RELAZIONE DI LABORATORIO

Sistemi Operativi 2017/2018

Abstract

Viene analizzata l'implementazione del progetto "Swordx", un' applicazione di sistema sviluppata in C per il conteggio delle occorrenze di parole in file testuali.

Belenchia Matteo
matteo.belenchia@studenti.unicam.it

Giacché Alessandro
alessandro.giacche@studenti.unicam.it

Contenuti

Introduzione

Il progetto si articola in 5 file sorgenti, di cui uno, swordx.c, è il programma principale mentre gli altri sono moduli contenenti le strutture dati necessarie.

I file stack.c e threadIdStack.c sono due simili implementazioni di uno stack, l'uno atto a contenere pathnames e l'altro per contenere puntatori a threads. I due file presentano funzioni molto simili con la sola differenza del tipo di dato contenuto in ogni nodo. Tale duplicazione di codice si sarebbe potuta evitare utilizzando dei puntatori a void.

Il file trie.c contiene l'implementazione di un albero digitale, o trie, che è la struttura dati che si occupa di mantenere in memoria le parole lette dai files.

Il file BST.c implementa un albero binario di ricerca collegato strettamente ai nodi di un trie, da cui dipende. Questo collegamento stretto si concretizza nel fatto che il contenuto di un nodo di tale albero non è altro un puntatore a un corrispettivo nodo di un trie.

Tutti i moduli sopracitati hanno un corrispettivo header file che funge da interfaccia verso il programma utilizzatore e contenente le definizioni delle strutture dati e le dichiarazioni di funzioni accessibili dall'esterno.

Il programma utilizzatore è ovviamente swordx.c, il quale invece non è dotato di header file e le cui dichiarazioni di funzioni sono localizzate nel file stesso prima delle loro definizioni. Il programma svolge un conteggio delle parole contenute nei files passati in input, offrendo una serie di funzioni aggiuntive osservabili mediante il comando --help. L'esecuzione avviene in gran parte single-threaded, con un multi-threading limitato all'analisi contemporanea dei files di input, a ognuno dei quali viene assegnato un thread. Una migliore implementazione di ciò potrebbe essere stata la fusione dei files di input in un unico file (a livello logico o fisico) e l'uso dei threads per analizzare diversi settori di quest'ultimo.Una tale implementazione risolverebbe l'inconveniente della diminuzione del grado di parallelismo alla ricezione di files di diverse dimensioni: con l'implementazione attuale, mentre i files più lunghi vengono elaborati, gli thread rimangono in attesa.

Il progetto è corredato di un makefile per la compilazione e la risoluzione delle dipendenze.

Librerie

Tutte le librerie utilizzate fanno parte della libreria standard di C implementata da GNU, la glibc. Nei sistemi GNU/Linux non è perciò necessaria alcuna installazione aggiuntiva, mentre nel porting del programma su altri sistemi, come Windows, potrebbe essere necessario importare nel programma il codice sorgente di Gnulib, la libreria GNU per la portabilità del software su altri sistemi.

Le librerie ISO utilizzate sono *ctypes.h*, *stdio.h*, *stdlib.h* e *string.h*.
La libreria *ctypes*, in particolare, contiene delle funzioni utili a stabilire di che tipo sia un dato carattere e viene usata principalmente nella lettura dei caratteri nel file.

Le librerie POSIX utilizzate sono *dirent.h*, *sys/stat.h*, *glob.h* e *pthread.h*.
La libreria *dirent* permette la gestione delle cartelle; viene utilizzata per scandire i files e, eventualmente, le sottocartelle delle directory passate in input.
Stat viene inclusa per sfruttare la funzione *lstat* e distinguere, quindi, i tipi di file tra file regolare, cartella e link simbolico.
La libreria *glob* viene utilizzata per il globbing delle espressioni regolari nel caso queste facciano rifermento a --explude o --ignore mentre, per i file di input, si fa affidamento sul globbing effettuato dalla shell. *Pthread*, infine, è la libreria usata per la creazione e la gestione dei threads.
L'unica libreria né ISO né POSIX, ma presente unicamente in glibc, è getopt.h, che serve per la lettura di opzioni e i loro argomenti in input a linea di comando.

