

# Trasování v Linuxu

---

- Autor Pavel Tišnovský, Red Hat
- Email <ptisnovs 0x40 redhat 0x2e com>
- Datum 2019-10-05

## Obsah přednášky (1)

---

- ▶ Trasování a ladění nativních aplikací v Linuxu
- ▶ Nástroj „ltrace“
  - ◆ ukázka použití nástroje ltrace
  - ◆ časová razítka volání
  - ◆ zjištění doby trvání volané funkce a filtrace výstupu
  - ◆ filtrace informací
  - ◆ zjištění statistiky volaných funkcí
  - ◆ připojení k běžící aplikaci
  - ◆ další možnosti nabízené nástrojem ltrace

## Obsah přednášky (2)

---

- ▶ Nástroj „strace“
  - ◆ ukázka použití nástroje strace
  - ◆ zobrazení tabulky volaných funkcí
  - ◆ časy volání syscallů
  - ◆ časy trvání volaných syscallů
  - ◆ zobrazení tabulky volaných syscallů
  - ◆ setřídění tabulky podle zvoleného sloupce
  - ◆ další možnosti nabízené nástrojem strace
- ▶ Nástroj „DTrace“
  - ◆ základní informace o nástroji DTrace
  - ◆ příklad naprogramované sondy

## Obsah přednášky (3)

---

- ▶ Nástroj „SystemTap“
  - ◆ základní informace o nástroji SystemTap
  - ◆ instalace, pomocný nástroj stap-prep
  - ◆ ukázky použití SystemTap (různé sondy)
  - ◆ backtrace u vybraných funkcí/syscallů
  - ◆ získání a výpis statisických informací
  - ◆ tisk histogramů
- ▶ GNU Debugger
  - ◆ rozhraní s příkazovým řádkem
  - ◆ tracepointy
  - ◆ atd.

## Obsah přednášky (4)

---

- ▶ BPF (Berkeley Packet Filter) a eBPF
  - ◆ front endy pro eBPF
- ▶ Nástroj bpftrace
  - ◆ základní informace o nástroji bpftrace
  - ◆ příklady použití
- ▶ Odkazy na další informační zdroje

## Poznámka: zvýrazněné řádky

- ▶ Na slajdech budeme používat tyto barvy

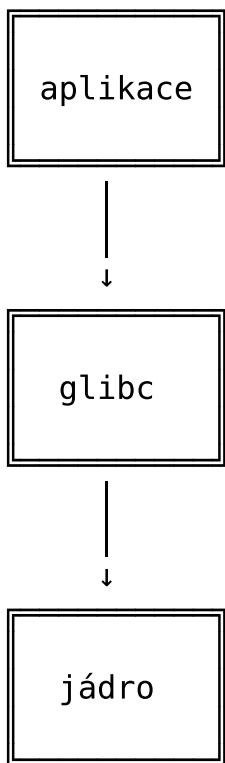
\$ příkaz\_spuštěný\_běžným\_uživatelem

# příkaz\_spuštěný\_rootem

↓

výstup (tisk) je většinou zobrazen pod šipkou

## Trasování a ladění nativních aplikací v Linuxu

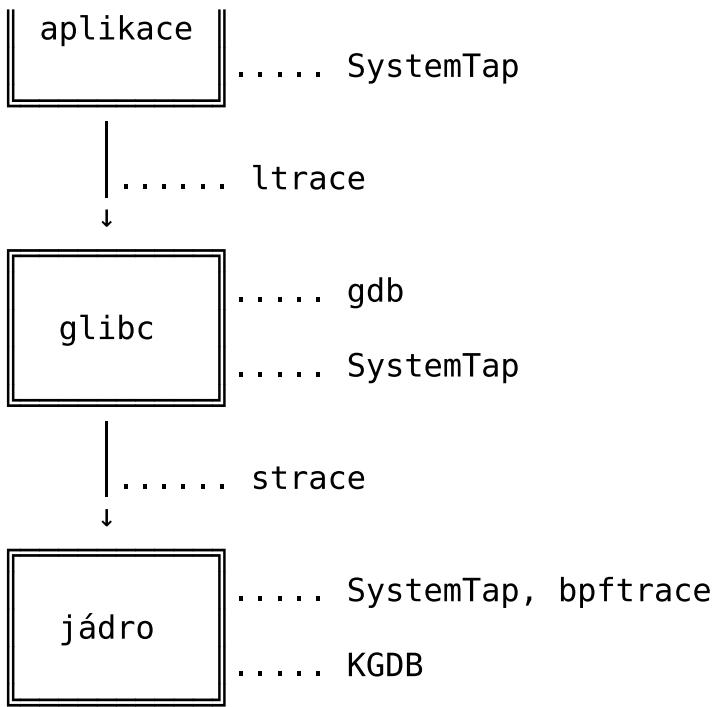


## Trasování a ladění nativních aplikací v Linuxu

- ▶ Systémová volání
  - ◆ strace
- ▶ Ladění a trasování aplikací i knihoven
  - ◆ ltrace
  - ◆ GNU Debugger
- ▶ Stav aplikací
  - ◆ SystemTap
- ▶ Front-end ke zdrojům dat z kernelu
  - ◆ SystemTap
  - ◆ ftrace
  - ◆ perf
  - ◆ catapult
  - ◆ bpftrace
  - ◆ ...

