UNIVERSITATEA BUCUREȘTI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licență

Analiza statică a primitivelor de sincronizare

Coordonator științific
Paul Irofti

Absolvent
Darius Marian

București, iulie 2020

Cuprins

1 Introducere **3**
1.1 SyncAnalysis 4
1.2 Comparație cu ThreadSanitizer 5
1.3 Comparație cu Clang-Tidy 7

2 Tehnologii folosite **8**
2.1 Limbajul C11 și biblioteca standard C 8
2.2 Limbajul C++17 și biblioteca standard C++ 8
2.3 Compilatorul Clang 8
2.4 Uneltele de dezvoltare CMake și CTest 9
2.5 Biblioteca pthread 9
2.6 Biblioteca DL 9
2.7 Biblioteca libunwind 9
2.8 Programele addr2line, respectiv atos 10
2.9 Biblioteca mcga-cli 10
2.10 Git și GitHub 11
2.11 Biblioteca Google Benchmark 11

3 Descrierea proiectului **12**
3.1 Instalarea și compilarea proiectului 12
3.2 Biblioteca de capturare a evenimentelor 13
3.2.1 Interfață 14
3.2.2 Structura unui eveniment 17
3.2.3 Stiva de execuție 19
3.2.4 Coada de evenimente și serializarea 21
3.2.5 Fișiserului DUMP 23
3.2.6 Măsurători de performanță 24
3.3 Programul de analiză post-mortem SyncAnalysis 25
3.3.1 Argumentele pentru linia de comandă 25
3.3.2 Înregistrarea analizorilor 26
3.3.3 Procedeul de analiză 27
3.3.4 Clasa `syan::Event` 28

3.3.5 Crearea unui raport 28
3.3.6 Baza de date a obiectelor active 29
3.3.7 Afișarea simbolurilor din stiva de execuție 30
3.3.8 Interfața C++ pentru un analizor 31

4 Exemplu de unealtă de analiză **33**
4.1 Tipuri de evenimente pentru primitive de sincronizare 33
4.2 Biblioteci de integrare 34
4.2.1 cxxsync (C++, începând cu C++11) 34
4.2.2 pthread shim (C/C++) 37
4.3 Analizori 38
4.3.1 useless-lock 39
4.3.2 lock-shadow 39
4.3.3 redundant-recursive-mutex 39
4.3.4 redundant-rwlock 39
4.4 Testele analizorilor (folosind cxxsync) 39

5 Concluzii **40**

1 Introducere

Programele cu mai multe fire de execuție au devenit mult mai comune în programarea modernă. Procesoarele noi și plăcile grafice conțin mai multe nuclee separate și capabilități de virtualizare, iar pentru a folosi eficient aceste resurse, programele axate pe performanță partajează sarcinile pe care le au de îndeplinit în mai multe fire de execuție ce pot fi executate în paralel pe același hardware.

Dar prin introducerea acestui nou model de programare, s-a introdus și o nouă gamă de dificultăți și posibilități de a face greșeli programatice. Firele de execuție dintr-un proces accesează același spațiu de memorie RAM, același disc, același monitor și alte dispozitive ale calculatorului. De pildă dacă mai multe fire de execuție ce rulează *concurent* încearcă să stocheze informații diferite la aceeași adresă de memorie RAM, rezultatul obținut nu este clar definit. Problemele de acest tip se rezolvă folosind *mecanisme de sincronizare* pentru a controla și arbitra accesul la resurse comune pe care mai multe fire de execuție le folosesc concurent. Majoritatea acestor mecanisme sunt construite în jurul unor *primitive de sincronizare* printre care se numără *mutex*, *semaphore*, *read-write lock* și *condition variable*.

Folosirea acestor primitive este considerată dificilă. Orice resursă partajată care este accesată concurent de mai multe fire de execuție poate duce la un *race condition* (o situație în care rezultatul este nepredictibil pentru că depinde de ordinea în care firele de execuție accesează resursa, dar de cele mai multe ori nu este cel intenționat), așa că toate situațiile de acest tip din program trebuie apărate folosind mecanisme de sincronizare. În același timp, orice exces de astfel de mecanisme poate duce la alte tipuri de erori cum ar fi *deadlock* (situație în care două sau mai multe fire de execuție se blochează reciproc, și niciuna nu mai poate progresa), *starvation* (un fir nu mai ajunge niciodată să își îndeplinească sarcina pentru că nu mai primește acces la resursa partajată) sau degradarea performanței programului (deoarece principalul motiv pentru care se folosesc mai multe fire de execuție este performanța, aceasta este de multe ori la fel de importantă ca și corectitudinea).

Dificultatea de înțelegere și utilizare a acestor concepte, împreună cu rezultatele dezastruoase care apar frecvent din cauza erorilor de programare crează o nevoie de unelte care să ajute dezvoltatorii de aplicații în a identifica și repara acest tip de greșeli. Deși există deja multe astfel de unelte în ecosistemul programării, nu toate problemele ce pot

apărea sunt rezolvate de unelte existente. De asemenea, fiecare program care folosește aceste primitive de sincronizare are alte nevoi, și o unealtă generică de multe ori nu poate rezolva problemele specifice întâmpinate de un anumit program.

Vom prezenta în continuare un proiect ce are ca scop nu doar crearea unei astfel de unelte, ci a unei platforme de dezvoltare ce facilitează conceperea și implementarea de unelte noi și specifice unei anumite nevoi printr-un efort minim.

1.1 SyncAnalysis

SyncAnalysis este un proiect ce își propune crearea unui sistem în care este ușor de dezvoltat unelte noi de diagnosticare a problemelor ce pot apărea în folosirea mecanismelor primitive de sincronizare (și nu doar).

Ideea din spatele acestui proiect vine din următoarea observație: prin capturarea unor evenimente cheie pe parcursul execuției unui program, se pot diagnostica o gamă largă de probleme pe care le întâmpină acesta. Un exemplu simplu ar fi că prin observarea tuturor cererilor de *lock* și *unlock* pe o instanță de *mutex*, se poate observa că *mutex*-ul respectiv nu este necesar dacă toate apelurile se întâmplă în contextul aceluiași fir de execuție.

Din această observație se vede că proiectul folosește strategia *post-mortem* pentru analiza programelor: de-a lungul execuției programului se înregistrează evenimentele de interes, pentru a fi procesate și analizate separat după terminarea acestuia. Această strategie a fost aleasă pentru că nu este deloc intrusivă: necesită schimbări minime sau nule în codul sursă al programului (în anumite situații nu necesită nici recompilarea codului sursă al unui program) și nu degradează prea tare performanța programului în sine pentru a face analiza. Alte strategii de analiză folosite în unelte existente includ analiză *on-the-fly*, în care analiza este făcută în paralel cu execuția programului în același proces, precum în ThreadSanitizer[1] și analiză *statică*, în care codul sursă în sine este analizat, nu programul obținut prin compilare, precum în Clang Thread Safety Analysis [2].

Se distinge astfel direcția generală a acestui sistem. Prima și cea mai importantă componentă este o bibliotecă ce înregistrează evenimentele interesante în timpul execuției programului și le serializează într-un fișier, pentru a fi analizate separat de alt program. Detalii despre această bibliotecă și implementarea ei se găsesc în secțiunea **3.2**.

După execuția programului client obținem o colecție de evenimente, capturate și

serializate eficient într-un fișier cu ajutorul bibliotecii descrise mai sus. În continuare, trebuie făcută o analiză a acestor evenimente, iar proiectul oferă o soluție pentru asta în forma unui program independent numit `SyncAnalysis`. Acest program, precum biblioteca de capturare a evenimentelor, este unul de uz general, și nu este direct legat de analiza primitivelor de sincronizare. Acesta nu execută pe cont propriu nicio analiză asupra evenimentelor, ci se bazează pe o suită de module separate numite *analizori*. Acești analizori sunt distribuiți sub formă de biblioteci dinamice pe care `SyncAnalysis` le încarcă la începutul execuției. Programul se ocupă de a deserializa și indexa evenimentele din fișier în memorie, pentru ca apoi *analizorii* să diagnosticheze erori, avertismente sau în general informații utile pe baza acestor evenimente, iar programul să creeze și să afișeze apoi rapoarte pentru utilizator. Mai multe detalii despre acest program se găsesc în secțiunea **3.3**, iar despre analizorii implementați ca exemplu pentru analiza primitivelor de sincronizare în secțiunea **4.3**.

Biblioteca și programul de analiză post-mortem sunt de uz general, neavând în mod direct o interfață specifică analizării primitivelor de sincronizare. De aceea proiectul conține și două exemple de biblioteci pentru a facilita folosirea bibliotecii de capturare a evenimentelor în scopul de analiză a primitivelor de sincronizare, descrise în detaliu în Secțiunea **4.2**.

În Figura 1 se poate vedea arhitectura software a proiectului SyncAnalysis. Cu albastru sunt marcate elementele de uz general, adică cele independente de unealta dezvoltată (folosibile chiar pentru unelte care nu se ocupă de analiza primitivelor de sincronizare), cu verde sunt marcate elementele ce sunt specifice unei anumite unelte, iar cu gri sunt marcate elementele programului client.

1.2 Comparație cu ThreadSanitizer

O unealtă populară ce ajută la diagnosticarea folosirii insuficiente a mecanismelor de sincronizare este ThreadSanitizer[1]. Folosirea acestuia implică compilarea programului cu un compilator care implementează suport pentru ThreadSanitizer, folosind opțiunile de linie de comandă documentate de compilator (de exemplu, pentru Clang pe Linux opțiunea este `-fsanitize=thread`). Executabilul obținut prin compilare verifică toate scrierile în memoria RAM și emite erori dacă o scriere este concurentă cu o alta sau cu o citire, și cele două nu sunt explicit sincronizate.

