Debugging Workshop Notes

The GNU debugger (GDB) is a tool that allows the inspection of your program whilst in execution, called *interactive* mode, or in *post-mortem* mode which is used to inspect the status of your program immediately after a crash. Post-mortem debugging requires the core file corresponding to the program execution.

An important concept to understand when using debugging is the *call stack* which is a dynamic list of active subroutine calls in a program, and each entry is called a *stack frame*. Stack frames hold information on a specific call to a subroutine and local variables. When a subroutine is called, a new stack from is *pushed* onto the top of the stack and when it completes, its stack frame is *popped* from the top of the stack. GDB is useful for debugging serial programs, but complicated for parallel programs. For parallel program debugging, a more specialist tool is required.

Valgrind provides a suite of tools for debugging and profiling executables. It prints out a lot of information, so it has to be sifted through to get the main warning and error messages that related to user programs.

When programs are created using a compiler, they are linked to system libraries. These libraries provide standard subroutines such as `print`, `allocate`, `malloc`, etc. GDB and Valgrind prints messages that related to such system libraries, but they are out of user control so any messages that relate to them can be ignored. It is highly unlikely that system libraries contain errors as they have been developed by groups of users, and they are thoroughly tested. Bugs are usually found in user programs!

To enable the creation of core dumps, type:

```
ulimit -c unlimited
```

On the Grace cluster, this is not required. Ensure that you clean up your core files by removing them after as they can be quite large. The main GDB web site is at:

http://www.gnu.org/software/gdb/documentation/

which contains lots of information on how to use GDB, and the main GDB manual can be found here:

http://sourceware.org/gdb/current/onlinedocs/gdb/

Commands for Interactive Debugging

The command for interactive debugging is:

```
gfortran -g program.f90 -o program
gdb ./program
```

The following are a set of useful GDB commands whilst in interactive debugging mode:

`break filename:n` - set a breakpoint on line `n` in the file `filename`. A breakpoint is a point in the program at which execution will stop so you can inspect variable contents and program state

`watch var` - set a watchpoint on variable `var`. A watchpoint is a point in the program at which execution will stop when the variable `var` changes so you can inspect variable contents and program state

`info break` - report where breakpoints are currently set

`info watchpoints` - report where watchpoints are currently set

`disable n` - disable breakpoint `n`, as numbered in the output from `info <break|watchpoints>`

`enable n` - enable breakpoint `n`, as numbered in the output from `info <break|watchpoints>`

`run` - start running the program within GDB. Your program does not start with GDB until this command is executed

`continue` - continue execution from a breakpoint

`step` - execute the next line, stepping into subroutine calls

`next` - execute the next line, stepping over subroutine calls

`print var` - print the value of variable `var` within the scope of the current stack frame

`quit` - quit the debugger (or use `ctrl-d`)

Commands for Post-Mortem Debugging

The command for interactive debugging is:

```
gfortran -g program.f90 -o program
./program
gdb ./program
```

The second command crashes the program and creates a core file which is the memory image of the program in execution immediately after the crash. The following are a set of useful GDB commands for post-mortem debugging mode:

`bt` - print the backtrace of the call stack to see where the program failed

`fame n` - select the stack frame n in the call stack to be the current focus

`frame` - print the program location (file and line number) in the current stack frame

`list n` - list source code around line n in the file in focus

`list filename:n` - list source code around line n in the specific file which may be different from the file in focus

`list` - list more lines continuing output from the previous list command

`print var` - print the value of a variable var in the scope of the current stack frame

`quit` - quite the debugger (or use `ctrl-d`)

Practical 1 - Interactive Debugging

Download the example programs onto your home directory using the command:

```
wget http://grace-head00.uea.ac.uk/grace-docs/dpo_examples.tar
```

Unpack the file:

```
tar -xvf dpo_examples
cd dpo_examples
```

Compile with debugging information on:

```
gfortran -g program1.f90 -o program1
gdb ./program1
```

Remember the last command above does not execute your program - it simply loads your program in debug mode in GDB. Follow the instructions below:

1. set a breakpoint on line 19
2. set a breakpoint on line 34
3. list the breakpoints
4. run the program
5. what is the value of variable `i`?
6. what is the value of variable `j` and why does it have this value?
7. move into the next instruction, skipping over the subroutine call (use `next` not `step` as `step` will go inside the subroutine call)
8. step over three more times (type `next` three times or `next 3`)
9. print the values of variables `i, k` and `j`
10. set a breakpoint on line 34 and print all the breakpoints
11. continue program execution using the `continue` command
12. print the value of variable `correct`
13. continue program execution and the program should exit normally
14. exit from gdb using either the `quit` command or `ctrl-d`

Practical 1 - Post-Mortem Debugging

Compile with debugging information on:

```
gfortran -g program2.f90 -o program2
./program2
gdb ./program2 ./core.PID
```

1. display the backtrace of the call stack to see where the program failed using the `backtrace` command. This will print system calls which we can ignore
2. select the frame that relates to `program2.f90` - where is the problematic line in the program?
3. list the lines of code around the problematic line using the `list` command with the line number
4. what has caused the program to crash?
5. exit from gdb using either the `quit` command or `ctrl-d`
6. fix the problem, re-compile the program and execute again

Practical 2 - Valgrind

Compile with debugging information on:

```
gfortran -g program3.f90 -o program3
module load valgrind/3.6.1
valgrind ./program3
```

The last command prints a lot of information, but the last section is the most useful:

```
==5443== Conditional jump or move depends on uninitialised value(s)
==5443==    at 0x4E38924: ??? (in /usr/lib64/libgfortran.so.1.0.0)
==5443==    by 0x4E9C6A9: ??? (in /usr/lib64/libgfortran.so.1.0.0)
==5443==    by 0x4E9E819: ??? (in /usr/lib64/libgfortran.so.1.0.0)
==5443==    by 0x400905: MAIN__ (program3.f90:13)
==5443==    by 0x40092D: main (in /gpfs/cpc/xca10fju/gdb_examples/program3)
```

The output indicates that there is an uninitialised variable.

1. where is the uninitialised variable?
2. assign the variable to any value before the problematic line
3. re-compile with the fix and run Valgrind again

You should see output similar to the one below:

```
==5976== HEAP SUMMARY:
==5976==     in use at exit: 0 bytes in 0 blocks
==5976==   total heap usage: 7 allocs, 7 frees, 25,688 bytes allocated
==5976== All heap blocks were freed -- no leaks are possible
==5976== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```