KIV/TSI - Seminář C++

10. Profilování kódu, hledání úniků paměti

Martin Úbl

KIV ZČU

2020/2021

Výkon

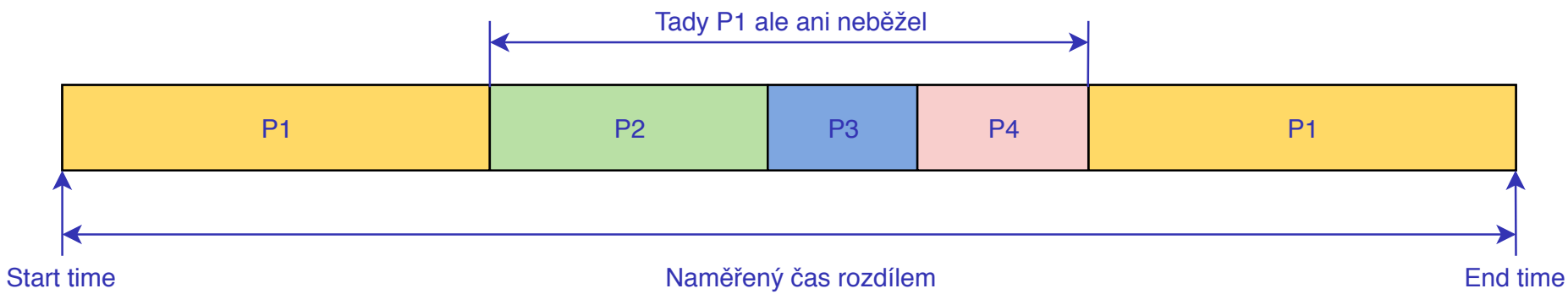
- *(fyz.)* množství práce za jednotku času
- počet instrukcí
  - nejednoznačné
  - každá instrukce může být provedena v rámci jiného počtu CPU cyklů
  - vektorové instrukce - početně klidně frakce CPU cyklů
- jiné měřítko?
- abstrahujeme na operace, potažmo čas
- ve výsledku nás stejně bude zajímat hlavně čas
- analýza výkonnosti programu: profilování

Výkon

- čas provádění
- ovlivněn množstvím aspektů
  - vlastnosti CPU (frekvence, instr. sada, ...)
  - velikost vstupních dat
  - kvantum dat na úrovně cache
  - rychlost RAM
  - vlastnosti architektury
  - plánovač OS
    - dávkový, real-time?
    - preemptivní?
    - priorita
  - paralelizace
  - a mnoho dalších...
- měření by mělo být nezávislé na velké části z nich
  - např. preemptivním plánování

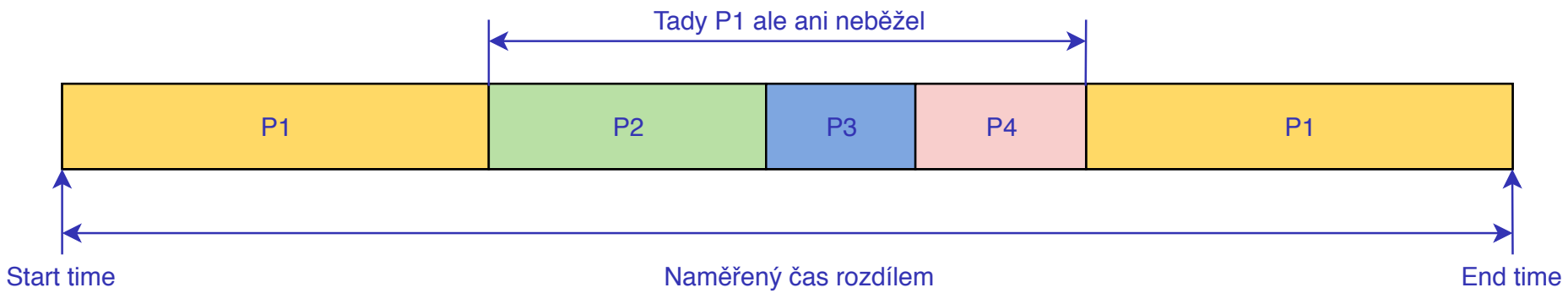
Výkon

- čas provádění
- možné měřit na úrovni programu
- `std::chrono::steady_clock` + odečtení start a end času
- vrací hodnotu časovače s určitými vlastnostmi
- neodfiltruje dobu, kdy proces neběžel
  - pasivní čekání
  - preempce na jiný proces