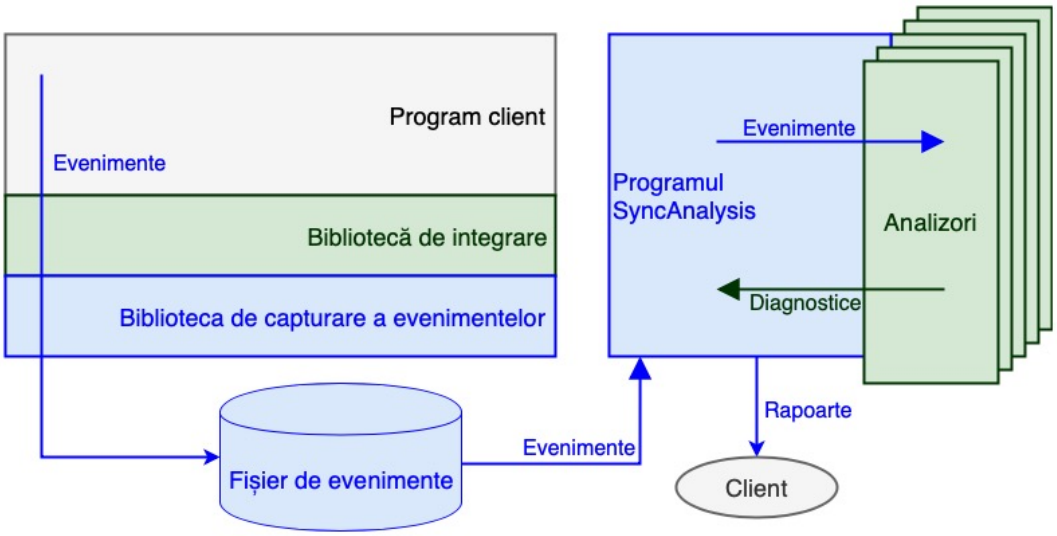

Figura 1: Arhitectura proiectului SyncAnalysis

În Fragmentul de cod 1 (parafrazat din site-ul oficial de documenție al proiectului [3]) se vede cum ThreadSanitizer analizează programul *on-the-fly*: analiza este făcută direct în cadrul execuției programului, în același proces. Asta duce la o încetinire drastică a programului și la un consum ridicat de memorie. Măsurătorile făcute pentru articolul [1] arată o creștere a timpului de execuție între 120% și 2760%, și în consumul de memorie între 20% și 660%, în funcție de programul analizat.

```
% clang -fsanitize=thread -g -O1 -o program tiny_race.c
% ./program
WARNING: ThreadSanitizer: data race (pid=19219)
  Write of size 4 at 0x7fcf by thread T1:
    #0 Thread1 tiny_race.c:4 (exe+0xa360)
  Previous write of size 4 at 0x7fcf by main thread:
    #0 main tiny_race.c:10 (exe+0xa3b4)
  Thread T1 (running) created at:
    #0 pthread_create tsan_interceptors.cc:705 (exe+0xc790)
    #1 main tiny_race.c:9 (exe+0xa3a4)
```

Fragment de cod 1: Exemplu de folosire ThreadSanitizer

De asemenea, ThreadSanitizer are nevoie de ajutor din partea compilatorului, deci

poate fi folosit doar cu compilatoare ce încorporează explicit suport pentru el. Deși această unealtă este folosită în companii mari din industrie [1], aceste dezavantaje îl fac inutilizabil pentru anumite proiecte obligate să folosească un anumit compilator care nu suportă ThreadSanitizer sau care sunt executate în contexte unde reducerea drastică a performanței nu este fezabilă nici măcar pentru teste sau simulări.

Prin comparație, o unealtă construită folosind proiectul SyncAnalysis face analiza evenimentelor *post-mortem*, și prin urmare nu duce la creșteri atât de mari în consumul de timp sau memorie în timpul execuției programului. În schimb fișierele de evenimente pot deveni destul de mari dacă programul analizat este executat mult timp sau capturează multe tipuri de evenimente.

1.3 Comparație cu Clang-Tidy

Clang-Tidy[4] este o unealtă ce efectuează analiză *statică* asupra codului sursă a unui proiect. Deoarece analiza se face static, fără a necesita execuția programului în sine, considerentele de performanță a programului sunt inexistente: programul compilat nu se schimbă din cauza folosirii uneltei, deci nu se aplică problema de o degradare a performanței sau o creștere a resurselor folosite din niciun punct de vedere.

Deși accentul nu este pus pe folosirea corectă sau eficientă a primitivelor de sincronizare în cel din urmă, comparația dintre proiectele *SyncAnalysis* și *Clang-Tidy* este aptă pentru că amândouă încearcă să creeze un sistem de unelte, mai degrabă decât să încerce să fie o unealtă de sine stătătoare. *Clang-Tidy* permite dezvoltatorilor să scrie noi analizori folosind o interfață elegantă de C++, în care este necesară strict implementarea analizei în sine, într-un mod declarativ și simplu, programul `Clang-Tidy` rezolvând în spate toate celelalte probleme pentru a duce la o unealtă completă.

În timp ce proiectul *Clang-Tidy* permite scrierea cu efort mic a unor analizori statici pentru codul sursă de C și C++, *SyncAnalysis* face asta pentru analizori *post-mortem*.

2 Tehnologii folosite

2.1 Limbajul C11 și biblioteca standard C

Limbajul C[5] este folosit pentru implementarea bibliotecii de înregistrare de evenimente și a uneia din bibliotecile de integrare. Am ales limbajul pentru componenta aceasta din proiect deoarece portabilitatea bibliotecii este o prioritate: această bibliotecă trebuie să poată fi folosită în orice program scris în aproape orice limbaj și rulând pe aproape orice sistem de operare. De asemenea este important ca biblioteca să fie cât mai eficientă, pentru a nu interfera prea tare cu execuția programului client. Această combinație de cerințe este îndeplinită perfect de limbajul C. Am ales versiunea C11 pentru a avea acces la modulul `<stdatomic.h>` din biblioteca standard, folosit în implementarea bibliotecii de înregistrare de evenimente din motive de performanță.

2.2 Limbajul C++17 și biblioteca standard C++

Limbajul C++[6] este folosit pentru implementarea majorității proiectului. Programul `SyncAnalysis`, una dintre bibliotecile de integrare date ca exemplu, toate exemplele de analizoare și toate testele automate au codul sursă scris în acest limbaj.

Înafară de preferință personală, argumentul principal pentru folosirea limbajului C++ este interoperabilitatea bună cu limbajul C. Astfel, a fost ușor de refolosit cod din biblioteca de înregistrare de evenimente în programul SyncAnalysis pentru părțile de deserializare a fișierului de evenimente, în timp ce restul programului și analizorii au putut fi scriși într-un limbaj modern și foarte eficient. Biblioteca standard C++ este folosită peste tot în implementarea programului și în implementarea analizorilor. Printre modulele folosite cel mai des se numără `<vector>`, `<map>` și `<string>`.

2.3 Compilatorul Clang

Limbajele C și C++, folosite în acest proiect, sunt limbaje a căror cod sursă se compilează în cod nativ, executabil direct pe hardware. Pentru a efectua această transformare în proiectul curent, am folosit exclusiv compilatorul Clang[7], pentru portabilitatea acestuia între Mac OS și Linux, viteza de compilare și integrarea elegantă cu alte unelte de care am avut nevoie în dezvoltare cum ar fi Clang-Tidy[4] și ThreadSanitizer[1].

2.4 Uneltele de dezvoltare CMake și CTest

Pentru proiecte mari de C++, invocarea manuală a compilatorului pentru a recompila fiecare fișier modificat devine repede impractică. CMake[8] este un *build system*: prin scrierea unui fișier concis de configurare ce descrie structura fișierelor de cod sursă din proiect, CMake automatizează recompilarea tuturor fișierelor impactate de o schimbare, ducând astfel la o experiență plăcută de dezvoltare a proiectului.

CTest este o componentă a proiectului CMake folosită pentru rularea automată a testelor. Din nou, cu creșterea numărului de teste ce trebuie efectuate după o schimbare în cod, devine impractică invocarea manuală a acestora. CTest ajută la automatizarea acestui procedeu, din nou spre a facilita o experiență plăcută de dezvoltare.

2.5 Biblioteca pthread

Biblioteca `pthread`[9] este interfața pentru crearea și manipularea mai multor fire de execuție într-un proces. Este parte din specificația standard POSIX pentru o interfață portabilă a sistemelor de operare.

Biblioteca `pthread` este folosită în ambele bibliotecile de integrare, una din acestea având ca scop chiar simularea unei părți din interfața publică oferită în antetul bibliotecii, `<pthread.h>`.

De asemenea, `pthread` este folosită și în implementarea bibliotecii de capturare de evenimente pentru crearea unui fir de execuție separat care serializează evenimentele capturate în fișierul de evenimente.

2.6 Biblioteca DL

Biblioteca `DL`[10] este interfața descrisă în specificația standard POSIX pentru găsirea, încărcarea și utilizarea bibliotecilor dinamice. Această bibliotecă este folosită în proiect în programul independent `SyncAnalysis` pentru a încărca analizorii specificați de client.

2.7 Biblioteca libunwind

Biblioteca `libunwind`[11] este o facilitate portabilă și eficientă pentru a observa stiva de apeluri de funcții în timpul execuției unui program.

Biblioteca este folosită pentru a înregistra stiva de apeluri de funcții când se capturează un eveniment. Folosirea bibliotecii acesteia este preferabilă altor variante mai convenabile ce oferă direct simbolurile asociate funcțiilor din stivă pentru că scopul bibliotecii de capturare de evenimente este să fie cât mai eficientă, iar obținerea acestor simboluri se poate face din programul `SyncAnalysis`, după terminarea programului client, doar pentru stivele de execuție necesare în rapoarte.

2.8 Programele addr2line, respectiv atos

După cum s-a menționat și în secțiunea pentru `libunwind`, biblioteca de capturare a evenimentelor serializează doar lista de adrese din stiva de execuție în datele evenimentului în timpul execuției programului client. Programul `SyncAnalysis` transformă apoi respectivele adrese în simboluri citibile de către client (nume de funcții sau fișier și linie în codul sursă unde este făcut apelul), pentru a fi atașate rapoartelor.

Pentru multe dintre sistemele de operare ce respectă standardul POSIX, problema este rezolvată de programul `addr2line` oferit de GNU ca parte a proiectului `binutils`[12]. Programul primește o listă de adrese de memorie ce reprezintă stiva de apeluri de funcții și executabilul din care acestea au provenit și afișează exact lista de simboluri asociate, și, dacă se pot afla, numele fișierului și linia unde se află codul sursă pentru adresa respectivă din program. Pentru că proiectul a fost dezvoltat în mare parte pe un calculator cu sistemul de operare Mac OS (cu kernel-ul Darwin) unde programul `addr2line` nu dă rezultate satisfăcătoare, când se rulează programul `SyncAnalysis` pe Mac OS, acesta folosește `atos`[13] pentru a îndeplini aceleași sarcini.

2.9 Biblioteca mcga-cli

Programul `SyncAnalysis` acceptă multe argumente de configurare în linia de comandă (de exemplu directoarele în care să caute analizori, reguli de includere și excludere pentru ce analizori dintre cei găsiți să fie folosiți, dacă să afișeze sau nu informații auxiliare pentru găsirea de erori în interiorul executabilului în sine și altele).

Astfel, argumentele din linia de comandă sunt citite și interpretate folosind o bibliotecă externă, `mcga-cli`[14]. Această bibliotecă a fost dezvoltată tot de autorul proiectului `SyncAnalysis` descris aici, dar este un proiect independent de acesta.

2.10 Git și GitHub

Git[15] este un sistem de versionare și menținere a istoricului de modificări. A fost folosit în dezvoltarea acestui proiect exact cu acest scop: de a menține un istoric central al tuturor modificărilor făcute asupra codului sursă al proiectului și al acestui document.

GitHub[16] este o platformă comercială ce oferă un server de git și o interfață web pentru a putea naviga ușor prin istoric. *Repository*-ul de git unde se găsește codul sursă al acestui proiect este salvat pe serverul de git oferit de GitHub, și este accesibil la adresa `https://github.com/darius98/sync-analysis`.

2.11 Biblioteca Google Benchmark

Deoarece o metrică importantă a bibliotecii de capturare de evenimente este performanța, proiectul include și un tabel de măsurători făcute pentru a estima latența și debitul acestei biblioteci (măsurătorile se găsesc în secțiunea **3.2.6**). Pentru efectuarea acestor măsurători a fost folosită biblioteca Google Benchmark[17], un proiect *open-source* ce are ca scop exact asta: efectuarea de măsurători exacte de performanță cu efort minim din partea programatorului.

