DIVINE: analýza programů v C++

Vladimír Štill

Masarykova univerzita

Brno, Česká republika

16. září 2017

DIVINE: analýza programů v C++

DIVINE je nástroj na analýzu programů v C a C++

- historicky býval zaměřený převážně na verifikace paralelních/distribuovaných systémů
- nyní především důraz na podporu C a C++ a na hledání těžko-nalezitelných problémů

DIVINE: analýza programů v C++

DIVINE je nástroj na analýzu programů v C a C++

- historicky býval zaměřený převážně na verifikace paralelních/distribuovaných systémů
- nyní především důraz na podporu C a C++ a na hledání těžko-nalezitelných problémů

- memory safety, assertion safety, problémy související s paralelismem (deadlocky, data race, memory modely)
- simulace chyb

DIVINE: analýza programů v C++

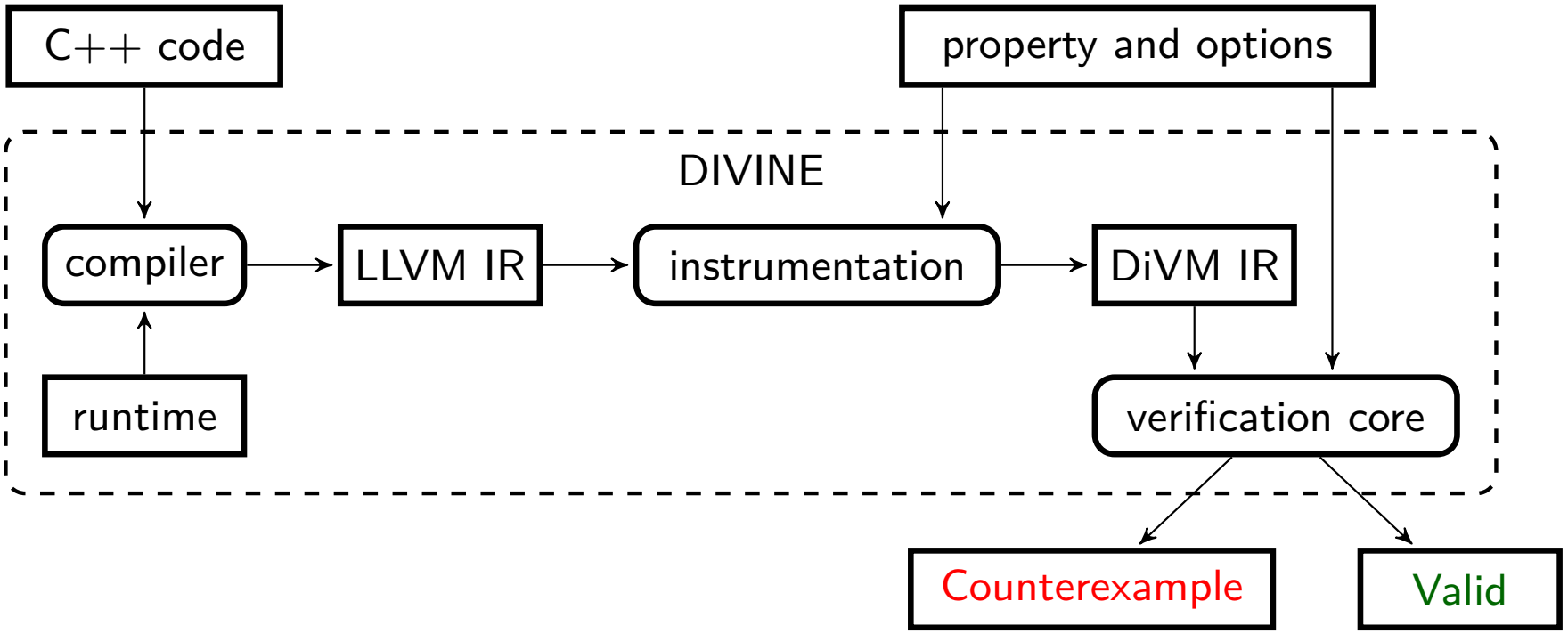
DIVINE je nástroj na analýzu programů v C a C++

- historicky býval zaměřený převážně na verifikace paralelních/distribuovaných systémů
- nyní především důraz na podporu C a C++ a na hledání těžko-nalezitelných problémů