## Trasování a ladění nativních aplikací v Linuxu





## Utilita ltrace

---

- ▶ Trasování volání knihovních funkcí
- ▶ Vyhledání problémových či pomalých částí kódu
- ▶ Zjištění způsobu alokace a dealokace paměti
- ▶ Či pouze jednoduché trasování aplikace

## Utilita ltrace

---

- ▶ Ovládání z příkazové řádky

```
$ ltrace --help
  ↓
Usage: ltrace [option ...] [command [arg ...]]
Trace library calls of a given program.

...
...
...
-a, --align=COLUMN  align return values in a specific column.
-A ARRAYLEN         maximum number of array elements to print.
```

## Zavolání nástroje ltrace

---

- ▶ Zavolání ltrace bez parametrů

```
$ ltrace jméno_binární_aplikace parametry_aplikace
```

- ▶ Zavolání ltrace s předáním parametrů

```
$ ltrace parametry_ltrace jméno_binární_aplikace parametry_aplikace
```

## Testovaný zdrojový kód - program typu „Hello world“

---

### Ukázka použití utility ltrace

---

```
$ ltrace ./hello
```

```
↓  
__libc_start_main(0x40053d, 1, 0x7ffd2e1f5f8, 0x400560 <unfinished ...>  
puts("Hello world!"Hello world!  
)  
13  
+++ exited (status 0) +++
```

## Testovaný zdrojový kód - bitmapa s náhodným obsahem

### Ukázka použití utility ltrace

```
$ ltrace ./random_bitmap  
↓  
__libc_start_main(0x400bf4, 1, 0x7fff18e8d758, 0x400c70 <unfinished ...>  
puts("processing:") = 12  
malloc(16) = 0xa11010  
malloc(921600) = 0x7f45fdb46010  
memset(0x7f45fdb46010, '\0', 921600) = 0x7f45fdb46010  
memset(0x7f45fdb46010, '\0', 921600) = 0x7f45fdb46010  
open("/dev/urandom", 0, 037560470000) = 3  
read(3, "\270\325\332\340r8\304\354\306A\001S", 1920) = 1920  
...  
...  
...  
vynecháno přibližně 480 řádků  
...  
...  
...  
...  
close(3) = 0  
fopen("random.bmp", "wb") = 0xa11030  
fwrite("BMF", 54, 1, 0xa11030) = 1  
printf("%d pixels written\n", 307200) = 22  
fwrite("\270\325\332\340r8\304\354\306A\001S", 921600, 1, 0xa11030) = 1  
fclose(0xa11030) = 0  
puts("done!\n") = 7  
+++ exited (status 0) +++
```

## Poslední testovaný zdrojový kód - vykreslení fraktálu

### Ukázka použití utility ltrace

```
$ ltrace ./fractal_renderer  
↓  
__libc_start_main(0x400ec2, 1, 0x7fff0293d968, 0x400f60 <unfinished ...>  
puts("processing:") = 12  
malloc(16) = 0xf7f010  
malloc(921600) = 0x7fb63dc5a010  
memset(0x7fb63dc5a010, '\0', 921600) = 0x7fb63dc5a010  
memset(0x7fb63dc5a010, '\0', 921600) = 0x7fb63dc5a010  
sin(0xf7f010, 1000, 0, 0xf7f010) = 0x3fa11111  
...  
...
```

- zde můžeme vidět, že jasně dominuje volání funkce `sin` z knihovny `libm`

Časová razítka volání funkcí

► Někdy nám může dostačovat sekundová přesnost

```
$ ltrace -t ./hello
```

↓

```
20:17:06 __libc_start_main(0x40053d, 1, 0x7fff74550708, 0x400560
<unfinished ...>
20:17:06 puts("Hello world!") = 13
20:17:06 +++ exited (status 0) +++
```

## Větší přesnost časových razítek

- ▶ Většinou však budeme vyžadovat větší přesnost časových razítek  
\$ ltrace -tt ./hello

**↓**

1

```
20:49:54.521242 __libc_start_main(0x400550, 1, 0x7ffffe000000, 0x400500
<unfinished ...>
20:49:54.521862 puts("Hello world!")
20:49:54.522886 +++ exited (status 0) +++
```

Použití časových razítek vypsaných ve formátu UNIX time

```
$ ltrace -ttt ./hello
```

↓

1569690376.636835 \_\_libc\_start\_main(0x8049780, 1, 0xbfa41934, 0x8052c50, 0x8052c40) <unfinished ...>

100

1

3

1569690376.672622 +++ exited (status 0) +++

## Zjištění doby trvání mezi voláními funkce a filtrace výstupu

- volba -r zajistí výpis relativních časů (offsetů)

```
$ ltrace -r ./hello
```

↓

```
0.000000 __libc_start_main(0x40053d, 1, 0x7ffffdd8d7208, 0x400560
<unfinished ...>
```

```
0.000479 puts("Hello world!") = 13
0.000985 +++ exited (status 0) +++
```

## Filtrace informací o vybrané funkci/funkcích

---

- Můžeme snadno vybrat ty funkce, o nichž potřebujeme získat další informace

```
$ ltrace -e malloc+free+open+close ./random_bitmap
      ↓
processing:
random_bitmap->malloc(16) = 0x2137010
random_bitmap->malloc(921600) = 0x7f4fb333c010
random_bitmap->open("/dev/urandom", 0, 026320350000) = 3
random_bitmap->close(3) = 0
307200 pixels written
done!

+++ exited (status 0) +++
```

## Odstranění informací o funkcích, které nás nezajímají

---

- Povšimněte si znaku - před jménem funkce

```
$ ltrace -e -read ./random_bitmap
      ↓
random_bitmap->__libc_start_main(0x400bf4, 1, 0x7fff7c9456e8, 0x400c70
<unfinished ...>
random_bitmap->puts("processing:") = 12
random_bitmap->malloc(16 <unfinished ...>
libc.so.6->(0x7f7140a97bd0, 0x7fff7c945550, 0x7fff7c945540, 0) =
0x7f7140ffa4c0
<... malloc resumed> )
0x1ee3010
random_bitmap->malloc(921600) =
0x7f7140ef3010
random_bitmap->memset(0x7f7140ef3010, '\0', 921600) =
0x7f7140ef3010
random_bitmap->memset(0x7f7140ef3010, '\0', 921600) =
0x7f7140ef3010
random_bitmap->open("/dev/urandom", 0, 010077240000) = 3
random_bitmap->close(3) = 0
random_bitmap->fopen("random.bmp", "wb" <unfinished ...>
libc.so.6->memalign(568, 0x400cf4, 1, 0) =
0x1ee3030
<... fopen resumed> )
0x1ee3030
random_bitmap->fwrite("BMF", 54, 1, 0x1ee3030) = 1
random_bitmap->printf("%d pixels written\n", 307200307200 pixels written
) = 22
random_bitmap->fwrite("Z\226\314H7\316\301\306\340\367iI\3240\332|
\005\035![BL\325u\271\335L", 921600, 1, 0x1ee3030) = 1
random_bitmap->fclose(0x1ee3030 <unfinished ...>
libc.so.6->(0x1ee3030, 0, 0x1ee3110, 0xfbcd000c) = 1
<... fclose resumed> )
random_bitmap->puts("done!\n") = 7
```

```
libc.so.6->_dl_find_dso_for_object(0x7f7140dd1d90, 0x7f7140dd26c8, 1,  
-1) = 0x7f7140fd5690  
+++ exited (status 0) +++
```

