1. creazione di processi e condivisione di file

Scrivere un programma in cui un padre e un figlio condividono un file aperto (inizialmente vuoto): il figlio modifica il file e il padre, dopo avere atteso la terminazione del figlio, stampa a video il contenuto del file.

2. wait()

- Che cosa restituisce la system call *wait*? (*man -s 2 wait*)
- A cosa serve l'argomento della *wait*? (*man -s 2 wait*)
- In che senso padre e figlio "condividono" le variabili?
- Dichiarare una variabile intera *myvar* e, prima della *fork()*, assegnarle un valore *X*. Modificare il valore di *myvar* nel corpo del figlio. Stampare *myvar* nel codice del padre prima e dopo la *wait()*. Che valori di *X* possiamo aspettarci? Perché?
- Cosa succede se il processo padre non attende la terminazione del figlio con una *wait()*?

3. fork()

Quanti processi generiamo usando questo codice?

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

E questo codice quanti processi genera?

```
for (i=0; i<3; i++) {
  n = fork();
  if (n == 0) printf("sono un figlio");
  else printf("sono un padre");
}
```

E invece quanti processi genera questo codice?

```
for (i=0; i<3; i++) {
  n = fork();
  if (n == 0) {
    printf("sono un figlio");
    exit(0);
  } else
    printf("sono un padre");
```

```
}
```

Vedete differenze tra i tre programmi? Quale programma si comporta "correttamente"?

4. creazione di processi

Scrivere un programma *creaproc* che effettua le seguenti operazioni:

- Crea 3 figli; Dopo aver creato i 3 figli si mette in attesa della loro terminazione con una *wait()*;
- Ogni volta che un processo figlio termina, il processo padre stampa il PID del figlio appena terminato
    - Rendere *creaproc* parametrico, passando in input il numero di figli da creare;
    - Fare in modo che ogni figlio, prima di terminare, attenda *n* secondi (dove *n* è l'ultima cifra del *PID* di quel figlio)

5. attesa della terminazione dei figli

Scrivere un programma che prende in input un numero variabile di interi $i_1, i_2, \ldots i_n$. 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 attende 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.

6. creazione di zombie

Scrivere un programma in cui viene generato un processo figlio. Il processo figlio termina immediatamente. Il padre si mette in *sleep* per 10 secondi e poi invia un messaggio *KILL* al figlio -già terminato- con l'istruzione

```
kill(childPid, SIGKILL)
```

Predisporre le opportune stampe a video per verificare lo stato del figlio dopo l'invio del segnale kill.

7. exec

Scrivere un programma in cui viene eseguita una *fork()*. Il processo figlio esegue una *execlp()* chiamando un secondo programma ('*saluta_persone*') e passando un certo numero di nomi propri di persona ('mario', 'ada', etc.) perché *saluta_persone* stampi sullo schermo questi nomi.

riscrivere il programma precedente (modificando opportunamente anche saluta_persone.c) eseguendo questa volta *execvp()* invece di *execlp()*.

8. re-implementazione del comando *cp*

Scrivere 2 programmi che implementino le funzionalità della seguente istruzione:

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

Il primo programma prende i nomi dei due file dagli argomenti, forca e il figlio invoca tramite *execl* il secondo programma, cui passa (sempre come argomenti della linea di comando) i nomi dei due file. Il secondo programma apre i file e copia il contenuto del primo nel secondo.

Modificare l'invocazione del secondo programma utilizzando la *execv*, e apportando al resto del programma le modifiche necessarie.