Disk Analyzer

GRUPA 241

Dumitrescu Nicolle-Alexandra
Lungu Laura-Vanesa
Negoiță-Crețu Raluca-Marina
Țîrdea Mihai-Cătălin

Universitatea din București
Departamentul de Informatică
Sisteme de operare
2023-2024

Table of Contents

Descrierea Problemei

Cerința:

Proiectul implică crearea unui daemon pentru analizarea recursivă a spațiului utilizat pe un dispozitiv de stocare și a unui program utilitar în linia de comandă care să asigure interacțiunea dintre daemon și userul final.

Ce este un daemon mai exact?

Un daemon este un tip de proces care rulează în fundal, în mod continuu, fără a fi legat de o sesiune de utilizator specifică. Aceste procese deservesc diverse funcționalități de sistem cum ar fi: execuția programelor, alocarea resurselor, protecție și securitate, etc.

Ce este și ce face un program utilitar?

Utilitarul permite crearea, anularea și interogarea joburilor de analiză, oferind și opțiuni de suspendare, repornire și afișare a rapoartelor de analiză pentru directoarele specificate ca instrucțiuni daemonului.

Utilitatea unui astfel de serviciu

Un astfel de disk analyzer este util în mai multe scenarii:

- **Gestionarea spațiului pe disc**: Utilizatorii identifică rapid directoarele sau fișierele care ocupă cel mai mult spațiu.
- **Optimizarea stocării**: Ajută la identificarea și eliminarea datelor redundante pentru eliberarea spațiului.
- **Planificarea și prioritizarea**: Oferă posibilitatea de a planifica și prioritiza analiza spațiului ocupat; permite utilizatorilor să stabilească ordinea efectuării analizelor
- **Monitorizarea modificărilor de spațiu**: Utilizatorii pot vedea modul în care se schimbă spațiul pe disc în timp, pentru urmărirea evoluției.
- **Automatizarea analizei**: Posibilitatea de a programa mai multe analize simultan.

Folosirea unui sistem de fișiere partajate

În contextul problemei noastre avem două programe principale: da.c și daemon.c.

- ***da.c***: acest program asigură interfața cu utilizatorul în linia de comandă, preia input-ul acestuia și dacă este corect îl trimite mai departe daemonului, iar în caz contrar afișează un mesaj de eroare de sintaxă.
- ***Daemon.c:*** preia mai departe instrucțiunile date de utilitar, pe care în funcție de un set de reguli prestabilite din cerință îl rezolvă.
- **comunicare**: Cele două programe comunică între ele printr-un **sistem de fișiere partajate**.

Programul utilitar da.c parsează argumentele din linia de comandă, stabilește instrucțiunea pe care o formatează corespunzător și o scrie într-un fișier de instrucțiuni pe care serviciul daemonul vine și îl citește pentru a-l rezolva. După ce termină task-ul, daemonul scrie în alt fișier rezultatul, iar da.c vine și citește rezultatul pe care i-l afișează userului care a făcut request-ul.
De exemplu, în implementare am folosit fișierele: input.txt, output.txt, debug.txt (o adaptare a syslog-ului)

