

PB173 – Binární programování Linux

X. Gdb a ladění výkonnosti

Jiri Slaby

Fakulta informatiky
Masarykova univerzita

22. 11. 2016

Minule

- Ladění funkčnosti
- ptrace, tracery, ...

- 1 gdb
 - Obrazy paměti

- 2 Ladění výkonnosti
 - Profilování
 - Pokrytí kódu

Sekce 1

`gdb`

- Pracuje s ELFem a DWARFem
 - DWARF není nutný, ale usnadňuje ladění
- Používá systémové volání ptrace
 - Sleduje proces
 - Upravuje vykonávání
 - Čte a mění paměť
 - ...
- Podpora Python skriptů
- Spuštění: `gdb --volby_gdb --args binarka --volby_binarky`
- Náповěda: `help`
- Manuál: GDB User Manual

- gcc generuje různé úrovně ladicích informací
 - -g1 (méně informací)
 - -g2 alias -g
 - -g3 (navíc informace o makrech)
 - -ggdb (i v kombinaci s -g<cislo>)
 - gcc použije i gdb rozšíření, pokud možno
 - **Demo:** makra, gdb a pb173-bin/10/mac.c
- S *optimalizacemi* se ladí se obtížněji
 - Proměnné zmizely, konstanty se propagovaly, ...
 - Ale bez optimalizací je pomalý kód
 - Někdy i nepřeložitelný
- V gcc 4.8 se objevilo -Og (00 < Og < 01)
 - Zapíná jen optimalizace, které (příliš) neztěžují ladění
 - Nezapíná samotnou generaci ladicích informací!
 - Používá se v kombinaci s -g...
 - **Demo:** gcc -Q --help=optimizers -Og

Operace s během

- Spustit: `run`
- Spustit po `main`: `start`
 - Nutné ladicí informace
- Přerušit: signálem (např. `Ctrl-c`)
- Pokračovat po návrat z funkce: `finish`
- Pokračovat: `continue`
- Krokovat s vnořením: `step` (po řádcích), `stepi` (po instrukcích)
- Krokovat bez vnoření: `next` (po řádcích), `nexti` (po instrukcích)
- Konec: `quit`

- Opakování předchozího příkazu: `ENTER`

Úkol: spusťte si `yes` v `gdb` a přerušte. Vyzkoušejte si `finish` a `continue`. Zkuste krokovat po instrukcích hlavní cyklus `yes`. Poté `quit`.

Výpis

- Zásobníku volání: `where`
 - Pohyb po rámcích zásobníku: `up`, `down`
- Zdrojového kódu: `list`
 - Nutné ladicí informace a zdrojový kód
- Assembleru: `disassemble`
- Ostatní: `info registers`, `info break`, `info macro`, ...

Většina příkazů má parametry – výpis lokací, funkcí, proměnných, registrů atd.

Výpisy v gdb

- 1 Přeložte si `pb173-bin/10/debug.c`
- 2 Spusťte `./debug`
 - Mělo by dojít k pádu
- 3 Otevřete debug v gdb
- 4 Spusťte (`run`)
- 5 Prohlédněte si, kde došlo k pádu (vypište):
 - Zásobník volání (`where`)
 - Místo pádu (`list` s parametrem `soubor:radek`)
 - Disassembly (`disassemble` s parametrem `adresa`)

- Dump (x) a print (p)
- Lze použít s formátováním (x/f a p/f, kde f je formát)
- Formát jsou tři hodnoty
 - 1 Počet vypisovaných hodnot
 - 2 Formát vypisovaných hodnot
 - x: hexa (podobně jako printf)
 - d: decimálně
 - u: decimálně bez znaménka
 - s: řetězec
 - a: adresa
 - i: instrukce
 - Ostatní: viz `help x`
 - 3 Velikost skupiny
 - b: 1 bajt (0x00 0x00 0x00 0x00)
 - h: 2 bajty (0x0000 0x0000)
 - w: 4 bajty (0x00000000)
 - g: 8 bajtů (...)

