1.4 TOOLS: STRACE, MAN, GCC, GDB

1. STRACE

STRACE is a debugging and diagnostic tool that monitors the system calls made by a program and the signals it receives. It helps understand how a program interacts with the operating system — useful for debugging, performance analysis, and troubleshooting

Key Points:

- Monitors the communication between a program and the kernel.
- Shows every system call (like open, read, write, fork, etc.) the program makes.
- Helps diagnose file access errors, permission issues, etc.

How It Works

Every program calls services from OS kernel (like open, read, write, fork).

strace traces these calls line by line.

Example

strace ls - Shows all system calls made by ls.

Common usages

```
strace ./a.out              #  trace program
strace -p 1234              # Traces the system calls of the process currently
                               running with PID = 1234.
strace -o output.txt ls      # save trace output to a file
```

2. MAN (MANUAL PAGES)

MAN is the online manual for Linux commands, libraries, system calls, configuration files, etc.

Purpose

- Get detailed help for commands
- Learn syntax, options, examples
- Read documentation for system calls and C library functions

Usage

```
man ls       # manual for ls command
man 2 open     # section 2: system call open()
man 3 printf   # section 3: library function printf()
```

Manual Sections (Important)

Section	Description
1	User commands
2	System calls
3	C library functions
4	Device/driver files
5	Configuration files
8	System administration commands

Useful options

```
man -k keyword   # search manual
man -f command   # brief description
```

3. GCC (GNU COMPILER COLLECTION)

- GCC is the default compiler for C, C++, and other languages in Linux.
- A compiler that converts source code (like C/C++) into executable machine code.

Purpose

- Compile C/C++ programs
- Optimize code
- Link object files
- Produce executables

Basic Compilation Flow

source.c → preprocessing → compilation → assembly → object file → linking → executable

Basic Usage

```
gcc hello.c -o hello
```

Compiles hello.c into an executable named hello.

Common Options

1. gcc -Wall file.c -o a.out   # show all warnings

- Wall means “warn all” — it enables most common compiler warnings.
- Helps you detect potential errors, unused variables, or suspicious code early.
- Recommended for every compilation during development.

2. gcc -g file.c -o a.out # include debug info for gdb

- The -g option adds symbolic debugging info into the executable.
- This allows GDB (GNU Debugger) to show variable names, line numbers, and function details.
- Essential when you plan to debug your program.

3. gcc -O2 file.c -o a.out # optimization

- Optimize the code for better performance.
- The -O flag controls the optimization level.
- -O2 applies a good balance of optimization without slowing compilation much.

Higher levels:

-O0 → no optimization (default)

-O1, -O2, -O3 → increasing levels of optimization

4. gcc -c file.c # compile only, produce .o

- Compile only; do not link.
- Generates an object file (file.o) from the source code.
- Used when building multi-file projects — you compile each .c file separately, then link them together
- This approach speeds up large builds and simplifies debugging.

4. GDB (GNU DEBUGGER)

- gdb is used to debug programs at runtime.
- A debugging tool used to analyze and control the execution of programs — step through code, set breakpoints, view variables, and track down bugs.

Purpose

- Track and fix run-time errors
- View program state (variables, memory, stack)

- Set breakpoints
- Step through code line by line
- Debug segmentation faults

Common Commands

Command	**Meaning**
run	run the program
break main	set breakpoint at main
break file.c:25	break at line 25
next	step over code
step	step into a function
print x	print variable x
backtrace	show stack trace
continue	continue execution
quit	exit gdb

Example

```
gdb ./program
```

you’re starting the **GNU Debugger (GDB)** and loading the executable program into it for debugging.

How It Works — Step by Step

1. Load the Program

- GDB loads your compiled program (./program) into memory but does not start running it yet.
- If the program was compiled with gcc -g, GDB also loads debugging symbols — like variable names, line numbers, and function info.
- This makes it possible to debug at source code level instead of raw machine code.

2. Set Breakpoints (Optional)

- You can tell GDB where to pause execution.
- A breakpoint stops the program before a specific line or function executes.

Example:

```
(gdb) break main
```

→ Sets a breakpoint at the start of main().

3. Run the Program

Start execution under GDB’s control:

```
(gdb) run
```

- GDB runs the program just like normal.
- When it hits a breakpoint or an error (like segmentation fault), it pauses and gives you control.

4. Inspect and Control Execution

While the program is paused, you can:

- **View variable values:**

  ```
  (gdb) print x
  ```

- **Step through code line by line:**

  ```
  (gdb) next      # step to next line (skip function calls)
  (gdb) step      # step into function
  ```

- **Continue until next breakpoint:**

  ```
  (gdb) continue
  ```

- **List the current code:**

  ```
  (gdb) list
  ```

5. Find the Cause of Errors

If your program crashes, GDB stops and shows where it happened:

```
Program received signal SIGSEGV, Segmentation fault.
```

Then you can inspect:

```
(gdb) backtrace
```

→ Shows the chain of function calls that led to the crash.

6. Exit GDB

When finished:

```
(gdb) quit
```