Základy testování výkonnosti aplikací a hledání problematických míst

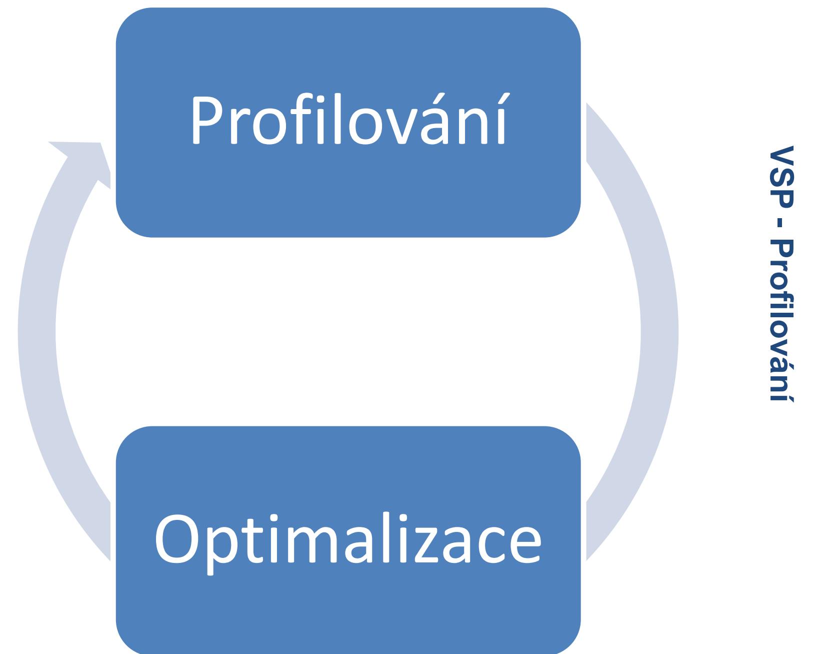
Richard Lipka

Cíle

- Optimalizace programu a detekce problémů

  Analýza běhu programu
  - Doba běhu jednotlivých metod
  - Četnost volání metod

  Analýza spotřeby paměti
  - Hledání memory leaků
  - Hledání velkých objektů

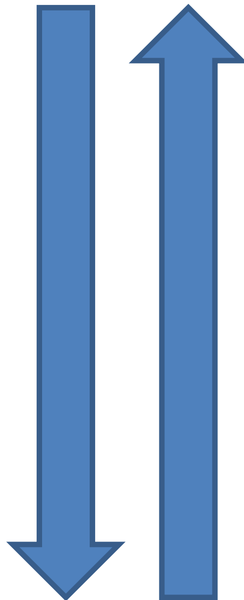


Profiling a Benchmarkování

- Není to totéž
  - Benchmarking – porovnání variant, blackbox
    → užitečné pro HW, cizí SW
    - Hodnotí stávající výkon a jeho závislost na zátěži
  - Profiling – analýza proč se program nějak chová, whitebox
    → užitečné pro vlastní SW
    - Hledá co spotřebovává zdroje
- Nutná izolace pokusů – za jakých podmínek?
  - Omezit další běžící aplikace
  - Pozor na bezpečnostní aplikace
- Nutné vhodně interpretovat výsledky

Instrumentace vs vzorkování

Instrumentace



Přesnost získaných dat

Vliv na program

Vzorkování

- Něco za něco
  - Ne vždy použitelné v reálném provozu
- Podpora specializovaných nástrojů

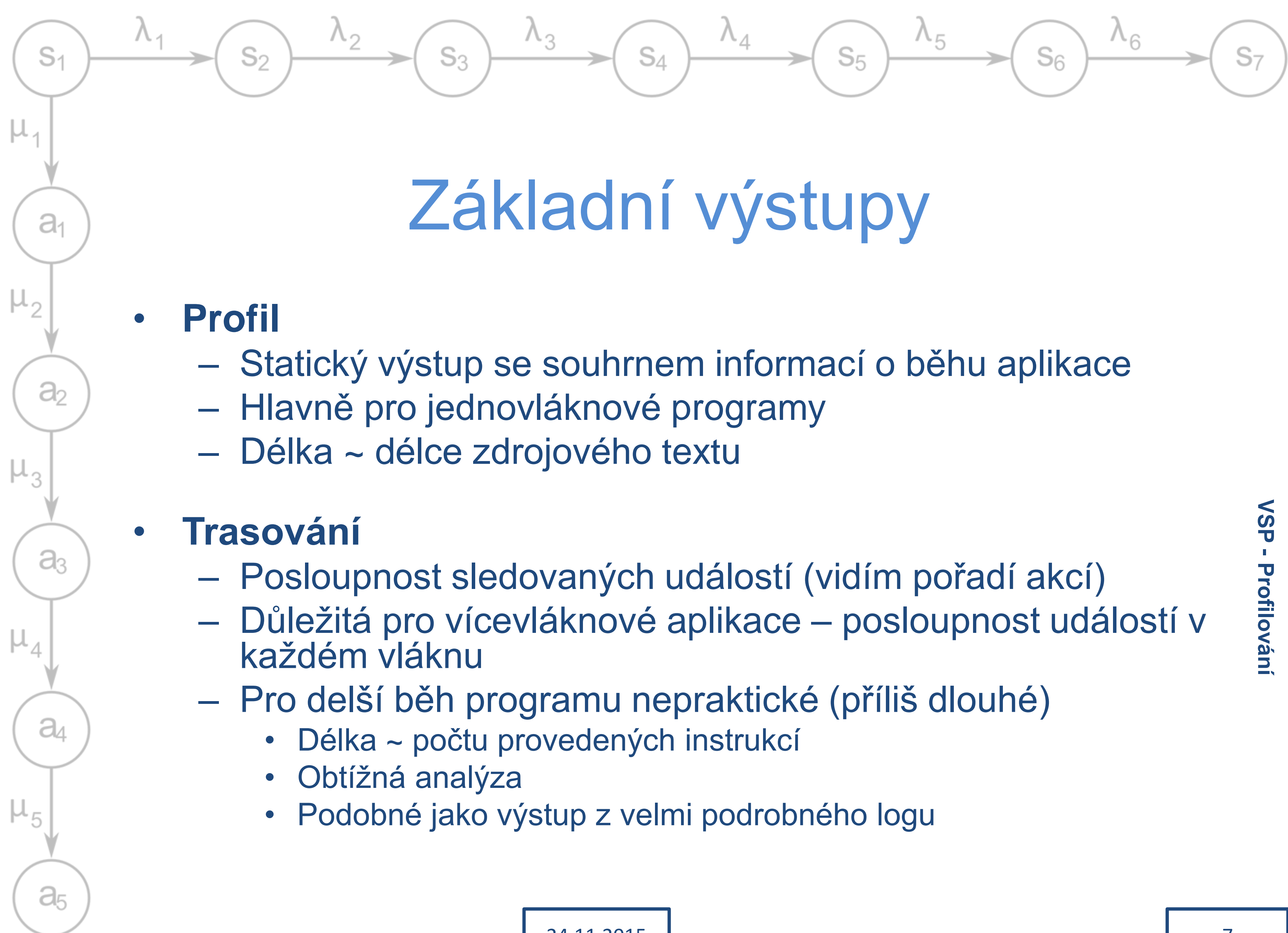
Instrumentace vs vzorkování

- Instrumentace
  - Před spuštěním programu
    - Binární soubory (= nepotřebuji zdrojové texty)
    - Zdrojové texty (podobné jako aserce)
- Vzorkování
  - Za běhu programu
    - Memory dump, stack trace, instrukce v CPU …
- Využití možností jazyka a systému
  - Přerušení, sledování událostí, speciální API
  - Snazší u interpretovaných jazyků

