VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

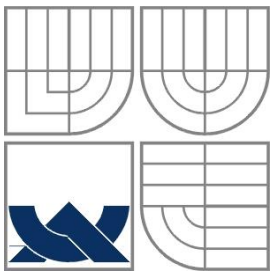
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

TESTOVÁNÍ VÍCEVLÁKNOVÝCH PROGRAMŮ POMOCÍ ŠUMU

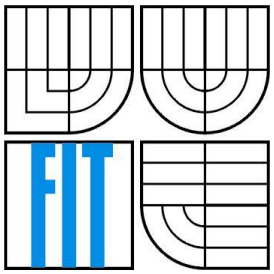
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAN KOTYZ
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

TESTOVÁNÍ VÍCEVLÁKNOVÝCH PROGRAMŮ POMOCÍ ŠUMU

TESTING OF CONCURRENT PROGRAMS WITH NOISE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN KOTYZ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK LETKO, Ph.D.

BRNO 2015

Abstrakt

Cílem této bakalářské práce je vytvoření nástroje pro testování vícevláknových programů vytvořených v jazyce Python. Tento nástroj pro testování sleduje běh testovaného vícevláknového programu, pomocí instrumentace bajtkódu, a na vybraných místech provádí vkládání šumu. Tím výrazně napomáhá ke zvýšení pravděpodobnosti projevu chyb a umožňuje tak efektivnější odhalení chyb typických pro vícevláknové programy. Výsledkem této práce je funkční nástroj pro testování vícevláknových programů v Pythonu.

Abstract

The aim of this bachelor thesis is to create a tool for testing of concurrent software written in Python. This testing tool monitors run of the concurrent program with bytecode instrumentation and performs noise injection at selected locations. This results in a dramatic increase in the probability of bug manifestation and therefore allows more efficient detection of bugs typical for concurrent software. The result of this thesis is concurrency testing tool for Python.

Klíčová slova

Python, testování, vícevláknové programy, šum, chyby vícevláknových programů

Keywords

Python, software testing, concurrency, multithreaded programs, noise injection, concurrency bugs

Citace

KOTYZ Jan: Testování vícevláknových programů pomocí šumu, bakalářská práce, Brno, FIT VUT v Brně, 2015

Testování vícevláknových programů pomocí šumu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Zdeňka Letka, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jan Kotyz
20. května 2015

Poděkování

Děkuji svému vedoucímu Ing. Zdeňku Letkovi, Ph.D. za jeho rady, připomínky a čas, který mi při vypracovávání této bakalářské práce věnoval.

Obsah

1 Úvod

Vícejádrové procesory jsou v dnešní době naprosto běžně využívány jak v osobních počítačích, tak v mobilních zařízeních. S jejich rozvojem samozřejmě dochází k velkému rozmachu programů, které umí vícejádrové procesory plně využít a pracovat paralelně. Tímto paralelním zpracováním dochází na vícejádrových procesorech k celkovému urychlení běhu programu. V případě, že je dispozici pouze procesor s jedním jádrem, paralelismus je simulován přepínáním kontextu, které řídí plánovač operačního systému, a program se navenek tváří, jako by běžel paralelně. Tím je umožněn alespoň multitasking.

Tvorba paralelně běžících programů ovšem zdaleka není v běžných programovacích jazycích jednoduchou záležitostí. S tím také souvisí zvýšená pravděpodobnost výskytu chyb v těchto programech. Na rozdíl od běžného sekvenčního programování je potřeba řešit synchronizaci mezi souběžně běžícími procesy a vlákny. Při špatné nebo žádné synchronizaci může docházet k chybám, které se v sekvenčních programech nevyskytují. Tyto chyby se velice špatně nalézají. Projevují se jen zřídka, případně jen na některých architekturách počítačů. V případě zjištění, že je v programu chyba, bývá velice složité najít příčinu chyby a její reprodukování bývá také velice obtížné.

Cílem této práce je tedy navrhnout a implementovat nástroj pro testování vícevláknových programů napsaných v jazyce Python, který pomocí vkládání šumu do testovaného programu, umožní mnohem efektivnější nalézání chyb ve vícevláknových programech a výrazně zvýší pravděpodobnost, že se tyto chyby projeví, aby mohly být následně opraveny. Vkládaný šum pozastavuje nebo zpomaluje běh vybraných běžících vláken, čímž způsobuje výrazně rozdílné proložení běhu vláken. Tímto způsobem je možné dostat se do stavů, do kterých by se program dostával jen zcela výjimečně.

V následující kapitole této bakalářské práce je obsaženo seznámení s programovacím jazykem Python a jeho možnostmi tvorby vícevláknových programů. V kapitole 3 jsou popsány typické chyby vícevláknových programů a v kapitole 4 jsou potom zmíněny možné přístupy k testování vícevláknových programů. Potom v další kapitole se již nachází text popisující návrh vytvářeného nástroje pro testování vícevláknových programů. A v 7. kapitole je zdokumentováno, jak je vytvářený nástroj implementován. V předposlední kapitole jsou potom zmíněny experimentální výsledky použití tohoto vytvářeného nástroje, na kterých lze vidět přínosnost tohoto nástroje. V závěru této bakalářské práce se poté nachází shrnutí dosažených výsledků a nástin dalšího možného vývoje implementovaného nástroje.

2 Python

Python [1, 10] je interpretovaný, objektově-orientovaný programovací jazyk. Řadí se mezi skriptovací jazyky a z hlediska typování je to dynamický, silně typovaný jazyk. Mezi jeho hlavní přednosti patří jeho jednoduchá syntaxe a obsáhlé standardní knihovny, obsahující téměř vše, co může vývojář k tvorbě aplikací potřebovat.

Původní implementace interpretu jazyka Python, taky známá jako CPython [12], je implementována v jazyce C a jedná se o nejběžnější implementaci interpretu jazyka. Pokud se někdy mluví jako o Python programu, téměř vždy se jedná o CPython implementaci a jinak to není ani v této bakalářské práci. Kromě CPython implementace existuje i spousta dalších alternativních implementací, z nichž neznámějšími jsou: Jython, IronPython, PyPy. [12]

- Jython - zdrojový kód je interpretován v Java Virtual Machine (JVM).
- IronPython - zdrojový kód je interpretován v prostředí .NET.
- PyPy - zdrojový kód je interpretován v interpretu napsaném v Pythonu.

Jazyk Python je v současné době rozdělen na dvě vývojové větve, verzi 2 a verzi 3, které mezi sebou navzájem nejsou plně kompatibilní. V současné době vývoj probíhá hlavně nad verzí 3, ale verze 2 je stále velmi rozšířená a stále podporována.

Python je vyvíjen jako Open Source, aktuálně pod licencí the Python Software Foundation – PSF, která je kompatibilní s licencí GPL.[2]

2.1 Datový a paměťový model

Pro pochopení souběžného zpracování a možných výskytů chyb při něm, je potřeba vědět, jak Python pracuje s pamětí, která může být sdílena mezi jednotlivými vlákny. A jak pracuje s proměnnými, které se do této paměti odkazují. V této kapitole se proto nachází krátký přehled.

Datový model

V Pythonu jsou všechna data reprezentována jako objekty. Každý objekt má svou identitu, typ a hodnotu. Identita objektu představuje adresu objektu v paměti. K zjištění identity objektu slouží vestavěná funkce `id()`. Typ objektu potom určuje, které operace daný objekt podporuje a také definuje možné hodnoty pro objekty tohoto typu. Ke zjištění typu objektu slouží vestavěná funkce `type()`. Jak identita, tak typ objektu jsou neměnné a nelze je po vytvoření objektu měnit. Hodnota objektu se měnit může v závislosti na tom, zda se jedná o objekt měnitelný (mutable) nebo neměnitelný (immutable). Jestli je objekt měnitelný nebo neměnitelný závisí na typu objektu. Mezi měnitelné typy patří například slovník (dictionary) nebo seznam (list), mezi neměnitelné se pak řadí například číselné typy, řetězce (strings) a n-tice (tuples).

Paměťový model

Na rozdíl od jiných programovacích jazyků, virtuální stroj Pythonu ukládá všechny objekty a datové struktury pouze na svůj privátní prostor na haldě (heap). Správa paměti je prováděna automaticky interpretem. O uvolňování paměti, kterou jednotlivé objekty zabírají se stará garbage collector, který funguje na principu počítání referencí.

Z každého vlákna lze tedy obecně přistupovat k jakémukoli objektu. Vyjímku tvoří pouze třída `threading.local`, která umožňuje vytvořit úložiště specifické pro každé vlákno.

Viditelnost proměnných

V Pythonu je viditelnost proměnných rozdělena na lokální a globální jmenný prostor. Proměnné, které jsou uloženy v lokálním jmenném prostoru, jsou přístupné pouze zevnitř bloku, v němž byly definovány. Ke globálním proměnným lze přistupovat ze všech nižších úrovní zanoření. Pro rozlišení jestli se bude proměnná dohledávat v lokálním nebo globálním paměťovém prostoru slouží klíčové slovo `global`, kterým je určeno, že se jméno proměnné odkazuje na globální jmenný prostor. Vždy pokud v lokálním jmenném prostoru není uložen název proměnné, dohledává se následně proměnná v globálním jmenném prostoru. V případě, že není název proměnné uložen ani v globálním jmenném prostoru, je vyvolána výjimka. [10]

2.2 Souběžný přístup

Jako téměř všechny moderní programovací jazyky dnešní doby, i Python umožňuje programování vícevláknových programů a programů s více procesy. Tato bakalářská práce je zaměřena především na testování vícevláknových programů, ale pro přehled možností implementace souběžného přístupu, které Python umožňuje, jsou v této kapitole zmíněny i další formy souběžnosti.

2.2.1 Vlákna

Vlákna jsou tzv. odlehčené procesy a běží pouze v rámci jednoho procesu. Vlákna mezi sebou, na rozdíl od procesů, sdílí svůj globální paměťový prostor, který je uložený na haldě. Vytvoření a přepínání mezi vlákny je časově a paměťově méně náročné, než u procesů. Předpokládá se tedy, že při použití vláken by měl program pracovat rychleji, než při použití procesů. Bohužel toto v Pythonu nemusí vždy platit, protože ve standardní implementaci interpretu Pythonu (CPythonu) je používán globální zámek interpretu-GIL (popsán v kapitole 2.4). Vlákna je tedy vhodné používat v případě, že je potřeba zpracovávat vstupně-výstupní operace. Na náročné paralelní výpočty je potom vhodnější zvolit použití více procesů.

