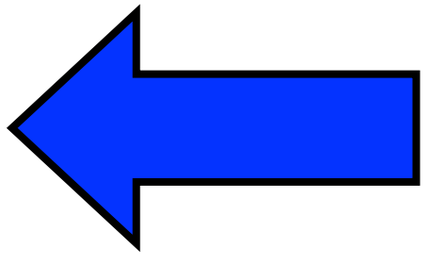
laboratorio di sistemi operativi

il controllo dei processi

Materiale preparato da Daniele Radicioni

argomenti del laboratorio UNIX

1. introduzione a UNIX;
2. integrazione C: operatori bitwise, precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
3. controllo dei processi;
4. segnali;
5. pipe e fifo;
6. code di messaggi;
7. memoria condivisa;
8. semafori;
9. introduzione alla programmazione bash.



- il materiale di queste lezioni è tratto da:
  - lucidi del Prof. Gunetti;
  - Michael Kerrisk, ***The Linux Programming interface - a Linux and UNIX® System Programming Handbook***, No Starch Press, San Francisco, CA, 2010;
  - W. Richard Stevens, Stephen A. Rago, ***Advanced Programming in the UNIX® Environment*** (2nd Edition), Addison-Wesley, 2005;

Process ID and Parent Process ID

- each process has a process ID (PID), a positive integer that uniquely identifies the process on the system
  - process IDs are used and returned by a variety of system calls

- the ***getpid()*** system call returns the process ID of the calling process

```
#include <unistd.h>

pid_t getpid(void);

        Always successfully returns process ID of caller
```

Process ID and Parent Process ID

- Each process has a parent: the process that created it. a process can find out the process ID of its parent using the ***getppid()*** system call

- the parent process ID attribute of each process represents the tree-like relationship of all processes on the system. the parent of each process has its own parent, and so on, going all the way back to process 1, ***init***, the ancestor of all processes

```
#include <unistd.h>

pid_t getppid(void);

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

Memory Layout of a Process

- The memory allocated to each process is composed of a number of parts, usually referred to as segments.
  - The text segment contains the machine-language instructions of the program run by the process.
    - It is ***read-only*** so that a process doesn't accidentally modify its own instructions via a bad pointer value.
    - ***sharable*** so that a single copy of the program code can be mapped into the virtual address space of all of the processes.

Memory Layout of a Process

- The initialized data segment contains global and static variables that are explicitly initialized. The values of these variables are read from the executable file when the program is loaded into memory.
- The uninitialized data segment contains global and static variables that are not explicitly initialized. Before starting the program, the system initializes all memory in this segment to 0.

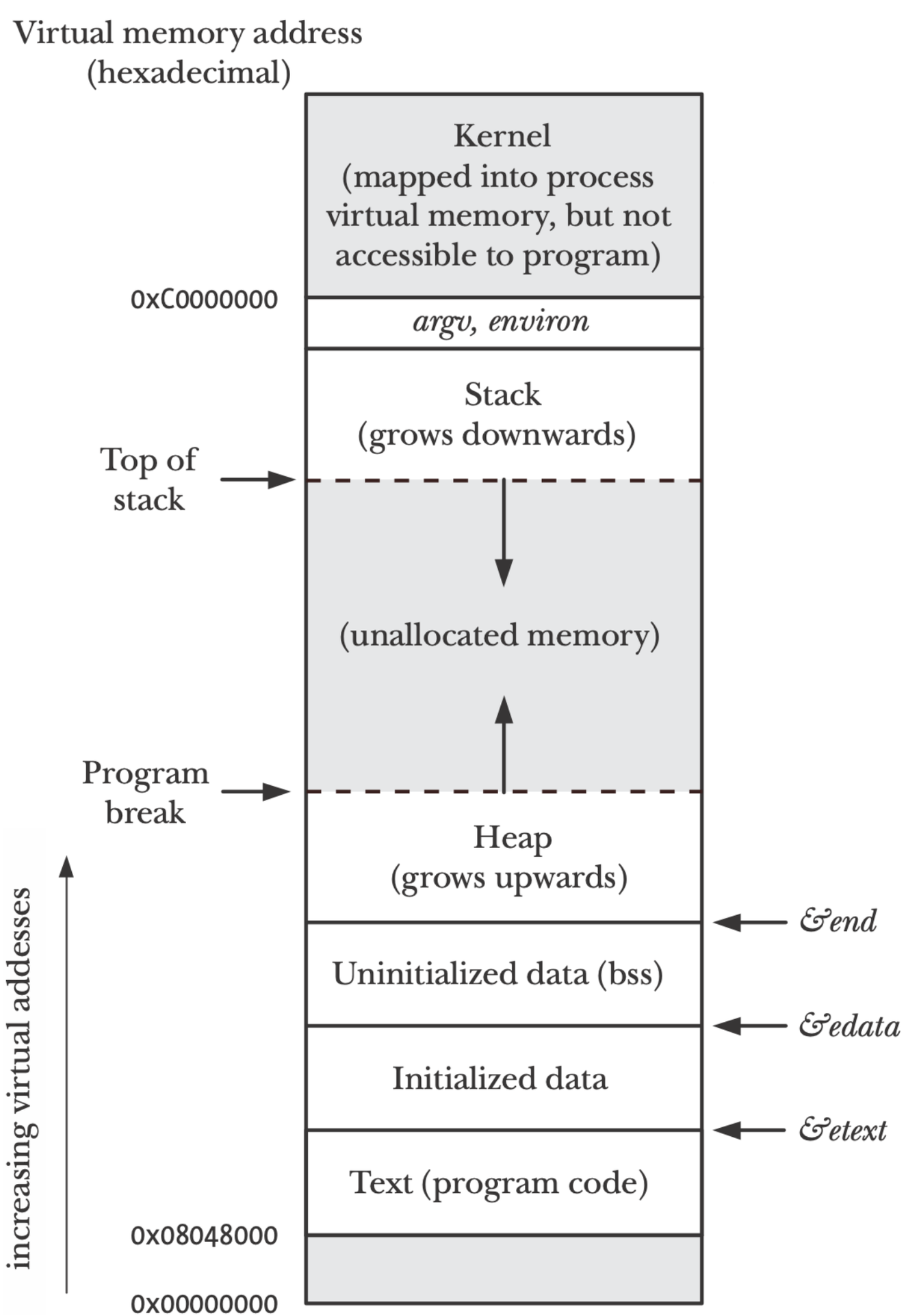
• The main reason for placing global and static variables that are initialized into a separate segment from those that are uninitialized is that, when a program is stored on disk, it is not necessary to allocate space for the uninitialized data.

Memory Layout of a Process

- The stack is a dynamically growing and shrinking segment containing stack frames.
  - One stack frame is allocated for each currently called function. A frame stores the function's local variables (so-called automatic variables), arguments, and return value.
- The heap is an area from which memory (for variables) can be dynamically allocated at run time.

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

char char_buf[65536];// segmento dei dati non inizializzati
int numeri_primi[] = { 2, 3, 5, 7 };// dati inizializzati

static int square(int x) { // allocato nel frame di square
    int result;   // allocato nel frame di square()
    result = x * x;
    return result;  // valore di ritorno
}

static void compute(int val) { // allocato nel frame di compute()
    printf("il quadrato di %d e` %d\n", val, square(val));

    if (val < 1000) {
        int t;    // allocato nel frame di compute()
        t = val * val * val;
        printf("il cubo di %d e` %d\n", val, t);
    }
}
...
```

```c
int main(int argc, char *argv[]) { // nel frame del main()

    static int key = 1234; // segmento dei dati inizializzati
    static char mbuf[10610000]; // segm. dati non inizializzati
    char *p;   // allocato nel frame del main()

    p = malloc(1024);  // allocato nello heap

    compute(key);
    exit(EXIT_SUCCESS);
}
```