Možnosti profilingu

Profiling

- *CPU*
  - Čas
  - Volání
  - Vzorkování
- *Paměť*
  - Využití
  - Alokace



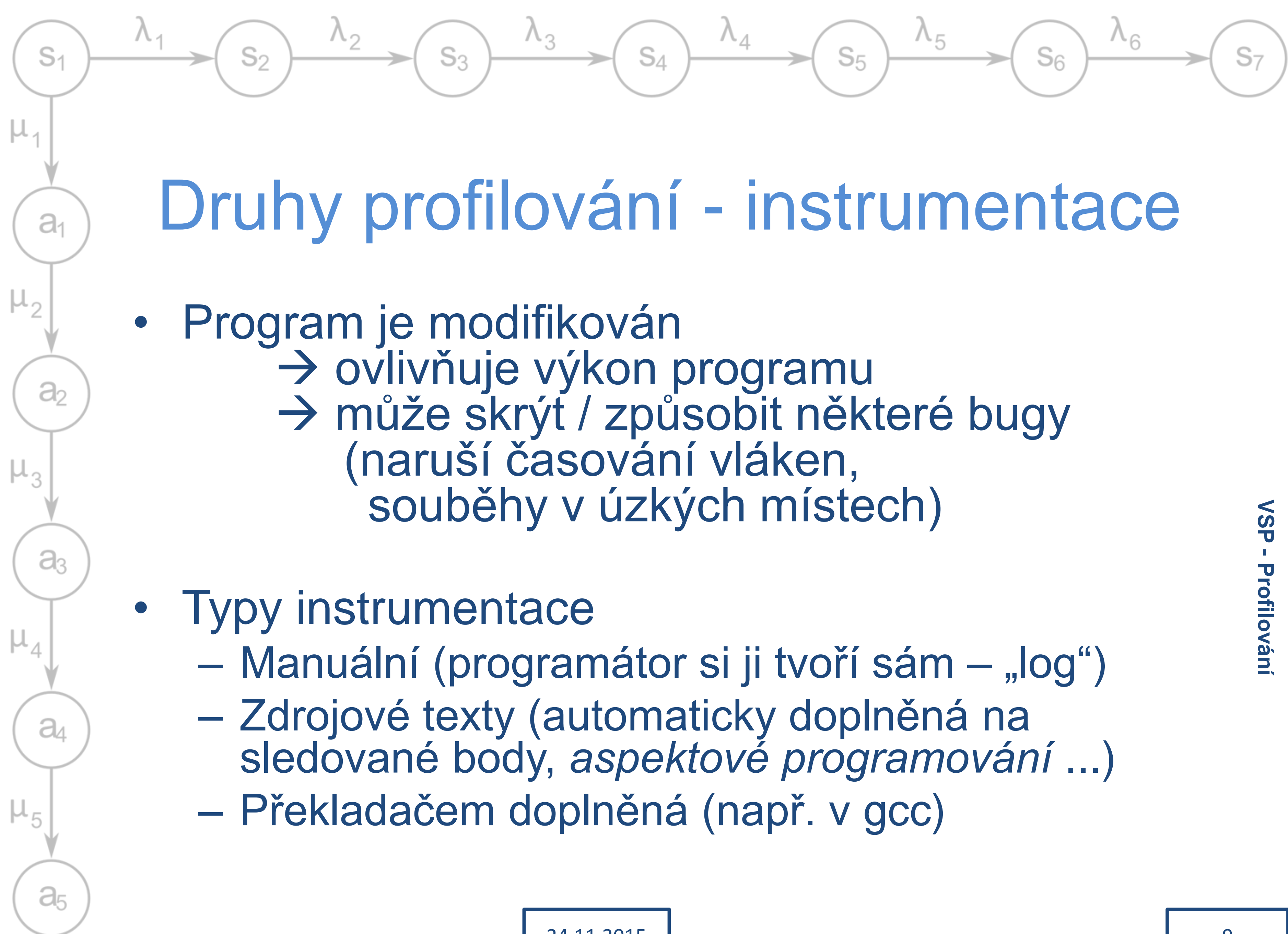
Základní výstupy

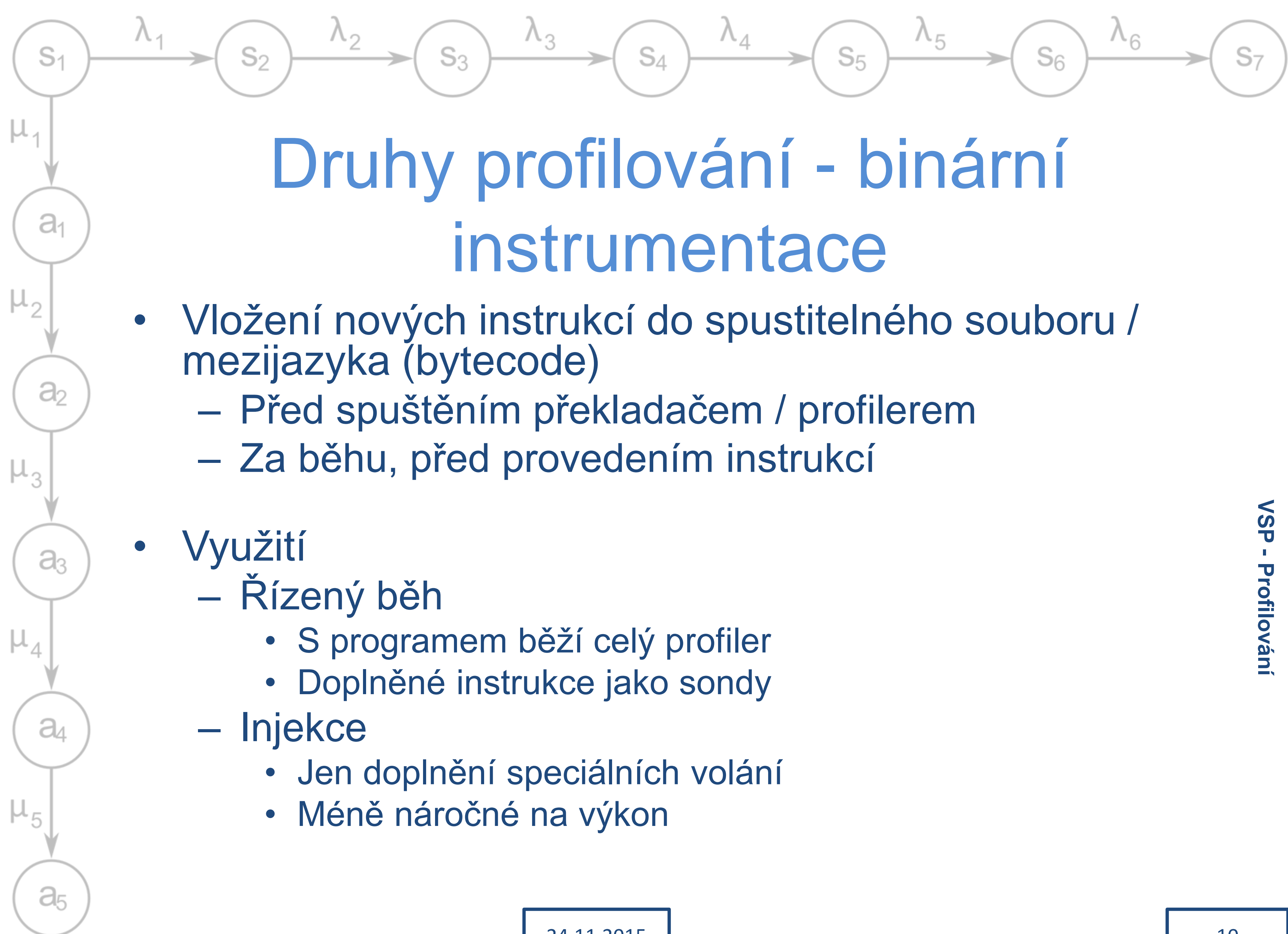
- **Profil**
  - Statický výstup se souhrnem informací o běhu aplikace
  - Hlavně pro jednovláknové programy
  - Délka ~ délce zdrojového textu
- **Trasování**
  - Posloupnost sledovaných událostí (vidím pořadí akcí)
  - Důležitá pro vícevláknové aplikace – posloupnost událostí v každém vláknu
  - Pro delší běh programu nepraktické (příliš dlouhé)
    - Délka ~ počtu provedených instrukcí
    - Obtížná analýza
    - Podobné jako výstup z velmi podrobného logu