Pro vytváření vláken jsou ve standardní knihovně dvě možnosti. Buďto je možno použít modul `threading` nebo modul `thread`. Doporučeno je ovšem použití modulu `threading` [3].
Modul `threading` implementuje API na vyšší úrovni, než modul `thread` a sám interně využívá funkce z modulu `thread`. Modul `threading` je implementován v Pythonu, zatímco `thread` je implementován v jazyce C. Z hlediska možnosti využití synchronizačních prostředků toho modul `threading` také nabízí více, než modul `thread`. Modul `thread` nabízí pouze klasický zámek (lock), ale v modulu `threading` jsou implementovány i další synchronizační prostředky, zmíněné v podkapitole 2.3 [1].

2.2.2 Procesy

Procesy jsou robustnější, než vlákna. Na rozdíl od vláken má každý proces svůj vlastní paměťový prostor a procesy mezi sebou mohou komunikovat pouze některou z forem mezi-procesové

komunikace. Ve standardní knihovně lze nalézt pro práci s procesy modul `multiprocessing`, který poskytuje prakticky stejné rozhraní, jako modul `threading`. Tím může být zjednodušeno předělání programu pracujícího s vlákny na program pracující s procesy a naopak. Pro komunikaci mezi procesy se v Pythonu používají fronty `Queue` nebo roury `Pipe` z modulu `multiprocessing` [1].

2.2.3 Koprogramy (coroutines)

Dalším možným přístupem k vytvoření souběžnosti v programech jsou koprogramy. Pro koprogramy se někdy používají názvy jako *microthreading, greenlets, green threads, tasklets* atd. Koprogramy jsou často používány v programech, které obsluhují velké množství vstupně-výstupních operací. Například u serveru, který souběžně obsluhuje vysoké množství připojených klientů, je vhodnější upřednostnit použití koprogramů před vlákny, protože každé vlákno vyžaduje své vlastní systémové prostředky a určitou režii spojenou s přepínáním mezi vlákny, jejich synchronizací apod. [8].

2.3 Synchronizační primitiva

Aby ve vícevláknových programech nedocházelo k chybám způsobeným současným přístupem více vláken ke sdíleným zdrojům, je potřeba tato vlákna nějakým způsobem synchronizovat. K tomu slouží použití synchronizačních primitiv, které umožňují výlučný přístup těchto vláken ke sdíleným zdrojům. Dříve zmíněné moduly `threading` a `multiprocessing` umožňují využití různých typů synchronizačních primitiv. Jednotlivá synchronizační primitiva jsou popsána níže. Ovšem pro bezpečnou synchronizaci na vyšší úrovni je doporučeno místo těchto prostředků použít fronty `Queue`, které synchronizaci řeší interně samy [1].

- **Zámek (Lock)**. Základní a nejjednodušší synchronizační prostředek. Umožňuje pouze dva stavy – zamčený a odemčený. Pro zamknutí zámku slouží metoda `acquire()`, pro odemknutí se používá metoda `release()`. Ve chvíli, kdy je zámek zamčený, není umožněno zámek znova zamknout a vstoupit tak do kritické sekce, dokud není zámek opět uvolněn. U metody `acquire([blocking])` lze volitelným argumentem specifikovat, zda bude blokující. Pokud je neblokující, je umožněno vstoupit do kritické sekce i přes zamčený zámek. Metoda `acquire()` vrací po zavolání hodnotu `False` v případě, že je volána nad zamknutým zámkem. V opačném případě vrací hodnotu `True` a uvede zámek do zamčeného stavu.

- **Vícenásobně přístupný zámek (Reentrant Lock).** Funguje naprosto stejně jako obyčejný zámek s tím rozdílem, že je vláknu, které zámek uzamklo umožněno opakovaně procházet přes zamčený zámek. Často se této vlastnosti využívá v rekurzivních funkcích.

- **Semafor (Semaphore).** Semafor využívá svůj vnitřní čítač, který je inicializován při vytvoření semaforu. Hodnota čítače je vždy snížena o 1 při každém vstupu do kritické sekce a navýšena o 1 při opuštění kritické sekce. Pokud je hodnota čítače rovna 0, není umožněno do kritické sekce vstoupit, dokud se hodnota čítače opět nezvýší opuštěním kritické sekce některým vláknem. Semafory tak jako zámky používají metody `acquire()` a `release()`. Metoda `acquire()` může nebo nemusí být blokující, tak jako u zámků.

- **Událost (Event).** Událost je jedna z nejjednodušších mechanismů pro komunikaci mezi vlákny. Jedno vlákno zasílá signál (všem) a další vlákno nebo vlákna čekají a nic nevykonávají, dokud neobdrží signál zaslaný vysílajícím vláknem. Událost používá metody `wait()`, `set()`, `clear()`.

- **Podmíněná proměnná (Condition).** Tento synchronizační prostředek podmíněné proměnné umožňuje čekání jednoho nebo více vláken, dokud nejsou probuzena signálem z jiného vlákna. Na rozdíl od Události je možno probudit zvolený počet vláken, ne všechny čekající. Používány jsou metody `wait()`, `notify()`, `notifyAll()`, `acquire()`, `release()`.

- **Bariéra (Barrier).** Bariéra je synchronizační prostředek, který pozastavuje práci vlákna, které doběhlo k bariéře, dokud k ní nedoběhnou i všechna zbývající vlákna. Poté je všem zároveň umožněno dále pokračovat ve vykonávání kódu. Bariéra je v Pythonu implementována od verze 3.2.

2.4 Globální zámek interpretu

Globální zámek interpretu (dále jen GIL) [13, 14, 15] je mutex, který brání spuštění bajtkódu vykonávaného programu ve více vláknech najednou. V interpretu tedy může být v dané chvíli aktivní vždy jen jedno systémové vlákno. Důvod, proč je GIL v CPythonu implementován, přestože brání paralelnímu využití vláken, je pro zabezpečení správy paměti, která by díky současné implementaci interpretu nebyla z hlediska současného přístupu více vláken bezpečná (thread-safe).

V některých alternativních interpretech jazyka Python využití GIL není potřeba, díky rozdílné implementaci interpretu (Jython, IronPython, …).[13]

GIL je vždy uvolněn, pokud dojde k vstupně-výstupní operaci nebo uspání vlákna. Aby se nestávalo, že by jedno vlákno drželo GIL po celou dobu svého běhu, například v případě, že by nepoužívalo žádné vstupně-výstupní operace, tak je v CPythonu implementováno automatické uvolnění GIL každých 100 tiknutí (angl. tick). Jedno tiknutí ve většině případů odpovídá jedné operaci bajtkódu, ale existuje pár výjimek, kdy toto neplatí. Poté co těchto 100 tiknutí proběhne, je běh aktuálního vlákna pozastaven a dochází k uvolnění GIL a signalizaci dalším vláknům, že je GIL volný. V této chvíli vlákna soutěží o to, které GIL první zabere, kdy původně běžící vlákno je oproti ostatním vláknům zvýhodněno, protože nemuselo čekat na probuzení signálem. Tím je způsobeno, že ke změně kontextu dochází jen výjimečně. Defaultní hodnotu 100 tiknutí je možné změnit pomocí nastavení `sys.setcheckinterval()`. [15]

3 Chyby vícevláknových programů

Vícevláknový program se vyznačuje tím, že se v programu nachází alespoň 2 nebo více vláken, které běží souběžně. Díky tomuto souběžnému běhu se mohou vyskytovat různé druhy chyb jako např. časově závislé chyby nad daty (race conditions), narušení atomicity (atomicity violation), uváznutí (deadlock) apod. Projevy těchto chyb mohou být různé, můžou se projevovat navracením nesprávných výsledků, vyvoláváním nečekaných výjimek, uváznutím nebo blokováním programu. Těmto chybám vyskytujícím se převážně ve vícevláknových programy lze předcházet správným použití synchronizačních primitiv. S použitím synchronizačních primitiv ovšem dochází k určitému zpomalení běhu vícevláknového programu. Chyby vícevláknových programů lze rozdělit do dvou kategorií. Na chyby v bezpečnosti a chyby v živosti. V této kapitole jsou jednotlivé chyby popsány podle informací čerpaných z odborné publikace [4].

3.1 Chyby bezpečnosti

Při výskytu některé z chyb bezpečnosti, dochází v programu k tomu, že nastane něco špatného. Tyto chyby mají vždy nějaký konečný protipříklad. Mezi tento typ chyb se řadí následující chyby.

1. **Časově závislá chyba nad daty (data race) -** Časově závislé chyby nad daty jsou jedny z nejčastějších chyb vyskytujících se ve vícevláknových programech. Tyto chyby se velice obtížně odhalují a reprodukují, jelikož se mohou projevovat s velice nízkou pravděpodobností. K časově závislé chybě nad daty dochází ve chvíli, kdy ke sdílené proměnné souběžně přistupují dvě nebo více vláken, a alespoň jedno z vláken provádí zápis do této proměnné. Zároveň vlákna nepoužívají žádný synchronizační prostředek pro výlučný přístup k této proměnné.

2. **Porušení atomicity (atomicity violation) -** Porušení atomicity nastává ve chvíli, kdy operace nebo sekvence operací, které se zdánlivě jeví jako atomické, ve skutečnosti atomické nejsou nebo nejsou atomické na stroji s jinou architekturou.

3. **Chyba nesprávného pořadí (order violation) -** K této chybě dochází ve chvíli, kdy se pořadí operací provádí v jiném, než očekávaném pořadí. Například čtení ze souboru před jeho otevřením.

4. **Uváznutí (deadlock) -** K uváznutí dochází ve chvíli, kdy je každé z vlákno z množiny vláken pozastaveno a čeká na událost, která by mohla nastat pouze, pokud by mohlo pokračovat některé z vláken z dané množiny. Jedná se o cyklickou závislost mezi těmito vlákny z dané množiny.

5. **Zmeškání signálu (missed signals) -** K chybě zmeškání signálu dochází, jestliže signál zaslaný jedním vláknem vláknu druhému, není danému druhému vláknu (nebo více vláknům) doručen. Tím bývá často způsobeno zablokování vlákna, ke kterému není signál doručen.

3.2 Chyby živosti

Při chybách živosti dochází v programu k tomu, že je bráněno něčemu dobrému, aby nastalo. Tyto chyby většinou nemají konečný protipříklad.

1. **Uváznutí s aktivním čekáním (livelock) -** Při uváznutí s aktivním čekáním dochází k nekonečnému běhu programu. Vlákno neustále něco provádí (není nijak blokováno), ale nedostává se k žádnému pokroku.

2. **Blokované vlákno (blocked thread) -** K chybě trvale blokovaného vlákna dochází, když je vlákno blokováno a čeká na nějakou událost, která by ho odblokovala. Ta ovšem nikdy nenastane.