Realizarea comunicării interprocess

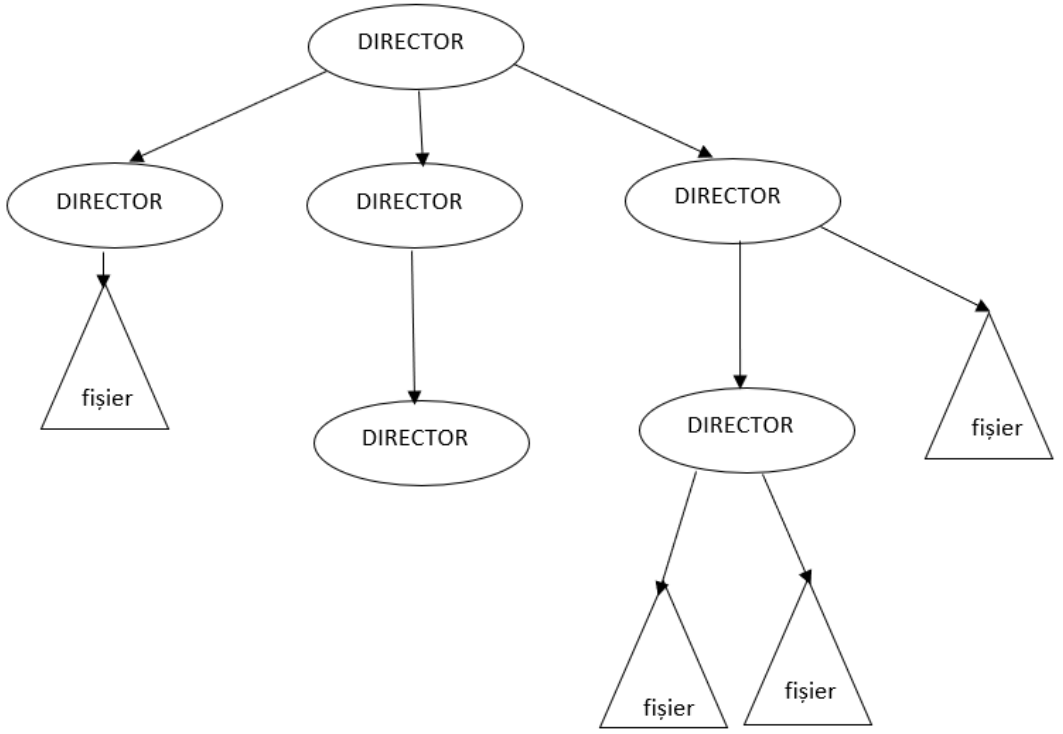
Cele două programe își cunosc **pid**-urile unul altuia prin intermediul unor fișiere partajate în care fiecare program își scrie pid-ul, ca celălalt program să poată să îl citească.
Cu ajutorul acestor pid-uri, programele comunică prin **transmiterea de semnale** (SIGUSR1 și SIGUSR2)
Când da.c dorește să transmită o instrucțiune daemonului, scrie instrucțiunea în fișierul input.txt și trimite semnalul SIGUSR1 către daemon. Daemonul, care a fost configurat să răspundă la acest semnal, citește instrucțiunea din fișier și o execută.
După ce daemonul finalizează o operațiune de analiză a spațiului pe disc, rezultatele sunt scrise în fișierul output.txt și trimite semnalul SIGUSR2 către da.c . Acesta când primește semnalul îl tratează prin apelarea funcției de citire a rezultatelor din output.txt pe care le afișează ulterior userului.

Utilitatea threadurilor

Thread-uri: fiecărui job de analiză nou i se asociază un thread. Pentru a ne asigura de rezolvarea task-urilor în funcție de prioritate și eficiență, threadurile sunt reprezentate/stocate sub forma unei **liste simplu înlănțuite**.
Totodată pentru evitarea de **race conditions** am inițializat **un mutex** care să asigure și sincronizarea accesului la resursele partajate.
Un mutex este *un mecanism de sincronizare utilizat pentru a evita conflictele între thread-uri care încearcă să acceseze sau să modifice resurse partajate în mod concurent*. Principiul de bază este să permită doar unui singur thread să acceseze o resursă partajată la un moment dat, evitând astfel condițiile de cursă (race conditions).
Un thread care dorește să acceseze o resursă partajată întâi trebuie să "închidă" mutex-ul asociat acelei resurse. Dacă mutex-ul este deja închis de către alt thread, thread-ul curent va fi blocat până când mutex-ul devine disponibil.
După ce un thread a terminat de utilizat resursa partajată, trebuie să "deschidă" mutex-ul, permițând astfel altor thread-uri să obțină acces la resursa respectivă.

Reprezentarea fișierelor și a directoarelor

Directoarele sunt reprezentate fiecare într-un struct asemănător unui **hashmap** care reține dimensiunea directorului și un vector de liste înlănțuite de fișiere/directoare conținute recursiv, în adâncime de acest director. Astfel, se formează o **structură de arbore** a path-urilor pe care disk-analyzer-ul le va analiza recursiv.