Makefile

La gestione della compilazione è affidata a *GNU Make*, la struttura delle directories è la seguente:

- In *src* sono presenti tutti i codici sorgente;
- *include* contiene gli headers;
- In *obj* vengono inseriti i codici oggetto generati.

Il *makefile* è composto da 5 variabili: *CC* (il compilatore), *CFLAGS* (i parametri di compilazione di tutti i moduli), *OBJDIR* (la directory dove vengono salvati i codici oggetto), *HEADIR* (la directory contenente gli headers) e *OBJ* (la lista dei codici oggetto da generare).

Vi sono poi 2 percorsi virtuali: src/*.c per i codici sorgente e include/*.h per gli headers.

Make (all) crea i codici oggetto senza eseguire il linking (opzione -c di gcc) utilizzando il codice sorgente e l'header di ogni file esclusi BST.o e swordx.o a cui saranno associati altri headers in quanto richiesti dai sorgenti stessi.

L'opzione *clean* è associata a una phony per evitare ambiguità nel caso in cui esista un file nominato, appunto, *clean*.

L'argomento -I$(HEADDIR) specifica la cartella dove sono inseriti gli headers mentre, nella fase di linking, viene passato l'argomento -pthread per abilitare l'utilizzo della libreria di gestione dei threads.

Trie.c

Questo modulo implementa un trie (anche noto come albero digitale) per il mantenimento in memoria delle parole lette dai file. Un trie è un tipo di albero di ricerca dove l'ordinamento dei nodi è ottenuto mediante chiavi di tipo stringa; la radice è la stringa vuota e ha un figlio per ogni possibile carattere del *CHARSET* (in questo caso si fa riferimento all'alfabeto inglese più le 10 cifre, altre implementazioni solite sono l'alfabeto latino o l'ASCII). Ad ogni nodo è quindi associata una data stringa corrispondente alla concatenazione di una lettera per ogni nodo attraversato nel cammino dalla radice al nodo stesso. Ogni nodo quindi contiene 3 dati:

- La stringa associata al nodo (`value`);
- Il numero di occorrenze di quella stringa nei files (`occurrencies`);
- Un array di puntatori ad altri nodi per ogni carattere del *CHARSET* (`children`);

Una implementazione alternativa e migliore consiste nel salvare nel nodo unicamente il carattere associatogli anziché l'intera stringa, mentre un'implementazione ancora migliore consisterebbe nel non salvare alcun carattere e ottenere la stringa valutando volta per volta l'indice del puntatore al quale si è acceduto. La complessità spaziale dell'implementazione corrente è $O\left(\sum_{i=0}^{m} \frac{n_i(n_i+1)}{2}\right)$ dove m è il numero di parole ed n_i è il numero di caratteri della parola i-esima.

Add

La funzione *add* aggiunge una data parola al trie, richiamando una seconda funzione (*_add*) ricorsiva che necessita di un parametro `level` rappresentante il livello di profondità dell'albero in cui ci si trova durante la ricorsione. La funzione *_add*, inizialmente, scorre l'albero selezionando il nodo corrispondente al carattere in posizione `level` e prosegue solo se il nodo non è NULL. In caso affermativo, se `level` è pari alla lunghezza della stringa, la stringa è già presente nel trie e il valore di `occurrencies` si incrementa di 1, altrimenti si incrementa `level` e si richiama *_add* con il nuovo valore di `level` e con il nodo corrente al posto della radice. Se invece si è incontrato un nodo NULL, allora la parola che si vuole inserire non è presente nel trie e viene chiamata un'altra funzione (*addRecord*), per aggiungerla.
Il flag serve per evitare di richiamare nuovamente *addRecord* al ritorno delle chiamate ricorsive.

AddRecord

La funzione crea uno o più nodi al fine di memorizzare una nuova parola nel trie. Il numero di nodi creati sarà pari alla differenza tra la lunghezza della stringa passata e `level`. Si assume che il nodo ricevuto come argomento sia l'ultimo nodo non *NULL* nella discesa nell'albero.
Viene creato un nuovo nodo che andrà a contenere:

- I primi `level` caratteri della parola (in `value`);
- Il valore 0 (in `occurrencies`);
- Un array di puntatori a *NULL* per ogni carattere del *CHARSET* (in `children`).