### Zjištění statistiky volaných funkcí

---

```
► volba -c  
$ ltrace -c ./hello  
↓  
Hello world!
```

% time	seconds	usecs/call	calls	function
100.00	0.000455	455	1	puts
100.00	0.000455		1	total

```
► zde nedošlo k většímu překvapení
```

### Zjištění statistiky volaných funkcí (2)

---

```
► Nyní vyzkoušíme přepínač -c společně s programem pro generování  
bitmapy
```

```
$ ltrace -c ./random_bitmap  
↓
```

% time	seconds	usecs/call	calls	function
97.81	0.228395	475	480	read
0.73	0.001701	850	2	fwrite
0.44	0.001032	516	2	memset
0.29	0.000670	335	2	puts
0.24	0.000559	279	2	malloc
0.13	0.000299	299	1	fclose
0.10	0.000245	245	1	fopen
0.10	0.000231	231	1	printf
0.09	0.000214	214	1	open
0.07	0.000174	174	1	close
100.00	0.233520		493	total

```
► Povšimněte si, že funkce read() se volala přesně 480x  
◆ to odpovídá zdrojovému kódu
```

### Zjištění statistiky volaných funkcí (3)

---

```
► Nakonec vyzkoušíme přepínač -c společně s programem pro vykreslení  
fraktálu
```

```
$ ltrace -c ./fractal_renderer  
↓
```

% time	seconds	usecs/call	calls	function
99.99	72.237624	117	614400	sin
0.00	0.001307	653	2	fwrite
0.00	0.000695	347	2	puts

```

0.00 0.000672    336      2 memset
0.00 0.000487    487      1 fclose
0.00 0.000409    409      1 fopen
0.00 0.000338    169      2 malloc
0.00 0.000160    160      1 printf
-----
100.00 72.241692           614411 total

```

- ▶ Podle očekávání
  - ◆ nejvíce času se celkově (kumulativně) strávilo ve funkci sin
  - ◆ (ovšem z volaných funkcí je nejrychlejší)

### Připojení k běžící aplikaci

- ▶ Terminál číslo 1

```
$ bash
$ echo $$
12345
```

- ▶ Terminál číslo 2

```
$ ltrace -p 12345
```

- ▶ Terminál číslo 1

[Enter]

### Další možnosti nabízené nástrojem ltrace

- ▶ Dekódování (demangle) jmen metod a funkcí z C++
- ▶ Sledování procesů, které vznikly zavoláním
  - ◆ fork()
  - ◆ clone()
- ▶ Odsazení výstupu u funkcí volaných z jiných funkcí
  - ◆ podstatným způsobem může zajistit lepší čitelnost
  - ◆ (dnes neuvedeno na příkladech - malá plocha slajdů)

### Utilita strace

- ▶ Zjištění (trasování) systémových volání
- ▶ Nezávisle na tom, kde volání vzniklo (aplikace, knihovna)
  - ◆ typicky v knihovně glibc
- ▶ Aplikace se spustí přes strace
  - ◆ prakticky stejně použití, jako u nástroje ltrace
- ▶ Alternativně připojení k běžící aplikaci přes -p{pid}
  - ◆ opět totožné s ltrace
- ▶ Význam mnoha přepínačů shodný s ltrace

### Ukázka použití utility strace

```
$ strace ./hello
↓
execve("./hello", ["./hello"], /* 53 vars */) = 0
brk(0)                                     = 0xa52000
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (No such file or
directory)
```

## Ukázka použití utility strace

- ▶ U každého systémového volání jsou uvedeny parametry
  - ▶ Inteligentní nahrazení číselných konstant za symbolické konstanty
    - ◆ PROT\_READ atd.
  - ▶ Vypisuje se i návratová hodnota z volaného bloku
    - ◆ opět s využitím symbolické konstanty, kde to dává smysl
    - ◆ alternativně se vypíše i zpráva "No such file or directory"

### Ukázka použití utility strace

- ```
► Filtrace syscallů  
$ strace -e trace=open,close whoami
```

## Časy volání syscallů

- Stejné přepínače jako v případě nástroje ltrace
- ```
$ strace -t whoami
$ strace -tt whoami
$ strace -ttt whoami
$ strace -r whoami
```

## Časy trvání volaných syscallů

```
$ strace -r whoami
close(1) = 0 <0.000046>
munmap(0xb7c4a000, 4096) = 0 <0.000071>
close(2) = 0 <0.000044>
exit_group(0) = ?
```

## Zobrazení tabulky volaných syscallů

% time	seconds	usecs/call	calls	errors	syscall
100.00	0.000087	12	7		munmap
0.00	0.000000	0	10		read
0.00	0.000000	0	1		write
0.00	0.000000	0	39	13	open
0.00	0.000000	0	30		close
0.00	0.000000	0	1		execve
0.00	0.000000	0	7	7	access
0.00	0.000000	0	3		brk
0.00	0.000000	0	8		mprotect
0.00	0.000000	0	2		_llseek
0.00	0.000000	0	34		mmap2
0.00	0.000000	0	26		fstat64
0.00	0.000000	0	1		geteuid32
0.00	0.000000	0	1		fcntl64
0.00	0.000000	0	1		set_thread_area
0.00	0.000000	0	2		socket
0.00	0.000000	0	2	2	connect
100.00	0.000087		175	22	total

## Setřízení tabulky podle zvoleného sloupce

% time	seconds	usecs/call	calls	errors	syscall
44.44	0.000088	2	39	13	open
0.00	0.000000	0	34		mmap2
0.00	0.000000	0	30		close
0.00	0.000000	0	26		fstat64
0.00	0.000000	0	10		read
0.00	0.000000	0	8		mprotect
55.56	0.000110	16	7	7	access
0.00	0.000000	0	7		munmap