Biblioteca FTS

Subprogramul disk_analyzer apelat de daemon are rolul de a analiza un director specificat prin calea path. Acesta folosește **API-ul FTS (File Tree Structure)** pentru a parcurge ierarhia de directoare și fișiere în două etape: postorder și preorder.
Postorder traversal: În timpul acestei etape, se colectează informații despre dimensiunea totală a fișierelor și subdirectoarelor din fiecare director, utilizând structura de directoare menționată anterior pentru a stoca aceste informații asociate cu **numărul inode-ului** al fiecărui director. De asemenea, se creează un fișier de ieșire pentru a stoca informațiile detaliate despre dimensiunea și procentajul de utilizare a spațiului pentru fiecare director și fișier.
Preorder traversal: În această etapă, se generează și se scriu în fișierul de ieșire informații detaliate despre

dimensiunea și procentajul de utilizare a spațiului pentru fiecare director, în ordinea în care acestea sunt parcurse.

Această etapă contribuie la crearea unei reprezentări vizuale a structurii de directoare și a spațiului de stocare utilizat. Se efectuează operațiile de generare a informațiilor pentru fiecare **nod de tip director (FTS_D),** fiecare **nod de tip fișier (FTS_F),** fiecare **nod de tip symbolic link (FTS_SL)** calculând dimensiunea și procentajul de utilizare a spațiului.

Utilizare

Pentru a putea testa funcționalitatea programului creat, este nevoie ca toate fișierele programelor .c și .h să fie plasate în același folder, astfel userului îi va fi mult mai ușor să ruleze programul în terminal.
Prima dată, dacă nu dorim să transformăm programul daemon.c într-un serviciu al sistemului de operare care să ruleze by default în background, putem simula această acțiune prin deschiderea și manipularea în pararlel a două terminale. Astfel, în terminalul I pe care îl deschidem în folderul aferent programelor rulăm programul daemon.c cu compilatorul gcc. (gcc –o nume_executabil1 daemon.c –pthread) și apoi pornim procesul cu ./nume_executabil1.
Al doilea pas este, să dechidem terminalul II, tot în folderul aferent programelor și să rulăm comanda care pornește execuția utilitarului da.c (gcc –o nume_executabil2 da.c –pthread).
In urma acestor rulări, ambele programe își vor scrie pid-urile în fișierele aferente (discutate anterior) și vor sta în așteptarea semnalui pe care să-l primească de la celălalt proces deschis.

Daemon.c proces de fundal

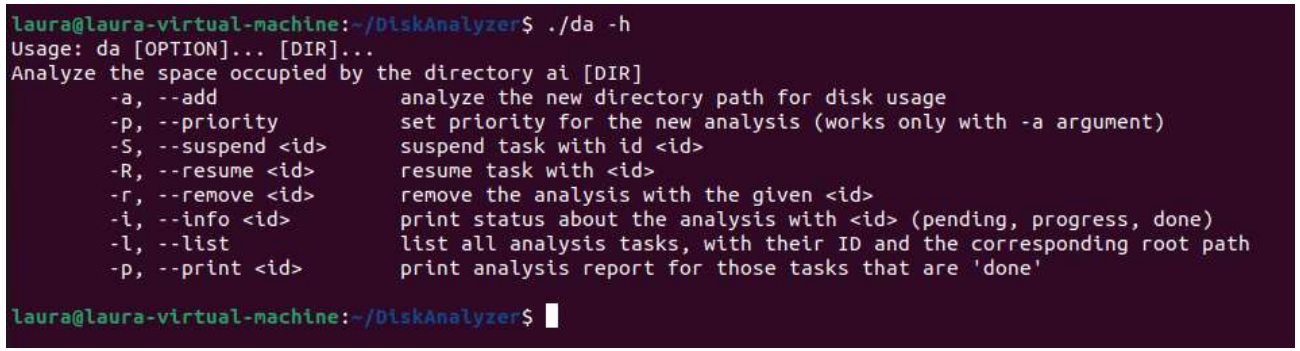
- După cum se poate observa din print-screen-ul alăturat, daemon.c rulează în mod continuu și așteaptă comenzile utilitarului.
-În acest moment, toate interacțiunile userului cu DiskAnalyzer-ul se vor face doar din terminalul II, adică cel din care am rulat comanda pentru pornirea procesului da.c
-Singurele moduri prin care daemon.c poate înceta din execuție este oprirea forțată (Ctrl-C) sau în cazul unui task dat din da.c care îi suprasolicită capacitățile.
(Despre limitările proiectului nostru urmează să discutăm mai târziu.)

```c
int main(){


    initialization();
    write_daemon_pid();


    struct thread_node *current_thread = (struct thread_node*)malloc(sizeof(struct thread_node));
    void *result;

    while(1){

        current_thread = *threads_head; //list of threads
        while(current_thread!=NULL){
            //while the thread is still runnning
            pthread_join(*current_thread->thr, &result);
            //each thread puts its result in the res variable,
            //the daemons waits for all the threads to finish via pthread_join
            current_thread = current_thread->next;
        }

    }

    return 0;

}
```

