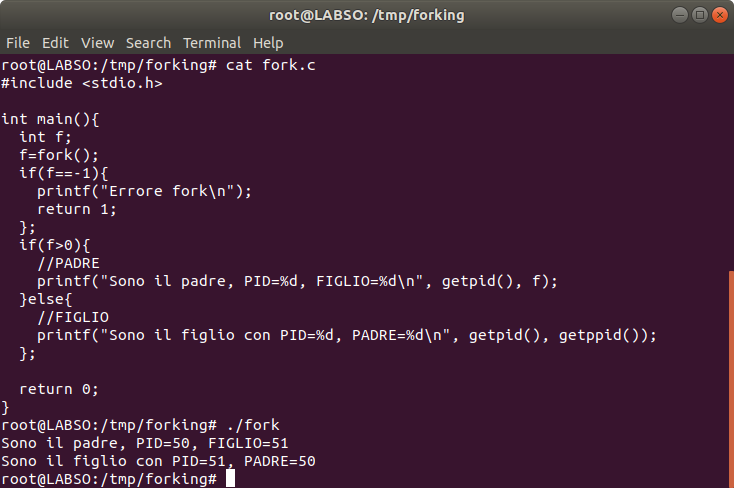
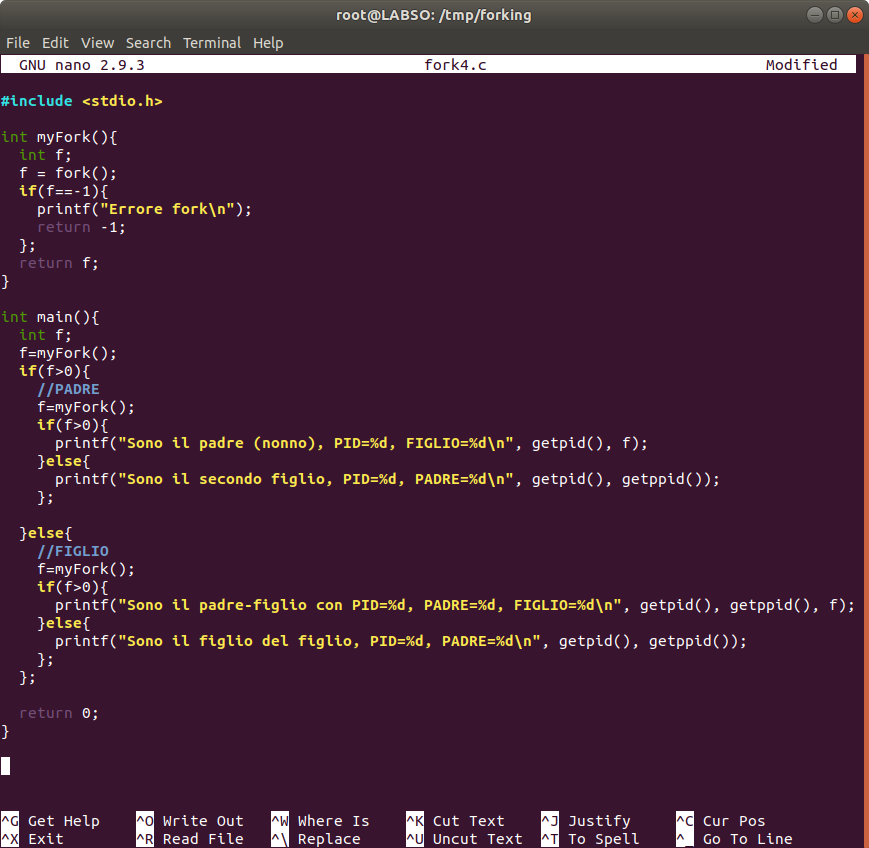
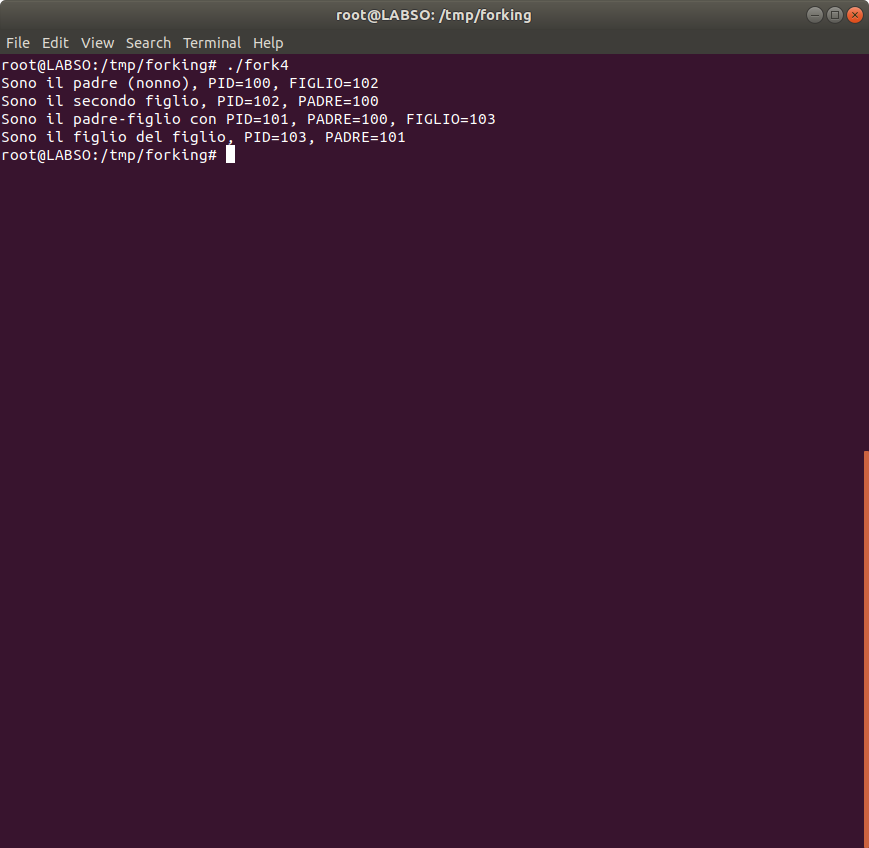
**Laboratorio Sistemi Operativi: Fork**