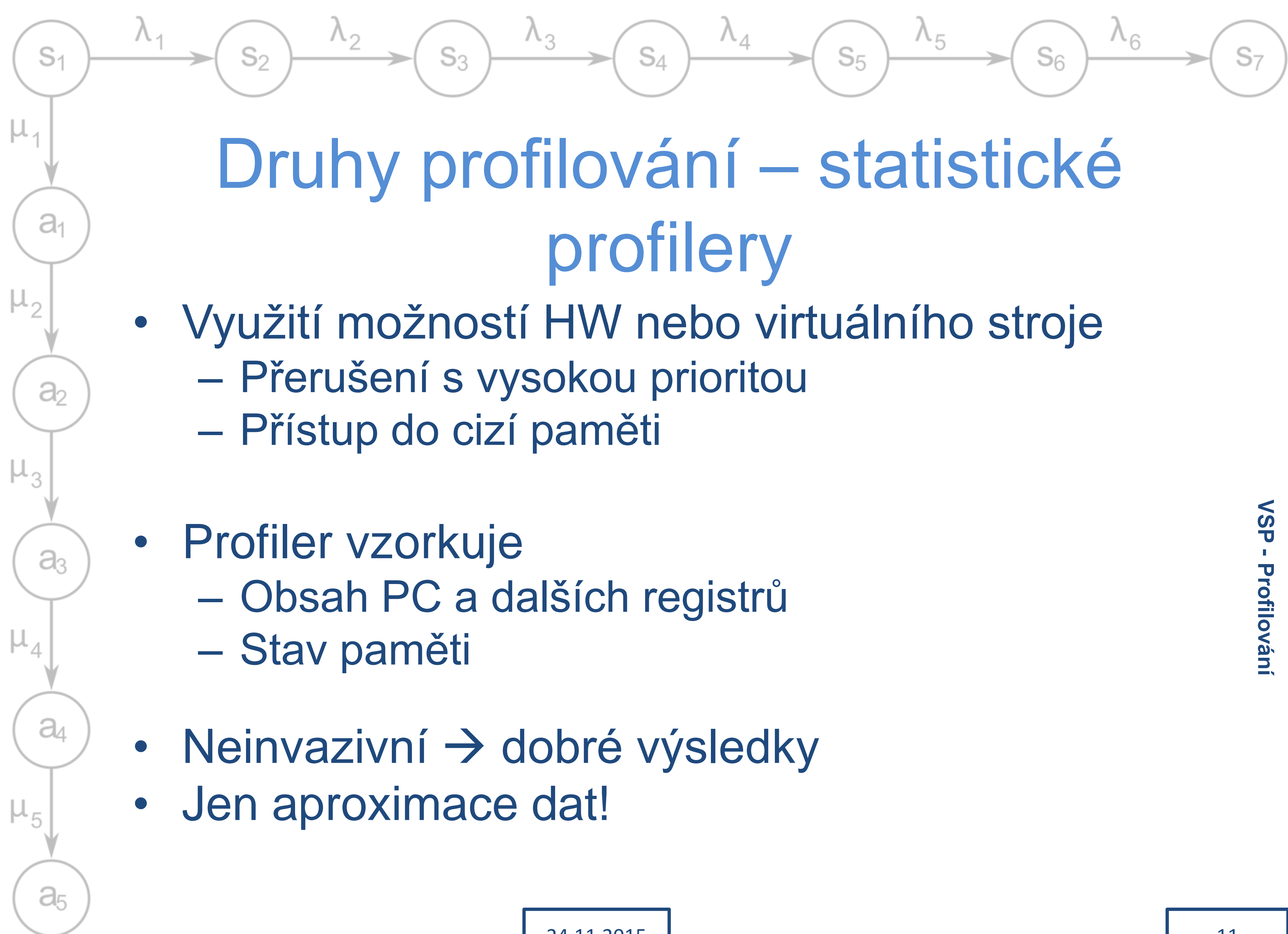
Základní výstupy

- **Paměťové statistiky**
  - Alokovaná paměť, běh GC
  - Struktury paměť využívají a jejich četnost
    - Sestavení grafu heapu – lze procházet
  - Data sbírána za běhu programu

- **Memory dump** (core dump, heap dump)
  - Aktuální stav adresního prostoru programu v daném okamžiku
  - Náročná analýza
    - Pomáhá mít tabulku symbolů – debugger


s1 s2 s3 s4 s5 s6 s7
λ1 λ2 λ3 λ4 λ5 λ6
a1 a2 a3 a4 a5
μ1 μ2 μ3 μ4 μ5
Druhy profilování - instrumentace
• Program je modifikován
→ ovlivňuje výkon programu
→ může skrýt / způsobit některé bugy
(naruší časování vláken,
souběhy v úzkých místech)
• Typy instrumentace
– Manuální (programátor si ji tvoří sám – „log“)
– Zdrojové texty (automaticky doplněná na
sledované body, aspektové programování ...)
– Překladačem doplněná (např. v gcc)
VSP - Profilování


Druhy profilování - binární instrumentace
• Vložení nových instrukcí do spustitelného souboru / mezijazyka (bytecode)
– Před spuštěním překladačem / profilerem
– Za běhu, před provedením instrukcí
• Využití
– Řízený běh
• S programem běží celý profiler
• Doplněné instrukce jako sondy
– Injekce
• Jen doplnění speciálních volání
• Méně náročné na výkon
VSP - Profilování
s1
s2
s3
s4
s5
s6
s7
λ1
λ2
λ3
λ4
λ5
λ6
μ1
μ2
μ3
μ4
μ5
a1
a2
a3
a4
a5

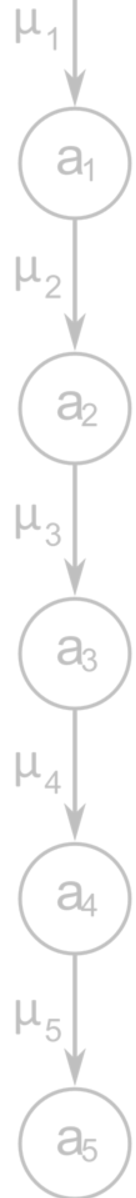

λ1 λ2 λ3 λ4 λ5 λ6
s1 s2 s3 s4 s5 s6 s7
μ1 μ2 μ3 μ4 μ5
a1 a2 a3 a4 a5
Druhy profilování – statistické profilery
• Využití možností HW nebo virtuálního stroje
– Přerušení s vysokou prioritou
– Přístup do cizí paměti
• Profiler vzorkuje
– Obsah PC a dalších registrů
– Stav paměti
• Neinvazivní → dobré výsledky
• Jen aproximace dat!
VSP - Profilování

Druhy profilování - události

- Pro interpretované jazyky
  - Podpora interpretu pro sledování událostí („hooky“)
  - Co vše lze zachytit záleží na API
  - Lze snadno tvořit vlastní profiler
- Funguje v
  - Java – JVM Tools Interface API
  - .NET – Profiling API
  - Python – modul hotshot
  - Ruby – modul profile.rb

Typické chyby

- Instrumentace ovlivní výsledek
  - Hlavně u vícevláknových aplikací
- Sledovaný program proběhne příliš rychle → nestihnu odebrat reprezentativní vzorky
- Profiling na nasazeném sw
  - Použití nestabilních / zdržujících funkcí
  - Porušení bezpečnosti