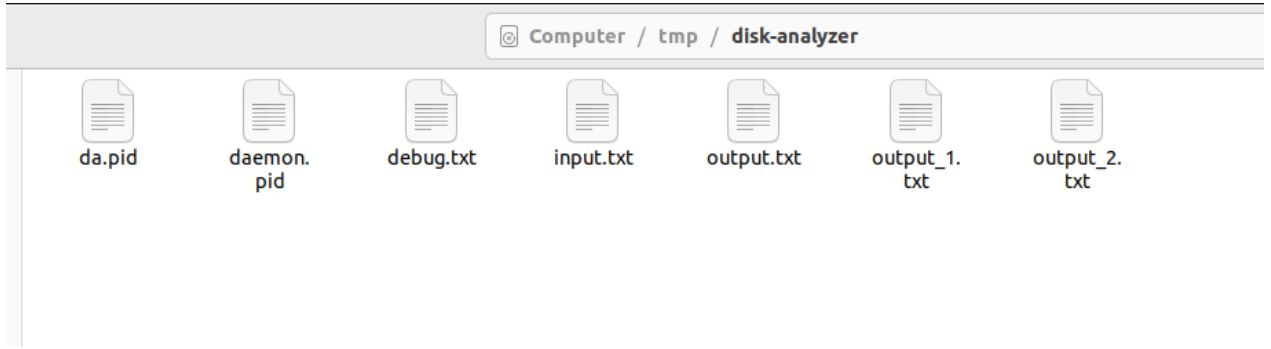
```
laura@laura-virtual-machine:~/DiskAnalyzer$ gcc -o daemon daemon.c -pthread
laura@laura-virtual-machine:~/DiskAnalyzer$ ./daemon
```

Da.c interfața cu userul

- Pentru a începe utilizarea efectivă a funcțiilor programului, userul poate rula in terminal ./da -h sau ./da --help pentru a vedea ce opțiuni are.



- Sarcina principală a programului da.c este să preia argumentele userului din linia de comandă, pe care le parsează și le transmite daemonului sub forma unor instrucțiuni pe care le scrie într-un fișier. Daemonul citește aceste instrucțiuni când primește semnalul SIGUSR1, le rezolvă și le scrie în fișiere output denumite sugestiv în funcție de numărul de joburi de analiză noi care se creează.



Experimente și Testare

Arhitectura hardware/software folosită pentru testare:

-Kernel: Linux (last version)

