Create a program that checks for all duplicate files in a given directory

[Solution 1:](#Solution) in bash, for each file in the directory it checks all the others using cmp (comparing 2 files byte by byte)

[Solution 2:](#Solution2) in bash, first we compute the sha1s for all files and then put the results into a file so that after we can parse the file and check for duplicates

[Solution 3:](#Solution3) Sequential method creating a list in which every element is a struct that has:

A filename

A checksum

An ingored flag

A next pointer to the next struct

First we compute the checksums for all files and we go recursively into the directory (also into the subdirectories) so that after we can parse the list and check for duplicates

[Solution 4:](#Solution4) Concurrent method using threads and synchronisation mechanism but using the core of solution 3

Solution 1:

#!/bin/bash

find "$1" -type f | while read F; do

find "$2" -type f | while read G; do

if [ "$F" != "$G" ] && cmp -s "$F" "$G"; then

echo "$F = $G"

fi

done

done

Solution 2:

#!/bin/bash

cat /dev/null > a.txt

find "$1" -type f | while read F; do

sha1sum "$F" >> a.txt

done

while read CA FA; do

while read CB FB; do

if [ "$FA" != "$FB" && "$CA" == "$CB" ]; then

echo "$FA = $FB"

fi

done < a.txt

done < a.txt

Solution 3:

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <dirent.h>

#include <fcntl.h>

#include <openssl/evp.h>

#include <sys/types.h>

#include <sys/stat.h>

/\*the length of the checksum\*/

#define L 20

/\*the lenght of the buffer in which we read "chunks" from the file\*/

#define B 1024

/\*defining a strunct, a linked list so that we can go recursively in a directory, find all files, load them into a linked list, calculate the checks sum and then go into the linked list and figure out duplicates\*/

typedef struct rec {

char\* filename;

unsigned char checksum[L];

/\*a flag so that if a file was not opened we know to ignore it\*/

int ignore;

struct rec\* next;

} rec;

/\*the head of the list\*/

rec\* list;

void sha1(rec\* r) {

/\*a buffer in which i read data from the file\*/

unsigned char buf[B];

/\*a file descriptor + a variable to tell us how much read actually read\*/

int fd, k;

EVP\_MD\_CTX\* ctx;

fd = open(r->filename, O\_RDONLY);

if(fd < 0) {

r->ignore = 1;

return;

}

/\*creating the context that calculates the check sum\*/

ctx = EVP\_MD\_CTX\_create();

/\*initializing it\*/

EVP\_DigestInit\_ex(ctx, EVP\_sha1(), NULL);

while(1) {

/\*read from the file into the buffer and the size of the buffer B\*/

k = read(fd, buf, B);

/\*k will hold the number of bytes reaf from the file\*/

if(k < 0)

r->ignore = 1;

if(k <=0)

break;

EVP\_DigestUpdate(ctx, buf, k);

}

close(fd);

EVP\_DigestFinal\_ex(ctx, r->checksum, NULL);

EVP\_MD\_CTX\_destroy(ctx);

}

/\*it takes the name of the directory and the name of the file and concatenates them into a path\*/

char\* path(char\* dir, char\* name) {

char\* p = (char\*)malloc(strlen(dir) + strlen(name) + 2);

/\* 2 : the terminal 0 + the path separator\*/

sprintf(p, "%s/%s", dir, name);

return p;

}

/\*a function that takes a filename and transforms it into a node\*/

rec\* to\_rec(char\* filename) {

rec\* n = (rec\*)malloc(sizeof(rec));

n->filename = filename;

n->ignore = 0;

n->next = NULL;

return n;

}

/\*always return the new tail of the list\*/

rec\* find(char\* dir, rec\*\* head, rec\* tail) {

struct dirent\* e;

DIR\* d = opendir(dir);

rec\* t = tail;

char\* subd;

/\* the opening of the directory might not be possible for example if we don't have permission

\* that is why we check if the opening returns NULL (when there has been and error on opening\*/

if( d != NULL) {

/\*this means i have either a file or a subdirectory\*/

while((e = readdir(d)) != NULL) {

if(e->d\_type == DT\_DIR && strcmp(e->d\_name, ".") != 0 && strcmp(e->d\_name, "..") != 0) {

subd = path(dir, e->d\_name);

t = find(subd, head, t);

free(subd);

}

else if (e->d\_type == DT\_REG) {

/\*the list is empty\*/

if(t == NULL) {

t = to\_rec(path(dir, e->d\_name));

\*head = t;

/\*when the list is empty the head and the tail are the same\*/

}

else {

t->next = to\_rec(path(dir, e->d\_name));

t = t->next;

}

}

}

closedir(d);

}

return t;