- memory safety, assertion safety, problémy související s paralelismem (deadlocky, data race, memory modely)
- simulace chyb

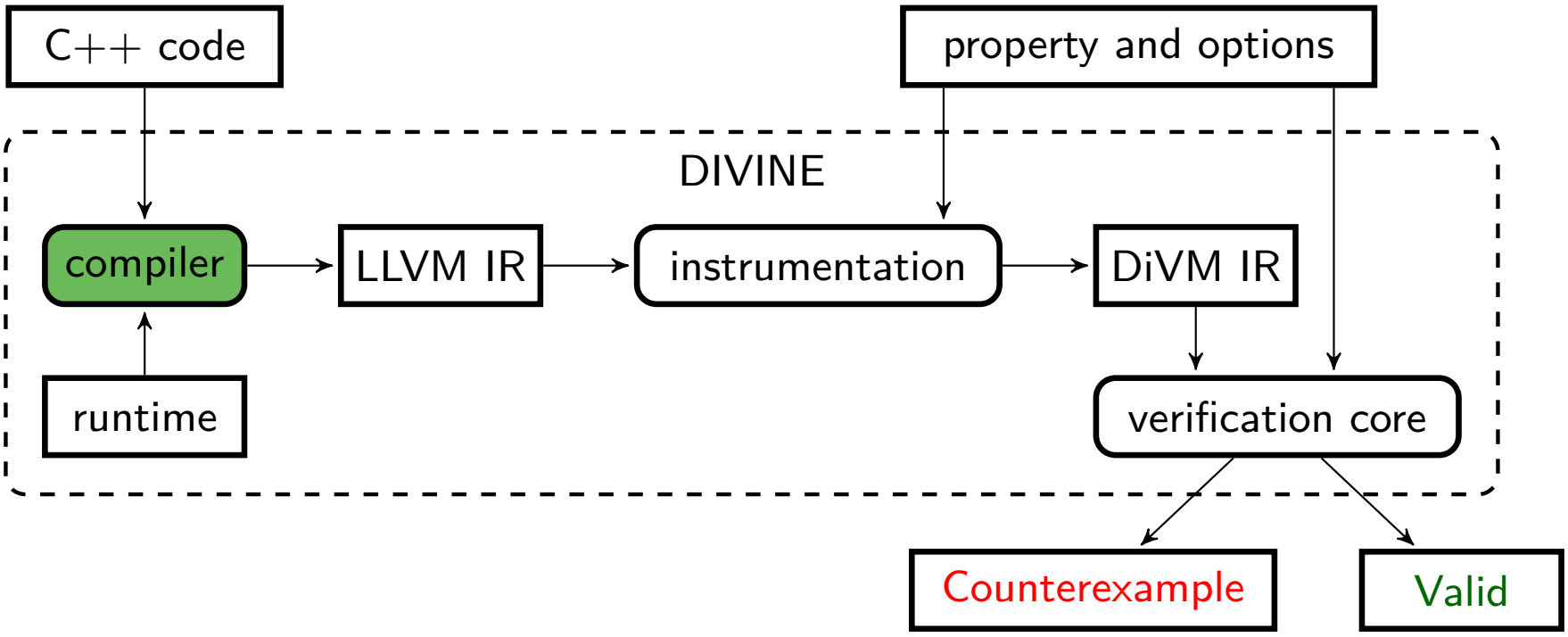
- rozsáhlá jazyková podpora, podpora části POSIX