controllo dei processi

process control

- 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()***, and ***execve()***.
  - A system call is a request for the operating system to do something on behalf of the user's program.
  - The system calls are functions used in the kernel itself. To the programmer it appears as a normal C function call.

fork()

- La syscall ***fork()*** permette a un processo, il ***padre***, di crearne un altro detto ***figlio.***
  - il figlio è (quasi!) una copia esatta del padre: ottiene copie degli ***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.

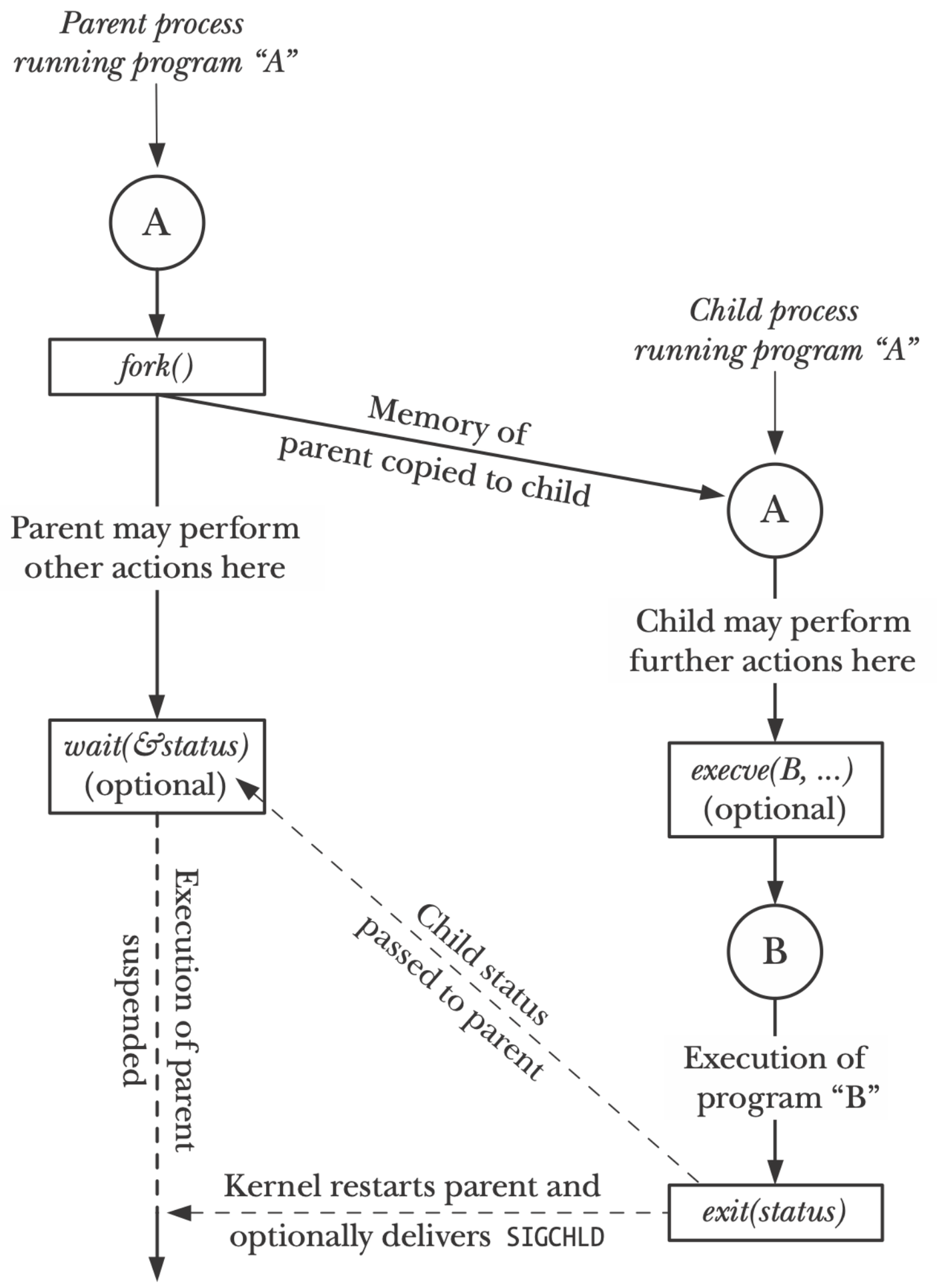
wait(&status)

