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()**:

```
root@LABSO:/tmp/forking# cat fork.c
#include <stdio.h>

int main(){
  int f;
  f=fork();
  if(f==-1){
    printf("Errore fork\n");
    return 1;
  };
  if(f>0){
    //PADRE
    printf("Sono il padre, PID=%d, FIGLIO=%d\n", getpid(), f);
  }else{
    //FIGLIO
    printf("Sono il figlio con PID=%d, PADRE=%d\n", getpid(), getppid());
  };

  return 0;
}
root@LABSO:/tmp/forking# ./fork
Sono il padre, PID=50, FIGLIO=51
Sono il figlio con PID=51, PADRE=50
root@LABSO:/tmp/forking#
```

```c
#include <stdio.h>

int myFork(){
  int f;
  f = fork();
  if(f==-1){
    printf("Errore fork\n");
    return -1;
  };
  return f;
}

int main(){
  int f;
  f=myFork();
  if(f>0){
    //PADRE
    f=myFork();
    if(f>0){
      printf("Sono il padre (nonno), PID=%d, FIGLIO=%d\n", getpid(), f);
    }else{
      printf("Sono il secondo figlio, PID=%d, PADRE=%d\n", getpid(), getppid());
    };

  }else{
    //FIGLIO
    f=myFork();
    if(f>0){
      printf("Sono il padre-figlio con PID=%d, PADRE=%d, FIGLIO=%d\n", getpid(), getppid(), f);
    }else{
      printf("Sono il figlio del figlio, PID=%d, PADRE=%d\n", getpid(), getppid());
    };
  };

  return 0;
}
```

```
root@LABSO:/tmp/forking# ./fork4
Sono il padre (nonno), PID=100, FIGLIO=102
Sono il secondo figlio, PID=102, PADRE=100
Sono il padre-figlio con PID=101, PADRE=100, FIGLIO=103
Sono il figlio del figlio, PID=103, PADRE=101
root@LABSO:/tmp/forking#
```

- **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** il suo **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:

```c
#include <stdio.h>  // for printf, ...
#include <unistd.h> // for fork, getpid, getppid, ...
#include <time.h>   // for time, ...
#include <stdlib.h> // for rand, wait, ...

void nl(){
  printf("\n");
};

void info(){
  printf("Fork example"); nl();
  printf("try command: ps -o pid,ppid,stat,command"); nl(); nl();
  // stat rappresenta lo stato del processo:
  //  - S: attesa interrompibile S
  //  - D: attesa non interrompibile
  //  - R: esecuzione normale
  //  - Z: zombie
  //  - +: appartiene ai processi in forground
  //  - ecc...
};

int rnd(int min, int max){
  srand(time(NULL) + getpid()); //init rand's seed
  int steps = (max - min) + 1;  //how many
  int r;                        //will hold result
  r = rand()%(steps);
  r += min;
  return r;
}

//funzione che attende un certo lasso di tempo (secondi) passato per parametro
int delay(double dly){
  time_t start = time(NULL); //restituisce il tempo corrente come secondi secondo la UNIX epoc time
  time_t current;
  do{
     time(&current); //valorizzo la variabile current, la passo per riferimento (puntatore)
  } while(difftime(current, start) < dly);
  return dly;
};

int main(){
  nl();
  info();
  delay(2);
  pid_t fid; //variabile che rappresenta l'identificativo di un processo
  int r; //valore casuale di secondi da aspettare
  int status;//stato d'uscita
  int res = 1;

  //Con \n, la funzione manda il buffer nel canale d'uscita (stdout) e viene stampato a video
  //printf("PRIMA...\n"); delay(5); printf("...DOPO\n");
  //Senza \n, non e' detto che la funzione mandi il buffer in uscita, quindi non vediamo la scritta a vi$
  //Per risolvere il problema basta usare la funzione fflush() che forza l'uscita dei dati sul canale st$
  //printf("PRIMA..."); fflush(stdout); delay(5); printf("...DOPO\n");

  fid = fork();
  if(fid == -1){
    printf("?Error. Forking failed\n");
    return res; //meglio avere un unico punto di uscita (un unico return)
  };

  printf("FORKED. PID=%d, PPID=%d\n", getpid(), getppid());
  delay(2);
  nl();

  if(fid>0){
    r = delay(rnd(2,4));
    printf("PARENT. Waited for %d secs. My PID is %d. My child has PID = %d\n", r, getpid(), fid);
    int cpid = wait(&status); //aspetta la terminazione di un figlio
    //Per recuperare lo stato d'uscita si usa WEXITSTATUS() vedere PDF
    printf("OK! My CHILD %d terminated with status=%d\n", cpid, WEXITSTATUS(status));
    res=0;
  }else{
    r = delay(rnd(5,7));
    printf("CHILD. Waited for %d secs. My PID is %d. My parent has PID = %d\n", r, getpid(), getppid());
    res=4;
  }

  nl();
  return res;
};
```