DIVINE: analýza programů v C++



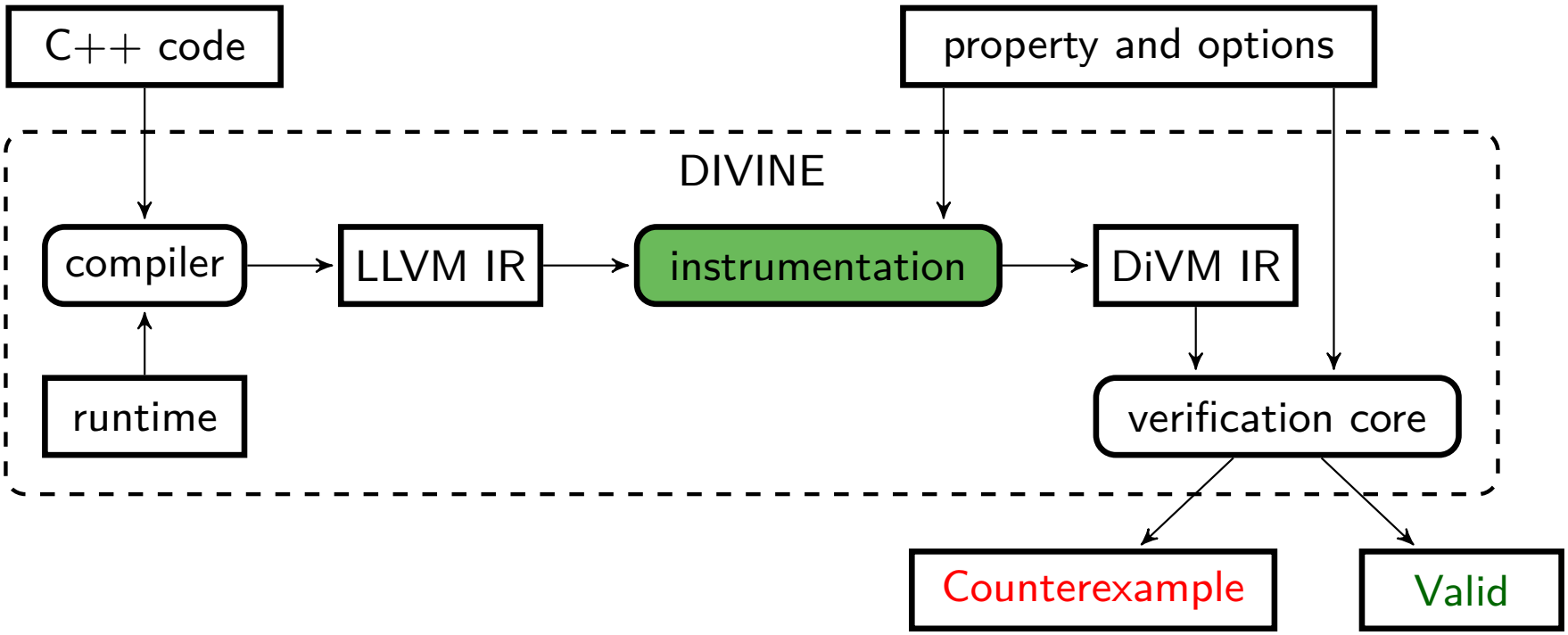
- DIVINE je postavený nad LLVM: využívá kompilátor clang a analyzuje LLVM bitkód

DIVINE: analýza programů v C++



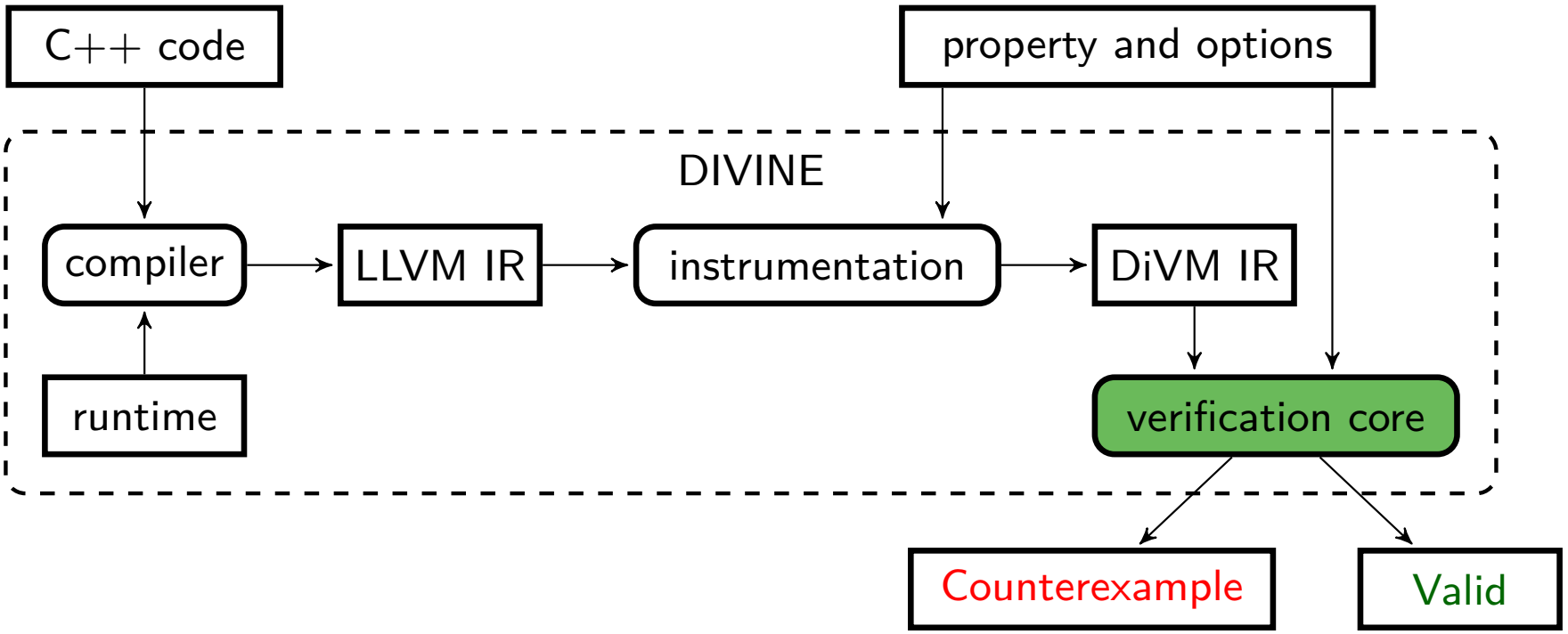
- kompilátor je clang integrovaný jako knihovna do DIVINE
- překládá nejen uživatelův program ale i knihovny, které program může volat

DIVINE: analýza programů v C++



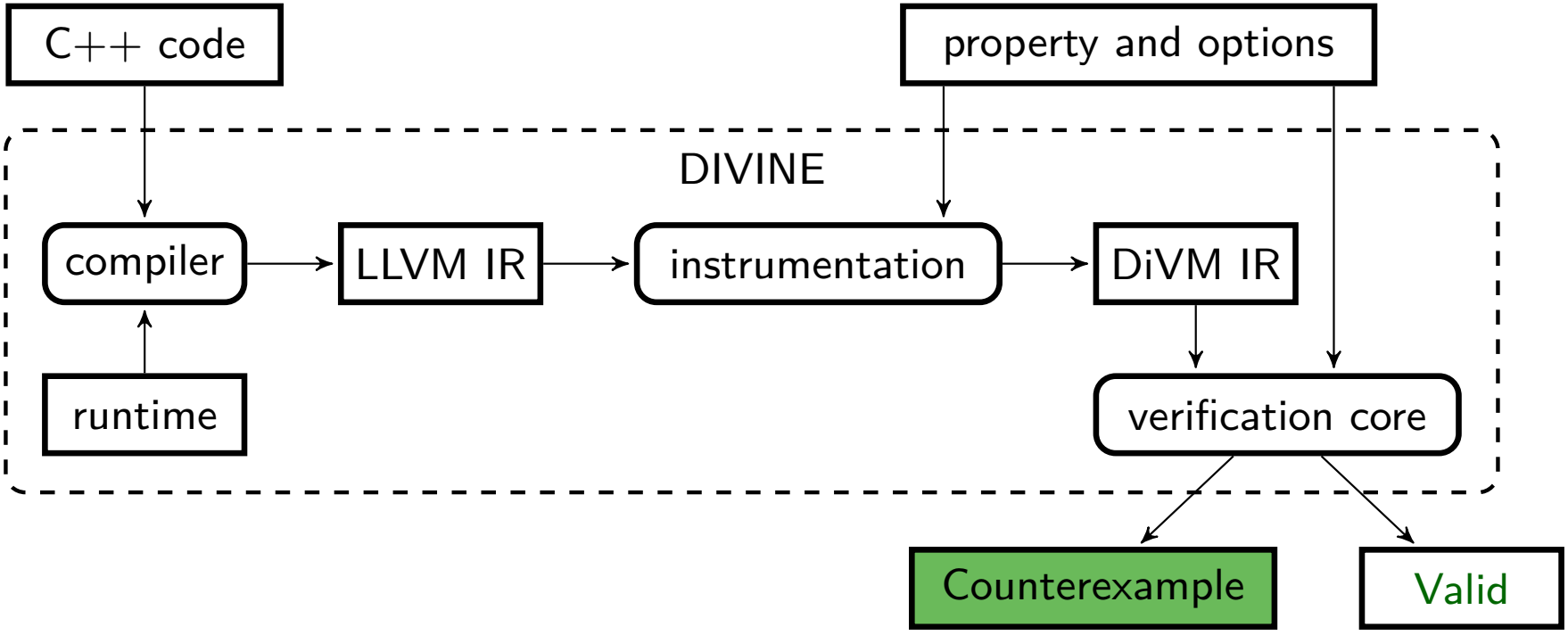
- část instrumentace je nutná pro chod DIVINE: sledování přístupu k paměti, detekce cyklů v control flow.
- část rozšiřuje funkcionalitu: symbolická verifikace, memory modely

