History of debuggers

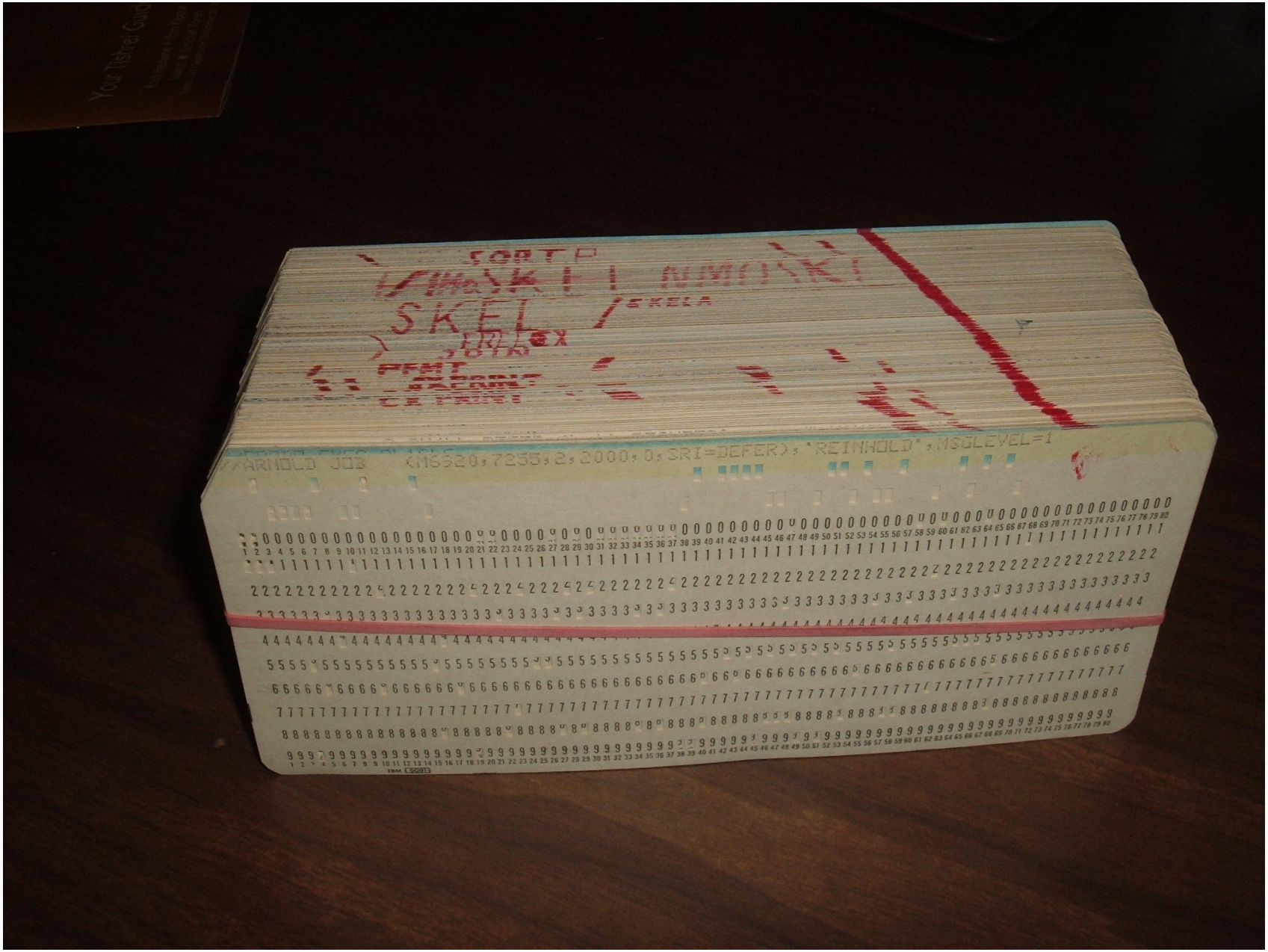
Credits of slides belong to Levente Kurusa <levex@linux.com>
Imperial College London

Single user machines

- One of the first computers in the world
- Small application was loaded at the top of the memory
  - single step
  - examine registers
  - read/write memory

TX-0 at MIT

Batch processing machines



Batch processing machines

Debugged by putting macro call in the punch card and generating:

- Snapshots *(register dump)*
- Core dumps *(contents of memory)*

printf

Then came CTSS *(Compatible Time-Sharing System)*, one of the first time-sharing operating systems!

Debugging suddenly became interactive.

printf-debugging

```
*ptr = 1337;
printf("Did we crash at line %d?\n", __LINE__);
*((int *) 0) = 1337;
printf("Did we crash at line %d?\n", __LINE__);
```

Unix-es

- The first version of UNIX had a debugger called, `DB`
- GNU had `GDB` and `LLDB`
- For Plan 9, `ADB` was created

These debuggers should be familiar!

Tracing processes

ptrace

Most debuggers heavily rely on a system call known as `ptrace`.

