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

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, 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 **3.4.5**.

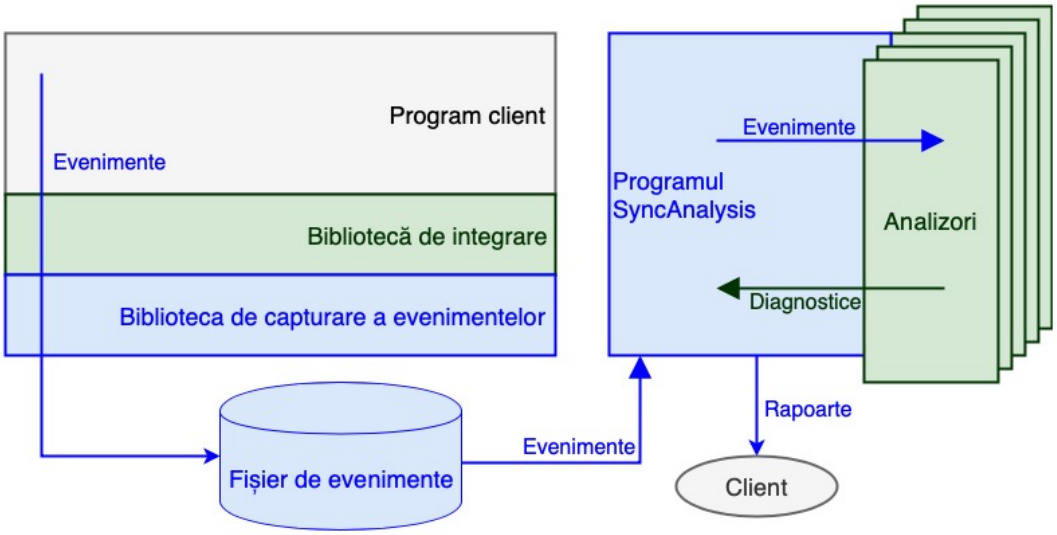

Figura 1: Arhitectura proiectului SyncAnalysis

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)
```

Fragment de cod 1: Exemplu de folosire ThreadSanitizer

După cum se vede din exemplul de mai sus (parafrazat din site-ul oficial de documenție 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 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. 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.

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 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

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 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 tot 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 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, trebuie executată comanda 3 separat pentru a inițializa submodulele de git și a descărca acest submodul.

Odată ce *repository*-ul principal și *submodul*-ul 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 (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/libsyan_pthread_shim.so (pentru Mac OS extensia este .dylib)
- biblioteca pentru integrare lib/libsyan_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/syan_analyzer_api/
- analizorii preinstalați directorul 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 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);
```

Fragment de cod 3: Interfața bibliotecii pentru capturare de evenimente

Funcționalitatea 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 principal 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);
```

```
}
```

Fragment de cod 4: Exemplul 1 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: 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 syan_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);
syan_capture_event(SYAN_THREAD_CREATE_EVENT, (void*)T);
```

Fragment de cod 5: Exemplul 2 folosit în proiectarea interfeței

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 syan_initialize_event și syan_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);
```

Fragment de cod 6: Exemplul 3 folosit în proiectarea interfeței

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);
}
```