- la system call ***wait(&status)*** ha due fini:
  - se un figlio non ha ancora concluso la propria esecuzione chiamando la ***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().***

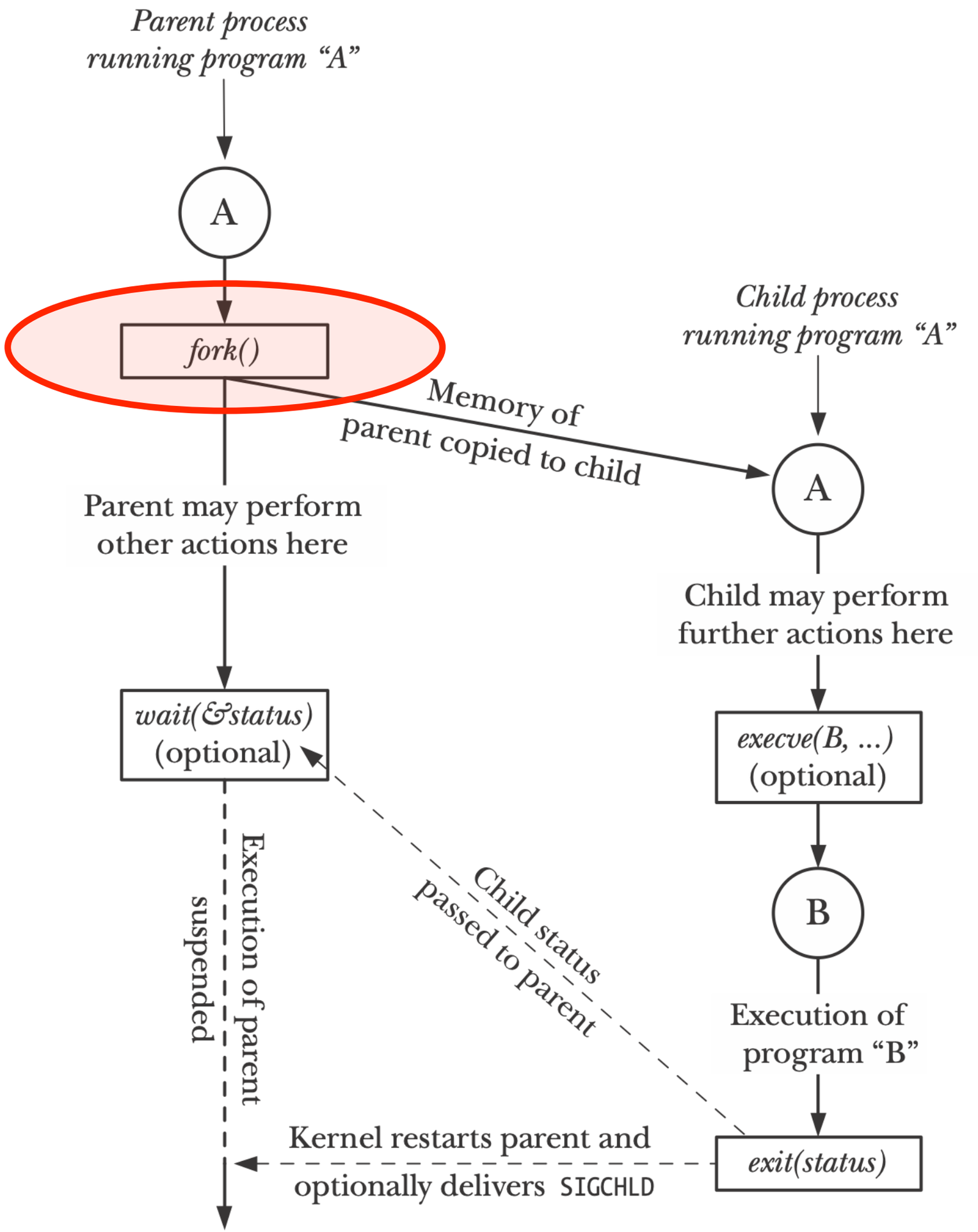
execve(pathname, argv, envp)

- 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 di processi


Parent process
running program "A"
A
fork()
Memory of
parent copied to child
Child process
running program "A"
A
Parent may perform
other actions here
Child may perform
further actions here
wait(&status)
(optional)
execve(B, ...)
(optional)
Execution of parent
suspended
Child status
passed to parent
B
Execution of
program "B"
Kernel restarts parent and
optionally delivers SIGCHLD
exit(status)

fork()

