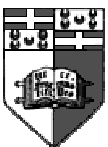




Program Debugging

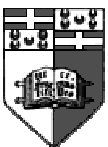
When a program is not working properly, one tries to debug it until the program is fixed up.

- How do we normally debug code:
 - read source code
 - display variable values at runtime
 - follow trace of execution
 - the value of parameters and how they change on exit from a function
 - value of expressions
- Inserting output statements is the most widely used debugging tool, but it relies on previous program knowledge.
- Special tools exist called debuggers which keep track of a process's execution and can make the debugging process much more easy.
- One of the main advantages of debuggers is that apart from helping the original programmer in debugging, it makes external team debugging possible.
- Obviously debuggers will not solve the problem for you but make life much easier.



Unix Debugging Symbols

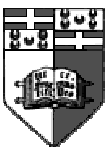
- Before a program can be used by a debugger, it has to be compiled in a certain way to store its debugging symbols.
- These debugging symbols are normally included in the object file, but are sometimes kept in a separate file.
- The only disadvantage is that they make the object file much larger (even twice the size at times).
- Debugging symbols are symbols used by the compiler to bind a name to a memory address in the code.
- Using these symbols it allows us to refer to locations in the code using the original names instead of using memory addresses.
- Debugging symbols consist of 2 types:
 - data labels : store variable names and are given when the data is declared.
 - code labels: names referring to a section in the code, for example function names.





`gdb`

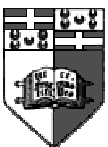
- As an example of a debugger we will use *gdb*.
- To compile your code using debugging symbols use the `-g` option to `gcc`:
 - `gcc -g myfile.c`
- Then execute `gdb` by calling it with your object file
 - `gdb ./a.out`
- `gdb` will respond by a default start up information and will then load the debugging symbols from your code.
- One thing to note, is that `gdb` will need to refer to the source code (upon need), and thus the source code should be placed in the same directory as where you run `gdb`.
- To get a list of help topics write *help* at the prompt and to display important information type:
 - `show warranty`
 - `show copying`



Unix gdb and Optimisation

- With gcc, you can optimize your code to make it run faster, or make it smaller in size.
- Now these optimization will tend to change the order your code runs:
 - variables might be set before the actual instructions;
 - loops might be inverted, etc
- This will tend to make debugging more confusing since there will not be a sequential execution trace, but gdb can still be used in these cases to examine variables and general flow.

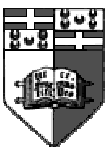
```
while (i<50) {  
    i++;  
}  
might just become i=50 !!!
```





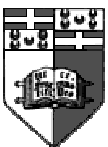
Stack Frames

- gdb uses the concept of stack frames where each stack frame is the data associated with each call to each function in the current execution trace.
- Each stack frame holds inside it the arguments given to the function, the function's local variables and the address at which the function is executing.
- Each stack frame is assigned a number by gdb starting from 0 for the innermost frame and incrementing for the frames that invoked it.
- One can view all stack frames that ended up in the invocation of the current function by using the command *backtrace* or *bt* for short.
- To change the current stack frame use *frame* followed by the new stack.



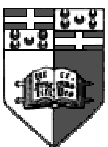
Unix gdb command line

- gdb has an extensive help system invoked using *help* followed by the topic to read about ex. *help data*
- Whenever one executes a command, pressing Enter again will execute the last command
- One can abbreviate commands by using their first characters, as long as there is no ambiguities
 - ba for backtrace will results in the same command.
- Pressing TAB will complete the command or give a list of possible command completions.