* **IPC**:**.**
  + **Meccanismo** che permette a più **processi di interagire e comunicare tra loro**.
  + Per poter comunicare tra loro, i processi, hanno bisogno di **condividere** qualche **risorsa**:
    - **Generation**: un processo genera una risorsa che viene condivisa con i processi figli.
    - **File system**: tramite file.
  + Ci deve essere una **metodologia** (protocollo) **in comune tra i processi** che vogliono interagire e comunicare tra loro.
  + **Ogni processo**, eseguito all’interno del sistema, ha una propria **immagine salvata in memoria** che comprende (oltre al codice, stack, heap) anche:
    - **Program Counter**: indice che indica la posizione dell’esecuzione attuale
    - **Registers**
    - **Variables**
    - **File Descriptors**: ad ogni processo è associato un elenco di file attivi (tabella dei descrittori)
* **Fork**:
  + **int fork():**
    - **Syscall** che permette la **creazione di un nuovo processo**, questo nuovo processo è un **duplicato del padre** (“*tutto quello che c’è dopo la chiamata alla funzione fork() verrà eseguito 2 volte*”).
    - Sia il nuovo processo che il processo padre eseguiranno il codice che trovano dopo la funzione fork().
  + La **fork()** non crea un processo “vuoto”, ma **clona il processo padre**:
    - Il nuovo processo creato è una copia del processo padre ma è una **copia “alterata”** così da poter creare una **gerarchia** (definita e gestita dal SO) tra padre e figlio.
    - Anche il **workflow dei due processi è identico** (andranno ad eseguire le stesse istruzioni, **hanno identico PC**) che continua la sua esecuzione, conclusa l’invocazione della funzione fork()
    - Se va a buon fine, **restituisce** valori diversi:
      * Al **processo padre** **restituisce il PID del processo appena creato**.
      * Al **processo figlio restituisce 0**.
    - Se non va a buon fine, **nessun processo viene creato** e **restituisce il codice d’errore -1**.
  + Una volta eseguita la fork(), sarà creato un nuovo processo e sarà lo **scheduler** a **decidere quale processo eseguire**:
    - Non sappiamo se verrà eseguito prima il processo padre o il figlio.
  + Per poter **distinguere** e **diversificare l’esecuzione dei due processi** (visto che hanno stessa logica e stesso codice) **viene usato il valore che restituisce la fork()**:







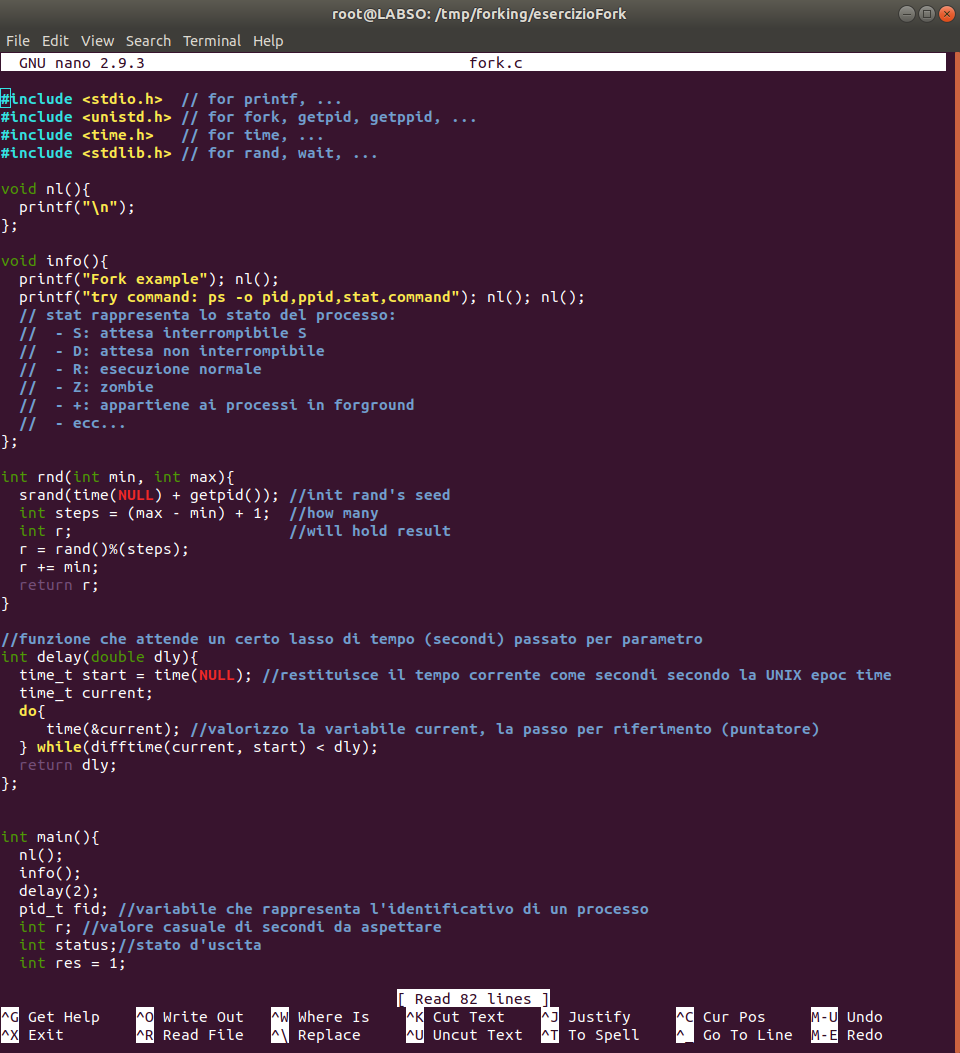
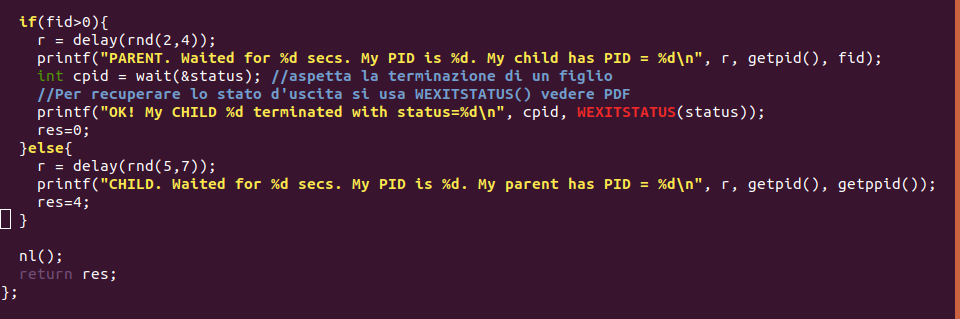
* **Forking**:
  + **pid\_t fork()**: genera un nuovo processo.
  + **pid\_t getpid()**: restituisce il **pid del processo** attuale.
  + **pid\_t getppid()**: restituisce il **pid del padre** del processo attuale.
  + Funzioni permettono la **sincronizzazione** tra processi:
    - **pid\_t wait(int \*status)**:
      * Funzione **bloccante** che fa si che il processo padre **aspetti che un figlio qualsiasi termini la sua esecuzione** prima di continuare con la sua esecuzione.
      * La **funzione restituisce il PID del processo figlio che è terminato**.
      * Se non ci interessa il codice d’uscita del processo figlio, basta invocare la **wait(arg)** con **arg=NULL**, mentre se viene passata una variabile alla funzione (**wait(&var)**) la funzione salva il codice d’uscita del processo figlio in quella variabile.
    - **pid\_t waitpid(pid\_t pid, int \*status, int options):**
      * Funzione simile alla wait() descritta sopra che permette al processo padre di **specificare quale figlio aspettare**, prima di continuare con la propria esecuzione.
      * **pid\_t pid**:
        + **pid > 0**: **aspetta** che il **processo figlio con quel PID** termini.
        + **pid = -1** o **pid = NULL**: aspetta la terminazione di un qualunque figlio.
        + **pid < -1**:

**Aspetta la terminazione di un figlio qualunque** che **appartenga** ad un **certo gruppo** (il gruppo identificato dal **valore assoluto** del pid).

* + - * + **pid = 0**:

**Aspetta la terminazione di un qualunque processo** che **appartiene** al **gruppo del processo che siamo eseguendo**.

* + - * **int options**:
        + Utili per **gestire casi particolari** (figlio terminato normalmente, è stato killato, ecc.).
        + Serve per verificare delle situazioni particolari.
      * Nota:
        + **wait(NULL) == waitpid(-1, NULL, 0)**
  + Tutti i processi hanno un **tempo di vita** (parte dalla creazione e finisce quando terminano).
  + Quando un processo crea un processo figlio, si possono avere due casi:
    - **Processo padre termini prima del figlio**:
      * C’è stata una **chiusura forzata del padre**, il processo figlio non ci interessa (viene **killato**).
    - **Processo figlio termini prima del padre**:
      * Il **figlio termina l’esecuzione**, il **valore di risposta viene mantenuto dal SO** a disposizione del processo padre.
      * Nell’arco di tempo in cui il figlio è terminato e il padre non ha ancora letto il valore d’uscita, anche se in realtà il **processo figlio è già terminato**, nel **SO questo processo rimane “pendente”,** cioè il **riferimento** di questo **processo figlio terminato** **rimane in memoria finché il processo padre non cattura** ilsuo **stato d’uscita**.
      * Questo processo viene definito come “**processo zombie**” anche se in realtà non è un processo, **è solo un riferimento pendente in memoria**, che può essere utilizzato dal padre nel momento in cui fa la wait() per leggere il suo valore d’uscita.
      * Il **padre deve sempre catturare lo stato d’uscita dei processi figli** con la **wait()** (se non lo fa, questi rimangono sempre pendenti in memoria).

Esercizio:

Esempio: