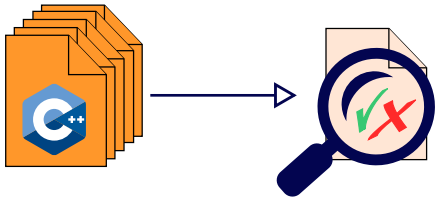
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2020
http://excel.fit.vutbr.cz

Detekce paralelních chyb ve víceprocesových programech

Monika Mužikovská*



Abstrakt

Dynamická analýza se s úspěchem využívá pro detekci chyb ve vícevláknových programech. Algoritmy, které byly za tímto účelem navrženy, jsou ale často využitelné i pro víceprocesové programy. Žádný ze známých nástrojů pro dynamickou analýzu ale monitorování procesů nepodporuje. Cílem této práce bylo rozšířit nástroj ANaConDA o analýzu a monitorování víceprocesových programů. Výsledkem je implementace rozšíření, které za vývojáře analyzátorů řeší problémy spojené s oddělenými adresovými prostory a synchronizací pomocí semaforů. Rozšíření bylo využito pro úpravu analyzátoru AtomRace pro detekci časově závislých chyb nad daty ve víceprocesových programech a použito na experimenty se studentskými projekty z předmětu Operační systémy. Výsledky experimentů ukázaly, že se nástroj ANaConDA může stát vítaným pomocníkem při implementaci nejen víceprocesových projektů.

Klíčová slova: ANaConDA – Dynamická analýza – Paralelní chyby – Víceprocesové programy

Přiložené materiály: N/A

*xmuzik05@stud.fit.vutbr.cz, *Fakulta informačních technologií, Vysoké učení technické v Brně*

1. Úvod

Vícevláknové a víceprocesové programy jsou náchylné na specifické chyby s nedeterministickým projevem, a proto se k jejich testování využívají jiné přístupy než pro typické sekvenční programy. Mezi úspěšné metody pro odhalování paralelních chyb patří dynamická analýza, která monitoruje běh programu, sleduje důležité události a ze získaných informací vyvozuje závěry o případných problémech. Jelikož je monitorování běhu programu implementačně velmi náročné, vznikly nástroje, které tuto funkcionalitu poskytují (např. ConTest [1], RoadRunner [2], můžeme zde zařadit i Valgrind [3] a samozřejmě nástroj ANaConDA [4]). Detekce chyb potom probíhá v analyzátorech, které často implementují speciální algoritmus zaměřený na konkrétní typy chyb. Aplikační oblast analyzátoru je tedy omezena tím, jaké typy programů jsou nástroje schopny monitorovat. V současné době se bohužel u všech existujících nástrojů jedná pouze o vícevláknové programy a paralelismus na úrovni procesů není podporován. Přitom navržené algoritmy často mezi vlákny a procesy nerozlišují a je možné je použít pro detekci chyb v obou typech programů. Pro plné využití potenciálu těchto algoritmů se tedy nabízí rozšíření již existujícího nástroje o podporu monitorování procesů.

Přirozenou volbou byl nástroj ANaConDA, který je aktivně vyvíjen zde na fakultě. ANaConDA slouží k dynamické analýze C/C++ programů a poskytuje sadu analyzátorů pro detekci časově závislých chyb nad daty (z angl. *data race*), uváznutí či porušení atomicity. Kromě monitorování běhu také nabízí sadu podpůrných funkcí a struktur pro snadnější imple-

mentaci nových analyzátorů. Podrobnosti jsou popsány v sekci 2.

1.1 Řešené problémy a přínos práce

Tento článek dále shrnuje, v čem se vlákna a procesy liší, a tím pádem se musí lišit i implementace analyzátorů. Hlavní rozdíl spočívá v tom, že vlákna mají společný adresový prostor, a tedy sdílí veškerá data. Všechny dynamické analyzátory implementované v nástroji ANaConDA se od tohoto faktu odrážejí. Předpokládají, že všechny identifikátory a adresy jsou sdílené mezi všemi vlákny, takže např. využívají logickou adresu jako jednoznačný identifikátor. Procesy ale mají oddělené adresové prostory a tento předpoklad je pro ně tedy chybný. Při monitorování programu tak analyzátory musí brát v potaz to, že stejná logická adresa v různých procesech může příslušet jiné fyzické adrese (ale nemusí) a ne všechna data jsou mezi procesy sdílená, a tedy ne každý přístup do paměti může způsobit chybu. Aby bylo možné rozhodovat, zda je přístup do paměti potenciálně nebezpečný či nikoli, je třeba monitorovat meziprocesovou komunikaci prostřednictvím sdílené paměti, což dosud nebylo potřeba. Také samotné analyzátory jsou paralelní programy a využívají stejnou formu paralelismu jako monitorovaný program. Současné implementace předpokládají vícevláknovost a přizpůsobují tomu mechanismy sdílení dat a synchronizaci. U víceprocesových programů je samozřejmě třeba volit jiné přístupy, a proto se i implementace analyzátoru musí přizpůsobit. Stávající analyzátory tak nelze beze změny na monitorování procesů použít a musí být vytvořeny analyzátory nové. Sice implementující stejný algoritmus pro detekci chyb, ale zohledňující problémy, ke kterým u procesů dochází. Aby úprava implementace analyzátorů za účelem monitorování procesů byla co nejmenší, byla většina problémů vyřešena v rámci této práce a implementována do jádra nástroje ANaConDA. Konkrétně tato práce přinesla:

1. API pro komunikaci procesů analyzátoru.
2. Řešení problému s identifikátory a logickými adresami.
3. Monitorování synchronizace pomocí obecných semaforů (dosud bylo poskytováno pouze pro binární zámky).
4. Rozšíření *happens-before* relace pro extrapolaci běhu programu na tyto obecné semafory.

Detaily jsou popsány v sekci 3.

Podpora pro monitorování procesů byla využita k implementaci analyzátoru AtomRace [5] pro detekci časově závislé chyby nad daty ve víceprocesových programech. S analyzátorem byly provedeny experimenty nad studentskými projekty z předmětu Operační systémy, které ukázaly, že i v projektech s plným bodovým hodnocením lze najít paralelní chybu.

2. Monitorování vláken

Tato sekce je věnována stručnému popisu nástroje ANaConDA a jeho funkci při dynamické analýze vícevláknových programů. Dále je uveden algoritmus AtomRace pro detekci časově závislé chyby nad daty, který je v nástroji ANaConDA implementován jako jeden z poskytovaných analyzátorů. Na závěr sekce je popsána tzv. *happens-before* relace, kterou využívají komplikovanější analyzátory pro extrapolaci běhu programu.

2.1 Nástroj ANaConDA

ANaConDA[1] (Adaptable Native-code Concurrency-focused Dynamic Analysis) je open-source nástroj pro dynamickou analýzu vícevláknových C/C++ programů na binární úrovni [4][6]. Skládá se ze tří hlavních částí: nástroje Pin, jádra nástroje a analyzátorů. Pin[2] [7] provádí dynamickou instrumentaci programu a monitoruje jeho běh. Při důležitých událostech informuje jádro prostředí ANaConDA, které nízkoúrovňové informace předzpracuje a v uživatelsky přívětivější podobě předává analyzátorům. Kromě toho také jádro poskytuje API, které obsahuje úložiště pro lokální data vláken (*Thread Local Storage*), načítání konfiguračních souborů, uzamykatelné objekty a další pomocné funkce.

Komunikace mezi nástrojem Pin a jádrem a také mezi jádrem a analyzátory probíhá pomocí tzv. zpětných volání (*callback*). Analyzátory pomocí speciálních funkcí registrují obslužné rutiny pro události jako získání či uvolnění zámku, vytvoření nebo ukončení vlákna, volání funkce nebo přístup do paměti, které se spustí vždy, když v monitorovaném programu daná událost nastane. Toto zpětné volání také získá doplňující informace jako identifikátor vlákna či zámku, logickou adresu paměti nebo argument a návratovou hodnotu funkce.

2.2 Algoritmus AtomRace

AtomRace [5] je jeden z nejjednodušších algoritmů pro detekci časově závislé chyby nad daty, který nástroj ANaConDA poskytuje. Je založen na monitorování přístupů do paměti a rozpoznání situace, kdy k této chybě dojde přesně dle její definice – tedy když dvě

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/

[2] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

různá vlákna přistoupí do stejného paměťového místa a alespoň jeden z přístupů je zápis.

ANaConDA pro každou událost poskytuje dvojici zpětných volání – jedno se vykoná před danou událostí v programu (tzv. *before callback*) a druhé po dané události v programu (tzv. *after callback*). Algoritmus AtomRace zapsaný pomocí těchto zpětných volání je na ukázce 1. Vlákna využívají sdílenou mapu, kde se pro jednotlivé logické adresy ukládají informace o přístupech. Ve zpětném volání `beforeAccess()` vlákno ověřuje, zda na danou adresu nepřistoupilo i jiné vlákno[3]. Pokud ano a alespoň jeden z přístupů je zápis, dojde k detekci chyby. Ve zpětném volání `afterAccess()` se ze sdílené mapy smaže informace o přístupu.

```
SharedMap Access;

beforeAccess(thread t, address a, mode m) {
  if (Access[a] == null) {
   Access[a] = (t, m);
  }
  else if (m == write ||
  Access[a].m == write)
   RACE DETECTED
}

afterAccess(thread t, address a, mode m) {
  if (Access[a].t == t)
    Access[a] = null;
}
```

Zdrojový kód 1. Pseudokód algoritmu AtomRace.