3 Descrierea proiectului

3.1 Instalarea și compilarea proiectului

Proiectul poate fi descărcat, instalat și folosit pe orice calculator cu sistemul de operare o distribuție de GNU/Linux sau Mac OS și cu acces la internet. De asemenea, calculatorul trebuie să aibă instalate următoarele programe:

- `as, ar, ld, addr2line` – de obicei aceste programe vin preinstalate cu orice distribuție de GNU/Linux sau Mac OS. Ele fac parte din colecția de utilitare `binutils`[12] și sunt disponibile gratuit pentru descărcare pe website-ul oficial GNU[18] în cazul în care nu sunt deja instalate.
- `CMake`[8] – versiunea `3.15` sau mai nouă
- `make` – orice versiune compatibilă cu versiunea de CMake instalată
- `atos`[13] – doar pentru Mac OS, unde este mereu preinstalat
- un compilator de C și C++ care suportă standardele C11, respectiv C++17 în întregime (de exemplu `Clang10`[7])
- `git`[15] – minim versiunea `2.0`

Proiectul este disponibil pe internet, stocat pe server-ul GitHub. Astfel, acesta poate fi descărcat și instalat prin următoarea secvență de comenzi `shell`:

```
1 git clone https://github.com/darius98/sync-analysis.git
2 cd sync-analysis/
3 git submodule update --init
4 cmake .
5 make all
6 make install
```

Fragment de cod 2: Instrucțiuni pentru instalarea proiectului

Comanda 1 descarcă *repository*-ul de git al proiectului de pe server-ul GitHub. Acesta conține tot codul sursă asociat proiectului. Toate comenzile mai departe trebuie executate

în directorul de bază al proiectului, așa că comanda 2 schimbă directorul curent pentru restul comenzilor.

Deoarece *repository*-ul include biblioteca `mcga-cli`[14] ca submodul, aceasta trebuie descărcată separat după inițializarea logicii de git pentru submodule. Comanda 3 execută ambele acțiuni.

Odată ce *repository*-ul principal și submodulul `mcga-cli` au fost descărcate cu succes, proiectul poate fi compilat prin comenzile 4-5. După compilarea proiectului, acesta poate fi instalat în sistem folosind comanda 6. Această comandă adaugă fișiere noi în directoarele de sistem. De obicei, aceste fișiere se adaugă în directorul `/usr/local/`, deci pe multe sisteme această comandă va avea nevoie de permisiuni de administrator. Soluția, de exemplu pe sistemul de operare Ubuntu, este de a rula comanda folosind `sudo` (`sudo make install`). Fișierele ce se adaugă în sistem prin instalare sunt următoarele:

- biblioteca pentru capturare de evenimente: `lib/libsync_analysis.so` (pe Mac OS extensia este `.dylib`) și interfața C a acesteia în fișierul de interfață `include/sync_analysis.h`
- programul independent `SyncAnalysis` în fișierul `bin/sync_analysis`
- interfața C++ pentru dezvoltarea analizorilor `include/syan_analyzer_api/`
- biblioteca pentru integrare `lib/libcxxsync.so` (respectiv `.dylib` pe Mac OS) și interfața C++ a acesteia în directorul `include/cxxsync/`
- biblioteca pentru integrare `lib/libsyan_pthread_shim.so` (pentru Mac OS extensia este `.dylib`)
- analizorii dați ca exemplu, în directorul `lib/syan-analyzers/`

3.2 Biblioteca de capturare a evenimentelor

Aceasta este prima și cea mai importantă dintre componentele proiectului. Pentru că această bibliotecă este încărcată direct în programul clientului, este proiectată să fie atât portabilă cât și performantă.

Pentru a maximiza portabilitatea, alegerea naturală de limbaj pentru bibliotecă este C[5]: pentru că limbajul are o interfață binară standardizată de ISO și implementată de majoritatea compilatoarelor folosite în industrie, împreună cu faptul că majoritatea limbajelor de nivel înalt implementează *foreign-function-interface* cu C, aplicațiile pot beneficia de această bibliotecă aproape indiferent de limbajul în care au fost programate sau platforma pe care sunt executate.

Deși există variante mai bune pentru performanță cum ar fi C++ sau *assembly*, C este un limbaj suficient de eficient pentru a fi cea mai bună alegere, luând în considerare avantajul copleșitor de portabilitate.

3.2.1 Interfață

Pentru că această bibliotecă este una generică, interfața este intenționat minimală: sunt expuse în mod public 3 funcții și 4 constante numerice. În limbajul C, interfața poate fi descrisă astfel:

```
void syan_capture_event(int event_type, void* object);

void* syan_initialize_event(int event_type);
void syan_finalize_event(void* event, void* object);

enum {
  SA_EV_CREATE = 1 << 28,
  SA_EV_DESTROY = 1 << 29,
  SA_EV_THREAD = 1 << 30,
  SA_EV_THREAD_ON_CREATE = SA_EV_THREAD | SA_EV_CREATE,
};
```

Fragment de cod 3: Interfața bibliotecii pentru capturare de evenimente

Această bibliotecă expune ca funcționalitate principală capturarea evenimentelor, așa că interfața publică importantă este funcția `syan_capture_event`, care îndeplinește exact acest scop. Toate celelalte componente ale interfeței expuse sunt pentru a trata cazuri particulare mai complexe sau a standardiza tipuri de evenimente pe care se bazează și programul `SyncAnalysis`, și sunt descrise mai jos. Exemplul principal de utilizare

după care a fost proiectată funcția `syan_capture_event` se poate vedea în Fragmentul de cod 4.

```
#define EVENT_MUTEX_BEFORE_LOCK 1331
#define EVENT_MUTEX_AFTER_LOCK 1332

pthread_mutex_t* m;
void f() {
    syan_capture_event(EVENT_MUTEX_BEFORE_LOCK, m);
    pthread_mutex_lock(m);
    syan_capture_event(EVENT_MUTEX_AFTER_LOCK, m);
}
```

Fragment de cod 4: Exemplul folosit în proiectarea interfeței

Funcția este apelată cu 2 parametri, unul pentru tipul de eveniment capturat și celălalt pentru identitatea obiectului țintă al evenimentului. Astfel, în analiza post-mortem a unui program se poate face o corelare între mai multe evenimente care au avut loc pe același obiect țintă. Cum obiectele în limbajul C au în mod implicit o identitate dată de adresa de memorie unde sunt stocate, este natural ca parametrul care identifică obiectul să aibă tipul de date **void***, care reprezintă în mod convențional o adresă oarecare de memorie. Apare totuși o problemă: aceași adresă de memorie poate fi refolosită de-a lungul execuției unui program, pentru a stoca mai multe obiecte logice diferite (după dealocarea primului obiect, la alocarea unui obiect nou se poate folosi aceași memorie; un exemplu comun este segmentul de memorie stivă al programului). Pentru a evita confuzii generate de această problemă, biblioteca expune două constante numerice `SA_EV_CREATE` și `SA_EV_DESTROY`, ce acționează ca *biți indicatori* în tipul unui eveniment: dacă în scrierea în baza 2 a tipului unui eveniment bitul `SA_EV_CREATE` (respectiv `SA_EV_DESTROY`) este 1, evenimentul este considerat unul care marchează crearea un obiect (respectiv distrugerea unui obiect).

Această convenție este folosită și de către programul `SyncAnalysis`, care se folosește de aceste evenimente pentru a reconstrui o bază de date de obiecte active în fiecare punct al execuției programului client, a include evenimentul ce marchează crearea unui obiect țintă în rapoarte și a curăța memorie după distrugerea unui obiect. Astfel, este recomandat ca

orice bibliotecă de integrare să folosească acești indicatori pentru a captura evenimente ce marchează crearea și distrugerea tuturor obiectelor de interes pentru evenimentele capturate de respectiva bibliotecă:

```
enum {
    EVENT_MUTEX = 1u << 15,
    EVENT_MUTEX_CREATE = EVENT_MUTEX | SA_EV_CREATE,
    EVENT_MUTEX_DESTROY = EVENT_MUTEX | SA_EV_DESTROY,
};
```

Fiecare eveniment capturat conține identitatea firului de execuție care l-a capturat. Este util atunci să se raporteze câte un eveniment care marchează crearea fiecărui fir de execuție, pentru a putea pune evenimentele capturate de un anumit fir de execuție într-un context. Astfel, biblioteca în sine când se încarcă capturează ca prim eveniment unul care marchează crearea firului de execuție care a încărcat biblioteca. Pentru că biblioteca raportează în interiorul său un eveniment de creare de fir de execuție, aceasta stabilește un număr `SA_EV_THREAD_ON_CREATE` (se observă în interfață că acest număr conține bitul `SA_EV_CREATE`) care să reprezinte evenimentele de acest tip. Orice bibliotecă de integrare trebuie să dea același număr pentru argumentul `event_type` când apelează funcția `syan_capture_event` pentru a raporta crearea unui fir nou de execuție.

Cu experiența implementării bibliotecilor de integrare s-a observat că atunci când se raportează crearea unui nou fir de execuție folosind biblioteca `pthread`, identitatea obiectului țintă al evenimentului este cunoscută abia după ce a pornit noul fir de execuție:

```
pthread_t T;
pthread_create(&T, nullptr, func, arg);
syan_capture_event(SA_EV_THREAD_ON_CREATE, (void*)T);
```

Fragment de cod 5: Capturare incorectă a creării unui fir de execuție

Dacă evenimentul care marchează crearea firului de execuție T este capturat *după* crearea lui T, pot apărea inconsistențe în analiza post-mortem: firul de execuție T poate începe să captureze evenimente, care pot ajunge să apară în fișierul de evenimente înaintea evenimentului care marchează crearea lui T. Pentru a rezolva această situație, biblioteca expune celelalte două funcții `syan_initialize_event` și `syan_finalize_event` în interfața bibliotecii. Funcția `syan_initialize_event` crează evenimentul și îi

alocă locul în fișierul de evenimente *înainte* de crearea firului T. După crearea firului T, evenimentul este doar marcat ca fiind *finalizat*, dar este scris în fișier ca și cum în locul unde a fost mai înainte apelat `syan_initialize_event` ar fi fost apelat `syan_capture_event`, rezolvând problema ridicată mai devreme. Exemplul de mai sus se repară astfel:

```
pthread_t T;
void* ev = syan_initialize_event(SA_EV_THREAD_ON_CREATE);
pthread_create(&T, nullptr, func, arg);
syan_finalize_event(ev, (void*)T);
```

Fragment de cod 6: Capturare corectă a creării unui fir de execuție

Se observă acum că funcția `syan_capture_event` are o implementare evidentă pe baza celor două funcții nou introduse:

```
void syan_capture_event(int event_type, void* object) {
    void* ev = syan_initialize_event(event_type);
    syan_finalize_event(ev, object);
}
```

Fragment de cod 7: Implementarea funcției `syan_capture_event`

3.2.2 Structura unui eveniment

Un eveniment capturat de bibliotecă are aceași structură indiferent de tipul evenimentului sau de identitatea obiectului țintă al acestuia. Aceasta a fost proiectată astfel încât mai multe evenimente să poată fi stocate ușor și eficient într-o zonă continuă de memorie și serializate în format binar pe disc. În codul sursă C al bibliotecii, structura ce definește un eveniment se poate vedea în Fragmentul de cod 8.

