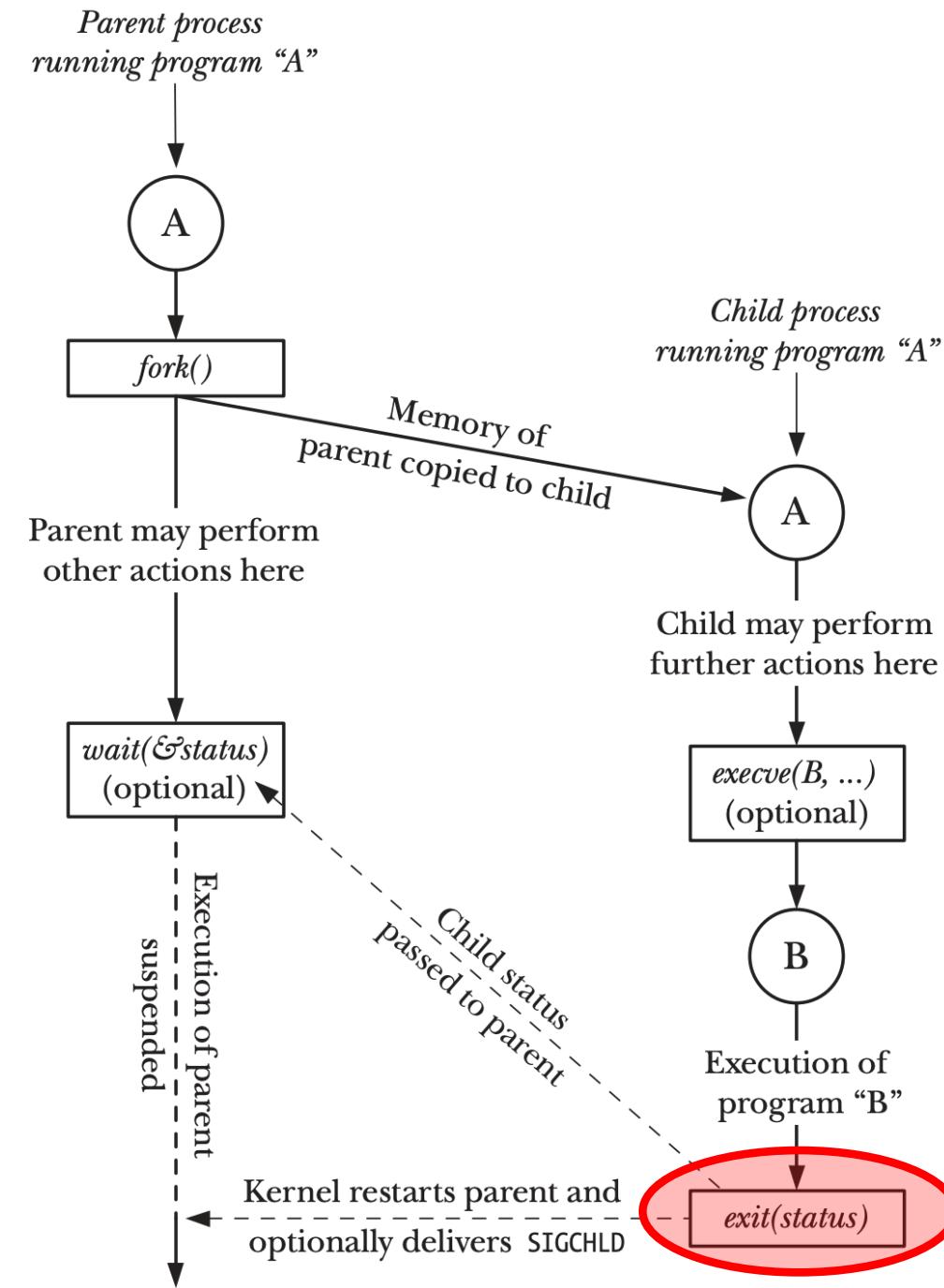
```
// headers  
= 111; // allocata nel segmento dati  
  
int main(int argc, char *argv[]) {  
    int istack = 222; // allocata nello stack  
    pid_t childPid;  
  
    switch (childPid = fork()) {  
        case -1:  
            errExit("fork"); // gestione dell'errore  
        case 0:  
            idata *= 3;  
            istack *= 3;  
            break;  
        default:  
            sleep(3); // lasciamo che venga eseguito il figlio  
            break;  
    }  
    // sia il padre sia il figlio eseguono la printf  
    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),  
          (childPid == 0) ? "(child)" : "(parent)", idata,  
          istack);  
    exit(EXIT_SUCCESS);  
}
```

VEDREMO MODI MIGLIORI DI FARE QUESTO!

File sharing tra padre e figlio

- All'esecuzione della `fork()`, il figlio riceve duplicati di tutti i descrittori di file del padre.
 - Quindi, gli attributi di un file aperto sono condivisi fra genitore e figlio.
 - Per esempio, se il figlio aggiorna l'offset del file, tale modifica è visibile attraverso il corrispondente descrittore nel padre.

