Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

David Mohar

Sledenje programom v operacijskem sistemu Linux

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: doc. dr. Jurij Mihelič

Ljubljana 2016

Rezultati diplomskega dela so intelektualna lastnina avtorja ter Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljane ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Vsa izvorna koda diplomskega dela je dostopna pod licenco *GNU General Public License*, verzija 3.0 (ali novejša). To pomeni, da je izvorna koda odprta in se lahko prosto distribuira in/ali predeluje pod pogoji licence. Podrobnosti so dostopne na spletni strani http://www.gnu.org/licenses.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Testiranje in sledenje programom je pomembno pri razvoju programske opreme. V okviru diplomske naloge se osredotočite na sistemsko programsko opremo in načine njenega testiranja. Preglejte področje testiranja, raziščite izvedbo sistemskih klicev v operacijskem sistemu Linux ter možnosti za njihovo prestrezanje, preučite implementacijo orodja strace in sorodnih orodij, implementirajte program za prestrezanje sistemskih klicev in prilagodite program, da iz prestrezov sistemskih klicev izlušči pomembne podatke, ki jih lahko uporabimo pri testiranju.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani David Mohar sem avtor diplomskega dela z naslovom:

Sledenje programom v operacijskem sistemu Linux

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 22. januar 2016 Podpis avtorja:

Zahvaljujem se predvsem mami in dekletu, ki sta me ves čas podpirali ter verjeli vame tako v dobrih kot tudi v slabih trenutkih. Zahvala gre tudi mentorju, doc. dr. Juriju Miheliču, ki mi je s svojim znanjem in entuziazmom dal dodatno motivacijo za dokončanje diplomskega dela.

Moji Vesni in staršem.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Testiranje in jedro Linux **3**

2.1 Metode testiranja 3

2.2 Nivoji testiranja 5

2.3 Jedro Linux 7

2.4 Sistemski klici 7

2.5 Sistemski klic ptrace 9

3 Sledenje **13**

3.1 Sledenje sistemskim klicem 14

3.2 Dostop do registrov procesorja 17

3.3 Nadzorovano sledenje 18

3.4 Drevo ustvarjenih procesov 19

3.5 Preštevilčenje številk PID 21

3.6 Odprte datoteke 22

3.7 Signali 24

3.8 Ostali načini uporabe 25

3.9 Odin 25

4 Uporaba **31**

4.1 Izpis končnih rezultatov 32

4.2 Testni program 32

4.3 Drevo procesov 33

4.4 Statistika uporabe datotek 33

4.5 Statistika signalov 34

4.6 Statistika sistemskih klicev 34

5 Zaključek **37**

Povzetek

V diplomskem delu bomo pokazali, kako lahko tehniko sledenja programom uporabimo ne samo za razhroščevanje, ampak tudi kot dodatno orodje pri njenem testiranju, ki je dandanes zelo pomemben del razvoja. V prvem delu je predstavljenega nekaj teoretičnega ozadja o testiranju programske opreme in delovanju sistemskih klicev v operacijskem sistemu Linux. Bolj natančno je predstavljen sistemski klic ptrace, s katerim lahko prestrezamo sistemske klice, ki jih poljubni program opravi. V praktičnem delu se diplomsko delo osredotoča na sledenje in predvsem njegovo aplikacijo v procesu testiranja programske opreme. Predstavljeni so različne možnosti sistemskega klica ptrace in primera, kako lahko na enostaven način dostopamo do registrov procesorja ter kako najbolj učinkovito začnemo samo sledenje. Najpomembnejši del praktičnega dela je demonstracija, kako lahko z uporabljenim znanjem pridobimo nekaj zelo koristnih informacij. Tako smo poskusili graditi drevo procesov, ki jih opazovani program ustvari, opazovali datoteke, ki jih odpira, in prestrezali njegove signale. Na koncu je predstavljen končni izdelek diplomskega dela, program, ki vsebuje predstavljene funkcionalnosti. Podane so tudi nekatere možnosti za njegovo razširitev.

