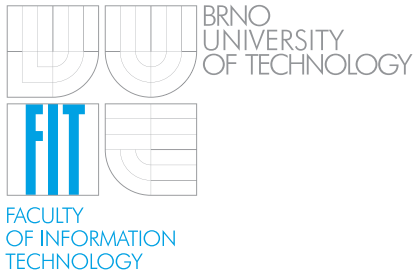
Chyby v programech, debugging, bug tracking, profiling

Ing. Zdeněk Materna

imaterna@fit.vutbr.cz

Ústav počítačové grafiky a multimédií
Fakulta informačních technologií

VUT v Brně



Osnova přednášky

- **Chyby v programech**
- **Debugging**
- **Bug tracking**
- **Profiling**

Chyby v programech

Slavné citáty

Při vývoji software se průměrně vyskytuje 15 až 50 chyb na 1000 řádků kódu.

Steve McConnel, autor knihy Dokonalý kód

Linusův zákon:
Je-li dost očí, jsou všechny chyby malé.

Eric S. Raymond

Odhalování chyb je dvakrát těžší než psaní kódu. Pokud tedy píšete kód jak nejlépe dovedete, z definice nejste schopni v něm najít chyby.

Brian Kernighan, spoluautor jazyka C

(Ne)Slavné bugy

- Mariner 1 (1962) ⇒ chybný přepis vzorce do programu
- Therac-25 (80. léta) ⇒ min. 5 mrtvých, přetečení registru
- Patriot (1991) ⇒ 28 mrtvých, chyba systémových hodin
- USS Yorktown (1997) ⇒ dělení nulou
- Mars Climate Orbiter (1999) ⇒ zmatek v jednotkách
- Y2K ⇒ rok pouze jako poslední dvě číslice
- ...

Chyby jsou nevyhnutelné

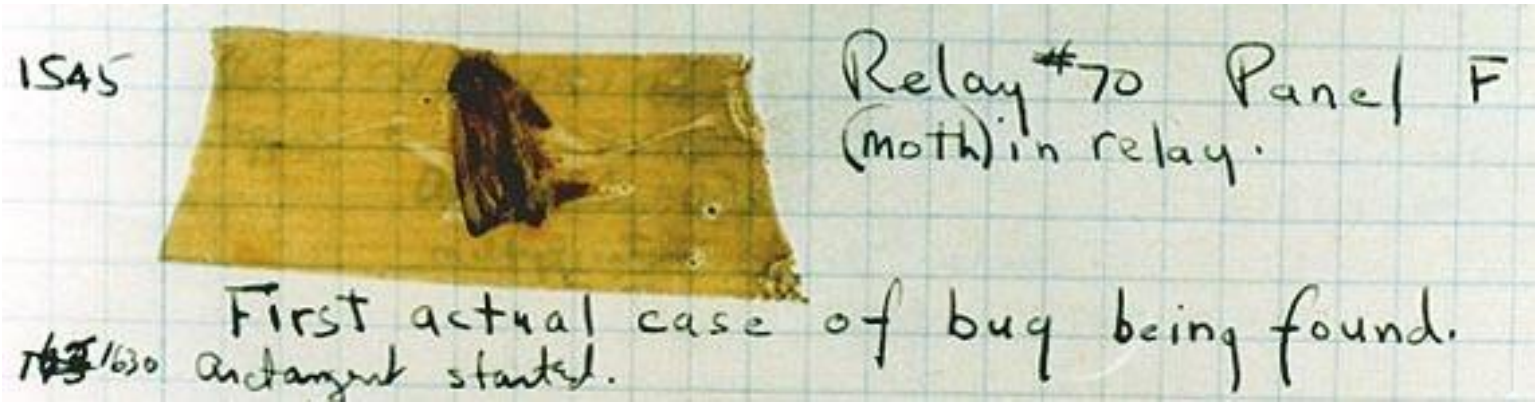
- vytváření bezchybného kódu je téměř nemožné
- pro většinu aplikací navíc není nutná absolutní bezchybnost
- výjimku tvoří zdravotnictví, vojenství, kosmický program...

Péče věnovaná testování kódu by měla být úměrná náročnosti, důležitosti a nebezpečnosti zakázky.

Metodologie: Extrémní programování (XP), Programování řízené testy (TDD), Lean development, Crystal, Adaptive Software Development, Dynamic Software Development Method...

