

FERRAMENTA DE PROFILING: Valgrind + Kcachegrind

Por: Gustavo Xavier Pereira

Descrição

A ferramenta de profiling escolhida foi o valgrind, no qual funciona juntamente com o kcachegrind. Valgrind é um software livre que auxilia o trabalho de depuração de programas para as linguagens de programação C e C++. O diferencial deste programa está no fato de que usa uma máquina virtual para simular o acesso à memória do programa em teste, eliminando a necessidade de uso de outras bibliotecas auxiliares ou mudanças drásticas no código. Assim, gerando um relatório mais “limpo” do código.

Características

O Valgrind suporta uma grande variedade de verificações que podem não ser cobertas pelo próprio compilador. É amplamente capaz de detectar bottlenecks e o número de instruções executadas pela máquina, além de possuir ferramentas que detectam erros decorrentes do uso incorreto da memória dinâmica, como por exemplo os vazamentos de memória, alocação e desalocação incorretas e acessos a áreas inválidas.

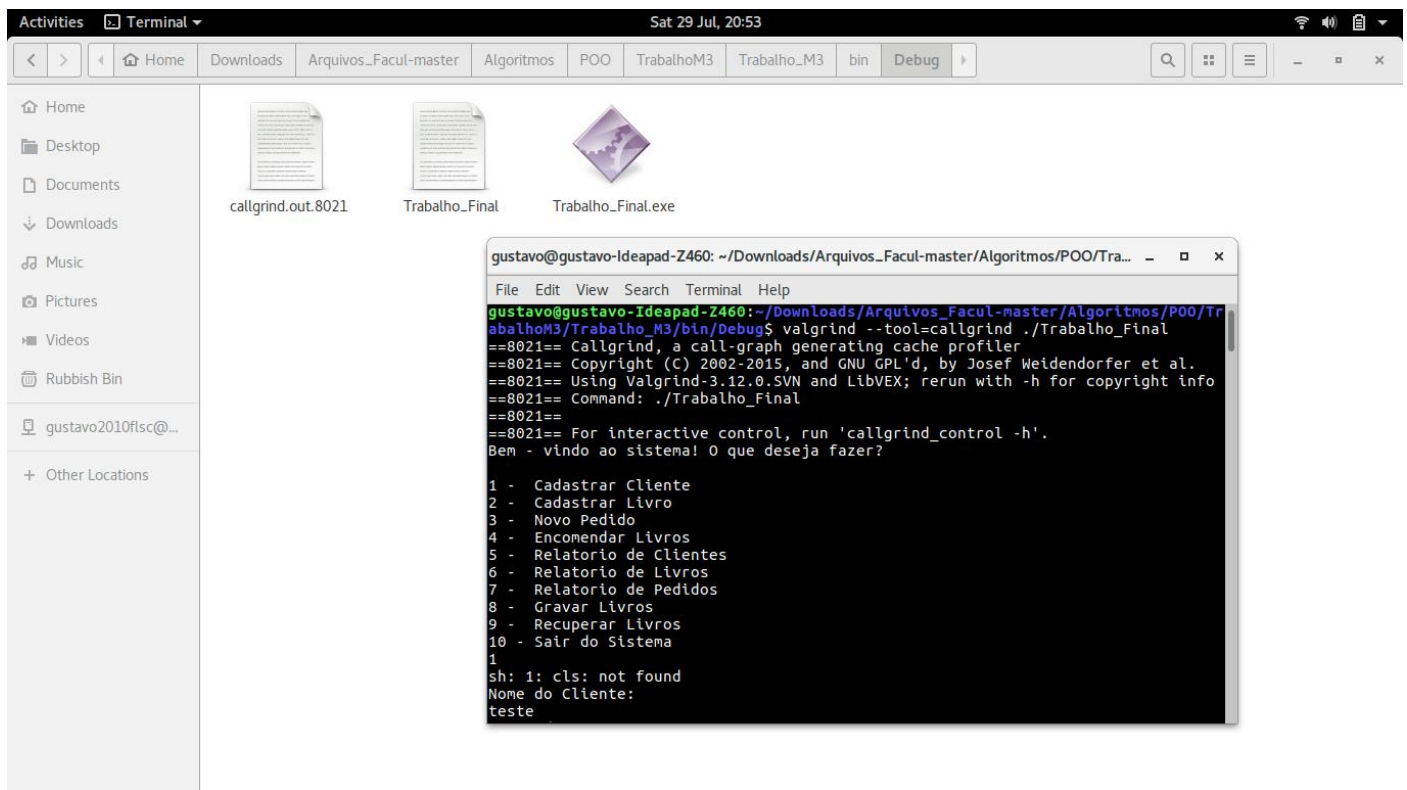
Modo de uso

Após instalar a ferramenta, é preciso executar o programa dentro da máquina virtual do valgrind, no qual gerará um arquivo log, que será lido e interpretado pelo kcachegrind. É recomendável utilizar todas as funcionalidades do programa para uma checagem completa.

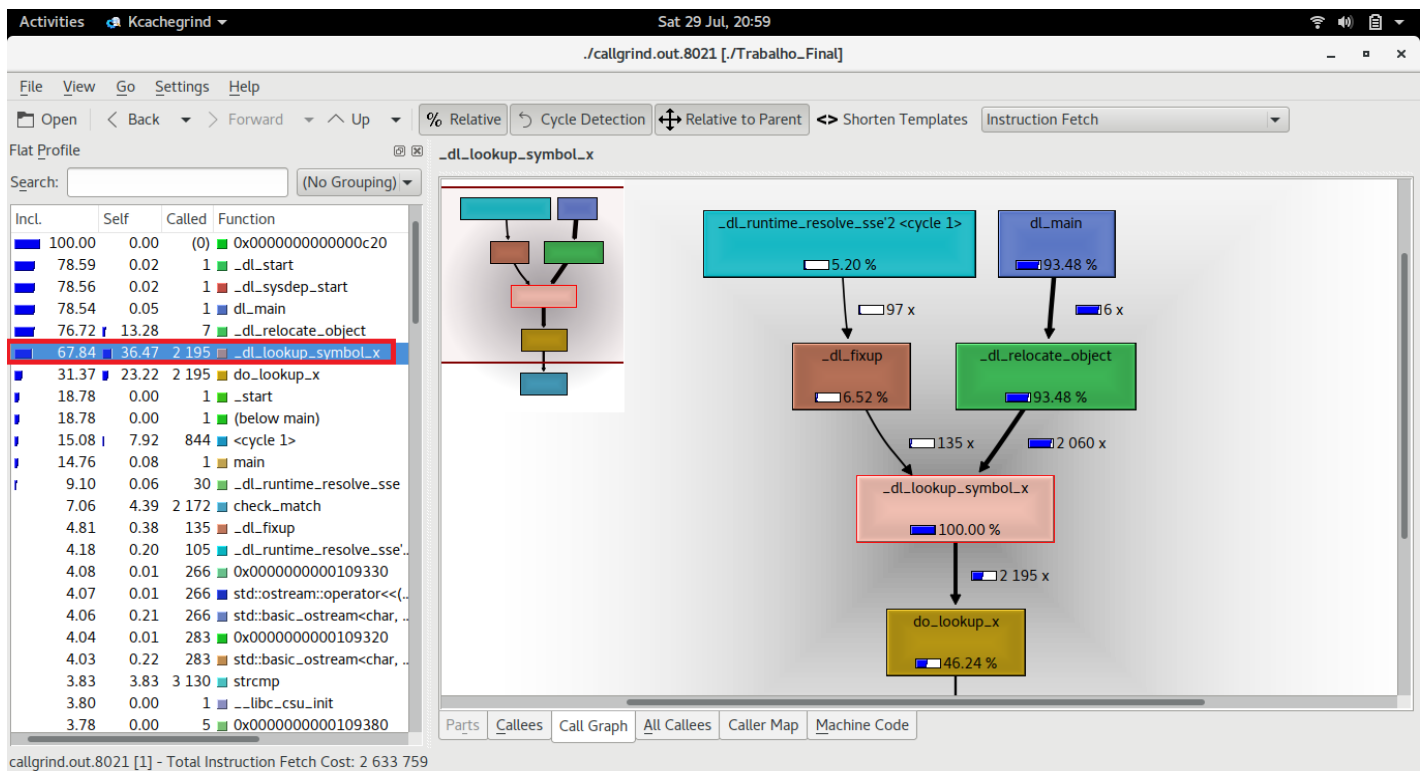
TESTES

Realizei um teste da ferramenta usando um programa gerenciador de vendas de uma livraria, no qual foi o trabalho de algoritmos no semestre passado, utilizando o paradigma da programação orientada a objetos. O programa realiza cadastro de clientes, de livros, gerencia pedidos, vendas, estoque, gravação e recuperação de livros em arquivo externo, etc.