Výkon

- někdy je vhodné tuto dobu zahrnout
- např. čekání na I/O operace může být relevantní součástí běhu programu
- při čekání na I/O operaci preemptivní plánovače obvykle přeplánují na jiné procesy



Výkon

- více způsobů profilování
  - instrumentace
    - zakompilování diagnostických funkčních volání do programu
    - ruční vs. automatická
    - nutná modifikace programu, jisté zpomalení
  - vzorkování
    - periodické/událostní vzorkování zásobníku a CS:RIP
    - méně přesné, ale minimální dopad na výkon
  - interpretace
    - binární kód je pojat jako mezikód a je spuštěn ve virtuálním stroji
    - „jako Java a bytecode“
    - potenciálně velmi přesné, ale obrovský dopad na výkon
  - kombinace

Instrumentace

- instrumentace
- ruční
  - sběr časových značek, odečtení časů
  - otravné
- automatická
  - kompilátor při překladu zakompiluje dodatečná volání
  - počet volání funkcí, časy běhu
- často kombinace s vzorkováním

```
double transform(double a, double b)
{
    __mcount();
   return a*15.5+b;
}
```

Vzorkování

- vzorkování
- prováděno buď na základě událostí nebo s pravidelnou periodou
- lze nastavit systémový časovač
  - softwarový
  - hardwarový
- obsluha časovače snímkuje stav programu
- systémové volání `profil()`

Vzorkování

- někdy je tedy vhodné jiné měřítko
- *Hardware Performance Counters*
- speciální registry procesoru, které při události inkrementují svou hodnotu
- při přetečení generují přerušení (NMI)
- operační systém signalizuje sběrný program
  - ten zaznamená událost
  - přidá časová razítka
  - někdy snímek zásobníku

Vzorkování

- *Hardware Performance Counters*
  - při změně programového čítače („čítač instrukcí“)
  - při výpadku rámce z L1/2/3 cache procesoru
  - při špatném predikování výsledku skoku
  - TLB miss
  - přerušení a systémová volání
  - a spousty dalších
- jsou součástí PCB a kontextu procesu
  - zaručují tedy izolovanost od ostatních procesů
  - až na sdílené prostředky, jako např. cache CPU a TLB

Profilery

- program pro sběr profilovacích dat - profiler
- obvykle jen program, co využívá vlastností jádra OS
- navěsí se na systémové události
  - časovač
  - přetečení HPC (NMI)
  - další volitelné
- od OS obdrží potřebné datové struktury
- uloží je v nějakém formátu
  - obvykle do paměti
  - jednou za čas „přesypání“ do souboru

Profilery

- MS Visual Studio Performance Profiling
  - MS Windows, součást Visual Studio
  - instrumentace, vzorkování, HPC
- gprof
  - GNU/Linux, macOS, součástí gcc
  - instrumentace, vzorkování, (HPC)
- perf
  - GNU/Linux
  - vzorkování, HPC
- Intel VTune Amplifier
  - viz KIV/PPR