-Architecture x86_64

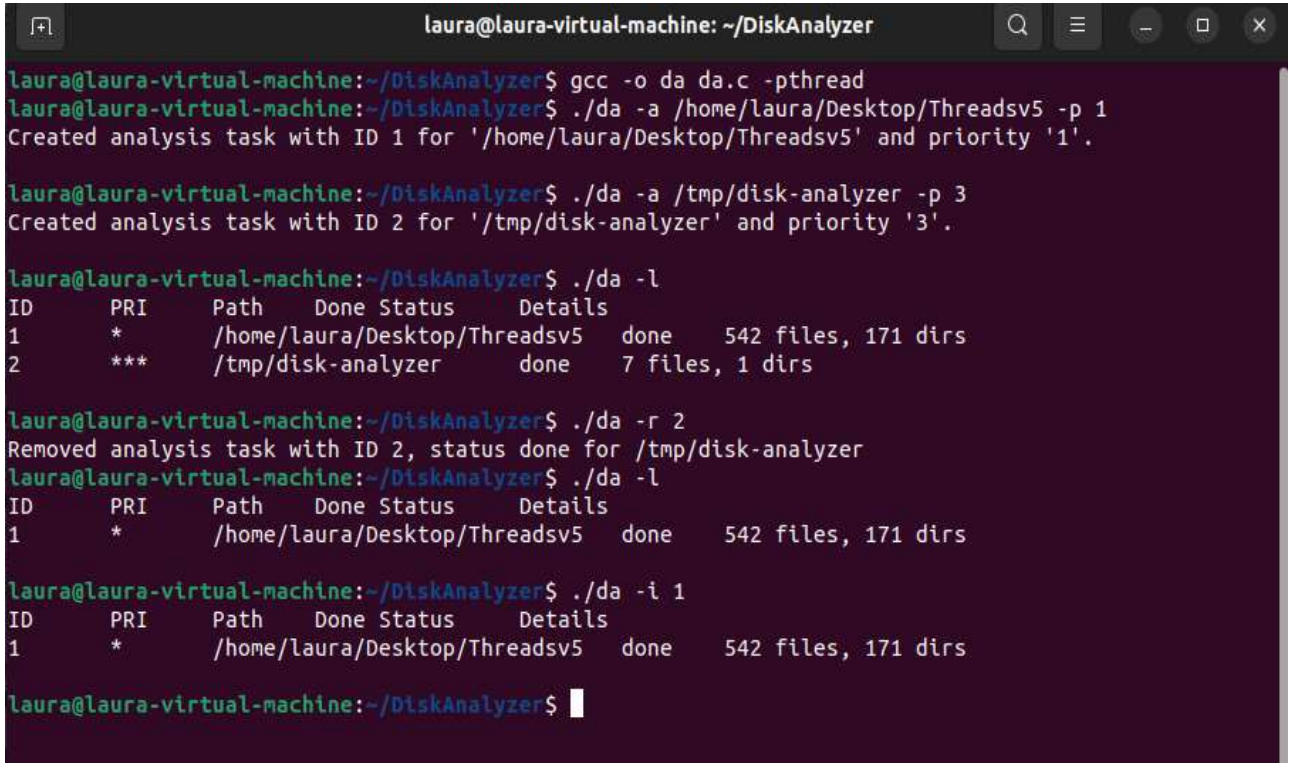
-CPU(s): 2

-Thread(s) per core: 1

-Core(s) per socket: 1

-Socket(s): 2

-No. Threads: utilitarul rulează pe un singur thread, iar daemon-ul poate crea mai multe threaduri în funcție de joburile de analiză pe care le pornește.