Algoritmus AtomRace má oproti komplikovanějším algoritmům, které sledují synchronizaci, jednu velkou výhodu – neprodukuje falešná chybová hlášení. Proto byl také upraven pro analýzu procesů a použit pro experimenty se studentskými projekty popsanými v sekci 4.

2.3 *hb*-relace a vektorové hodiny

Dynamické analyzátory získané informace typicky využívají pro tzv. extrapolaci a snaží se detekovat i takovou chybu, ke které v aktuálním plánu přepínání kontextu nedošlo, ale v jiném podobném by mohlo. Za tímto účelem je často nutné sledovat synchronizaci. Vhodnými nástroji jsou tzv. *happens-before* relace (dále značena jako *hb*-relace) a vektorové hodiny (*vector clock*). V nástroji ANaConDA ji využívá např. analyzátor FastTrack [8] pro detekci časově závislé chyby nad daty nebo analyzátory pro detekci porušení kontraktu [9]. Jelikož se jedná o důležitou techniku, je vhodné ji přizpůsobit i procesům, a proto je zde stručně popsána.

hb-relace je částečné uspořádání a využívá se pro určování pořadí událostí v různých vláknech s ohledem na synchronizaci. Formálně je *hb*-relace $\prec_{hb}$ nejmenší tranzitivně uzavřená relace nad množinou událostí $\{e_1, \cdots, e_n\}$ stopy $\tau = e_1 \cdots e_n$. Dvě události e_j a e_k jsou v *hb* relaci $e_j \prec_{hb} e_k$ vždy, když platí $j < k$ a zároveň alespoň jedna z následujících podmínek:

- Obě události byly vykonány stejným vláknem.
- Událost e_j uvolňuje zámek, který událost e_k obdrží.
- Událost e_j je vytvoření nového vlákna u ve vlákně t a událost e_k je provedena ve vláknu u.
- Událost e_j je provedena ve vláknu u a událost e_k je čekání vlákna t na ukončení vlákna u.

Pokud mezi dvěma událostmi *hb*-relace není, jsou považovány za souběžné a mohou být zdrojem paralelních chyb.

V nástroji ANaConDA se *hb*-relace implementuje s využitím vektorových hodin navržených pro algoritmus FastTrack, které udržují informaci o logickém čase každého vlákna v programu. Formálně jsou vektorové hodiny zobrazení z množiny všech vláken T v programu do množiny přirozených čísel, $VC : T \to \mathbb{N}$. Každé vlákno $t \in T$ si udržuje vlastní hodiny VC_t. Hodnota $VC_t(t)$ představuje aktuální logický čas vlákna t. Hodnota $VC_t(u)$ pro každé $u \in T, u \neq t$ představuje poslední čas, kdy se vlákno t synchronizovalo s vláknem u. Pro každou událost e_u provedenou vláknem u v čase $tm \leq VC_t(u)$ tedy platí, že je v *hb*-relaci s aktuální událostí e_t ve vláknu t: $e_u \prec_{hb} e_t$.

Při synchronizaci dvou vláken dojde k aktualizaci jejich vektorových hodin s využitím operátoru spojení, který je definován následovně:

$$VC_1 \sqcup VC_2 = \lambda t.max(VC_1(t), VC_2(t))$$

Pro účely monitorování vícevláknových programů bylo popsáno pravidlo pro aktualizaci vektorových hodin při synchronizaci pomocí zámků. Každý zámek L má vlastní vektorové hodiny VC_L. Při události, kdy vlákno t uvolní zámek L, dojde k aktualizaci vektorových hodin zámku: $VC'_L = VC_t$. Tím se tedy uloží informace o vektorových hodinách vlákna t v době provedení události. Při události, kdy vlákno u obdrží zámek L, dojde k aktualizaci vektorových hodin vlákna u: $VC'_u = VC_u \sqcup VC_L$. Tím se tedy vlákno u jednak synchronizuje s vláknem t, ale tranzitivně také s ostatními vlákny.

[3] Ověření `Access[a].t != t` není nutné, protože stejné vlákno nemůže dvakrát přistoupit na stejnou adresu, aniž by se vykonalo i zpětné volání `afterAccess()`, ve kterém se záznam o přístupu odstraní. Pokud je tedy `Access[a] != null`, jedná se o přístup jiného vlákna.