4 Testování vícevláknových programů

Testování je nejběžněji používaným přístupem k ověřování kvality software. Jedná se o spouštění testovaného programu, kdy je ověřováno podle specifikace programu, zda zadaným vstupům odpovídají výstupy. Testování je samo o sobě dosti náročnou disciplínou, ale testování vícevláknových programů je mnohem náročnější. Je tomu tak díky tomu, že není umožněno dopředu předvídat, jak plánovač operačního systému naplánuje pořadí běhu vláken. V případě testování vícevláknových programů se tedy jedná o nedeterministický proces. To způsobuje, že není zaručeno, že při každém spuštění vícevláknového programu se stejným vstupem, bude vždy stejný výsledek na výstupu. Při testování programu s více vlákny tedy nelze test spouštět pouze jednou a z toho vyvozovat závěry, ale je potřeba testy nad testovaným programem spouštět opakovaně, aby bylo dosaženo co nejvíce různých možných proložení vláken.

4.1 Zátěžové testování

Při zátěžovém testování vícevláknových programů se testy nad testovaným programem spouští s vysokým množstvím vláken, které soutěží o přístup ke sdíleným prostředkům. Tímto přístupem může být mírně zvýšena pravděpodobnost projevu chyby v programu. Toto navýšení pravděpodobnosti však zdaleka není tak výrazné, jako v následujících popsaných přístupech. Tato metoda je výpočetně náročná a zvýšení pravděpodobnosti nalezení případné chyby není příliš výrazné. [5]

4.2 Vkládání šumu

Vkládání šumu[5] je další z možností testování vícevláknových programů. Šum pozastavuje nebo zpomaluje běh vybraných běžících vláken, čímž způsobuje výrazně rozdílné proložení běhu vláken. Tímto způsobem je možné dostat se do stavů, do kterých by se program dostával jen zcela výjimečně.

Šum může být tvořen více různými způsoby:

- krátké uspání vlákna
- uvolnění procesu instrukcí yield (v Pythonu má yield jiný význam, lze nahradit uspáním vlákna na nulovou dobu `time.sleep(0)`) [14]
- prováděním nadbytečných instrukcí
- synchronizací s jiným vláknem, které je nad rámec testovaného programu

Místa v programu, kam je vkládán šum, mají spojitost s událostmi, které jsou z hlediska vícevláknových programů zajímavé (synchronizace vláken, přístup ke sdílené paměti). Rozhodnutí, zda se na těchto místa bude šum vkládat, může být založeno na náhodě nebo na nějaké sofistikovanější heuristice, se kterou je poté zpravidla dosahováno lepších výsledků testování.

Tento přístup je relativně levný a bývá s ním dosaženo mnohem lepších výsledků, než u zátěžového testování. Ani testování pomocí vkládání šumu ale nezaručuje pokrytí všech možných proložení vláken. Z existujících nástrojů používajících tento přístup stojí za zmínku IBM Java Concurrency Testing Tool - ConTest nebo Microsoft Driver Verifier [5].

4.3 Deterministické plánování

Při tomto přístupu se využívá deterministického plánovače, kdy se testovaný program spouští postupně se všemi možnými proloženími vláken. Jelikož možností proložení vláken je enormně mnoho a s délkou programu nebo s každým dalším vláknem počet možných proložení vláken exponenciálně narůstá, je tento přístup omezen rozsahem testovaného programu a výpočetním výkonem stroje, na kterém testování probíhá. Nespornou výhodou tohoto přístupu je, že testy lze, díky deterministickému plánovači, jednoduše replikovat. Přístupu deterministického plánování je využíváno například v nástrojích jako CHESS od Microsoftu nebo v Maple [5].

4.4 Dynamická analýza

Při použití dynamické analýzy je testovaný program za běhu sledován a jsou sbírány různé informace o tomto běžícím programu. Tyto sbírané informace jsou následně analyzovány za účelem nalezení možných chyb, které mohou být obsaženy v programu, ale nemuselo k nim při běhu dojít. Efektivnost dynamické analýzy lze zvýšit tím, že je procházeno více různých cest běhu programu. Toho lze dosáhnou například současným použitím s technikou vkládání šumu nebo s deterministickým plánováním [5].

4.5 Metriky pokrytí

Metriky pokrytí poskytují informace, jak dobře byl testovaný program otestován podle zvoleného kritéria. Tím je možné odhalit, které části testovaného programu nebyly dosud testovány, a rozhodnout tak, zda se s testováním již může skončit, nebo zdali je potřeba přidání dalších testovacích případů, kterými by bylo dosaženo otestování dosud netestovaných částí testovaného programu [5].

Metriky pokrytí běžně používané a využívané v sekvenčních programech, jako například metrika pokrytí všech příkazů, poskytují dobrý přehled, které části v kódu byly při testování pokryty. Mají velmi malou vypovídající hodnotu z pohledu vícevláknových programů. Na rozdíl od sekvenčních programů, kdy nemá smysl spouštět jednotlivé testy opakovaně, protože dopadnou vždy stejně, se u testování vícevláknových programů testy opakovaně spouští se snahou dosáhnout co největšího pokrytí všech možných proložení vláken. Proto je potřeba použití jiných metrik pokrytí než těch, které se používají pro testování sekvenčních programů. Je potřeba aby používané metriky pokrytí zohledňovaly běh vícevláknových programů. Některé z metrik pokrytí, vhodných pro použití ve vícevláknových programech jsou zmíněny níže [5].

4.5.1 Metrika pokrytí synchronizace

Tato metrika pokrytí se zaměřuje na pokrytí chování synchronizačních primitiv. Jednotlivé chování, zkoumané touto metrikou pokrytí, se liší v závislosti na zvoleném synchronizačním primitivu. Dále je tedy rozebrána aplikace této metriky pokrytí pouze pro zámky. U zámků jsou sledovány 3 možné režimy, ve kterých může zámek být (*navštívený, blokující, blokovaný*). Případ *navštívený* je pokryt kdykoliv běh programu dojde k místu, kde se zamyká zámek. Zbývající 2 případy jsou už pak závislé na aktuálním stavu daného zámku v době, kdy k nim právě běžící vlákno dorazilo. Když vlákno *v1* dorazí k zámku *Z*, který brání vstupu do bloku *A*, který je chráněný zámkem *Z* a zastaví se, protože daný zámek *Z* momentálně drží vlákno *v2*, které se momentálně nachází v bloku *B*, který je chráněn stejným zámkem *Z* jako blok *A*, je blok *A* označen jako *blokovaný* a blok *B* jako *blokující*. Bloky *A* a

B se mohou nacházet na stejném místě v programu. Informace získané z této metriky pokrytí jsou potom uchovávány jako dvojice (*místo v programu, režim zámku*). [5]

5 Návrh

V této kapitole je popsán návrh vytvářeného nástroje pro testování vícevláknových programů. V první části této kapitoly jsou popsány požadavky na implementovaný nástroj, které vyplývají ze zadání této bakalářské práce. Dále je potom popsáno, jak vytvářený nástroj pro testování vlastně funguje.

5.1 Požadavky na vytvářený nástroj

Vytvářený nástroj pro testování vícevláknových programů musí zvládat sledovat chod vícevláknového programu. Díky tomuto sledování pak může na vybraných místech generovat šum a sbírat data pro měření zvolené metriky pokrytí. Samotný nástroj bude možné spouštět a konfigurovat přes příkazovou řádku, což umožní použití tohoto nástroje pro skriptování.

Chování nástroje bude záviset na zadaných parametrech, kdy budou 4 hlavní možnosti spuštění nástroje a to: spuštění procesu instrumentace, spuštění odinstrumentace, zobrazení výsledků měření pokrytí a opakované spouštění instrumentovaného testovaného programu.

Při spuštění procesu instrumentace bude možno nastavit, které soubory se budou instrumentovat, dále půjde nastavit parametry šumu a půjde vypnout měření metriky pokrytí. Pro zobrazení výsledků měření bude možnost volby, jestli zobrazit souhrn výsledků do terminálu a nebo do HTML souboru, který zobrazuje podle pokrytí jednotlivé synchronizační bloky barevně na zdrojovém kódu. Při volbě opakovaného spouštění instrumentovaného programu bude možné nastavit, kolikrát po sobě se program spustí.

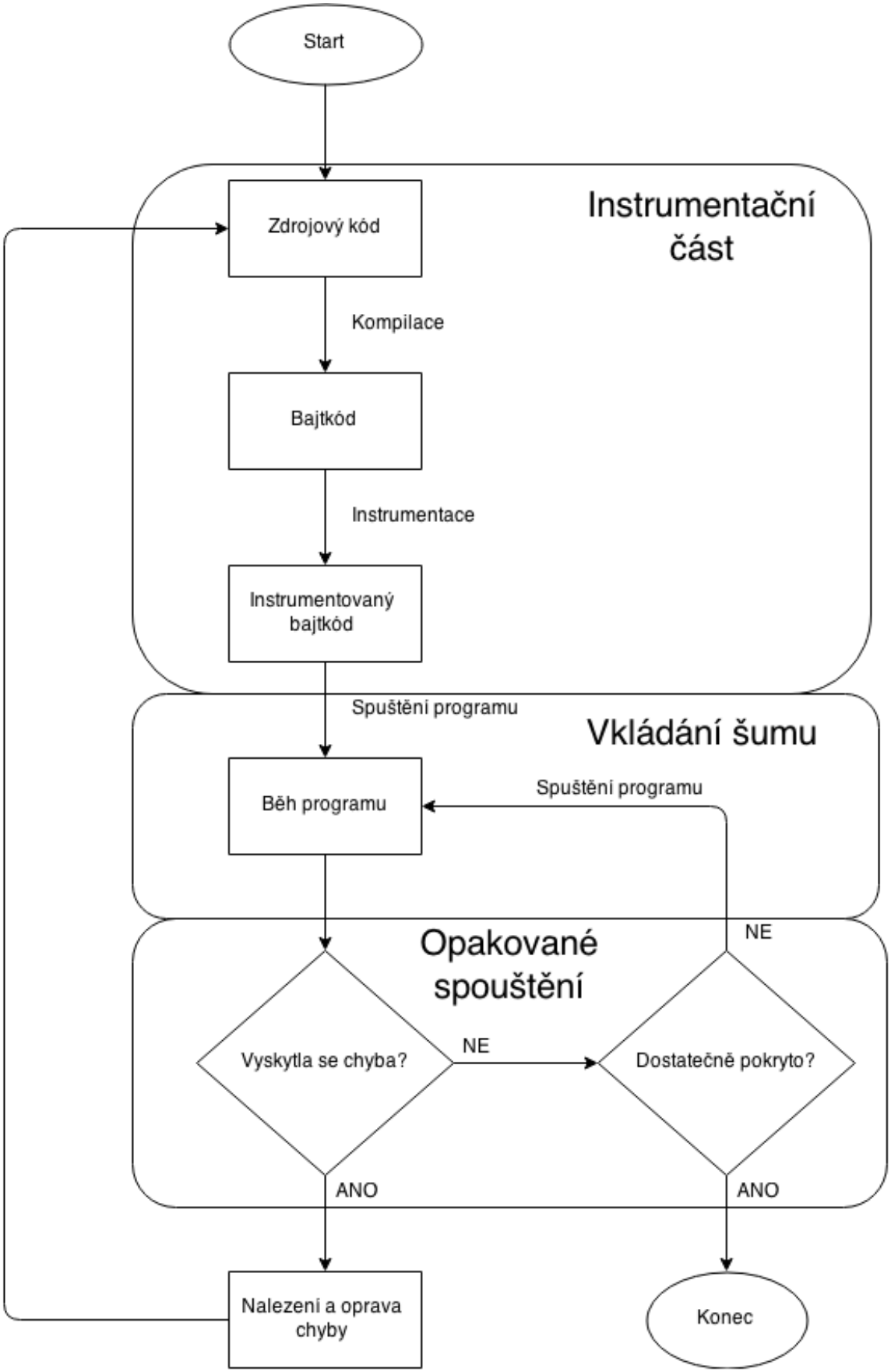
Šum bude generován v okolí použití synchronizačních primitiv nebo v okolí přístupu k datům, která potenciálně mohou být sdílena mezi vlákny. Pro šum půjde nastavit pravděpodobnost vložení šumu u každého místa, kde vkládání může probíhat a také půjde nastavit rozsah délky uspání vlákna.