DIVINE: analýza programů v C++



- interpret LLVM s grafovou pamětí, který zajišťuje samotné spouštění programu
- algoritmus pro prohledávání stavového prostoru

DIVINE: analýza programů v C++



- DIVINE dále obsahuje simulátor/debugger pro procházení běhů programu

Jádro DIVINE: požadavky

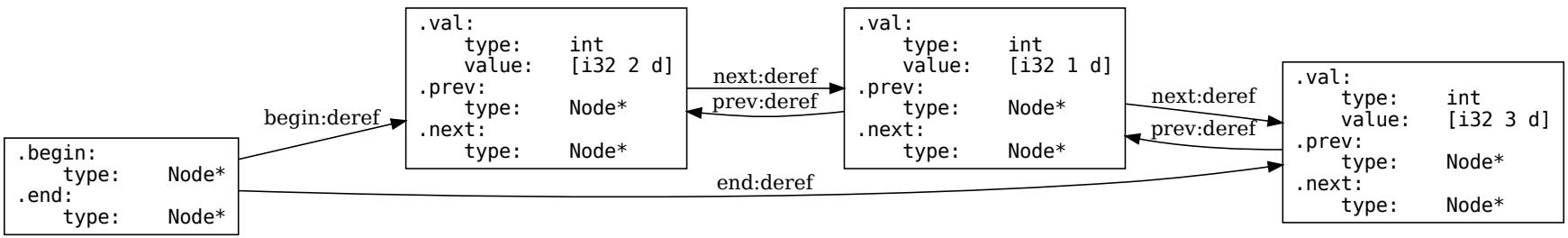
- DIVINE používá vlastní interpret LLVM
  - DIVINE Virtual Machine = DiVM

Jádro DIVINE: požadavky

- DIVINE používá vlastní interpret LLVM
  - DIVINE Virtual Machine = DiVM
- interpret musí být velmi striktní
  - kontrola přístupu k paměti, ...
- také musí být pokud možno rychlý
- musí umožnit uložit snapshot stavu programu
- musí umožnit spustit vše potřebné pro běh programu

Jádro DIVINE: DiVM

- LLVM IR je nejprve přeložen do interní reprezentace: DiVM IR
- přidává k LLVM IR: alokace paměti, nízkoúrovňová manipulace zásobníku, hlášení chyb, . . .
- DiVM pracuje s pamětí reprezentovanou jako graf



Obrázek 1: Paměť jako graf

Knihovny

- programy nejsou uzavřené, používají externí hlavičnové soubory, externí knihovny

Knihovny

- programy nejsou uzavřené, používají externí hlavičnové soubory, externí knihovny
- používat systémové hlavičkové soubory je nebezpečné

Knihovny

- programy nejsou uzavřené, používají externí hlavičnové soubory, externí knihovny
- používat systémové hlavičkové soubory je nebezpečné

```
$ ./symbiotic mem_heap_bug01.c
cc: /usr/include/stdlib.h:33:10: fatal error:
    'stddef.h' file not found
cc: 1 warning and 1 error generated.
```

Knihovny

- programy nejsou uzavřené, používají externí hlavičnové soubory, externí knihovny
- používat systémové hlavičkové soubory je nebezpečné

```
$ ./symbiotic mem_heap_bug01.c
cc: /usr/include/stdlib.h:33:10: fatal error:
    'stddef.h' file not found
cc: 1 warning and 1 error generated.
```