Esempio:

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

int figlio(){
  int r=3;
  printf("FIGLIO\n");
  return r;
};

int padre(){
  int r=4;
  int c;
  int st;

  printf("PADRE\n");
  c = wait(&st); //attendo che un figlio termini, passo st per riferimento

  //c = identificatore del processo terminato (PID del figlio)
  //st = stato di ritorno, non solo il codice d'errore numerico (contiene tante informazioni)
  //utilizzando delle macro definite nella libreria wait.h, e' possibile estrapolare i singoli
  //componenti di questo stato passando la variabile inizializzata tramite wait
  //WEXITSTATUS(var) restituisce lo stato d'uscita del processo terminato
  printf("wait result: %d, %d\n", c, WEXITSTATUS(st));
  return r;
};

int main(){
  int r=0;
  int f;
  f = fork();
  if(f==-1) {
    printf("?Error fork\n");
    r=1;
  }else{
    if(f == 0){
      r = figlio();
    }else{
      r = padre();
    }
  }
  return r;
};
```

WAIT/WAITPID

Informazioni aggiuntive (parlando di WAIT negli esempii, ma WAITPID è analoga)

Lo "status" restituito nell'eventuale variabile passata come argomento NON è SOLO il codice di ritorno (return/exit) del figlio: include altre informazioni.

Importando la libreria <sys/wait.h> abbiamo diverse "macro" a disposizione (utilizzabili a mo' di funzione) per recuperare informazioni più precise.

Esempio:

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

int figlio() {
	int r = 2;
	printf("FIGLIO!\n");
	return r;
};

int padre() {
	int r = 3, c, st;
	printf("PADRE!\n");
	c = wait(&st);
	// per recuparerare dallo stato il codice di ritorno
	// uso WEXITSTATUS: andrebbe usata dentro un
	// if (WIFEXITED(st)) { … } qui non presente per brevità
	printf("wait result: %d, %d\n", c, WEXITSTATUS(st)); // <---
	return r;
};

int main() {
	int r=0;
	int f;
	f = fork();
	if (f==0) {
		r = figlio();
	} else {
		r = padre();
	};
	return r;
};
```

Alcune delle macro più utili:

- WIFEXITED(status): processo terminato normalmente, poi:
  WEXITSTATUS(status): codice di ritorno del processo terminato

- WIFSIGNALED(status): processo terminato da un segnale, poi:
  WTERMSIG(status): codice del segnale di terminazione

- WIFSTOPPED(status): processo interrotto da STOP, poi:
  WSTOPSIG(status): codice del segnale di interruzione

In ciascuno dei 3 casi indicati bisogna utilizzare la seconda MACRO “dentro” un IF in cui verifichiamo in quale delle 3 situazioni ci troviamo, tipo:

```
…
if (WIFEXITED(status)) {
	printf("Normal exit with code=%d\n",WEXITSTATUS(status));
}
if (WIFSIGNALED(status)) {
	printf("Signal exit as signal %d\n",WTERMSIG(status));
}
…
```

ATTENZIONE!!! le “wait” attendono NON solo la terminazione del figlio, ma rispondono anche nel caso arrivi loro un segnale gestito!!!