Soluzioni Esercizi C - Gestione Processi (`fork` e `wait`)

November 17, 2025

Consegne degli Esercizi

Esercizio 1: Padre e Figlio con Attesa

Obiettivo: Creare un singolo processo figlio e garantire che il padre aspetti la sua terminazione.

- Lanciare un processo che genera un figlio.
- Il **padre** deve stampare il proprio PID e indicare che sta aspettando.
- Il **figlio** deve stampare il proprio PID, eseguire una breve attesa (`sleep(2)`) e terminare (`exit(0)`).
- Il **padre DEVE** attendere esplicitamente la terminazione del figlio usando `wait(NULL)`.

Esercizio 2: Generazione di N Figli e Attesa

Obiettivo: Gestire la creazione e l'attesa di processi multipli in un ciclo.

- Prendere in input un numero intero $N > 0$.
- Il processo deve creare **N figli** (usare un ciclo `for`).
- Ogni figlio stampa il proprio PID e termina.
- Il padre **DEVE attendere** la terminazione di **tutti** i figli (usare un secondo ciclo `for` con `wait(NULL)`).

Esercizio 3: Gerarchia Composta di Processi

Obiettivo: Creare una struttura gerarchica multilivello e garantire l'attesa ordinata.

- Lanciare un processo (**ROOT**) che genera due figli (**A** e **B**).
- **A** genera a sua volta il figlio **A1**.
- **B** genera i figli **B1** e **B2**.
- Ogni processo deve mostrare in output una cosa del tipo: "Sono B2, figlio di B (PID: ..., Padre: ...)".
- Bisogna far terminare i processi in modo ordinato utilizzando le opportune `wait()` (ogni padre attende i propri figli).

Esercizio 4: Calcolo Distribuito della Somma

Obiettivo: Utilizzare i processi per la parallelizzazione dividendo un compito computazionale.

- Il processo **Padre** prende in input la dimensione N dell'array (max `MAX_SIZE`) e i suoi elementi.
- Il **Padre** divide l'array in due metà.
- Il **Padre** genera due figli: **Figlio 1** (per la prima metà) e **Figlio 2** (per la seconda metà).
- Ciascun figlio calcola e stampa la **somma parziale** della propria sezione dell'array.
- Il **Padre** attende la terminazione di entrambi i figli prima di concludere.

Esercizio A: Catena di Processi e Sincronizzazione a Cascata

Obiettivo: Implementare una sequenza di processi a catena per garantire che le operazioni avvengano in un ordine preciso e che ogni padre attenda il suo diretto discendente.

- Creare una catena di quattro processi: **ROOT** $\rightarrow$ **A** $\rightarrow$ **B** $\rightarrow$ **C**.
- Ogni processo (**A, B, C**) deve stampare nome, PID e PID del padre.
- Introdurre un **ritardo casuale** (`sleep()`) in A, B e C per simulare un carico di lavoro.
- L'attesa è strettamente locale: **ROOT attende A**, **A attende B**, e **B attende C**.
- Ogni processo stampa un messaggio quando **termina** l'attesa del proprio figlio.

Esercizio B: Controllo Condizionale e Messaggi di Stato

Obiettivo: Utilizzare la struttura condizionale di `fork()` per differenziare il lavoro e usare `exit()` per simulare uno stato condizionale non analizzato dal padre.

- Lanciare un processo che genera un figlio.
- Il **figlio** prende in input un numero intero **X**.
- Il **figlio** stampa un messaggio che indica se X è **pari** o **dispari**, e poi termina.
- Il **padre** attende la terminazione del figlio con `wait(NULL)`.
- Dopo l'attesa, il padre stampa un messaggio di conferma che il figlio ha completato il suo compito.
- (Interno al codice del figlio) Assicurarsi che il figlio termini con un valore diverso se X è pari (`exit(0)`) o dispari (`exit(1)`), senza che il padre debba analizzare questo valore.

1 Esercizio 1: Padre e Figlio con Attesa

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

int main(){
    pid_t pid = fork();

    if (pid < 0){
        perror("Errore avvenuto nella funzione fork.");
        return 1;
    }
    else if (pid == 0){
        printf("Sono il processo figlio: %i \n", getpid());
        sleep(2);
        exit(0);
    }
    else{
        printf("Sono il processo padre %i e attendo che il processo %i termini.\n",
    getpid(),pid);
        wait(NULL);
        printf("Padre: Processo figlio %i terminato.\n", pid);
    }
    return 0;
}
```

Listing 1: es1.c

2 Esercizio 2: Generazione di N Figli e Attesa

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

int main(){
    int n;
    pid_t pid;

    do{
        printf("Inserisci il numero di processi figli che desideri creare. \n");
        scanf("%i",&n);
    }while(n<=0);

    for(int i = 0; i < n; i++){
        pid = fork();
        if(pid < 0){
            perror("Errore avvenuto nella fork.");
            return 1;
        }
        else if (pid == 0){
            printf("Sono il processo figlio %i \n",getpid());
            exit(0);
        }
    }

    for(int i = 0; i < n; i++){
        pid_t pid_terminazione = wait(NULL);
        if (pid_terminazione > 0 ){
            printf("Terminazione rilevata del processo figlio %i. \n",pid_terminazione);
        }
    }

    printf("Il processo padre termina.\n");
    return 0;
}
```

Listing 2: es2.c

3 Esercizio 3: Gerarchia Composta di Processi

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

int main() {
    pid_t pidA;
    pid_t pidB;

    printf("Sono il processo ROOT (Padre di A e B), PID: %i\n", getpid());

    pidA = fork();

    if (pidA < 0) {
        perror("Errore nella fork per A");
        return 1;
    }
    else if (pidA == 0) {
        printf("Sono A, figlio di ROOT (PID: %i, Padre: %i)\n", getpid(), getppid());

        pid_t pidA1 = fork();

        if (pidA1 < 0) {
            perror("Errore nella fork per A1");
            exit(1);
```

```
        } else if (pidA1 == 0) {
            printf("Sono A1, figlio di A (PID: %i, Padre: %i)\n", getpid(), getppid());
            exit(0);
        } else {
            printf("A (PID %i) attende A1 (PID %i)...\n", getpid(), pidA1);
            wait(NULL);
            printf("A: A1    terminato.\n");
            exit(0);
        }
    }


    pidB = fork();

    if (pidB < 0) {
        perror("Errore nella fork per B");
        waitpid(pidA, NULL, 0);
        return 1;
    }
    else if (pidB == 0) {
        printf("Sono B, figlio di ROOT (PID: %i, Padre: %i)\n", getpid(), getppid());

        pid_t pidB1 = fork();
        if (pidB1 == 0) {
            printf("Sono B1, figlio di B (PID: %i, Padre: %i)\n", getpid(), getppid());
            exit(0);
        }

        pid_t pidB2 = fork();
        if (pidB2 < 0) {
            perror("Errore nella fork per B2");
            exit(1);
        } else if (pidB2 == 0) {
            printf("Sono B2, figlio di B (PID: %i, Padre: %i)\n", getpid(), getppid());
            exit(0);
        }

        printf("B (PID %i) attende i suoi 2 figli...\n", getpid());
        wait(NULL);
        wait(NULL);
        printf("B: B1 e B2 sono terminati.\n");
        exit(0);
    }
    else {
        printf("ROOT (PID %i) attende i figli A (PID %i) e B (PID %i)...\n", getpid(),
    pidA, pidB);
        wait(NULL);
        wait(NULL);
        printf("ROOT: A e B sono terminati. Programma concluso.\n");
    }

    return 0;
}
```

Listing 3: es3.c

4 Esercizio 4: Calcolo Distribuito della Somma

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

#define MAX_SIZE 100

void calculate_sum(int arr[], int start, int end, const char *name) {
    long long sum = 0;
    if (start <= end) {
        for (int i = start; i <= end; i++) {
            sum += arr[i];
```

```
        }
    }
    printf("Sono il processo %s (PID: %i, Padre: %i), ho calcolato la somma degli elementi da indice %d a %d.\n",
           name, getpid(), getppid(), start, end);
    printf("Risultato del processo %s: Somma parziale = %lld\n", name, sum);
}

int main() {
    int arr[MAX_SIZE];
    int n, mid;
    pid_t pid1, pid2;

    do {
       printf("Inserisci la dimensione dell'array (N, max %d): ", MAX_SIZE);
       scanf("%d", &n);
       if (n <= 0 || n > MAX_SIZE) {
         fprintf(stderr, "Dimensione non valida. Riprova (deve essere tra 1 e %d).\n", MAX_SIZE);
       }
    } while (n <= 0 || n > MAX_SIZE);

    printf("Inserisci i %d elementi dell'array:\n", n);
    for (int i = 0; i < n; i++) {
        printf("Elemento [%d]: ", i);
        if (scanf("%d", &arr[i]) != 1) {
            fprintf(stderr, "Errore nell'input. Programma terminato.\n");
            return 1;
        }
    }

    mid = n / 2;

    printf("\nProcesso Padre (PID: %i): Array diviso in due parti.\n", getpid());

    // --- Fork per il Figlio 1 (Prima met : da 0 a mid-1) ---
    pid1 = fork();

    if (pid1 < 0) {
        perror("Errore nella fork del Figlio 1");
        return 1;
    }
    else if (pid1 == 0) {
        calculate_sum(arr, 0, mid - 1, "Figlio 1 (Prima met )");
        exit(0);
    }

    // --- Fork per il Figlio 2 (Seconda met : da mid a n-1) ---
    if (pid1 > 0) {
        pid2 = fork();

        if (pid2 < 0) {
            perror("Errore nella fork del Figlio 2");
            waitpid(pid1, NULL, 0);
            return 1;
        }
        else if (pid2 == 0) {
            calculate_sum(arr, mid, n - 1, "Figlio 2 (Seconda met )");
            exit(0);
        }

        // --- Codice Padre (Attesa dei figli) ---
        else {
            printf("Processo Padre (PID: %i): Attendo la terminazione dei figli %i (F1) e %i (F2).\n", getpid(), pid1, pid2);
            wait(NULL);
            wait(NULL);
            printf("Processo Padre (PID: %i): Entrambi i figli sono terminati. Programma concluso.\n", getpid());
        }
    }

    return 0;
```

```
83 }
```

Listing 4: es4.c