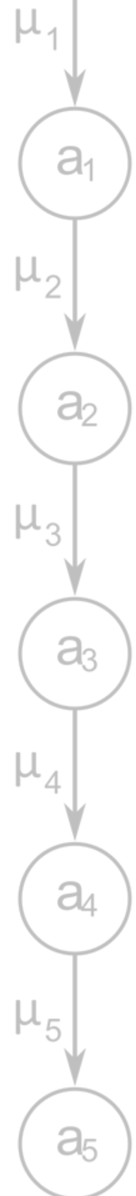


Profilování pro ladění

- Podobný problém jako benchmarkování
  - Izolace běhu experimentu
  - Stanovení cílů před začátkem
  - Soustředění se na důležití věci
    - Málo volaná metoda zabírá 10% času → max. 10% zrychlení (zhoršuje se s nárůstem režie volání)

Postup ladění

- Spustit měření (profiler / benchmark)
- Najít úzká místa, identifikovat ta důležitá
  - Odhadnout příčiny zpomalení
  - Navrhnout protidůvody – co kdyby byla úzká místa jinde
- Připravit izolovaný test k ověření
  - Na základě protidůvodu
- Provést test a změřit výsledky
- Upravit aplikaci
- Otestovat jestli úprava pomohla (+ standardní testy jestli není zanesena chyba)



Postup ladění – best practices

- Důsledně verzovat všechno
  - zdrojový text a všechny změny
  - testy
  - informace o výsledcích testů
  - nastavení pro testy (co je třeba volat, spustit …)
- Provádět unit testy s každou změnou
- Pozor na souběhové chyby

Uživatelé a výkon

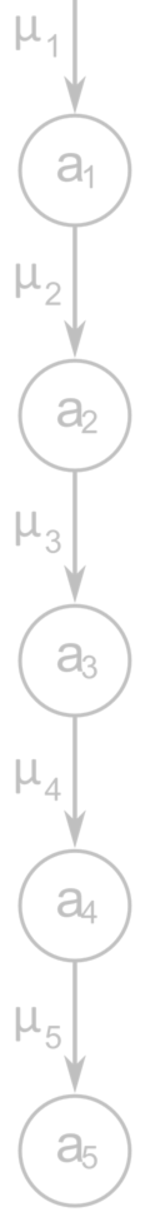
- Uživatelé málokdy vnímají skutečný výkon
  - Progresbar působí rychleji než žádné hlášení o stavu
    - Nesmí lhát! Pak ho uživatelé ignorují
  - Souběžná práce působí rychleji
    - Dlouhé úlohy ve vláknech na pozadí
  - Streamování a zpracování částečných dat působí rychleji
  - Caching může aplikaci doopravdy zrychlit

Profilování v Javě

- Interpretovaný jazyk
  - Lze se napojit na interpret – JVMTI
- Řada nástrojů součástí standardní Javy
  - Heap dump, sledování aplikací
- Velké množství volných i placených nástrojů

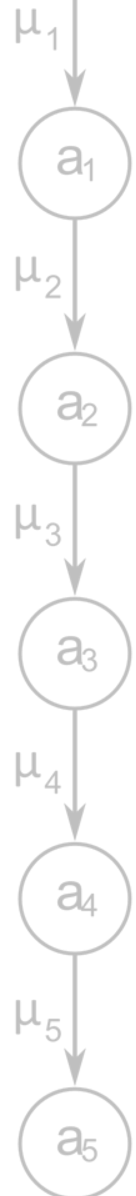
Vlastní profilování

- Využití znalostí JVM a bytecode
  - Nemusím se učit nic nového
  - Prohlubuji si znalost toho co denně používám
- Profilování uvnitř aplikace
  - Není problém s bezpečností
  - Existují podrobné logy
  - Nepotřebuji speciální nástroje nebo rozšíření JVM
  - Nutné nějak odstranit z produkční verze – zpomaluje běh



Vlastní profilování - možnosti

- Doplnění kódu do Java programů
  - Dobře prostudovat standardní třídy a metody – obvykle velmi stabilní
- Možnosti JVM
  - **`-X`** a **`-XX`** volby při spouštění programu
- Manipulace s bytecode
  - Nastudovat specifikaci - potenciálně nebezpečné



Vlastní profilování - knihovny

- Ručně vytvořené sledování
  - Třídy **`java.System, java.Runtime`**
- Funkce pro
  - Měření času
  - Sledování paměti
  - Práce s třídami a metodami
  - Volání jiných aplikací

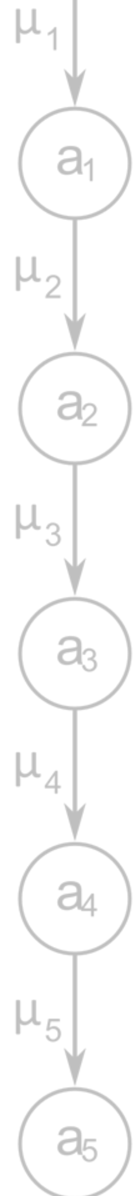
μ1 a1 μ2 a2 μ3 a3 μ4 a4 μ5 a5

Vlastní profilování - čas

- Měření času v daném okamžiku

  `currentTimeMillis()`
  - Aktuální čas v milisekundách (timestamp)
  - Granuralita závisí na četnosti časových přerušení (až 10 ms!)

  `nanoTime()`
  - Časová známka v nanosekundách
  - Nemusí jít o skutečný čas, bez synchronizace mezi vlákny nebo instancemi JVM!
  - Trvá dlouho – tvorba timerů