Il puntatore al nuovo nodo viene quindi salvato nella posizione consona (ottenuta attraverso la funzione `getIndex`) all'interno dell'array di puntatori del nodo passato come argomento.
Una volta creato il nodo e associato al nodo passato viene controllata l'uguaglianza tra `level` e la `strlen` della parola: se questi valori sono uguali, `occurrencies` viene incrementato di 1, altrimenti si incrementa `level` e si richiama *addRecord* passando il nuovo valore di `level` e il nodo appena creato come nodo di partenza.

Search

La funzione search restituisce 1 se la stringa è non *NULL* e presente nell'albero, 0 altrimenti. Si basa su una seconda funzione (`_search`) che necessita del parametro `level` per le chiamate ricorsive. La funzione si comporta in modo simile a *_add*: prende in considerazione il nodo corrispondente al carattere in posizione `level` nella stringa e controlla che questo non sia *NULL*. In caso affermativo, se `level` è uguale a `strlen` della parola, si ritorna 1, altrimenti si incrementa `level` e si richiama `_search` passando il nuovo valore di `level` e il nodo corrente come nodo di partenza.

Se non si ha una corrispondenza a un qualsiasi livello di ricorsione, si ritorna 0.

getIndex

Questa funzione, ottiene l'indice corrispondente nell'array di puntatori di ogni nodo di un carattere passato, sfruttando, laddove possibile, le proprietà del codice ASCII.
Se il carattere è una cifra, viene convertito in un indice da 0 a 9 (quindi i primi 10 puntatori di ogni nodo corrispondono alle 10 cifre), altrimenti si ottiene l'indice da 0 a 26 corrispondente alla lettera a cui si somma 10. Modificando in modo opportuno questa funzione e cambiando il valore di *CHARSET* si può facilmente modificare il set di caratteri supportati.

writeTrie

La funzione scrive il trie su un file indicato.
Se il valore di *occurrencies* è maggiore di 0, il nodo contiene una parola presente nei files di input e la funzione *writeNodeInformation* si prenderà carico di scrivere una nuova riga sul file.
Viene poi richiamato ricorsivamente *writeTrie* su ognuno dei puntatori contenuti nel nodo. Dato che i puntatori sono fatti corrispondere a caratteri ordinati alfabeticamente da *getIndex* (con prima le cifre da 0 a 9 e poi le lettere dell'alfabeto) e che un nodo è sempre lessicograficamente inferiore rispetto ai suoi figli, è assicurato che la scrittura su file avviene in ordine alfabetico.

BST.c

Questo modulo implementa un albero binario di ricerca associato al trie delle parole. Il contenuto di ogni nodo è costituito dai puntatori al figlio destro e sinistro e da un puntatore a un nodo di un trie contenente una parola valida. A causa di ciò questo modulo dipende da trie.c e nel suo header viene incluso trie.h.

La chiave che stabilisce l'ordinamento di quest'albero è il valore di `occurrencies` dei nodi corrispondenti alle parole del trie.

Le uniche funzioni di questo modulo sono l'aggiunta all'albero e la scrittura dell'albero su file, e il suo uso nel progetto è limitato all'implementazione della funzionalità --sortbyoccurrency.

addBST

La funzione aggiunge il nodo del trie passato al BST.
Se il nodo corrente è *NULL*, viene creato un nuovo nodo nel quale viene salvato il nodo del trie, altrimenti si visita il figlio sinistro o destro dell'albero, a seconda del valore di `occurrencies`, e si richiama ricorsivamente *addBST* usando come nodo di partenza il figlio selezionato.

writeTree

writeTree scrive il contenuto del BST sul file indicato mediante una visita simmetrica dell'albero, utilizzando `writeNodeInformation` (contenuta in *trie.c*) per la scrittura su file del campo `wordInfo`: un nodo del trie.

stack.c & ThreadIdStack.c

Vista l'estrema similitudine tra le due strutture (come approfondito nell'introduzione) queste verranno analizzate nella stessa parte di relazione.

Lo *stack* viene utilizzato per tenere traccia delle path delle cartelle e dei files da analizzare, il *ThreadIdStack* viene invece utilizzato per tenere traccia degli id dei threads in esecuzione.