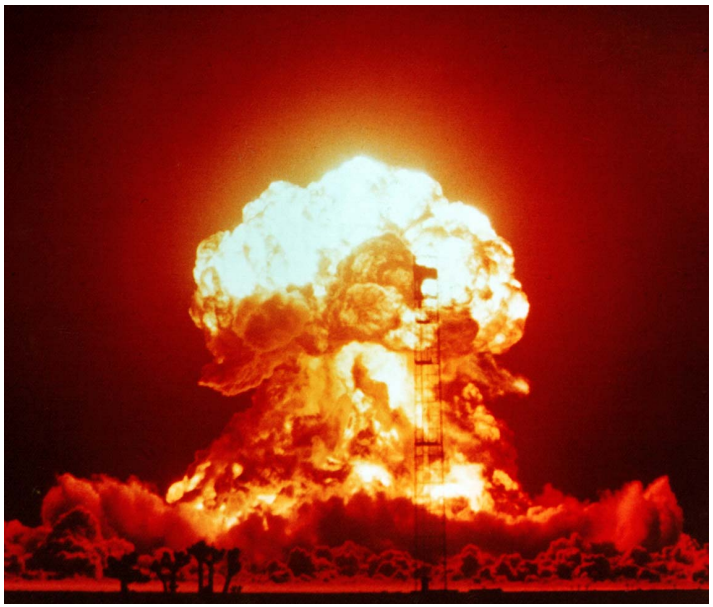
Kde se vzalo slovo bug

- používáno techniky od nepaměti
- i v češtině se říká, že má něco mouchy
- prokazatelně Edison slovo bug použil v dopise psaném roku 1878
- legenda o jeho vzniku roku 1947 (Mark II, můra v kontaktech relé



Základní dělení chyb

- **syntaktické**
  prohřešky proti gramatice jazyka
- **sémantické**
  program nedělá co má



Syntaktické chyby

- program vůbec nejde přeložit
- chybu odhalí již překladač, nebo chytrý editor
- nejčastěji překlep, chybějící středník, závorka...
- „přepínání“ programátora mezi různými jazyky
- může pomoci zvýrazňování syntaxe

```
unsigned int result
resolt = computeResult(getData(10);
return result;
```

Sémantické chyby

- obtížná automatická detekce
- právě těmito chybami se budeme v praxi většinou zabývat

Program může...

- spadnout za běhu (i laikovi je jasné, že je v programu chyba)
- skončit v nekonečném cyklu (nemusí být jasné jestli program něco dělá nebo ne)
- vracet chybné hodnoty (asi nejhorší možnost)
- atd.

```
if (tmp==2 && tmp==5) action();
```

Jiný pohled na typy chyb

- textové (překlepy)
- pády aplikace ⇒ úkol pro debugger
  - segmentation fault, buffer overflow...
- úniky paměti (memory leak) ⇒ valgrind
  - někde se zapomněla uvolnit alokovaná paměť
- problémy s výkonem ⇒ profilování
  - program nepracuje tak rychle jak bychom chtěli
- neočekávané chování
  - program dělá něco jiného než by měl (chyba v dokumentaci?)
- bezpečnostní problémy
  - neošetřené vstupy (SQL injection apod.), práva, souběh

Debugging

Postup při ladění programu

- nalezení chyby (vývojář, uživatel, tester)
- reprodukce selhání
- zjednodušení problému
- nalezení možných příčin
- zaměření se na pravděpodobné příčiny
- identifikace příčin
- oprava chyby
- změna dokumentace, testování

Důležitá je prevence!

Debugger

Nástroj pro nalézání chyb v programech umožňující:

- krokování programu
  - Step Into 
  - Step Over 
- sledování hodnot proměnných
- změnu hodnot proměnných
- zastavení běhu programu v definovaném místě (breakpoint)
  - nepodmíněné
  - podmíněné - zastavení pouze při splnění podmínky
  - datové (watchpoint) - zastavení kdykoliv se změní hodnota dané proměnné

Kompilované jazyky vyžadují pro použití debuggeru přidání ladících informací při překladu.

Nejznámější debuggery pro C

GDB - The GNU Project Debugger
Standardní debugger pro UNIX-like systémy. Má konzolové rozhraní. Podporuje mnoho jazyků, jako například Ada, C, C++, Objective-C, Pascal.