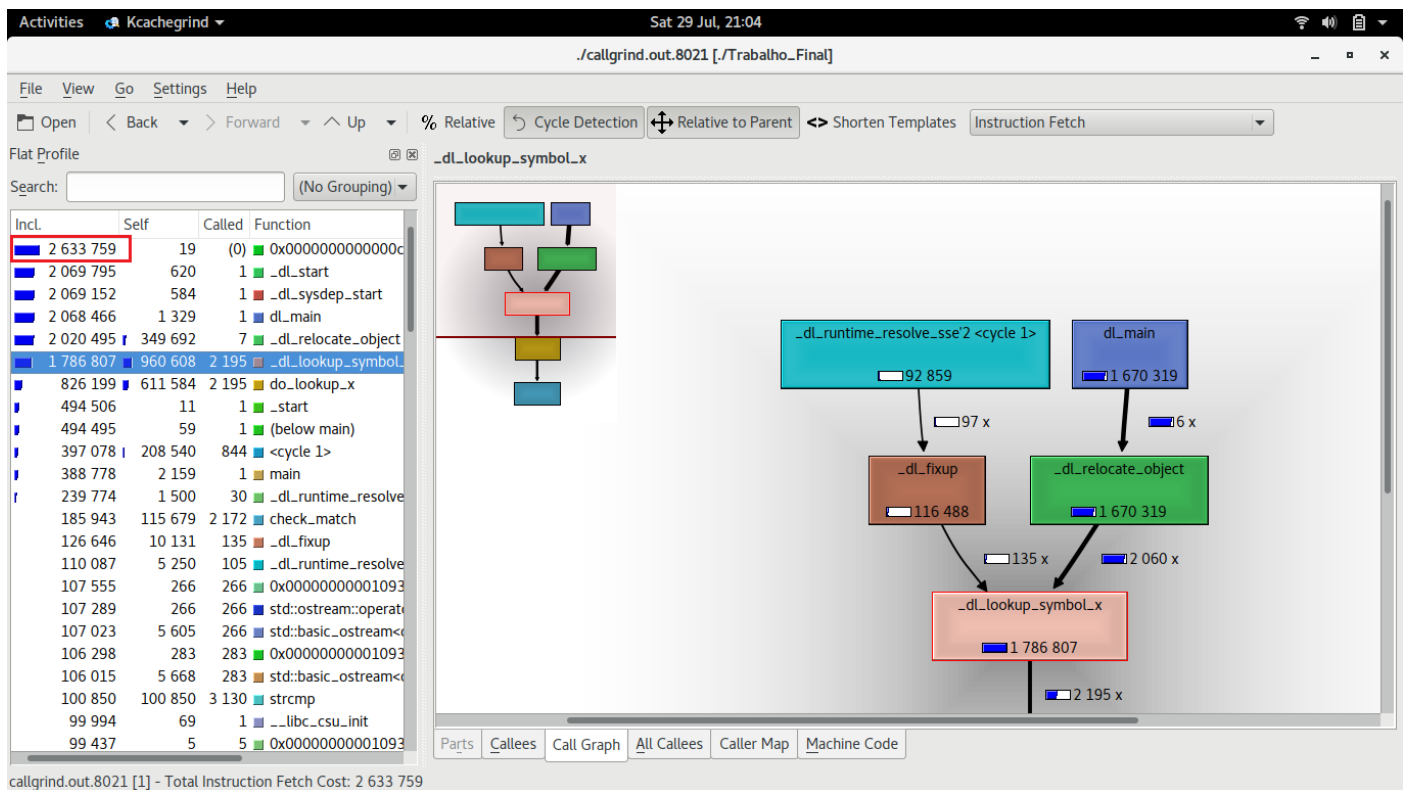
Primeiramente, deve-se executar um scanearamento do programa dentro da interface do valgrind, abaixo uma screenshot:



Note que o programa gerou um arquivo log na pasta com o nome “callgrind.out.8021”, que será aberto posteriormente com o kcachegrind.



