Ayudantía 2

Herramientas debugging y profiling

Ayudante: Matías Francia

Email: matias.francia@usm.cl

Debugging

Es el proceso de identificar, localizar y corregir errores en un programa. Pueden ser errores de lógica, de *runtime*, o de sintaxis.

El debugging generalmente implica:

1. Identificación del error
2. Localización del error
3. Corrección del error
4. Pruebas

Profiling

Es el proceso de analizar el rendimiento de un programa para identificar las áreas que consumen más recursos, como tiempo de CPU o memoria. Se enfoca en optimizar el código para mejorar su eficiencia.

El profiling generalmente implica:

1. Medición de rendimiento
2. Identificación de cuellos de botella
3. Optimización
4. Verificación de mejoras

Herramientas básicas

Debugging: “Chivatos”

Consiste en poner *prints* en el código para localizar en qué sección se encuentra el error. Puede ser para ver en qué momento se cae el programa, o para comprobar que los valores sean los esperados.

Uso de *asserts*: buena práctica.

Es una técnica **simple**, pero **lenta** de implementar en programas grandes.

Profiling: Llamada al sistema

Hacer llamada al sistema para conocer hora del reloj antes y después de una sección de interés en el código. Imprimir diferencia de tiempos.

Es una estrategia **simple** pero **lenta** de implementar en programas grandes. Sólo da cuenta del uso del tiempo, pero no de otros factores importantes como el **uso de memoria**.

El panorama no es completo y el sistema operativo puede interferir en las mediciones.

Herramientas avanzadas: Debugging

GDB/LLDB, Valgrind

GDB/LLDB: herramientas que sirven para debuggear el código. Permiten introducir *breakpoints*, ver valores de variables en tiempo de ejecución y obtener más detalles de errores, como por ejemplo, en qué línea del código se cayó el programa.

Valgrind: herramienta que permite detectar problemas relacionados al uso de la memoria de un programa en C o C++. Ayuda a detectar problemas difíciles de encontrar como: fugas de memoria, acceso a espacios de memoria no inicializada, o uso incorrecto de memoria dinámica.

GDB vs LLDB

1. GDB fue desarrollado por el proyecto GNU, LLDB fue desarrollado inicialmente por Apple en el proyecto LLVM
2. GDB fue lanzado en 1986, LLDB en 2010
3. GDB se integra principalmente con GCC, LLDB con Clang, pero ambos funcionan con otros compiladores
4. GDB no funciona con las últimas versiones de Mac (Chips M1, M2…), LLDB sí lo hace
5. LLDB suele ser más rápido, ya que está más optimizado, especialmente en Mac.

Documentación GDB: https://www.sourceware.org/gdb/

Documentación LLDB: https://lldb.llvm.org/

Comandos útiles de GDB

run (r): Iniciar ejecución del programa

break (b): Establecer breakpoints en funciones o líneas específicas

continue (c): Continuar la ejecución hasta el siguiente breakpoint

step (s): Avanzar a la siguiente línea, entrando en funciones

next (n): Avanzar a la siguiente línea sin entrar en funciones

print (p): Inspeccionar el valor de variables y expresiones

backtrace (bt): Ver pila de llamadas y entender el flujo de ejecución

info locals: Ver todas las variables locales en el contexto actual

finish: Continuar ejecución hasta que la función actual termine y regrese al que la llamó

quit (q): Salir de GDB

Comandos útiles de LLDB

run (r): Iniciar la ejecución del programa

breakpoint set (b): Establecer breakpoints en funciones o líneas específicas

continue (c): Continuar la ejecución hasta el siguiente breakpoint

step (s): Avanzar a la siguiente línea, entrando en funciones

next (n): Avanzar a la siguiente línea sin entrar en funciones

print (p): Imprimir el valor de variables y expresiones

frame variable (v): Mostrar todas las variables locales en el contexto actual

backtrace (bt): Ver la pila de llamadas y entender el flujo de ejecución

finish: Continuar la ejecución hasta que la función actual termine y regrese al que la llamó

quit (q): Salir de LLDB

Cómo debuggear con LLDB?