- proto si s sebou DIVINE nese knihovny
  - standardní knihovny C a C++
  - POSIX threads

Knihovny, kompilace, DiOS

- DIVINE překládá C/C++ soubory pomocí integrovaného kompilátoru
  - clang jako knihovna
- knihovny jsou přeloženy při překladu DIVINE
- kompilátor je přilinkuje k programu

Knihovny, kompilace, DiOS

- DIVINE překládá C/C++ soubory pomocí integrovaného kompilátoru
  - clang jako knihovna
- knihovny jsou přeloženy při překladu DIVINE
- kompilátor je přilinkuje k programu

- dále je třeba poskytnout programu plánování vláken, procesů, případně simulaci práce se soubory, . . .

Knihovny, kompilace, DiOS

- DIVINE překládá C/C++ soubory pomocí integrovaného kompilátoru
  - clang jako knihovna
- knihovny jsou přeloženy při překladu DIVINE
- kompilátor je přilinkuje k programu

- dále je třeba poskytnout programu plánování vláken, procesů, případně simulaci práce se soubory, ...
- to vše v DIVINE zajišťuje DiOS (DIVINE Operating System)

Používání DIVINE: verifikace

- `divine verify file.cpp [args...]`
- `divine check file.cpp [args...]`
  - méně striktní režim (neselhává alokace paměti, . . .)

Používání DIVINE: verifikace

- `divine verify file.cpp [args...]`
- `divine check file.cpp [args...]`
  - méně striktní režim (neselhává alokace paměti, ...)
- ovlivnění kompilace:
  - `divine verify -std=c++14 file.cpp`
  - `-std=`, `-l`, přímo, další přepínače přes -C
  - `divine verify -C,-O3 file.cpp`

Používání DIVINE: verifikace

- `divine verify file.cpp [args...]`
- `divine check file.cpp [args...]`
  - méně striktní režim (neselhává alokace paměti, ...)
- ovlivnění kompilace:
  - `divine verify -std=c++14 file.cpp`
  - `-std=`, `-l`, přímo, další přepínače přes -C
  - `divine verify -C,-O3 file.cpp`
- oddělená kompilace a verifikace
  - `divine cc -almost-any-clang-flags file1.cpp file2.cpp`
  - `divine verify file1.bc`

Používání DIVINE: simulátor

- `divine sim file.cpp`
  - pro krokování programu od začátku
- `divine sim --load-report file.report.????`
  - načtení reportu z `verify`/`check`
  - jméno reportu je na konci výpisu z verifikace
  - automaticky skočí před chybu
  - na chybu se obvykle lze dostat přes příkaz `stepa`

Používání DIVINE: základní příkazy simulátoru

- `up/down` – skákání po zásobníku
- `show VAR` – výpis hodnoty
  - `show .localvar`
  - `show $globals.globalvar`
- `backtrace`
  - `backtrace $state` pro všechna vlákna
- `step [--over] [--out]` – krokování po „příkazu“
- `stepi [--over] [--out]` – krokování po instrukci
- `stepa [--over] [--out]` – krokování po atomickém kroku
- `thread pid:tid` – výběr vlákna

Používání DIVINE: další příkazy simulátoru

- `rewind #stav` – návrat do předchozího stavu
- `start` – skok na začátek `main`
- `break file:line`
- `break function`
- `draw VAR`
- přidání okna se zdrojovým kódem
  - `setup --xterm src`
  - `setup --sticky "source --output-to src --clear-screen" --pygmentize`

DEMO:

- `ssh login@arke`
- `source /var/obj/divine-fakos`
- `git clone https://github.com/vlstill/presentations.git`
- `cd presentations/fakos_2017/demo`

Instrumentace – paměťové modely

Paměťový model

- při běhu paralelních programů vstupují do hry specifika chování daného hardware
  - přeuspořádávání instrukcí a cache paměti způsobují, že paměťové efekty vlákna nemusí být vidět v očekávaném pořadí
  - Intel `x86` umí „jen“ pozdržovat zápisy, ale zachovává jejich pořadí
  - ARM umí přeuspořádat téměř libovolné nezávislé a některé závislé operace
- kompilátor rovněž může přeuspořádávat operace
- programátor musí správně využívat synchronizaci aby zabránil problémům s relaxovanou pamětí
  - `std::atomic` v C++