Terminazione dei processi



Terminazione di un processo

- Quando un processo termina
 - Le funzioni che sono state registrate come handler con la funzione `atexit()` vengono eseguite in ordine inverso rispetto alla registrazione
 - Tutti gli stream, come ad esempio i files, sono svuotati e chiusi
 - Un segnale di `SIGCHILD` è mandato al processo padre
 - Ogni figlio del processo appena terminato è assegnato ad un nuovo processo padre, il processo `init`
 - Le risorse (memoria, descrittori di file aperti) sono rilasciate
 - Il valore di stato in uscita viene memorizzato (troncato a 8 bit)

Terminazione di un processo

- Un processo può essere terminato da
 - La system call `exit(status)`
 - La ricezione di un segnale, ad esempio inviato premendo CTRL+C. Nel caso succeda questo, le funzioni registrate con `atexit` non sono invocate
- Il valore di stato di ritorno può dare le informazioni sul motivo della terminazione del processo. Il suo valore può essere compreso tra 0 e 255
- Due macro sono definite in `stdlib.h` (e dovrebbero essere usate):
 - `EXIT_SUCCESS` (di solito 0)
 - `EXIT_FAILURE` (di solito 1)

Terminazione di un processo

- Un processo può terminare in due modi principali
 - Una di queste è la terminazione inaspettata, causata dalla ricezione di un segnale, la cui azione di default è di terminare il processo (con o senza la generazione di un file di dump)
 - In alternativa, un processo può terminare normalmente usando la system call `_exit()`

```
#include <unistd.h>

void _exit(int status);
```

System call _exit()

- L'argomento status passato alla funzione `_exit()` definisce lo stato di terminazione del processo, che sarà disponibile al processo padre quando egli chiamerà la funzione `wait()`
- Sebbene sia definito come un intero, solamente gli otto bit più significativi sono usati
 - Per convenzione, un valore di terminazione di 0 indica che il processo è terminato con successo, mentre un valore diverso da 0 una una terminazione inaspettata

La funzione exit()

- Di solito, il codice dei programmi non contengono la chiamata alla funzione `_exit()`, ma contengono invece la chiamata a `exit()`
- Le azioni eseguite dalla `exit()` sono:
 - Gli `exit` handlers (le funzioni registrate con `atexit()` e `on_exit()`) sono invocate;
 - I buffer dello stream `stdio` stream viene svuotato
 - La funzione `_exit()` viene invocata, usando il valore di stato passato come parametro.

```
#include <unistd.h>

void exit(int status);
```

L'handler atexit()

Done with cleaning up
Game over!!

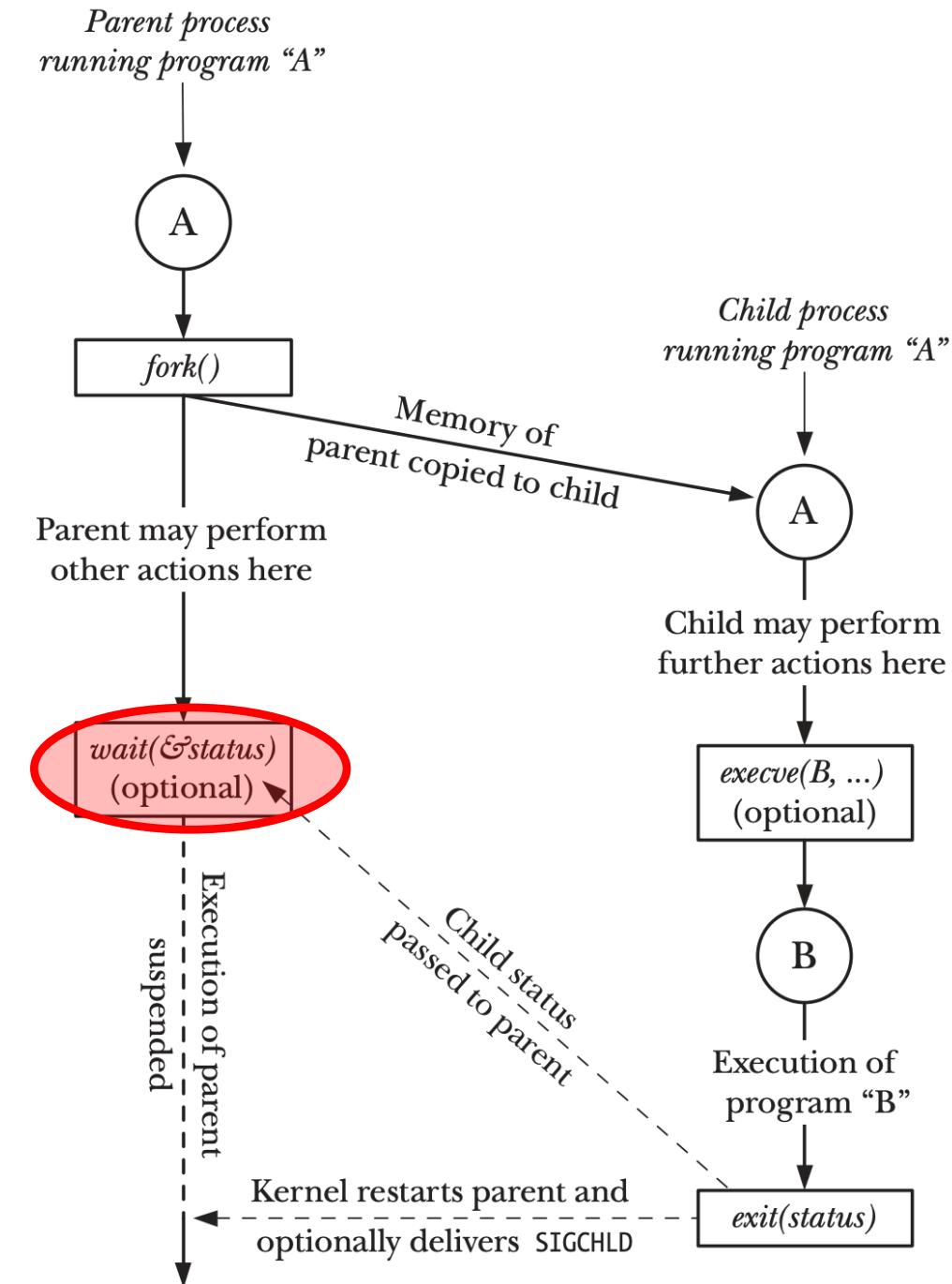
```
#include <stdio.h>
#include <stdlib.h>

void cleanup(void)
{
    /* Cleaning up */
    printf("Done with cleaning up\n");
}

void bye(void)
{
    printf("Game over!!\n");
}

int main (void)
{
    /* My code */
    atexit (bye);
    atexit (cleanup);
    exit (EXIT_SUCCESS);
}
```