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 este următoarea:

```
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

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 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 un detaliu de implementare folosit pentru a împiedica biblioteca din a scrie un eveniment pe disc înainte de apelarea funcției syan_finalize_event. Aceasta nu va scrie evenimentul pe disc până când valoarea câmpului signature nu devine egală cu valoarea prestabilită 666013. În cadrul funcției syan_initialize_event valoarea câmpului este marcată 0, dar acesta este deja pus în coada de evenimente ce trebuie scrise pe disc. La apelul funcției syan_finalize_event, valoarea este schimbată în 666013, permițând bibliotecii să continue serializarea de evenimente.

Câmpurile event_type și object_id vor conține valorile cerute de clientul bibliotecii în apelul funcției syan_capture_event, sau respectiv în apelurile la syan_initialize_event și la 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 **3.4.4** 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

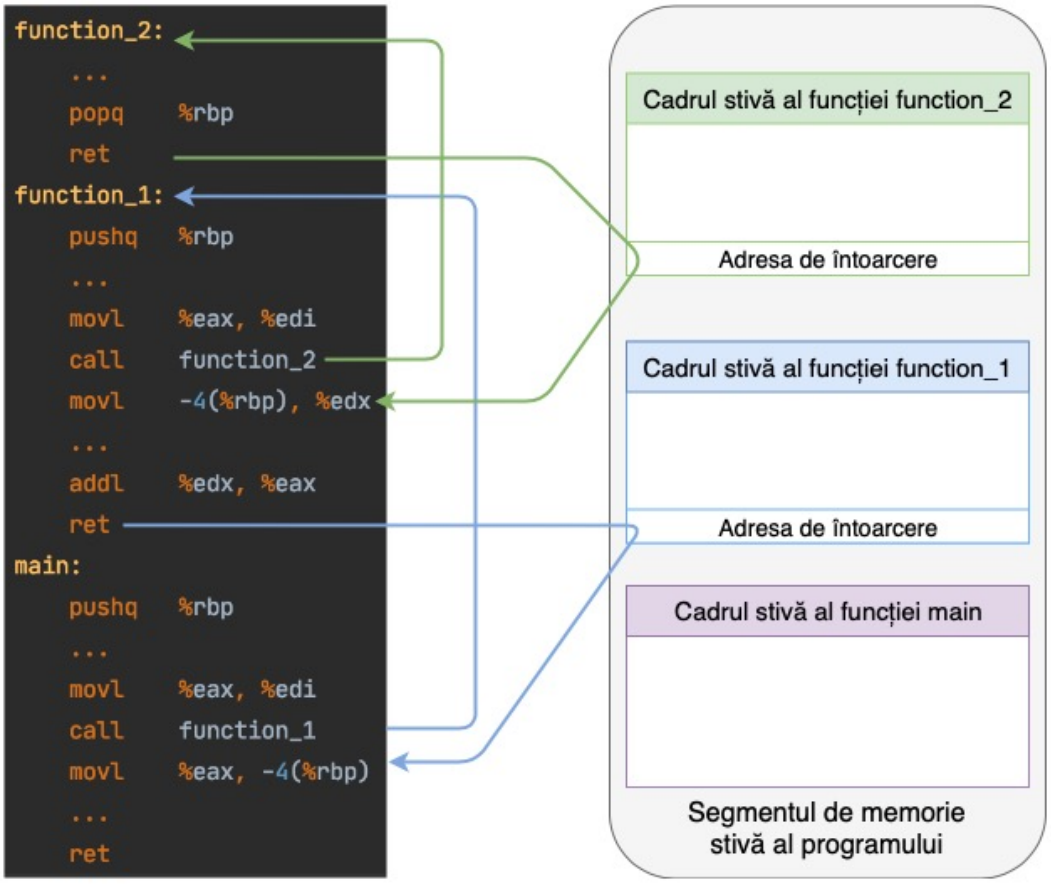

Figura 2: Stiva de execuție în arhitectura X86

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 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] prin următoarea funcție C:

```
#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

Astfel, fiecare eveniment conține tot șirul de adrese de întoarcere din sitva de execuție a firului ce a capturat evenimentului, urmând ca această informație să fie interpretată în analiza post mortem pentru a obține numele funcțiilor din care fac parte aceste adrese (după cum este descris în secțiunea **3.4.4**).

3.2.4 Coada de evenimente și serializarea

Am menționat deja mai devreme că un obiectiv clar 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.