Paměťový model a C++

```
int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

Dosažitelné $x = 0 \wedge y = 2$?

Paměťový model a C++

```
int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

Dosažitelné $x = 0 \wedge y = 2$? ano

- nedefinované chování – může dělat cokoli
- `thread2` může přečíst `y` jen jednou

Paměťový model a C++

```
volatile int x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

Dosažitelné $x = 0 \wedge y = 2$? ano

- opět nedefinované chování
- na většině kompilátorů zajistí, že se `y` přečte v každé iteraci cyklu
- to však na některých platformách nestačí k zajištění korektnosti (ARM/POWER)

Paměťový model a C++

```
std::atomic< int > x = 0, y = 0;

void thread1() { x = 1; y = 2; }
void thread2() {
    while ( y == 0 ) { }
    cout << "y = " << y << endl;
    cout << "x = " << x << endl;
}
```

Dosažitelné $x = 0 \wedge y = 2$? ne

- garance atomického přístupu k `x`, `y`
- `y` načteno v každé iteraci cyklu
- procesor nesmí přeuspořádat atomické zápisy

Jak na paměťový model

- verifikační nástroje často neberou ohled na paměťový model
- v DIVINE používáme instrumentaci k zohlednění paměťového modelu

Jak na paměťový model

- verifikační nástroje často neberou ohled na paměťový model
- v DIVINE používáme instrumentaci k zohlednění paměťového modelu

- čtení a zápisy nahrazeny za funkce které simulují zpožďování operací
  - zpožďování zápisů pomocí store bufferů
  - předbíhání zápisů je komplikovanější

Zpožďování zápisů

```
int x = 0, y = 0;

void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

Zpožďování zápisů

```
int x = 0, y = 0;

void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

main memory

0x04	0x08
x = 0	y = 0

thread 0

```
store y 1;
load x;
```

thread 1

```
store x 1;
load y;
```

store buffer for thread 0

store buffer for thread 1

Zpožďování zápisů

```
int x = 0, y = 0;