Ordinea și dimensiunea în biți a fiecărui câmp a fost aleasă astfel încât pe platformele cele mai populare, unde arhitectura procesorului este pe 64 de biți și astfel dimensiunea unui `intptr_t` este 8 bytes, un eveniment să aibă dimensiunea exact 128 bytes, fără spațiu irosit din cauze precum *padding* sau *alignment*. Această dimensiune, fiind o putere a lui 2 ($2^7 = 128$) mai are un avantaj: dimensiunea paginii de RAM este în general tot o putere de 2, de obicei 4096 (2^{12}), și astfel pe o pagină de memorie încap exact 32 de

```
typedef struct {
    int32_t signature;
    int32_t event_type;
    int64_t timestamp;
    intptr_t thread_id;
    intptr_t object_id;
    intptr_t backtrace[12];
} SyanEvent;
```

Fragment de cod 8: Structura unui eveniment capturat

evenimente. Această proprietate este exploatată în bibliotecă pentru a obține viteze mari de serializare a evenimentelor în memorie, și mai apoi pe disc.

Câmpul `signature` este folosit intern pentru a împiedica biblioteca din a scrie un eveniment pe disc înainte de apelarea funcției `syan_finalize_event` pentru acel eveniment. Aceasta nu va scrie evenimentul pe disc până când nu citește în câmpul `signature` valoarea prestabilită 666013. Valoarea câmpului este 0 după apelul la funcția `syan_initialize_event`, dar evenimentul este deja pus în coada de evenimente ce trebuie scrise pe disc. Când este apelată funcția `syan_finalize_event`, valoarea este schimbată în 666013, permițând bibliotecii să continue scrierea evenimentelor din coadă pe disc.

Câmpurile `event_type` și `object_id` vor conține valorile date ca parametri de către client când acesta apelează funcția `syan_capture_event`, sau respectiv funcțiile `syan_initialize_event` și `syan_finalize_event`.

Câmpul `timestamp` este o reprezentare pe 64 de biți al numărului de nanosecunde scurse de la inițializarea bibliotecii până la momentul capturării evenimentului. Fișierul DUMP (descris în secțiunea **3.2.5**) conține în *header* momentul de timp al inițializării bibliotecii, astfel momentul real de timp când a avut loc evenimentul poate fi reconstruit post-mortem cu precizie de nanosecundă, iar stocarea numărului de nanosecunde pe 64 de biți permite capturarea evenimentelor într-un program cu durată de viață până la 146 de ani, suficient pentru orice scop practic.

Câmpul `thread_id` conține identitatea firului de execuție ce a capturat evenimentul. Acesta este obținut printr-un apel la funcția `pthread_self()`, descrisă în specificația POSIX pentru fire de execuție portabile[9]. Deși specificația nu precizează dacă tipul

de date returnat de `pthread_self()`, anume `pthread_t`, poate fi stocat cu succes într-o variabilă cu tipul de date `intptr_t`, în practică toate implementările bibliotecii `pthread` pentru platformele pe care a fost testată biblioteca implementează `pthread_t` ca pe un *pointer* la structura de control a firului de execuție (iar acest *pointer* poate fi stocat, conform definiției tipului de date `intptr_t`).

Câmpul `backtrace` conține adresele de întoarcere din stiva de execuție. Acestea sunt utile pentru ca în analiza post mortem să se poată reconstrui stiva de apeluri de funcții așa cum este descris în secțiunea **3.3.7** și să se poată construi rapoarte cât mai exacte despre condițiile în care s-a capturat un eveniment. Modul cum sunt capturate aceste adrese este descris mai pe larg în secțiunea **3.2.3**.

3.2.3 Stiva de execuție

După compilarea programelor în cod de asamblare, tot codul este structurat în funcții. Când o funcție apelează o altă funcție, aceasta trebuie să își stocheze valorile variabilelor locale în memorie și trebuie să îi transmită funcției apelate adresa de memorie unde să reîntoarcă controlul odată ce și-a terminat execuția. Zona de memorie unde funcțiile stochează aceste informații se numește segmentul de stivă al programului. Fiecare fir de execuție are propriul segment stivă, independent de celelalte fire.

În Figura 2 se poate vedea acest procedeu pentru arhitectura X86: în momentul apelului funcției `function_1` din funcția `main`, se crează un nou cadru în memoria stivă a programului (identificată pe desen prin culoarea albastru). La începutul acestui cadru se stochează adresa de memorie a instrucțiunii următoare apelului din funcția `main`. În timpul execuției funcției `function_1`, aceasta apelează o a doua funcție `function_2`. Se crează deci un al treilea cadru în memoria stivă (identificată pe desen prin culoarea verde), din nou stocându-se adresa de întoarcere. La finalul funcției `function_2`, aceasta invocă instrucțiunea `ret`, care întoarce controlul la adresa de memorie stocată în cadrul de stivă al funcției curente, adică înapoi la funcția `function_1`. Când și aceasta se termină prin invocarea instrucțiunii ret, controlul este întors funcției `main`. Dacă aceasta nu mai este apelată la rândul său de nicio altă funcție, instrucțiunea `ret` de la sfârșitul acesteia va încheia execuția programului, întorcând controlul sistemului de operare.

A se observa că explicația dată mai sus nu descrie în mod exact cum funcționează

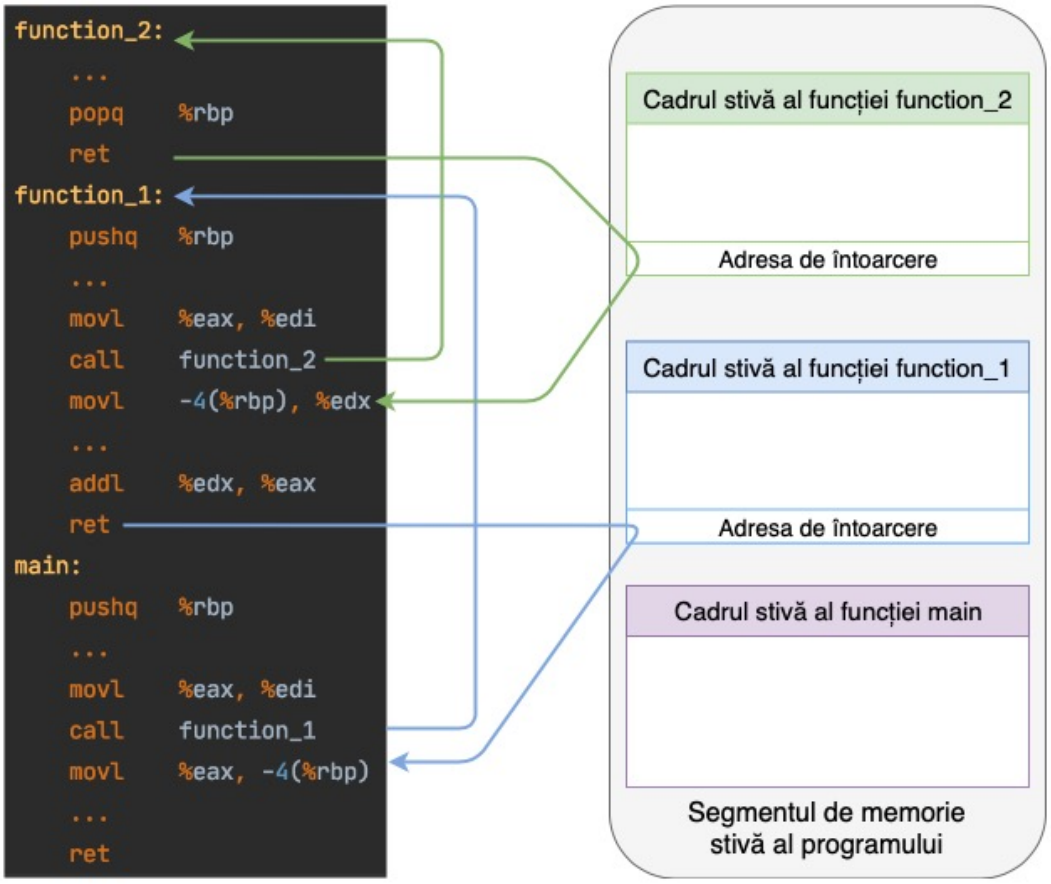

Figura 2: Stiva de execuție în arhitectura X86

procesoarele și programele moderne, însă putem ajunge la rezultate corecte pentru scopul nostru și fără a înțelege alte detalii sau complicații istorice ale stivei de execuție. Singurul detaliu demn de menționat este că în unele situații, compilatoarele de C și C++ pot decide să *optimizeze* programul prin omiterea stocării adresei de întoarcere pe stivă, caz în care reconstruirea stivei de execuție devine imposibilă. Pentru compilatorul Clang[7], această optimizare poate fi oprită prin argumentul de compilare `-fno-omit-frame-pointer`.

Dacă funcția `function_2` face un apel la funcția `syan_capture_event` pentru a captura un eveniment, vrem să obținem lista de adrese de întoarcere din toate cadrele funcțiilor din stivă. Acest rezultat poate fi obținut folosind biblioteca *libunwind*[11], după cum se poate vedea în Fragmentul de cod 9.

```
#include <libunwind.h>
void syan_get_backtrace(intptr_t* backtrace) {
    unw_context_t ctx;
    unw_getcontext(&ctx);
    unw_cursor_t crs;
    unw_init_local(&crs, &ctx);
    int n_frames = 0;
    while (n_frames < 12 && unw_step(&crs) > 0) {
        unw_word_t ip;
        if (unw_get_reg(&crs, UNW_REG_IP, &ip) < 0) break;
        backtrace[n_frames++] = (intptr_t)ip;
    }
}
```

Fragment de cod 9: Capturarea stivei de execuție folosind `libunwind`

3.2.4 Coada de evenimente și serializarea

Am menționat deja mai devreme că un obiectiv în proiectarea acestei biblioteci este ca performanța programului analizat să fie impactată cât se poate de puțin. Trebuie luat în considerare astfel că scrierea evenimentelor pe disc este o operațiune lentă. Prima decizie luată în implementarea serializării este deci de a scrie evenimentele doar în memoria RAM în cadrul firelor de execuție ale programului client, pentru ca mai apoi un alt fir de execuție creat de bibliotecă să copieze datele din memoria RAM pe disc. Pentru a obține rezultate satisfăcătoare de performanță, am proiectat o *coadă de evenimente* (intern numită `SyanBuffer`) specifică pentru această bibliotecă. Interfața acestei cozi se poate vedea în Fragmentul de cod 10.

Prima proprietate importantă a acestei cozi este că nu conține niciun mecanism explicit de sincronizare, ci doar câteva variabile cu tipul de date *atomic*. Minimizarea latenței punerii evenimentelor în coadă este mai importantă decât eficiența citirii din coadă, pentru a interfera cât mai puțin cu firele de execuție ale programului client. Când un astfel de fir capturează un eveniment, memoria folosită pentru a stoca informațiile acelui eveniment este direct memorie din câmpul `storage` al paginii curente din coadă (mai exact pagina `b->pages[b->pages_back]`). Asta omite necesitatea de a apela funcții precum `malloc` pentru a obține memorie pentru fiecare eveniment individual sau `memcpy` pentru a muta evenimentul în coadă abia după ce acesta este complet format.

```
#define SYAN_BUFFER_PAGE_SIZE 256
#define SYAN_BUFFER_NUM_PAGES 1024
typedef struct {
    SyanEvent storage[SYAN_BUFFER_PAGE_SIZE];
    int_fast32_t storage_front;
    atomic_int_fast32_t storage_back;
} SyanBufferPage;
typedef SyanBufferPage* SyanBufferPagePtr;
typedef struct {
    SyanBufferPagePtr pages[SYAN_BUFFER_NUM_PAGES];
    int_fast32_t pages_front;
    atomic_int_fast32_t pages_back;
} SyanBuffer;
int syan_buffer_init(SyanBuffer** b);
SyanEvent* syan_buffer_acquire_event_slot(SyanBuffer* b);
SyanBufferPagePtr syan_buffer_get_front_page(SyanBuffer* b);
void syan_buffer_release_front_page(SyanBuffer* b);
```

Fragment de cod 10: Interfața cozii de evenimente folosite în bibliotecă

Așadar funcția `syan_initialize_event` obține memoria necesară pentru stocarea evenimentului apelând funcția `syan_buffer_acquire_event_slot`. Aceasta este implementată folosind un algoritm *lock-free* bazat pe operațiile de *compare-and-swap* și *atomic-fetch-add* pe variabilele atomice `pages_back` și respectiv `storage_back` din acea pagină. Implementarea este similară cu cele descrise în articolele *Lock-free linked lists using compare-and-swap* [19] și *A practical nonblocking queue algorithm using compare-and-swap* [20].