Odată luată această decizie, performanța bibliotecii se împarte în două componente: durata capturării unui eveniment și eficiența firului de execuție ce serializează datele pe disc. Pentru a obține rezultate satisfăcătoare pe ambele planuri, am proiectat o *coadă de evenimente* (intern numită SyanBuffer) specifică pentru această bibliotecă. Interfața acestei cozi în limbajul C poate fi văzută în Fragmentul de cod 10.

```
#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

Încă din interfață se poate observa prima proprietate importantă a acestei cozi: nu conține niciun mecanism explicit de sincronizare, ci doar două variabile cu tipul de date *atomic*.

Punerea rapidă a 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 (pagina b−>pages[b−>pages_back]). Asta omite necesitatea de a apela funcții precum malloc pentru a obține memorie pentru eveniment sau memcpy pentru a muta evenimentul în coadă abia după ce acesta este complet format. Așadar în funcția syan_initialize_event se obține o zonă de memorie pentru stocarea respectivului eveniment printr-un apel la funcția syan_buffer_acquire_event_slot, care implementează un algoritm *lock-free* bazat pe operațiile primitive 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* [18] și *A practical nonblocking queue algorithm using compare-and-swap* [19].

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 doar 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 are de scris pe disc, firul apelează funcția syan_buffer_get_front_page și verifică valorile câmpurilor storage_front și storage_back de pe acea pagină. 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*. Biblioteca asumă că între apelurile funcțiilor syan_initialize_event și syan_finalize_event trece relativ puțin timp. 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. O variantă simplificată a implementării în limbajul C poate fi văzută mai jos:

```
SyanBufferPagePtr p = syan_global_buffer_get_front_page();
int_fast32_t front = p->storage_front;
int_fast32_t back = atomic_load(&p->storage_back);
for (int i = front; i < back; i++)
    while (atomic_load(&p->storage[i].signature)
                != SYAN_EVENT_SIGNATURE)
        ;
int num_bytes = (back - front) * sizeof(SyanEvent);
fwrite(&p->storage[front], 1, num_bytes, DUMP_file);
if (back == SYAN_BUFFER_PAGE_SIZE)
    syan_global_buffer_release_front_page();
else
    buf_page->storage_front = back;
return (back - front > 0);
```

Fragment de cod 11: Scrierea evenimentelor pe disc

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 **3.4.4**.

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

După cum am menționat în secțiunea **3.2.4**, performanța bibliotecii se împarte în două metrici diferite: timpul necesar capturării unui eveniment și volumul de evenimente ce poate fi serializat într-o perioadă de timp. Ambele măsurători depind de frecvența evenimentelor și de numărul de fire de execuție ce capturează evenimente. În Figura 3 se pot vedea două tabele cu măsurătorile făcute cu ajutorul bibliotecii Google Benchmark [20] pentru ambele metrici interesante.

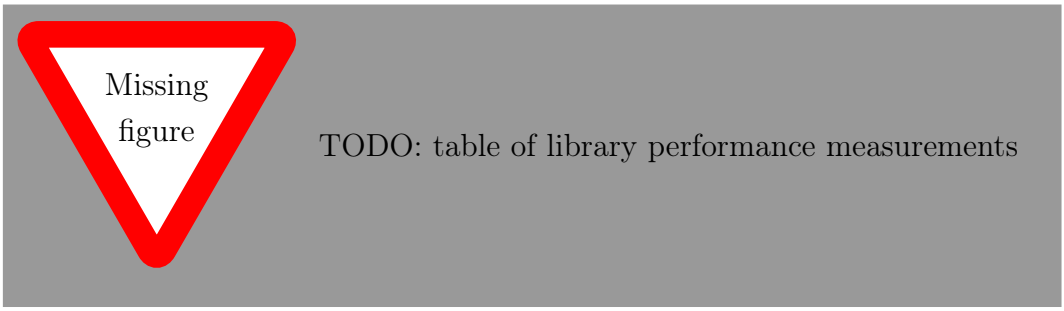

Figura 3: Măsurători de performanță a bibliotecii

3.3 Biblioteci de integrare

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

3.3.2 pthread shim (C/C++)

3.3.3 libc++ shim (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

Anexa I. Lista figurilor

Anexa II. Lista fragmentelor de cod

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].

[18] 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.

[19] 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.

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