DDD - Data Display Debugger
Grafické uživatelské rozhraní ke konzolovým debuggerům (GDB, DBX, WDB, Ladebug aj.). Kromě běžných benefitů GUI umožňuje zobrazovat grafy na základě datových struktur v programu.

GDB

- textové rozhraní, velké množství funkcí
- HW i SW debuggování
- podporuje řadu procesorů a jazyků
- základem debuggeru v různých IDE (Eclipse)
- vzdálené debuggování (u embedded systémů)
- simulátor různých procesorů (bez periferií)
- breakpointy, watchpointy, krok zpět, podpora vláken...
- více viz. online dokumentace:
  http://sourceware.org/gdb/current/onlinedocs/gdb/

Ukázka použití GDB

```
user@host: /gdb my-program
GNU gdb (Ubuntu/Linaro 7.3-0ubuntu2) 7.3-2011.08
...
(gdb) run
Starting program: my-program
Program received signal SIGSEGV, Segmentation fault.
0x0804835b in my_print (message=0x0) at my-program.c:8
8 printf("%c", (*message)+i);
(gdb) backtrace
#0 0x0804835b in my_print (message=0x0) at my-program.c:8
#1 0x080483c3 in main () at my-program.c:20
```

Co z toho poznáme?

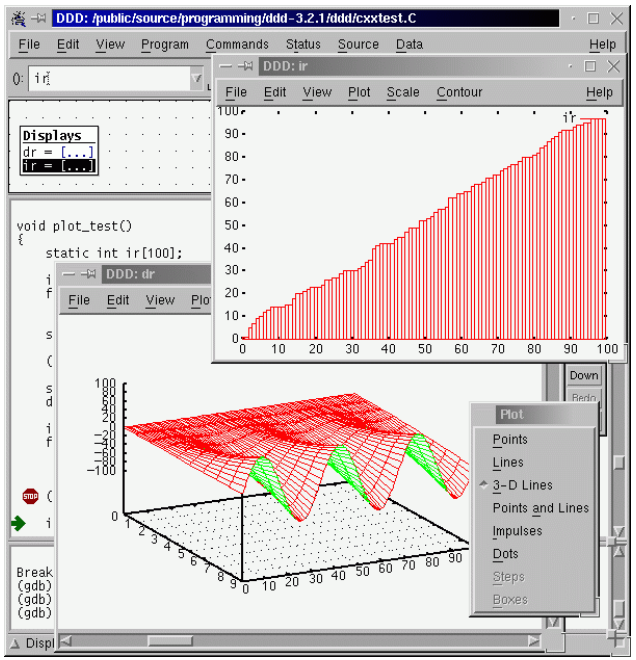
Základní příkazy GDB

- **help [command]** - funkce je snad jasná :-)
- **run** - spuštění programu
- **backtrace** - výpis zásobníku
- **brake funkce** - zastavení při zavolání funkce
- **brake program.c:6** - breakpoint na řádku 6
- **brake program.c:6 if i>=10** - podmíněný breakpoint
- **step** - postup o jeden krok (Step Into)
- **next** - jako step, ale nevstupuje do cyklů a funkcí (Step Over)
- **print prom** -výpis hodnoty proměnné
- **watch prom** - zastavení při změně hodnoty prom.

V konzoli GDB funguje (stejně jako v shellu) doplňování tabem, šipkami je možné procházet historii.

DDD

- nadstavba nad GDB, zobrazení datových struktur
- nepříliš aktivní vývoj
- tutoriál: http://www.youtube.com/watch?v=4aLaHTg45Sw



Volby překladače GCC pro ladění

- **-g** přidá ladící informace pro debugger
- **-ggdb** včetně rozšíření GDB

- **-Wall** zapnutí všech varování
- **-Wextra** povolení dalších varování
- **-pedantic** striktně vyžaduje dodržování normy ISO (přepínače -std=xx / -ansi)

http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

Všeobecné tipy pro snazší ladění

- je potřeba hledání chyb co nejvíce usnadnit
- jasné názvy proměnných, dodržování systému pojmenování
- přehledné (a konzistentní) formátování
- dostatečně komentovaný kód (použití např. Doxygen)
- sdružování částí kódu do skupin
- vyhnout se duplikaci kódu (DRY - Don't Repeat Yourself)
- snažit se o co nejméně úrovní zanoření (if...)
- držet se „best practices“ pro daný jazyk (např. Google Style Guide)

Časté chyby v programech (C/C++)

- neinicializované proměnné
- rozsahy pole
- přetečení nebo podtečení proměnné
- zaokrouhlování
- nekonečná smyčka
- záměna operátoru = a ==
- „Off-By-One Error“ (o jedno opakování cyklu více/méně)

Pěkný seznam nejčastějších chyb:
www.fit.vutbr.cz/∼martinek/clang/noerrors.html

Debugging bez debuggeru

Hledat chyby můžeme i bez pomoci speciálního softwaru...