Implementare. Stocarea datelor

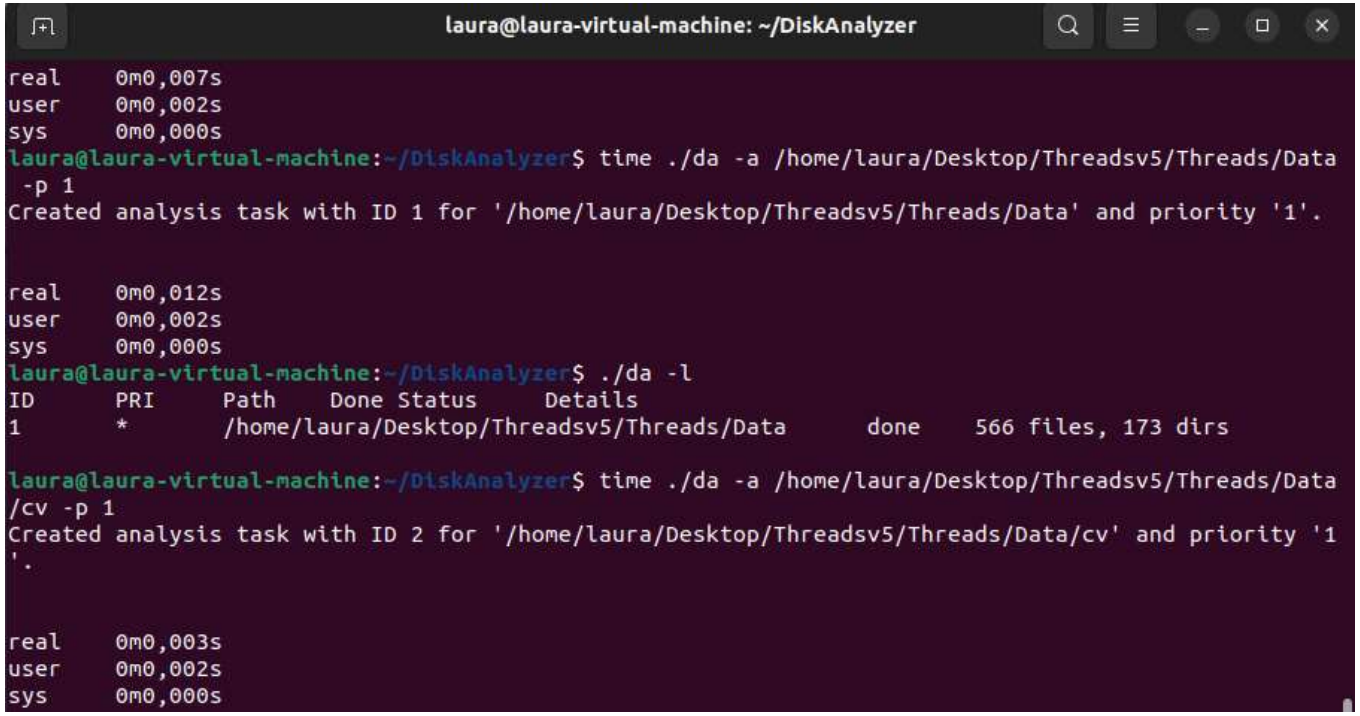
Pentru a afișa informații despre fiecare fișier în parte descoperit în path, ne folosim de *structura file_directory*. Aceasta are un id asociat, reprezentat de inode-ul aferent găsit prin intermediul bibliotecii FTS, despre care am discutat la secțiunea Design. În această structură reținem și path aferent al fișierului descoperit, dar și un pointer către următorul fișier, în cazul în care în parcurgerea recursivă se mai găsește unul.
Pentru a reține informații despre arborele de fișiere creat de path-ul dat ca instrucțiune, folosim *structura directory*, care are un design ce seamănă cu un hashmap, aici reținem dimensiunea directorului și un vector de liste înlănțuite de fișiere conținute în fiecare director.
Pentru a gestiona lista înlănțuită de threaduri utilizăm *structura thread_node* care reține id-ul jobului de analiză, prioritatea, thread-ul, statusul analizei, numărul de fișiere/directoare conținute și un pointer către următorul thread din lista înlănțuită de threaduri.

```
struct file_directory {
  int id;         // ino number associa
  void *fd_path; // file_directory pat
  struct file_directory *next; // poin
};

You, 5 hours ago | 1 author (You)
struct directory {
  int size;
  struct file_directory **content;
};
```

```
struct thread_node {
  int id; // job ID;
  int priority;
  pthread_t *thr;
  char *status;
  int no_file, no_dirs, no_all_dirs;
  struct thread_node *next;
};
```

Experimente

În urma unor teste efectuate cu directoarele cele mai stufoase și mai puțin stufoase care se găsesc pe mașina virtuală, am observat ca timpul de execuție variază între un interval redus de câteva milisecunde, ceea ce atestă o oarecare eficiență a algoritmului pentru directorare de maxim 600 de fișiere.

