

UNIVERSITATEA BUCUREȘTI  
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
SPECIALIZAREA INFORMATICĂ



Lucrare de licență  
Analiza statică a primitivelor de sincronizare

Coordonator științific  
Paul Irofti

Absolvent  
Darius Marian

București, iulie 2020

# Cuprins

<b>1</b>	<b>Introducere</b>	<b>3</b>
1.1	SyncAnalysis . . . . .	4
1.2	Comparație cu ThreadSanitizer . . . . .	6
1.3	Comparație cu Clang-Tidy . . . . .	7
<b>2</b>	<b>Tehnologii folosite</b>	<b>8</b>
2.1	Limbajul C11 și biblioteca standard C . . . . .	8
2.2	Limbajul C++17 și biblioteca standard C++ . . . . .	8
2.3	Compilerul Clang . . . . .	8
2.4	Unelte 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
<b>3</b>	<b>Descrierea proiectului</b>	<b>12</b>
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	Obținerea stivei de execuție . . . . .	17
3.2.4	Coadă de evenimente . . . . .	17
3.2.5	Serializarea fișierului DUMP . . . . .	17
3.3	Biblioteci de integrare . . . . .	17
3.3.1	cxxsync (C++, începând cu C++98) . . . . .	17
3.3.2	pthread (C/C++) . . . . .	17
3.3.3	libc++ (pentru C++, începând cu C++11) . . . . .	17
3.4	Programul independent SyncAnalysis . . . . .	17
3.4.1	Parcurgerea și parsarea fișierului DUMP . . . . .	17
3.4.2	Stocarea și observarea obiectelor active . . . . .	17

3.4.3	Crearea de rapoarte . . . . .	17
3.4.4	Afişarea simbolurilor din stiva de execuţie . . . . .	17
3.4.5	Înregistrarea analizorilor . . . . .	17
3.5	Analizori pre-instalaţi . . . . .	17
3.5.1	mutex-lock-order . . . . .	17
3.5.2	lock-shadow . . . . .	17
3.5.3	redundant-recursive-mutex . . . . .	17
3.5.4	redundant-rwlock . . . . .	17
3.5.5	useless-lock . . . . .	17
<b>4</b>	<b>Exemple de utilizare</b>	<b>18</b>
4.1	Testele analizorilor (folosind cxxsync) . . . . .	18
4.2	Folosind LD_PRELOAD şi <pthread.h> . . . . .	18
4.3	Folosind libc++ . . . . .	18
<b>5</b>	<b>Concluzii</b>	<b>19</b>

# 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 capacități 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 *concurrent* 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 concurrent. 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 concurrent 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 apărute 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, 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**.

Biblioteca este una de uz general, neavând în mod direct o interfață specifică analizării primitivelor de sincronizare. De aceea proiectul conține și o suită de biblioteci pentru a

facilita folosirea bibliotecii de capturare a evenimentelor în scopul de analiză a primitivelor de sincronizare. Acestea oferă interfețe identice cu alte biblioteci populare cum ar fi `<pthread.h>` sau modulele `<thread>` și `<mutex>` din biblioteca standard C++, pentru a permite programelor ce folosesc aceste biblioteci să folosească *SyncAnalysis* fără schimbări în codul sursă. Detalii despre aceste biblioteci se găsesc în secțiunea **3.3**.

După execuția programului client obținem o colecție de evenimente, capturate și serializate eficient într-un fișier cu ajutorul bibliotecilor descrise mai sus. În continuare, trebuie făcută o analiză a acestor evenimente, iar proiectul oferă o soluție pentru asta în forma unui program independent. 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 programul 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 alte 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 **3.4**, iar despre analizorii implementați ca exemplu pentru analiza primitivelor de sincronizare în secțiunea **3.5**.