0.00	0.000000	0	3	brk
0.00	0.000000	0	2	_llseek
0.00	0.000000	0	2	socket
0.00	0.000000	0	2	connect
0.00	0.000000	0	1	write
0.00	0.000000	0	1	execve
0.00	0.000000	0	1	geteuid32
0.00	0.000000	0	1	fcntl64
0.00	0.000000	0	1	set_thread_area
<hr/>				
100.00	0.000198	175	22	total

## Další možnosti nabízené nástrojem strace

---

- ▶ Dekódování (demangle) jmen metod a funkcí z C++
- ▶ Sledování procesů, které vznikly zavoláním
  - ◆ fork()

## DTrace

---

- ▶ D=dynamic
- ▶ Jeden z pokročilejších nástrojů pro sledování a trasování
- ▶ Dostupné na
  - ◆ Solaris/OpenSolaris
  - ◆ FreeBSD
  - ◆ macOS
- ▶ Původně nikoli na Linuxu!
  - ◆ licenční problémy
  - ◆ první verze pro Linux v roce 2008
  - ◆ verze pro Windows 10 letos
  - ◆ portace pro QNX

## Základní informace o nástroji DTrace

---

- ▶ Takzvané sondy (probe)
  - ◆ krátké skripty spuštěné při vzniku události
  - ◆ v jádře
  - ◆ v aplikaci
- ▶ Samotná sonda má přístup k zásobníkovému rámcí
  - ◆ lokální proměnné
  - ◆ parametry
- ▶ Pro zápis sond se používá jazyk D
  - ◆ ovšem ne D z <http://dlang.org>

## Příklad naprogramované sondy

---

```
#pragma D option flowindent

syscall::write:entry
/pid == $target/
{
    printf("Written %d bytes\n", arg2);
}
```

## SystemTap

---

- ▶ Použití
  - ◆ události v kernelu
  - ◆ události v userspace
  - ◆ samozřejmě i syscalls
- ▶ Základní koncept podobný DTrace
  - ◆ sondy
  - ◆ deklarace, kdy se má sonda spustit
- ▶ Instalace
  - ◆ záleží na distribuci, obecně složitější
  - ◆ pomocný nástroj `stap-prep`
    - poradí, které balíčky nainstalovat
    - někdy je i nainstaluje

## Kontrola instalace SystemTapu

---

- ▶ Skript převzatý z dokumentace SystemTapu
  - ◆ přepínač `-v`: verbose režim
  - ◆ přepínač `-e`: zadání skripti na příkazovém řádku

```
# stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'  
↓  
Pass 1: parsed user script and 118 library scripts using 233764virt/  
36948res/7372shr/29452data kb, in 200usr/30sys/471real ms.  
Pass 2: analyzed script: 1 probe, 1 function, 4 embeds, 0 globals using  
372776virt/170852res/8824shr/168464data kb, in 3560usr/400sys/4282real  
ms.  
Pass 3: translated to C into "/tmp/stapW5VWo/  
stap_5ec9a356f694741098a16e111db111c9_1654_src.c" using 372776virt/  
171052res/9024shr/168464data kb, in 10usr/10sys/19real ms.  
Pass 4: compiled C into "stap_5ec9a356f694741098a16e111db111c9_1654.ko"  
in 16650usr/3190sys/20283real ms.  
Pass 5: starting run.  
read performed  
Pass 5: run completed in 20usr/60sys/440real ms.
```

## Vybrané typy sond (triviální příklady)

---

- ▶ begin
  - ◆ spuštěna po inicializaci SystemTapu
- ▶ end
  - ◆ opak předchozí sondy
- ▶ oneshot
  - ◆ spuštěna jen jedenkrát
- ▶ never
  - ◆ nikdy nebude spuštěna

## Vybrané typy sond (složitější sondy)

---

- ▶ `process(jméno).begin`
- ▶ `process(jméno).end`
- ▶ `process(PID).begin`
- ▶ `process(PID).end`

- ◆ spuštění či naopak ukončení procesu
- syscall.open
- syscall.write
  - ◆ ruzné typy syscallů
- process().function().call

### Výpis všech typů sond, filtrace syscallů

---

```
# stap --dump-probe-types
# stap --dump-probe-aliases |grep ^syscall
```

### Ukázky použití SystemTapu

---

```
► Sonda, která pouze vypíše zprávu a ukončí se
probe begin
{
    printf ("hello world\n")
    exit ()
}
```

```
► Spuštění
$ stap hello.stp
      ↓
hello world
```

Lze použít i středníky atd.

---

```
► Může být výhodné - některé editory lze zapnout do „C“ režimu
```

```
probe begin
{
    printf ("hello world\n");
    exit ();
}
```

### Funkce println() namísto složitější printf()

---

```
probe begin
{
    println("Hello world!");
    exit();
}
```

### Zápis sondy přímo na příkazovém řádku

---

```
# stap -e 'probe begin {printf("Hello world!\n");exit();}'
```

### Sonda typu oneshot

---

```
► Provede se přesně jedenkrát
```

```
probe oneshot
{
    println("Hello world!");
```

}

## Více sond stejného typu s uvedením pořadí

- Libovolné celočíselné hodnoty

```
probe begin(3)
{
    println(" !");
}
probe begin(100)
{
    exit();
}
probe begin(2)
{
    print(" world");
}
probe begin(1)
{
    print("Hello");
}
```

## Ukázky použití SystemTapu

- Sonda, která vypíše ID uživatele

```
probe begin
{
    printf ("hello %d\n", uid())
    exit ()
}
```

- Spuštění (pod rootem)

```
# stap hello2.stp
      ↓
hello 0
```

## Ukázky použití SystemTapu

- Sonda reagující každou sekundu na časovač

```
probe timer.ms(1000)
{
    printf ("tiktak\n")
```

