Prova finale
(Progetto di Algoritmi e Strutture Dati)

SimpleFS - un semplice filesystem

20 giugno 2017

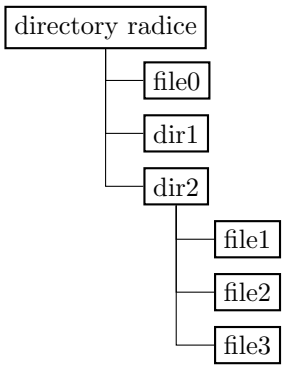
1 Sommario

L'obiettivo del progetto di Algoritmi e Strutture Dati è implementare un semplice filesystem gerarchico con stoccaggio unicamente in memoria principale. Un filesystem gerarchico organizza le risorse secondo una struttura ad albero, ed identifica ogni risorsa univocamente tramite il percorso che la collega alla radice. Le risorse contenute in un filesystem gerarchico possono essere *file* o *directory*. Entrambi sono dotati di un nome, rappresentato come una stringa di caratteri. Per i primi è unicamente possibile essere inseriti come foglie dell'albero, mentre le seconde possono apparire sia come foglie che come nodi intermedi. La radice dell'albero è convenzionalmente una directory, detta appunto directory radice. Solo i nodi file possono contenere dei dati, rappresentati come una sequenza di byte, mentre le directory non hanno dati associati. Tutti i nodi dell'albero possono contenere metadati, ma ai fini di questo progetto, solo le directory ne contengono. I metadati della directory sono i nomi dei suoi diretti discendenti.

Il programma che implementa il filesystem riceverà un diario delle azioni da compiere dallo standard input e stamperà il risultato delle stesse su standard output. Il programma deve essere implementato in `C` standard, con il solo ausilio della libreria standard (`libc`) e del runtime di base. Il funzionamento del programma prevede che esso legga una riga del diario delle azioni, effettui l'azione corrispondente sulla rappresentazione interna del filesystem da esso mantenuta, e scriva il risultato su standard output prima di procedere alla successiva azione (l'esecuzione delle azioni è completamente sequenziale).

2 Formato dell'ingresso

I percorsi del filesystem sono rappresentati con l'usuale sintassi UNIX: un percorso è dunque la sequenza di nomi di risorse che dalla directory radice raggiunge la risorsa identificata dal percorso. I nomi sono separati dal carattere separatore di percorso `/`). A titolo di esempio, si consideri il seguente filesystem:



Il percorso che identifica la risorsa file0 è `/file0`, quello che identifica file3 è `/dir2/file3`.

Valgono le seguenti restrizioni:

- i nomi delle risorse sono alfanumerici e possono essere lunghi al massimo 255 caratteri;
- l'altezza massima dell'albero è 255;
- il numero massimo di figli di un nodo è 1024.

Il programma riceve uno tra i seguenti comandi per ogni riga del file di diario dato in input, dove ⟨`percorso`⟩ indica un generico percorso e ⟨`nome`⟩ una stringa alfanumerica di al massimo 255 caratteri.

- `create` ⟨`percorso`⟩: Il comando ha come effetto la creazione di un file vuoto, ovvero senza dati associati, all'interno del filesystem. Stampa in output l'esito "ok" se il file è stato creato regolarmente, "no" se la creazione del file non è andata a buon fine (ad esempio, se si tenta di creare un file in una directory inesistente, o se si eccedono i limiti del filesystem).
- `create_dir` ⟨`percorso`⟩: Crea una directory vuota all'interno del filesystem. Stampa in output l'esito "ok" se la directory è stata creata regolarmente, "no" se la creazione non è andata a buon fine.
- `read` ⟨`percorso`⟩: Legge per intero il contenuto di un file, stampando in output "contenuto" seguito da un carattere spazio e dal contenuto del file se il file esiste oppure stampa "no" se il file non esiste.
- `write` ⟨`percorso`⟩ ⟨`contenuto`⟩: Scrive, per intero, il contenuto di un file, che deve esistere già (se il file aveva già del contenuto, questo viene sovrascritto); stampa in output "ok" seguito dal numero di caratteri scritti se la scrittura è andata a buon fine, "no" in caso contrario. Il parametro ⟨`contenuto`⟩ ha la forma di una sequenza di caratteri alfanumerici e spazi delimitata da doppie virgolette. Esempio:
  `write /poems/jabberwocky "It was brillig and the slithy toves"`
- `delete` ⟨`percorso`⟩: Cancella una risorsa, stampa in output l'esito ("ok"-"no"). Una risorsa è cancellabile solamente quando non ha figli.
- `delete_r` ⟨`percorso`⟩: Cancella una risorsa e tutti i suoi discendenti se presenti. Stampa in output l'esito ("ok"-"no").

- `find` ⟨`nome`⟩: Cerca la posizione della risorsa ⟨`nome`⟩ all'interno del filesystem. Stampa in output "ok" seguito dal percorso della risorsa per ogni risorsa con nome corretto trovata. Le stampe delle risorse devono essere effettuate in ordine lessicografico del percorso che va stampato. I caratteri di separatore di percorso vanno considerati nell'ordinamento, il confronto per l' ordinamento va quindi fatto tra intere stringhe rappresentanti percorsi. Nel caso nessuna risorsa sia trovata, il comando stampa "no".

- `exit`: Termina l'esecuzione del gestore di risorse. Non stampa nulla in output.

3 Requisiti di complessità temporale

Detti l la lunghezza di un percorso, d il numero di risorse dell'intero filesystem, $d_{percorso}$ il numero di risorse figlie di quella specificata dal percorso,e f il numero di risorse trovate da una ricerca, le complessità temporali attese delle primitive specificate nella precedente sezione sono le seguenti.

comando	complessità
`create`	$\mathcal{O}(l)$
`create_dir`	$\mathcal{O}(l)$
`read`	$\mathcal{O}(l + \|contenutofile\|)$
`write`	$\mathcal{O}(l + \|contenutofile\|)$
`delete`	$\mathcal{O}(l)$
`delete_r`	$\mathcal{O}(d_{percorso})$
`find`	$\mathcal{O}(d + f^2)$