3. Návrh rozšíření

V této sekci budou popsány základní rozdíly mezi vlákny a procesy a návrhy řešení jednotlivých problémů, které také byly implementovány do nástroje ANaConDA. Nejpodstatnější otázkou je samotný přístup k analýze. Analyzátory jsou totiž také paralelní programy a mají stejný počet vláken (případně procesů) jako monitorovaný program. Všechna vlákna vykonávají stejný algoritmus a sdílí mezi sebou informace, na základě kterých obvykle jedno vlákno detekuje chybu. Pro procesy se nabízelo několik možností, z nichž se jako nejzajímavější jevil stejný přístup jako pro vlákna nebo centralizovaná analýza, kterou bude provádět jeden proces a ostatní se využijí pro sběr dat. První možnost má velkou výhodu, protože umožňuje použít stávající analyzátory také pro procesy (s lehkými modifikacemi kvůli problémům popsaným dále). Proto také byla implementována.

3.1 Zpětná volání a identifikátory

Většinu zpětných volání poskytovaných analyzátorům lze využít jak pro vlákna, tak i pro procesy. Problém je pouze s událostmi specifickými pro vlákna, jako je jejich vytvoření, start a ukončení. Tato zpětná volání se často využívají pro inicializaci struktur, ale také se jedná o důležité synchronizační události pro *hb*-relaci. Je tedy potřeba doplnit odpovídající volání také pro procesy.

Všechna zpětná volání získávají identifikátor vlákna, které provedlo danou událost. Stejnou informaci je potřeba dodat i pro procesy. Jelikož ale nástroj Pin poskytuje funkci `PIN_GetPid()`, není nezbytně nutné dosavadní volání přepisovat a je možné si v nich identifikátor vyžádat dodatečně.

3.2 Lokální a globální data

Jelikož je analyzátor paralelní program a na detekci chyby se podílí všechna vlákna (resp. procesy), je nutné využívat určité principy sdílení dat. Konkrétně existují dva důvody, proč je toto sdílení nutné:

1. Sdílení získaných informací mezi jednotlivými vlákny nebo procesy (např. vektorové hodiny jednotlivých vláken nebo sdílená mapa využívaná v algoritmu AtomRace).
2. Sdílení lokálních dat mezi jednotlivými zpětnými voláními jednoho vlákna nebo procesu.

Jelikož vlákna sdílí adresový prostor, oba problémy je možné řešit pomocí globálních proměnných. Pokud proměnná obsahuje data sdílená mezi všemi vlákny, je nutné ji chránit zámkem, aby v samotném analyzátoru nevznikla paralelní chyba. Pro ukládání lokálních dat se využívá *Thread Local Storage* (TLS), ke kterému vlákna přistupují pomocí jednoznačného identifikátoru.

Pro procesy je situace opačná. Jejich adresový prostor je oddělený, takže nepotřebují TLS a lokální data mezi jednotlivými zpětnými voláními mohou být uložena v globálních proměnných. Sdílení dat mezi procesy je ale mnohem komplikovanější a je třeba přistoupit k mechanismům meziprocesové komunikace, jako např. sdílená paměť.

3.2.1 API pro komunikaci procesů

Vlákna nebo procesy často sdílí informace v dynamických STL kontejnerech, jako jsou vektory, fronty nebo mapy. Jejich uložení do sdílené paměti o fixní velikosti s sebou přináší problémy spojené s detekcí nedostatku paměti, jejím zvětšením a případnou realokací. Protože se implementačně jedná o komplikovaný problém, bylo API nástroje ANaConDA obohaceno o sdílené datové struktury, které jej řeší za vývojáře analyzátorů. API analyzátorům poskytuje sdílené číselné datové typy, vektor, mapu, oboustrannou frontu, řetězec, zámek apod. spolu s metodami a operacemi, aby se jejich použití nelišilo od klasických datových typů.

Problém detekce nedostatku paměti a jejího zvětšení se podařilo vyřešit díky knihovně `boost`, která poskytuje speciální alokátor právě pro ukládání dynamických datových struktur do sdílené paměti a objekt pro správu a zvětšování sdílené paměti. Zvětšení ale může způsobit realokaci, po které je nutné, aby si všechny procesy znovu namapovaly danou sdílenou proměnnou do svého adresového prostoru. Z tohoto důvodu je přístup k dynamickým strukturám realizován přes prostředníka, který procesům poskytuje informace nutné pro rozpoznání, zda od posledního přístupu došlo ke zvětšení sdílené paměti s proměnnou.