Nástroj umožní také přidávání různých rozšíření, které bude možno navázat na jednotlivé sledované události pomocí registrování volání dalších přidaných metod ke zvoleným událostem. Funkčnost nástroje bude poté ověřena na sadě 10 vícevláknových programů, kde každý z těchto programů bude obsahovat některý z typu chyb vyskytujících se ve vícevláknových programech.

5.2 Návrh nástroje pro testování

Jak již bylo zmíněno dříve, testování vícevláknových programů se od běžného testování sekvenčních programů liší v tom, že je potřeba testy spouštět opakovaně, aby bylo možné otestovat co nejvíce možných stavů, ve kterých by se testovaný vícevláknový program mohl nacházet. Tomu také odpovídá návrh vytvářeného nástroje.

Navrhovaný nástroj pro testování vícevláknových lze rozdělit na 3 části. První část se stará o instrumentaci testovaného programu, druhá část o vkládání šumu a sledování testovaného programu za běhu a třetí část umožní opakované spouštění testovaného programu. Proces toho, jak testování s vytvářeným nástrojem probíhá je ilustrováno na vývojovém diagramu 5.1.

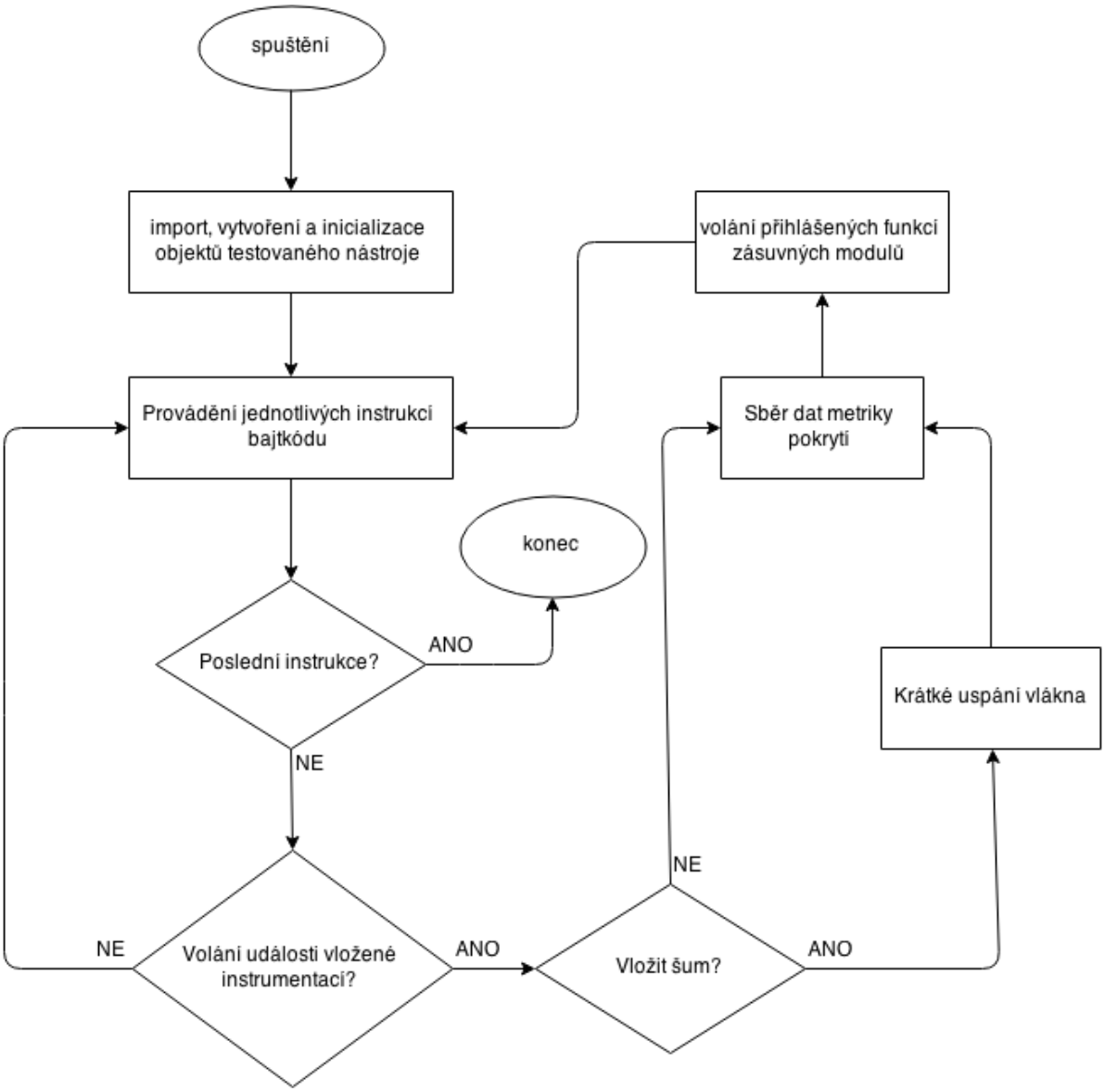


Vývojový diagram 5.1: Průběh testování s nástrojem

Jak lze vidět na vývojovém diagramu 5.1, testování s pomocí vytvářeného nástroje probíhá následovně:

1. V prvním kroku nástroj pro testování vícevláknových programů přeloží všechny zdrojové soubory napsané v Pythonu do bajtkódu, který je uložen v souborech s názvy odpovídajícími zdrojovým souborům a příponou `.pyc`.
2. Nad těmito soubory s bajtkódem potom probíhá proces instrumentace, což je jednou z hlavních funkcí vytvářeného nástroje pro testování vícevláknových programů. Co to instrumentace vlastně je a jak se využívá v tomto nástroji je přesněji popsáno v kapitole 5.3.
3. Když je již testovaný program úspěšně instrumentován, je možno jej spustit a testovat, zda se neprojeví chyba. V případě, že byla v testovaném programu nalezena chyba. Je potřeba tuto chybu opravit ve zdrojových kódech a následně znovu program přeložit, instrumentovat a opakovaně spouštět. Rozpoznání a následná oprava chyby je na programátorovi.
4. Toto spouštění testovaného programu se provádí opakovaně, aby bylo dosaženo co nejvíce možných stavů, ve kterých by se testovaný vícevláknový program mohl nacházet. Rozhodnutí, kdy s opakovaným spouštěním testovaného programu skončit, je možné učinit například na základě míry pokrytí testovaného programu metrikou pokrytí.

První 2 body odpovídají instrumentační části, třetí bod souvisí s částí běhu programu, který je dále názorně ilustrován ve vývojovém diagramu 5.2. A poslední bod souvisí s třetí části, která umožňuje opakované spouštění.



Vývojový diagram 5.2: Fungování při běhu programu

Na vývojovém diagramu 5.2 lze vidět, že na začátku běhu již instrumentovaného programu je vždy importován modul testovacího nástroje a volána inicializační funkce, která vytvoří potřebné objekty testovacího nástroje. Po této inicializaci následuje standardní běh programu až do chvíle, než se program dostane k místu v programu, které souvisí s vícevláknovým během. V takovémto místě se potom nachází vyvolání události, která je testovacím nástrojem obsloužena. Obsloužení zpravidla obsahuje rozhodování, jestli na dané místo vložit šum nebo ne a v případě události související s měřenou metrikou pokrytí je proveden záznam o navštívení daného místa. Dále v rámci obsloužení vyvolané události můžou být provedeny další operace, které byly dříve registrovány k dané události. Po obsloužení události pokračuje klasický běh programu do výskytu další události, kdy proběhne opět obsluha této události, nebo do konce programu. Jednotlivé události, které mohou být obslouženy testovacím nástrojem, jsou popsány v následující podkapitole 5.2.1.

5.2.1 Sledované událostí

Nástroj pro testování vícevláknových programů umožňuje sledování různých událostí souvisejících s během vícevláknového programu. Na takto sledované události je poté možné navázat i další funkcionalitu pomocí registrování dalších posluchačů (Listeners) na tyto události. Sledovány jsou následující události.

- Přístup ke globálním proměnným
- Přístup k jednotlivým prvkům seznamů (list), n-tic (tuple) a slovníků (dictionary)
- Přístup k atributům tříd
- Zamykání a odemykání zámků – metody `acquire()` a `release()`
- Volání metod semaforů – `acquire()` a `release()`
- Volání metod podmíněných proměnných – `acquire()`, `release()`, `wait()`, `notify()`, `notifyAll()`.
- Volání metod synchronizačních událostí – `wait()`, `set()`, `clear()`
- Spuštění a ukončení běhu vlákna
- Volání metod a návrat z metod

5.2.2 Vkládání šumu

Vkládání šumu může být realizováno buďto jako obsluha vícevláknových událostí, které jsou přidány instrumentací a nebo při samotné instrumentaci, kdy může být šum vkládán přímo do bajtkódu. Ve vytvářeném nástroji pro testování vícevláknových programů je pro vkládání použita první zmíněná možnost, tedy v rámci obsluhy vyvolaných událostí. Výhodou tohoto přístupu je možnost za běhu testovaného nástroje jednoduše měnit nastavení vkládaní tohoto šumu. Další výhodou naproti pevnému vložení do bajtkódu je například možnost vložení jen do některých iterací cyklu.

