Sistemi Operativi, Secondo Modulo
A.A. 2018/2019
Testo del Secondo Homework

Emiliano Casalicchio, Igor Melatti

Come si consegna

Il presente documento descrive le specifiche per l'homework 2. Esso consiste di 3 esercizi, per risolvere i quali occorre creare 3 cartelle, con la cartella di nome i che contiene la soluzione all'esercizio i. La directory `1` dovrà contenere almeno un file `1.c` ed un file `Makefile`. Quest'ultimo dovrà generare un file `1` quando viene invocato. La directory `2` dovrà contenere almeno un file `2.server.c`, un file `2.client.c` ed un file `Makefile`. Quest'ultimo dovrà generare un file `2.server` ed un file `2.client` quando viene invocato. Infine, la directory `3` dovrà contenere almeno un file `3` ed un file `Makefile`. Quest'ultimo dovrà generare un file `3` quando viene invocato. Tutti i `Makefile` devono anche avere un'azione `clean` che cancella i rispettivi file eseguibili. Per consegnare la soluzione, seguire i seguenti passi:

1. creare una directory chiamata `so2.2018.2019.2.matricola`, dove al posto di `matricola` occorre sostituire il proprio numero di matricola;
2. copiare le directory `1`, `2` e `3` in `so2.2018.2019.2.matricola`
3. creare il file da sottomettere con il seguente comando: `tar cfz so2.2018.2019.2.matricola.tgz [1-3]`
4. andare alla pagina di sottomissione dell'homework `151.100.17.205/upload/index.php?id_appello=65` e uploadare il file `so2.2018.2019.2.matricola.tgz` ottenuto al passo precedente.

Come si auto-valuta

Per poter autovalutare il proprio homework, si hanno 2 possibilità:

- usare la macchina virtuale Debian-9 del laboratorio "P. Ercoli";
- installare VirtualBox (`https://www.virtualbox.org/`), e importare il file OVA scaricabile dall'indirizzo `https://drive.google.com/open?`

`id=1LQORjuidpGGt9UMrRupoY73w_qdAoVCp`; maggiori informazioni sono disponibili all'indirizzo `http://twiki.di.uniroma1.it/twiki/view/SO/SO1213AL/SistemiOperativi12CFUModulo220182019#software`. Si tratta di una macchina virtuale quasi uguale a quella del laboratorio. Si consiglia di configurare la macchina virtuale con NAT per la connessione ad Internet, e di settare una "Shared Folder" (cartella condivisa) per poter facilmente scambiare files tra sistema operativo ospitante e Debian. Ovvero: tramite l'interfaccia di VirtualBox, si sceglie una cartella x sul sistema operativo ospitante, gli si assegna (sempre dall'interfaccia) un nome y, e dal prossimo riavvio di VirtualBox sarà possibile accedere alla cartella x del sistema operativo ospitante tramite la cartella `/media/sf_`y di Debian.

All'interno delle suddette macchine virtuali, scaricare il pacchetto per l'autovalutazione (*grader*) dall'URL `151.100.17.205/download_from_here/so2.grader.2.20182019.tgz` e copiarlo in una directory con permessi di scrittura per l'utente attuale. All'interno di tale directory, dare il seguente comando:

```
tar xfzp so2.grader.2.20182019.tgz && cd grader.2
```

È ora necessario copiare il file `so2.2018.2019.2.matricola.tgz` descritto sopra dentro alla directory attuale (ovvero, `grader.2`). Dopodiché, è sufficiente lanciare `grader.2.sh` per avere il risultato: senza argomenti, valuterà tutti e 3 gli esercizi, mentre con un argomento pari ad i valuterà solo l'esercizio i (in quest'ultimo caso, è sufficiente che il file `so2.2018.2019.1.matricola.tgz` contenga solo l'esercizio i).

Dopo un'esecuzione del grader, per ogni esercizio $i \in \{1, 2, 3\}$, c'è un'apposita directory `input_output.`i contenente le esecuzioni di test. In particolare, all'interno di ciascuna di tali directory:

- sono presenti dei file `inp_out.`j`.sh` ($j \in \{1, \ldots, 6\}$) che eseguono la soluzione proposta con degli input variabili;
- lo standard output (rispettivamente, error) di tali script è rediretto nel file `inp_out.`j`.sh.out` (rispettivamente, `inp_out.`j`.sh.err`);
- l'input usato da `inp_out.`j`.sh` è nella directory `inp.`j;
- l'output creato dalla soluzione proposta quando lanciata da `inp_out.`j`.sh` è nella directory `out.`j;
- nella directory `check` è presente l'output corretto, con il quale viene confrontato quello prodotto dalla soluzione proposta.