Na imagem acima, com o log já aberto no kcachegrind, podemos finalmente estudar o código. Note que o kcachegrind exibe uma árvore de funções e seus respectivos consumos. A função selecionada é considerada o “bottleneck”, pois é a função com maior consumo em todo o código. Sozinha está ocupando 36,47% das instruções.



Aqui, desabilitando a opção “Relative”, podemos observar o consumo de instruções de máquina das funções. Note que o programa inteiro utilizou 2 633 759 instruções.

```

Activities | Text Editor | Sat 29 Jul, 22:12
logfile.out
~/Downloads/Arquivos_Facul-master/Algoritmos/POO/TrabalhoM3/Trabalho_M3/bin/Debug

--8840-- REDIR: 0x4ec7b20 (libstdc++.so.6:operator delete[](void*)) redirected to 0x4c2f6f0
(operator delete[](void*))
--8840-- REDIR: 0x545fd40 (libc.so.6:free) redirected to 0x4c2ecf0 (free)
==8840==
==8840== HEAP SUMMARY:
==8840==   in use at exit: 192 bytes in 4 blocks
==8840==   total heap usage: 16 allocs, 12 frees, 93,785 bytes allocated
==8840==
==8840== Searching for pointers to 4 not-freed blocks
==8840== Checked 114,880 bytes
==8840==
==8840== 192 (136 direct, 56 indirect) bytes in 1 blocks are definitely lost in loss record 4 of 4
==8840==   at 0x4C2E19F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==8840==   by 0x10C28E: main (main.cpp:140)
==8840==
==8840== LEAK SUMMARY:
==8840==   definitely lost: 136 bytes in 1 blocks
==8840==   indirectly lost: 56 bytes in 3 blocks
==8840==   possibly lost: 0 bytes in 0 blocks
==8840==   still reachable: 0 bytes in 0 blocks
==8840==   suppressed: 0 bytes in 0 blocks
==8840==
==8840== Use --track-origins=yes to see where uninitialised values come from
==8840== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
==8840==
==8840== 1 errors in context 1 of 2:
==8840== Syscall param write(buf) points to uninitialised byte(s)
==8840==   at 0x54D18D0: __write_nocancel (syscall-template.S:84)
==8840==   by 0x4EEB515: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.22)
==8840==   by 0x4F29856: std::basic_filebuf<char, std::char_traits<char> >::_M_convert_to_external
(char*, long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.22)
==8840==   by 0x4F29C96: std::basic_filebuf<char, std::char_traits<char> >::overflow(int) (in /
usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.22)
==8840==   by 0x4F29ABA: std::basic_filebuf<char, std::char_traits<char> >::_M_terminate_output()
(in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.22)
==8840==   by 0x4F2CE1C: std::basic_filebuf<char, std::char_traits<char> >::close() (in /usr/lib/
x86_64-linux-gnu/libstdc++.so.6.0.22)
==8840==   by 0x4F2EBAC: std::basic_ofstream<char, std::char_traits<char> >::close() (in /usr/lib/

```

A imagem acima exibe o log do memcheck, no qual mostrou que há vazamentos de memória no arquivo main.cpp e necessita de atenção.

Conclusão

O Valgrind é uma ferramenta extremamente poderosa para o profiling avançado, possui diversas funcionalidades e permite ao programador total conhecimento sobre o que está acontecendo no seu código. Um ponto fraco seria a usabilidade, pois é uma ferramenta complexa e requer um pouco de maestria para ser executada, mas nada que algumas horas “fuçando” não resolvam!

Referências

<http://valgrind.org/> - Página oficial do Valgrind

<https://pt.wikipedia.org/wiki/Valgrind> - Valgrind Wiki

<http://kcachegrind.sourceforge.net/html/Home.html> - Página na web do Kcachegrind