Monitoraggio dei processi



La system call wait()

- In molti casi un processo padre deve essere informato quando uno dei figli cambia stato, quando termina o è bloccato da un segnale.
- Con la system call `wait()` un processo genitore attende che uno dei processi figli termini e scrive lo stato di terminazione di quel figlio nel buffer puntato da `status`.

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

pid_t wait(int *status);
```

Returns process ID of terminated child, or -1 on error

La system call wait()

- Se nessun figlio del processo chiamante ha già terminato, la chiamata si blocca finché uno dei figli termina. Se un figlio ha già terminato al momento della chiamata, wait() restituisce immediatamente.
- Se status non è NULL, l'informazione sulla terminazione del figlio è assegnata all'intero cui punta status (NB: status è di tipo int *).
- Cosa restituisce wait(): restituisce il process ID del figlio che ha terminato la propria esecuzione.

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

```
//Returns process ID of terminated child, or -1 on error
```

La system call wait(): errori

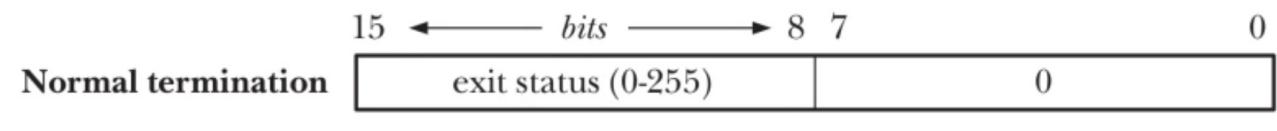
- In caso di errore, `wait()` restituisce `-1`. Un possibile errore è che il processo chiamante potrebbe non avere figli, il che è indicato dal valore `ECHILD` di `errno`.
- possiamo utilizzare questo ciclo per attendere la terminazione di tutti i figli di un processo:

```
while ((childPid = wait(NULL)) != -1)
    continue;

if (errno != ECHILD) // errore inatteso
    errExit("wait"); // gestione errore...
```

La system call wait(): valore di ritorno di un processo figlio

- La funzione wait() ha come parametro un puntatore ad intero
- Il parametro viene usato per comunicare al processo padre lo stato di terminazione del processo figlio (se terminato correttamente)
 - Viene passato il valore indicato nella funzione exit() del figlio (ma con una particolare codifica...)
- La macro WEXITSTATUS(status) viene utilizzata per estrarre il valore di ritorno del processo figlio



```
int status=0;
while ((child_pid = wait(&status)) != -1) {
    printf("Valore di ritorno: %d\n", WEXITSTATUS(status));
}
```

```
#define WEXITSTATUS(x) ((x) >> 8)
```

07-test-wait.c
08-test-wait-error.c
09-test-wait-2.c

Esercizio

- scrivere un programma in cui un padre e un figlio condividono un file aperto: il figlio modifica il file e il padre, dopo avere atteso la terminazione del figlio, stampa a video il contenuto del file.

La system call `waitpid()`

- Se un processo padre ha creato vari figli, non è possibile attendere la terminazione di un particolare figlio con la `wait()`; la `wait()` permette semplicemente di attendere che uno dei figli del chiamante termini.
- Se nessun figlio ha già terminato, la `wait()` si blocca. In alcuni casi è preferibile eseguire una nonblocking wait, in modo da ottenere immediatamente l'informazione che nessun figlio ha ancora terminato la propria esecuzione.
- Con la `wait()` è possibile avere informazioni sui figli che hanno terminato. Non è invece possibile ricevere notifiche quando un figlio è bloccato da un segnale (come `SIGSTOP`) o quando un figlio stopped è risvegliato da un segnale `SIGCONT`.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);

//Returns process ID of child, 0, or -1 on error
```