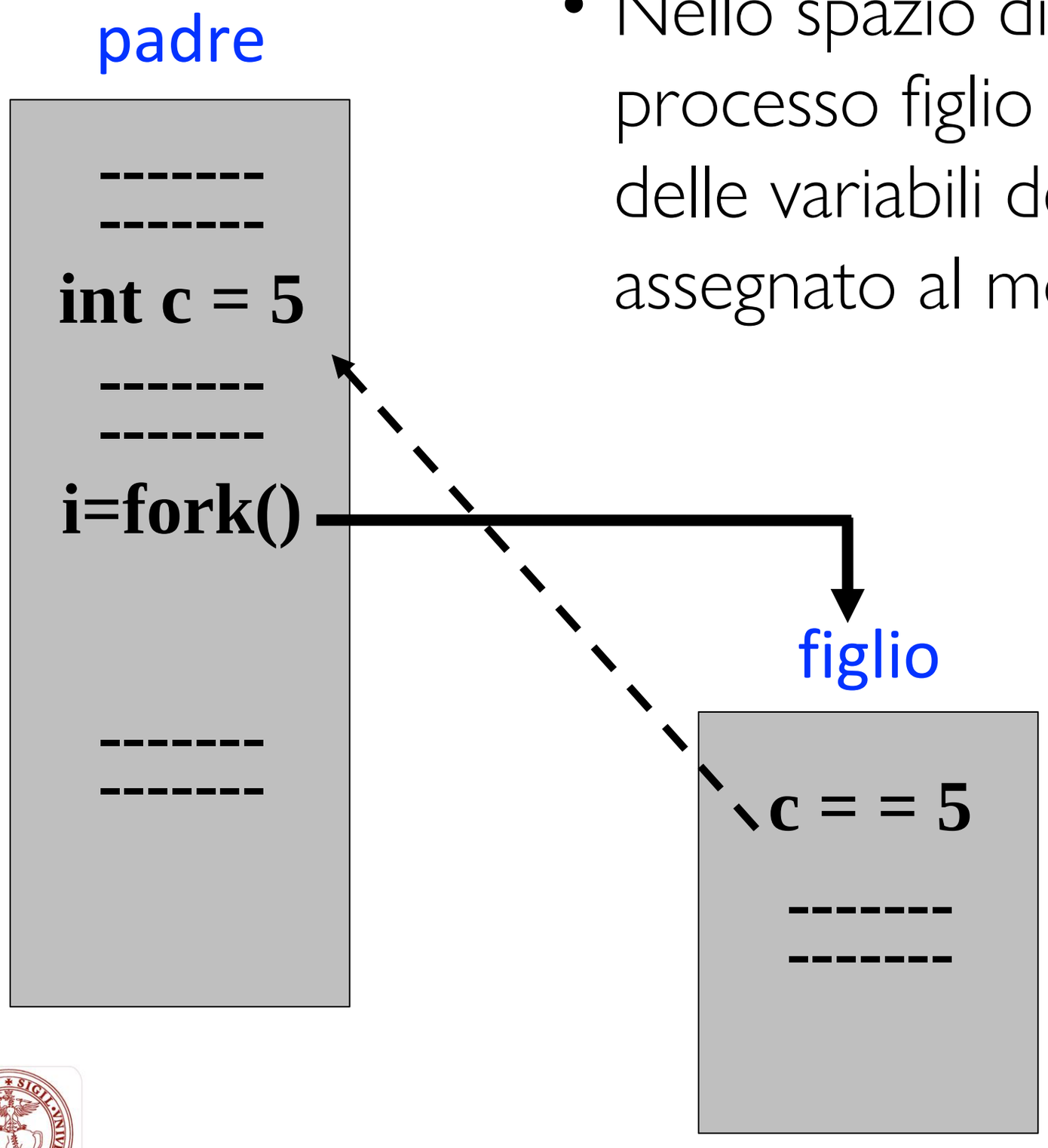
- La creazione di processi può essere uno strumento utile per suddividere un compito.
  - Per esempio, un server di rete può ascoltare le richieste da parte dei client e creare un nuovo processo figlio per gestire ciascuna richiesta, e nel frattempo continuare a restare in ascolto di ulteriori contatti 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.



- 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***, e ***heap***.
  - stack, dati, e heap 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 tipico di utilizzo della *fork()*

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

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

schema tipico di utilizzo della *fork()*

```
pid_t procPid;

switch (procPid = 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 noti come ***race conditions***.
  - Se invece abbiamo bisogno di garantire un particolare ordine di esecuzione, è necessario utilizzare una qualche tecnica di sincronizzazione.

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

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

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

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

```
$ ./t_fork
PID=28557 (child) idata=333 istack=666
PID=28556 (parent) idata=111 istack=222
```

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

    switch (procPid = fork()) {
    case -1:
        errExit("fork");

    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(),
            (procPid == 0) ? "(child) " : "(parent)", idata,
            istack);
    exit(EXIT_SUCCESS);
}
```