Vkládaný šum může být reprezentován v různých formách, které byly popsán v kapitole 4.2. V tomto nástroji je zvoleno použití krátkého uspání vlákna, jelikož je jednoduše implementovatelné a při uspání je vždy zaručeno uvolnění GIL a přepnutí na další vlákno, soutěžící o přidělení zdroje. U vložení menšího počtu bezvýznamných instrukcí nemusí být GIL vůbec uvolněn a vložení takového šumu by pak nemělo moc velký význam. Při vložení většího počtu instrukcí, než má GIL nastavený svůj čítač, by již bylo zaručeno chvilkového uvolnění GIL, ale změna běžícího vlákna zaručena není.

Pro rozhodování, na která místa v programu bude šum vkládán, je použito generátoru pseudonáhodných čísel z knihovny `random`, kdy je rozhodováno podle zadané pravděpodobnosti, jestli bude šum vložen nebo ne. O vložení se rozhoduje jen na místech souvisejících s událostmi vícevláknových programů.

5.2.3 Měření metriky pokrytí

Jako sbíraná metrika pokrytí byla zvolena metrika pokrytí synchronizace, která se zaměřuje na sledování použitých stavů synchronizačních primitiv. Tato metrika byla podrobně popsána v sekci 4.5.1. Ve vytvářeném nástroji je podporováno pouze měření této metriky u běžných zámků.

Pro měření této metriky je potřeba sledovat, kdy se program nachází před vstupem do kritické sekce a kdy se nachází v kritické sekci a blokuje tak přístup dalším vláknům. Zároveň je potřeba vždy znát aktuální stavy jednotlivých zámků.

Jelikož mezi vyvoláním události před vstupem do kritické sekce a samotným vstoupením vlákna do kritické sekce může dojít k přepnutí kontextu, je nutné pro sledování aktuálního stavu zámků zajistit správnou synchronizaci mezi událostmi před vstupem do kritické sekce a po vstupu do kritické sekce. Jak je tato synchronizace použita, je zmíněno v kapitole 6.7.

5.3 Instrumentace

Instrumentace je proces vkládání dalších instrukcí do zdrojového programu, které umožní sledovat daný program. Instrumentace může být prováděna na různých úrovních, popsaných níže [6].

Níže jsou popsány možnosti, jakými je možno přistupovat k potřebné instrumentaci z hlediska implementace pro Python a také je názorně ukázáno, jak se jednotlivé přístupy liší. Na ilustraci 5.1 se nachází příklad pseudokódu, který bude následně instrumentován. Na řádku 2 této ilustrace se nachází pro nás zajímavá instrukce (v tomto případě volání funkce).

```
1  ...předchozí kód
2  interesting_event()
3  ...následující kód
```

Ilustrace 5.1: Zdrojový pseudokód před instrumentací

Instrumentace na úrovni zdrojového kódu

Instrumentace na úrovni zdrojového kódu je implementačně nejjednodušší. Jedná se o vložení volání požadované funkce. Často se používá při ladění programu ve formě kontrolních výpisů například na standardní výstup. Pro potřeby vytvářeného nástroje se instrumentace využívá k sledování a následnému zpracování výskytu události související s vícevláknovým programováním. Na ilustraci 5.2 je vidět příklad, jak by instrumentace na úrovni zdrojového kódu vypadala a jak se liší od původního pseudokódu v ilustraci 5.1. Na řádku 3 v ilustraci 5.2 lze vidět původní zajímavá instrukce volání funkce z ilustrace 5.1. Před a za tuto událost, na řádcích 2 a 4 bylo vloženo volání dalších funkcí, které mohou například dále předávat informace o aktuálním stavu programu.

```
1  ...předchozí kód
2  before_interesting_event()
3  interesting_event()
4  after_interesting_event()
5  ...následující kód
```

Ilustrace 5.2: Příklad instrumentace na úrovni zdrojového kódu

Při takovéto úpravě zdrojového kódu je ale potřeba zajistit, aby po ukončení testování byl zdrojový kód vždy navrácen do původního stavu před instrumentací. Zároveň by se muselo zajistit, aby byl zdrojový kód navrácen do původního stavu i v případě, že by při instrumentaci nastala chyba.

Dále je toto řešení problémové v tom, že pro rozpoznání globální proměnné by bylo potřeba využít statického analyzátoru. A dalším možným problémem může také být posun v číslování řádků, kdy nebude odpovídat číslování před a po instrumentaci, což může ztížit čtení například ladících informací. Pro použití ve vytvářeném nástroji pro testování vícevláknových programů tento přístup tedy není vhodný.

Instrumentace bajtkódu

Jedná se o instrumentaci již přeloženého zdrojového programu. Pro instrumentování bajtkódu je potřeba pouze do vygenerovaného .pyc nebo .pyo souboru, vytvořeného při překladu, vložit potřebné další instrukce. Na úrovni bajtkódu je zaručeno, že každá jednotlivá instrukce je atomická, což zajišťuje globální zámek interpretu. Nevýhodou instrumentace na úrovni bajtkódu je, že se bajtkód může mezi jednotlivými verzemi Pythonu lišit.

Příklad, jak vypadá instrumentace na úrovni bajtkódu lze vidět na ilustraci 5.3, kde v prvním sloupci jsou čísla řádků odpovídající zdrojovým kódům. Tento bajtkód přesně odpovídá kódu z ilustrace 5.2 s tím, že se jeví, jako by vše bylo na jednom řádku - řádku 2. Každé volání jednotlivých metod obsahuje v tomto případě právě 3 instrukce bajtkódu a to LOAD_GLOBAL, což načte na zásobník funkci, která bude následně volána instrukcí CALL_FUNCTION. Instrukce POP_TOP potom zahodí návratovou hodnotu volané funkce.

```
1         ...předchozí instrukce
2         0 LOAD_GLOBAL              0 (before_interesting_event)
          3 CALL_FUNCTION            0
          6 POP_TOP

          7 LOAD_GLOBAL              1 (interesting_event)
         10 CALL_FUNCTION            0
         13 POP_TOP

         14 LOAD_GLOBAL              2 (after_interesting_event)
         17 CALL_FUNCTION            0
         20 POP_TOP
3         ...následující instrukce
```

Ilustrace 5.3: Příklad instrumentace na úrovni bajtkódu

Instrumentace interpretu

Tento přístup by byl nejnáročnějším z výše uvedených a pravděpodobně nejméně přenositelný. K provedení této instrumentace na pro vícevláknové testování důležitých místech by bylo potřeba nastudování a detailní znalosti fungování interpretu Pythonu, který je implementován v jazyce C.

Z výše uvedených možností byla zvolena instrumentace na úrovni bajtkódu, jelikož zdrojové kódy zanechává beze změny a probíhá na nižší úrovni, čímž je možné instrumentovat i například mezi čtením a zápisem při přiřazení hodnoty jedné proměnné k proměnné druhé, což na úrovni zdrojového kódu možné není. Značnou výhodou instrumentace bajtkódu je také zaručení atomicity každé jednotlivé instrukce.

Instrumentace probíhá v okolí míst programu, které jsou z pohledu souběžného běhu více vláken zajímavé a je díky ní umožněno vkládání šumu, sledování běhu programu pro sběr metrik pokrytí nebo případně provádění dalších funkcí, reagujících na výskyt některé z vícevláknových událostí.

6 Implementace

Celý nástroj pro testování vícevláknových programů je implementován v jazyce Python 2.7, což je v současné době jediná verze, kterou implementovaný nástroj pro testování vícevláknových programů podporuje. Z důvodů určitých změn v bajtkódu a změn v modulech standardní knihovny mezi různými verzemi jazyka, není momentálně nástroj schopen fungovat ve verzi 3. Při implementaci tohoto nástroje ovšem bylo usilováno o co největší možnou kompatibilitu syntaxe mezi Pythonem verze 2 a 3, což do budoucna ulehčí případné rozšíření podpory pro novější verze jazyka.

Ze začátku této kapitoly jsou popsány použité nástroje a technologie, které umožnily vznik implementovaného nástroje pro testování vícevláknových programů. Poté následuje obecný popis implementace architektury tohoto vytvářeného nástroje, na který později navazuje detailnější popis zajímavých částí kódu.

6.1 Použitá knihovna pro instrumentaci

Program má pouze jednu závislost na externí modul, který není obsažen ve standardní knihovně Pythonu, a to na modul byteplay [7], který zjednodušuje práci s bajtkódem pro jeho instrumentaci. Jeho popis se nachází níže.

byteplay

Modul byteplay převádí bajtkód Pythonu na ekvivalentní objekt kódu - `Code`, který je pro člověka dobře čitelný a lze jednoduše upravovat. Jednotlivé instrukce bajtkódu jsou reprezentovány jako dvojice, kdy prvním členem dvojice je konstanta reprezentující samotnou instrukci a druhým členem je její argument. Tyto dvojice jsou potom uloženy v seznamu.

Pro názornost je zde uveden příklad použití tohoto modulu:

```
1  >>> def f(a, b):
2  ...     print (a, b)
3  ...
4  >>> f(3, 5)
5  (3, 5)
```

Ilustrace 6.1: Příklad jednoduché funkce.[7]

V ilustraci 6.1 se na řádcích 1 a 2 nachází definice funkce `f()`, která tiskne své zadané argumenty. Na řádku 4 je potom volání této funkce a na řádku 5 se nachází to co funkce `f()` vytiskla. Ukázka je pořízena z integrovaného vývojového prostředí Pythonu, dále jen IDLE.

V ilustraci 6.2, která je také pořízena z IDLE, je potom předvedeno použití modulu byteplay. Na prvních dvou řádcích se nachází importování knihoven. Na řádku 3 je bajtkód funkce `f()` z ukázky 6.1 převeden do byteplay `Code` objektu, ve kterém jsou jednotlivé instrukce bajtkódu reprezentovány jako seznam. Na řádku 4 je potom příkaz pro tisk a od řádku 5 lze vidět jednotlivé instrukce bajtkódu uložené v seznamu.

```
1  >>> from byteplay import *
2  >>> from pprint import pprint
3  >>> c = Code.from_code(f.func_code)
4  >>> pprint(c.code)
5  [(SetLineno, 2),
    (LOAD_FAST, 'a'),
    (LOAD_FAST, 'b'),
    (BUILD_TUPLE, 2),
    (PRINT_ITEM, None),
    (PRINT_NEWLINE, None),
    (LOAD_CONST, None),
    (RETURN_VALUE, None)]
```

Ilustrace 6.2: Reprezentace bajtkódu v bytelay [7]

Byteplay je publikována pod open source licencí Lesser General Public Licence.

6.2 Použití nástroje

Nástroj pro testování vícevláknových programů je implementován primárně jako konzolová aplikace. Konfigurace tohoto nástroje je realizována pomocí argumentů a přepínačů na příkazové řádce. Touto konfigurací je umožněno specifikovat chování programu a zvolit si tak například, které vybrané soubory se budou instrumentovat nebo vlastnosti vkládaného šumu. Všechny možnosti jsou popsány dále v této podkapitole.

