Sistemas de Tiempo Real
Departamento de Ingeniería Electrónica
Universidad de Antioquia
2025-2

Práctica No. 2
Profiling - Análisis de desempeño de software

Realización: En parejas.

Fecha de entrega: Viernes 19 de septiembre del 2025.

Preguntas orientadoras

1. Consulte y describa brevemente en qué consisten las herramientas **Valgrind, Kcachegrind** y **Massif**.
2. Consulte: ¿qué papel desempeña la creación de los ***CallGraph*** en el *"profiling"* de un determinado código?
3. En el contexto de *profiling*, ¿en qué consisten las fugas de memoria (*memory leaks*)?
4. ¿En qué consisten la *heap memory*, y la *stack memory*? Indique sus diferencias.
5. ¿En qué consisten los *snapshot* en la plataforma **Massif**?
6. ¿Por qué cree usted que el análisis de fugas y uso de memoria es importante en el diseño de **Sistemas de Tiempo-Real**? **(Explique)**.

1. Instalación de herramientas requeridas para la presente Práctica

Instalación del Valgrind, Kcachegrind y Massif-visualizer:

En su distribución de Linux instalada, realice la instalación de los paquetes **Valgrind, Kcachegrind** y **Massif-Visualizer**. Para ello ejecute el procedimiento indicado a continuación:

sudo apt-get update

sudo apt-get install build-essential

sudo apt-get install Valgrind

sudo apt-get install Kcachegrind

sudo apt-get install massif-visualizer

2. Descarga y análisis de código requerido en la presente Práctica

2.1. En su carpeta de usuario, dentro de la distribución Linux escogida para la presente Práctica, cree la carpeta **LAB2_RTS**.

2.2. De la siguiente **carpeta**[1] del laboratorio, descargue el código fuente llamado **sensor_data_processing.cpp**, y copie este archivo en su nueva carpeta **LAB2_RTS**.

2.3. Estudie y analice el código fuente suministrado, e indique sus aspectos más relevantes. Igualmente explique las **palabras claves** de C++ que para usted son novedosas[2].

3. Instrumentalización de código fuente

3.1. Consulte la utilidad y el uso de los macros de **Valgrind** denominados "**CALLGRIND_START_INSTRUMENTATION**", "**CALLGRIND_STOP_INSTRUMENTATION**" y "**CALLGRIND_TOGGLE_COLLECT**".

[1] https://drive.google.com/drive/folders/1_Wcq-6zRvCT7BP4vpSjFbRD0La_I3Ch_?usp=sharing

[2] Esté código fuente está escrito en la versión **C++ 11**.

3.2. Modifique el código descargado, y agregue la linea de código **#include <Valgrind/callgrind.h>,** a fin de incluir los macros de **Valgrind** requeridos para instrumentalizar la generación de ***CallGraphs***.

3.3. A partir de su estudio realizado anteriormente del código fuente descargado, determine una sección de dicho código que merezca un análisis detallado del tiempo de ejecución (ciclos, llamados a funciones, recursiones, etc.) y delimite dicha sección con los macros **"CALLGRIND_START_INSTRUMENTATION"** y **"CALLGRIND_STOP_INSTRUMENTATION".**

3.4. Escriba comentarios alrededor de las modificaciones que usted realice en el código fuente suministrado, y de la sección o área que usted seleccionó para llevar a cabo el análisis requerido.

4. Análisis del Tiempo de Ejecución de programas en C++

A continuación, considere que el nombre de archivo **codigofuente.cpp** representa cualquier código fuente a ser analizado.