3.3 Logické adresy

Oddělený adresový prostor také způsobuje problémy ve zpětných voláních pro přístupy do paměti, která jako jednoznačný identifikátor dotčeného paměťového místa využívají logickou adresu. Tento přístup je korektní ve vícevláknových programech a je využit i ve výše popsaném algoritmu AtomRace, ale pro procesy nefunguje. Pokud totiž dva různé procesy přistoupí do stejného místa ve sdílené paměti, logické adresy mohou být naprosto odlišné. Intuitivně se nabízí využít fyzickou adresu, ale ta se za běhu programu může změnit (např. realokace). Byl tedy implementován překlad z logické adresy na jednoznačný identifikátor paměťového místa, který je stejný pro všechny procesy.

Překlad logické adresy je potřebný pouze tehdy,

když monitorovaný program přistupuje do sdílené paměti. Tuto novou adresu je pak možné odvodit z identifikátoru dané sdílené paměti a posuvu v rámci ní. Algoritmus pro překlad je tedy založen na monitorování vytváření a odstraňování sdílené paměti v analyzovaném programu. Každý segment sdílené paměti je v nástroji ANaConDA reprezentován pomocí tzv. tokenu. To je lokální datová struktura, v níž si každý proces uchovává informace o jednoznačném identifikátoru sdílené paměti a rozsahu logických adres, které jí přísluší. Jeho využití pro překlad je popsáno v algoritmu 1. Pro logickou adresu je potřeba najít token, který reprezentuje danou sdílenou paměť. Poté se z identifikátoru sdílené paměti vytvoří hash[4], který je pro všechny procesy stejný. K této nové "adrese" se přičte posun v rámci paměti a výsledkem je jednoznačný identifikátor paměťového místa.

Algoritmus 1 Algoritmus pro překlad adres.

Vstup: logická adresa a, vektor vc tokenů t
Výstup: pokud a identifikuje místo m ve sdílené paměti, pak jednoznačný identifikátor místa m

1: **if** $\exists t \in vc : a \in t.range()$ **then**
2: $h = t.hash()$
3: $offset = a - t.base()$
4: return $h + offset$

Implementace překladu v nástroji ANaConDA také řeší problémy spojené se změnou velikosti sdílené paměti a rozdíly v identifikaci POSIX a System V sdílené paměti (včetně anonymní sdílené paměti). Výsledný algoritmus se stal součástí API poskytované nástrojem ANaConDA a byl s úspěchem využit v implementaci algoritmu AtomRace.

Překlad logických adres se netýká pouze zpětných volání pro přístupy do paměti, ale také identifikátorů zámků a argumentů či návratových hodnot funkcí (např. synchronizačních funkcí pro semafory, které pracují s ukazateli).

3.4 Synchronizace

Posledním důležitým rozdílem mezi vlákny a procesy je způsob synchronizace. Mezi nejčastěji používané mechanismy patří zámky a semafory. Oba způsoby se dají využít v obou typech paralelních programů, ale protože vlákna častěji využívají zámky, je v nástroji ANaConDA aktuálně podpora pouze pro ně. U rozšíření pro procesy se od počátku předpokládalo, že by mohlo být užitečné i pro studenty předmětu Operační systémy, kteří jako jeden z projektů programují víceprocesový program s využitím semaforů. Z tohoto důvodu byla do nástroje ANaConDA přidána podpora pro jejich monitorování.

Podpora pro monitorování semaforů spočívá v dodání zpětných volání pro odpovídající synchronizační funkce spolu s podstatnými informacemi o daném semaforu. Pro účely algoritmů využívajících *hb*-relaci je také potřeba ji definovat pro semafory a určit pravidla pro aktualizaci vektorových hodin.

3.4.1 Zpětná volání pro semafory

Rozšíření poskytuje zpětná volání pro inicializaci semaforu a operace V (*down*) a P (*up*). Pokrývá POSIX[5] i System V[6] semafory. Analyzátorům je poskytován identifikátor semaforu, jeho aktuální hodnota, počet čekajících procesů a číslo, o kterou daná operace změní hodnotu semaforu (protože System V semafory umožňují inkrementovat a dekrementovat hodnotu semaforu o více než 1).

Z argumentů ani návratových hodnot synchronizačních funkcí bohužel není možné zjistit aktuální hodnotu semaforu ani počet čekajících procesů. Aktuální hodnotu je možné získat pomocí funkcí `sem_getvalue()` nebo `semctl()`. Proces analyzátoru a monitorovaný program ale nesdílí adresový prostor, takže by uvedené funkce musely být volány v kontextu aplikace, což má velmi negativní vliv na dobu běhu analýzy. Nástroj ANaConDA si tudíž pro každý semafor, který monitorovaný program používá, udržuje reprezentaci, na které zrcadlí operace prováděné v monitorovaném programu. Zná tak aktuální hodnotu a je schopen odvodit, kdy bude proces zablokován a zařazen do fronty. Tyto informace potom sděluje analyzátorům.