Při spuštění nástroje je vždy nutné zvolit jeden z příkazů, které určují nástrojem prováděnou akci. Je možno volit z následujících příkazů.

- **run** – Tento příkaz provede (opakované) spouštění zadaného programu.
- **instrument** – Provede instrumentaci vybraných souborů
- **uninstrument** – Odinstrumentuje vše, co bylo instrumentováno příkazem *instrument*
- **report** – Vytiskne na standardní výstup výsledky měření metriky pokrytí pro každý soubor
- **htmlreport** – Zapíše do html souborů výsledky měření metriky pokrytí

Přesné chování jednotlivých příkazů je poté možno upřesnit pomocí následujících přepínačů.

- **-n –** Určuje při spuštění s příkazem *run*, kolikrát se bude program opakovaně spouštět
- **-d, --dir –** Specifikuje složku (a její podsložky), ve které se bude zvolený příkaz provádět
- **-m, --maxlevels –** Určuje míru zanoření do podsložek
- **--include –** Provede operaci pouze nad soubory specifikovanými pomocí unix wildcards
- **--exclude –** Vynechá takto specifikované soubory, také užívá unix wildcards
- **-p, --probability –** Určení s jakou pravděpodobností se bude generovat šum
- **--mintime –** Minimální délka uspání vlákna (šumu)
- **--maxtime –** Maximální délka uspání vlákna (šumu)
- **--disable-noise** – Do testovaného programu nebude vkládán šum
- **--disable-coverage** – Nebude měřena metrika pokrytí
- **-h, --help –** Zobrazení nápovědy

6.3 Rozdělení do modulů

Pro lepší přehlednost a oddělení jednotlivých součástí nástroje pro testování vícevláknových aplikací je tento nástroj rozdělen do několika modulů. V této podkapitole jsou pro přehled tyto moduly uvedeny a krátce popsány.

main.py

Jedná se o hlavní modul, který slouží jako vstupní bod programu. Jsou zde zpracovávány parametry příkazové řádky a vytvořena instance řídící třídy, která podle zadaných parametrů spouští běh programu.

instrumentation.py

V tomto modulu se nachází funkce a třídy, které se starají o instrumentaci bajtkódu. Tento proces podrobně popsán v kapitole 7.4.

event.py

Zde se nachází třída EventHook, která se stará o odchytávání vícevláknově zajímavých událostí, které nastávají, díky proběhlé instrumentaci testovaného programu, a jejich následné vyhodnocení a zpracování. K jednotlivým událostem je poté také možno registrovat vlastní callbacky a rozšířit tak nástroj pro testování vícevláknových programů o další funkčnost.

coverage.py

Tento modul obsahuje implementaci třídy pro sběr metriky pokrytí. A také třídy pro uchování získaných dat mezi jednotlivými spuštěními testovaného programu.

report.py

Modul sloužící pro generování výsledků metrik pokrytí.

htmlreport.py

Tento modul umožňuje zobrazit výsledky měření pokrytí v html souboru, kde je barevně vyznačeno, které části sledovaného programy byly testováním pokryty a které nikoli.

__init__.py

Modul obsahující funkce, které slouží jako rozhraní knihovny. Slouží hlavně pro možnost přidání dalších posluchačů, které naslouchají událostem, nebo lze využít ke změně konfigurace. Například nastavení parametrů generování šumu.

6.4 Implementace instrumentace

Instrumentace je nejdůležitější a zároveň implementačně nejnáročnější součástí vytvářeného nástroje. O instrumentaci jednotlivých souborů bajtkódu (soubory s příponou .pyc) se stará třída `FileInstrumenter`, ve které celý proces instrumentace probíhá v následujících krocích:

1. Otevření souboru
2. Uložení verze bajtkódu a časového razítka, nacházejících se na prvních 8 bajtech souboru
3. Deserializace binárních dat na Code objekt (objekt ve kterém Python uchovává informace o bajtkódu) pomocí nástroje byteplay.

4. Z načteného Code objectu se vytvoří byteplay Code objekt.
5. Provedení instrumentace nad byteplay Code objektem.
6. Převedení instrumentovaného byteplay Code objectu na Code object Pythonu
7. Serializace instrumentovaného bajtkódu na binární data
8. Společně s uchovanou verzí bajtkódu a časovým razítkem se data zapíšou zpět do souboru
9. Zavření souboru

Samotná instrumentace je poté prováděna tak, že se prochází bajtkódem po jednotlivých instrukcích, a pokud se narazí na instrukci nebo posloupnost instrukcí spojených s vícevláknově zajímavou událostí, jsou před a za tuto posloupnost instrukcí vloženy instrukce obsahující vyvolání události, které jsou dále zpracovávány třídou `EventHook`, která obsluhuje tyto významné události.

Instrumentací se vkládá také na začátek souboru bajtkódu testovaného programu importování knihovny nástroje pro testování a poté volání inicializační funkce, která připraví prostředí a vytvoří jednotlivé instance tříd, které se poté používají za běhu testovaného programu. Zároveň se již při provádění instrumentace zapisují zajímavá místa pro měření metriky pokrytí a jsou v této chvíli označena jako nenavštívená.

Pro názornost je v kódech 6.3, 6.4 a 6.5 ukázáno, jak vypadá bajtkód jednoduché funkce před a po instrumentaci. V kódu 6.3 stojí za všimnutí řádek 4, kde dochází k přístupu ke 2 globálním proměnným. V kódu 6.4 je potom tento 4. řádek z kódu 6.3 převeden do bajtkódu, kde jdou vidět, v kódu 6.4 na řádcích 1 a 2 instrukce LOAD_GLOBAL, což je pro nástroj pro testování vícevláknových programů zajímavá instrukce a proto je před a za těmito instrukcemi vloženo volání sledovaní událostí, jak lze vidět v kódu 6.5, kde se původní instrukce nachází na řádcích 13, 26, 33, 40. Zbytek je volání sledování událostí vstupu (řádky 1-6) a návratu (řádky 34-39) z funkce a sledování LOAD_GLOBAL instrukce (na zbývajících řádcích).

```
1  a = 2
2  b = 3
3  def fce():
4      return a + b
```

Kód 6.3: Jednoduchý příklad funkce

```
1   4           0 LOAD_GLOBAL              0 (a)
2               3 LOAD_GLOBAL              1 (b)
3               6 BINARY_ADD
4               7 RETURN_VALUE
```

Kód 6.4: Bajtkód před instrumentací

```
 1   4           0 LOAD_NAME                0 (concurnoise)
 2               3 LOAD_ATTR                1 (_eventhook)
 3               6 LOAD_ATTR                2 (method_entry_event)
 4               9 LOAD_CONST               1 (('/path/example.py', 'fce', 4, 0))
 5              12 CALL_FUNCTION            1
 6              15 POP_TOP
 7              16 LOAD_NAME                0 (concurnoise)
 8              19 LOAD_ATTR                1 (_eventhook)
 9              22 LOAD_ATTR                3 (before_load_global_event)
10              25 LOAD_CONST               2 (('/path/example.py', 'fce', 4, 1))
11              28 CALL_FUNCTION            1
12              31 POP_TOP
13              32 LOAD_GLOBAL              4 (a)
14              35 LOAD_NAME                0 (concurnoise)
15              38 LOAD_ATTR                1 (_eventhook)
16              41 LOAD_ATTR                5 (after_load_global_event)
17              44 LOAD_CONST               2 (('/path/example.py', 'fce', 4, 1))
18              47 CALL_FUNCTION            1
19              50 POP_TOP
20              51 LOAD_NAME                0 (concurnoise)
21              54 LOAD_ATTR                1 (_eventhook)
22              57 LOAD_ATTR                3 (before_load_global_event)
23              60 LOAD_CONST               3 (('/path/example.py', 'fce', 4, 2))
24              63 CALL_FUNCTION            1
25              66 POP_TOP
26              67 LOAD_GLOBAL              6 (b)
27              70 LOAD_NAME                0 (concurnoise)
28              73 LOAD_ATTR                1 (_eventhook)
29              76 LOAD_ATTR                5 (after_load_global_event)
30              79 LOAD_CONST               3 (('/path/example.py', 'fce', 4, 2))
31              82 CALL_FUNCTION            1
32              85 POP_TOP
33              86 BINARY_ADD
34              87 LOAD_NAME                0 (concurnoise)
35              90 LOAD_ATTR                1 (_eventhook)
36              93 LOAD_ATTR                7 (method_exit_event)
37              96 LOAD_CONST               4 (('/path/example.py', 'fce', 4, 4))
38              99 CALL_FUNCTION            1
39             102 POP_TOP
40             103 RETURN_VALUE
```

Kód 6.5: Bajtkód po instrumentaci

6.5 Reakce na události

O odchytávání a další delegování událostí se stará třída `EventHook`, která při výskytu některé z událostí volá postupně registrované posluchače, kteří se přihlásili k naslouchání výskytu dané události.

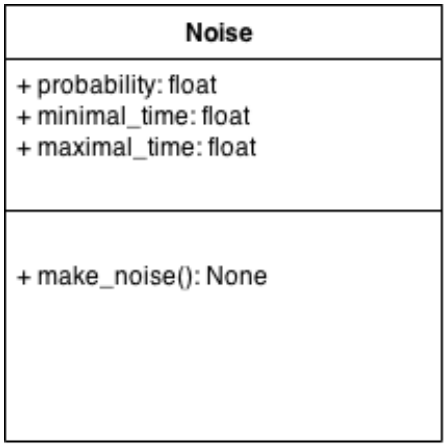
```
1  def  before_load_global_event(self, program_location):
2       for callback in self._listeners['before_load_global_event']:
3           callback()
```

Kód 6.6: Ukázka metody volající jednotlivé posluchače

V kódu 6.6 lze vidět názorně, jak je implementováno volání jednotlivých registrovaných posluchačů na vybranou událost ve třídě `EventHook`. Jednotlivé posluchače lze k událostem registrovat pomocí metody `add_listener()`.