- Spuštění (pod rootem)

```
# stap clock.stp
      ↓
tiktak
tiktak
tiktak
```

## Ukázky použití SystemTapu

---

- ▶ Reakce na spuštění programu nazvaného „hello“
- ▶ Reakce na ukončení tohoto programu
- ▶ Reakce na vstup do funkce main

```
probe process("hello").begin
{
    printf ("started\n")
}
probe process("hello").end
{
    printf ("finished\n")
}
probe process("hello").function("main")
{
    printf ("main function\n")
}
```

## Výpis PID každého spuštěného procesu „ls“

---

```
probe begin
{
    println("STAP prepared");
}
probe process("ls").begin
{
    printf("ls with PID=%d started\n", pid());
}
probe process("ls").end
{
    printf("ls with PID=%d finished\n", pid());
}
```

## Ukázky použití SystemTapu

---

- ▶ Registrace a spuštění sondy  
# stap process.stp
- ▶ Spuštění programu ./hello v jiném terminálu  
\$ ./hello
- ▶ Zprávy vypsané sondou
  - started
  - main function
  - finished

## Ukázky použití SystemTapu

---

- ▶ Výpis informací o všech otevřených souborech/zařízeních  
probe syscall.open
{
 filename = user\_string(\$filename);
 printf("ls opened file %s\n", filename);

```
}
```

► Ukázka výstupu  
STAP prepared  
ls with PID=22086 started  
ls opened file /etc/ld.so.cache  
ls opened file /lib64/libselinux.so.1  
ls opened file /lib64/libcap.so.2  
ls opened file /lib64/libacl.so.1  
ls opened file /lib64/libc.so.6  
ls opened file /lib64/libpcre.so.1  
ls opened file /lib64/libdl.so.2  
ls opened file /lib64/libattr.so.1  
ls opened file /lib64/libpthread.so.0  
ls opened file /usr/lib/locale/locale-archive  
ls with PID=22086 finished

► Lze porovnat s:

```
$ strace -e trace=open ls
```

## Ukázky použití SystemTapu

- Použití správného PID
- Globální proměnné ve skriptech
  - ◆ obsah těchto proměnných se automaticky vypíše na konci
  - ◆ pokud se do proměnné jen zapisuje

```
global pid=-1
probe begin
{
    println("STAP prepared");
}
probe process("ls").begin
{
    pid=pid()
    printf("ls with PID=%d started\n", pid());
}
probe process("ls").end
{
    printf("ls with PID=%d finished\n", pid());
}
probe syscall.open
{
    filename = user_string($filename);
    if (pid==pid()) {
        printf("ls opened file %s\n", filename);
    }
}
```

## Použití automaticky vypisovaných globálních proměnných

```
global read_count=0
global write_count=0
global read_bytes=0
```

```

global write_bytes=0
global unused_variable=0

probe syscall.open
{
    filename = user_string($filename);
    printf("ls opened file %s\n", filename);
}
probe syscall.read
{
    bytes=$count
    into=$fd
    read_bytes += bytes
    printf("read %d bytes from file descriptor %d\n", bytes, into);
    read_count++
    if (read_count>10000) exit()
}
probe syscall.write
{
    bytes=$count
    into=$fd
    write_bytes += bytes
    printf("write %d bytes to file descriptor %d\n", bytes, into);
    write_count++
    if (write_count>10000) exit()
}

```

### Sledování dalších syscallů

---

- ▶ U jednotlivých syscallů máme k dispozici další informace
  - ◆ u write je to: buf, dount, fd

```

global pid=-1
probe begin
{
    println("STAP prepared");
}
probe process("ls").begin
{
    pid=pid()
    printf("ls with PID=%d started\n", pid());
}
probe process("ls").end
{
    printf("ls with PID=%d finished\n", pid());
}
probe syscall.open
{
    filename = user_string($filename);
    if (pid==pid()) {
        printf("ls opened file %s\n", filename);
    }
}
probe syscall.write

```

```
{
    bytes=$count
    into=$fd
    if (pid==pid()) {
        printf("write %d bytes to file descriptor %d\n", bytes, into);
    }
}
```

► Výsledek (zkrácený)

```
write 67 bytes to file descriptor 1
write 55 bytes to file descriptor 1
```

### Opis řetězců zapisovaných do souborů

```
global pid=-1

probe begin
{
    println("STAP prepared");
}
probe process("ls").begin
{
    pid=pid()
    printf("ls with PID=%d started\n", pid());
}
probe process("ls").end
{
    printf("ls with PID=%d finished\n", pid());
}
probe syscall.open
{
    filename = user_string($filename);
    if (pid==pid()) {
        printf("ls opened file %s\n", filename);
    }
}
probe syscall.write
{
    bytes = $count;
    into = $fd;
    msg = user_string_n($buf, bytes);
    if (pid==pid()) {
        printf("write %d bytes to file descriptor %d\n", bytes, into);
        println(msg);
    }
}
```

### Specifikace jména procesu na příkazové řádce

► Volání

```
# stap check.stp whoami
```

► Sondy

```
probe process(@1).begin
```

```

{
    pid=pid()
    printf("ls with PID=%d started\n", pid());
}
probe process(@1).end
{
    printf("ls with PID=%d finished\n", pid());
}

```

## Výpis volaných funkcí

---

- Použití žolíkového znaku „\*“

```

probe process("ls").function("*").call
{
    println(ppfunc());
}

```

## Výpis backtrace zvolené volané funkce (zadání sondy)

---

- Použití explicitního jména funkce

```

probe process("ls").function("xmalloc").call
{
    println(ppfunc());
    print_ubacktrace();
}

```

## Výpis backtrace zvolené volané funkce (spuštění)

---

```

xmalloc
0x4112f0 : xmalloc+0x0/0x20 [/usr/bin/ls]
0x4051c8 : sort_files+0x148/0x190 [/usr/bin/ls]
0x403c72 : main+0x12e2/0x2160 [/usr/bin/ls]
0x7f51b4e71700 [/usr/lib64/libc-2.21.so+0x20700/0x3c1000]


```

```

xmalloc
0x4112f0 : xmalloc+0x0/0x20 [/usr/bin/ls]
0x4060fc : calculate_columns+0xcc/0x2d0 [/usr/bin/ls]
0x407abe : print_current_files+0x33e/0x4e0 [/usr/bin/ls]
0x403d73 : main+0x13e3/0x2160 [/usr/bin/ls]
0x7f51b4e71700 [/usr/lib64/libc-2.21.so+0x20700/0x3c1000]