- L'argomento pid permette di selezionare il figlio da aspettare, secondo queste regole:
 - se `pid > 0`, attendi per il figlio con quel pid.
 - se `pid == 0`, attendi per qualsiasi figlio nello stesso gruppo di processi del chiamante (padre).
 - se `pid < -1`, attendi per qualsiasi figlio il cui process group è uguale al valore assoluto di pid.
 - se `pid == -1`, attendi per un figlio qualsiasi. la chiamata `wait(&status)` è equivalente a `waitpid(-1, &status, 0)`.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);

//Returns process ID of child, 0, or -1 on error
```

- L'argomento options è una bit mask che può includere (in OR) zero o più dei seguenti flag:
 - WUNTRACED: oltre a restituire info quando un figlio termina, restituisci informazioni quando il figlio viene bloccato da un segnale.
 - WCONTINUED: restituisci informazioni anche nel caso il figlio sia stopped e venga risvegliato da un segnale SIGCONT.
 - WNOHANG: se nessun figlio specificato da pid ha cambiato stato, restituisci immediatamente, invece di bloccare il chiamante. In questo caso, il valore di ritorno di `waitpid()` è 0. Se il processo chiamante non ha figli con il pid richiesto, `waitpid()` fallisce con l'errore ECHILD.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);

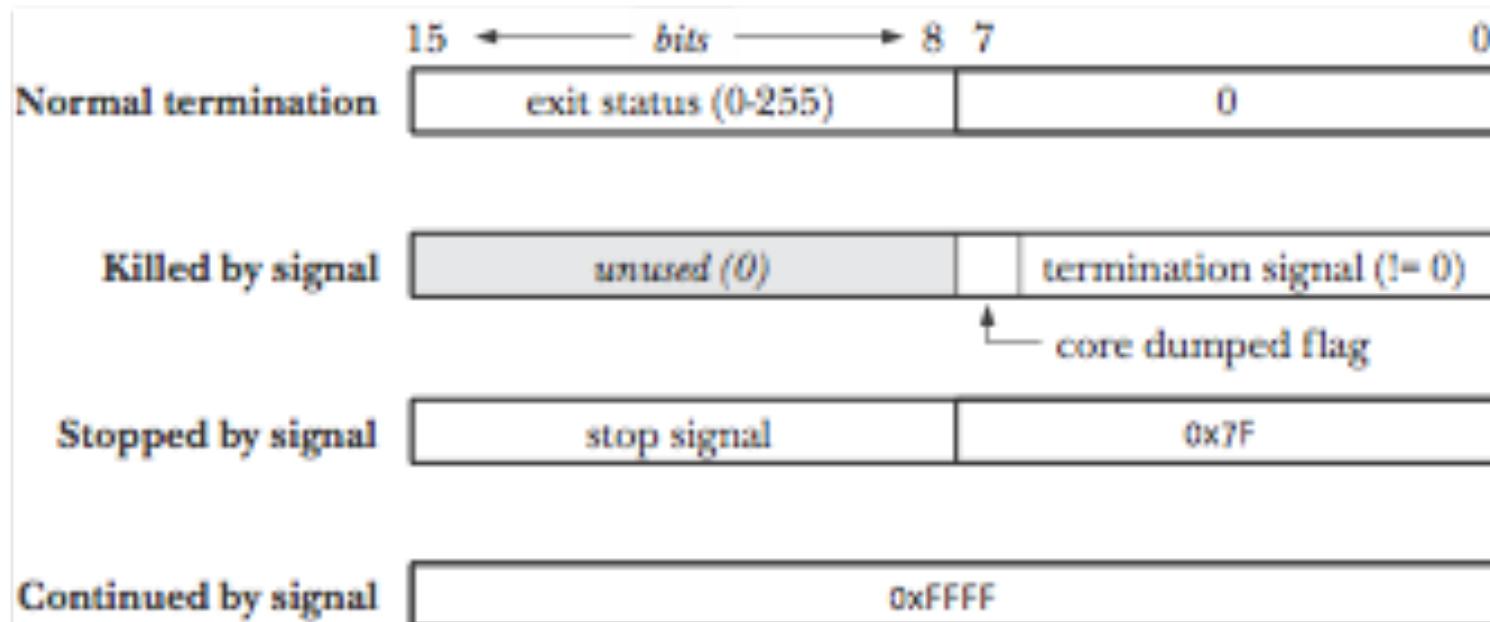
//Returns process ID of child, 0, or -1 on error
```

- Il valore status restituito da `wait()` e `waitpid()` ci consente di distinguere fra i seguenti eventi per il figlio:
 - il figlio ha terminato l'esecuzione chiamando `_exit()` (o `exit()`), specificando un codice d'uscita (exit status) intero.
 - il figlio ha terminato l'esecuzione per la ricezione di un segnale non gestito.
 - il figlio è stato bloccato da un segnale, e `waitpid()` è stata chiamata con il flag `WUNTRACED`.
 - Il figlio ha ripreso l'esecuzione per un segnale `SIGCONT`, e `waitpid()` è stata chiamata con il flag `WCONTINUED`.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);