Profilery

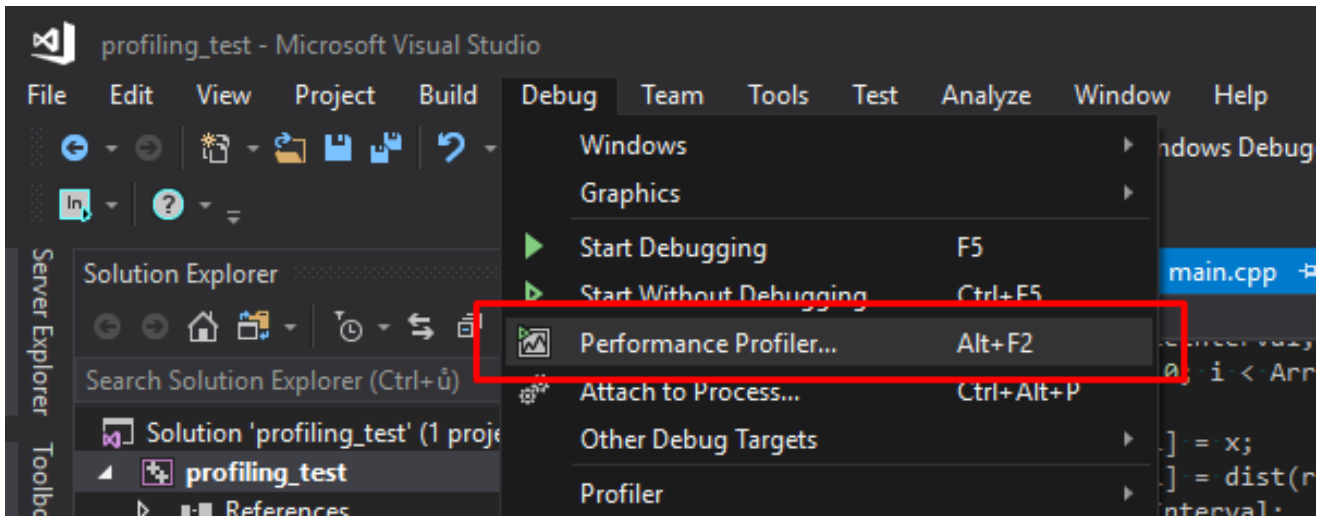
- občas může být problém s výstupy kvůli některým optimalizacím
  - implicitní inlinování funkcí
    - `-fno-inline-small-functions` (GCC)
    - `/Ob0` (MSVS)
  - nepoužití frame pointerů
    - `-fno-omit-frame-pointer` (GCC)
    - `/Oy-` (MSVS)

Profilery

- čas/vzorky
  - inkluzivní
    - čas/vzorky ve funkci a všech, které volá
  - exkluzivní
    - čas/vzorky pouze v rámci instrukčního toku dané funkce