```

ls with PID=5278 finished

## Uživatelem definované funkce

---

- Základní tvar

```

function count_add(arg1, arg2) {
    return arg1 + arg2
}

```

- Explicitní uvedení návratové hodnoty

```

function concatenate:string(arg1:long, arg2) {

```

```
    return sprintf("%d%s", arg1, arg2)
}
```

## Statistika a histogramy

---

- ▶ Přidání hodnoty do pole operátorem <<<
- ▶ Vytvoření histogramu z pole @hist\_log
- ▶ Další funkce
  - ◆ @count() počet prvků (vzorků)
  - ◆ @sum součet (suma) hodnot prvků
  - ◆ @min nalezení vzorku s minimální hodnotou
  - ◆ @max nalezení vzorku s maximální hodnotou
  - ◆ @avg výpočet průměrné hodnoty
  - ◆ @hist\_linear normální histogram s lineární osou
  - ◆ @hist\_log histogram s logaritmickou osou

## Statistika a histogramy

---

```
global histogram

probe begin {
    printf("Capturing...\n")
}
probe netdev.transmit {
    histogram <<< length
}
probe netdev.receive {
    histogram <<< length
}
probe end {
    printf( "\n" )
    print( @hist_log(histogram) )
}
```

## Statistika a histogramy

---

```
global reads
global writes
global read_write_count=0

probe process("ls").begin
{
    printf("ls with PID=%d started\n", pid());
}
probe process("ls").end
{
    printf("ls with PID=%d finished\n", pid());
}
probe syscall.open
{
    filename = user_string($filename);
    printf("ls opened file %s\n", filename);
}
probe syscall.read
```

```

{
    bytes=$count
    into=$fd
    reads <<< bytes
    printf("read %d bytes from file descriptor %d\n", bytes, into);
    read_write_count++
    if (read_write_count>10000) exit()
}
probe syscall.write
{
    bytes=$count
    into=$fd
    writes <<< bytes
    printf("write %d bytes to file descriptor %d\n", bytes, into);
    read_write_count++
    if (read_write_count>10000) exit()
}
probe end
{
    println("Reads:")
    print(@hist_linear(reads, 0, 1000, 100))
    println()
    println("Writes:")
    print(@hist_linear(writes, 0, 1000, 100))
}

```

### Statistika a histogramy - ukázka výstupu

Reads:

value	count
0	0
100	2
200	0
300	10
~	
800	0
900	0
1000	1
>1000	35621

### Statistika a histogramy - ukázka výstupu

Writes:

value	count
0	17940
100	9492
200	468
300	873
400	1070
500	211
600	91
700	26
800	12
900	1

Statistické informace vypsané po ukončení procesu

```
global reads
global writes
global read_write_count=0

probe syscall.open
{
    filename = user_string($filename);
    printf("ls opened file %s\n", filename);
}
probe syscall.read
{
    bytes=$count
    into=$fd
    reads <<< bytes
    printf("read %d bytes from file descriptor %d\n", bytes, into);
    read_write_count++
    if (read_write_count>10000) exit()
}
probe syscall.write
{
    bytes=$count
    into=$fd
    writes <<< bytes
    printf("write %d bytes to file descriptor %d\n", bytes, into);
    read_write_count++
    if (read_write_count>10000) exit()
}
probe end
{
    println("Reads:")
    printf("Count: %d operations\n", @count(reads))
    printf("Total: %d bytes\n", @sum(reads))
    printf("Min: %d bytes\n", @min(reads))
    printf("Max: %d bytes\n", @max(reads))
    printf("Avg: %d bytes/operation\n", @avg(reads))
    println()
    println("Writes:")
    printf("Count: %d operations\n", @count(writes))
    printf("Total: %d bytes\n", @sum(writes))
    printf("Min: %d bytes\n", @min(writes))
    printf("Max: %d bytes\n", @max(writes))
    printf("Avg: %d bytes/operation\n", @avg(writes))
}
```

Statistické informace vypsané po ukončení procesu

Reads:  
Count: 5003 operations  
Total: 538640392 bytes

```
Min: 8196 bytes
Max: 131072 bytes
Avg: 107663 bytes/operation
Writes:
Count: 4999 operations
Total: 828740 bytes
Min: 40 bytes
Max: 2312 bytes
Avg: 165 bytes/operation
```

## GNU Debugger

---

- ▶ Vyvíjen od roku 1986
- ▶ Portace na mnoho OS
  - ◆ Unixy
  - ◆ Linux
  - ◆ ale například i DOS
- ▶ Portace na všechny významné architektury CPU
  - ◆ x86
  - ◆ x86-64
  - ◆ ARM (32bit, většina rodin)
  - ◆ AArch64
  - ◆ MIPS
  - ◆ PowerPC

## GNU Debugger

---

- ▶ Kooperace s překladači z rodiny GNU i dalšími překladači
  - ◆ Ada
  - ◆ C
  - ◆ C++
  - ◆ Go
  - ◆ Objective-C
  - ◆ D
  - ◆ Fortran
  - ◆ Modula-2
  - ◆ Pascal
  - ◆ Java
  - ◆ ...
- ▶ Chování GNU Debuggeru ovlivněno jazykem
  - ◆ formát výpisu hexadecimálních hodnot
  - ◆ struktury záznamů
  - ◆ ...