L'implementazione consiste in un blocco iniziale (detto "sentinella") che non contiene alcun valore ma punta a un blocco che è l'effettiva testa dello stack. Ogni nodo è composto da un valore (*char** per lo *stack*, *pthread_t** per il *ThreadIdStack*) e un riferimento all'elemento successivo. Se si fosse usato un doppio puntatore si sarebbe potuto evitare l'utilizzo di un blocco aggiuntivo (il blocco sentinella).

createStack & createThreadIdStack

Queste funzioni allocano la memoria e inizializzano i valori del primo blocco (il blocco "sentinella") a NULL, poi ritornano lo stack.

push & threadIdPush

Push e *threadIdPush* creano un nuovo "nodo" nello stack e gli associano il valore del secondo argomento passato alla funzione (*str* per *stack* e *intid* per *threadIdPush*). *Push* alloca uno spazio per la stringa e copia il contenuto del secondo parametro all'interno dello spazio allocato mentre *threadIdPush* asserisce che l'elemento da inserire nello stack sia già stato allocato nell'heap e copia perciò il puntatore passato nel campo *tid*.

Una volta inizializzato il nodo, il primo elemento dello stack (cioè il *next* del nodo "sentinella") viene associato al *next* di quello appena creato e questo viene puntato dal *next* del nodo sentinella, diventando effettivamente la testa dello stack.

pop & threadIdPop

Se lo stack è vuoto, viene ritornato *NULL*, altrimenti viene restituito il valore del nodo successivo a quello sentinella, viene deallocata la memoria associata all'elemento e la testa dello stack diviene il nodo successivo a quello deallocato.

La deallocazione dell'elemento ritornato viene fatta gestire al chiamante.

visitStack & visitThreadsStack

Queste funzioni ricorsive mostrano il contenuto di ogni nodo dello stack (le *stringhe* per lo *stack* e il puntatore a *pthread_t* per il *ThreadIdStack*).

L'uso delle funzioni ricorsive è necessario in quanto, per leggere il valore di ogni nodo, si sfrutta la funzione *top* (o *topT*) passando ricorsivamente il *next* dell'elemento in analisi.

searchStack

La funzione ricorsiva *searchStack* (appartenente unicamente alla struttura dati *stack*) ricevuto uno *stack* e una *stringa* ritorna *1* se la stringa esiste nello stack, *0* altrimenti.
L'uso di una funzione ricorsiva è necessario in quanto, per leggere il valore di ogni nodo, si sfrutta la funzione *top*, passando ricorsivamente il *next* dell'elemento in analisi.

Swordx.c

main

La gestione degli argomenti a linea di comando avviene attraverso la libreria *getopt.h*, in particolare mediante la funzione *getopt_long_only*, che permette di accettare opzioni sia a singolo trattino che doppio trattino, e supporta opzioni a singolo trattino multi–carattere. Le opzioni multi–carattere sono gestite attraverso un array di *struct option*. Ogni struct contiene il nome dell'opzione, se richiede o no argomenti, un puntatore che non è necessario ai nostri scopi e il valore di ritorno della funzione *getopt_long_only* in caso di matching. Le opzioni singolo–carattere, invece, vengono gestite direttamente nella chiamata a *getopt_long_only*, la quale accetta *argc*, *argv*, una stringa rappresentante le opzioni a singolo–carattere (con i *due punti* che stanno a significare che l'opzione della lettera precedente richiede un argomento), l'array di *struct option* e un ultimo parametro a *NULL* che non ci interessa. La funzione restituisce il valore dell'opzione se viene incontrata una opzione valida in *argv* (il carattere stesso nel caso fosse un'opzione a singolo–carattere o il numero definito nelle struct option altrimenti) e *-1* quando non ci sono più corrispondenze. L'argomento di una opzione viene salvato nella variabile *extern optarg* mentre argomenti non relativi ad alcuna opzione sono permutati in fondo ad *argv* automaticamente. Una volta analizzate tutte le opzioni, in *argv* restano quindi da analizzare tutti i files e le cartelle considerati "di input" al programma. Le espressioni regolari sono gestite direttamente dalla shell.
Nel caso delle opzioni --min e --output, il valore viene salvato in una variabile, per --ignore e --explude, invece, il valore viene inserito in un apposito stack. Per le opzioni --recurisve, --follow, --alpha e --sortbyoccurrency viene effettuato un *OR* tra delle macro e una variabile *unsigned char* da 1 byte (inizialmente di valore 0).