//Returns process ID of child, 0, or -1 on error
```

- Sebbene sia definito come **int**, solo gli ultimi 2 byte del valore puntato da **status** sono effettivamente utilizzati. Il modo in cui questi 2 byte sono scritti dipende da quale è evento è occorso per il figlio



```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);

//Returns process ID of child, 0, or -1 on error
```

- L'header file `<sys/wait.h>` definisce un insieme standard di macro che possono essere utilizzate per interpretare un wait status.
- Applicate allo status restituito da `wait()` o `waitpid()`, solo una delle seguenti macro restituirà true.
 - `WIFEXITED(status)`. Restituisce true se il processo figlio è terminato normalmente. In questo caso la macro `WEXITSTATUS(status)` restituisce l'exit status del processo figlio.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);

//Returns process ID of child, 0, or -1 on error
```

- WIFSIGNALED(status). Restituisce true se il figlio è stato ucciso da un segnale. In questo caso, la macro WTERMSIG(status) restituisce il numero del segnale che ha causato la terminazione del processo.
- WIFSTOPPED(status). Restituisce true se il figlio è stato bloccato da un segnale. In questo caso, la macro WSTOPSIG(status) restituisce il numero del segnale che ha bloccato il processo.
- WIFCONTINUED(status). Restituisce true se il figlio è stato risvegliato da un segnale SIGCONT.

11a-test_WONOHANG.c
11-test_wstatus_value.c

Esercizio

- Sia NUM_KIDS una macro definita con #define (di valore 20, per esempio). Si scriva un programma in cui il processo padre genera NUM_KIDS processi figli. Ogni processo figlio genera casualmente un numero intero n da 1 a 6, stampa il suo PID e ne esce con exit status uguale al numero casuale estratto (man 3 rand per la generazione di numeri interi casuali. Si usi anche srand(getpid()) per l'inizializzazione del seed del random). Il processo padre attende la terminazione di tutti i processi figli (con waitpid) e stampa la somma dei valori di uscita dei propri figli.

Processi orfani

- In generale o il padre sopravvive al figlio, o viceversa.
 - Chi diventa il padre di un processo figlio orfano? il figlio orfano è adottato da init, il progenitore di tutti i processi, il cui process ID è 1.
 - In altre parole, dopo che il genitore di un processo figlio termina, una chiamata a getppid() restituirà il valore 1 (l'id dipende dal SO).
 - può essere utile per capire se il vero padre di un processo figlio è ancora vivo.

Processi zombie

- Cosa capita a un figlio che termina prima che il padre abbia avuto modo di eseguire una wait()?
 - Sebbene il figlio abbia terminato, il padre dovrebbe poter avere la possibilità di eseguire una wait() in un momento successivo per determinare come è terminato il figlio.
 - il kernel garantisce questa possibilità trasformando il figlio in uno zombie. Gran parte delle risorse gestite da un figlio sono rilasciate al sistema per essere assegnate ad altri processi.
 - L'unica parte del processo che resta è un'entry nella tabella dei processi che registra il process ID del figlio, il termination status, e le statistiche sull'utilizzo delle risorse.