Výpisy

- 1 Otevřete si znovu program debug v gdb
- 2 Spusťte po pád (run)
- 3 Vypište
 - Adresu aktuální instrukce (p/a \$pc)
 - 20 instrukcí od aktuální (x/20i \$pc)
 - 32 hexa byte hodnot na zásobníku (\$sp)
 - 8 hexa giant hodnot na zásobníku
- 4 Pokuste se najít na zásobníku adresu, kam se vrátí po ret
- 5 Proměnnou b (print b)
- 6 Proměnnou argc v main (up a print argc)

- Zastavení vykonávání v definovaném bodě
- Příkaz: `break`
- Parametr
 - Adresa
 - Funkce
 - `soubor:radek`
 - ...

Úkol

- 1 Přidejte breakpoint na začátek `main` a spusťte (`run`)
 - Vyzkoušejte alespoň 2 způsoby
- 2 Krokujte až k pádu (`next`)
- 3 Zopakujte pro `step` a také `stepi`

- Spousta dalších informací v `info`
- `functions`, `types`, `variables`, `locals`
- `sources`, `line`, `macros`, `scope`
- `symbol`
- `threads`
- Další viz `help info`

Demo: `thread a pb173-bin/10/thread.c`

- `set, set variable`
- Registry procesoru: `$rax, $rbx, ...`
- Proměnné gdb: `$pc, $sp, ...`
- Vnitřní proměnné: `$ostatni`
 - Pro vlastní použití
- Paměť: `{typ}adresa`

Demo: `thread a parm`

- Jádro může při pádu zapsat obraz paměti („coredump”)
 - Výhoda: žádné zpomalení běhu
 - Nevýhoda: proces je mrtvý, nelze krokovat apod.
 - Nutno zvýšit limit velikosti (bývá 0): `ulimit -c`
 - Cíl, kam se ukládají: `sysctl kernel.core_pattern`
- Potom se dá analyzovat v gdb
 - gdb binarka `core_soubor`

Práce s obrazem paměti

- 1 Nastavte limit na obrazy na unlimited (`ulimit -c`)
- 2 Podívejte se, kam se obraz uloží (`sysctl kernel.core_pattern`)
- 3 Spusťte program z předchozího příkladu
 - Pád by měl vygenerovat coredump
- 4 Otevřete v gdb s obrazem paměti (`gdb ./debug path_to_core`)
- 5 Vypište si podobné informace jako předtím
 - `where`
 - `print b`
 - A podobně

Sekce 2

Ladění výkonosti

Co zabírá nejvíce času při vykonávání?

1 gprof

- Při překladu: `gcc -pg ...`
- Potom se program spustí a data se uloží na disk
- gprof data načte a zobrazí

2 perf

- Používá čítače v procesoru (nutná podpora CPU)
- Zahrnuje ve výpisech i chování jádra
- Umí pracovat s `gcc -pg` binárkami
- Spuštění: `perf record -g -- binarka --volby`
- Dokumentace: <http://perf.wiki.kernel.org>

Profilování

- 1 Projděte si `pb173-bin/10/prof.c`
- 2 Spusťte `prof_pg`
- 3 Prostudujte výstup `gprof -b prof_pg`

- 4 Nahrajte si záznam z `perf`
 - `perf record -g prof`
- 5 Zobrazte si výstup
 - `perf report -g`
- 6 Zopakujte `perf` pro `prof_pg`

Která část kódu se vůbec nevykonává?

- gcov
- Při překladu: `gcc --coverage ...`
- Potom se program spustí a data se uloží na disk
- gcov data načte a zapíše `cov.c.gcov`

Pokrytí kódu

- 1 Projděte si `pb173-bin/10/cov.c`
- 2 Spusťte `cov`
- 3 Prostudujte výstup `gcov cov`
- 4 Prostudujte `cov.c.gcov`