Unix Executing and Setting Breakpoints

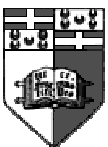
- When you wish to execute your program, you can issue the *run* command. If you wish to pass any command line arguments, state them after the *run* command.
- If your program stops due to some termination signal, the line number will be displayed.
- You can set a breakpoint, which will stop the program by using *break* followed by one of:
 - function name
 - line number
 - address
- Whenever the program halts, you will be returned to the gdb prompt and then you can inspect all variables and current stack frame.
- To continue execution use the *continue* command.
- To list your source file use *list* followed by:
 - line number
 - function nameand an optional preceding ‘filename:’
ex: list main.c:60
- You can exit the current function by giving the command *return* followed by a value if relevant or by running the function to completion using *finish*.





Stepping

- After meeting a breakpoint, one can either continue execution normally or step execute.
- Using the *step* command, one will execute one source file instruction at a time.
- Every time *step* is issued, the next instruction **to be** executed is displayed on screen.
- If you do not wish to step into function calls, use *next* which will execute whole functions at a time.
- All breakpoints remain active until disabled.
- To list all breakpoints use *info break*.
- You can disable breakpoints using *disable* followed by the breakpoint number or clear them using the *clear* command followed by the line number or function name.





Conditional Breakpoints

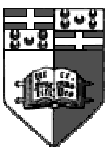
- If you wish to set a breakpoint to be active only if a certain condition is met, you can use *cond* or *if*

ex:

- `cond 5 i=6`
will set breakpoint 5 valid if *i* becomes equal to 6
- `break 45 if j<10`
will set a breakpoint at line 45 that will occur if *j*<10.

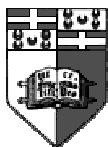
Note that variables must be in context at the line number stated.

- One can also set watch points which will display the value of a variable every time the program stops and the variable is in context. Use the command *watch* followed by an expression to set.
- If you are at the end of a loop, calling *until* will execute the program until the loop is exited, instead of stepping through the loop.



Unix Variable Inspection

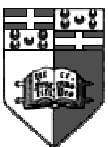
- The command *print* followed by a variable displays its value.
- Context must be observed for proper display.
- You can also use formatted output using *printf*.
 - *printf* “value of *i* = %d” *varname*
- One can set automatic variable display every time that the program stops using *display*
 - *display varname*
- To see what displays are active use *info display* and to remove items use *undisplay* followed by the number of the automatic display.
- A useful setting when displaying structures and unions is to set pretty printing:
 - set p pretty on





gdb and Signals

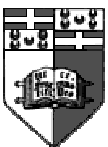
- gdb knows all about program signals. To display all gdb actions for signals using *info handle*.
- To set how gdb react when a signal is encountered use
 - handle *signal keywords* where keywords can be one of
 - *nostop*: Do not stop program when this signal occurs
 - *stop*: stop program when signal occurs
 - *pass*: allow program to see signal
 - *nopass*: block signal from program





Altering Execution

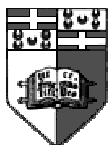
- After finding an error in your code, you can actually test if your code will work properly without compiling again.
- Using the *set variable* command you can change the value of variables
 - set variable b=5
- You can also continue at a different line using the *jump* command followed by a line number.
- You can send a signal directly to your program using *signal* followed by the signal number or its symbolic name. This is different than if a signal was passed from the kernel since a kernel sent signal depends on the action specified in *info handle* while a gdb signal is sent unconditionally.
- You can execute a specific function using *call* followed by a function name.





Other Commands

- To exit from gdb call *quit*.
- gdb is also able to trap a running program and debug it, on the fly.
- gdb also has many formatting abilities listed under the *show* command.
- gdb can also be setup to find the source files in another location apart from the local directory.
- This debugger is also able to run a whole series of commands grouped in a script file.
- Use the help manual pages to find out about extra features not covered in the slides.





Exercises

- Write a small program and try using it with gdb.
- Write a signal handler for SIGALRM and then set it and then block it from delivery using gdb.
- Write a program which crashes and then use gdb to inspect what happened. You can pass the core to gdb like this:
 - `gdb -c core ./a.out`
- Try using gdb from now onwards. Learning it well will ease your shift to other debuggers.