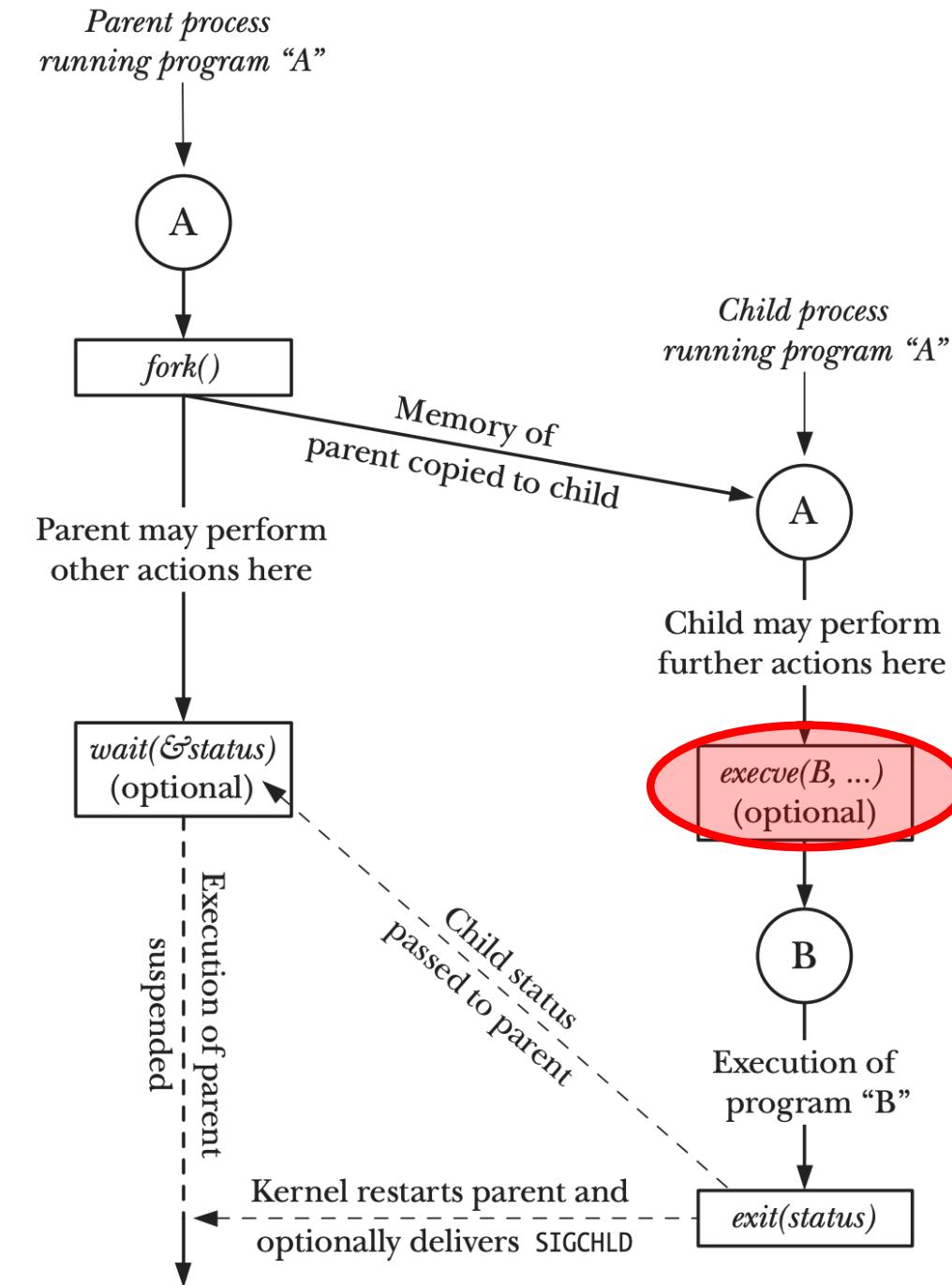
Processi zombie

- Un processo zombie non può essere ucciso da un segnale, neppure SIGKILL. Questo assicura che il genitore possa sempre eventualmente eseguire una `wait()`.
 - Quando il padre esegue una `wait()`, il kernel rimuove lo zombie, dal momento che l'ultima informazione sul figlio è stata fornita all'interessato.
- Se il genitore termina senza fare la `wait()`, il processo `init` adotta il figlio ed esegue automaticamente una `wait()`, rimuovendo dal sistema il processo zombie.

Processi zombie

- Se un genitore crea un figlio, ma fallisce la relativa `wait()`, un elemento relativo allo zombie sarà mantenuto indefinitamente nella tabella dei processi del kernel.
 - Se il numero degli zombie cresce eccessivamente, gli zombie possono riempire la tabella dei processi, e questo impedirebbe la creazione di altri processi.
- Poiché gli zombie non possono essere uccisi da un segnale, l'unico modo per rimuoverli dal sistema è uccidere il loro padre (o attendere la sua terminazione). A quel momento gli zombi possono essere adottati da `init` e rimossi.