1. Al trabajar con C++: compilar el programa con el comando -g

```
% g++ -g -o program program.cpp
```

2. Ejecutar lldb pasándole el binario (el programa)

```
% lldb program
```

```
(lldb) target create "build_debug/roundingsat"
Current executable set to '/Users/matiasfranciacarraminana/
oundingsat' (arm64).
(lldb)
```

Nombre de mi programa, en nuestro caso sería program

Cómo debuggear con LLDB?

3. Establecer un breakpoint en una función

```
(lldb) breakpoint set --name 'Solver::cleanVarState'
```

4. Comprobar que se agregó correctamente

```
(lldb) breakpoint list
Current breakpoints:
1: name = 'Solver::cleanVarState', locations = 1
  1.1: where = roundingsat`void rs::Solver::cleanVar
, 0ul, (boost::multiprecision::cpp_integer_type)1, (
boost::multiprecision::expression_template_option)1>
1, (boost::multiprecision::cpp_integer_type)1, (boos
t::multiprecision::expression_template_option)1>>(st
<int>>&, std::__1::vector<int, std::__1::allocator<i
00000010011b274], unresolved, hit count = 0

(lldb)
```

Cómo debuggear con LLDB?

5. Ejecutar el comando run

```
(lldb) run program
```

```
   1106 void Solver::cleanVarState(const std::vector<Var> ordered, std::vector<int>& sta
   1107   int& stableCount, bool& flippableActive) {
   1108
-> 1109   int solSize = (int)lastSol.size() - stats.NAUXVARS;
   1110   std::vector<Lit> flippableSoFar(solSize, true); // flippable means that is not
   1111   int nModelDiff = 0; // number of variables that changed from the last model
   1112
Target 0: (roundingsat) stopped.
(lldb)
```

6. Revisar variable de interés

```
(lldb) v solSize
(int) solSize = 1
(lldb)
```

Valgrind

Es un framework que provee herramientas de debugging y profiling que ayudan a crear programas más rápidos y correctos en C y C++.

La herramienta más popular se llama Memcheck, y ayuda a detectar errores de memoria comunes en C y C++ que pueden llevar a caídas del programa y comportamientos no esperados.

Documentación: https://valgrind.org

Quick Start: https://valgrind.org/docs/manual/quick-start.html

Valgrind

Para instalarlo en Linux se puede hacer usando “sudo apt-get install valgrind”, en caso de no funcionar se puede descargar el *tarball* del sitio oficial y compilar manualmente (revisar https://stackoverflow.com/questions/24935217/how-to-install-valgrind-properly).

En Mac se puede hacer usando *homebrew*: “brew install valgrind” (sin embargo, sigue en desarrollo para las versiones de ARM). Una alternativa para usuarios de Mac es la herramienta *leaks*.

En Windows hay que hacerlo utilizando WSL.

Cómo usar Memcheck de Valgrind?

1. Preparar el programa

   Compilar con la *flag* -g para incluir información de debugging, esto hará que Memcheck muestre mensajes de error con los números exactos de las líneas de código.

2. Correr el programa con Memcheck

```
myprog arg1 arg2
```

```
valgrind --leak-check=yes myprog arg1 arg2
```

   La ejecución puede demorar entre 20 y 30 veces más que una ejecución normal.

Cómo usar Memcheck de Valgrind?

3. Ejecuta algún programa con Valgrind y Memcheck.

```c
#include <stdlib.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
    return 0;
}
```

Ven algún problema?

Cómo usar Memcheck de Valgrind?

4. Revisar el output de Memcheck

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

Tipo de error

Seguimiento del error

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

Memory Leak

Manual de Memcheck: https://valgrind.org/docs/manual/mc-manual.html

Cómo usar Memcheck de Valgrind?

5. Corregir el código

```c
#include <stdlib.h>

void f(void)
{
    int* x = malloc(11 * sizeof(int));
    x[10] = 0;
    free(x);
}

int main(void)
{
    f();
    return 0;
}
```

Herramientas avanzadas: Profiling

Perf

Es una gran herramienta de profiling para sistemas linux, la cual se usa principalmente para medir el rendimiento del programa e identificar cuellos de botella, en términos de tiempo, uso de CPU, uso de memoria (principalmente de la caché) y operaciones de I/O.