Le macro sono definite nel modo seguente:

```
#define RECURSE_FLAG (1<<0) //00000001
#define FOLLOW_FLAG (1<<1) //00000010
#define ALPHA_FLAG (1<<2) //00000100
#define SBO_FLAG (1<<3) //00001000
```

Per sapere se un'opzione è attiva sarà sufficiente effettuare un *AND* tra il bit field e la macro interessata: l'espressione ritornerà un valore diverso da 0 se l'opzione è attiva.

I valori di --min e --output vengono salvati in un array *args*, anche se *NULL*, dato che successivamente possono essergli assegnati valori di default.

Una volta terminato di analizzare le opzioni, restano da recuperare gli argomenti di input. La variabile *extern optind* indica il prossimo indice da leggere in *argv*, tutti gli elementi di *argv* (da quell'indice fino ad *argc)* sono perciò argomenti di input che vengono salvati in un array *params*. Successivamente questo array (che può contenere sia file che cartelle) viene convertito in uno stack di soli file dalla funzione *arrayToStack*.

A questo punto viene analizzato l'argomento dell'opzione --ignore (anche nel caso sia *NULL*) trasformando lo stack dei suoi valori in uno stack di soli files e, successivamente, creando un trie di parole da ignorare che viene popolato dalla funzione *getBlacklist*, utilizzando quest'ultimo stack.

Viene quindi richiamata la funzione *execute* per l'analisi dei files di input considerando le varie opzioni passate attraverso l'array *args*, il trie delle parole da ignorare *ignoreTrie* e il bit field *flags*.

arraytoStack

Questa funzione converte un array di path, riferiti a file o cartelle, in uno stack di soli files. Richiede uno stack di files da escludere (per la funzionalità --explude) e il bit field *flags* del quale analizza i bit riferiti alle opzioni --recursive e --follow. Lo stack dei file da escludere viene risolto dalla funzione *expand* in uno stack di soli files.

Vengono successivamente utilizzate la struct e le macro definite da *stat.h* per distinguere i diversi tipi di file nell'array di input.

I files regolari vengono inseriti direttamente nello stack di output, mentre le cartelle vengono analizzate tramite la funzione *scan*, tenendo conto anche dello stack dei files da escludere e dei flags.

expand

Questa funzione, ricevuto uno stack di pathname, lo converte in uno stack analogo dove eventuali espressioni regolari vengono risolte in path corrispondenti. La funzione viene usata esclusivamente nel contesto di conversione dei pathnames per le opzioni --ignore ed --explude, permettendo di utilizzare espressioni regolari come argomenti, passandole tra doppi apici.
Ogni elemento dello stack di input viene passato alla funzione *glob* e, se l'espressione regolare è valida, viene analizzato l'array restituito da glob: se vi sono cartelle vengono aggiunte al risultato passandole alla funzione *scan* senza flags e con uno stack anonimo (che sarebbe dovuto essere lo stack di --explude), i files e links, vengono inseriti direttamente nello stack di output (i link vengono risolti attraverso *absPath*).

absPath

La funzione richiama *canonicalize_file_name* per restituire la path assoluta del file, risolvendo eventualmente i link simbolici.

scan

La funzione *scan*, presa la directory da scandire, lo stack dei files da ignorare (contenente percorsi assoluti) e il bit field (di cui analizzare i bit FOLLOW e RECURSIVE), inserisce gli indirizzi assoluti dei files all'interno dello stack passato.

Si asserisce che *path* contenga effettivamente una cartella e che non sia necessario un controllo esplicito.

Viene utilizzato uno stack per tenere traccia delle directory da analizzare, la prima cartella esaminata è quella passata alla funzione mentre le altre verranno aggiunte all'esecuzione del codice interno al primo ciclo *while*.

Le cartelle "." e ".." di ogni folder vengono ignorate, viene allocata una stringa *toInsert* che conterrà l'indirizzo assoluto del file da inserire in *files* e, se il file non è presente nello stack dei documenti da ignorare, viene utilizzata la funzione *lstat* per decidere dove inserire il percorso corrente: nello stack delle cartelle (nel caso fosse una directory), nello stack dei files (se fosse effettivamente un regular file) o seguire il percorso assoluto attraverso la funzione *absPath* (se fosse un link) ed effettuare nuovamente il controllo del tipo di file. Questo viene eseguito per tutti i files all'interno della cartella analizzata (che viene presa dallo stack *folders*).