}

/\*the check sums are not strigs, they are an array of characters so we need this function to parse and comapre them\*/

int equal(rec\* a, rec\* b) {

int i;

if(a->ignore || b->ignore)

return 0;

for (i = 0; i < L; i++)

if (a->checksum[i] != b->checksum[i])

return 0;

return 1;

}

int main(int argc, char\*\* argv) {

rec\* a ;

rec\* b;

find(argv[1], &list, NULL);

/\*list : global variable, the head of the list\*/

a = list;

while(a != NULL) {

sha1(a);

a = a->next;

}

a = list;

while( a != NULL) {

if (!a->ignore) {

b = list;

while( b != NULL) {

if(strcmp(a->filename, b->filename) != 0 && equal(a, b))

printf("%s = %s\n", a->filename, b->filename);

b = b->next;

}

}

a = a->next;

}

while(list != NULL) {

a = list->next;

/\*first we need to free the space allocated for the filename\*/

free(list->filename);

free(list);

list = a;

}

return 0;

}

Solution 4:

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <dirent.h>

#include <fcntl.h>

#include <openssl/evp.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <pthread.h>

#include <semaphore.h>

/\*the number of threads\*/

#define T 8

/\*the maximum number of threads that ca do a simultaneous read\*/

#define R 8

/\*the length of the checksum\*/

#define L 20

/\*the lenght of the buffer in which we read "chunks" from the file\*/

#define B 1024

/\*how much we read from a file has a huge impact on the implementation\*/

/\*defining a strunct, a linked list so that we can go recursively in a directory, find all files, load them into a linked list, calculate the checks sum and then go into the linked list and figure out duplicates\*/

typedef struct rec {

char\* filename;

unsigned char checksum[L];

/\*a flag so that if a file was not opened we know to ignore it\*/

int ignore;

struct rec\* next;

} rec;

/\*the head of the list\*/

rec\* list;

/\*the synchronization mechanisms\*/

pthread\_mutex\_t mtx;

pthread\_cond\_t cond;

pthread\_rwlock\_t rwl;

pthread\_barrier\_t bar;

sem\_t sem;

void sha1(rec\* r) {

/\*a buffer in which i read data from the file\*/

unsigned char buf[B];

/\*a file descriptor + a variable to tell us how much read actually read\*/

int fd, k;

EVP\_MD\_CTX\* ctx;

fd = open(r->filename, O\_RDONLY);

if(fd < 0) {

r->ignore = 1;

return;

}

/\*creating the context that calculates the check sum\*/

ctx = EVP\_MD\_CTX\_create();

/\*initializing it\*/

EVP\_DigestInit\_ex(ctx, EVP\_sha1(), NULL);

while(1) {

/\*wrap the read in a semaphore so that no more than R threads can read at the same time\*/

sem\_wait(&sem);

/\*read from the file into the buffer and the size of the buffer B\*/

k = read(fd, buf, B);

/\*k will hold the number of bytes reaf from the file\*/

sem\_post(&sem);

if(k < 0)

r->ignore = 1;

if(k <=0)

break;

EVP\_DigestUpdate(ctx, buf, k);

}

close(fd);

EVP\_DigestFinal\_ex(ctx, r->checksum, NULL);

EVP\_MD\_CTX\_destroy(ctx);

}

/\*it takes the name of the directory and the name of the file and concatenates them into a path\*/

char\* path(char\* dir, char\* name) {

char\* p = (char\*)malloc(strlen(dir) + strlen(name) + 2);

/\* 2 : the terminal 0 + the path separator\*/

sprintf(p, "%s/%s", dir, name);

return p;

}

/\*a function that takes a filename and transforms it into a node\*/

rec\* to\_rec(char\* filename) {

rec\* n = (rec\*)malloc(sizeof(rec));

n->filename = filename;

n->ignore = 0;

n->next = NULL;

return n;

}

/\*always return the new tail of the list\*/

rec\* find(char\* dir, rec\*\* head, rec\* tail) {

struct dirent\* e;

DIR\* d = opendir(dir);

rec\* t = tail;

char\* subd;

/\* the opening of the directory might not be possible for example if we don't have permission

\* that is why we check if the opening returns NULL (when there has been and error on opening\*/

if( d != NULL) {

/\*this means i have either a file or a subdirectory\*/

while((e = readdir(d)) != NULL) {

if(e->d\_type == DT\_DIR && strcmp(e->d\_name, ".") != 0 && strcmp(e->d\_name, "..") != 0) {

subd = path(dir, e->d\_name);

t = find(subd, head, t);

free(subd);

}

else if (e->d\_type == DT\_REG) {

pthread\_rwlock\_wrlock(&rwl);

/\*the list is empty\*/

if(t == NULL) {

t = to\_rec(path(dir, e->d\_name));

\*head = t;

