

PROCESSI IN C ➤

Ogni processo ha un Id detto PID rappresentato come intero non negativo.

Ogni processo ha anche l'ID del processo padre, entrambi richiamabili tramite `<unistd.h>`:

```
#include <unistd.h>
pid_t getpid(void);
    Always successfully returns process ID of caller
```

Id Processo

```
#include <unistd.h>
pid_t getppid(void);
    Always successfully returns process ID
    of parent of caller
```

Id Parent Processo

LAYOUT DI MEMORIA DI UN PROCESSO ➤

Ogni processo ha un'area di memoria allocata chiamata segment.

Esso è formato da:

- Text Segment;

Area immutabile e condivisibile in cui sono contenute le istruzioni in linguaggio macchina

- Initialized Data Segment:

Contiene le variabili statiche e globali inizializzate

- Uninitialized Data Segment

Contiene le variabili statiche e globali non inizializzate

Le 2 aree sono divise per evitare di allocare mem. per le non inizializzate.

- Stack :

Contiene stack frames. Ogni frame contiene funzioni, variabili locali e argomenti.

- Heap :

Contiene area di mem allocabile a runtime.

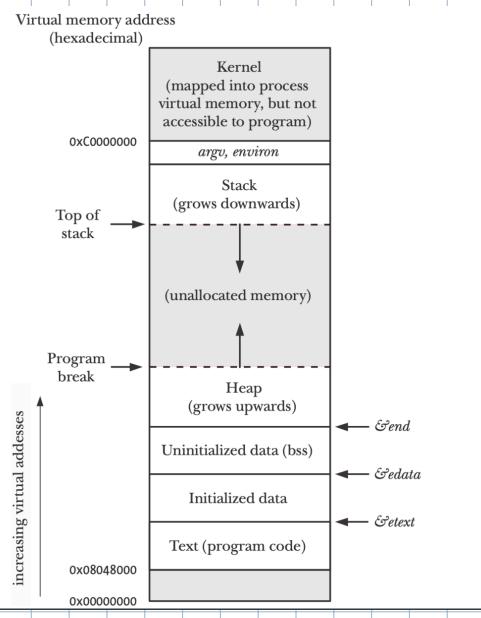
```
#include <stdio.h>
#include <stdlib.h>

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);
    }
}
```



CONTROLLO DEI PROCESSI ➤

È possibile gestire i processi attraverso le system call:

- fork:

Per creare un "processo figlio". Il nuovo processo erediterà copie ^{distinte} di Data, Heap, Stack e Text

- exit (status)

Termina il processo corrente liberando la memoria.

L'argomento "status" rappresenta lo stato di terminazione del processo.

- wait (&status)

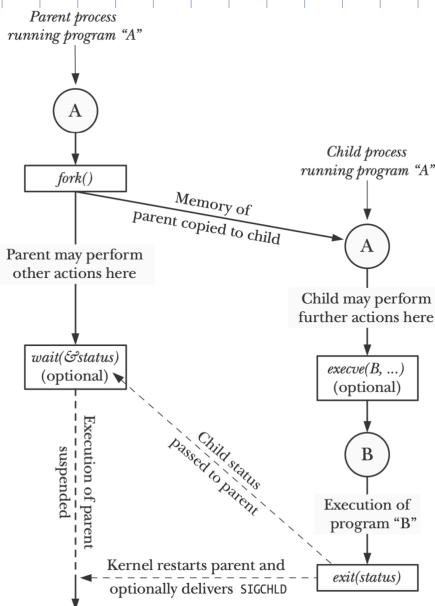
- Se processo figlio non ha ancora terminato, allora sospende l'esecuzione del chiamante fino alla terminazione

- Dopo che il figlio termina, lo status del figlio viene restituito in status.

- execve (pathname, argv, envp)

Carica un nuovo programma localizzato col pathname in memoria passandogli gli argomenti e l'environment.

Il text segment del programma prec. viene cancellato e Data, Heap e Stack sono allocati



Fork NEL PRATICO

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.

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
}
```

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;
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 --- */
}
```

A volte si rende necessario garantire sequenzialità tra i processi

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.

```
static int idata = 111; // headers
static int istack = 222; // 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=%d %d %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[])
{
    pid_t procPid;
    int idata = 111;
    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=%d %d %d\n", (long) getpid(),
        (procPid == 0) ? "(child)" : "(parent)", idata,
        istack);
    exit(EXIT_SUCCESS);
}
```

Siccome Data, Stack e Heap sono copiati, i puntatori a file puntano allo stesso file, consentendo sharing

TERMINAZIONE NELL'PRATICO >

Il processo si termina con la system call `_exit(status)` anche se solitamente si usa `exit(status)` perché effettua altre operazioni aggiuntive:

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);
```

46

WAIT NELL'PRATICO >

La `wait` termina non appena il processo figlio termina, altrimenti, rimane in attesa del 1°.

Restituisce

- il PID del processo terminato e lo status se la `wait` termina correttamente
- 1 in caso di errore.

Se il chiamante non ha figli, allora questa info sarà disponibile su `errno`

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...
```

54

WAIT PCD

Istruzione che consente di aspettare processi filtrandoli

```
#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)**.



Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

57

```
#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, restituisce informazioni quando il figlio viene bloccato da un segnale.
 - **WCONTINUED**: restituisce informazioni anche nel caso il figlio sia stopped e venga risvegliato da un segnale **SIGCONT**.
 - **WNHANG**: se nessun figlio specificato da **pid** ha cambiato stato, restituisce 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**.

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

58

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 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**.

59

STATUS VALUE

È possibile usare l'header `<sys/wait.h>` per usare delle macro per gestire gli status value

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

bits		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Normal termination		exit status (0-255)															
Killed by signal		unused (0)		termination signal (= 0)													
Stopped by signal				core dumped flag													
Continued by signal		stop signal			0x7f												



Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

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**.

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

62

ORFANI E ZOMBIE

Sono orfani i processi figli che hanno un padre che termina prima di loro.

Gli orfani sono "adottati" dal processo "padre di tutti" con **pid=1**.

I zombie sono processi figli che sono terminali prima che il parent abbia chiamato **wait()**;

Sono "creati" dal Kernel che li mantiene in una tabella dei processi.

COME UCCIDERE LO ZOMBIE

Si possono "far fuori" solo con la **wait** del padre.

Se il padre muore prima di fare la **wait**, ci penserà **pid=1** ad eseguirlo in automatico.

SE LA WAIT FALLISCE

L'unico modo sarà di far fuori il parent e lasciar fare a PID=1, altrimenti satureranno la tab. dei processi

ECCEZIONE NEL PRATICO >

L'istruzione carica in mem il programma specificato e abbandona quello attuale.

Questo vuol dire che, dopo aver eseguito execve();, le istruzioni successive saranno eseguite solo in caso d'errore.

```
#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 stringhe, 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.

Marco Botta- Laboratorio di Sistemi Operativi, corso B - turno T3

78

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 ricerchabile (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.

Marco Botta- Laboratorio di Sistemi Operativi, corso B - turno T3

80

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

```
#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.

84

```
int execp(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.

Marco Botta- Laboratorio di Sistemi Operativi, corso B - turno T3

85

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

```
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 delle funzioni terminano con la lettera **e** (da **environment**) per indicare questa caratteristica.

86

SYSTEM >

È un comando che consente di invocare la shell ed eseguire un comando arbitrario.

È più comodo per eseguire i programmi ma inefficiente

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");
```

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

93

Giorno 13

94

Esempio di Re-implementazione

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 `exec()` con gli argomenti corrispondenti al comando `sh`:

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

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

100

```
/* 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;
}
```