- někdy to jinak nejde (např. v embedded systémech bez podpory debuggování)
- výpisy informující o průchodu daným místem programu
- vypisování stavu proměnných, chybové hlášky
- mohu použít makra preprocessoru (__FILE__, __LINE__ atd.)
- logování - možnost využít syslog
- výhody logování: umožní najít chybu zpětně, lze poslat emailem, podrobnost logování je možné měnit za běhu
- přidání kódu (logování) do programu může ovlivnit jeho chování!
- funkce assert (ověření platnosti podmínky)

Debugging bez debuggeru

Ukázka...

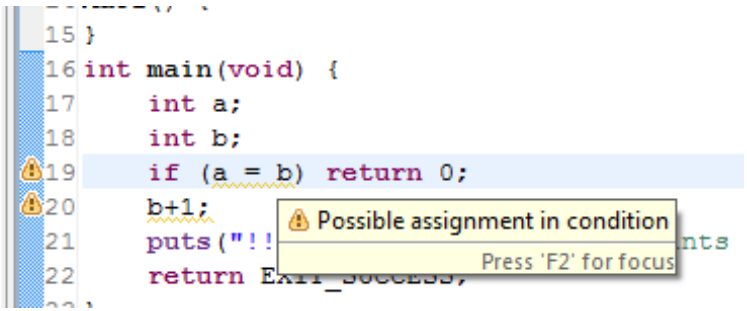
```
#define LOG(...) { char _bf[1024] = {0}; snprintf(_bf,
  sizeof(_bf)-1, __VA_ARGS__); syslog(LOG_INFO, "%s", _bf); }

#ifdef DEBUG
#define DBG(...) fprintf(stderr, "␣DBG(%s,␣%s(),␣%d):␣",
  __FILE__, __FUNCTION__, __LINE__); fprintf(stderr, __VA_ARGS__)
#else
#define DBG(...)
#endif
 ...
#define MAIN "[main()␣␣␣␣␣␣␣␣␣␣]␣"
LOG("%sSetting␣default␣port␣(%d)␣instead␣of␣%s",MAIN,DEF_COM_PORT,optarg);
```

Statická analýza kódu

- nástroje hledající problémy ve zdrojovém kódu bez jeho spuštění
- detekují nejrůznější programátorské chyby (neodhalí ale zdaleka všechny)
  - typová kontrola, neinicializace dat
  - kontrola indexace polí, dereference null ukazatele
  - přenositelnost konstrukcí apod.
- je možné doplnit vlastní pravidla
- častým problémem jsou false positive hlášení
- do jisté míry to samé provádějí i kompilátory

Nástroje pro C: Lint, Cppcheck, Mygcc, Codan (Eclipse CDT)



Shrnutí

- bugy mohou mít i exotické příčiny (bug v kompilátoru)
- při ladění v debuggeru se bug nemusí projevit
- pozor na optimalizace
- důležité je opravovat příčinu, ne následek
- chybám je lepší předcházet

Bug tracking

- aby mohl být problém opraven, je třeba informovat vývojáře
- více kódu, více chyb $\Rightarrow$ nutná sofistikovaná evidence

Nástroje pro bug tracking

- nejčastěji webové aplikace (interní vs. veřejné)
- nazýváno také „Issue Tracking System“, nebo „Ticket System“
- efektivní evidence a zpracování nahlášených chyb, požadavků
- přiřazování chyb k vývojářům
- sledování životního cyklu chyb
- možné provázání s verzovacím systémem
- závislosti chyb
- např. Bugzilla, Trac, Redmine, Mantis, Google Code