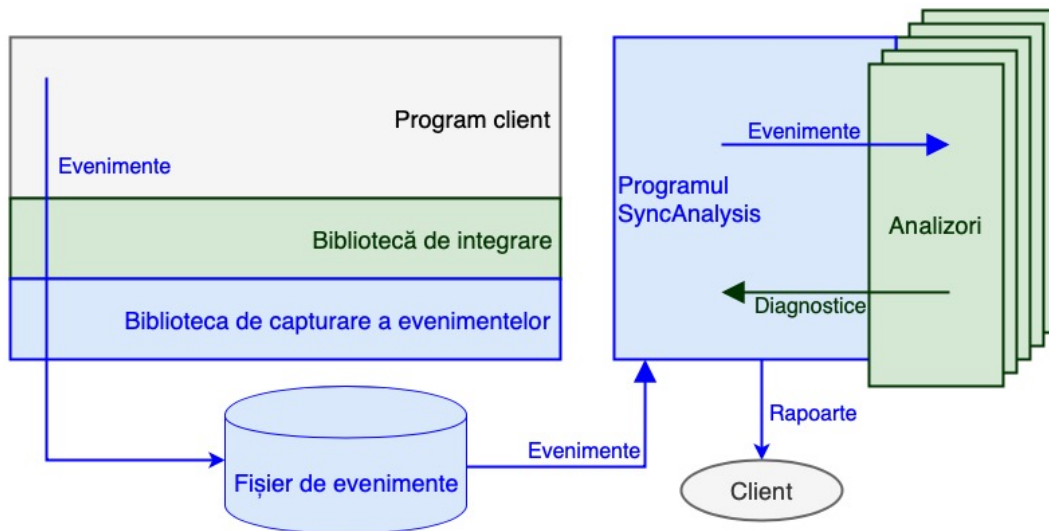


Figura 1: Arhitectura proiectului SyncAnalysis

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 clientului.

## 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 conține verificări împotriva acceselor concurente nesincronizate la memorie:

```
% 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)
```

După cum se vede din exemplul de mai sus (parafrizat din site-ul oficial de documentație al proiectului [3]), 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. 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 unelei, 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. În timp ce proiectul *Clang-Tidy* permite scrierea cu efort mic a unor analizori statici pentru codul sursă de C/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 orice limbaj și rulând pe 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, două dintre bibliotecile de integrare, toate exemplele de analizoare și toate testele automate au codul sursă scris în acest limbaj.

În afară 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 stil modern, rămânând totuși 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

Limbafele C și C++, folosite în acest proiect, sunt limbafe 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 standard 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 două dintre bibliotecile de integrare, una din acestea având ca scop chiar simularea interfeței publice oferite în header-ul `<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 utilizator.

## 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 trebuie apoi să transforme respectivele adrese în simboluri citibile de către client, pentru a fi atașate rapoartelor.

Pentru majoritatea sistemelor de operare ce respectă standardul POSIX, programul `addr2line` oferit de GNU ca parte din proiectul `binutils` [12] rezolvă această problemă. 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 principal pe un calculator cu sistemul de operare Mac OS (cu kernel-ul Darwin) unde programul `addr2line` nu funcționează prea bine, 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 de autorul proiectului SyncAnalysis, descris aici, dar este un proiect independent de articolul curent.

## 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>.

## 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 sau alte programe echivalente 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 [17].
- `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 Clang 10 [7])
- `git` [15] – minim versiunea 2.0

Proiectul este disponibil pe internet, stocat pe server-ul GitHub. Astfel, acesta poate fi descărcat prin următoarea comandă shell:

```
% git clone https://github.com/darius98/sync-analysis.git
```

Deoarece *repository*-ul de git include biblioteca `mcga-cli` [14] ca submodule, trebuie executată o comandă separată pentru a inițializa submodulele de git:

```
% cd sync-analysis/  
% git submodule update --init
```

Odată ce *repository*-ul principal și *submodul*-ul `mcga-cli` au fost descărcate cu succes, proiectul poate fi compilat prin următoarea secvență de comenzi, executate în directorul de bază al proiectului:

```
% cmake .  
% make all
```