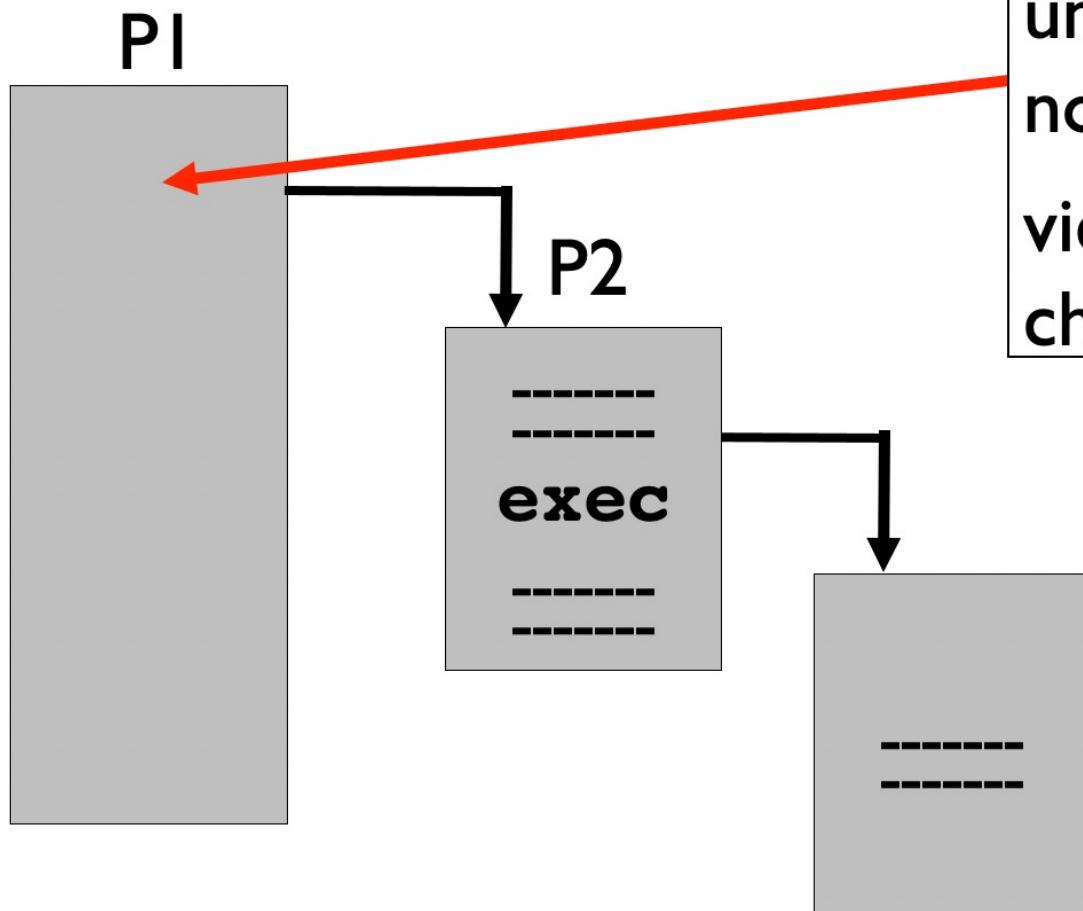
Esecuzione di programmi



System call execve()

- la system call execve() carica un nuovo programma nella memoria di un processo.
- con questa operazione, il vecchio programma è abbandonato, e lo stack, i dati, e lo heap del processo sono sostituiti da quelli del nuovo programma.
 - dopo avere eseguito l'inizializzazione del codice, il nuovo programma inizia l'esecuzione dalla propria funzione main().
- varie funzioni di libreria, tutte con nomi che iniziano con exec, sono basate sulla system call execve().
 - ciascuna di queste funzioni fornisce una diversa interfaccia alla stessa funzionalità.

System call execve()



il codice che viene dopo
una exec in un programma
non verrà mai eseguito!
viene eseguito solo se la
chiamata alla exec fallisce...

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[],
            char *const envp[]);
//Never returns on success; returns -1 on error
```

- L'argomento pathname contiene il pathname del programma che sarà caricato nella memoria del processo.
- L'argomento argv specifica gli argomenti della linea di comando da passare al nuovo programma. Si tratta di una lista di puntatori a stringa, terminati da puntatore a NULL.
- Il valore fornito per argv[0] corrisponde al nome del comando. Tipicamente, questo valore è lo stesso del basename (i.e., l'ultimo elemento) del pathname.
- L'ultimo argomento, envp, specifica la lista environment list per il nuovo programma. L'argomento envp corrisponde all'array environ; è una lista di puntatori a stringhe (terminata da ptr a NULL) nella forma name=value.

Valori di ritorno di execve

- Poiché sostituisce il programma che la ha chiamata, una chiamata di `execve()` che va a buon fine non restituisce. Non abbiamo quindi bisogno di controllare il valore di ritorno di `execve()`; sarà sempre `-1`.
- Il fatto che abbia restituito un qualche valore ci informa che è occorso un errore, e come sempre è possibile utilizzare `errno` per determinarne la causa.

Condizioni di errore di execve

- Fra gli errori che possono essere restituiti in errno:
 - EACCES. l'argomento pathname non si riferisce a un file normale, il file non è un eseguibile, o una delle componenti del pathname non è ricercabile (i.e., sono negati i permessi di esecuzione sulla directory).
 - ENOENT. Il file riferito dal pathname non esiste.
 - ENOEXEC. Il file riferito dal pathname è marcato come un eseguibile ma non è riconosciuto come in un formato effettivamente eseguibile.
 - ETXTBSY. Il file riferito dal pathname è aperto in scrittura da un altro processo.
 - E2BIG. Lo spazio complessivo richiesto dalla lista degli argomenti e dalla lista dell'ambiente supera la massima dimensione consentita.

Per conoscere gli errori (reminder)

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

...
// --- stampa l'elenco degli errori noti sul sistema
int idx = 0; // numero errore

for( idx = 0; idx < sys_nerr; idx++ )
    printf( "Error #%-3d: %s\n", idx, strerror( idx ) );
```

Un esempio d'uso di execve

```
...
int main(int argc, char *argv[]) {
    int i;
    char *argVec[VEC_SIZE] = {"buongiorno", "ciao",
                             "cordiali saluti", "all the best", NULL};
    char *envVec[VEC_SIZE] = {"silvia", "paolo",
                             "mario", "carla", NULL};

    switch(fork()) {
        case -1:
            fprintf(stderr, "fork fallita\n");
            exit(0);

        case 0:
            printf("PID(figlio): %d\n", getpid());
            execve(argv[1], argVec, envVec);
            exit(EXIT_FAILURE);
    }
}
```

Alternative a execve

```
#include <unistd.h>
int execle(const char *pathname, const char *arg, ...
           /* , (char *) NULL, char *const envp[] */);
int execlp(const char *filename, const char *arg, ...
           /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
           /* , (char *) NULL */);

//None of the above returns on success; all return -1 on error
```

```
int execlp(const char *filename, const char *arg, ...  
           /* , (char *) NULL */);  
int execvp(const char *filename, char *const argv[]);
```

- Gran parte delle funzioni exec() si aspettano un pathname per specificare il nuovo programma da caricare.
- Invece, execlp() e execvp() ci consentono di specificare solo il filename. Il filename è cercato nella lista di directory specificata dalla variabile d'ambiente PATH.
- La variabile d'ambiente PATH non è usata se il filename contiene uno slash (/), nel qual caso è trattato come un percorso relativo o assoluto.