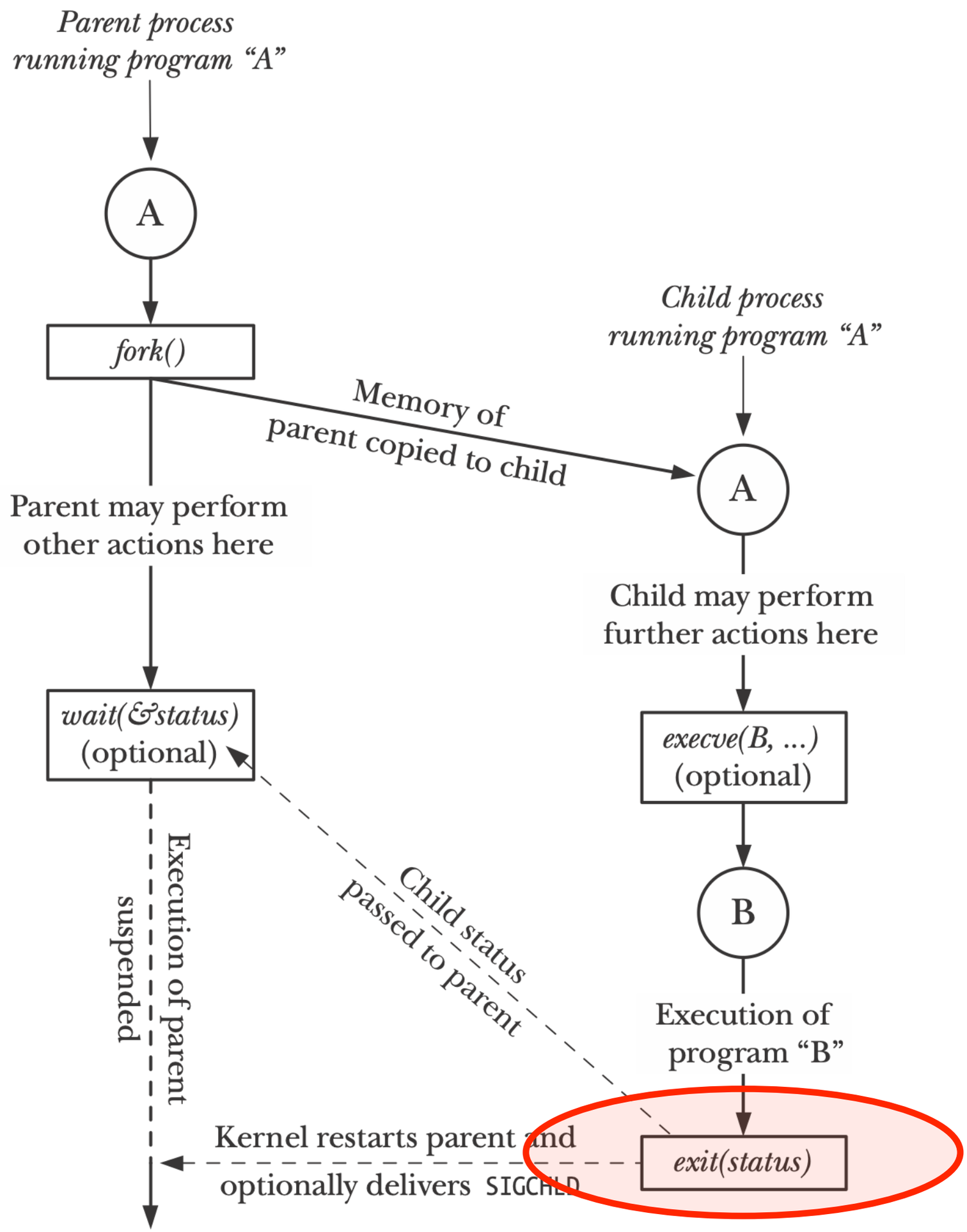
File Sharing Between Parent and Child

- 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.

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.

terminazione di processi


Parent process
running program "A"
A
fork()
Memory of
parent copied to child
Child process
running program "A"
A
Parent may perform
other actions here
Child may perform
further actions here
wait(&status)
(optional)
execve(B, ...)
(optional)
Child status
passed to parent
Execution of parent
suspended
B
Execution of
program "B"
Kernel restarts parent and
optionally delivers SIGCHLD
exit(status)

Terminating a Process

- A process may terminate in two general ways.
  - One of these is ***abnormal*** termination, caused by the delivery of a signal whose default action is to terminate the process (with or without a core dump).
  - Alternatively, a process can terminate normally, using the ***_exit()*** system call.

```
#include <unistd.h>

void _exit(int status);
```

system call *_exit()*

- The status argument given to ***_exit()*** defines the termination status of the process, which is available to the parent of this process when it calls ***wait()***.
- Although defined as an ***int***, only the bottom 8 bits of status are actually made available to the parent.
  - By convention, a termination status of 0 indicates that a process completed successfully, and a nonzero status value indicates that the process terminated unsuccessfully.

la funzione *exit()*

- Programs generally don't call ***_exit()*** directly, but instead call the ***exit()*** library function, which performs various actions before calling ***_exit()***.

- The following actions are performed by ***exit()***:
  - Exit handlers (functions registered with ***atexit()*** and ***on_exit()***) are called;
  - The ***stdio*** stream buffers are flushed.
  - The ***_exit()*** system call is invoked, using the value supplied in ***status***.

```
#include <unistd.h>

void exit(int status);
```

Interactions with *fork()*, *stdio* Buffers and *_exit()*

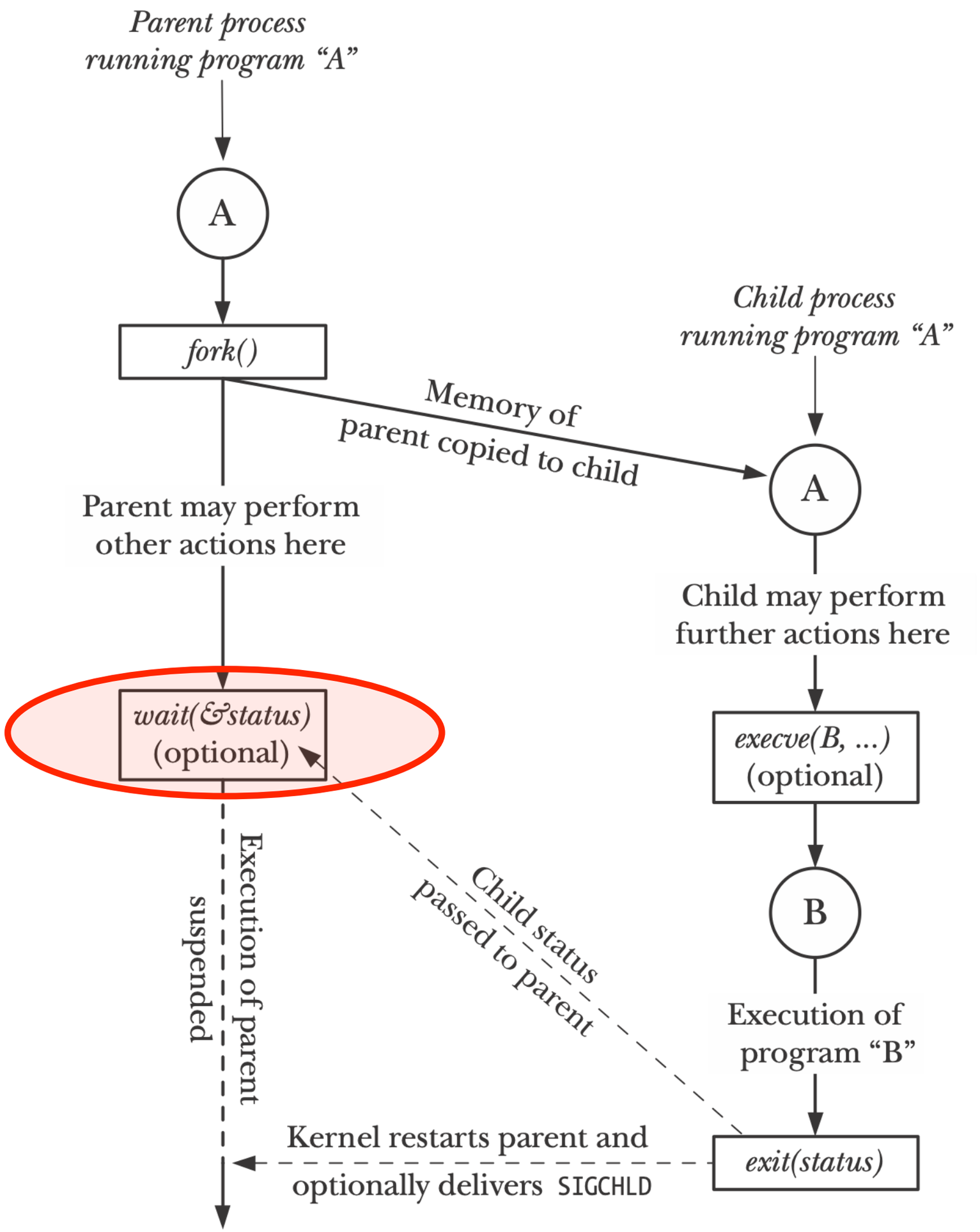
```
int main(int argc, char *argv[]) {
    printf("Hello world\n");
    write(STDOUT_FILENO, "Ciao\n", 5);

    if (fork() == -1)
        errExit("fork");

    // sia padre sia figlio eseguono la exit
    exit(EXIT_SUCCESS);
}
```

esercizio

- compilare ed eseguire il programma nel lucido precedente provando una prima volta ad eseguirlo con output diretto sul terminale, e redirigendo l'output su file.