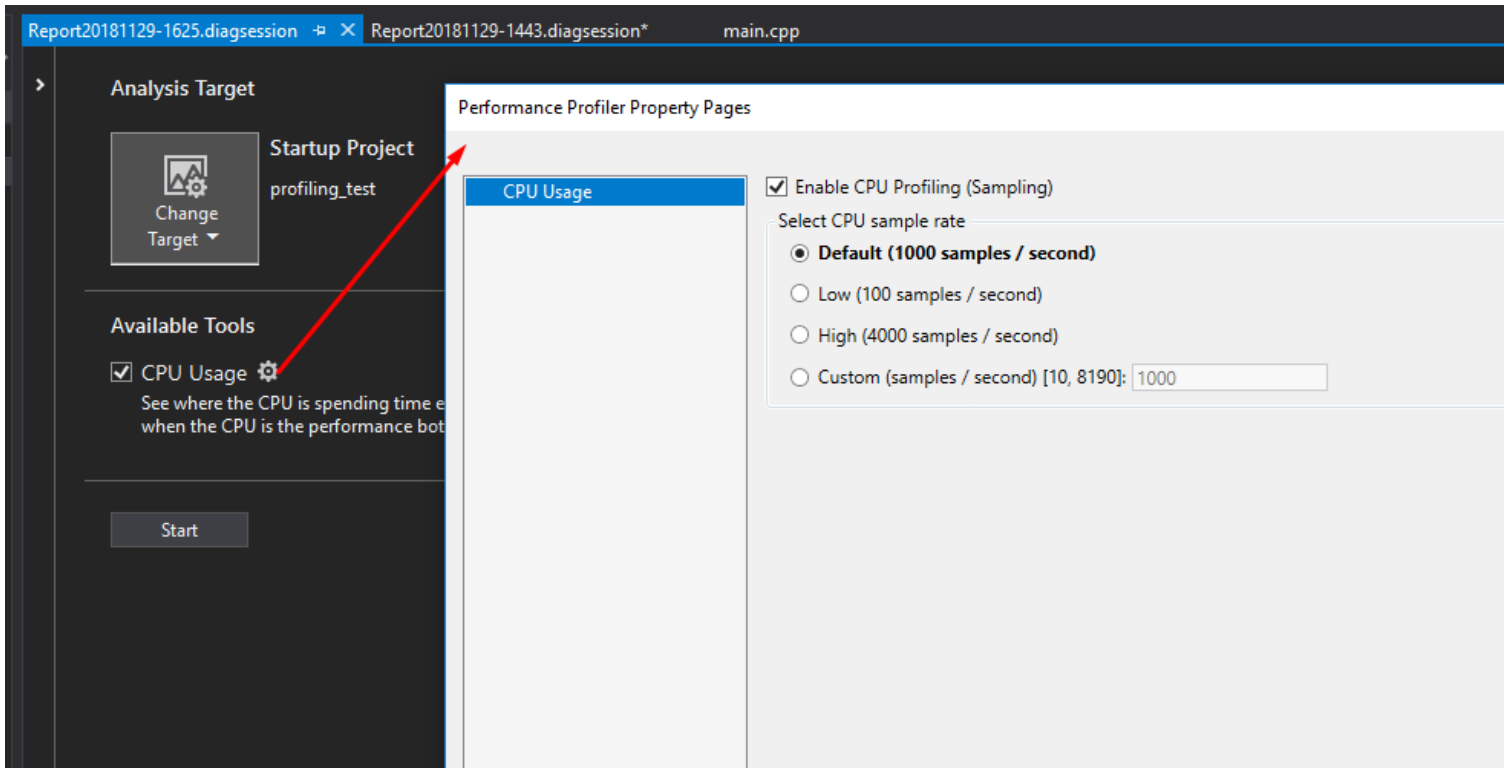
Profilery

- MS Visual Studio Performance Profiling



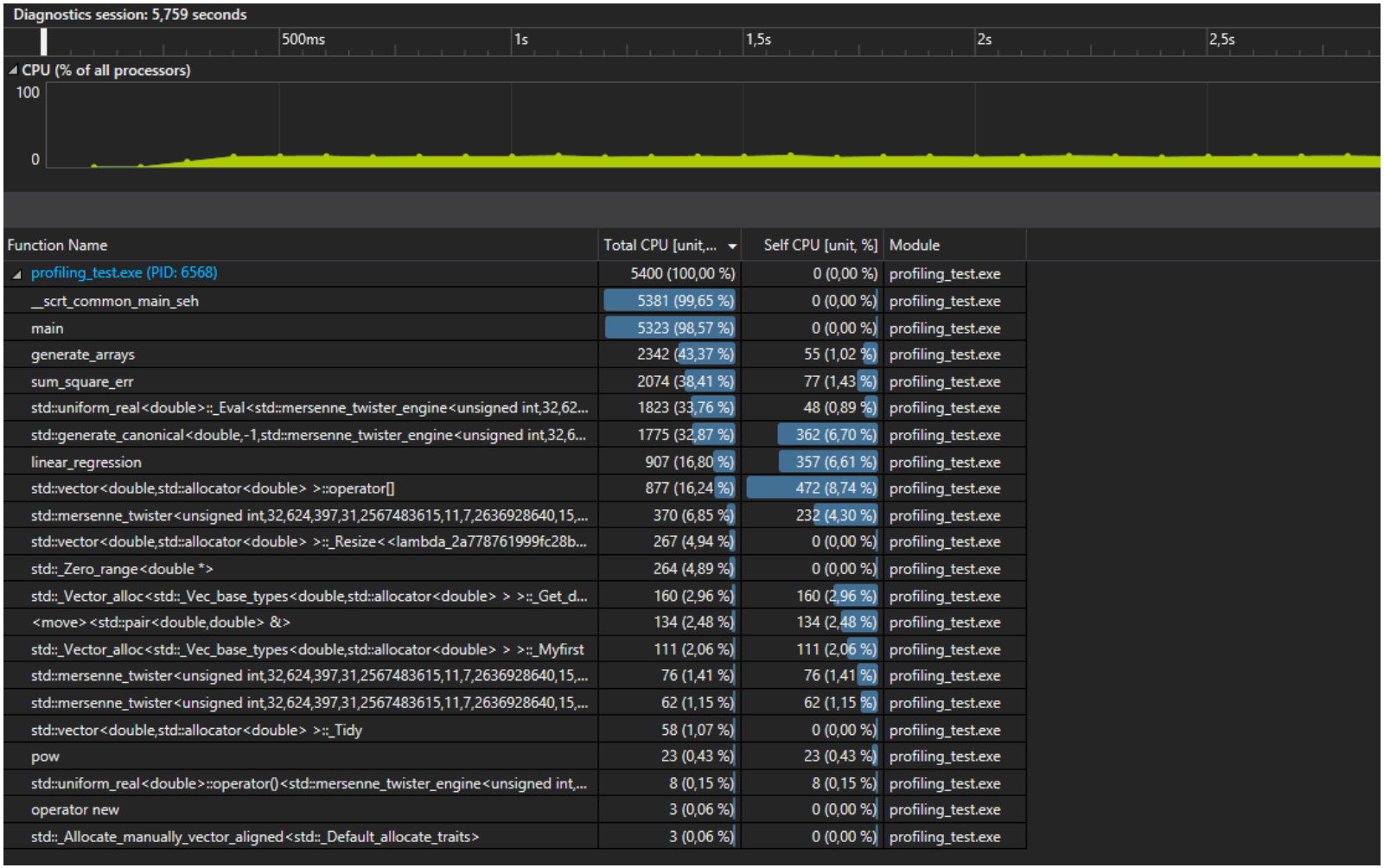
Profilery

- MS Visual Studio Performance Profiling



Profilery

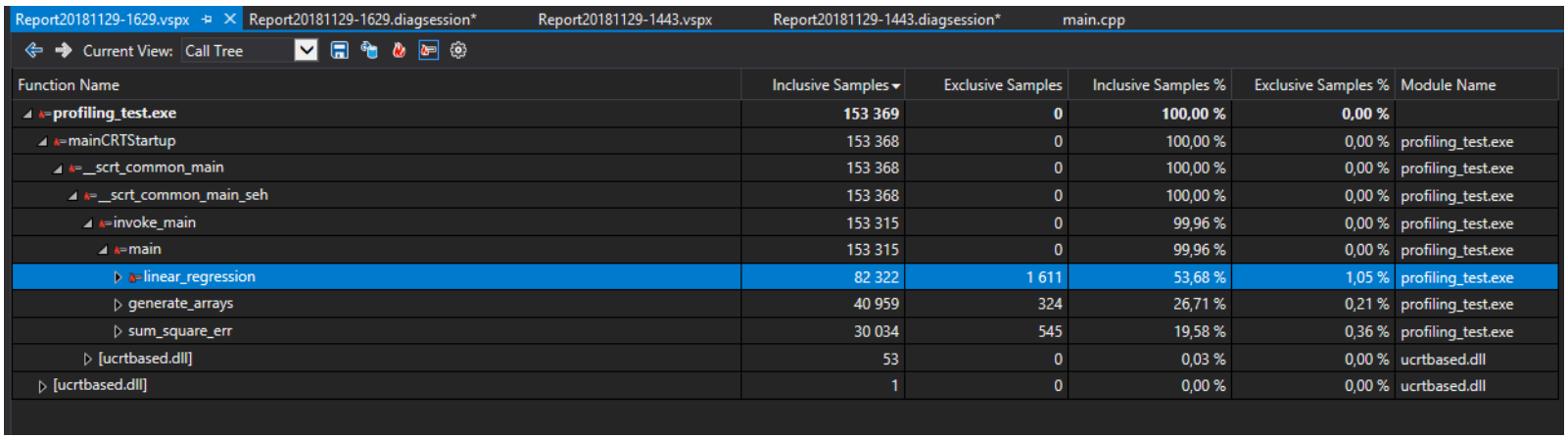
- flat view



Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
profiling_test.exe (PID: 6568)	5400 (100,00 %)	0 (0,00 %)	profiling_test.exe
__scrt_common_main_seh	5381 (99,65 %)	0 (0,00 %)	profiling_test.exe
main	5323 (98,57 %)	0 (0,00 %)	profiling_test.exe
generate_arrays	2342 (43,37 %)	55 (1,02 %)	profiling_test.exe
sum_square_err	2074 (38,41 %)	77 (1,43 %)	profiling_test.exe
std::uniform_real<double>::_Eval<std::mersenne_twister_engine<unsigned int,32,62...	1823 (33,76 %)	48 (0,89 %)	profiling_test.exe
std::generate_canonical<double,-1,std::mersenne_twister_engine<unsigned int,32,6...	1775 (32,87 %)	362 (6,70 %)	profiling_test.exe
linear_regression	907 (16,80 %)	357 (6,61 %)	profiling_test.exe
std::vector<double,std::allocator<double> >::operator[]	877 (16,24 %)	472 (8,74 %)	profiling_test.exe
std::mersenne_twister<unsigned int,32,624,397,31,2567483615,11,7,2636928640,15,...	370 (6,85 %)	232 (4,30 %)	profiling_test.exe