Třídění chyb

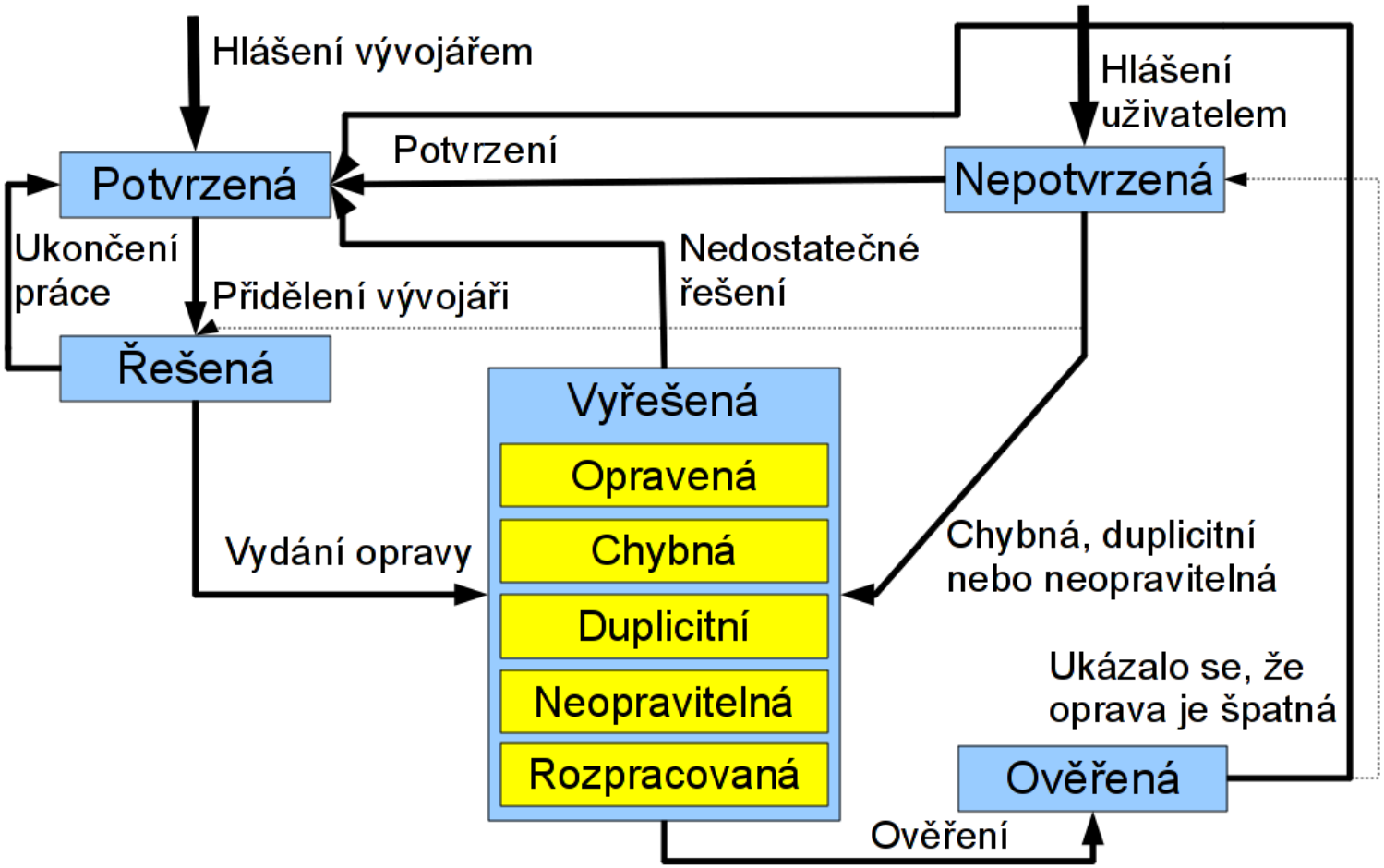
Podle závažnosti problému:

- blokující
- kritický
- významný
- normální
- méně důležitý
- triviální (drobnosti, chyby v překladu)
- rozšíření (požadavek na vylepšení)

Podle priority:

- chybám je podle závažnosti a množství výskytů přidělena priorita, podle které jsou zpracovávány

Životní cyklus chyby



Zpráva o chybě

Hlášení o chybě by mělo obsahovat co nejvíce relevantních informací. Typicky to je:

- jméno požadavku / chyby
- stručný souhrn problému
- verze sw
- platforma, verze OS, knihoven apod.
- postup jak chybu reprodukovat
- popis očekávaného chování
- popis chování, které nastalo
- kontakt

Profiling

Úvod

80% strojového času je stráveno vykonáváním 20% kódu.

