Bugging and debugging

Luka Markušić @ Microblink

July, 2022

Agenda

- Putting bugs in
- Bugs buzzing around
- Discovering bugs
- A view under the hood

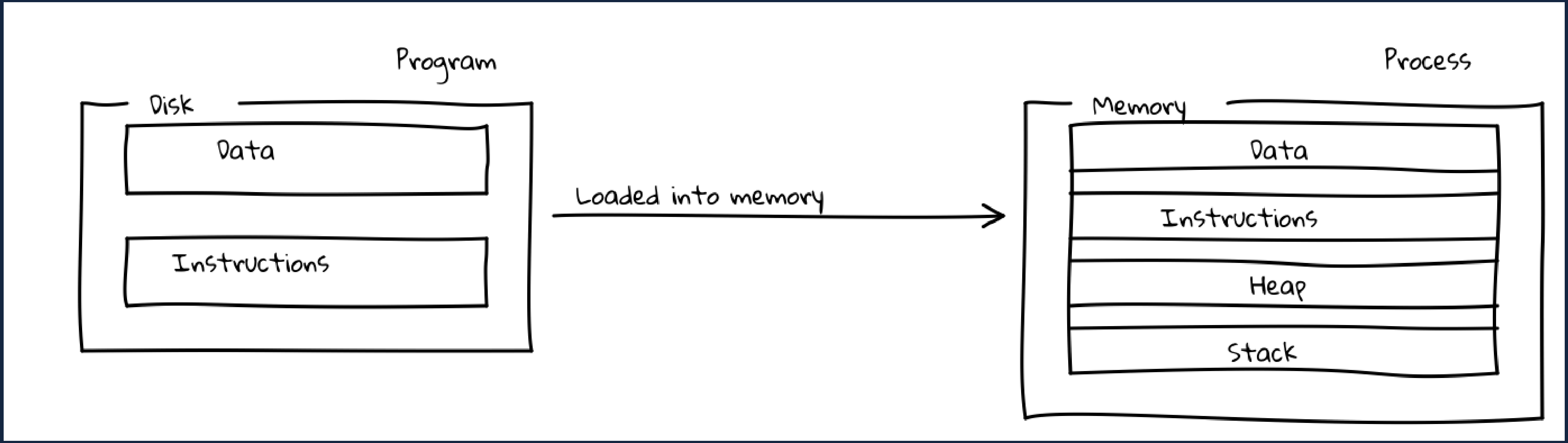
What does running a program entail?

- what *is* a process
- the lifecycle of a process

Linux Processes 101

- process ID (PID), parent PID, owner, command, resource usage
- managed by the kernel/OS

Linux Processes 101



The initial process

- init with `PID = 1`
- manages all other processes
- no parent

What can we do with processes?

fork (clone)

- forking creates a new (*child*) process
- the child is a clone: stack, heap, file descriptors (`stdin`, `stdout`)
- notable differences are: `PID`, `PPID`, `memory locks`, `pending signals`
- execution starts at the same instruction where the parent forked

exec

- replaces the current process image
- passes through arguments
- `execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe`

Let's see it in action

Debuggers in a nutshell

- process wrappers
- able to:
  - manage process state
  - list process info (memory, registers)
  - tons more

Basic process wrapper

Meet our new friends

ptrace (process trace)

```
long ptrace(
    enum __ptrace_request request,
    pid_t pid,
    void * addr,
    void * data);
```

ptrace requests

- `TRACEME`, `CONT`, `KILL`, `PEEKDATA`, `POKEDATA`, `GETREGSET`...
- write, read, restart, suspend...

Signals

- a form of *interprocess communication* (`IPC`)
- `SIGINT`, `SIGILL`, `SIGKILL`, `SIGSTOP`, `SIGCONT`, `SIGTRAP`...

Process state

- `Running` `(R)` - crunching numbers
- `Interruptable sleep` `(S)` - waiting on data, idling
- `Uninterruptable sleep` `(D)` - waiting on *something*
- `Stopped` `(T)` - suspended, waiting for `SIGCONT` or `SIGKILL`
- `Zombie` `(Z)` - dead, but not 'reaped'

What do we usually do with a debugger?

```
void func() {
    // this function has no bugs, I tested it myself
    Bug b{};
    b.messStuffUp();
}

bool func_TEST_is_correct() {
    func();
    return true;
}
```

What do we usually do with a debugger?

```
void func() {
    // this function has no bugs, I tested it myself
    Bug b{};
    b.messStuffUp(); <-- stop here and spill your secrets
}

bool func_TEST_is_correct() {
    func();
    return true;
}
```

Breakpoints

- changing the process state
- software (unlimited) and hardware (limited)
- architecture dependant

Software breakpoints

- tripwire
- the debugger handles what happens
- whenever a thread attempts to execute a piece of code

Hardware breakpoints

- more powerful and flexible
- special `Dr` registers
- can be triggered when reading, writing, or executing a memory address

I need a break(point)!

Registers

- processor's storage
- architecture dependant
- `data`, `address`, `general-purpose`, `status`, `floating-point`, `vector`...
- e.g. `RSP`, `RIP`

The lowest of lows

Must have features

- source-level stepping
- source-level breakpoints
- manipulating variables

Dwarves and elves

- Debug With Arbitrary Record Format (`DWARF`)
- specification developed for symbolic, source-level debugging
- consists of a tree-like `DIE` structure (`Debugging Information Entry`)
- `Line Number Table`, `Call Frame Information` table

Notable dwarves

- `.debug_line` - line number program
- `.debug_info` - core data containing DIEs
- `.debug_frame` - call frame information
- `.debug_types` - type descriptions
- Gimli

DWARF examples

DWARF uses

- which function am I in?
- how do I set a breakpoint on a function?
- reading variables

Source level stepping

- `single instruction`: `ptrace( PTRACE_SINGLESTEP, pid, nullptr, nullptr )`
- `step out`: set breakpoint at the return address (it's on the stack)
- `step in`: keep stepping until we get to a new line
- `step over`: an exercise for the listener

Reading variables

- `DWARF info sections`
- down the rabbit hole
- poking the stack

Bells and whistles

- stack unwinding
- remote debugging
- expression evaluation
- multithreaded support

Conclusion

- all boils down to `ptrace` and `DWARF`
- a lot of parsing

Windows debugging?

We'll leave that to <REDACTED>

Thank you for listening.

Any questions?

`:wq`