std::vector<double,std::allocator<double> >::_Resize<<lambda_2a778761999fc28b...	267 (4,94 %)	0 (0,00 %)	profiling_test.exe
std::_Zero_range<double *>	264 (4,89 %)	0 (0,00 %)	profiling_test.exe
std::_Vector_alloc<std::_Vec_base_types<double,std::allocator<double> > >::_Get_d...	160 (2,96 %)	160 (2,96 %)	profiling_test.exe
<move><std::pair<double,double> &>	134 (2,48 %)	134 (2,48 %)	profiling_test.exe
std::_Vector_alloc<std::_Vec_base_types<double,std::allocator<double> > >::_Myfirst	111 (2,06 %)	111 (2,06 %)	profiling_test.exe
std::mersenne_twister<unsigned int,32,624,397,31,2567483615,11,7,2636928640,15,...	76 (1,41 %)	76 (1,41 %)	profiling_test.exe
std::mersenne_twister<unsigned int,32,624,397,31,2567483615,11,7,2636928640,15,...	62 (1,15 %)	62 (1,15 %)	profiling_test.exe
std::vector<double,std::allocator<double> >::_Tidy	58 (1,07 %)	0 (0,00 %)	profiling_test.exe
pow	23 (0,43 %)	23 (0,43 %)	profiling_test.exe
std::uniform_real<double>::operator()<std::mersenne_twister_engine<unsigned int,...	8 (0,15 %)	8 (0,15 %)	profiling_test.exe
operator new	3 (0,06 %)	0 (0,00 %)	profiling_test.exe
std::_Allocate_manually_vector_aligned<std::_Default_allocate_traits>	3 (0,06 %)	0 (0,00 %)	profiling_test.exe

