Progetti del corso di Laboratorio Sistemi Operativi

1 Una shell

Lo scopo di questo progetto è creare una shell, che sia usabile e che sia in grado di supportare redirect e pipe. Quando opportuno deve essere in grado di eseguire comandi esterni ed invece, deve poter usare un "sano" default interno (che sarà dettagliato nel corso di questo documento).
Lo scopo principale del progetto è di familiarizzare lo studente con le strutture del filesystem, l'esecuzione di comandi dall'interno del listato C (con i conseguenti problemi di sicurezza), con il sistema, in generale.

1.1 Descrizione del progetto

Lo studente in questo progetto scriverà una shell, nominata *feshell*.

La shell deve essere in grado di ricevere comandi, parsarli e capire cosa farne. Nel caso in cui il comando non sia definito come "interno" la shell deve usare un opportuno comando della famiglia exec (si veda 'man 3 exec') ed eseguire in quell'ambiente il comando richiesto. L'unico comando da definire come interno è 'ls'. In questo caso la shell scritta dallo studente non deve usare il binario tipicamente in /usr/bin ma piuttosto reimplementare, come libreria linkata alla shell o come funzioni definite nel tree della shell stessa, una propria versione di ls.

Non è importante che la versione della shell e di ls realizzate dallo studente siano in grado di supportare *tutte* le opzioni che bash ed ls sono in grado di supportare. Per alcune opzioni, però è necessario che lo studente scriva il codice necessario. Queste sono:

1. l'equivalente della opzione '-l' di ls canonico. Questo mostra tutte le informazioni relative ai file (permessi, data di ultimo accesso o modifica, owner e gruppo, nome)
2. l'equivalente della opzione '-a' di ls canonico. Questo mostra anche i file normalmente nascosti (e pertanto la versione dello studente di ls deve rispettare la convenzione secondo la quale i file che iniziano con un punto (.) non devono altrimenti essere mostrati)
3. l'equivalente della opzione '-t' di ls canonico. Questa opzione ordina i file/directory nel listato secondo il tempo di creazione/modifica: dal piu' recente al piu' vecchio
4. l'equivalente della opzione '-h' di ls canonico. Questa opzione usa suffissi per le unità: KB per KiloByte, MB per MegaByte, ecc. e questo dovrebbe diminuire la lunghezza in cifre del numero mostrato per la dimensione a 3 caratteri o meno.
5. deve supportare la combinazione di più opzioni, esempio '-lat'.

Inoltre, la colorazione dell'output è opzionale (per esempio un colore per i file, secondo il tipo, ed un colore diverso per le directory).

feshell deve essere in grado di supportare la redirezione (<e >e ») e le pipe (|).

Questo supporto deve essere sostenuto anche in casi misti: e.g. “ps aux | grep ciccio > processiDiCiccio”.

2 Web Server

Lo scopo di questo progetto è creare un server web, in grado di fornire su socket pagine html statiche o file, per esempio immagini.
L'obiettivo principale del progetto è di familiarizzare lo studente con tecniche di programmazione concorrente.
Obiettivo secondario, ma comunque necessario al completamento del progetto è la profilazione/benchmarking del server scritto, per capire:

- quali sono i principali colli di bottiglia a runtime e quali sono le possibili cure,
- le capacità, in termini di reazione, del server sotto stress.

Può essere considerato un altro obiettivo secondario, qualora lo studente decidesse di volerlo perseguire, la familiarizzazione con i principali rudimenti di sicurezza.

La valutazione del progetto avviene durante una relazione orale che lo studente svolge all'esame, in cui gli verrà chiesto di presentare il suo codice, i suoi risultati e di essere in grado di discuterli criticamente. Può anche succedere che, durante la discussione, vengano proposte modifiche. In tal caso lo studente deve dimostrare, nei limiti del ragionevole, di essere in grado di applicare tali modifiche durante l'esame.

2.1 Specifiche del server

Il server deve essere in grado di rispondere ad una chiamata GET fornendo il documento richiesto, oppure, se nulla è specificato, il file index.html.

Se il file richiesto non è stato trovato il server deve ritornare un codice di errore, 404, oppure, nel caso in cui il server non sia in condizione di rispondere - per esempio per l'eccessivo carico - rispondere con codice 500 (Internal Server Error), se ne ha possibilità.
Idealmente la connessione dovrebbe funzionare così:

- il client si collega al socket su cui il server è in ascolto
- una volta avvenuto l'accept il server legge ciò che il client chiede (ed il client chiede qualcosa). Per esempio

  GET / HTTP/1.1
  host: hostname

  (da notare l'ultima riga vuota)
  seguendo queste specifiche: https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
- il server risponde con un flusso che corrisponde al contenuto del file da servire.