Nota bene: per evitare soluzioni "furbe", le soluzioni corrette nella directory `check` sono riordinate a random dal grader stesso. Pertanto, ad esempio, `out.1` potrebbe dover essere confrontato con `check/out.5`. L'output del grader mostra di volta in volta quali directory vanno confrontate.

Nota bene bis: tutti i test vengono avviati con `valgrind`, per controllare che non ci siano errori nella gestione della memoria.

Esercizio 1

Scrivere un programma C che implementi un `ls` modificato (nel seguito, quando si parla di "file" si intende un file o una directory). In particolare, il programma dovrà avere la seguente sinossi:

`1 [options] [files]`

dove le opzioni sono le seguenti (si consiglia l'uso della funzione `getopt`):

- `-d`
- `-R`
- `-l mod`

Se invocato con la riga di comando `./1` *opts dirs*, il programma deve restituire lo stesso output del comando `ls -1 --color=never` *opts dirs*. Come eccezione, nel caso dell'opzione `-l`, vanno stampati solo i permessi, l'hard link count, la dimensione ed il nome (se `mod` vale 0), oppure solo i permessi ed il nome (se `mod` ha un qualsiasi altro valore); come separatore tra questi campi, usare sempre il tab. Se il file è un link simbolico, va mostrata la destinazione, come con l'opzione `-l` di `ls`. Per quanto riguarda la riga "total" delle directory, occorre tenere conto che il suo valore si ottiene sommando la dimensione di ciascun file (eventuali sottodirectory contano come file speciali, senza guardarne il contenuto), *arrotondata* al multiplo di x bytes superiore; alla fine, occorre dividere per il valore della variabile d'ambiente `BLOCKSIZE` (se non data, il valore di tale variabile è 1024). Il valore di x è quello ritornato da `stat -f --format=%s`. Diversamente dal comando `ls` di sistema, assumere che `BLOCKSIZE` non contenga suffissi come `kB`, `kiB` eccetera. I link simbolici non partecipano al conto totale.

Si possono manifestare solamente i seguenti errori.

- Uno dei file f dati come argomento non esiste. Il programma dovrà continuare con l'elaborazione, scrivendo su standard error lo stesso errore del comando `ls` di sistema (nota: gli errori di sistema iniziano con `ls:`, da sostituire con il nome del programma attualmente in esecuzione). Diversamente dal comando `ls` di sistema, al termine dell'esecuzione, l'exit status dovrà essere il numero di file non corretti passati.
- Viene data un'opzione non corretta. Il programma dovrà allora terminare con exit status 20 (senza scrivere nessun output), e scrivendo su standard error `Usage:` p `[-dR] [-l mod] [files]`, dove p è il nome del programma stesso.
- Fallisce una qualsiasi altra system call. Il programma dovrà allora terminare con exit status 100 (senza scrivere nessun altro output), e scrivendo su standard error `System call` s `failed because of` e, dove e è la stringa di sistema che spiega l'errore ed s è la system call che ha fallito.

In tutti gli altri casi, l'exit status dev'essere 0.

Attenzione: non è permesso usare le system call `system`, `popen`, `exec` e `sleep`. Il programma non deve scrivere nulla sullo standard error, a meno che non si tratti di uno degli errori descritti sopra. Per ogni test definito nella valutazione, il programma dovrà ritornare la soluzione dopo al più 10 minuti.

Suggerimento: usare `scandir`.

Esempi

Da dentro la directory `grader.2`, dare il comando `tar xfzp all.tgz input_output.1 && cd input_output.1`. Ci sono 6 esempi di come il programma `./1/1` possa essere lanciato, salvati in file con nomi `inp_out.`i`.sh` (con $i \in \{1, \ldots, 6\}$). Per ciascuno di questi script, la directory di input è `inp.`i. La directory con l'output atteso è `check/out.`i. La directory `check/out_tmp.`i contiene dei log di esempio di `valgrind`. Quelli prodotti dalla soluzione proposta dovranno essere simili a questi, ovvero non contenere messaggi d'errore (questo controllo viene fatto in `inp_out.`i`.sh`).

Esercizio 2

Scrivere due programmi C, un client (`2.client.c`) ed un server (`2.server.c`), che comunichino tramite named pipes. Più in dettaglio, il client dovrà avere i seguenti argomenti (nell'ordine dato):

- due nomi di named pipe n_r, n_w;
- una stringa s;
- un intero f.

Il server, invece, dovrà avere due argomenti: due nomi di named pipe n_r, n_w.