După ce un eveniment a fost scris în coadă, acesta trebuie scris mai departe pe disc. După cum am spus mai sus, biblioteca crează un fir de execuție special pentru această operație. Fiind un singur fir care se ocupă de scrierea pe disc a evenimentelor de la începutul cozii, se poate observa și în interfață că variabilele `pages_front` și `storage_front` nu sunt atomice: aceasta sunt atât scrise cât și citite doar de firul intern al bibliotecii, așa că nu necesită sincronizare. Am menționat și mai devreme cum un eveniment pus în coadă nu este scris pe disc până când câmpul `signature` al evenimentului nu are valoarea corectă, scrisă de apelul funcției `syan_finalize_event`. În practică, firul de execuție al bibliotecii citește aceste semnături din evenimentele de

la începutul cozii folosind operația *atomic-load*, pentru că se așteaptă ca scrierea să fie făcută de firul de execuție care raportează evenimentul.

Firul de execuție intern bibliotecii funcționează pe bază de iterații: o iterație scrie cât de multe evenimente poate din coadă pe disc. Dacă iterația a scris cu succes date pe disc, se trece direct la următoarea iterație. Altfel, înseamnă că nu mai există evenimente noi capturate, și firul apelează funcția de sistem `usleep` pentru a permite firelor programului client să își continue execuția înainte de a începe o nouă iterație. Pentru a afla ce evenimente noi trebuie scrise pe disc, firul verifică valorile câmpurilor `storage_front` și `storage_back` de pe cea mai veche pagină a cozii, obținută printr-un apel la funcția `syan_buffer_get_front_page`. Dacă valorile acestea sunt diferite, atunci toate evenimentele dintre sunt fie evenimente noi capturate, fie evenimente în curs de capturare. Firul așteaptă finalizarea tuturor acestor evenimente pe modelul *busy-wait*, deoarece se asumă că apelurile funcțiilor `syan_initialize_event` și `syan_finalize_event` se fac la puțin timp unul după celălalt. S-a experimentat aici și cu alte modele de așteptare (`pthread_sched_yield`, `usleep`) și s-a observat empiric că *busy-wait* se comportă cel mai bine în practică. După ce toate aceste evenimente sunt finalizate, ele sunt scrise împreună pe disc printr-un singur apel al funcției standard C `fwrite`. Dacă prin scrierea acestor evenimente s-au scris toate evenimentele din pagină, aceasta este eliberată. Altfel, se modifică doar câmpul `storage_front` al paginii pentru a marca până unde au fost scrise evenimentele din pagină.

3.2.5 Fișiserului DUMP

Am menționat în repetate rânduri că evenimentele capturate în timpul execuției sunt scrise într-un fișier, denumit în continuare fișierul DUMP. Dar acest fișier nu conține doar evenimentele, ci are și un antet ce conține informații în plus despre execuție, utile pentru analiza post-mortem a evenimentelor:

- momentul când a fost inițializată biblioteca (o aproximare bună a momentului când a început execuția programului)
- calea către executabilul invocat
- argumentele din linia de comandă date executabilului
- adresa la care a fost încărcat executabilul

Momentul când a fost inițializată biblioteca este important pentru a putea reconstrui momentul de timp când a fost capturat un eveniment, pentru că valoarea câmpului `timestamp` din fiecare eveniment este egală cu numărul de nanosecunde ce au trecut între acest moment de inițializare și momentul capturării evenimentului.

Calea către executabiul invocat și argumentelee din linia de comandă sunt utile pentru a putea identifica ce reprezintă acest fișier DUMP. Nu sunt folosite de către programul `SyncAnalysis` decât pentru a fi afișate în raportul de analizare.

Adresa de încărcare a executabilului este importantă pentru a putea interpreta adresele din `backtrace`-urile evenimentelor. Mai multe detalii despre acest procedeu se găsesc în secțiunea **3.3.7**.

Fișierul DUMP este creat și deschis pentru scriere printr-un apel la funcția standard C `fopen("sync_analysis.dump", "wb")`. Conform specificației acestei funcții, dacă un fișier cu același nume exista deja, versiunea anterioară este ștearsă și un nou fișier gol este creat oricum. Apelul la `fopen` se face la inițializarea bibliotecii, împreună cu scrierea antetului și pornirea firului intern de execuție. În continuarea antetului sunt scrise direct evenimentele, unul după celălalt, serializate în formatul binar descris în secțiunea **3.2.2**.

3.2.6 Măsurători de performanță

Performanța bibliotecii se împarte în două metrici diferite: timpul necesar capturării unui eveniment (metrica de *latență*) și volumul de evenimente ce poate fi serializat într-o perioadă de timp (metrica de *debit*). În Figura 3 se pot vedea două tabele cu măsurătorile făcute cu ajutorul bibliotecii `Google Benchmark`[17] pentru ambele metrici interesante.

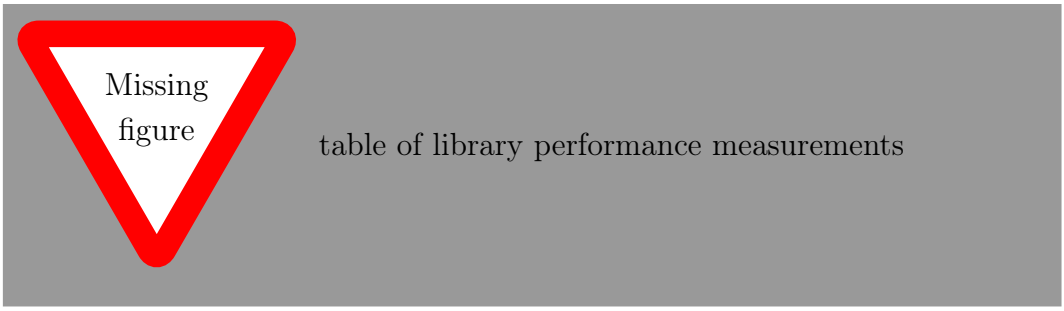

Figura 3: Măsurători de performanță a bibliotecii

Metrica de *latență* este măsurată în *nanosecunde* (cât durează o operație) și depinde

de numărul de fire de execuție ce capturează evenimente concurent, pentru că operațiile *compare-and-swap* implicate în acest procedeu au șansă mai mare să eșueze cu cât sunt mai multe fire care încearcă să le execute în același timp.

Metrica de *debit* este măsurată în *MB/secundă* (câte date despre evenimente pot fi serializate pe disc într-o perioadă de timp) și depinde de volumul de evenimente capturate și plasate în coada de evenimente.

3.3 Programul de analiză post-mortem SyncAnalysis

Programul `SyncAnalysis` este a doua componentă a sistemului de creare de unelte. Acesta se ocupă de a citi evenimentele din fișierul DUMP scris de biblioteca de capturare de evenimente și a le analiza prin intermediul unor analizori scriși de client.

Acest program are ca scop rezolvarea unor probleme ce fac scrierea de unelte de analiză *post-mortem* dificilă: parsarea în mod eficient a evenimentelor capturate în timpul execuției, acces ușor la informații de interes (cum ar fi indentitatea obiectelor țintă a evenimentelor) și scrierea de rapoarte ce conțin toate informațiile necesare pentru rezolvarea problemelor descoperite prin analiză. Prin folosirea acestui program, scrierea unui nou *analizor* (componentă a unei unelte de analiză ce diagnostichează un anumit tip de problemă) trebuie să se rezume la a implementa strict algoritmul de diagnosticare a problemei și nimic altceva. În secțiunea **4.3** se pot vedea câteva exemple de astfel de analizori ce rezolvă probleme reale, de multe ori implementate în mai puțin de 100 de linii de cod.

3.3.1 Argumentele pentru linia de comandă

Programul `SyncAnalysis` este proiectat pentru a fi invocat de la linia de comandă:

```
% sync_analysis -b program sync_analysis.dump
```

Primul lucru pe care îl face programul este să interpreteze argumentele date în comanda de invocare (în cazul de mai sus `"-b"`, `"program"` și `"sync_analysis.dump"`). În general, forma acestor argumente este următoarea:

```
% sync_analysis [-optiuni] calea/la/sync_analysis.dump
```

Argumentul pentru calea la fișierul `sync_analysis.dump` este singurul obligatoriu pentru funcționarea executabilului. Acesta indică fișierul DUMP ce trebuie analizat.

Toate celelalte argumente reprezintă *opțiuni*, și pentru a urma tiparul executabilelor preinstalate pe sistemele GNU/Linux (de exemplu programele din `binutils`[12]), acesta încep cu - sau --. Pentru interpretarea argumentelor din linia de comandă în aceste opțiuni s-a folosit biblioteca `mcga-cli`[14]. Aceasta oferă un mod elegant de a defini în opțiunile sub formă de variabile în codul sursă, a căror valoare este modificată conform argumentelor date în linia de comandă, indiferent de ordinea în care apar acestea. De asemenea, biblioteca generează și afișează automat o listă a tuturor opțiunilor pe care le poate primi programul și descrieri ale acestora într-un format ușor de citit prin invocarea programului cu un singur argument, `--help` sau `-h`.

Cea mai importantă dintre opțiuni este `--binary=calea/programului/client` (sau varianta mai scurtă `-b calea/programului/client`), folosită pentru a indica fișierul executabil prin a cărui execuție s-a generat fișierului `sync_analysis.dump` analizat. Această opțiune este necesară pentru a reconstrui lista simbolurilor funcțiilor din stiva de execuție a capturării unui eveniment dintr-un raport. Detalii despre cum se face această reconstrucție se găsesc în secțiunea **3.3.7**.

O altă opțiune importantă pentru clienții ce vor să invoce programul în procedeele lor de testare și integrare automată este `--report-file` (sau varianta scurtă `-r`). Folosirea acestei opțiuni face ca rapoartele să fie scrise în fișierul specificat în loc să fie afișate în consolă (cum se întâmplă în mod implicit).

Opțiunile `--debug` și `--print-header` controlează afișarea unor informații în plus despre execuția programului `SyncAnalysis` în sine. Sunt opțiuni utile pentru dezvolatatorii de analizori, care pot astfel observa greșeli în modul cum au programat sau integrat analizorii în program, sau pot chiar observa și semnala probleme cu funcționarea programului propriu-zis.