## GNU Debugger

---

- ▶ Ovládání
  - ◆ příkazový řádek
  - ◆ TUI
  - ◆ Frontend (GUI, různá IDE) přes protokol
  - ◆ „stub“ přidaný přímo do laděné aplikace
- ▶ Příkazový řádek
  - ◆ jednoznačkové a dvojznačkové zkratky příkazů
    - bt=backtrace

- c=continue
  - f=frame
  - ◆ Tab completion
  - ◆ historie příkazového řádku
- gdbtui

## GNU Debugger a trasování

---

- Tracepoints
    - ◆ zjištění stavu procesu bez jeho (po)zastavení
  - Přidání ladících informací
- ```
$ gcc -g -ansi -pedantic -Wall -o hello hello.c
```

► Spuštění debuggeru

```
$ gdb hello
```

```
...  
...  
...
```

```
Reading symbols from hello...done.
```

## Orientace v kódu

---

```
(gdb) info functions  
All defined functions:  
  
File hello.c:  
int main(int, char **);  
void print_hello();
```

## Orientace v kódu

---

```
(gdb) list  
1      #include <stdio.h>  
2  
3      void print_hello()  
4      {  
5          puts("Hello world!");  
6      }  
7  
8      int main(int argc, char **argv)  
9      {  
10         print_hello();
```

```
(gdb) list main  
4      {  
5          puts("Hello world!");  
6      }  
7  
8      int main(int argc, char **argv)  
9      {  
10         print_hello();  
11         return 0;  
12     }
```

## Orientace v kódu

---

```
(gdb) info sharedlibrary
From To Syms Read Shared Object
Library
0x00007ffff7ddaae0 0x00007ffff7df54e0 Yes (*) /lib64/ld-linux-
x86-64.so.2
(*): Shared library is missing debugging information.
```

## Breakpoinky

---

```
(gdb) break main
Breakpoint 1 at 0x40054c: file hello.c, line 10.

(gdb) run
Starting program: /home/tester/temp/presentations/tracing/hello

Breakpoint 1, main (argc=1, argv=0x7fffffff158) at hello.c:10
10          print_hello();

(gdb) n
Hello world!
11          return 0;

(gdb) c
Continuing.
[Inferior 1 (process 7040) exited normally]
```

## Zobrazení hodnot

---

```
(gdb) print(i)
$1 = 30

(gdb) print(argc>0)
$2 = 1

(gdb) print(atoi("42"))
$3 = 42

(gdb) print(isdigit('4'))
$4 = 2048

(gdb) print(isdigit('a'))
$5 = 0
```

## Parametry breakpointů

---

```
#include <stdio.h>

short factorial(short n)
{
    if (n==0 || n==1) return 1;
    return n*factorial(n-1);
}
```

```
int main(int argc, char **argv)
{
    printf("%d\n", factorial(8));
    return 0;
}
```

## Parametry breakpointů

```
(gdb) break factorial
Breakpoint 1 at 0x40053b: file factorial.c, line 5.

(gdb) info breakpoints
Num      Type            Disp Enb Address          What
1        breakpoint      keep y   0x00000000040053b in factorial at
factorial.c:5

(gdb) ignore 1 5
Will ignore next 5 crossings of breakpoint 1.

(gdb) info b
Num      Type            Disp Enb Address          What
1        breakpoint      keep y   0x00000000040053b in factorial at
factorial.c:5
                  ignore next 5 hits
```

## Výpis obsahu zásobníkových rámčů

### ► Po zastavení po pěti průchodem breakpointem

```
(gdb) bt
#0  factorial (n=3) at factorial.c:5
#1  0x00000000040055f in factorial (n=4) at factorial.c:6
#2  0x00000000040055f in factorial (n=5) at factorial.c:6
#3  0x00000000040055f in factorial (n=6) at factorial.c:6
#4  0x00000000040055f in factorial (n=7) at factorial.c:6
#5  0x00000000040055f in factorial (n=8) at factorial.c:6
#6  0x000000000400583 in main (argc=1, argv=0x7fffffff158) at
factorial.c:11
```

## Poslední čtyři volání

```
(gdb) bt 4
#0  factorial (n=3) at factorial.c:5
#1  0x00000000040055f in factorial (n=4) at factorial.c:6
#2  0x00000000040055f in factorial (n=5) at factorial.c:6
#3  0x00000000040055f in factorial (n=6) at factorial.c:6
(More stack frames follow...)
```

## Začátek historie

```
(gdb) bt -4
#3  0x00000000040055f in factorial (n=6) at factorial.c:6
#4  0x00000000040055f in factorial (n=7) at factorial.c:6
#5  0x00000000040055f in factorial (n=8) at factorial.c:6
```

```
#6 0x0000000000400583 in main (argc=1, argv=0x7fffffff1c8) at factorial.c:11
```

## Ladění zhavarovaných aplikací

---

```
void set_mem(int *address, int value)
{
    *address = value;
}

int main(int argc, char **argv)
{
    set_mem((int*)0, 42);
    return 0;
}

$ gcc -g -Wall -ansi -pedantic npe.c
$ ./npe
Segmentation fault
```

## Ladění zhavarovaných aplikací

---

```
$ulimit -c unlimited
$./npe
Segmentation fault (core dumped)
```

## GDB TUI

---

```
► GDB TUI
$ gdbtui hello
```

## Sledování přímo v kernelu

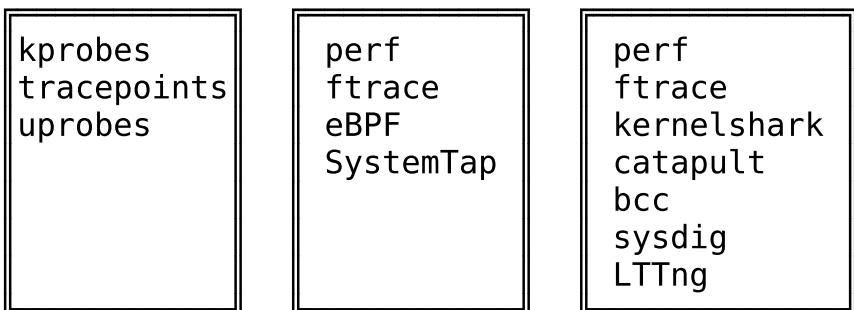
---

- Zdroj dat -> Extrakce dat -> Frontend
- LTTng - Linux Trace Toolkit, next generation
- LKST - Linux Kernel State Tracer

## Sledování přímo v kernelu

---

Zdroj dat -> extrakce dat -> frontend



## BPF (eBPF)

---

- eBPF (extended Berkeley Packet Filter)
  - ◆ Přidán do kernelů řady 4.x

- ◆ Statické a dynamické trasování
- ◆ Profilování

## Front endy pro BPF

---

- BCC
  - ◆ pro tvorbu komplexnějších nástrojů
- bpftrace
  - ◆ hodí se i pro kratší skripty a one-linery