Sólo funciona para Linux. Suele venir instalado en la mayor parte de las distribuciones grandes de Linux, pero si no está disponible se debe instalar manualmente a través de la bash.

Documentación: https://perf.wiki.kernel.org/index.php/Tutorial

Guía de ayuda: https://www.brendangregg.com/perf.html

Como hacer profiling con perf?

1. Compila el programa con *-g*

```
g++ -g print_for_n.cpp -o print_for_n
```

2. Correr el programa usando *perf record*

```
perf record ./program
```

3. Correr *perf report*

```
perf report
```

Como hacer profiling con perf?

4. Ver la ventana con la información desplegada por *perf report*

```
Samples: 32  of event 'cycles:u', Event count (approx.): 8692860
Overhead  Command      Shared Object        Symbol
  24.03%  print_for_n  [unknown]            [k] 0xffffffff942001e5
  15.92%  print_for_n  [unknown]            [k] 0xffffffff94200bc7
   7.75%  print_for_n  ld-2.28.so           [.] do_lookup_x
   7.63%  print_for_n  ld-2.28.so           [.] _dl_lookup_symbol_x
   7.53%  print_for_n  ld-2.28.so           [.] _dl_relocate_object
   7.16%  print_for_n  libc-2.28.so         [.] _IO_fwrite
   6.36%  print_for_n  libc-2.28.so         [.] _IO_file_xsputn@@GLIBC_2.2.5
   6.05%  print_for_n  libstdc++.so.6.0.25  [.] std::ostream::sentry::~sentry
   5.61%  print_for_n  libstdc++.so.6.0.25  [.] std::num_put<char, std::ostrea
   4.55%  print_for_n  libstdc++.so.6.0.25  [.] std::operator<< <std::char_tra
   3.68%  print_for_n  libstdc++.so.6.0.25  [.] 0x000000000008d294
   2.53%  print_for_n  libstdc++.so.6.0.25  [.] std::__ostream_insert<char, st
   1.20%  print_for_n  libstdc++.so.6.0.25  [.] std::num_get<char, std::istrea
```

Buenas prácticas al programar

Tal y como vimos, las herramientas de debugging y profiling son poderosas. Sin embargo, por sí solas no son suficientes para detectar todos los errores, errores de memoria o cuellos de botella. Es por esto que es importante **modularizar el código** y escribir un código **estructurado** y **ordenado**.

Es importante el uso de *asserts* en el código, estos permiten ver si se cumple una condición (booleano) en tiempo de ejecución. Si no se cumple, el programa lanza un *Assertion Error*, en caso contrario, es ignorado.

Para qué sirven? para ver que la lógica funcione tal y como se pensó, y en caso de que no lo haga, saber exactamente la línea del código en la que hubo problemas.

Otras herramientas

Debugging: debugger del IDE

Muchos IDE de programación cuentan con debuggers integrados, los cuales permiten ejecutar código línea por línea y poner breakpoints. Tienen funcionalidades similares a gdb y lldb, pero el entorno visual suele ser más amigable.

Ejemplos:

- Debugger de Eclipse
- Debugger de VSCode
- Debugger de Visual Studio

Profiling: tiempo del proceso (time)

Es un comando que sirve para medir el tiempo que demora un proceso en ejecutarse. Sirve no sólo con programas, sino que con comandos de shell.

Ejemplo:

```
time ./print_for_n
```

```
./print_for_n  0.00s user 0.01s system 0% cpu 6.377 total
```

1. *user*: tiempo que el programa pasó ejecutando comandos en CPU en **espacio de usuario**
2. *system*: tiempo que el programa pasó ejecutando comandos en CPU en **espacio de kernel**
3. *cpu*: porcentaje de la CPU usada por el programa
4. *total*: tiempo que utilizó el programa en total (*wall-clock time*)

Dudas?

Gracias por su atención!
Fin