labso2021-1b--esame

INDICAZIONI

All'interno della cartella principale creare solo ed esattamente 2 file: un sorgente C "`main.c`" (che dovrà generare un eseguibile denominato "`app`") e un makefile "`Makefile`" (eliminare file temporanei, di servizio, generati, etc. prima della consegna!!!)

Realizzare un'applicazione in C che accetta in input due argomenti ordinati: un percorso del file-system (argomento "path") e un intero da 1 a 10 (compresi gli estremi, argomento "n") in tale ordine. Il codice di uscita dell'applicazione deve essere 0 se non ci sono errori.

Esempio di chiamata: `./app /tmp 7`

L'applicazione deve realizzare le funzionalità indicate più avanti riportando eventuali messaggi d'errore (ad esempio numero argomenti errato o valori non accettabili, errori di I/O, altro) su *stderr* con un codice di uscita maggiore di 0.

- [2 punti] Il Makefile deve funzionare in modo che eseguendo il comando "`make`" senza alcun parametro l'applicazione sia correttamente compilata generando l'eseguibile denominato "`app`".

- L'applicazione realizzata con il file "`main.c`" deve:

    - [8 punti] creare una sottocartella "`info`" sotto <path> (che DEVE esistere, altrimenti si ha un errore) con flags `0755`, creare un file al suo interno con permessi analoghi denominato "`key.txt`" contenente il "pid" del processo principale + "a capo", generare "n" processi discendenti e creare tanti "logfile" (file di testo, inizialmente vuoti) con flags `0755` e nome *pid*.txt (dove *pid* è ovviamente il "pid" corrispondente) per ciascuno sotto <path>/info e poi terminare lasciandoli attivi (senza farli terminare a loro volta).

      Se ad esempio si esegue l'applicazione con `./app /tmp 2` si ha che l'argomento "path" è "/tmp" e l'argomento "n" è 2. Supponendo che il suo pid sia 99 e i processi generati hanno ad esempio pid 100 e 101, allora devono essere creati i file inizialmente vuoti `/tmp/info/100.txt` e `/tmp/info/101.txt` oltre che il file `/tmp/info/key.txt` con all'interno la stringa "99"+"a capo" (ovviamente i "pid" cambieranno di volta in volta).

    - [4 punti] stampare, prima di restituire il prompt all'utente, su *stdout* l'elenco dei PID dei discendenti generati come sequenza - nell'ordine di generazione - separata da un spazi singoli (es. "`100 101`") e con un "a capo" in coda.

      Se si eseguisse ad esempio `./app /tmp 2 > /tmp/pids.txt` allora deve avvenire tutto come indicato ma l'utente vedrà solo di nuovo il prompt e l'output sarà invece salvato nel file `/tmp/pids.txt` che quindi conterrà la stringa "`100 101`"+"a capo".

    - [8 punti] usando le "message queues" creare una coda - eliminandola prima di crearla di nuovo se esiste già - usando come chiave per generare l'id il valore restituito dalla funzione `ftok` cui passare come argomenti il file (ovviamente con il percorso completo) "`key.txt`" generato in precedenza e l'intero 32 e scriverci dentro un messaggio con testo uguale al "pid" (del processo principale, convertito ovviamente

in stringa) senza alcun “a capo” aggiuntivo in coda e tipo pari ad 1

- [4 punti] far sì che ogni discendente se riceve un segnale SIGUSR1 scriva sul corrispondente file txt una riga con il testo “SIGUSR1” con “a capo” in coda

- [4 punti] far sì che ogni discendente se riceve un segnale SIGUSR2 scriva sulla coda generata in precedenza un messaggio con testo uguale al proprio “pid” (convertito ovviamente in stringa) senza alcun “a capo” aggiuntivo in coda e tipo pari ad 1

L’effetto per l’utente, se l’applicazione è completa, deve consistere nel fatto che una volta eseguita si deve visualizzare un eventuale messaggio d’errore su *stderr* oppure su *stdout* la lista dei discendenti generati come sopra indicato (in entrambi i casi eventualmente su file se si usa il redirezionamento da bash) e avere di nuovo il controllo del prompt. Se si ritiene utile si possono generare ulteriori processi di appoggio oltre a quanto indicato per le interazioni specificate.

Alcuni richiami e indicazioni aggiuntive:

- per i permessi si può usare “`0755`” anziché combinazioni di label mnemoniche tipo “`S_IRUSR`”, “`S_IWUSR`” e altre
- come limiti tramite `#define` usare tra altri eventuali:
  `#define PATH_MAXLEN  60` (per lunghezza massima gestita dei percorsi nel file-system)
  `#define CHILDREN_MAX 10` (per numero massimo di figli gestibili)
  `#define PIDLEN  8` (per lunghezza massima di stringa che rappresenti un pid)
- Per concatenare stringhe si può usare `strcat`, per convertire da intero a stringa `sprintf`.

ESECUZIONI (esempi con applicazione completa)

- `make` → genera l’eseguibile app
- `./app /tmp/prova 2` → (se non esiste la cartella `/tmp/prova`) mostra un messaggio di errore su stderr e termina con un codice di uscita maggiore di 0 (perché non esiste la cartella)
- `./app /tmp/prova 11` → mostra un messaggio di errore su stderr e termina con un codice di uscita maggiore di 0 (a prescindere dall’esistenza della cartella il secondo parametro è errato)
- `./app` oppure `./app x` oppure `./app 10` → mostra un errore perché non ha due argomenti validi
- `./app /tmp/prova 3` → (se esiste la cartella `/tmp/prova` e se il processo parte con pid=99 e genera 3 figli con pid=100, 101 e 102) mostra a video “`100 101 102`” e va a capo mostrando di nuovo il prompt dei comandi, lasciando attivi i processi 100, 101 e 102. Crea la cartella file `/tmp/prova/info`, il file `/tmp/prova/info/key.txt` contenente la stringa “99” e un “a capo”, e i files `/tmp/prova/info/100.txt`, `/tmp/prova/info/101.txt` e `/tmp/prova/info/102.txt` vuoti. Crea una “message queue” opportuna e ci salva dentro un messaggio con testo “99” e tipo “1”. Inviando un SIGUSR1 a uno dei processi 100, 101 o 102, il processo stesso risponde scrivendo sul corrispondente file txt la stringa “`SIGUSR1`” e un “a capo”. Inviando invece un SIGUSR2 il processo risponde scrivendo nella coda creata in precedenza un messaggio con testo corrispondente al proprio pid e tipo “1”.