Profilery

- call tree



Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %	Module Name
profiling_test.exe	**153 369**	**0**	**100,00 %**	**0,00 %**	
mainCRTStartup	153 368	0	100,00 %	0,00 %	profiling_test.exe
__scrt_common_main	153 368	0	100,00 %	0,00 %	profiling_test.exe
__scrt_common_main_seh	153 368	0	100,00 %	0,00 %	profiling_test.exe
invoke_main	153 315	0	99,96 %	0,00 %	profiling_test.exe
main	153 315	0	99,96 %	0,00 %	profiling_test.exe
linear_regression	82 322	1 611	53,68 %	1,05 %	profiling_test.exe
generate_arrays	40 959	324	26,71 %	0,21 %	profiling_test.exe
sum_square_err	30 034	545	19,58 %	0,36 %	profiling_test.exe
[ucrtbased.dll]	53	0	0,03 %	0,00 %	ucrtbased.dll
[ucrtbased.dll]	1	0	0,00 %	0,00 %	ucrtbased.dll

gprof

- nutno kompilovat s přepínačem `-pg`
- pak klasicky spustit program
- vygeneruje se soubor `gmon.out`
- analýza příkazem `gprof`

```
g++ -pg main.cpp
./a.out
gprof
```

perf

- nutno spouštět jako root
- spuštění příkazem `perf record`
  - jelikož budeme chtít zaznamenávat i strom volání, přidáme přepínač `-call-graph=fp`
- vygeneruje se soubor `perf.data`
- analýza příkazem `perf report`

```
g++ main.cpp
perf record --call-graph=fp ./a.out
perf report
```