Per la scansione dei files nella cartella viene utilizzata la libreria *dirent* che permette una veloce lettura di tutti i files in una data cartella (inserendoli in una *struct dirent*); per analizzare il file, invece, viene utilizzata *stat* poiché *dirent* non permette un accesso diretto al file ma richiede il passaggio della directory.

getBlacklist

Questa funzione popola un trie con le parole dei files contenuti nello stack passato in input.
Tale blocco è essenziale per la funzionalità --ignore: i percorsi dei files contenenti parole da ignorare sono contenuti nello stack di input per venire poi estratti e letti, aggiungendo le parole al trie. Successivamente, quando si andranno a prelevare parole dai files di input, al programma basterà controllare se la parola letta è contenuta in questo trie (se lo stack è vuoto, questo trie sarà NULL e la ricerca ritornerà banalmente 0).
La scelta di utilizzare un trie invece di un array o di una linkedList è giustificata dall'efficienza : la ricerca, nel trie, richiede O nel caso medio e $O(n)$ in quello peggiore (dato che non c'è nessuna operazione di bilanciamento), mentre l'inserimento è O. In una linked list gli inserimenti sarebbero costati $O(1)$ ma la ricerca avrebbe richiesto $O(n)$ mentre, in un array, oltre al costo iniziale necessario per determinarne la dimensione, si sarebbe inserito in $O(1)$ e, dopo aver speso un tempo medio di O per ordinarlo mediante un quicksort, si avrebbe atteso O (nel caso peggiore) per la ricerca degli elementi.

Abbiamo considerato più prioritario ridurre il tempo di ricerca rispetto a quello di inserimento poiché, comunemente, il numero di confronti da effettuare è strettamente maggiore del numero di parole con cui

confrontare. Per utilizzare gli array, inoltre, si sarebbero dovuti allocare spazi di memoria esageratamente grandi al fine di poterli utilizzare in (quasi) ogni contesto o, in alternativa, effettuare letture preliminari dei file per stabilire una dimensione consona.

Abbiamo quindi deciso di utilizzare un trie, con l'ulteriore vantaggio di aver già implementato tutte le funzioni per la sua gestione.

getWord & _getWord

Queste due funzioni si occupano di leggere una parola da un file. La funzione *getWord* richiama la seconda ed effettua dei controlli aggiuntivi filtrando le parole che per qualche ragione (lunghezza della parola, presenza della stessa nel file specificato in --ignore, etc…) non debbono essere contate.
La funzione *_getWord* considera una parola come una qualsiasi sequenza di caratteri contigui esclusivamente alfanumerici, perciò due parole possono essere spezzate, oltre che dagli spazi o dalla newline, anche da un qualsiasi simbolo, segno di punteggiatura o carattere speciale.

_getWord()

Una limitazione che questa funzione impone al programma è che la dimensione massima di una parola debba essere di 500 caratteri, resa evidente dalla dimensione del buffer salvato nella stack area, questa limitazione è necessaria perché è doveroso impostare un limite alla dimensione della parola al fine di evitare la sottomissione in input di parole spropositatamente lunghe. Un limite di 500 caratteri è, in ogni caso, tutt'altro che stretto e non offre perciò una reale limitazione alle capacità del programma.
Un'alternativa sarebbe potuta essere l'allocazione del buffer nell'heap e ampliare dinamicamente la sua dimensione se necessario, non abbiamo, tuttavia, considerato plausibile uno scenario in cui fosse necessario contare parole così lunghe.
La presenza di questa "limitazione", inoltre, permette di interrompere l'analisi di un file binario, nel caso sia più grande di 500 byte, vista l'assenza di controlli per questa tipologia di file.

Mediante un ciclo *while* la funzione analizza e scarta ogni carattere non alfanumerico e non *EOF*, se viene letto *EOF* viene ritornato ritorna *NULL* (e ciò indica che il file è terminato), altrimenti rimette in posizione l'ultimo carattere letto. In secondo ciclo *while* legge poi i caratteri finché questi sono alfanumerici e ogni carattere viene inserito nel buffer. Viene mantenuto un contatore rappresentante sia il numero di caratteri letti che la prossima posizione vuota nel buffer.
Al termine di questo ciclo si alloca uno spazio sull'heap consono alla dimensione della parola e si copiano i caratteri scritti nel buffer in questo spazio, se ne ritorna quindi il puntatore.

getWord

Questa funzione svolge un'operazione di "filtro" sulle parole ritornare da *_getWord* a seconda del valore dei parametri passati. Il parametro *min* indica la lunghezza minima della parola per essere considerata valida e implementa la funzionalità --min. Il parametro *ignoreTrie* è l'albero contenente le parole da ignorare ed è collegato alla funzionalità --ignore, *flags*, infine, è un bit field che interessa questa funzione unicamente per il bit associato alla funzionalità --alpha.