4.1. Compilación del código fuente:

Mediante el compilador **GCC,** compile el código fuente descargado anteriormente, teniendo en cuenta la activación de las opciones de depuración. Para ello, desde la carpeta **LAB2_RTS** ejecute la siguiente línea de comando[3]:

g++ -ggdb3 -O0 -o CodigoExe codigofuente.cpp

4.2. Generación del *Callgraph* de cada programa:

4.2.1. Genere el ***CallGraph*** del ejecutable haciendo uso de la herramienta **Callgrind** de la plataforma **Valgrind**. Para ello ejecute la siguiente línea de comando:

[3] Evite *"copiar y pegar"* desde este archivo **pdf**. Preferiblemente copie manualmente estos comandos en su ejecución en la Consola de su Linux instalado. Al *"copiar y pegar"* puede pegar caracteres ASCII irreconocibles por la Consola de Linux, lo cual puede generar error en la ejecución del comando.

valgrind --tool=callgrind --dump-instr=yes --cache-sim=yes --instr-atstart=no ./CodigoExe

4.2.2. Consulte e indique qué hacen las opciones **--dump-instr=yes, --cache-sim=yes** y **--instr-atstart=no**, indicadas en el comando anterior.

4.2.3. Aún en la carpeta **LAB2_RTS**, identifique el nombre del archivo **callgrind.out.XXXX** que ha sido generado con el comando anterior[4].

4.2.4. A fin de observar el *"profiling"* realizado sobre el programa **"CodigoExe"**, utilice la herramienta **Kcachegrind**. Para ello ejecute el siguiente comando:

kcachegrind callgrind.out.XXXX

4.2.5. Para conocer mayores detalles del uso de la herramienta **Kcachegrind** y la información allí presentada, observe el siguiente tutorial:

https://www.youtube.com/watch?v=h-0HpCblt3A

4.2.6. En el entorno gráfico del **Kcachegrind** observe el contenido de la ventana inferior llamada ***"Call Graph"***. Analice este grafo presentado.

- En su análisis, interprete los datos numéricos presentados sobre dicho grafo, e indique cuáles funciones descritas en el código fuente analizado presentan el mayor y el menor tiempo de ejecución, respectivamente.
- En el menú desplegado con el *click* derecho de su mouse, seleccione la opción ***"Export Graph"***, y guarde su grafo como una imagen **PNG**.

4.2.7. Ahora, en la barra superior de herramientas del **Kcachegrind**, desactive la opción "*relative*".

[4] El número **XXXX** indicado en su archivo de salida, corresponde al número de identificación (**PID**) del respectivo proceso dentro de su distribución Linux.

- Indique cuánto es el estimado total de ciclos de ejecución de cada una de las funciones presentes en su código analizado.

4.2.8. En el centro del **Kcachegrind**, en las pestañas superiores, seleccione la pestaña ***Types***.

- Allí, observe el tiempo que su código utiliza en diversas etapas del procesamiento en CPU (**Instruction Fetch**, **Data Read Access**, **Cycle Estimation**, etc), y acceso a la memoria cache (**L1/LL Instr. Fetch Miss**, **L1/LL Data Read Miss**, **L1 Miss Sum**, etc).
- Analice estos valores, e indique si su código analizado presenta faltas de lectura y de escritura en la cache (*misses*) (**Explique**).

4.2.9. En su **Informe Final** de la Práctica presente el anterior análisis realizado.

5. Análisis de Fugas y Uso de Memoria en C++

5.1. Análisis de Fuga de Memoria

5.1.1. Uno de los aspectos importantes del *profiling* de un determinado código fuente, es el análisis de sus posibles fugas de memoria debidas a fallas en la creación y manipulación dinámica de reservas de memoria. A fin de analizar el manejo de memoria en el código descargado para la presente práctica, ejecute ahora el siguiente comando:

valgrind --leak-check=full ./CodigoExe

5.1.2. Consulte e indique qué hace la opción **-leak-check=full** indicada en el comando anterior.

5.1.3. Explique la salida obtenida luego de ejecutar el anterior comando. ¿Qué fallas de memoria presenta el programa analizado? **(Explique)**.

5.1.4. Resuelva las eventuales fallas de memoria presentadas, y compile nuevamente su programa analizado, siguiendo el comando indicado en la **Sección 4.1**. Verifique que las eventuales fugas de memoria fueron resueltas.

5.1.5. Escriba comentarios y explicaciones de las fallas presentadas y de las correcciones llevada a cabo en el código fuente analizado.