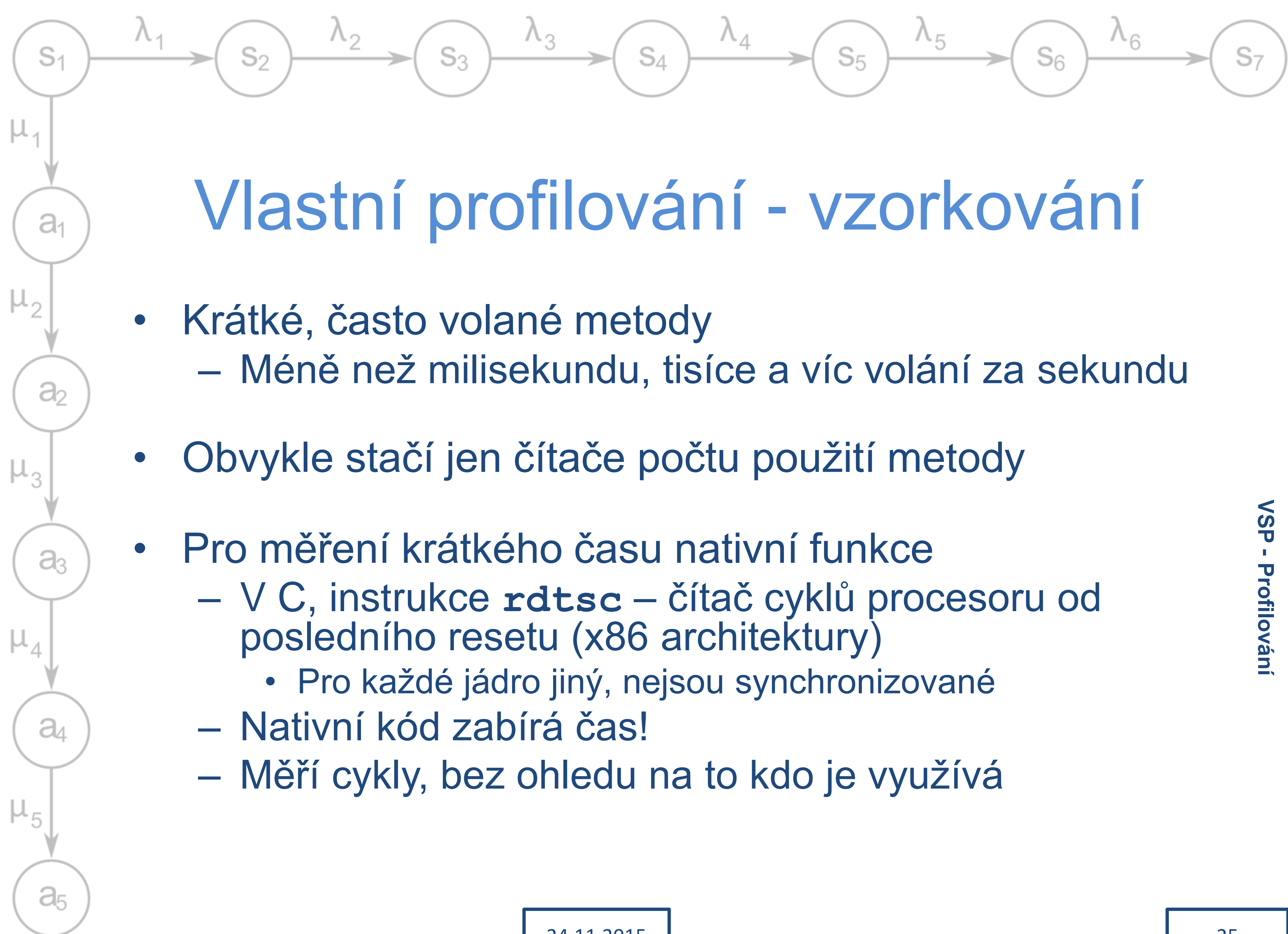


Vlastní profilování - čas

- Pozor na overhead
  - Funkce měřící čas potřebují čas
  - Zápisy do souborů potřebují čas
  - Měří skutečný čas, ne čas CPU
  - Pozor na souběh
    - **`nanoTime()`** není bezpečný mezi vlákny

Vlastní profilování - čas

- Výsledky ukládat do mapy
  - Alokace paměti v Javě rychlá (tvorba nových objektů pomalá – raději primitivní datové typy)
  - U vícevláknové aplikace pozor na souběh
- Poslouží i jako logger
  - Dobrá data pro hledání runtime chyb
  - Lze trasovat program
- Pokud je požadované vždy



Vlastní profilování - vzorkování

- Krátké, často volané metody
  - Méně než milisekundu, tisíce a víc volání za sekundu
- Obvykle stačí jen čítače počtu použití metody
- Pro měření krátkého času nativní funkce
  - V C, instrukce **`rdtsc`** – čítač cyklů procesoru od posledního resetu (x86 architektury)
    - Pro každé jádro jiný, nejsou synchronizované
  - Nativní kód zabírá čas!
  - Měří cykly, bez ohledu na to kdo je využívá

μ1 a1 μ2 a2 μ3 a3 μ4 a4 μ5 a5

Vlastní profilování - vzorkování

- Získání thread dumpu v náhodnou dobu
  - Pokud se nějaká metoda volá často, nejspíš budu v ní

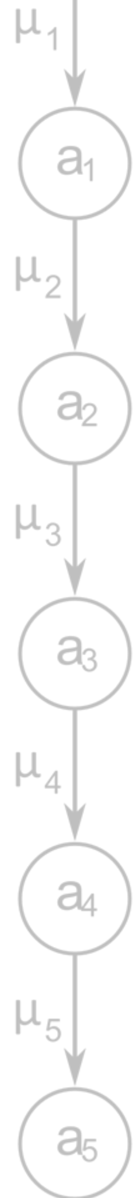
- Možnosti (v Javě):
  - Ukončení aplikace – „**`kill -3`**“ nebo **`Ctrl-Break`** – dostanu stackTrace
  - V konzoli „**`jstack <pid>`**“ (vhodné pro skriptování)
  - V programu **`Thread.getStackTrace()`** nebo **`Thread.getAllStackTraces()`**

μ1 a1 μ2 a2 μ3 a3 μ4 a4 μ5 a5

Vlastní profilování - paměť

- Před měřením garbage kolekce
  - **`gc()`** - jen návrh na spuštění, nelze vynutit

- Metody pro měření
  - **`freeMemory()`** – volná paměť JVM
  - **`totalMemory()`** – celková paměť obsazená JVM
  - **`maxMemory()`** – maximální paměť kterou JVM může chtít použít



Vlastní profilování - paměť

- Měření

```
Runtime r = Runtime.getRuntime();
gc();
long mem1 = r.freeMemory();
… /*measured code*/
gc();
long mem2 = r.freeMemory();
```

Vzorkování paměti

- Jmap + JHat
  - Konzolové aplikace
  - Získání memory dumpu a jeho analýza
  - K dispozici všude kde je JDK
- Použití

```
jmap -dump:file=my_stack.bin 4365
jhat my_stack.bin
```

  - `jmap` získá dump programu, `jhat` ho zobrazí
  - Výsledky na `http://localhost:7000`

Vzorkování paměti

- Jmap neprovádí garbage kolekci
  - Lze získat jen živé objekty
    `jmap -histo:live PID`
- Dump jen v bezpečných bodech
  - Není dokumentována možnost obejití
- Nejlépe při problémech
  - **`-XX:+PrintClassHistogram`**
    - Vypíše histogram po ukončení procesu
  - **`-XX:+HeapDumpOnOutOfMemoryError`**
    - Vypíše dump po výjimce

Alokace paměti

- Lze sledovat alokace
  - Parametr JVM `–Xaprof`
  - Po ukončení vypíše seznam alokovaných objektů a jejich počty
- Zjištění kde a kdo paměť alokuje obtížnější
  - Jediná možnost - instrumentace
    - Využití aspektů
    - Manipulace s bytecode

Změny bytecode

- Lze provádět za běhu → funguje i po změnách zdrojových textů
- Lze sledovat konkrétní volání
- Knihovna ObjectWeb ASM
  - Lze nastavit jakou instrukci hledat
  - Lze nastavit co za ní / před ní vložit

- Využití hooků v classloaderu
  - Balík **`java.lang.instrument`**
    - Instrumentace bytecode
    - Umí zasáhnout i do tříd Javy!

Sledování Garbage kolektoru
• Java umožňuje za běhu sledovat činnost garbage kolekce
– Parameter verbose:gc vypisuje činnost GC
– Další nastavení
• -XX:+PrintGCDetails – chod GC
• -XX:+PrintGCTimeStamps – doba spuštění GC
• -XX:+Print\ClassHistogramAfterFullGC (Before) – výpis histogramů
VSP - Profilování
24.11.2015
33

Sledování Garbage kolektoru

Základní výpis (jen **-XX:+PrintGC**)

kolektor stav před a po kolekci doba

```
[GC 279616K->143286K(1013632K), 0.5776077 secs]
[GC 422902K->287195K(1013632K), 0.6870628 secs]
MAP size: 3000000
[Full GC 379997K->337043K(1013632K), 0.8727914 secs]
[GC 616659K->337044K(1013632K), 0.0583699 secs]
MAP size: 1000000
[Full GC 409996K->159399K(1013632K), 0.5447442 secs]
```

Profiler v JRE - HotSpot

- Parametr **`–Xprof`**
  - Sumarizace běhu interpretovaných, přeložených a nativních metod
  - Ploché vzorkování první metody z vrcholu zásobníku
  - Poznám co je kompilované HotSpotem