Esercizio

- scrivere un programma in cui viene eseguita una fork().
 - il processo figlio esegue una execlp() chiamando un secondo programma ('saluta_persone.c') e un certo numero di nomi propri di persona ('mario', 'ada', etc.) che stampi sullo schermo questi nomi.
- riscrivere il programma precedente (modificando opportunamente anche saluta_persone.c) eseguendo questa volta execvp() invece di execlp().

16-fork-execlp.c
16-fork-execvp.c

```
int execle(const char *pathname, const char *arg, ...
           /* , (char *) NULL, char *const envp[] */);
int execlp(const char *filename, const char *arg, ...
           /* , (char *) NULL */);
int execl(const char *pathname, const char *arg, ...
           /* , (char *) NULL */);
```

- Invece di utilizzare un array per specificare la lista argv per il nuovo programma, execle(), execlp(), e execl() richiedono al programmatore di specificare gli argomenti come una lista di stringhe.
- La lista di argomenti deve essere terminata da un puntatore a NULL come terminatore della lista. Questo formato è indicato dal (char *) NULL commentato nei prototipi riportati sopra.
 - I nomi delle funzioni che richiedono la lista di argomenti come un array (execve(), execvp(), and execv()) contengono la lettera v (da vector).

```
int execle(const char *pathname, const char *arg, ...
           /* , (char *) NULL, char *const envp[] */ );
```

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

- Le funzioni execle() e execve() permettono al programmatore di specificare esplicitamente l'environment per il nuovo programma, utilizzando envp, un array di puntatori a stringhe terminato dal puntatore a NULL.
 - I nomi di queste funzioni terminano con la lettera e (da environment) per indicare questa caratteristica.

File descriptors e exec()

- Per default, tutti i descrittori di file aperti da un programma che chiama exec() restano aperti attraverso la exec() e sono pertanto disponibili per il nuovo programma.
- Questo è spesso utile, poiché il programma chiamante può aprire file con particolari descrittori, e questi file sono automaticamente disponibili per il nuovo programma, senza che questo debba sapere i loro nomi e/o aprirli

Eseguire un comando con system()

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

- La funzione system() permette di chiamare un programma per eseguire un comando di shell arbitrario.
- La funzione system() crea un processo figlio che invoca una shell per eseguire il comando command. Esempio di chiamata di system():
- `system("ls -lt | wc -l");`

Il valore di ritorno di system()

- Valore di ritorno di system():
 - se command è un NULL pointer, system() restituisce un valore diverso da 0 se una shell è disponibile, e 0 se nessuna shell è disponibile.
 - se non è stato possibile creare un processo figlio o il suo stato di terminazione non è stato ricevuto, system() restituisce -1.
 - se non è stato possibile eseguire la shell nel processo figlio, system() restituisce un valore come se la shell del processo figlio avesse terminato con la chiamata _exit(127).
 - se tutte le system calls hanno avuto successo, system() restituisce lo stato di terminazione della shell figlia utilizzata per eseguire il comando: lo status di terminazione di una shell è lo status di terminazione dell'ultimo comando eseguito.

```
int main(int argc, char *argv[] ) {
    char str[MAX_CMD_LEN];
    int status;
    for (;;) {
        printf("Command: ");
        fflush(stdout);
        if (fgets(str, MAX_CMD_LEN, stdin) == NULL)
            break;
        status = system(str);
        printf("system() returned: status=0x%04x\n",
               (unsigned int) status);
        if (status == -1)
            errExit("system");
        else {
            if(WIFEXITED(status) && WEXITSTATUS(status) == 127)
                printf("(Probably) could not invoke shell\n");
            else // la shell ha eseguito il comando correttam.
                print_wait_status(NULL, status);
        }
    }
    exit(EXIT_SUCCESS);
}
```

Eseguire un comando con system()

- Il vantaggio principale offerto da system() è la semplicità d'uso
 - non dobbiamo gestire i dettagli relativi alle chiamate di fork(), exec(), wait(), e exit().
 - la gestione degli errori e dei segnali è affidata a system() per nostro conto.
 - poiché system() utilizza la shell per eseguire un comando, il processamento legato alle sostituzioni, redirezioni è effettuato sul comando prima che esso sia eseguito.

Eseguire un comando con system()

- il principale costo della system() è l'inefficienza.
 - Eseguire un comando usando system() richiede la creazione di almeno 2 processi (uno per la shell e uno o più per i comandi eseguiti), ciascuno dei quali esegue una exec().
 - Se l'efficienza o la velocità sono richiesti, è preferibile utilizzare chiamate fork() e exec() per eseguire il programma desiderato.