Alcune note:

- il server giri in modalità daemon: all'avvio si esegua una fork, proceda il figlio nell'esecuzione mentre il padre viene fermato, restituendo così controllo alla shell che lo ha chiamato
- si controlli sempre il valore di ritorno di funzioni di libreria, possibilmente usando delle funzioni wrapper che permettano la facile lettura del codice

- si controlli sempre quello che il client invia sul socket. Se non ha senso, nel senso che non è la richiesta di un documento conosciuto, si reagisca di conseguenza
- il server giri su una porta "alta", come detto a lezione, per evitare di dover avere privilegi di super utente

2.2 Struttura generale

L'architettura generale del server sia fatta in questo modo (sostanzialmente un produttore-consumatore):

1. un thread master, che detiene il socket e sta in attesa dei client.

2. una serie (un numero massimo $\mathcal{N}$, che è passato dalla linea di comando che esegue il server) di thread worker che, all'accept, ricevono il socket, aprono il file richiesto (se non è specificato si apra index.html) e lo inviano sul socket

3. finita la transazione il thread istanziato esce

È chiaramente arbitrario il modo in cui le richieste vengono servite, in base alla policy di scheduling che viene selezionata. Si usi come base uno scheduling di tipo FCFS. Esiste la possibilità di cambiare questa policy, come descritto nella apposita sezione.

Si eviti, perentoriamente, di eseguire fork successivamente alla fase di startup (cioè, un fork va bene all'inizio, ma poi basta).

Sarebbe interessante poter sperimentare a fare pooling dei thread, eseguendo sempre al più $\mathcal{C}$ thread, dove $\mathcal{C}$ è il numero dei core che sono presenti sul computer. Idealmente in questo caso ci sono $\mathcal{C}$ threads già istanziati, che ricevono istruzioni su cosa fare ogni volta che ce ne è il bisogno.

Si veda: https://github.com/Pithikos/C-Thread-Pool per una libreria in C che implementa i thread pool.

2.3 Scheduling

Come detto nella sezione precedente è possibile cambiare l'algoritmo di scheduling, implementandolo come alternativa (da notare che comunque si deve avere FCFS disponibile nel codice ed a runtime).

Una possibile idea di modifica è quella di servire il documento più breve prima (una sorta di SJF, ma qui per Shortest intendiamo il documento più breve).

Chiaramente questo richiede un pre-processing della richiesta che arriva al server prima che questa vada al thread worker, ed il punto giusto per implementare questo pre-processing e' il thread master: bisognerà ottere la dimensione del file richiesto e usarlo per ordinare le richieste.
Questo permette anche di evitare di passare ad un worker un filename non esistente, risultando in 404 più veloci.

Questo modo di schedulare i thread evidentemente pone un problema di starvation (perchè?). Si implementi questo scheduler e si verifichi che si può presentare un problema di starvation, mostrando i risultati nella relazione e nella discussione.

2.4 Benchmarking

Si usi un tool come ab o siege per capire quanto bene funziona il server web che si è creato.
Questi tool possono essere trovati qui:

- https://www.joedog.org/siege-home/
- https://httpd.apache.org/docs/2.2/en/programs/ab.html

Insieme a questi si può capire cosa sta succedendo a run time durante un test e quindi capire quale frammento di codice sta usando tempo e risorse in modo non ottimale con gprof
https://sourceware.org/binutils/docs/gprof/index.html

2.5 Aggiunte

È possibile implementare alcune migliorie al server web appena descritto. Si discuta brevemente nella relazione quali sono e, volendo raggiungere un punteggio superiore, li si implementi (o se ne implementino alcuni che si sono ritenuti più importanti).
Si analizzino i problemi di sicurezza che si possono presentare nel server, e come si possono ovviare (se si possono ovviare).

3 Spiaggia

Lo scopo di questo progetto è creare un sistema client server per la gestione delle occupazioni dei lettini di una spiaggia, in grado di fornire informazioni sulla disponibilitàdegli ombrelloni, e impegnarli o disdirli. L'architettura è di tipo client/server ed allo studente è chiesto di creare il server ed il client.
L'obiettivo principale del progetto è di familiarizzare lo studente con socket e programmazione concorrente, oltre che con architetture client/server.

La valutazione del progetto avviene durante una relazione orale che lo studente svolge all'esame, in cui gli verrà chiesto di presentare il suo codice, i suoi risultati e di essere in grado di discuterli criticamente. Può anche succedere che, durante la discussione, vengano proposte modifiche. In tal caso lo studente deve dimostrare, nei limiti del ragionevole, di essere in grado di applicare tali modifiche durante l'esame.

3.1 Specifiche del server

Il server è strutturato come segue:

- un thread master, che all'avvio legge un file di configurazione in cui sono descritti gli ombrelloni disponibili ed il loro stato. Quando il thread master viene chiuso (anche tramite segnali di tipo SIGTERM, SIGINT, SIGQUIT) è responsabilità del thread master stesso garantire che i dati sul disco siano sempre aggiornati all'ultima operazione avvenuta in memoria.

  Ci sono chiaramente diversi approcci per raggiungere questa consistenza: se ne scelga uno e lo si implementi.