- optimalizace rychlosti běhu programu
- nejjednodušší test: time ./muj-program
- profiler je nástroj, který zjistí, ve kterých funkcích a cyklech tráví program nejvíce času
- jen ty má potom cenu optimalizovat

Postup

1. krok ⇒ měření:
   - získání údajů o běžícím programu
   - ukládá se počet vyvolání jednotlivých funkcí, počty iterací v cyklech atd.
2. krok ⇒ analýza:
   - získání statistik z naměřených dat
   - grafické zobrazení pomocí tabulek a grafů
3. krok ⇒ optimalizace (pokud je skutečně třeba)

Přístupy

Statistický přístup (sampling):

- v periodickém přerušení se zaznamená aktuální poloha v programu
- menší přesnost, velmi málo zpomaluje běh programu
- příklady: AMD CodeAnalyst, Apple Shark, Intel VTune

Instrumentace programu (code instrumentation):

- do zdrojového kódu jsou přidána volání speciální funkce, která realizuje měření
- změny v rychlosti vykonávání kódu (zpomalení)
- některé bugy se díky tomu nemusí projevit, jiné se mohou objevit $\Rightarrow$ změna časování

Dále: interpretování programu (Valgrind), přístup založený na událostech (pro jazyky jako Java, Python apod.)

Výstupy

Flat profile:

- čas strávený v jednotlivých funkcích
- počet volání

Call graph:

- pro každou funkci - odkud byla volána, jaké další funkce volala
- jak dlouho které volání trvalo

Annotated source:

- ke každému řádku zdrojového kódu je přidán počet vykonání

gprof

- kombinuje statistický přístup s instrumentací
- čas strávený v jednotlivých funkcích se zjišťuje periodickým sledováním stavu programu (statistická chyba)
- čas běhu programu by měl být výrazně větší než je perioda vzorkování (0.01 s)
- počet volání funkcí se zjišťuje pomocí instrumentace (fce mcount)
- výstupem je flat profile a call graph

Pěkný tutoriál: www.cs.utah.edu/dept/old/texinfo/as/gprof.html

Ukázka práce s gprof

```
#gcc -Wall -Wextra -O2 -g -pg ./program.c -o program
#./program
#ls
program gmon.out
# gprof program gmon.out > output-file.txt

#mv gmon.out gmon.sum
#gprof -s program gmon.out gmon.sum

#gprof program gmon.sum > output-file-s.txt
```

- gmon.out je vytvořen v pracovním adresáři programu - je nutné mít právo pro zápis
- funkce nezkompilované pro profilování se ve výstupu neobjeví (knihovny apod.)

Flat profile - ukázka

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

- funkce mcount a profil jsou součástí každého profilu
- časy které nejsou mnohem větší než vzorkovací perioda nejsou příliš věrohodné

Call graph profile - ukázka

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]     100.0   0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]     100.0   0.00    0.05       1           main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]     100.0   0.00    0.05       1           report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]      59.8   0.01    0.02       8+472     <cycle 2 as a whole> [4]
                0.01    0.02     244+260         offtime <cycle 2> [7]
                0.00    0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

Sysprof

- statistický profiler pro Linux
- zkoumaný program nemusí být přeložen pro profilování (ale pro debuggování ano)
- podpora pro balíčky debugovacích informací (např. chromium-browser-dbg)
- skládá se z jaderného modulu a GUI
- modul v jádře periodicky ukládá stav zásobníku běžícího procesu
- umožňuje sledovat celý systém, ne jen jeden program

Shrnutí

- profiler napoví, které funkce se vyplatí optimalizovat
- je zbytečné zabývat se funkcí, která se vyvolá jednou a není časově kritická
- výstupem je: flat profile, call graph, annotated source
- program je nutné přeložit se speciální volbou (zpomalení)
- optimalizace kódu mohou zhoršit jeho čitelnost
- proto optimalizujeme především algoritmy!
- údaje profileru mohou být zatíženy statistickou chybou
- klasickým nástrojem je gprof, existují i pokročilejší (ale složitější)
- profilování je vhodné pro rozsáhlejší programy
- může pomoci s odhalením bugů (více/méně volání funkce než je žádáno), heisenbugs

Děkuji za pozornost