6.6 Generování šumu

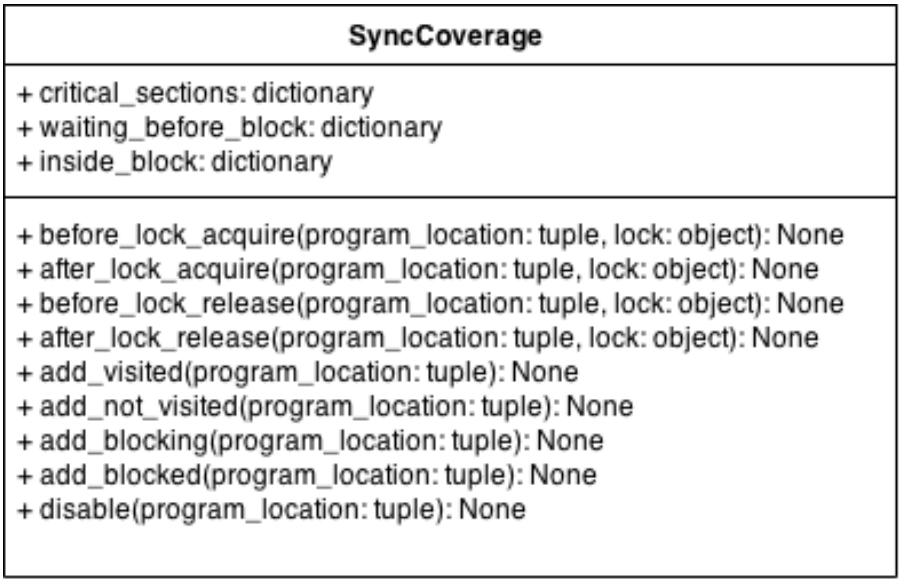
O samotné generování šumu se stará třída Noise, která má momentálně jedinou metodu `make_noise()`, která při zavolání uspí s pravděpodobností danou atributem `probability` vlákno na náhodně dlouhou dobu, která je dána rozsahem atributů `minimal_time` a `maximal_time`.



Obrázek 6.1: Třída Noise

6.7 Měření metriky pokrytí

Měření metriky pokrytí je implementováno ve třídě `SyncCoverage`. Tato třída si ukládá informace o navštívených synchronizačních primitivech a jejich stavech v atributu `critical_sections`, jak lze vyčíst z obrázku 6.2.



Obrázek 6.2: Třída SyncCoverage

Pro účely měření pokrytí synchronizace je tedy potřeba sledovat a ukládat si v jakém stavu byly jednotlivé zámky, když k nim přistoupilo některé z vláken. Z toho je potom možno určit, zda je zámek v dané chvíli blokovaný nebo naopak blokující. Aby bylo možno měřit tuto metriku pokrytí, je potřeba sledovat, kdy nastanou následující události.

- **before_lock_acquire(program_location, lock)** Při této události je zámek označen jako *navštívený* a je poznamenáno, že se vlákno nachází před zamykáním zámku.

- **after_lock_acquire(program_location, lock)** Při této události je odstraněna informace o tom, že se vlákno nachází před zámkem a je zaznamenána informace, že právě drží daný zámek.

- **before_lock_release(program_location, lock)** U této události je kontrolováno, jestli nečeká nějaké vlákno před zámkem. Pokud ano, je poznamenán stav *blokující* pro místo, kudy vlákno, které ještě momentálně zámek drží, vstoupilo a *blokovaný* pro místa, kde někdo čeká, než bude zámek odemčen.

- **after_lock_release(program_location, lock)** Sledování této události slouží pouze pro vnitřní synchronizaci, použitou pro správné fungování měření pokrytí.

Aby měření pokrytí synchronizace fungovalo správně, je potřeba zajištění vnitřní synchronizace. Bez této synchronizace by například hrozilo, že by se do seznamu zamčených zámků dostal již v dané chvíli neblokovaný zámek, a docházelo by tak k nesprávným výsledkům měření pokrytí. Tato synchronizace je realizována jedním zámkem, který do synchronizačního bloku uzavírá zpracování události `before_lock_acquire()`, dále v dalším synchronizačním bloku uzavírá zpracování události `after_lock_acquire()` a poté se zamyká na začátku zpracování události `before_lock_release()` a odemyká po skončení `after_lock_release()` události.

Pro zobrazení míry pokrytí měřené touto metrikou pokrytí lze využít třídu `Reporter` nebo `HtmlReporter`, které zobrazí report výsledků měření pokrytí na standardní výstup terminálu nebo do HTML souboru.

7 Experimenty

Pro ověření, zda vytvořený nástroj funguje správně, byla vytvořena sada 10 různých vícevláknových programů, které obsahují různé typy chyb, vyskytujících se ve vícevláknových programech. V této kapitole jsou jednotlivé programy nejprve popsány a poté následuje srovnání četnosti projevů chyb, obsažených v těchto programech při spuštění bez použití implementovaného nástroje pro testování vícevláknových programů a při spuštění s použitím tohoto nástroje.

7.1 Popis testovací sady o 10 programech

Experimenty probíhaly na programech, popsaných dále v této kapitole. Pro rychlý přehled těchto programů slouží tabulka 7.0, ve které lze v prvním sloupci vidět název programu, ve druhém sloupci je zmíněn počet řádků kódů a ve třetím sloupci se nachází informace, o kterou chybu se v programu jedná. Dále jsou poté jednotlivé programy této sady rozebrány podrobněji.

název	loc	typ chyby
auction.py	43	data race
bank.py	55	data race
counter.py	27	data race
deadlock.py	35	deadlock
deadlock2.py	32	deadlock
mergesort.py	83	order violation
orderviol.py	31	order violation
philosophers.py	59	deadlock
primenumbers.py	40	data race
produceconsum.py	52	data race

Tabulka 7.0: Přehled případů

auction.py

Jedná se o jednoduchou simulaci blížícího se konce aukce, ve které 10 vláken představuje 10 přihazujících, kteří se aukci snaží v posledních chvílích vyhrát. Každý přihazující opakovaně navyšuje svůj příhoz, poté krátkou dobu čeká a poté znovu přihazuje. V celém programu není nijak řešena synchronizace vláken.

```
1  def place_bid(self, bidder, amount):
2      if amount > self.highest_bid: # data race
3          self.bids.append((bidder, amount))
4          self.highest_bid = amount
```

Kód 7.1: Funkce place_bid obsahující chybu

Funkce, která je vidět v ukázce kódu 7.1 může kdykoliv volat kterékoli z vláken. Tím může docházet k časově závislé chybě nad daty, kdy se přistupuje k proměnné `self.highest_bid` jednou s operací čtení, jednou s operací zápisu. Pro ověření, zda došlo k chybě, se po doběhnutí programu kontroluje, jestli je seznam jednotlivých příhozů seřazen od nejnižší hodnoty po největší.

bank.py

Jedná se o jednoduchou simulaci banky, kdy 100 vláken současně provádí převody mezi účty banky. Opět jako u předchozí příkladu zde není nijak řešena synchronizace vláken.

```
1   def transfer(self, name, afrom, ato, amount):
2       if self.accounts[afrom] < amount: return
3       self.accounts[afrom] -= amount # data race
4       self.accounts[ato] += amount # data race
```

Kód 7.2: Funkce execute_deposit obsahující chybu

V ukázce 7.2 je vidět opět vznik časově závislých chyb nad daty, jelikož přičítání nebo odečítání peněz z účtu může nad tímto účtem být prováděno více vlákny zároveň.

counter.py

V tomto příkladu je využito dvou vláken, kdy v jednom vlákno provádí v cyklu opakované zvýšení hodnoty globálního čítače o 1, ve druhém vlákně probíhá odečítání o 1 od globálního čítače. Každé vlákno provádí stejný počet cyklů a proto je očekávaná výsledná hodnota 0. Přístup ke globálnímu čítači ovšem není nijak synchronizován a proto dochází k chybám.

```
   Vlákno 1                    Vlákno 2
1  for i in range(COUNT):      for i in range(COUNT):
2      counter += 1                counter -= 1
```

Kód 7.3: Ukázka sdíleného čítače

V kódu 7.3 jde na řádku s inkrementací (respektive dekrementací) vidět běžný příklad časově závislé chyby nad daty.

deadlock.py

Jedná se o krátký program, kdy každé z vláken zamyká zámky v opačném pořadí.

```
   Vlákno 1                          Vlákno 2
1  lock2.acquire()                   lock1.acquire()
2  for _ in range(100): pass         for _ in range(100): pass
3  lock1.acquire()                   lock2.acquire()
4  for _ in range(100): pass         for _ in range(100): pass
5  lock1.release()                   lock2.release()
6  lock2.release()                   lock1.release()
```

Kód 7.4: Příklad uváznutí

Jak lze vidět z ukázky kódu 7.4, jedná se o klasický příklad uváznutí dvou vláken.

deadlock2.py

V tomto programu se opět jedná o uváznutí, ale na rozdíl od předchozího případu z programu `deadlock.py`, je zde použito 3 zámků, jejichž uzamykání je ve všech spouštěných vláknech prováděno ve stejném pořadí (v tomto případě ve 2 vláknech).

mergesort.py

V tomto krátkém programu je implementován algoritmus merge sort, který rozděluje práci řazení jednotlivých dílů do vláken. Opět zde není využito žádného synchronizačního mechanismu vláken, a proto se občas stane, že program se snaží vybírat prvek z prázdného pole, což způsobí vyvolání výjimky a pád programu.

orderviol.py

Jedná se o krátký ukázkový příklad nesprávného pořadí vykonávaných operací, kdy se v jenom vlákně soubor otevírá, ve druhém vlákně se do něj zapisuje a třetí vlákno má za úkol tento soubor zavřít. Není ale žádným synchronizačním mechanismem zaručeno pořadí provedení jednotlivých vláken, přestože spuštění vláken je provedeno ve správném pořadí.

philosophers.py

V tomto programu se jedná o známý synchronizační problém obědvajících filosofů. Dochází zde k uváznutí, kdy si filozofové cyklicky zaberou každý jednu vidličku.

primenumbers.py

Tento program zjišťuje, zda je zadané číslo prvočíslo. Umožňuje také zjišťování více prvočísel najednou, kde pro zjištění každého prvočísla využívá nového vlákna. Zkoumaná čísla se ukládají do sdíleného asociativního pole, kde klíčem je zkoumané číslo a hodnotou je `False`. Při zjištění prováděnými výpočty, že zkoumané číslo je prvočíslo, je hodnota `False` přepsána na `True`. Tím, že je toto prováděno nad sdíleným asociativním polem, může docházet k nesprávným výsledkům z důvodu časově závislé chyby nad daty.

produceconsum.py

Jedná se o příklad synchronizačního problému producenta a konzumenta, kdy není využito žádného synchronizačního prostředku. V tomto příkladu je jeden producent a dva konzumenti a sdílený zdroj představuje obyčejný seznam.

```
Konzument:
1  if queue:
2       num = queue.pop(0)
```

Kód 7.5: Ukázka konzumenta s chybou

Jelikož není úsek v kódu, který je uveden v ukázce 7.5 nijak synchronizován, může docházet k chybnému přístupu do této fronty implementované seznamem.

7.2 Výsledky experimentů

Funkčnost nástroje pro testování vícevláknových programů byla ověřována na jednotlivých příkladech popsaných v předchozí podkapitole. Provádění všech měření experimentů probíhalo na počítači s dvoujádrovým procesorem Intel Core i3-380UM (3M Cache, 1.33GHz). Jednotlivé experimenty byly spouštěny na operačním systému Fedora 21 a Pythonu ve verzi 2.7.8. Pravděpodobnost vložení šumu, při vyvolání některé ze sledovaných vícevláknových událostí, byla u všech měření nastavena na 5%. Délka uspání vlákna způsobená šumem byla vždy v rozmezí 0-5 milisekund.