The prototype of `ptrace(2)`

```
#include <sys/ptrace.h>

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

Signals

How does `int a = 3, b = 0, c = a / b` result in a `SIGFPE`?

Signals

How does `int a = 3, b = 0, c = a / b` result in a `SIGFPE`?

1. Division by zero is noticed by the CPU

Signals

How does `int a = 3, b = 0, c = a / b` result in a `SIGFPE`?

1. Division by zero is noticed by the CPU
2. The CPU raises a divide-by-zero error (`#DE`)

Signals

How does `int a = 3, b = 0, c = a / b` result in a `SIGFPE`?

1. Division by zero is noticed by the CPU
2. The CPU raises a divide-by-zero error (`#DE`)
3. A handler in the kernel is eventually called

Signals

How does `int a = 3, b = 0, c = a / b` result in a `SIGFPE`?

1. Division by zero is noticed by the CPU
2. The CPU raises a divide-by-zero error (`#DE`)
3. A handler in the kernel is eventually called
4. The kernel sends a `SIGFPE` to the offending process

Signals

How does `int a = 3, b = 0, c = a / b` result in a `SIGFPE`?

1. Division by zero is noticed by the CPU
2. The CPU raises a divide-by-zero error (`#DE`)
3. A handler in the kernel is eventually called
4. The kernel sends a `SIGFPE` to the offending process
5. Your signal handler is called (or not if it is `SIGKILL`)

Implementation

- Enable tracing
- Run until system call
- Monitoring registers
- Single stepping
- Memory manipulation

Implementation

- Enable tracing
- Run until system call
- Monitoring registers
- Single stepping
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
- Monitoring registers
- Single stepping
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
- Monitoring registers
- Single stepping
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
  - `PTRACE_SYSCALL`
  - `PTRACE_SYSEMU`
- Monitoring registers
- Single stepping
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
  - `PTRACE_SYSCALL`
  - `PTRACE_SYSEMU`
- Monitoring registers
- Single stepping
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
  - `PTRACE_SYSCALL`
  - `PTRACE_SYSEMU`
- Monitoring registers
  - `PTRACE_PEEKUSER` / `PTRACE_POKEUSER`
  - `PTRACE_GETREGS` / `PTRACE_SETREGS`
- Single stepping
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
  - `PTRACE_SYSCALL`
  - `PTRACE_SYSEMU`
- Monitoring registers
  - `PTRACE_PEEKUSER` / `PTRACE_POKEUSER`
  - `PTRACE_GETREGS` / `PTRACE_SETREGS`
- Single stepping
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
  - `PTRACE_SYSCALL`
  - `PTRACE_SYSEMU`
- Monitoring registers
  - `PTRACE_PEEKUSER / PTRACE_POKEUSER`
  - `PTRACE_GETREGS / PTRACE_SETREGS`
- Single stepping
  - `PTRACE_SINGLESTEP`
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
  - `PTRACE_SYSCALL`
  - `PTRACE_SYSEMU`
- Monitoring registers
  - `PTRACE_PEEKUSER` / `PTRACE_POKEUSER`
  - `PTRACE_GETREGS` / `PTRACE_SETREGS`
- Single stepping
  - `PTRACE_SINGLESTEP`
- Memory manipulation

Implementation

- Enable tracing
  - `PTRACE_TRACEME`
- Run until system call
  - `PTRACE_SYSCALL`
  - `PTRACE_SYSEMU`
- Monitoring registers
  - `PTRACE_PEEKUSER / PTRACE_POKEUSER`
  - `PTRACE_GETREGS / PTRACE_SETREGS`
- Single stepping
  - `PTRACE_SINGLESTEP`
- Memory manipulation
  - `PTRACE_PEEKTEXT / PTRACE_POKETEXT`
  - `PTRACE_PEEKDATA / PTRACE_POKEDATA`

Architectural support

Interrupting a process

`PTRACE_SINGLESTEP`

Interrupting a process

PTRACE_SINGLESTEP

%EFLAGS

Interrupting a process

PTRACE_SINGLESTEP

%EFLAGS.TF

Interrupting a process

`PTRACE_SINGLESTEP`

Trap flag *(Sometimes referred to as "Trace flag")*

Interrupting a process

PTRACE_SINGLESTEP

Trap flag *(Sometimes referred to as "Trace flag")*

- After each instruction, trap into `#DB` interrupt
- The kernel delivers a `SIGTRAP`
- This fact is delivered via a wait-event to the debugger

Interrupting a process

Breakpoints

Interrupting a process

Breakpoints

- `ud2` (machine code: `0x0F 0x0B`)
  - Triggers `#UD` – Undefined instruction exception

Breakpoints

- `ud2` (machine code: `0x0F 0x0B`)
  - Triggers `#UD` – Undefined instruction exception
- `int $3` (machine code: `0xCC`)
  - Triggers `#BP` – Breakpoint exception

Breakpoint implementation

- Replace with "`int $3`"

Breakpoint implementation

- Replace with "`int $3`"
- Take note of the previous instruction

Breakpoint implementation

- Replace with "`int $3`"
- Take note of the previous instruction
- When the breakpoint is hit, replace with the previous instruction

Breakpoint implementation

- Replace with "`int $3`"
- Take note of the previous instruction
- When the breakpoint is hit, replace with the previous instruction
- Try executing the instruction again

Nota bene:

- Could have used "`ud2`"

Debug registers

- `DR0` – `DR3`: Linear addresses

Debug registers

- `DR0` – `DR3`: Linear addresses
- `DR6`: Debug control
  Contains bitmasks for:
  - 1, 2, 4 or 8 bytes monitored
  - Break on read, write, execute, or read+write

Debug registers

- `DR0` – `DR3`: Linear addresses
- `DR6`: Debug control
  Contains bitmasks for:
  - 1, 2, 4 or 8 bytes monitored
  - Break on read, write, execute, or read+write
- `DR7`: Debug status
  Bitmask showing which of `DR0`–`DR3` triggered the `#DB`

Debug registers

- `DR0` – `DR3`: Linear addresses
- `DR6`: Debug control
  Contains bitmasks for:
  - 1, 2, 4 or 8 bytes monitored
  - Break on read, write, execute, or read+write
- `DR7`: Debug status
  Bitmask showing which of `DR0`–`DR3` triggered the `#DB`
- `DR4` & `DR5`: Obsolete aliases to `DR6` & `DR7`