perf

- perf call tree

```
Samples: 864  of event 'cycles', Event count (approx.): 672421625
  Children      Self  Command  Shared Object       Symbol
+   97,51%     0,00%  a.out    [unknown]           [k] 0x220e258d4c544155
+   97,51%     0,00%  a.out    libc-2.27.so        [.] __libc_start_main
+   97,51%     0,00%  a.out    a.out               [.] main
+   88,34%     2,28%  a.out    a.out               [.] generate_arrays
+   39,52%    39,52%  a.out    a.out               [.] std::generate_canonical<double, 53ul, std::mersenne_twister_
+   34,37%     4,86%  a.out    libc-2.27.so        [.] _mcount
+   30,44%    30,09%  a.out    libc-2.27.so        [.] __mcount_internal
+   12,45%     2,31%  a.out    a.out               [.] std::vector<double, std::allocator<double> >::_M_default_app
+    8,23%     1,64%  a.out    [kernel.kallsyms]   [k] async_page_fault
+    6,59%     0,14%  a.out    [kernel.kallsyms]   [k] __do_page_fault
+    6,45%     0,35%  a.out    [kernel.kallsyms]   [k] handle_mm_fault
+    6,10%     1,06%  a.out    [kernel.kallsyms]   [k] __handle_mm_fault
+    4,20%     4,20%  a.out    a.out               [.] linear_regression
+    3,64%     3,53%  a.out    a.out               [.] sum_square_err
+    3,18%     0,11%  a.out    [kernel.kallsyms]   [k] alloc_pages_vma
+    3,15%     0,22%  a.out    [kernel.kallsyms]   [k] get_page_from_freelist
+    3,15%     0,00%  a.out    [kernel.kallsyms]   [k] __alloc_pages_nodemask
+    2,55%     2,55%  a.out    [kernel.kallsyms]   [k] clear_page_erms
+    1,92%     0,00%  a.out    [kernel.kallsyms]   [k] entry_SYSCALL_64_after_hwframe
+    1,92%     0,00%  a.out    [kernel.kallsyms]   [k] do_syscall_64
+    1,77%     0,00%  a.out    [kernel.kallsyms]   [k] do_munmap
+    1,77%     0,00%  a.out    [kernel.kallsyms]   [k] unmap_region
+    1,73%     0,00%  a.out    libc-2.27.so        [.] __munmap
+    1,73%     0,00%  a.out    [kernel.kallsyms]   [k] __x64_sys_munmap
+    1,73%     0,00%  a.out    [kernel.kallsyms]   [k] vm_munmap
+    1,40%     0,80%  a.out    [kernel.kallsyms]   [k] error_entry
+    1,37%     0,68%  a.out    [kernel.kallsyms]   [k] unmap_page_range
+    1,37%     0,00%  a.out    [kernel.kallsyms]   [k] unmap_vmas
+    1,04%     0,23%  a.out    [kernel.kallsyms]   [k] release_pages
```

Úniky paměti

- úniky paměti
- zapomenuté dealokace
- vynechané dealokace vlivem chybného řízení programu
- pro malý jednoúčelový program s minimem alokací nemusí vadit
- pro velký, třeba interaktivní systém je to velký problém

Úniky paměti

- možná řešení
  - nepoužívat dynamickou alokaci
  - používat RAII obalenou dynamickou alokaci
  - používat co chceme, ale odladit program s nástrojem pro detekci
- nástroje pro detekci
  - CrtDebug (MSVS)
  - Visual Leak Detector (MSVS)
  - Dr. Memory (MS Windows)
  - valgrind memcheck (GNU/Linux, macOS)
    - nejen memory leaky

Úniky paměti

- princip fungování
- obalení alokací vlastní rutinou
- ta zaznamenává veškeré alokace a dealokace
- co v momentě ukončení sezení není uvolněno, reportuje

CrtDebug

- nastavení

```
_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF
               | _CRTDBG_LEAK_CHECK_DF
               | _CRTDBG_CHECK_ALWAYS_DF);
_CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);
_CrtSetReportFile(_CRT_WARN, _CRTDBG_FILE_STDOUT
```

- výpis neuvolněných bloků

```
_CrtDumpMemoryLeaks();
```

- pozn. pro identifikaci místa alokace potřeba předefinovat new

```
#define MYDEBUG_NEW new( _NORMAL_BLOCK, \
                         __FILE__, __LINE__)
#define new MYDEBUG_NEW
```

CrtDebug

- vytvoření snímku heapu, a tím "checkpointu" v běhu programu

```
_CrtMemState state;
_CrtMemCheckpoint(&state);
```

- výpis neuvolněných bloků od daného "checkpointu"

```
_CrtMemDumpAllObjectsSince(&state);
```

VLD

- stažení, instalace
  - `https://kinddragon.github.io/vld/`
- vložení do projektu

```
#include "<cesta k vld>\include\vld.h"
#pragma comment(lib, "<cesta k vld>/vld_x64.lib"
```

- start

```
VLDEnable();
```

- výpis neuvolněných bloků

```
VLDReportLeaks();
```

- pozn. nástroj již není vyvíjen

valgrind

- díky `LD_PRELOAD` překryje alokátory bez nutnosti zasahovat do kódu
- není třeba speciálního kódu nebo překladu
  - doporučuje se však přeložit s debug info (přepínač `-g`)
- spuštění

```
valgrind --tool=memcheck ./program param1
```

- někdy vhodné připojit příznaky pro zjištění původu a maximalizaci hledání

```
valgrind --tool=memcheck --leak-check=full
         --track-origins=yes ./program param1
```