Experimenty byly zaměřeny na dopad nástroje pro testování na celkový výkon spouštěného programu a na účinnost nástroje v pomoci odhalování chyb typických ve vícevláknových programech. Výsledky jednotlivých měření lze vidět v tabulkách 7.2 a 7.3.

název	čas běhu [s]	směrodatná odchylka [s]	instrumentace	šum
auction.py	0.483	0.0352	1.29	3.01
bank.py	0.723	0.0309	3.43	12.51
counter.py	0.031	0.0030	3.05	5.73
deadlock.py	0.031	0.0033	2.79	3.00
deadlock2.py	0.032	0.0044	2.88	3.09
mergesort.py	0.079	0.0063	3.78	53.56
orderviol.py	0.037	0.0057	2.52	2.56
philosophers.py	0.051	0.0044	1.80	2.60
primenumbers.py	0.032	0.0049	2.70	3.32
produceconsum.py	0.210	0.0786	1.36	4.42

Tabulka 7.2: Měření dopadů na výkon

V tabulce 7.2 lze vidět srovnání, k jakému zpomalení běhu programu dochází při použití nástroje pro testování. V prvním sloupci této tabulky jsou jména jednotlivých experimentálních programů. Druhý sloupec, pojmenovaný *čas běhu* značí průměrnou dobu, jak dlouho běží jednotlivé programy bez použití nástroje pro testování. V dalším sloupci se pak nachází směrodatná odchylka k naměřené průměrné době běhu z druhého sloupce. Ve čtvrtém sloupci s názvem *instrumentace* se nachází údaj, který udává, ke kolikanásobnému zpomalení došlo při spouštění pouze instrumentovaného programu bez vkládání šumu. A v posledním sloupci s názvem *šum* je uvedeno, kolikrát se běh programu zpomalil při použití vkládání šumu. Pro měření byl každý jednotlivý program spouštěn 100 krát bez použití nástroje pro testování vícevláknových programů, 100 krát pouze instrumentován se zakázaným vkládáním šumu a 100 krát s vkládáním šumu.

název	**originál**	**instrumentace**	**šum**
auction.py	0.281	0.503 (1.79)	1 (35.56)
bank.py	0.033	0.424 (12.86)	1 (30.3)
counter.py	0.003	0.009 (3)	0.974 (324.6)
deadlock.py	0.006	0.001 (0.17)	0.111 (18.5)
deadlock2.py	0.001	0.005 (5)	0.251 (251)
mergesort.py	0.002	0.088 (44)	0.506 (253)
orderviol.py	0.139	0.204 (1.47)	0.486 (3.5)
philosophers.py	0.005	0.01 (2)	0.038 (7.6)
primenumbers.py	0.05	0.079 (1.58)	0.356 (7.12)
produceconsum.py	0.014	0.02 (1.43)	0.94 (67.14)

Tabulka 7.3: Četnost projevů chyb

Tabulka 7.3 srovnává četnosti projevů chyb s použitím nástroje a bez něj. V prvním sloupci této tabulky jsou názvy programu, v druhém sloupci jsou hodnoty četnosti projevu chyby při nepoužití nástroje pro testování vícevláknových programu. Uváděné číselné hodnoty jsou relativní, kdy hodnota 1 představuje projev chyby při každém spuštění, při číselné hodnotě rovné 0 k projevu chyby nedošlo. Ve třetím sloupci jsou pak výsledky měření při pouhé instrumentaci, bez vkládání šumu, kdy před závorkami je relativní hodnota četnosti chyby a v závorkách je porovnání, kolikrát se četnost zvýšila oproti nepoužití nástroje pro testování. A v posledním sloupci jsou vidět výsledky četností s použitím šumu. Kdy význam jednotlivých hodnot je stejný jako ve sloupci třetím. Pro toto měření byl každý program spouštěn 1000 krát bez využití nástroje pro testování, 1000 krát byl každý program spouštěn pouze s instrumentací a 1000 krát byl spouštěn s vkládáním šumu.

Z výsledků měření tedy vyplývá, že nástroj pro testování vícevláknových programů pomocí svého vkládání šumu mnohonásobně zvyšuje šanci projevu chyb. Při srovnání tohoto navýšení projevů četností chyb z tabulky 7.3 se snížením výkonnosti z tabulky 7.2 vyplývá, že přínos odhalení chyby několikanásobně převyšuje možné zpomalení běhu programu.

8 Závěr

Cílem této bakalářské práce bylo vytvořit nástroj pro testování vícevláknových programů vytvořených v jazyce Python, který bude pracovat s pomocí vkládání šumu do testovaného programu. Tento vytvořený nástroj potom slouží k výraznému zvýšení pravděpodobnosti odhalení chyb, které se běžně vyskytující ve vícevláknových programech. Dále nástroj umožňuje sběr metriky pokrytí synchronizace, díky čemuž je možno určit, jak moc je daný program otestovaný z hlediska možných stavů synchronizačních primitiv.

Nástroj se povedlo úspěšně implementovat, kdy vkládání šumu a sběru informací pro generování metrik pokrytí bylo docíleno pomocí instrumentace bajtkódu. Pro přehlednější zobrazení metriky pokrytí synchronizačních primitiv bylo navíc implementováno i generování tohoto reportu pokrytí do jednoduchého HTML souboru. Nástroj také umožňuje tvorbu dalších rozšíření zásuvnými moduly, které se mohou napojit na vybrané události, sledované díky instrumentaci, a nějakým způsobem na ně reagovat.

Ověření funkčnosti nástroje probíhalo na 10 různých vícevláknových programech, na kterých bylo ukázáno, že s pomocí generování šumu pravděpodobnost projevu chyby opravdu vzrůstá. Bohužel ale dochází k určitému zpomalení testovaného programu, což je ovšem vyváženo efektivnějším nalezením chyb. V jednom z proběhlých experimentálních případů bylo dokonce, při použití vytvořeného nástroje pro testování, naměřeno 324 násobného navýšení pravděpodobnosti projevu chyby, zatímco délka běhu se zvýšila jen 5,7 krát.

Do budoucna je v plánu rozšířit tento nástroj o podporu Pythonu od verze 3.3 nahoru. Také by bylo vhodné program rozšířit o sběr dalších metrik pokrytí vhodných pro vícevláknové zpracování. Dále by šlo program rozšířit o dynamický analyzátor, který by byl schopen odhalovat možné výskyty chyb, aniž by muselo dojít k jejich projevu při testování. S tím by následně mohla souviset i implementace vkládání šumu do programu s využitím nějaké lepší heuristiky, než pouze náhodného generování a vkládání.

Literatura

[1] SUMMERFIELD, Mark. *Python in practice: create better programs using concurrency, libraries and patterns*. Upper Saddle River, NJ: Addison Wesley, c2014, xiv, 306 s. Developer´s library. ISBN 978-0-321-90563-5.

[2] PYTHON SOFTWARE FOUNDATION. *History and License* [online]. [cit. 2015-03-20]. Dostupné z: https://docs.python.org/2/license.html

[3] PYTHON SOFTWARE FOUNDATION. *threading - Higher-level threading interface* [online]. [cit. 2015-03-20]. Dostupné z: https://docs.python.org/2/library/threading.html

[4] FIEDOR, Jan, et al. A uniform classification of common concurrency errors. In:*Computer Aided Systems Theory–EUROCAST 2011*. Springer Berlin Heidelberg, 2012. p. 519-526.

[5] FIEDOR, J., V. HRUBÁ, B. KŘENA, Z. LETKO, S. UR a T. VOJNAR. Advances in noise-based testing of concurrent software. *Software Testing, Verification and Reliability* [online]. 2015, vol. 25, issue 3, s. 272-309 [cit. 2015-03-20]. DOI: 10.1002/stvr.1546. Dostupné z: http://doi.wiley.com/10.1002/stvr.1546

[6] EDITED BY PETER M.A. SLOOT, Edited by Peter M.A.David Abramson. *Computational Science " ICCS 2003 International Conference, Melbourne, Australia and St. Petersburg, Russia, June 2 4, 2003 Proceedings, Part IV*. Springer-Verlag Berlin Heidelberg, 2003. ISBN 978-354-0448-648.

[7] Byteplay Module Documentation. [online]. [cit. 2015-03-20]. Dostupné z: https://wiki.python.org/moin/ByteplayDoc

[8] BEAZLEY, David M. *Python: essential reference*. 4th ed. Upper Saddle River: Addison-Wesley Professional, 2009, xxi, 717 s. developer's library. ISBN 978-0-672-32978-4.

[9] KŘENA Bohuslav, LETKO Zdeněk, NIR-BUCHBINDER Yarden, TZOREF-BRILL Rachel, UR Shmuel a VOJNAR Tomáš. *A Concurrency Testing Tool and its Plug-ins for Dynamic Analysis and Runtime Healing*. FIT-TR-2009-01, Brno, 2009.

[10] PILGRIM, Mark. *Dive into Python*. New York: Distributed to the Book trade in the United States by Springer-Verlag, c2004, xviii, 413 p. ISBN 15-905-9356-1.

[11] CARVER, Richard H. *Modern multithreading: implementing, testing, and debugging multithreaded Java and C /Pthreads/Win32 programs*. Hoboken: Wiley, 2006, xiv, 465 s. ISBN 978-0-471-72504-6.

[12] PYTHON SOFTWARE FOUNDATION.: *PythonImplementations* [online]. [cit. 2015-05-14]. Dostupné z: https://wiki.python.org/moin/PythonImplementations

[13] PYTHON SOFTWARE FOUNDATION.: *GlobalInterpreterLock* [online]. [cit. 2015-05-14]. Dostupné z: https://wiki.python.org/moin/GlobalInterpreterLock

[14] CHUN, Wesley a Wesley CHUN. *Core Python applications programming*. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2012, xxxii, 852 p. ISBN 978-013-2678-209.

[15] FROESE, Erik, KAPLAN, Antony, *Multicore Programming* [online]. [cit. 2015-05-14]. Dostupné z: http://www.cs.nyu.edu/~lerner/spring10/projects/Python_GIL.pdf

A. Obsah CD

Na CD se nachází následující soubory a adresáře

- **nastroj** – zdrojové soubory implementovaného nástroje pro testování
  - **benchmark –** sada 10 krátkých programů s chybami
  - **concurnoise –** zdrojové soubory
  - **setyp.py –** instalátor balíčku s nástrojem
  - **install.txt** – návod
- **zdroj –** adresář se zdrojovými soubory textové části
- **bp_text.pdf –** text této bakalářské práce v elektronické formě
- **poster.pdf -** plakát