- verificare se vi sono differenze e ragionare sulle possibili cause e soluzioni.
  - suggerimento: consultare il manuale per ***printf()*** e per ***write()***, con 'man -s 2 write'
  - suggerimento: studiare il funzionamento delle funzioni ***fflush()*** e ***setbuf()***.

monitoraggio dei processi


Parent process
running program "A"
A
fork()
Memory of
parent copied to child
Child process
running program "A"
A
Parent may perform
other actions here
Child may perform
further actions here
wait(&status)
(optional)
execve(B, ...)
(optional)
B
Execution of parent
suspended
Child status
passed to parent
Execution of
program "B"
Kernel restarts parent and
optionally delivers SIGCHLD
exit(status)

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

pid_t wait(int *status);

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

- 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.

The wait() System Call

1. 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.
2. Se ***status*** non è ***NULL***, l'informazione sulla terminazione del figlio è assegnata all'intero cui punta ***status*** (NB: ***status*** è di tipo ***int ****).
3. Cosa restituisce ***wait()***: ***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
```

error

- 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...
```

esercizio

- scrivere un programma che prende in input un numero variabile di interi i_1, i_2, … i_n.

  - il programma genera ***n*** figli: ogni figlio si mette in attesa (usare il comando ***sleep***) il primo per i_1 secondi, il secondo per i_2 etc., e termina.

  - il genitore aspetta la terminazione di tutti i propri figli e termina a sua volta.

  - predisporre un insieme di stampe a video per provare che vengano eseguite tutte le operazioni richieste.

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 ***waitpid(–1, &status, 0)*** è equivalente a ***wait(&status)***.

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

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

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

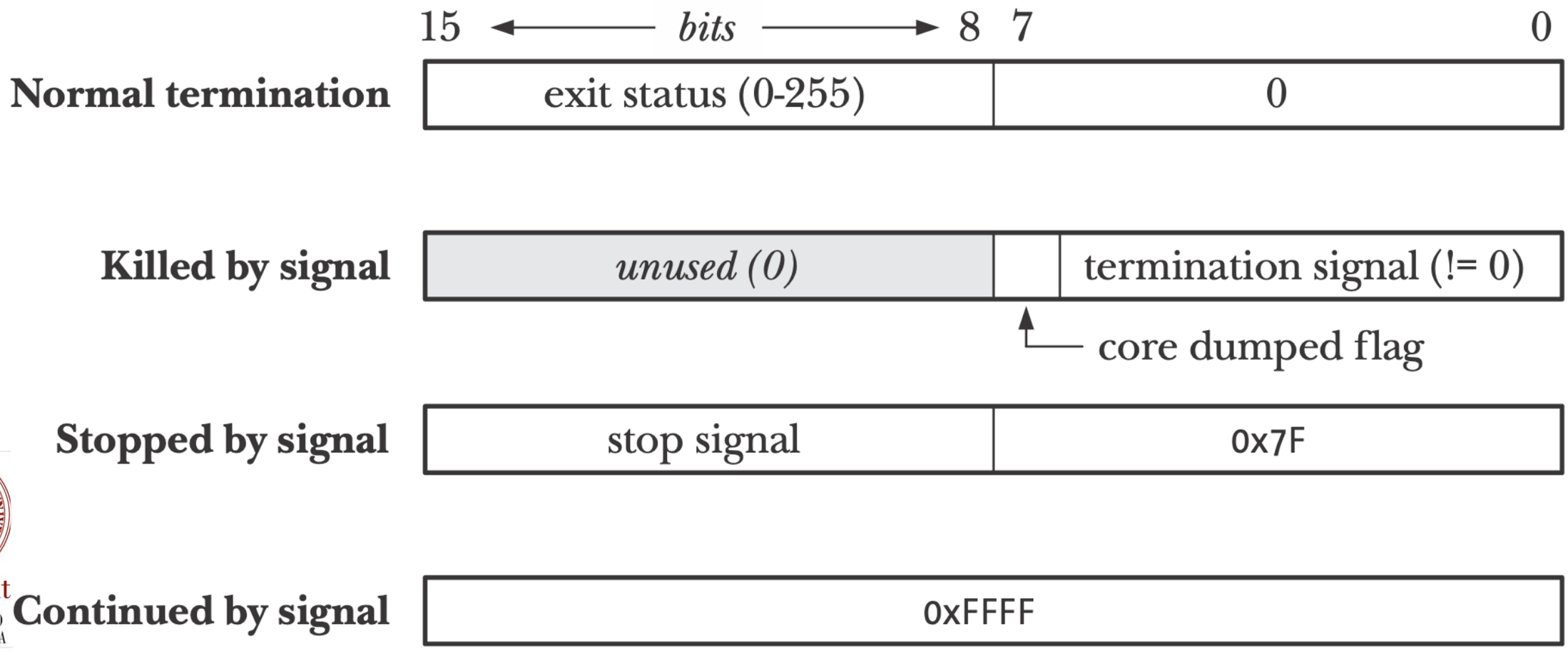
- L'argomento ***options*** è una bit mask che può includere (in OR) zero o più dei seguenti flag:
  - *WUNTRACED*: oltre a restituire info quando un figlio termina, restituisci informazioni quando il figlio viene bloccato da un segnale.
  - *WCONTINUED*: restituisci informazioni anche nel caso il figlio 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***.