După compilarea proiectului, acesta poate fi instalat în sistem folosind următoarea comandă, executată tot în directorul de bază al proiectului:

```
% make install
```

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` (pentru Mac OS extensia este `.dylib`) și interfața C a acesteia: `include/sync_analysis.h`
- biblioteca pentru integrare `lib/libcxxsync.so` (pentru Mac OS extensia este `.dylib`) și interfața C++ a acesteia: directorul `include/cxxsync/`
- biblioteca pentru integrare `lib/libsyman_thread_shim.so` (pentru Mac OS extensia este `.dylib`)
- biblioteca pentru integrare `lib/libsyman_stdcxx_shim.so` (pentru Mac OS extensia este `.dylib`) și interfața C++ a acesteia: directorul `include/stdcxx_shim/`
- programul independent SyncAnalysis `bin/sync_analysis`
- interfața C++ pentru dezvoltarea analizorilor `include/syman_analyzer_api/`
- analizorii preinstalați directorul `syman-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 o constantă numerică. În limbajul C, interfața poate fi vizualizată astfel:

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

#define SYAN_THREAD_CREATE_EVENT 17

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

Singura funcționalitate a acestei biblioteci este de a captura evenimente, așa că interfața publică importantă este doar funcția `syan_capture_event`, care îndeplinește exact acest scop. Exemplul de utilizare după care a fost proiectată funcția `syan_capture_event` este următorul:

```
#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);
}
```

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: aceeași 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 aceeași memorie; un exemplu comun este memoria stivei de execuție). Pentru a evita confuzii generate de această problemă, se recomandă ca pentru un obiect ce este ținta unor evenimente capturate să se captureze și un eveniment care marchează dealocarea obiectului.

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 unui fir de execuție într-un context. Astfel, biblioteca în sine când se încarcă raportează 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 `SYAN_THREAD_CREATE_EVENT` (aleator ales 17) care să reprezinte evenimentele de acest tip. Orice bibliotecă de integrare trebuie să dea același număr pentru argumentul `event_type` când apelează funcția `syon_capture_event` pentru a raporta crearea unui fir 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`, ținta evenimentului este cunoscută abia după ce a pornit firul de execuție:

```
pthread_t T;  
pthread_create(&T, nullptr, func, arg);  
syon_capture_event(SYAN_THREAD_CREATE_EVENT, (void*)T);
```

Dacă evenimentul care marchează crearea firului de execuție *T* este raportat *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 `syon_initialize_event` și `syon_finalize_event` în interfața bibliotecii. Exemplul de mai sus se repară astfel:



```

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

```

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.

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);
}

```

### **3.2.2 Structura unui eveniment**

### **3.2.3 Obținerea stivei de execuție**

### **3.2.4 Coadă de evenimente**

### **3.2.5 Serializarea fișierului DUMP**

## **3.3 Biblioteci de integrare**

### **3.3.1 cxxsync (C++, începând cu C++98)**

### **3.3.2 pthread (C/C++)**

### **3.3.3 libc++ (pentru C++, începând cu C++11)**

## **3.4 Programul independent SyncAnalysis**

### **3.4.1 Parcurgerea și parsarea fișierului DUMP**

### **3.4.2 Stocarea și observarea obiectelor active**

### **3.4.3 Crearea de rapoarte**

### **3.4.4 Afișarea simbolurilor din stiva de execuție**

### **3.4.5 Înregistrarea analizorilor**

## **3.5 Analizori pre-instalați**

### **3.5.1 mutex-lock-order**

### **3.5.2 lock-shadow**

### **3.5.3 redundant-recursive-mutex**

### **3.5.4 redundant-rwlock**

### **3.5.5 useless-lock**

## 4 Exemple de utilizare

4.1 Testele analizorilor (folosind `cxxsync`)

4.2 Folosind `LD_PRELOAD` și `<pthread.h>`

4.3 Folosind `libc++`

## 5 Concluzii

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