void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

main memory

0x04	0x08
x = 0	y = 0

thread 0

```
store y 1;
load x;
```

thread 1

```
store x 1;
load y;
```

store buffer for thread 0

0x08	1	32

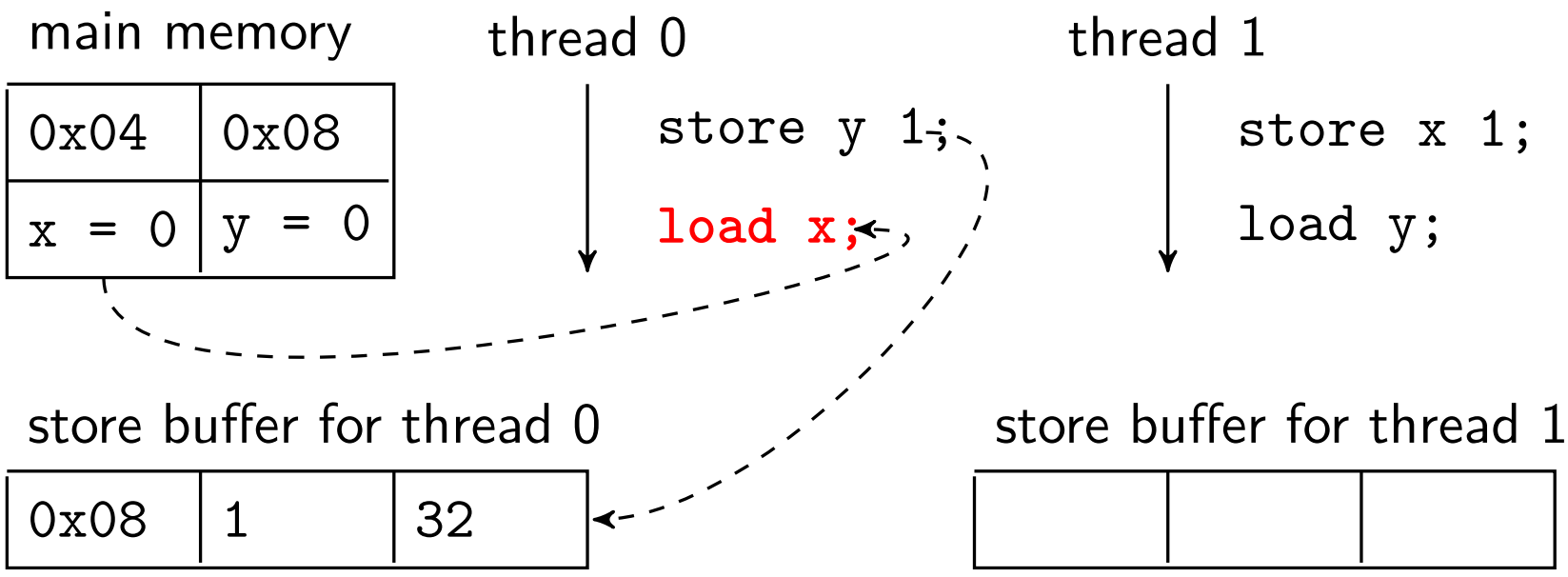
store buffer for thread 1

Zpožďování zápisů

```
int x = 0, y = 0;

void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```

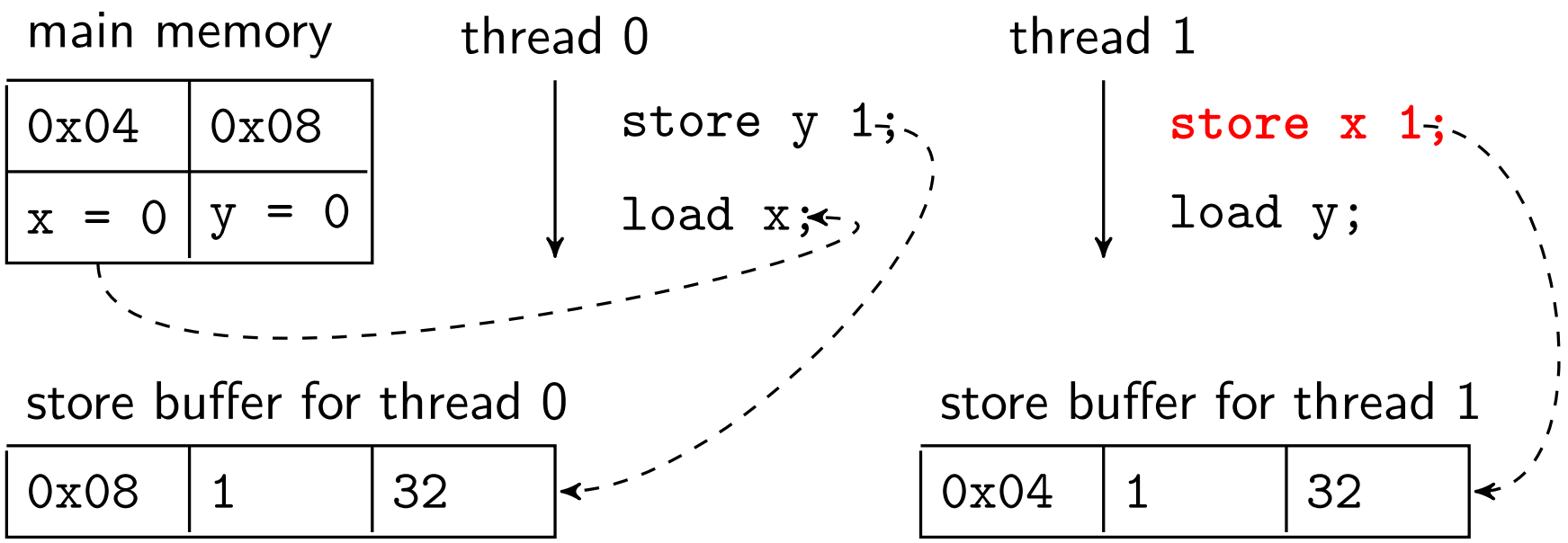


Zpožďování zápisů

```
int x = 0, y = 0;

void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```



Zpožďování zápisů

```
int x = 0, y = 0;

void thread0() {
    y = 1;
    cout << "x = " << x;
}
```

```
void thread1() {
    x = 1;
    cout << "y = " << y;
}
```