  Si sia pronti a discutere anche gli altri modi

- alcuni thread worker: questi thread sono responsabili di:
  - accettare il socket ottenuto dalla accept() e continuare la comunicazione con il client
  - decodificare il messaggio ricevuto dal client
  - verificare quello che il client chiede
  - dare risposta al client e contestualmente modificare le disponsabilità

Evidentemente questa non è l'unica architettura possibile. Se si decide di implementare una diversa architettura, si motivi durante la relazione orale, il perchè. Si consideri per esempio la possibilità di usare un thread pool: https://github.com/Pithikos/C-Thread-Pool

3.2 Gestione dei dati

L'architettura proposta favorisce chiaramente l'uso di un database per la gestione dei dati. Per questo progetto comunque, preferiremo non usare un database "vero" per non appesantire la scrittura del codice. Invece, gestiremo le informazioni necessarie al server in modo più semplice, usando un file.

Mentre non viene data alcuna specifica per il formato in cui scrivere sul disco le informazioni, esse devono essere:

- numero ombrellone
- stato (libero, occupato, temporaneamente prenotato)

- data di "scadenza" (cioe' quando si libera, se è occupato)

Il consiglio è di tenere il formato semplice.

Alternativamente si può pensare di usare un sistema di journaling per la gestione dei dati. Si veda https://it.wikipedia.org/wiki/Journaling e si chieda a lezione se si è interessati a maggiori dettagli. Idealmente si potrebbe scrivere un log con tutti i cambiamenti da applicare al file e ad intervalli regolari di tempo applicare i cambiamenti. Il vantaggio è che nel caso di incosistenza dello stato del file, basta riapplicare i cambiamenti che sono segnati nel log (questa tecnica per esempio è usata nei file system journaled, ma nulla vieta di applicarla ad altri casi).

evitare di apportare modifiche al protocollo se non strettamente necessarie

3.3 Protocollo di comunicazione client-server

Il client comunica con il server nel seguente modo, una volta stabilita la connessione al socket:

COMANDI DA IMPLEMENTARE

- il client chiede di poter occupare un ombrellone: trasmette "BOOK"
- il server risponde con "OK" nel caso in cui sia pronto per la prenotazione oppure con "NOK" nel caso in cui non lo sia. Nel secondo caso la comunicazione è terminata. Se non ci sono ombrelloni disponibili, il server risponde con "NAVAILABLE"
- il client procede a richiedere un ombrellone specifico: "BOOK $OMBRELLONE" dove $OMBRELLONE è un indentificativo valido per un ombrellone (dipende sostanzialmente da come vengono identificati gli ombrelloni nel server che si è creato)
- il server verifica la disponibilità. Se l'ombrellone è disponibile il server cambia lo stato dell'ombrellone in "temporaneamente prenotato" e comunica la disponibilità al client: "AVAILABLE". Se non è disponibile, il server risponde con "NAVAILABLE" e chiude la connessione
- quando il client riceve "AVAILABLE" decide se vuole confermare la prenotazione. In tal caso risponde con "BOOK $OMBRELLONE DATE" dove DATE è la data di scadenza della prenotazione. Se invece non vuole confermare la prenotazione trasmette "CANCEL"
- nel caso in cui il client desideri prenotare un ombrellone nel futuro (attenzione alle date), può trasmettere "BOOK $OMBRELLONE DATESTART DATEEND". Il server verifica la disponibilità e risponde nel modo convenzionale indicato poco sopra

Oltre a questo flusso di comunicazione esiste anche la possibilità di mandare altri messaggi al server, per il client:

- il client può inviare una richiesta di tipo "AVAILABLE" per sapere se ci sono ombrelloni disponibili, e quanti sono. Il server risponde "NAVAILABLE" nel caso in cui non ci siano ombrelloni disponibili e con "AVAILABLE $NUMERO" dove $NUMERO è il numero di ombrelloni disponibili
- il client può inviare una richiesta del tipo "AVAILABLE 1" per richiedere tutti gli ombrelloni disponibili nella prima fila. Ovviamente questo significa che nel formato in cui queste informazioni sono salvate nel file questa informazione deve essere registrata in qualche modo
- il client può chiedere di cancellare una prenotazione inviando "CANCEL $OMBRELLONE". Nel qual caso il server risponde con "CANCEL OK" e ripone l'ombrellone $OMBRELLONE in stato disponibile

3.4 Specifiche del client

Si scriva un piccolo client che esegua la connessione e trasmetta un singolo comando, da accettarsi alla linea di comando.

Per chiarezza: il client deve essere chiamato a linea di comando con un argomento che sia l'argomento da trasmettere al server. Da quel punto si gestisca il flusso di comunicazione in modo interattivo.

Volendo è anche possibile usare telnet come client, collegandosi alla porta del server direttamente. Nel qual caso si veda la seguente sezione a riguardo dell'uso di expect.

3.5 Testing

Utile per verificare il corretto funzionamento del codice

Si può automatizzare il testing del server e del client usando un tool di scripting chiamato expect: http://expect.sourceforge.net/

Expect permette di usare uno linguaggio di scripting (derivato da tcl) per interagire con eseguibili che sono interattivi, come il client descritto nella sezione precedente.