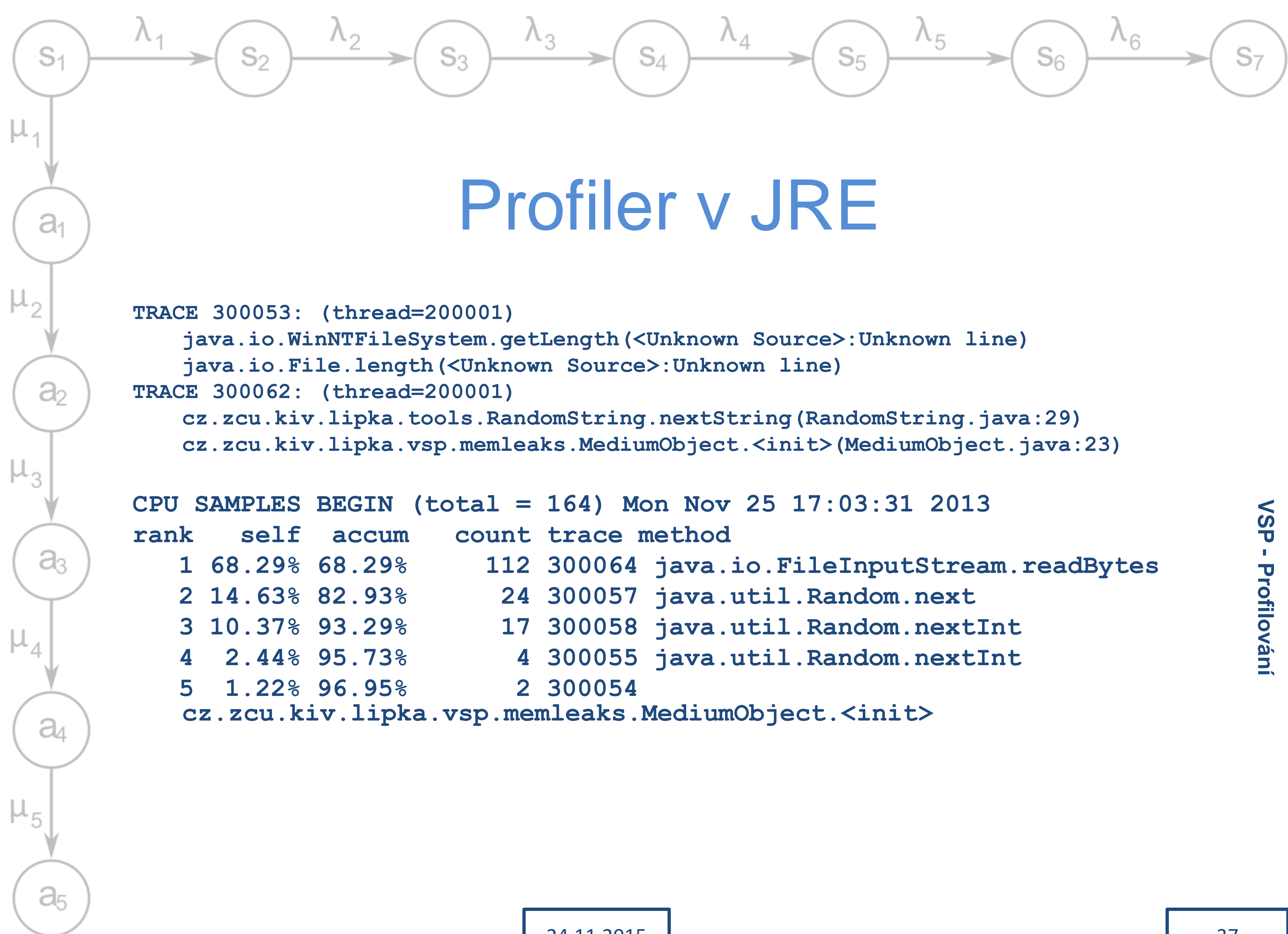
```
Interpreted + native   Method
 66.4%      0  +   101    java.io.FileInputStream.readBytes
  0.7%      1  +     0    cz.zcu.kiv.lipka.vsp.memleaks.MediumObject.<init>

Compiled + native   Method
 19.1%     29  +     0    java.util.Random.nextInt
 11.8%     18  +     0    java.util.Random.next
```

Profiler v JRE

- Parametr –**`Xrunhprof`**
  - Sleduje celý zásobník (podle nastavení hloubky - **`depth=5`**)
  - Vzorkování (nastavitelný interval - **`cpu=samples,interval=10`**) nebo měření všech volání (pomalejší – **`cpu=times`**)
  - Sledování běhu vláken (**`thread=y`**)