Wait Status Value

- 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***.

Wait Status Value

- 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



Wait Status Value

- 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

```
if (WIFEXITED(status)) {
    printf("child exited, status=%d\n", WEXITSTATUS(status));
}
```

Wait Status Value

- ***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***.

orphans and zombies

- In generale o il padre sopravvive al figlio, o viceversa.
  1. 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*.
    - può essere utile per capire se il vero padre di un processo figlio è ancora vivo.

orphans and zombies

2. 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.

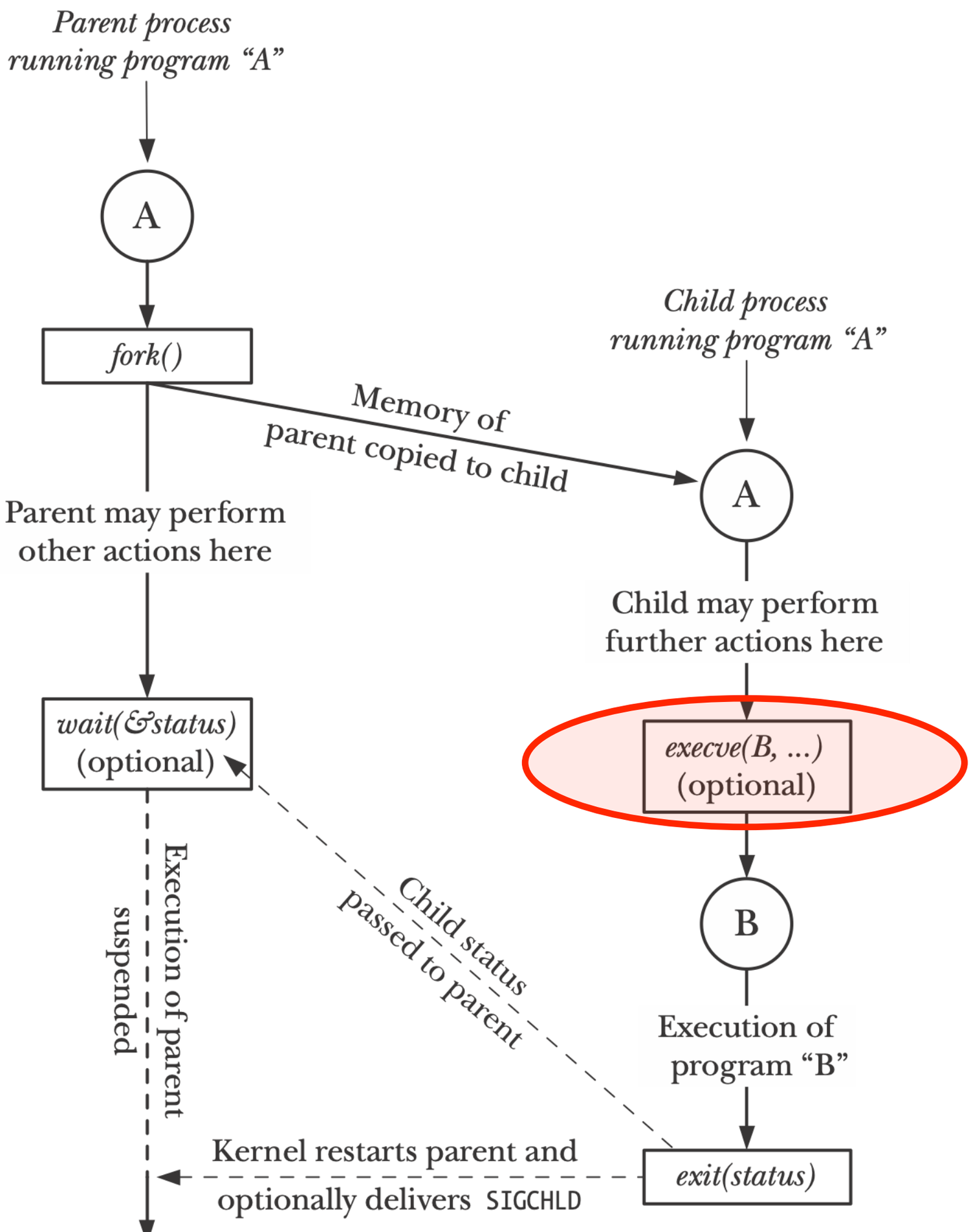
zombies

- 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.