5.2. Análisis de Uso de Memoria

5.2.1. Así como la ejecución de un programa o una aplicación puede ser optimizada luego de identificar las fugas de memoria presentadas, también resulta importante conocer el uso de memoria durante el tiempo que se ejecuta una determinada aplicación. A fin de analizar el uso de memoria de los programas analizados en la presente Práctica, ejecute el siguiente comando para cada uno de los dos programas ejecutables:

valgrind --tool=massif --peak-inaccuracy=0.0 --stacks=yes --heap=yes --time-unit=ms --detailed-freq=1 --threshold=0.0 ./CodigoExe

5.2.2. Consulte e indique qué hacen las opciones **--stacks=yes** y **--heap=yes**, indicadas en el comando anterior.

5.2.3. En su carpeta **LAB2_RTS**, observe el nombre del archivo **massif.out.XXXX** que ha sido generado con el comando anterior[5].

5.2.4. A fin de observar el uso de memoria presentado en el programa ejecutable analizado, utilice la herramienta **Massif-Visualizer**. Para ello ejecute el siguiente comando:

massif-visualizer massif.out.XXXX

5.2.5. Presente capturas de pantalla de los gráficos de memoria generados.

[5] El número **XXXX** indicado en su archivo de salida, corresponde al número de identificación (**PID**) del respectivo proceso dentro de su distribución Linux.

5.2.6. Analice los resultados obtenidos, los picos de memoria presentados y los instantes de tiempo asociados a cada pico de memoria.

- Específicamente, indique el instante de tiempo en que se presenta el pico de memoria. Para mayor precisión en la medida de este tiempo, puede ayudarse del comando **ms_print** (consulte).

5.2.7. Considere ahora el análisis de memoria de su versión modificada del código fuente, donde corrigió y eliminó las fugas de memoria encontradas en la **Sección 5.1** de esta guía.

- Dado este análisis realizado en el programa corregido, indique el instante de tiempo en el cual se libera la memoria en cada programa.

5.2.8. En su **Informe Final** de esta Práctica presente el anterior análisis realizado.

Informe

1. Explique la relevancia del trabajo realizado en esta **Prácticas No. 2** para el diseño e implementación de **Sistemas de Tiempo-Real**.
2. Presente en su informe las respuestas a las preguntas realizadas en la presente guía de laboratorio.
3. Igualmente, agregue las diferentes capturas gráficas pedidas, así como su explicación e interpretación de cada resultado observado.
4. Adjunte a este informe un archivo **zip** de su carpeta **LAB2_RTS**, incluyendo los diferentes archivos de código fuente analizados en esta Práctica de Laboratorio.
5. Presente conclusiones y bibliografía adicional utilizada para realizar este informe.

Referencias Bibliográfica

- How to profile a C program with Valgrind/Callgrind. Available on: https://medium.com/@jacksonbelizario/profiling-a-c-program-with-valgrind-callgrind-b41f15b31527
- Profiling with Valgrind and visualization with KCachegrind. Available on: http://schellcode.github.io/profiling-and-visualization
- Measuring code performance. CERN. Available on: https://indico.cern.ch/event/561981/contributions/2579523/attachments/1462475/2259348/measure-perf.pdf
- Paul Floyd. Valgrind Part 4: Cachegrind and Callgrind. Available on: https://accu.org/journals/overload/20/111/floyd_1886/
- Nate Hardison and Julie Zelenski. Valgrind Callgrind. Available on: https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/resources/callgrind
- Paul Floyd. Valgrind Part 5 – Massif. Available on: https://accu.org/journals/overload/20/112/floyd_1884/
- Stephane Carrez. Optimization with Valgrind Massif and Cachegrind. Available on: https://blog.vacs.fr/vacs/blogs/post.html?post=2013/03/02/Optimization-with-Valgrind-Massif-and-Cachegrind
- Massif: a heap profiler. Available on: https://valgrind.org/docs/manual/ms-manual.html
- Analysis of modules with massif. Available on: https://github.com/wazuh/wazuh/issues/9873