Il ciclo do-while preleva una parola *ret* mediante *_getWord*, la confronta poi con una serie di condizioni, che, se verificate, provocano il rilascio della parola corrente e la lettura della prossima. La condizione che deve essere verificata è, in primi, *ret != NULL*: in caso contrario significa che il file non ha più parole da leggere e non ha senso ripetere il ciclo.

Le altre condizioni per cui la parola viene scartata sono:

- *Strlen(ret) < min*: la parola letta è più corta del parametro *min*
  (in questo modo è implementata la funzionalità --min);
- *Search(ret,ignoreTrie)*: la parola letta è presente nell'albero delle parole da ignorare
  (in questo modo è implementata la funzionalità --ignore);
- *(flags & ALPHA_FLAG) && _isalphanum(ret)* :
  il flag della funzionalità --alpha è settato e la parola contiene almeno una cifra
  (in questo modo è implementata la funzionalità --alpha)

Il ciclo, dunque, non si ripete e la parola letta viene ritornata solo se almeno una delle seguenti condizioni è verificata:

- La parola corrente è *NULL*
- Le 3 condizioni precedenti non sono verificate

_isalphanum

La funzione restituisce 1 se la stringa passata contiene almeno una cifra numerica, 0 altrimenti.

execute

Ottenuto lo stack di files e cartelle da analizzare, l'albero delle parole da ignorare, l'array di stringhe contenente il numero minimo di caratteri delle parole "valide" e il nome del file di output e `flags`, `execute` scrive le coppie *<parola , occorrenze>* nel file output designato.

Inizialmente vengono controllati i valori di `args`: se `min` è *NULL* viene impostato a 1, altrimenti si utilizza `atoi` (appartenente alla libreria `stdlib`) per la conversione da stringa a intero. Se la stringa passata non è valida si restituisce un errore all'utente.

La funzione inizializza il trie globale e crea un nuovo thread per ogni file di input, inizializza la `struct ThreadArgs` e aggiunge il `tid` (thread id) allo stack dei threads, dopo averne creato uno mediante la funzione `pthread_create`.

Creare una struttura `ThreadArgs` è risultato obbligatorio in quanto `pthread_create` (della libreria `pthread`) accetta un puntatore a funzione la cui definizione è

`void *(*`*`start_routine`***`) (void *).`**

Una volta terminati tutti i threads (svuotato, perciò, lo stack dei threads), viene inizializzato il file di output attraverso la funzione makeFile, dopodiché vengono inserite le parole con le relative frequenze in ordine di occorrenza (se richiesto) o in ordine alfabetico attraverso `writeTrie`.

threadFun

`threadFun` è la funzione eseguita da tutti i thread creati in `execute`. Il parametro `arg` è un puntatore a struttura che contiene la path del file da leggere, il minimo numero di lettere per ritenere "valida" una parola, il trie delle parole da ignorare e il bit filed.

La funzione apre il file `src`, richiama `getWord` per ottenere le parole del file (filtrate in base ai parametri di cui sopra), e aggiunge la parola al trie principale.

L'aggiunta della parola al trie è effettuata in zona critica in quanto l'accesso all'albero deve essere mutualmente esclusivo. La "chiave condivisa" `mutex` per il controllo della zona critica e il trie principale `t` sono variabili globali.

sbo & sortTrie

Queste funzioni si occupano di implementare la funzionalità --sortbyoccurrency.
`sbo` richiama la seconda per creare un BST ordinato in base al numero di occorrenze delle parole contenute nel trie e chiama la funzione `writeTree` (di *BST.c*) per scrivere le informazioni del BST su file.
`sortTrie` prende in input un trie e un BST e aggiunge al secondo ogni nodo del trie avente l'attributo *occurrencies* strettamente positivo.