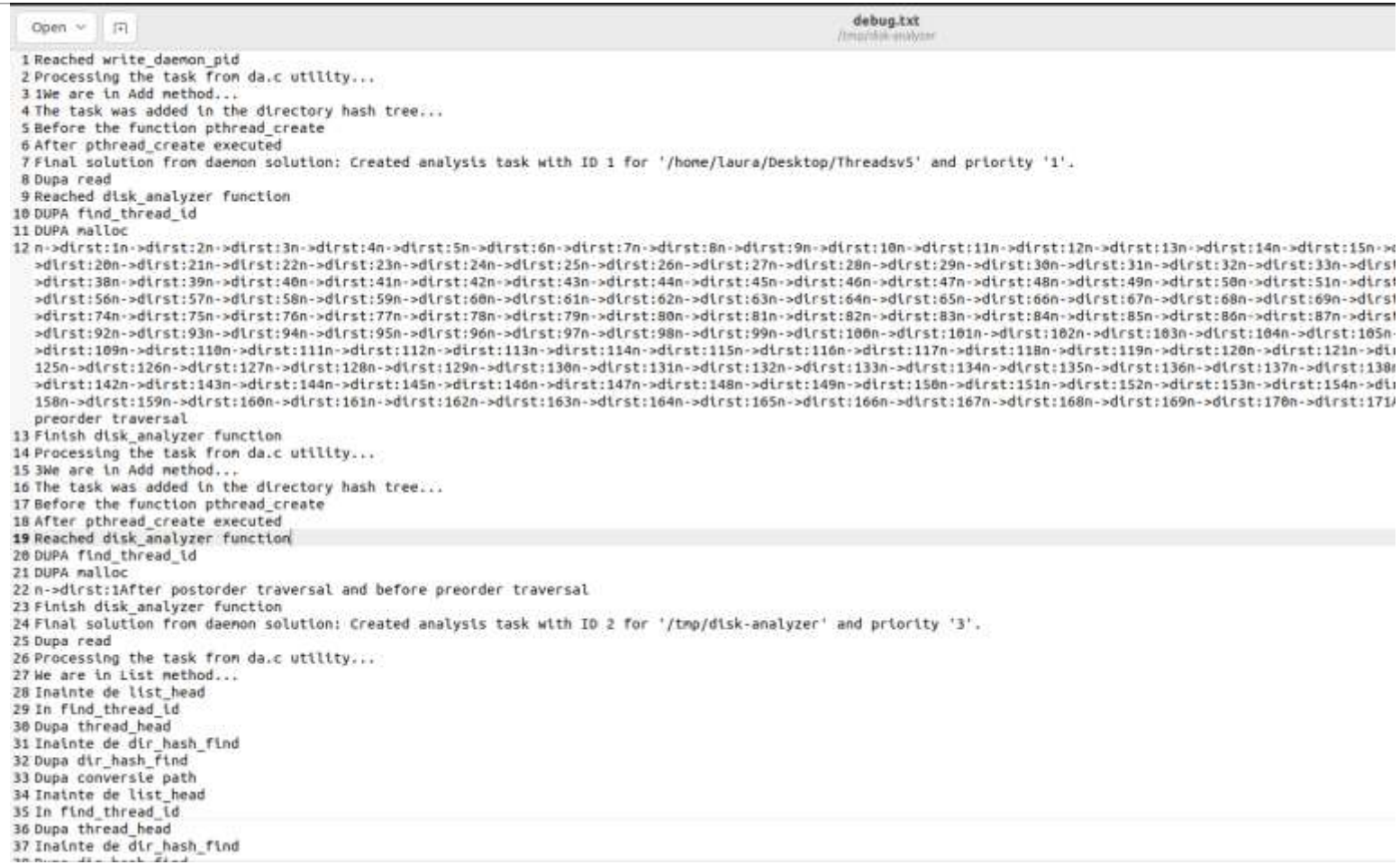


Limitările proiectului

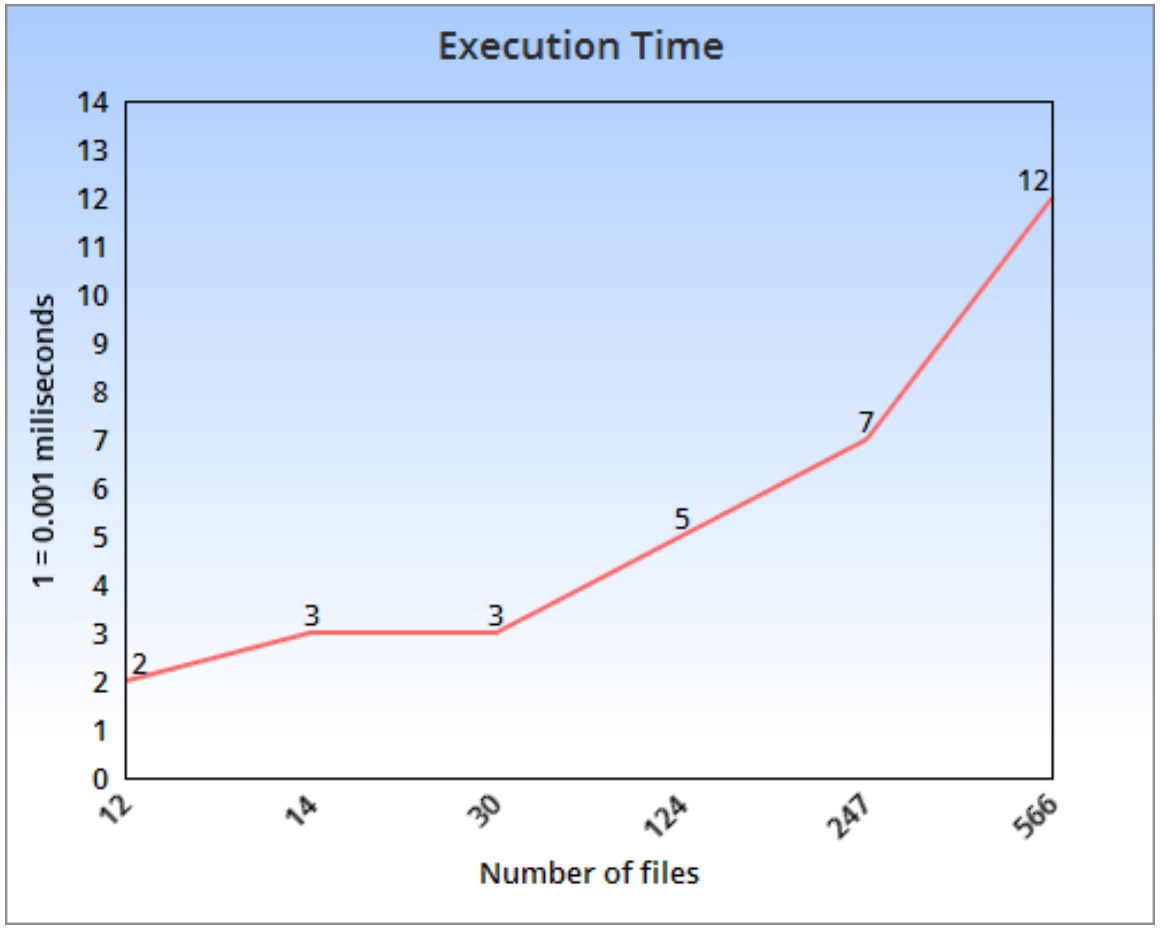
• Continuând analiza începută anterior, ne-am propus să descoperim care este punctul din care daemonul ar putea genera excepția segmentation fault, și am descoperit că programul nostru nu e chiar atât de eficient pe cât ne-am propus inițial din cauza unor erori încă nedepistate. Am observat că dacă testăm analyzerul pentru un director care conține mai mult de 600 de fișiere, răspunsul daemonului este imprevizibil, problemele fiind generate în principiu de alocarea dinamică. Această problemă la malloc, o mai întâlnisem o dată în prima etapă de dezvoltare, și reușisem să o soluționăm prin simpla alocare a unui spațiu de memorie mai mare pentru analiza directoarelor.
•De asemenea, ne-am gândit să testăm dacă daemonul poate să primească task-uri simultan de la mai multe programe utilitare. Practic am mai deschis un terminal în care am încercat să dăm task-uri în paralel daemonului și am obsevat că de asemenea răspunsul daemonului este segmentation fault. Din această cauză am decis să nu mai creăm un nou serviciu pentru daemon, întrucât am constatat destule nereguli.