Ključne besede: testiranje, Linux, sistemski klic, sledenje, prestrezanje, ptrace.

Abstract

In this thesis we demonstrate how to use the technique of tracing not only for debugging purposes, but also as an extra tool during testing. The first part contains the theoretical background about software testing and system calls inside the Linux operating system. A special system call ptrace is described in detail because it can intercept all of application's system calls. The practical part mainly focuses on tracing and its application in software testing. We will take a look at how to access the processor's registers and how to start tracing as efficiently as possible. The main goal of the practical part is to demonstrate how we can use our newly-gained knowledge to acquire some incredibly valuable information about our program's execution. We will try to build a tree of processes it creates, intercept open files and all sent and received signals. The main result of this thesis is a program that contains all of the presented functionalities.

Keywords: testing, Linux, system call, tracing, intercepting, ptrace.

Poglavje 1

Uvod

Dandanes se od programerjev pričakuje, da bodo programsko opremo dostavili hitro in s karseda malo napakami. Tej potrebi so se z leti prilagodile tudi metode in tehnike razvoja programske opreme. Tako so v začetku leta 2001 razvijalci izdelali in objavili t. i. manifest agilnega razvoja programske opreme (angl. *Manifesto for Agile Software Development [6]*). Iz tega manifesta so se kasneje razvile različne metode razvoja, kot so npr. scrum, scrumban, ekstremno programiranje, kanban, agilno modeliranje in še mnoge druge. Vse te metode v ospredje postavljajo kratke razvojne cikle, v katerih programerji dostavljajo manjše zaključene celote, ki jih stranka lahko testira. Ker poleg hitrosti same dostave stranke zahtevajo tudi kvalitetno programsko opremo z majhnim številom napak, se agilne metode močno zanašajo tudi na različne tehnike testiranja.

Zaradi zadnje zahteve je v zadnjih letih močno narasla priljubljenost testno usmerjenega razvoja (angl. *test-driven development*), ki narekuje, da je pred pisanjem kakršnekoli kode treba napisati test, ki bo preveril njeno pravilnost. S časom se testi prilagajajo poslovnim zahtevam in temu seveda sledi tudi koda. Cilj takšnega načina programiranja je, da že na začetku ujamemo karseda veliko napak in jih seveda tudi pravočasno odpravimo (pred dostavo končnemu uporabniku).

Opazimo lahko, da dandanes razvijalci pišejo vse več testov in iščejo nove

načine, kako testirati tiste dele kode, ki so slabo ali pa sploh niso pokriti s testi, bodisi zaradi tehničnih omejitev bodisi zaradi sistema, na katerem programirajo.

V diplomskem delu se bomo osredotočili predvsem na sistemsko testiranje, ki mu razvijalci posvečajo čedalje več pozornosti. Pogledali si bomo, kako lahko izkoristimo postopek sledenja programski opremi in s tem pridobimo ogromno novih podatkov za izvajanje sistemskega testiranja. Glavni cilj je napisati program, ki bo spremljal interakcijo med programom in operacijskim sistemom ter vrnil uporabne statistike.

Na začetku se bomo osredotočili na teorijo testiranja in si bolj podrobno ogledali operacijski sistem Linux ter njegovo implementacijo sistemskih klicev. Poudarek bo predvsem na sistemskem klicu `ptrace`, ki bo naše izhodišče za sledenje. V drugi polovici si bomo ogledali nekaj praktičnih primerov sledenja z uporabo prej omenjenega sistemskega klica in njegovo aplikacijo pri zbiranju sistemskih podatkov procesa ter vse skupaj združili v uporaben program, ki nam bo pomagal pri testiranju. Čisto na koncu bomo predstavili končni izdelek in tudi nekaj konkretnih primerov.