/\*when the list is empty the head and the tail are the same\*/

}

else {

t->next = to\_rec(path(dir, e->d\_name));

t = t->next;

}

pthread\_rwlock\_unlock(&rwl);

pthread\_mutex\_lock(&mtx);

/\*broadcast so that all thread are woken up because something new was added to the list\*/

pthread\_cond\_broadcast(&cond);

pthread\_mutex\_unlock(&mtx);

}

}

closedir(d);

}

return t;

}

/\*the check sums are not strigs, they are an array of characters so we need this function to parse and comapre them\*/

int equal(rec\* a, rec\* b) {

int i;

if(a->ignore || b->ignore)

return 0;

for (i = 0; i < L; i++)

if (a->checksum[i] != b->checksum[i])

return 0;

return 1;

}

void\* worker(void\* x){

/\* n is a counter, better to be local not global as it would need synchronization\*/

int n = 0;

int idx = \*(int\*)x;

rec\* a;

rec\* b;

/\*wait for list head to be non NULL\*/

pthread\_mutex\_lock(&mtx);

while(list == NULL)

/\*wait take as parameter the mutex

\* it unlocks the mutex (so that things can happen somewhere elese) before going to sleep

\* when signaled/broardcasted it wakes up and lock the mutex before returning

\* !!! the mutex is free only insed the wait\*/

pthread\_cond\_wait(&cond, &mtx);

a = list;

pthread\_mutex\_unlock(&mtx);

/\*we don't exactly know the condition so we'll do a break when the right moment comes\*/

while(1) {

pthread\_rwlock\_rdlock(&rwl);

/\*no need to wait here\*/

while(1) {

/\*if it is the end of the list\*/

if(a->filename == NULL)

break;

/\*this means that the thread is on an element that it needs to process\*/

if(n % T == idx)

sha1(a);

n++;

/\*this means that the thread is on thge last element that it has to process\*/

if(a->next == NULL)

break;

a = a->next;

}

/\*we have to write it again as we cannot break more than 1 loop in C\*/

if(a->filename == NULL) {

pthread\_rwlock\_unlock(&rwl);

break;

}

/\*wait for more elements\*/

pthread\_mutex\_lock(&mtx);

while(a->next == NULL) {

/\* unlock the read lock before signaling wait

\* this way the program won't lock up

\* because it won't go to sleep with the readlock locked\*/

pthread\_rwlock\_unlock(&rwl);

pthread\_cond\_wait(&cond, &mtx);

pthread\_rwlock\_rdlock(&rwl);

}

a = a->next;

pthread\_mutex\_unlock(&mtx);

}

/\*wait for all workers to finish sha1 calculation\*/

pthread\_barrier\_wait(&bar);

/\*detect duplicates\*/

a = list;

n = 0;

while(a->filename != NULL) {

if(n % T == idx && !a->ignore) {

b = list;

while(b->filename != NULL) {

if(strcmp(a->filename, b->filename) != 0 && equal(a,b))

printf("%s = %s", a->filename, b->filename);

b = b->next;

}

}

n++;

a = a->next;

}

return NULL;

}

int main(int argc, char\*\* argv) {

pthread\_t t[T];

int i, a[T];

/\*the blank element\*/

rec\* n;

pthread\_mutex\_init(&mtx, NULL);

pthread\_cond\_init(&cond, NULL);

pthread\_rwlock\_init(&rwl, NULL);

pthread\_barrier\_init(&bar, NULL, T);

sem\_init(&sem, 0, R);

for(i = 0; i < T; i++) {

a[i] = i;

pthread\_create(&t[i], NULL, worker, &a[i]);

}

/\*main creates the threads and only then it starts parsing the folder

\* so it may be that some threads get to the point (a = list, in worker function)

\* when there is nothing there

\* SOLUTION: wait for list to be non NULL at the beginning of the function

\* not before the second while

\* \*/

n = find(argv[1], &list, NULL);

pthread\_rwlock\_wrlock(&rwl);

/\*this is the marker for the end of the list\*/

n->next = to\_rec(NULL);

pthread\_rwlock\_unlock(&rwl);

pthread\_mutex\_lock(&mtx);

/\*broadcast so that all thread are woken up because something new was added to the list\*/

pthread\_cond\_broadcast(&cond);

pthread\_mutex\_unlock(&mtx);

for(i = 0; i < T; i++)

pthread\_join(t[i], NULL);

pthread\_mutex\_destroy(&mtx);

pthread\_cond\_destroy(&cond);

pthread\_rwlock\_destroy(&rwl);

pthread\_barrier\_destroy(&bar);

sem\_destroy(&sem);

while(list != NULL) {

n = list->next;

/\*first we need to free the space allocated for the filename\*/

free(list->filename);

free(list);

list = n;

}

return 0;

}