3.4.2 *hb*-relace pro semafory

hb-relace popsaná v sekci 2.3 definuje uspořádání dvou událostí vzhledem k synchronizaci pomocí zámků. Na ty je možno nahlížet jako na speciální semafory splňující dvě podmínky:

1. Jejich maximální hodnota je 1.
2. Uvolnit zámek (čili provést operaci *up* nad binárním semaforem) může jenom to vlákno nebo proces, které jej aktuálně drží (čili bylo poslední, které úspěšně provedlo operaci *down* nad binárním semaforem).

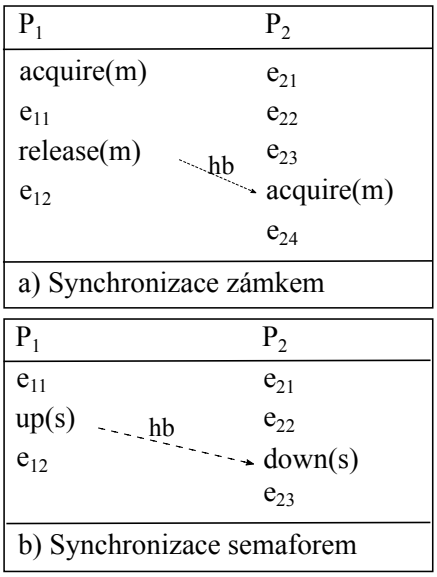
Protože obecné semafory nesplňují ani jednu podmínku, je nutné *hb*-relaci rozšířit o definici uspořádání

[4] Prozatím je tolerována možnost případného konfliktu, který stejný nebo podobný hash pro dvě různá paměťová místa může způsobit, protože očekáváme, že k němu bude docházet velmi zřídka. Pokud by konflikt způsoboval problémy, je samozřejmě možné dodat přísnější porovnávání na rovnost dvou přeložených "adres".

[5] https://linux.die.net/man/7/sem_overview

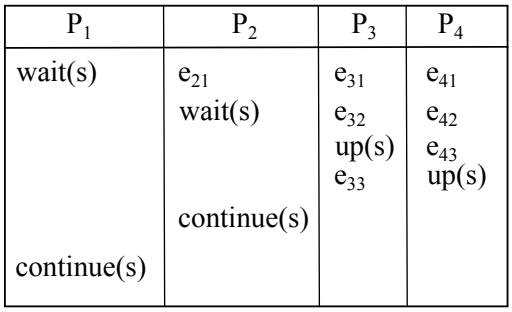
[6] http://man7.org/linux/man-pages/man7/sysvipc.7.html

dvou událostí vzhledem k synchronizaci pomocí semaforů.



Obrázek 1. Formování *hb*-relace mezi dvěma procesy při synchronizaci.

Na obrázku 1 je ukázka *hb*-relace mezi dvěma procesy způsobená synchronizací. Případ a) zachycuje vznik relace při synchronizaci zámky, kdy uvolnění zámku jistě předchází (*happens-before*) jeho obdržení jiným procesem. Teoreticky je na semafory možno nahlížet stejným způsobem a spojovat relací operace *up* a *down* jako na ukázce b). Semafory ale může najednou používat (a inkrementovat) několik procesů a je problém určit, které dvojice operací *up* a *down* nakonec relaci vytvoří.



Obrázek 2. Stopa programu se 4 procesy, které používají semafor s. Procesy P_1 a P_2 jsou zablokovány, zařazeny do fronty čekajících procesů a následně uvolněny procesy P_3 a P_4.

V dalším výkladu bude operace *down* rozdělena na dvě fáze:

1. Fáze *wait* nastane, když proces nebo vlákno provádí operaci *down* nad semaforem, jehož hodnota je 0, což způsobí jeho pozastavení.
2. Fáze *continue* nastane, když proces úspěšně dokončí operaci *down* a může pokračovat v exekuci. K této fázi dojde buď v případě, kdy hodnota semaforu byla kladná a proces vůbec nebyl zablokován, nebo byl uvolněn z čekání díky tomu, že jiný proces daný semafor inkrementoval.

Uvažujme nyní situaci na obrázku 2. Procesy P_1 a P_2 provedou operaci *down* nad semaforem s a oba budou zablokovány. Následně proces P_3 provede operaci *up*, inkrementuje hodnotu semaforu a jako následek by měl být uvolněn jeden z čekajících procesů[7]. Plánovač ale místo toho přidělí procesor procesu P_4, který taktéž provede operaci *up* a uvolní druhý proces. Procesy P_2 a P_1 jsou následně v tomto pořadí uvolněny, úspěšně dokončí operaci a provedou fázi *continue*. Otázkou je, které události by měly být spárovány *hb*-relací.