## BCC

---

- BPF Compiler Collection
  - ◆ front endy pro
  - ◆ Python
  - ◆ Lua
  - ◆ C++
  - ◆ Go

## bpftrace

---

- Ve Fedoře od verze 28

```
# bpftrace
```

### USAGE:

```
bpftrace [options] filename  
bpftrace [options] -e 'program'
```

### OPTIONS:

```
-B MODE          output buffering mode ('line', 'full', or 'none')
```

## Otestování funkcionality bpftrace

---

- Skript je zadán přímo na příkazové řádce přes „-e“

```
# bpftrace -e 'BEGIN { printf("Hello BPF!\n"); exit(); }'  
Attaching 1 probe...  
Hello BPF!
```

## Všechny dostupné sondy

---

```
# bpftrace -l 'tracepoint:syscalls:sys_enter_*'  
tracepoint:syscalls:sys_enter_socket  
tracepoint:syscalls:sys_enter_socketpair  
tracepoint:syscalls:sys_enter_bind  
...  
tracepoint:syscalls:sys_enter_sysctl  
tracepoint:syscalls:sys_enter_exit  
tracepoint:syscalls:sys_enter_exit_group  
tracepoint:syscalls:sys_enter_waitid  
...  
...
```

## Výpis všech souborů, které jsou otevřírány

---

```
# bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n",  
comm, str(args->filename)); }'
```

```
bash /etc/profile.d/which2.sh
bash /etc/profile.d/sh.local
bash /etc/bashrc
bash /root/.bash_profile
bash /root/.bashrc
bash /etc/bashrc
bash /root/.bash_history
bash /root/.bash_history
bash /root/.inputrc
bash /etc/inputrc
```

### Histogram s výsledky sondy sys\_exit\_read

```
$ bash
$ echo $$
12345
# bpftrace -e 'tracepoint:syscalls:sys_exit_read /pid == 12345/
{ @bytes = hist(args->ret); }'
Attaching 1 probe...
^C
@bytes:
[1]          13 |
@@@ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc|
[2, 4)        0
|
[4, 8)        0
|
[8, 16)       0
|
[16, 32)      1 |
@@@@|
```

Nastavení dalších vlastností histogramu - rozsah, krok (ovlivní počet řádků)

---

```
# bpftrace -e 'kretprobe:vfs_read { @bytes = lhist(retval, 0, 2000,
200); }'
Attaching 1 probe...
^C
@bytes:
[0, 200)      8 |
@@@ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc|
[200, 400)    1 |
@@@@@@|
[400, 600)    1 |
@@@@@@|
[600, 800)    1 |
@@@@@@|
[800, 1000)   1 |
@@@@@@|
[1000, 1200)  0
|
[1200, 1400)  0
```

```
| [1400, 1600)          0
| [1600, 1800)          0
| [1800, 2000)          0
| [2000, ...)           1 |
@oooooooooooo
```

## Odkazy na další informační zdroje

---

- ▶ Tracing (software)
  - ◆ [https://en.wikipedia.org/wiki/Tracing\\_%28software%29](https://en.wikipedia.org/wiki/Tracing_%28software%29)
- ▶ ltrace(1) - Linux man page
  - ◆ <http://linux.die.net/man/1/ltrace>
- ▶ ltrace (Wikipedia)
  - ◆ <https://en.wikipedia.org/wiki/Ltrace>
- ▶ strace(1) - Linux man page
  - ◆ <http://linux.die.net/man/1/strace>
- ▶ strace (stránka projektu na SourceForge)
  - ◆ <https://sourceforge.net/projects/strace/>
- ▶ strace (Wikipedia)
  - ◆ <https://en.wikipedia.org/wiki/Strace>
- ▶ SystemTap (stránka projektu)
  - ◆ <https://sourceware.org/systemtap/>
- ▶ SystemTap (Wiki projektu)
  - ◆ <https://sourceware.org/systemtap/wiki>
- ▶ SystemTap (Wikipedia)
  - ◆ <https://en.wikipedia.org/wiki/SystemTap>
- ▶ Dynamic Tracing with DTrace & SystemTap
  - ◆ <http://myaut.github.io/dtrace-stap-book/>
- ▶ DTrace (Wikipedia)
  - ◆ <https://en.wikipedia.org/wiki/DTrace>
- ▶ GDB - Dokumentace
  - ◆ <http://sourceware.org/gdb/current/onlinedocs/gdb/>
- ▶ GDB - Supported Languages
  - ◆ <http://sourceware.org/gdb/current/onlinedocs/gdb/Supported-Languages.html#Supported-Languages>
- ▶ GNU Debugger (Wikipedia)
  - ◆ [https://en.wikipedia.org/wiki/GNU\\_Debugger](https://en.wikipedia.org/wiki/GNU_Debugger)
- ▶ The LLDB Debugger
  - ◆ <http://lldb.llvm.org/>
- ▶ Debugger (Wikipedia)
  - ◆ <https://en.wikipedia.org/wiki/Debugger>
- ▶ 13 Linux Debuggers for C++ Reviewed
  - ◆ <http://www.drdobbs.com/testing/13-linux-debuggers-for-c-reviewed/240156817>
- ▶ Getting started with ltrace: how does it do that?
  - ◆ <https://www.ellexus.com/getting-started-with-ltrace-how-does-it-do-that/>
- ▶ Reverse Engineering Tools in Linux – strings, nm, ltrace, strace, LD\_PRELOAD
  - ◆ <http://www.thegeekstuff.com/2012/03/reverse-engineering-tools/>

- ▶ 7 Strace Examples to Debug the Execution of a Program in Linux
  - ◆ <http://www.thegeekstuff.com/2011/11/strace-examples/>
- ▶ Oracle® Solaris 11.3 DTrace (Dynamic Tracing) Guide
  - ◆ [http://docs.oracle.com/cd/E53394\\_01/html/E53395/gkwo.html#scrolltoc](http://docs.oracle.com/cd/E53394_01/html/E53395/gkwo.html#scrolltoc)