zombies

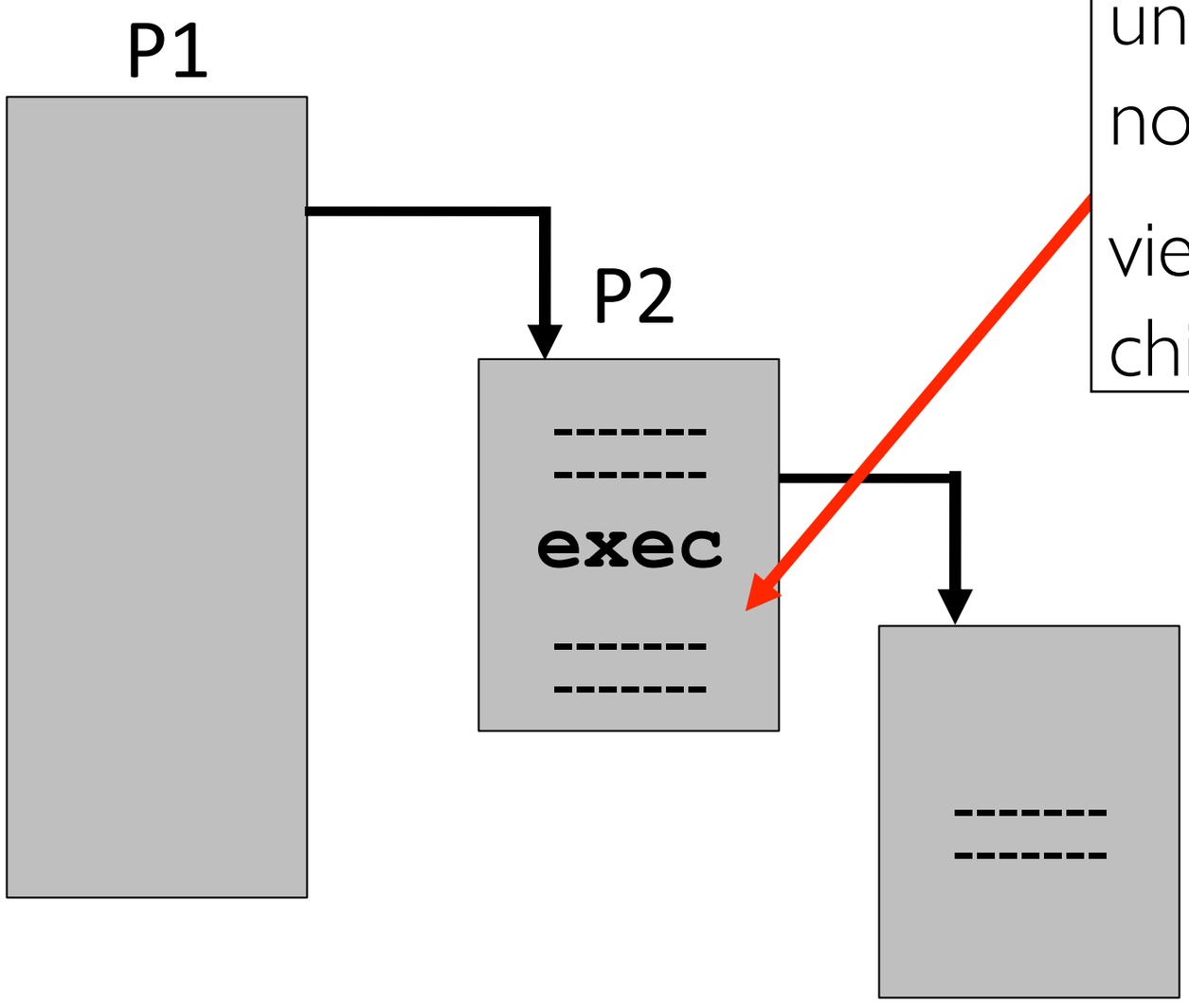
- Se un genitore crea un figlio, ma fallisce la relativa ***wait()***, un elemento relativo allo zombie sarà mantenuto indefinitamente nella tabella dei processi del kernel.
  - Se il numero degli zombie cresce eccessivamente, gli zombie possono riempire la tabella dei processi, e questo impedirebbe 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 i***nit*** e rimossi.

esecuzione di programmi


Parent process
running program "A"
A
fork()
Memory of
parent copied to child
Child process
running program "A"
A
Parent may perform
other actions here
Child may perform
further actions here
wait(&status)
(optional)
execve(B, ...)
(optional)
Execution of parent
suspended
Child status
passed to parent
B
Execution of
program "B"
Kernel restarts parent and
optionally delivers SIGCHLD
exit(status)

Executing a New Program: ***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.



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 puntatore a ***NULL***) nella forma ***name=value***.

valori di ritorno

- 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

- 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...

```
#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 ) );
```

da dove viene strerror()? cercare sul manuale….

esempio d'uso

```
...
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);
```

esempi

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

   default:
   printf("PID(padre): %d\n", getpid());
   printf(" --- padre dorme per 3 secondi --- \n");
   for (i = 0; i < 3; i++)  {
      sleep(1);
      printf("*\n");
   }
   printf(" --- terminazione padre --- \n");
   exit(EXIT_SUCCESS);
  }
}
```

```
#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
```

- Esistono alcune funzioni di libreria che forniscono API alternative per eseguire una ***exec()***. Tutte queste funzioni utilizzano la ***execve()***, e differiscono le une dalle altre e dalla ***execve()*** per il modo in cui sono specificati il nome del programma, la lista degli argomenti e l'ambiente del nuovo programma.

```
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()***.

```
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.

esercizio

- scrivere 2 programmi che implementino le funzionalità della seguente istruzione:

```
cp file1.txt file2.txt
```

- il primo programma esegue una fork, e il figlio chiama ***exec*** mandando in esecuzione il secondo (prodotto dal sorgente ***copia.c***).

Eseguire un comando di shell: *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.

```c
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);
}
```

```c
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:
            (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);
}
```

WIFEXITED(status)
Evaluates to a non-zero value if status was returned for a child process that terminated normally.

WEXITSTATUS(status)
If the value of WIFEXITED(status) is non-zero, this macro evaluates to the low-order 8 bits of the status argument that the child process passed to _exit() or exit(), or the value the child process returned from main().

Eseguire un comando di shell: *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 di shell: 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.

implementazione di *system()*

- l'opzione ***-c*** del comando ***sh*** fornisce un modo semplice per eseguire una stringa che contiene comandi di shell arbitrari:

```
$ sh -c "ls | wc"
      38      38     444
```

- per re-implementare ***system()***, dobbiamo utilizzare una ***fork()*** per creare un figlio che faccia una ***execl()*** con gli argomenti corrispondenti al comando ***sh***:

execl("/bin/sh", "sh", "-c", command, (char *) NULL);

```c
/* implementazione comando system */
int system_reimplemented(char *command) {
  int status;
  pid_t childPid;

  switch (childPid = fork()) {

  case -1: /* Error */
  return -1;

  case 0: /* Child */
  execl("/bin/sh", "sh", "-c", command, (char *) NULL);
  _exit(127); /* Failed exec */
  default: /* Parent */
  if (waitpid(childPid, &status, 0) == -1)
    return -1;
  else
    return status;
  }
}
```

```
int main(int argc, char** argv){
  int status = -1;
  char command[CMDSIZE];

  if(fgets(command, CMDSIZE, stdin)==NULL)
    exit(EXIT_FAILURE);

  status = system_reimplemented(command);
  printf("status: %d\n", status);

  exit(EXIT_SUCCESS);
}
```

```
    _exit(127); /* Failed exec */
  default: /* Parent */
  if (waitpid(childPid, &status, 0) == -1)
    return -1;
  else
    return status;
  }
}
```

quanti processi genera?

```
for (i=0; i<3; i++)
    fork();
```