Il server dovrà creare le named pipe, se non esistono. Ogni riga letta dalla pipe n_r va intesa come una richiesta, che deve essere servita eseguendo il programma `/bin/sed`, cui dovrà passare, come programma da eseguire, la stringa s data input al client. L'input di `sed` sul server deve essere costituito da quanto viene letto dal client sul file descriptor f, nuovamente da assumere come già aperto al momento di chiamare il client. Il programma `sed` scriverà una risposta su standard output e/o su standard error, su una o più righe. Tale risposta, se su standard output, va rimandata al client tramite la named pipe, senza alcuna modifica. Se invece la risposta di `sed` è su standard error, occorrerà prefiggere ogni riga con la scritta `ERROR`n`:`, dove n è il numero progressivo della riga di errore. Dopodiché, il risultato va mandato sia al client su n_w, sia sul file descriptor 3 (da assumersi come già aperto) del server stesso. È possibile assumere che le risposte di `sed` contengano solo caratteri ASCII standard, e che alla fine della risposta di `sed` ci sia una andata a capo (sia su standard output che su standard error).

Il server potrà essere terminato se, come risultato dell'esecuzione di una richiesta di un client, ottiene come risposta da `sed`, su standard output, la stringa `EXIT`. In tal caso, l'exit status dovrà essere 155. Inoltre, il server potrà servire, sulle stesse named pipe, una sola richiesta alla volta.

Il client, invece, dovrà mandare al server, tramite n_w, tutto quanto letto dal file descriptor f (sempre da assumere come già aperto). Ogni riga ricevuta dal server (su n_r) va scritta sul corrispondente stream del client: su standard output se il server l'aveva ricevuta dallo standard output di `sed`, e sullo standard error altrimenti.

Si possono manifestare solamente i seguenti errori (in seguito a tali errori, è sempre necessario cancellare le named pipe):

- Il server non viene avviato con i 2 argomenti richiesti. Il programma dovrà allora terminare con exit status 10 (senza eseguire alcuna azione), e scrivendo su standard error `Usage:` p `piperd pipewr`, dove p è il nome del programma stesso.
- Una delle pipe n_r, n_w passate al server non esiste, ma non è possibile crearla. Il programma dovrà allora terminare con exit status 40 (senza eseguire alcuna azione), e scrivendo su standard error `Unable to create`

`name pipe` n `because of` e, dove e è la stringa di sistema che spiega l'errore e n il nome della pipe che ha causato l'errore.

- Una delle pipe n_r, n_w passate al client non esiste. Il programma dovrà allora terminare con exit status 80 (senza eseguire alcuna azione), e scrivendo su standard error `Unable to open named pipe` n `because of` e, dove e è la stringa di sistema che spiega l'errore e n il nome della pipe che ha causato l'errore.

- Uno degli argomenti n_r, n_w passati al server o al client esiste, ma non è una pipe. Il programma dovrà allora terminare con exit status 30 (senza eseguire alcuna azione), e scrivendo su standard error `Named pipe` n `is not a named pipe`, dove n è il nome della pipe che ha causato l'errore.

- Il client non viene avviato con i 3 argomenti richiesti. Il programma dovrà allora terminare con exit status 20 (senza eseguire alcuna azione), e scrivendo su standard error `Usage:` p `piperd pipewr sed_program fd`, dove p è il nome del programma stesso.

- Fallisce una qualsiasi altra system call. Il programma dovrà allora terminare con exit status 100 (senza eseguire alcun'altra azione), e scrivendo su standard error `System call` s `failed because of` e, dove e è la stringa di sistema che spiega l'errore ed s è la system call che ha fallito.

Attenzione: non è permesso usare le system call `system`, `popen` e `sleep`. I programmi non devono scrivere nulla sullo standard error, a meno che non si tratti di uno dei casi esplicitamente menzionati sopra (errori nelle opzioni). Per ogni test definito nella valutazione, i programmi dovranno ritornare la soluzione dopo al più 10 minuti.

Suggerimento: ovviamente, il client deve inviare al server il suo secondo argomento s, di modo che il server lo possa passare a `sed`. Per distinguere s dal resto dell'input, conviene usare un mini-protocollo: prima il client invia la lunghezza di s poi s stesso. In tal modo, il server sa quando ha finito di leggere s e avviare `sed`, passandogli come input ciò che legge dalla pipe.

Suggerimento bis: attenzione ad evitare stalli all'avvio di server e client; anche qui un mini-protocollo può aiutare.

Esempi

Da dentro la directory `grader.2`, dare il comando `tar xfz all.tgz input_output.2 && cd input_output.2`. Ci sono 6 esempi di come i programmi `./2/2.server` e `./2/2.client` possano essere lanciati, salvati in file con nomi `inp_out.`i`.sh` (con $i \in \{1, \ldots, 6\}$). Per ciascuno di questi script, la directory di input è `inp.`i. La directory con l'output atteso è `check/out.`i. La directory `check/out_tmp.`i contiene dei log di esempio di `valgrind`. Quelli prodotti dalla soluzione proposta dovranno essere simili a questi, ovvero non contenere messaggi d'errore (questo controllo viene fatto in `inp_out.`i`.sh`).