3.3.2 Înregistrarea analizorilor

După interpretarea opțiunilor, programul încarcă analizorii, urmând indicațiile date de opțiunile `-A` și `-a`. Opțiunea `-A` este folosită pentru a specifica directoare în care să se caute analizori. Un fișier găsit în oricare din directoarele specificate prin această opțiune este considerat un *analizor* dacă respectă următoarele condiții:

- Numele fișierului începe cu prefixul `libsyan-analyzer-` și are extensia `.so` (sau `.dylib` pe Mac OS);

- Fișierul poate fi încărcat cu succes printr-un apel la funcția `dlopen` (este un fișier în formatul ELF[21] ce reprezintă o bibliotecă dinamică);
- Simbolul `"syan_analyzer"` poate fi găsit în biblioteca reprezentată în fișier printr-un apel la funcția `dlsym` și simbolul este asociat unui șir de caractere;

Șirul de caractere asociat simbolului `"syan_analyzer"` este numele analizorului. Acest nume este folosit pentru a stabili dacă analizorul este folosit sau nu. Opțiunea `-a` specifiă reguli pentru ce nume trebuie să aibă un analizor pentru a fi folosit sau ignorat: *ultima* regulă dintre cele date prin opțiunea `-a` pe care o respectă numele unui analizor trebuie să fie una *pozitivă* pentru ca analizorul să fie folosit. O regulă este un *glob* ce conține doar litere, cifre, `-`, `_` sau `*`. Numele unui analizor *respectă* regula dacă aparițiile caracterului `*` din regulă pot fi înlocuite prin secvențe de alte caractere permise astfel încât să se obțină numele analizorului (de exemplu, numele `"redundant-recursive-mutex"` respectă regulile `"redundant-*"` sau `"*mutex"`, dar nu `"rwlock-*"`). O regulă este *negativă* dacă primul său caracter este `-` (și atunci primul caracter `-` este ignorat în verificarea regulii), altfel este *pozitivă*.

Acest mod de a filtra analizorii pe bază de listă de reguli *glob* este inspirat din modul de folosire al Clang-Tidy[4], care folosește mai mult sau mai puțin același algoritm de filtrare pe baza unui argument numit `-checks`.

3.3.3 Procedeul de analiză

După încărcarea analizorilor, începe procedeul de analiză propriu-zis. Mai întâi se execută codul de inițializare înregistrat de fiecare analizor. Reamintim aici că analizorii sunt încărcați de program ca biblioteci dinamice prin apeluri manuale la funcția `dlopen` din biblioteca de sistem `DL`[10]. Dacă biblioteca unui analizor expune public simbolul `"syan_analyzer_start_up"`, acesta trebuie să fie asociat unei funcții ce nu primește parametri, și această funcție este apelată acum, la începutul analizei.

După inițializarea analizorilor, programul citește evenimentele din fișierul DUMP și le expune unul câte unul analizorilor, în ordinea din fișier. Următoarea funcție este disponibilă analizorilor:

```
namespace syan { syan::Event current_event(); }
```

Pentru fiecare eveniment, acesta este pus într-o variabilă globală ce poate fi accesată de analizori prin funcția de mai sus. Apoi, pentru fiecare analizor a cărui bibliotecă expune public simbolul `"syan_analyzer_on_event"`, acesta trebuie să fie asociat unei funcții ce nu primește parametri, și această funcție este apelată.

După ce tot fluxul de evenimente din fișierul DUMP este epuizat, se caută în biblioteca fiecărui analizor simbolul `"syan_analyzer_shut_down"`, care trebuie din nou să fie asociat unei funcții ce nu primește parametri. Pentru analizorii unde acest simbol există, funcția asociată este apelată acum, după parcurgerea tuturor evenimentelor.

3.3.4 Clasa `syan::Event`

Am menționat deja că un analizor are acces la evenimentul curent procesat prin funcția `current_event()` care returnează o instanță a clasei `syan::Event`. Această clasă oferă informațiile despre evenimentul respectiv prin câteva metode simple:

```
int type() const;
ObjectId object() const;
ObjectId thread() const;
int64_t time_rel_to_program_start() const;
```

Aceste metode expun exact câmpurile ce au fost serializate în fișierul DUMP.

O instanță a acestei clase poate fi copiată de un analizor dacă acesta are nevoie de informația din acest eveniment pentru un posibil raport viitor. Pentru că un eveniment ocupă suficient de multă memorie ca să fie ineficient de copiat de mai mulți analizori, informația în sine despre eveniment este stocată o singură dată, iar clasa `syan::Event` acționează similar cu un *shared pointer*[22]: de fiecare dată când o instanță este copiată, se incrementează un contor pentru numărul de referințe, iar de fiecare dată când o instanță este distrusă este decrementat acest contor. Când contorul ajunge la 0, memoria asociată unicei instanțe a informației propriu-zise a evenimentului este dealocată. Copierea unui `syan::Event` este deci eficientă, și mai mulți analizori pot păstra referințe către un eveniment fără a avea un impact negativ asupra performanței analizei.

3.3.5 Crearea unui raport

Scopul principal al analizorilor este să găsească probleme prin observarea evenimentelor capturate. Când un analizor găsește o astfel de problemă, acesta trebuie să informeze

programul că trebuie creat un raport pentru problema respectivă. Următoarea funcție este disponibilă analizorilor:

```
namespace syan { syan::Report create_report(); }
```

Instanța clasei `syan::Report` astfel obținută controlează trimiterea unui raport către utilizator. Această clasă expune public următoarele 4 metode:

```
Report& set_level(Level level);
Report& set_description(std::string description);
Report& add_section(std::string description, Event event);
void send();
```

Un raport are un *nivel de severitate*, o *descriere* și zero sau mai multe *secțiuni*. Nivelul de severitate poate fi *informație* (`Level::info`), *avertisment* (`Level::warning`) sau *eroare* (`Level::error`), oferind o indicație simplă pentru tipul de diagnostic emis. În general se recomandă ca nivelul de *eroare* să fie folosit pentru diagnostice care mereu indică o greșeală ce trebuie rezolvată în program, nivelul de *avertisment* să fie folosit pentru diagnostice care *cel mai probabil* indică o problemă, dar pot fi totuși incorecte în anumite situații și nivelul de *informație* pentru orice alt tip de diagnostic.

Descrierea este mesajul principal al raportului: aceasta spune ce problemă a fost descoperită de analiză.

Secțiunile unui raport sunt atașamente la mesajul principal dat de *descriere*. O *secțiune* conține un eveniment și o descriere. Descrierea ar trebui să explice ce rol joacă acest eveniment în diagnostic sau de ce este important. Evenimentul este inclus pentru ca această secțiune a raportului să conțină momentul de timp când a avut loc acest eveniment (atât în valoare absolută cât și relativ la începutul programului), firul de execuție pe care s-a capturat evenimentul și funcțiile din stiva de execuție a capturării evenimentului (vezi Secțiunea **3.3.7**).

După construirea unui raport, acesta poate fi trimis manual folosind metoda `.send()`. Dacă nu se face apel la metoda `.send()` pentru un raport, acesta va apela automat metoda în destructor.

3.3.6 Baza de date a obiectelor active

Pe baza evenimentelor ce conțin în câmpul de tip *biții indicatori* `SA_EV_CREATE` și `SA_EV_DESTROY` (vezi secțiunea **3.2.1**), programul menține (independent de analizori)

o bază de date internă dar ușor accesibilă de obiecte *active* (care au fost create dar nu distruse), incluzând o listă cu toate firele de execuție create (acestea nu se consideră niciodată distruse). Pentru fiecare din aceste obiecte este stocat evenimentul care a creat obiectul. De asemenea unui obiect îi este asociat un identificator unic numeric (independent de cel din câmpul de `object_id`), pentru a putea fi folosit în rapoarte ca *numele* obiectului (fără riscul de a apărea coliziuni, risc ce există pentru `object_id`, după cum este descris în secțiunea **3.2.1**).

Cu ajutorul acestei baze de date, pentru orice secțiune adăugată unui raport, se mai adaugă automat până la două alte secțiuni: evenimentul ce a creat obiectul țintă al evenimentului din secțiune (dacă există în baza de date) și evenimentul ce a creat firul de execuție al evenimentului din secțiune (dacă există în baza de date).

Baza de date este accesibilă analizorilor prin următoarea funcție:

```
namespace syan { const syan::Database& database(); }
```

Analizorii se pot folosi de baza de date pentru a avea acces programatic la informațiile menționate mai sus, pentru care clasa `syan::Database` oferă metode de acces.

3.3.7 Afișarea simbolurilor din stiva de execuție

Cea mai importantă informație din rapoartele oferite de programul `SyncAnalysis` este lista funcțiilor din stiva de execuție pentru fiecare din secțiunile unui raport. Lista de funcții ajută utilizatorul să pună capturarea unui eveniment în context. Dar după cum este menționat în secțiunea **3.2.3**, evenimentul capturat conține doar lista de *adrese* în câmpul `backtrace`.

Programele `addr2line`[12] și `atos`[13] sunt invocate de programul `SyncAnalysis` pentru a transforma această listă de adrese în lista de simboluri de funcții. Teoretic, aceste programe primesc ca argumente în linia de comandă exact adresele și calea către programul din care au venit adresele și afișează numele funcțiilor și (dacă executabilul a fost compilat cu *debug symbols*) locația în codul sursă de unde provine adresa. Apare însă o problemă: din cauza măsurii de securitate pe care o iau sistemele de operare numită ASLR (*Address Space Layout Randomization*[23]), aceste adrese pot fi decalate cu o valoare fixă, egală cu adresa virtuală la care a fost încărcat programul în execuția originală. Această adresă este capturată de biblioteca pentru capturarea de evenimente și stocată în antetul fișierului DUMP (vezi Secțiunea **3.2.5**) în timpul execuției, pentru a fi disponibilă programului

`SyncAnalysis`. Acesta folosește adresa pentru a da programului `addr2line` adresele direct *normalizate* (prin decalarea acestora cu opusul decalajului inițial), respectiv pentru a da programului `atos` această adresă ca argumentul `-l` în comanda de invocare.

Sunt folosite aceste două programe diferite pentru că pe sistemul de operare Mac OS, programul `addr2line` nu oferă rezultate satisfăcătoare. Programul `atos` este menținut și recomandat oficial de Apple Inc. pentru a îndeplini aproximativ același scop, dar cu rezultate semnificativ mai exacte.

3.3.8 Interfața C++ pentru un analizor

Se recomandă scrierea analizorilor în limbajul C++, pentru că funcțiile expuse de program pentru analizori (`current_event`, `database` și `create_report`) respectiv clasele a căror instanțe sunt întoarse de aceste funcții (`Event`, `Database` și `Report`) sunt interfețe C++.

Un mod rudimentar de a scrie un analizor în C++ ce expune toate cele 4 simboluri căutate de program este văzut în Fragmentul de cod 11.

```
extern "C" const char* syan_analyzer = "analizorul-meu";
extern "C" void syan_analyzer_start_up() {
    // cod pentru initializare
}
extern "C" void syan_analyzer_on_event() {
    const syan::Event event = syan::current_event();
    // cod pentru procesarea unui eveniment
}
extern "C" void syan_analyzer_shut_down() {
    // cod pentru oprire
}
```

Fragment de cod 11: Interfața rudimentară a unui analizor

Motivul pentru care simbolurile pe care le caută programul în bibliotecile analizorilor sunt simboluri C și pentru care funcțiile ce trebuie asociate acestor simboluri au signatură C este pentru că interfața binară a limbajului C++ nu este standardizată oficial de ISO, pentrum cea de C.

