**GNU debugger**

**Introduction**

This is a very short introduction into using gdb. Target audience is a unix user which has a serious problem with some application and wants to help fixing the issue by giving detailed informations to the developer or maintainer of the application.

**Debug information**

Unix binaries can have debug informations attached. gdb can use the debug informations to provide more detailed informations (like function prototypes, source file names and line numbers, ...), thus it is generally a good idea to use a binary which has debug informations.

gcc has a command line switch (-g) which will make gcc write debug informations into object files and executables. It is common practice in the free software community to compile projects with debug info, but strip it off when installing with "make install". Thus you'll usually see a big file size difference between the executables within the source tree and the installed ones, and using the executables from the source tree with gdb gives better results.

**Analyzing a segfault**

The linux kernel is able to write a so called core dump if some application crashes. This core dump records the state of the process at the time of the crash. gdb can read such a core dump and get informations out of it.

Most distributions have core dumps disabled by default, thus you'll have to reenable them first. "ulimit -c unlimited" will do that for the current shell and all processes started from it. Check the bash manpage if you want to know more about the ulimit command.

The next time some application crashes with a segfault you'll see that the message "Segmentation fault" changed to "Segmentation fault (core dumped)" and you'll find a file named "core" or "core.*pid*" in the current directory. Note: You need also write access for the current directory, otherwise that isn't going to happen.

Now it is time to start gdb to see what exactly happened. The first argument for gdb should be the executable of the crashed application, the second the core dump written due to the crash. gdb will read alot of informations and will great you with a "(gdb)" prompt when it is done. The most useful piece of information for a developer is a so called **stacktrace**. Typing "bt" at the prompt asks gdb to print one (which you can mail to the developer). With "quit" you'll leave gdb.

**Other ways to use gdb**

You can start a application completely under gdb's control. To do that just type "gdb*executable*". At the gdb prompt type "run *arguments*" to start the application. If the application receives some signal gdb will give you a prompt asking for commands. You can also type Ctrl-C at any time to get a gdb prompt. The most useful gdb commands are:

**bt**

print a stacktrace (as mentioned above).

**c**

continue running the application.

**print *expression***

print the value of the specified expression. Can be used to inspect the value of certain variables for example, simply specify the variable name as expression

**quit**

quit gdb.

It is also possibel to attach gdb to a already running process. This can be done with "gdb*executable pid*". gdb will attach to the process specified by the process id. That might be useful it some application seems to be stuck in a endless loop and you want to figure where exactly it hangs.

**Advanced gdb debugging**

If you start to do more things with gdb than just printing a stacktrace you likely notice some odd effects like not being able to inspect certain variables. That may happen due to compiler optimizations. If gcc decides to make a variable a register variable (i.e. never store it somewhere in memory) gdb will not see it. To avoid that rebuild the application you are going to debug without optimizations.

More common gdb commands are:

**s**

single step though the application.

**break *argument***

set breakpoints, i.e. make the application stop (and gdb give you a prompt) once it reaches some specific point, some function for example.

**watch *expression***

set a watchpoint, i.e. stop if *expression* has changed. Can be used to catch the line of code which modifies some variable for example.

**up, down**

walk up and down in the stack, to inspect variables in other places of the stack too, not just the current function.

**help**

gdb online help. Go read there all the details, the scope of this document ends just here :-)   
BTW: a nice info manual for gdb is available too.

**other useful debug tools**

**electric fence**

malloc debugger, helps to catch typical memory management bugs like accessing already freed memory, buffer overflows, ...   
Comes as library which can either explicitly linked against the application or loaded via LD\_PRELOAD.

**strace**

print a log of all (or selected) system calls some application performes.   
One common programming mistake is missing error handling, **not** checking the return value of system calls for example. Instead of notifying the user with a message like "opening file *foo* failed with error *bar*" applications simply assume that opening the file worked and just crash later if it didn't. That kind of bug often can easily be found with strace because you'll see the system call return values in the strace log.   
Of course strace is useful for alot of other purposes too.