Laboratorio 3: Processi

Si realizzi in C una versione semplificata del comando GNU `parallel`. Il comando `parallel` serve per eseguire in parallelo una sequenza di comandi Bash, in modo da sfruttare un sistema multicore.

Il programma deve accettare tre argomenti:

1. Il nome di un **file** dove il programma legge, uno per **riga**, i **parametri** del comando che deve eseguire
2. Un **numero** che rappresenta **quanti processi** in parallelo deve eseguire
3. Una stringa che rappresenta il **comando** da eseguire. La stringa deve contenere il carattere `%` che verrà sostituito dal programma con i parametri letti dal file passato come primo argomento.

Esempio:

Se viene eseguito **`./parallel args.txt 2 "ls % -lh"`** e il file **`args.txt`** contiene:

```
/etc/resolv.conf
/etc/fstab
/etc/hostname
/etc/hosts
```

Il programma deve leggere le **4** righe dal file **`args.txt`** e sostituirle al carattere `%` nella stringa passata come terzo argomento per creare **4 comandi** che esso deve eseguire usando **2 processi** in parallelo. I comandi da **eseguire** saranno pertanto:

```
ls /etc/resolv.conf -lh
ls /etc/fstab -lh
ls /etc/hostname -lh
ls /etc/hosts -lh
```

Nota: il terzo argomento del programma è `"ls % -lh"`. Esso contiene degli spazi, ma viene trattato come un unico argomento in quanto incluso tra virgolette. Pertanto, è contenuto in un singolo elemento nel vettore `argv`, precisamente sarà contenuto in `argv[3]`. In `argv[3]`, esso non ha le virgolette per tanto `argv[3]` conterrà la stringa `ls % -lh`. In altre parole, **non** è necessario scrivere del codice per gestire le virgolette.

Nota: Si suggerisce di risolvere l'esercizio con le *pipe* ma sono possibili e validi altri approcci.

Nota: Per sostituire il carattere `%` con il parametro del comando, si considerino le funzioni `strstr()`, e `strcpy()`.

Suggerimenti:

- Il programma deve creare un numero di **processi figli** pari al numero passato come secondo argomento. Si usi la `fork()` per creare i processi e la `wait()` per attenderne la terminazione.
- È consigliato usare delle ***pipe*** con cui il processo padre fornisce i comandi da eseguire ai processi figli. Si consiglia di creare una *pipe* per ogni processo figlio in cui il padre scrive i

comandi che il dato figlio deve eseguire. Il padre suddivide i comandi da eseguire tra i vari processi figli. Per creare un *vettore* di pipe bisogna di fatto creare una *matrice* del tipo:

```
int pipes [MAXPROCESSES][2];
```

Per semplicità, `MAXPROCESSES` è una costante definita tramite una `#define`. La si setti per comodità a `256`.

Ogni figlio, riceve un certo numero di comandi e li esegue usando la funzione `system()`

- Siccome i comandi sono di **lunghezza variabile**, è necessario stabilire una convenzione per **delimitarli**.
  Sono possibili almeno tre diversi approcci, si vedano le slide. L'approccio più semplice è definire una **lunghezza massima** in caratteri che può avere un comando (ad esempio tramite `#define MAXCMD 256`) e quindi leggere/scrivere sempre buffer di quella lunghezza.
  Ad esempio, se bisogna trasmettere i due comandi `echo a` ed `echo b` e la lunghezza massima dei comandi è 8, verranno trasmessi i seguenti Byte.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
'e'	'c'	'h'	'o'	' '	'a'	'\0'	'\0'	'e'	'c'	'h'	'o'	' '	'b'	'\0'	'\0'
Messaggio 1								Messaggio 2							

- I processi figli devono terminare nel momento in cui hanno eseguito tutti i comandi. Si ricordi però che:
  - La System Call `read()` blocca il chiamante finché non viene letto almeno un carattere. Bisogna evitare quindi che un figlio resti bloccato su una `read()` aspettando i dati di un comando che non arriverà mai
  - La soluzione più semplice è che il padre chiuda la *pipe* dopo aver scritto tutti i comandi. Questo indica la fine della trasmissione dei dati sulla *pipe*, facendo sì che una `read()` ritorni 0 quando tutti i dati sono stati letti. Un figlio, quando la `read()` ritorna 0, realizza che non ha altri comandi da eseguire e termina.
  - La *pipe* è considerata chiusa se il *file descriptor* di scrittura viene chiuso in ***tutti i processi*** che l'hanno aperto. Si ricordi a tale proposito che i *file descriptor* vengono ereditati dai processi figli (e per tanto vanno chiusi in tutti i figli).
  - Perciò, il seguente spezzone di codice deve essere eseguito dai figli appena nascono e dal padre dopo che esso ha scritto tutti gli opportuni comandi nella *pipe*

```
for (i=0; i<N; i++)
        close(pipes[i][1]);
```

Istruzioni per la consegna

Il programma fornire un output con lo ***stesso esatto formato*** usato negli esempi.

Il programma deve poter lavorare con qualsiasi dato di input. La correzione consisterà nell'esecuzione automatizzata del programma con un input generato dal docente.

Per la consegna del programma, usare la funzione *Assignments* su Microsoft Teams, facendo l'upload di ***un singolo file*** nell'*assignment* relativo a questa esercitazione. Nel caso l'esercitazione richieda un programma in C, va consegnato il **file sorgente** e non il programma compilato.