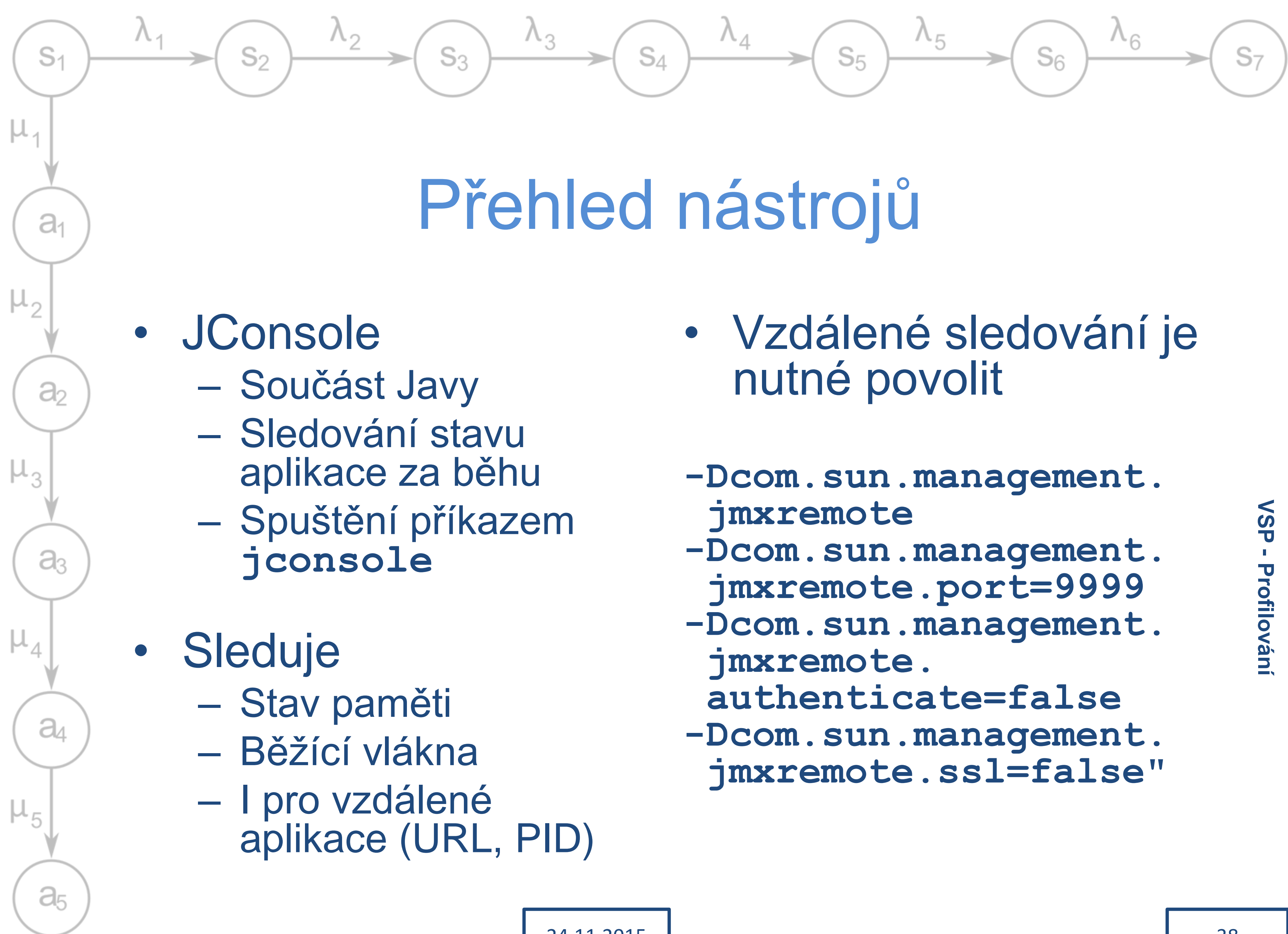
- Výstup do souboru, pro další analýzu



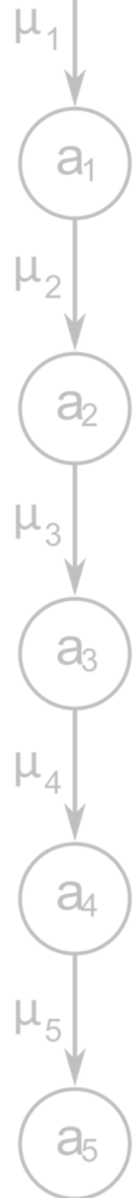
Profiler v JRE

```
TRACE 300053: (thread=200001)
    java.io.WinNTFileSystem.getLength(<Unknown Source>:Unknown line)
    java.io.File.length(<Unknown Source>:Unknown line)
TRACE 300062: (thread=200001)
    cz.zcu.kiv.lipka.tools.RandomString.nextString(RandomString.java:29)
    cz.zcu.kiv.lipka.vsp.memleaks.MediumObject.<init>(MediumObject.java:23)

CPU SAMPLES BEGIN (total = 164) Mon Nov 25 17:03:31 2013
rank   self  accum   count trace method
   1 68.29% 68.29%     112 300064 java.io.FileInputStream.readBytes
   2 14.63% 82.93%      24 300057 java.util.Random.next
   3 10.37% 93.29%      17 300058 java.util.Random.nextInt
   4  2.44% 95.73%       4 300055 java.util.Random.nextInt
   5  1.22% 96.95%       2 300054
    cz.zcu.kiv.lipka.vsp.memleaks.MediumObject.<init>
```



s1 λ1 s2 λ2 s3 λ3 s4 λ4 s5 λ5 s6 λ6 s7
μ1 a1 μ2 a2 μ3 a3 μ4 a4 μ5 a5
Přehled nástrojů
JConsole
Součást Javy
Sledování stavu aplikace za běhu
Spuštění příkazem jconsole
Sleduje
Stav paměti
Běžící vlákna
I pro vzdálené aplikace (URL, PID)
Vzdálené sledování je nutné povolit
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false"
VSP - Profilování



Přehled nástrojů

- Java misson control
  - Od Javy 8 (ve starších jen JConsole)
  - Podrobnější sledování běhu (jako JConsole)
  - **Flight recorder** – záznam běhu programu pro pozdější zpracování
    - Paměť, CPU, stav vláken, profilování běhu …

Přehled nástrojů

- SAP Memory analyzer
  - Na platformě Eclipse, samostatný
  - Analýza heap dumpu
    - Snaží se najít možné memory leaky
    - Hledá co drží objekty „naživu“

Přehled nástrojů

- VisualVM
  - Součást NetBeans, distribuován s JDK od Java7

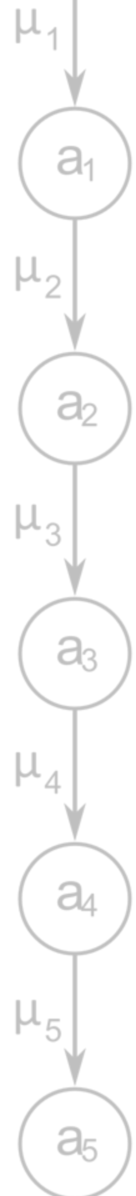
  → Může fungovat i samostatně
  - Umožňuje
    - Analyzovat paměť – spotřeba, heap dump
    - Analyzovat běh vláken
    - Sledovat běh programu

Přehled nástrojů - zdarma

Nástroj	Použití	Metody
Btrace	Sledování alokací paměti, stav vláken, doby odezvy	Instrumentace bytecode
Eureka JProfiler	Sledování stavu webové aplikace, spotřeby paměti, vyhozených výjimek	Instrumentace bytecode + komunikace s API
Jensor	Tvorba vlastních sond	Instrumentace bytecode
JIP	Sledování dob odezvy, trasování programu	Aspektové změny zdrojových textů
hprof	Čas v CPU, stav vláken, stav heapu	Součást JVM

Přehled nástrojů - placené

Nástroj	Cena
JProbe	750 USD
Jprofiler	400 USD
YourKit	500 USD



Použití VisualVM

- Jako součást NetBeans
  - Existuje i plugin pro Eclipse
- Jako samostatná aplikace
  - Připojuje se k běžícím aplikacím

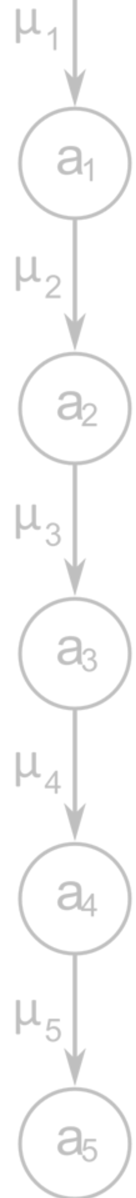
Použití VisualVM - NetBeans

- 3 spuštěcí dialogy
  - Sledování aplikace (běh vláken)
  - Profilování CPU
  - Profilování paměti
- Dynamická i statická analýza
- Možnost vzorkování / instrumentace

μ1 a1 μ2 a2 μ3 a3 μ4 a4 μ5 a5

Použití VisualVM - NetBeans

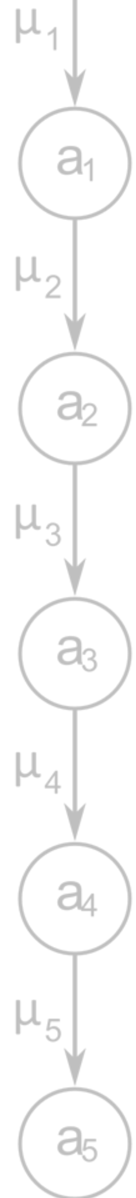
- Do zdrojových textů lze vložit profilovací body
  - Nastavitelné vlastnosti, triggery
  - Stopky – měření času
  - Získání snapshotu
    - Stavu aplikace
    - Heapu
- V kontextovém menu – profiling



Použití VisualVM - NetBeans

- Profil aplikace
  - Seznam všech volaných metod
  - Strom volání
  - Doba strávená v metodě
    - V jejím vlastním kódu
    - V dalších voláních

→ lze najít co má cenu optimalizovat

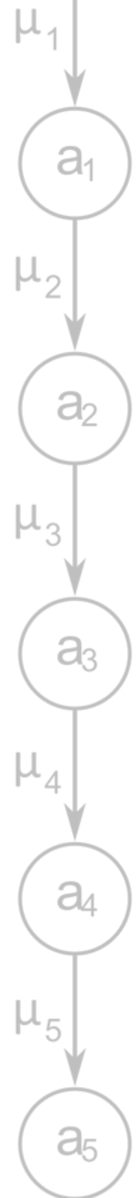


Použití VisualVM - NetBeans

- Sledování vláken
  - Podobné jako sledování programu
  - Stav vlákna a jeho historie
    - Jak dlouho vlákno spalo / čekalo

Použití VisualVM - NetBeans

- Procházení uloženého heapu
  - Přehled všech tříd a jejich instancí
  - Počty instancí, obsazená paměť
- Analýza referencí
  - Všechny atributy objektu
  - Reference ukazující na objekt