Článek [10] pro události $e_j \prec_{hb} e_k$ přidává následující pravidlo (zde zjednodušeno):

- e_j je operace *up* provedená procesem p_1, která odblokovala proces p_2 pozastavený operací *down*, a e_k je fáze *continue* v procesu p_2.

Pro situaci na obrázku z něho vyplývají následující relace:

$$P_3.up() \prec_{hb} P_2.continue()$$

$$P_4.up() \prec_{hb} P_1.continue()$$

Po důkladném zvážení (popis dalších možných přístupů je nad rámec tohoto článku) byl pro rozšíření v nástroji ANaConDA zvolen podobný přístup, který výše uvedenou definici rozšiřuje o další pravidlo:

- e_k je neblokující provedení operace *down* (čili došlo pouze k fázi *continue*) a e_j je operace *up*, která byla nutná pro to, aby událost e_k mohla být neblokující.

Důsledek tohoto pravidla je znázorněn na obrázku 3. Jedná se o velmi podobnou stopu jako v předchozím případě, ale operace *up* byly provedeny dříve, než operace *down*, takže procesy P_1 a P_2 nebyly zablokovány. Z logiky synchronizace by ale *hb*-relace měla vypadat stejně, jako pro obrázek 2.

3.4.3 Implementace *hb*-relace pro semafory

Při implementaci algoritmu pro formování *hb*-relace je potřeba pro každou operaci *down* zjistit buď:

1. operaci *up*, která uvolnila proces z čekání, nebo
2. operaci *up*, která procesu umožnila pokračovat bez zablokování.

[7] POSIX ani System V implementace nezaručují, že fronta čekajících procesů je FIFO, a tudíž dopředu nelze určit, který z čekajících procesů bude uvolněn.

P_1	P_2	P_3	P_4
e_{11}	e_{21}	e_{31}	e_{41}
e_{12}	e_{22}	e_{32}	e_{42}
e_{13}	e_{23}	up(s)	e_{43}
e_{14}	e_{24}	e_{33}	up(s)
e_{15}	continue(s)		
continue(s)			

Obrázek 3. Stopa programu, kdy jsou operace *up* nad semaforem *s* provedeny dříve než operace *down*, takže ty proběhnou bez blokování odpovídajících procesů.

Součástí rozšíření se stala implementace, která je schopna oba body splnit, ale její popis je nad rámec tohoto článku. Je založena na ukládání informací o operacích *up* do fronty spolu s informací o čekajících procesech a následném hledání vhodného záznamu, který mohl daný proces uvolnit. Dále je uveden algoritmus znázorňující, jak se synchronizace pomocí semaforů dotkne implementace *hb*-relace pomocí vektorových hodin za předpokladu, že výše uvedené informace jsou k dispozici.

V sekci 2.3 byly popsány vektorové hodiny, operátor spojení a jeho použití pro aktualizaci vektorových hodin vláken, která se synchronizovala pomocí zámků. Dvojice vláken si předává informace přes vektorové hodiny zámku a formování relace probíhá v okamžiku obdržení zámku. Algoritmus pro semafory je velmi podobný, ale protože se pomocí jednoho semaforu může synchronizovat více dvojic vláken/procesů, je potřeba mít vektorové hodiny pro každou operaci *up*. Formování relace taktéž proběhne ve fázi *continue*.

Algoritmus 2 zjednodušeně popisuje aktualizaci vektorových hodin procesu, který provádí fázi *continue*. Oproti synchronizaci pomocí zámků přibyl řádek 2, který hledá vhodnou operaci *up*, a řádek 3, kdy je nutné znát vektorový čas procesu v době provádění dané operace. Samotná aktualizace na řádku 4 zůstává stejná.

Algoritmus 2 Algoritmus pro formování *hb*-relace.

Vstup: semafor s s frontou $s.q$ operací *up*, proces p_d provádějící fázi *continue*, vektorové hodiny vc_d procesu p_d

Výstup: aktualizované hodiny vc'_d procesu p_d

1: **for** $up \in s.q$ **do**
2: **if** up uvolnil $p_d \vee up$ způsobil neblokující *down* procesu p_d **then**
3: $vc_u = up.vc()$
4: $vc'_d = vc_d \sqcup vc_u$
5: $s.q.remove(up)$
6: **break**

4. Experimenty

Implementace navržených řešení v nástroji ANaConDA byla ověřena sadou automatických testů, které využívají tři speciální analyzátory implementované za tímto účelem (a nedetekují žádné paralelní chyby). Pro detekci časově závislých chyb nad daty ve víceprocesových programech byla vytvořena nová implementace analyzátoru AtomRace, která je téměř totožná s implementací pro vícevláknové programy, ale využívá překlad logických adres a sdílené API pro komunikaci procesů. S tímto analyzátorem byly provedeny experimenty nad studentskými projekty z předmětu Operační systémy.