•Pe lângă problemele menționate anterior, analyzer-ul nostru nu poate îndeplini nici funcțiile de suspend și resume. Aceste două funcții au fost doar testate la nivel de debug, însă nu am reușit să venim cu o soluție funcțională care să nu genereze erori, așa că am comentat implementarea acestor două funcții.

Soluționarea problemelor

Pentru a identifica și rezolva mai ușor erorile pe care le-am întâlnit pe parcursul dezvoltării, am apelat la crearea unui "syslog personalizat" menit să afișeze mesaje de debug în funcție de traseul parcurs de daemon printre funcțiile pe care le implementează/apelează.



Graficul experimentelor noastre



Conluzii

Proiectul DiskAnalyzer își dovedește importanța atât în ceea ce presupune utilitatea acestuia pentru utilizatorul final, cât și pentru dezvoltatori.

În ceea ce ne privește, proiectul a presupus un research amănunțit în care a trebuit să împletim mai multe concepte ce țin de arhitectura sistemelor de operare Linux (semnale, thread-uri, mutex-uri, servicii, fișiere partajate, comunicare interproces, race conditions, Biblioteca FTS, etc.), dar și cunoștințe avansate de algoritmică și structuri de date(parcurgeri de arbori, lucrul cu pointeri, liste înlănțuite, hash-tables).