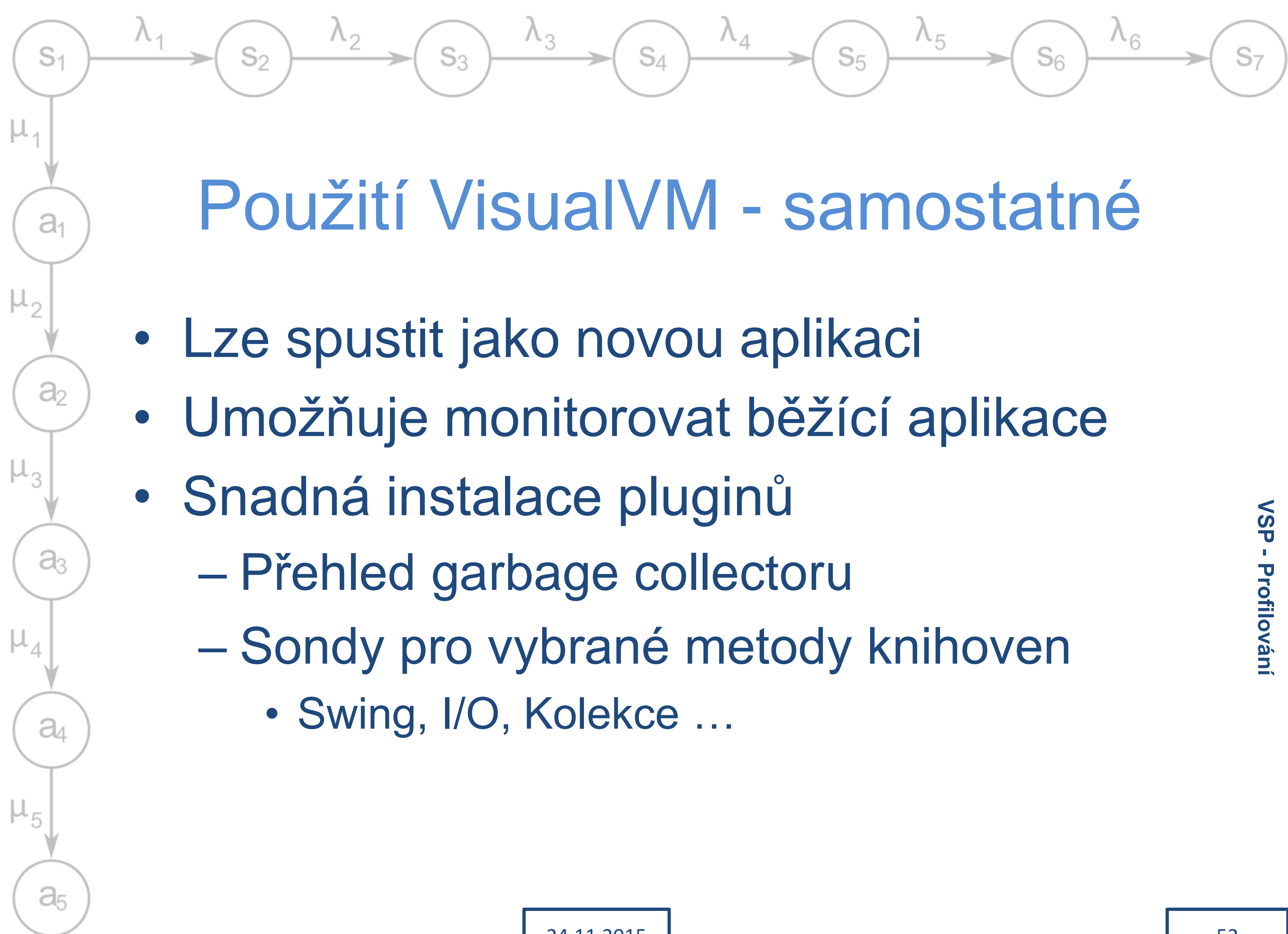
Použití VisualVM - NetBeans

- Hledání možných memory leaků
  - Lze porovnat dva memory dumpy a zobrazit rozdíly
  - Před získáním dumpu provést garbage kolekci

Použití VisualVM - NetBeans

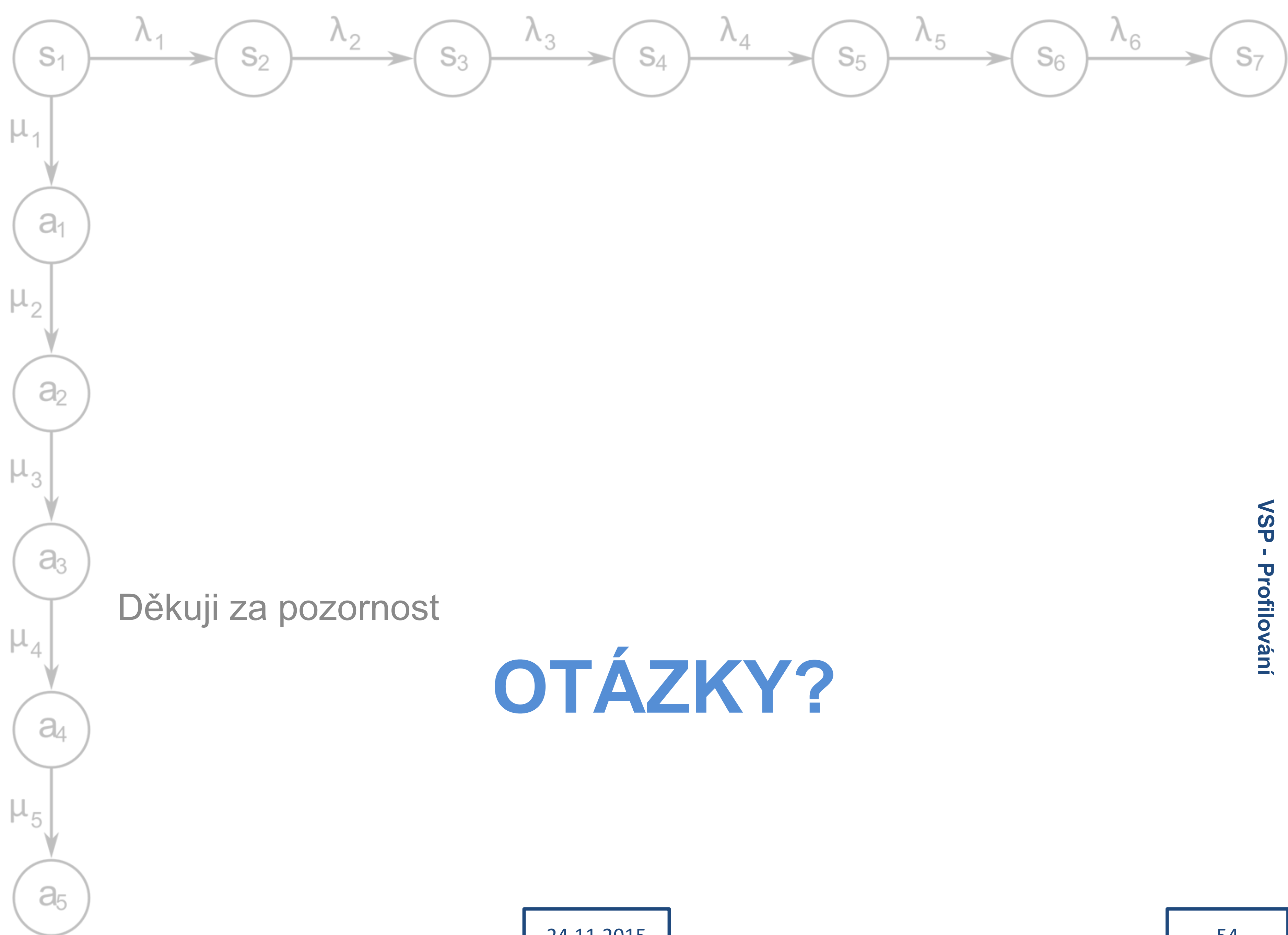
- Možnost využít OQL jazyk
  - Podobné jako SQL
  - Hledání konkrétních instancí

```
select a from
java.util.LinkedList a where
a.size > 10
```


Použití VisualVM - samostatné
• Lze spustit jako novou aplikaci
• Umožňuje monitorovat běžící aplikace
• Snadná instalace pluginů
– Přehled garbage collectoru
– Sondy pro vybrané metody knihoven
• Swing, I/O, Kolekce …
VSP - Profilování
s_1 s_2 s_3 s_4 s_5 s_6 s_7
λ_1 λ_2 λ_3 λ_4 λ_5 λ_6
a_1 a_2 a_3 a_4 a_5
μ_1 μ_2 μ_3 μ_4 μ_5

Rizika

- Zahlcení informacemi
- Chybná interpretace výsledků
  - Např. metoda čekající na vstup



Děkuji za pozornost

OTÁZKY?