K dispozici bylo 19 projektů, které obdržely maximální bodové hodnocení, protože v nich klasické testy neodhalily žádnou chybu. V 17 z těchto projektů chybu nenašel ani AtomRace, ale ve dvou z nich úspěšně souběh ve sdílené paměti detekoval. Nástroj ANaConDA umí získávat informace o průběhu stopy (*backtrace*), který ulehčuje hledání chyby ve zdrojovém programu. Pomocí sdíleného API byla tato funkcionalita implementována i do analyzátoru AtomRace. Výstup analýzy pak vypadá podobně jako na ukázce:

```
Data race on memory address
0x7f5a6f008000 detected.
  Process 19298 read from <unknown>
    accessed at line 412 in file x.c
  Process 19243 written to <unknown>
    accessed at line 416 in file x.c
```

Získávání názvu proměnné zatím není podporováno, proto se ve výpisu vyskytuje `<unknown>`. Uživateli jsou tedy sděleny přímo řádky kódu, kde k chybě došlo, a může tak chybu snáze opravit. V budoucnu se plánuje poskytnutí nástroje ANaConDA s rozšířením pro analýzu procesů právě studentům tohoto kurzu, aby jim pomohla při vývoji projektů. Informace pro nalezení chyby pro ně tedy budou klíčové.

5. Závěr

Výsledkem práce je unikátní rozšíření pro detekci paralelních chyb ve víceprocesových programech, které žádný ze známých nástrojů pro dynamickou analýzu paralelních programů neposkytuje. Článek popisuje problémy, ke kterým u procesů dochází, a zjednodušený návrh jejich řešení. Nástroj ANaConDA tak vývojářům analyzátorů poskytuje sdílené datové typy, překlad logických adres, potřebná zpětná volání, monitorování synchronizace pomocí semaforů a také vzorovou implementaci formování *hb*-relace. Některé z těchto funkcionalit byly úspěšně využity v algoritmu

AtomRace, který detekoval časově závislou chybu nad daty ve studentských projektech, kterou klasické testy nenašly.

V blízké budoucnosti bude rozšířena sada automatických testů pro ověření funkcionality rozšíření i na okrajových případech. Také budou provedeny další experimenty. Pro případnou navazující práci se nabízí hned několik rozšíření. Je možné dodat podporu pro programy, které kombinují procesy i vlákna, pro distribuované systémy, nebo pro jiné formy analýzy (např. zmíněná centralizovaná analýza). Popsané rozšíření také nepodporuje Windows semafory a není definována *hb*-relace pro operace *up* a *down* o více než 1. Také je třeba vytvořit další analyzátory využívající implementované rozšíření (např. implementovat analyzátor FastTrack využívající *hb*-relaci).

Poděkování

Ráda bych poděkovala zejména Ing. Aleši Smrčkovi, Ph.D. a Ing. Janu Fiedorovi, Ph.D., za jejich vedení a rady po celou dobu mé práce na nástroji ANaConDA.

Literatura

[1] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[2] Cormac Flanagan and Stephen Freund. The roadrunner dynamic analysis framework for concurrent programs. *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 1–8, January 2010.

[3] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 42 of *PLDI '07*, pages 89–100, New York, NY, USA, 2007. ACM.

[4] Jan Fiedor and Tomáš Vojnar. Anaconda: A framework for analysing multi-threaded C/C++ programs on the binary level. In *Proc. of 3rd International Conference on Runtime Verification—RV'12*, volume 7687 of LNCS, pages 35–41, Istanbul, Turkey, 2012.

[5] Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. Atomrace: Data race and atomicity violation detector and healer. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '08, pages 7:1–7:10, New York, NY, USA, 2008. ACM.

[6] Jan Fiedor, Monika Mužikovská, Aleš Smrčka, Ondřej Vašíček, and Tomáš Vojnar. Advances in the anaconda framework for dynamic analysis and testing of concurrent c/c++ programs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 356–359, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[8] Cormac Flanagan and Stephen Freund. Fasttrack: efficient and precise dynamic race detection. In *Communications of the ACM*, volume 53(11), pages 93–101, 01 November 2010.

[9] Ricardo J. Dias, Carla Ferreira, Jan Fiedor, João M. Lourenço, Aleš Smrčka, Diogo G. Sousa, and Tomáš Vojnar. Verifying concurrent programs using contracts. In *2017 IEEE International Conference on Software Testing, Verification and Validation*, pages 196–206, 2017.

[10] Rahul Agarwal and Scott Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. pages 51–60, 01 2006.