Esercizio 3

Scrivere un programma C di nome `3.c`, in grado di leggere e modificare file binari con un certo formato. Più in dettaglio, il programma dovrà prendere 5 argomenti: il nome di un file f_{in}, il nome di un file f_{out}, una stringa s e due numeri positivi i_1, i_2. Il file f_{in} è un file binario di un testo "offuscato", organizzato come segue: due interi positivi a 4 byte n_1, n_2, seguiti da almeno $n_1 + n_2$ bytes. I primi $8 + n_1$ bytes sono codici ASCII standard del testo memorizzato; i bytes che vanno dal $8 + n_1$ al $8 + n_1 + n_2$ sono codici ASCII complementati bit a bit, e i restanti sono di nuovo standard. Pertanto, per ricomporre il testo non offuscato, occorre leggere normalmente i caratteri da $8 + 1$ a $8 + n_1$, complementare bit-a-bit quelli da $8 + n_1 + 1$ a $8 + n_1 + n_2$, e leggere di nuovo normalmente i restanti. Il programma dovrà eseguire il comando `/usr/bin/gawk` s, facendo in modo che lo standard input di tale comando sia costituito dal contenuto del file "deoffuscato" (ovvero, ricostruendo il testo come descritto sopra). Il programma dovrà poi prendere il risultato del comando sopraindicato, "rioffuscarlo" e scrivere il risultato di tale "rioffuscamento" su f_{out}. Il rioffuscamento va effettuato usando i_1, i_2 come nuovi n_1, n_2. Il risultato è da intendersi come le scritte di `awk` su standard output, seguite da quelle su standard error.

Si possono manifestare solamente i seguenti errori:

- Il programma `3` non viene avviato con gli argomenti richiesti. Il programma dovrà allora terminare con exit status 10 (senza eseguire alcuna azione), e scrivendo su standard error `Usage:` p `filein fileout awk_script i1 i2`, dove p è il nome del programma stesso.

- Il file f_{in} non esiste o non è accessibile in lettura. Il programma dovrà allora terminare con exit status 20 (senza eseguire alcuna azione), e scrivendo su standard error `Unable to open file` f_{in} `because of` e, dove e è la stringa di sistema che spiega l'errore.

- Il file f_{in} letto da `3` non è ben formattato: non contiene almeno 8 bytes, oppure non contiene almeno $n_1 + n_2 + 8$ bytes. In questo caso, `3` deve terminare con exit status 30, generando un file di output di dimensione 0 e scrivendo su standard error `Wrong format for input binary file` f_{in}.

- La risposta di `awk` non ha almeno $i_1 + i_2$ bytes. Allora, il file andrà creato con $i_1 = i_2 = 0$ (quindi, senza offuscamento ma con i primi 8 bytes speciali), e `3` devrà terminare con exit status 80.

- Il file f_{out} non esiste o non è accessibile in scrittura. Il programma dovrà allora terminare con exit status 70 (senza eseguire alcuna azione), e scrivendo su standard error `Unable to open file` f_{out} `because of` e, dove e è la stringa di sistema che spiega l'errore.

- Fallisce una qualsiasi altra system call. Il programma dovrà allora terminare con exit status 100 (senza eseguire alcun'altra azione), e scrivendo

su standard error `System call` s `failed because of` e, dove e è la stringa di sistema che spiega l'errore ed s è la system call che ha fallito.

Attenzione: non è permesso usare le system call `system`, `popen` e `sleep`. Il programma non deve scrivere nulla sullo standard error, a meno che non si tratti di un errore nelle opzioni da riga di comando come descritto sopra. Per ogni test definito nella valutazione, il programma dovrà ritornare la soluzione dopo al più 10 minuti.

Esempi

Da dentro la directory `grader.2`, dare il comando `tar xfz all.tgz input_output.3 && cd input_output.3`. Ci sono 6 esempi di come il programma `./3/3` possa essere lanciato, salvati in file con nomi `inp_out.`i`.sh` (con $i \in \{1, \ldots, 6\}$). Per ciascuno di questi script, i file di input sono nella directory `inp.`i. La directory con l'output atteso è `check/out.`i. La directory `check/out_tmp.`i contiene dei log di esempio di `valgrind`.

Attenzione: se viene lanciato il grader, la corrispondenza tra input ed output viene persa: ad `out.`i corrisponde un `check/out.`i' con $i \neq i'$. Fare riferimento a ciò che scrive il grader stesso per trovare la giusta corrispondenza input-output.