Această interfață nu este cea mai elegantă pentru limbajul C++. Interfața de mai sus încurajează abuzul de variabile globale, încurajează eliberarea manuală a memoriei și alte practici periculoase pentru care C++ oferă soluții mai bune. Așadar, în interfața expusă analizorilor este oferit și un *macro* ce are ca scop construirea tuturor celor 4 simboluri necesare pe baza unei clase. Clasa trebuie să expună public un constructor fără niciun argument (din care se va genera simbolul `"syan_analyzer_start_up"`), un destructor (din care se va genera simbolul `"syan_analyzer_shut_down"`), o variabilă membru statică cu tipul de date **`const char*`** și numele `name` (din care se va genera simbolul `"syan_analyzer"`) și o suprascriere a operatorului `()` ce primește ca unic parametru un obiect de tipul `syan::Event` (din care se va genera simbolul `"syan_analyzer_on_event"`). Un exemplu de folosire al acestui *macro* poate fi văzut în Fragmentul de cod 12.

```
struct AnalizorulMeu {
    static constexpr const char* name = "analizorul-meu";

    AnalizorulMeu() { /* cod pentru initializare */ }
    ~AnalizorulMeu() { /* cod pentru oprire */ }

    void operator()(Event event) {
        // cod pentru procesarea unui eveniment
    }
};
SYAN_ANALYZER_CLASS(AnalizorulMeu);
```

Fragment de cod 12: Interfața pe bază de clasă a unui analizor

4 Exemplu de unealtă de analiză

Biblioteca pentru capturare de evenimente și programul de analiză `SyncAnalysis` sunt componente generice, refolosibile pentru orice fel de analiză post-mortem de evenimente. Vom arăta în acest capitol pașii luați spre a scrie o unealtă de analiză specifică pentru primitivele de sincronizare folosind infrastructura discutată în capitolul **3**. O unealtă este compusă din următoarele componente:

- O enumerație a tipurilor de evenimente de interes ce trebuiesc capturate în execuția unui program;
- Una sau mai multe biblioteci de integrare ce folosesc biblioteca descrisă în Secțiunea **3.2** pentru a captura evenimentele de interes;
- Unul sau mai mulți analizori ce să fie încărcați de programul `SyncAnalysis` descris în Secțiunea **3.3** pentru a analiza evenimentele capturate.

4.1 Tipuri de evenimente pentru primitive de sincronizare

Astfel, am creat o interfață C ce conține constante numerice pentru tipurile de evenimente specifice primitivelor de sincronizare, ce complimentează constantele numerice expuse în interfața bibliotecii pentru capturarea de evenimente.

Definim mai întâi câteva constante ce reprezintă *biți indicatori* din tipul evenimentului: tipul de obiect țintă al evenimentului (`SA_MUTEX` pentru *mutex*, `SA_REC_MUTEX` pentru *mutex recursiv* și `SA_RWLOCK` pentru *read-write lock*, similare cu `SA_EV_THREAD` pentru *fire de execuție*, definită în Secțiunea **3.2.1**) și acțiunea ce are loc pe acel obiect țintă (`SA_BEFORE_WR_LOCK`, `SA_AFTER_WR_LOCK` și altele, din nou similare cu constantele `SA_EV_CREATE` și `SA_EV_DESTROY` din Secțiunea **3.2.1**). Lista completă a tipurilor de evenimente capturate, împreună cu valorile acestora obținute din biții indicatori este următoarea:

```
SA_EV_THREAD_ON_JOIN        = SA_EV_THREAD | SA_JOIN
SA_EV_THREAD_ON_DETACH      = SA_EV_THREAD | SA_DETACH
SA_EV_MUTEX_ON_CREATE       = SA_MUTEX     | SA_EV_CREATE
SA_EV_MUTEX_ON_TRY_LOCK     = SA_MUTEX     | SA_TRY_WR_LOCK
SA_EV_MUTEX_BEFORE_LOCK     = SA_MUTEX     | SA_BEFORE_WR_LOCK
```

```
SA_EV_MUTEX_AFTER_LOCK      = SA_MUTEX     | SA_AFTER_WR_LOCK
SA_EV_MUTEX_ON_UNLOCK       = SA_MUTEX     | SA_WR_UNLOCK
SA_EV_MUTEX_ON_DESTROY      = SA_MUTEX     | SA_EV_DESTROY
SA_EV_REC_MUTEX_ON_CREATE   = SA_REC_MUTEX | SA_EV_CREATE
SA_EV_REC_MUTEX_ON_TRY_LOCK = SA_REC_MUTEX | SA_TRY_WR_LOCK
SA_EV_REC_MUTEX_BEFORE_LOCK = SA_REC_MUTEX | SA_BEFORE_WR_LOCK
SA_EV_REC_MUTEX_AFTER_LOCK  = SA_REC_MUTEX | SA_AFTER_WR_LOCK
SA_EV_REC_MUTEX_ON_UNLOCK   = SA_REC_MUTEX | SA_WR_UNLOCK
SA_EV_REC_MUTEX_ON_DESTROY  = SA_REC_MUTEX | SA_EV_DESTROY
SA_EV_RWLOCK_ON_CREATE      = SA_RWLOCK    | SA_EV_CREATE
SA_EV_RWLOCK_ON_TRY_RD_LOCK = SA_RWLOCK    | SA_TRY_RD_LOCK
SA_EV_RWLOCK_BEFORE_RD_LOCK = SA_RWLOCK    | SA_BEFORE_RD_LOCK
SA_EV_RWLOCK_AFTER_RD_LOCK  = SA_RWLOCK    | SA_AFTER_RD_LOCK
SA_EV_RWLOCK_ON_RD_UNLOCK   = SA_RWLOCK    | SA_RD_UNLOCK
SA_EV_RWLOCK_ON_TRY_WR_LOCK = SA_RWLOCK    | SA_TRY_WR_LOCK
SA_EV_RWLOCK_BEFORE_WR_LOCK = SA_RWLOCK    | SA_BEFORE_WR_LOCK
SA_EV_RWLOCK_AFTER_WR_LOCK  = SA_RWLOCK    | SA_AFTER_WR_LOCK
SA_EV_RWLOCK_ON_WR_UNLOCK   = SA_RWLOCK    | SA_WR_UNLOCK
SA_EV_RWLOCK_ON_DESTROY     = SA_RWLOCK    | SA_EV_DESTROY
```

Fragment de cod 13: Tipurile de evenimente pentru primitivele de sincronizare

4.2 Biblioteci de integrare

4.2.1 cxxsync (C++, începând cu C++11)

`cxxsync` este o bibliotecă de C++ proiectată pentru a fi un exemplu de integrare cu biblioteca de capturare de evenimente. Nu este neapărat cea mai bună soluție pentru integrarea în proiecte existente, deoarece are o interfață diferită de alte biblioteci existente, deci sunt necesare schimbări în codul sursă pentru integrare. Această bibliotecă este folosită de exemplu în testele analizorilor, după cum este descris în secțiunea **4.4**.

Biblioteca expune 4 clase principale, pentru cele 4 primitive ale căror evenimente ne interesează: `Thread` pentru *fir de execuție*, `Mutex` pentru *mutex*, `RecursiveMutex` pentru *mutex recursiv* și `RWLock` pentru *read-write lock*. În implementarea tuturor

claselor se folosesc funcții din biblioteca `pthread` și se capturează evenimente folosind funcția `syan_capture_event`, descrisă în Secțiunea **3.2.1**, și tipuri de evenimente dintre cele descrise în Fragmentul de cod 13.

Clasa `Thread` crează un fir de execuție printr-un apel al funcției `pthread_create` în constructor. Pentru a admite sintaxa elegantă de funcții *lambda* din C++11 pentru inițializarea firului de execuție, s-a folosit tehnica C++ cunoscută ca *type erasure*[24], mai exact transformarea unui tip de date necunoscut, primit ca *parametru de template*, într-unul cunoscut, în acest caz **void***(*)(**void***) (*pointer* la o funcție ce primește ca parametru un **void*** și returnează un **void***; acesta este tipul de date pe care îl primește funcția `pthread_create` ca parametru pentru a inițializa noul fir de execuție). Implementarea constructorului clasei `Thread` folosind această tehnică, și de asemenea folosirea funcțiilor `syan_initialize_event` și `syan_finalize_event` în locul funcției `syan_capture_event` din motivele precizate în secțiunea **3.2.1** se pot vedea în Fragmentul de cod 14.

```
template<class Lambda> Thread::Thread(Lambda lambda) {
    init(&erased_lambda<Lambda>, new Lambda(lambda));
}

void Thread::init(void* (*func)(void*), void* arg) {
    auto e = syan_initialize_event(SA_EV_THREAD_ON_CREATE);
    pthread_create(&pt_thread, nullptr, func, arg);
    syan_finalize_event(e, (void*)pt_thread);
}

template<class Lambda> void* erased_lambda(void* arg) {
    Lambda* lambda = static_cast<Lambda*>(arg);
    (*lambda)();
    delete lambda;
    return nullptr;
}
```

Fragment de cod 14: Tehnica *type-erasure* în clasa `Thread` din `cxxsync`

Clasele `Mutex` și `RecursiveMutex` expun fiecare câte 3 metode publice:

```
bool try_lock() noexcept;
void lock();
void unlock();
```

Aceste trei metode sunt implementate la fel în ambele clase, apelând în interior funcțiile `pthread_mutex_trylock`, `_lock` respectiv `_unlock` din biblioteca `pthread`. Clasa `RWLock` expune 6 metode publice (3 pentru *read-lock* și 3 pentru *write-lock*):

```
bool try_rd_lock() noexcept;
void rd_lock();
void rd_unlock();
bool try_wr_lock() noexcept;
void wr_lock();
void wr_unlock();
```

Metodele acestea sunt implementate prin apeluri la funcțiile echivalente din `pthread`: `pthread_rwlock_tryrdlock`, `_rdlock`, `_unlock`, `_trywrlock` și `_wrlock`.

Toate cele 4 clase principale din bibliotecă fac unul sau mai multe apeluri la funcția `syan_capture_event`, pentru a raporta diverse evenimente. Spre exemplu, clasa `RecursiveMutex` face în constructor următorul apel:

```
syan_capture_event(SA_EV_REC_MUTEX_ON_CREATE, this);
```

Metodele `try_lock`, `try_rd_lock`, `try_wr_lock` pot face unul sau două apeluri, în funcție de dacă primitiva a fost *locked* cu succes:

```
bool RWLock::try_rd_lock() noexcept {
  syan_capture_event(SA_EV_RWLOCK_ON_TRY_RD_LOCK, this);
  if (pthread_rwlock_tryrdlock(&pt_rwlock) != 0)
    return false;
  syan_capture_event(SA_EV_RWLOCK_AFTER_RD_LOCK, this);
  return true;
}
```

Fragment de cod 15: Metoda `RWLock::try_rd_lock` din `cxxsync`

Pe lângă cele 4 clase principale, biblioteca mai expune și 4 clase auxiliare atașate pentru a implementa tehnica *RAII*[25], foarte populară și utilă în programele moderne C++

(clasele sunt numite `MutexGuard`, `RecursiveMutexGuard`, `RWLockReadGuard` și `RWLockWriteGuard`). Aceste clase au o interfață foarte simplă (compusă doar din constructor și destructor) și o implementare la fel de simplă, expusă în Fragmentul de cod 16. Scopul acestor clase este doar ca o instanță cu *automatic storage* să fie creată unde se vrea un *lock* pe o primitivă și metoda asociată de *unlock* se va apela automat la sfârșitul blocului respectiv de cod (indiferent dacă se iese în mod natural prin terminarea blocului sau printr-un **`throw`**, **`goto`**, **`break`**, **`continue`** sau **`return`**).

```
struct MutexGuard {
    explicit MutexGuard(Mutex* m): mtx(m) { mtx->lock(); }
    ~MutexGuard() { mtx->unlock(); }
private:
    Mutex* mtx;
}
```

Fragment de cod 16: Clasa `MutexGuard` din `cxxsync`

4.2.2 pthread shim (C/C++)

Multe programe ce folosesc mai multe fire de execuție se folosesc de interfața bibliotecii `pthread` pentru a crea și sincroniza aceste fire, pentru că aceasta este disponibilă pe orice sistem de operare ce respectă interfața POSIX.

Biblioteca `syan_pthread_shim` este o bibliotecă dinamică ce exportă o parte din simbolurile disponibile în biblioteca `pthread`. Toate aceste simboluri sunt funcții care *interceptează* apelurile programului la funcții din `pthread`, capturează evenimentele asociate respectivelor funcții prin apeluri la `syan_capture_event` și apelează mai departe funcțiile cu același simbol din `pthread`. Această bibliotecă nu oferă o interfață în cod sursă, pentru că interfața la nivel de sursă este o submulțime a funcțiilor din interfața `pthread`. Precum orice alt *shim* (bibliotecă ce interceptează în mod transparent apelurile la altă bibliotecă [26]), această bibliotecă trebuie încărcată de un program ce folosește `pthread` *înainte* de încărcarea bibliotecii `pthread` propriu-zisă, pentru ca apelurile la funcții cu simbolurile vizibile din `pthread` să fie interpretate ca apeluri la funcții din biblioteca `syan_pthread_shim`.

Pentru a folosi această bibliotecă, ne vom folosi de bine-cunoscutul *LD_PRELOAD trick*[27] pentru a garanta că biblioteca este încărcată prima. Trucul se bazează pe

```
typedef int(*original_func_t)(pthread_rwlock_t*);
int pthread_rwlock_tryrdlock(pthread_rwlock_t* rwl) {
  original_func_t f = (original_func_t)dlsym(
        RTLD_NEXT, "pthread_rwlock_tryrdlock");
  syan_capture_event(SA_EV_RWLOCK_ON_TRY_RD_LOCK, rwl);
  int r = f(rwl);
  if (r == 0)
    syan_capture_event(SA_EV_RWLOCK_AFTER_RD_LOCK, rwl);
  return r;
}
```

Fragment de cod 17: Funcția `pthread_rwlock_tryrdlock` din `pthread_shim`

variabila de mediu `LD_PRELOAD`: dacă această variabilă conține o listă de nume de biblioteci dinamice, atunci aceste biblioteci sunt încărcate înaintea tuturor celorlalte cerute de program:

```
% LD_PRELOAD="libsyan_pthread_shim.so" ./program
```

Folosind acest truc, această bibliotecă poate fi folosită fără a necesita măcar recompilarea programului client. Când programul face un apel la o funcție din `pthread`, spre exemplu la funcția `pthread_rwlock_tryrdlock`, în loc să se apeleze funcția cu acel nume din biblioteca `pthread`, se va apela funcția din biblioteca noastră (Fragmentul de cod 17). Aceasta face apelul mai departe la funcția originală, găsită printr-un apel manual al funcției `dlsym` cu parametrul `RTLD_NEXT` (parte din interfața DL[10]), însă capturează de asemenea evenimentele interesante prin apeluri la funcția `syan_capture_event`.

4.3 Analizori

După execuția programului client integrat cu biblioteca pentru capturare de evenimente și una din bibliotecile de mai sus, evenimentele capturate trebuie acum analizate folosind programul `SyncAnalysis`. Însă după cum am mai spus, acest program nu face pe cont propriu nicio analiză, ci se bazează pe *analizorii* încărcați să emită diagnosticele în sine, programul doar ajutând acești analizori cu munca de rutină. Astfel, ca parte dintr-o unealtă de analiză trebuie compusă și o suită de analizori care să facă propriu-zis analiza.

Vom descrie în continuare un exemplu de astfel de suită: 4 analizori ce au ca scop observarea folosirii *excesive* de primitive de sincronizare.

4.3.1 useless-lock

Observație: dacă un *mutex*, *mutex recursiv* sau un *read-write lock* este *locked* de un singur fir de execuție, primitiva respectivă nu este necesară programului.

Pe baza observației de mai sus se poate concepe un simplu algoritm care analizează apariția acestei situații. Pentru fiecare eveniment care marchează crearea unui *lock* (oricare din primitivele de mai sus), se crează o intrare pentru respectivul lock într-un tabel. Apoi, pentru orice eveniment capturat cu câmpul `object_id` egal cu obiectul inițial se înregistrează câmpul `thread_id` al evenimentului în intrarea pentru obiectul respectiv în tabel. Dacă un obiect ajunge să aibă 2 fire de execuție diferite stocate în intrarea sa din tabel, acesta poate fi șters din tabel, deoarece nu mai prezintă interes. Dacă atunci când se capturează evenimentul de distrugere a *lock*-ului (sau când se termină lista de evenimente) intrarea acestuia încă se află în tabel, este emis un diagnostic cu nivelul de severitate de *avertisment*, pentru că respectivul *lock* nu este neapărat necesar.

Implementarea acestui analizor conține un total de 89 de linii de cod C++.

4.3.2 lock-shadow

Implementarea acestui analizor conține un total de 103 linii de cod C++.

4.3.3 redundant-recursive-mutex

Implementarea acestui analizor conține un total de 73 de linii de cod C++.

4.3.4 redundant-rwlock

Implementarea acestui analizor conține un total de 76 de linii de cod C++.

4.4 Testele analizorilor (folosind cxxsync)

Write text

5 Concluzii

Complexitatea aplicațiilor software crește constant, și pentru ca dezvoltatorii de aplicații să poată păstra același nivel de calitate a produselor în ciuda cerințelor din ce în ce mai dificile, au nevoie de unelte din ce în ce mai puternice care să îi ajute. Dar dezvoltarea acestor unelte, mai ales a celor *specifice* unei aplicații, necesită de asemenea efort și timp, de multe ori mai mult decât sunt dispuse companiile și dezvolatorii să investească.

Proiectul prezentat în această lucrare și-a îndeplinit cu succes scopul de a simplifica procedeul de dezvoltare a acestor unelte. Evenimente de interes sunt capturate eficient și automat în timpul execuției aplicației, necesitând schimbări minime (sau uneori deloc) în codul sursă al aplicațiilor analizate. Analizori *post-mortem* ai acestor evenimente, cu aplicații practice reale, pot fi compuși și folosiți cu ușurință, atât pentru analiza primitivelor de sincronizare cât și pentru orice alt domeniu, uneori necesitând mai puțin de 100 de linii de cod.

Industria software are nevoie de unelte de dezvoltare, iar proiectul prezentat este un pas spre facilitarea compunerii acestora.

Anexa I. Lista figurilor

1 Arhitectura proiectului SyncAnalysis 6
2 Stiva de execuție în arhitectura X86 20
3 Măsurători de performanță a bibliotecii 24

Anexa II. Lista fragmentelor de cod

1 Exemplu de folosire ThreadSanitizer 6
2 Instrucțiuni pentru instalarea proiectului 12
3 Interfața bibliotecii pentru capturare de evenimente 14
4 Exemplul folosit în proiectarea interfeței 15
5 Capturare incorectă a creării unui fir de execuție 16
6 Capturare corectă a creării unui fir de execuție 17
7 Implementarea funcției `syan_capture_event` 17
8 Structura unui eveniment capturat 18
9 Capturarea stivei de execuție folosind `libunwind` 21
10 Interfața cozii de evenimente folosite în bibliotecă 22
11 Interfața rudimentară a unui analizor 31
12 Interfața pe bază de clasă a unui analizor 32
13 Tipurile de evenimente pentru primitivele de sincronizare 33
14 Tehnica *type-erasure* în clasa `Thread` din `cxxsync` 35
15 Metoda `RWLock::try_rd_lock` din `cxxsync` 36
16 Clasa `MutexGuard` din `cxxsync` 37
17 Funcția `pthread_rwlock_tryrdlock` din `pthread_shim` 38

Bibliografie

[1] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[2] D. Hutchins, A. Ballman, and D. Sutherland. C/C++ thread safety analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 41–46, 2014.

[3] The Clang Team. Thread Sanitizer - Clang 11 documentation. `https://clang.llvm.org/docs/ThreadSanitizer.html`, 2009. [Online; accessed 6-June-2020].

[4] The Clang Team. Clang-Tidy - Clang 11 documentation. `https://clang.llvm.org/extra/clang-tidy/index.html`, 2009. [Online; accessed 6-June-2020].

[5] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[6] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, fifth edition, December 2017.

[7] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[8] Ken Martin and Bill Hoffman. *Mastering CMake: a cross-platform build system : version 3.1.* Kitware, New York, NY, 2015.

[9] IEEE and The Open Group. *IEEE Std 1003.1-2017 (aka POSIX)*, volume 2, chapter 3, pages 1560–1744. IEEE, New York, NY USA, seventh edition, 2017.

[10] IEEE and The Open Group. *IEEE Std 1003.1-2017 (aka POSIX)*, volume 2, chapter 3, pages 739–747. IEEE, New York, NY USA, seventh edition, 2017.

[11] David Mosberger Tang. The libunwind project. `https://www.nongnu.org/libunwind/`, 2011. [Online; accessed 7-June-2020].

[12] Roland H Pesch and Jeffrey M Osier. The GNU binary utilities. *Free Software Foundation*, 1993.

[13] Apple Inc. atos(1) [osx man page]. https://www.manpagez.com/man/1/atos/, 2010. [Online; accessed 7-June-2020].

[14] Darius Marian. The command line arguments library of the mcga project. https://github.com/darius98/mcga-cli, April 2019. [Online; accessed 7-June-2020].

[15] Ravishankar Somasundaram. *Git: Version control for everyone*. Packt Publishing Ltd, 2013.

[16] GitHub Inc. Github. https://github.com, 2007. [Online; accessed 8-June-2020].

[17] Google Inc. A microbenchmark support library. https://github.com/google/benchmark, 2015. [Online; accessed 10-June-2020].

[18] Free Software Foundation Inc. Gnu official website. https://www.gnu.org/software/binutils/, 2007. [Online; accessed 8-June-2020].

[19] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, 1995.

[20] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Proceedings Seventh International Conference on Parallel and Distributed Systems (Cat. No.PR00568)*, pages 470–475, 2000.

[21] TIS Committee et al. Tool Interface Standard (TIS) executable and linking format (ELF) specification version 1.2, 1995.

[22] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*, volume 1, chapter 20.11.3, pages 634–643. International Organization for Standardization, Geneva, Switzerland, fifth edition, 2017.

[23] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, pages 17–17, 2005.

[24] Thomas Becker. Type erasure in C++: The glue between object oriented and generic programming. *Multiparadigm Programming*, 2007.

[25] Scott Meyers. In *Effective C++: 55 specific ways to improve your programs and designs*, pages 63–65. Pearson Education, 2005.

[26] Wikipedia. Shim (computing). `http://en.wikipedia.org/w/index.php?title=Shim\%20(computing)&oldid=946620767`, 2020. [Online; accessed 13-June-2020].

[27] Kevin Pulo. Fun with LD_PRELOAD. In *linux. conf. au*, volume 153, 2009.