Poglavje 2

Testiranje in jedro Linux

Testiranje programske opreme predstavlja zelo široko in, dandanes, kritično področje razvoja programske opreme. Pojem ne zajema samo testiranja programske kode, ampak celoten proces testiranja od samega nastanka aplikacije do testiranja na strani končnih uporabnikov. Celoten uspeh testiranja programske opreme temelji na *neuspehu*, kar nekaterim na prvi pogled deluje protislovno, vendar ni. Pri testiranju se moramo zavedati, da nam vsak uspešen test doprinese dodatno vrednost samo v primeru, da je bil pred tem neuspešen. Samo tako lahko pokažemo, da smo odpravili nepravilnost v našem programu. Kadar se lotimo testiranja, moramo paziti, da si postavimo realne in dosegljive cilje. Za vsako majhno funkcionalnost namreč obstaja neskončno mnogo testov. Naša naloga je, da iz te poplave testnih primerov izluščimo tiste, ki nam pokrijejo karseda veliko množico vseh možnih vhodnih podatkov.

2.1 Metode testiranja

Danes poznamo veliko različnih pristopov oz. metod testiranja aplikacij, ki jih lahko razdelimo v dve večji skupini, in sicer **statično** in **dinamično testiranje**. Statično testiranje izvajajo, po navadi, že sama razvojna okolja in prevajalniki v okviru statične analize kode, medtem ko moramo za di-

namično testiranje naš program zagnati z določenimi testnimi primeri. V statično testiranje štejemo tudi sam pregled kode s strani drugega razvijalca (angl. *review*), kar se pogosto uporablja pri načinu vodenja projekta *Scrum*. Najprej si oglejmo nekaj različnih metod testiranja.

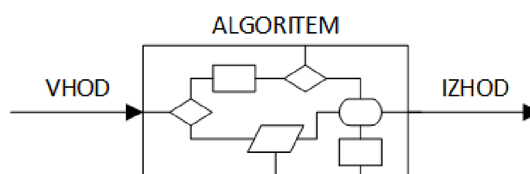
2.1.1 Testi črne škatle

Najbolj klasično preverjanje delovanja predstavljajo **testi črne škatle** (glej Sliko 2.1), kjer testiramo komponente brez znanja o njihovi notranji zgradbi (implementaciji). Komponento si tukaj predstavljamo kot nekakšno "črno škatlo", ki prejme nek nabor vhodnih podatkov in nam vrne ustrezen rezultat. Za testerja je dovolj, da ve, kaj naj bi aplikacija počela, dejanski algoritem pa mu ni pomemben.

Slika 2.1: Diagram testa črne škatle

2.1.2 Testi bele škatle

Testi bele škatle, za razliko od testov črne škatle, preverjajo interno delovanje določene komponente in zahtevajo, da tester pozna tudi algoritem, s pomočjo katerega pridemo do zahtevanega rezultata (glej Sliko 2.2). Poleg vsega povedanega so pri tej metodi pomembni tudi primerno izbrani testni primeri, saj mora tester zagotoviti, da le-ti pokrijejo vse možne veje izvajanja (stavki if). Pri teh testih, in tudi pri testih črne škatle, se programerji ter testerji poslužujejo t. i. orodij za analizo pokritosti kode, ki razvijalcu povedo, kolikšen delež in kateri deli določene komponente so testirani.



Slika 2.2: Diagram testa bele škatle

2.1.3 Testi sive škatle

Testi sive škatle so nekakšna kombinacija zgornjih dveh metod testiranja. Razvijalec oz. tester mora tukaj poznati interno delovanje določene komponente (testiranje bele škatle), vendar teste izvaja na uporabniškem nivoju (testiranje črne škatle). S poznavanjem algoritma testerju omogočimo, da izbere boljše testne primere.

2.1.4 Vizualno testiranje

Vizualno testiranje predstavlja idejo, da lahko razvijalec napako odpravi hitreje, če vidi, kaj se je dogajalo v trenutku, ko je do le-te prišlo. Poleg tega se vizualno testiranje zanaša na to, da so podatki podani jasno (nedvoumno) in v takšni obliki, da lahko programer takoj izlušči informacije, ki jih potrebuje.

2.2 Nivoji testiranja

Celoten proces testiranja lahko po dokumentu IEEE SWEBOK [1] razdelimo na več nivojev oz. faz, in sicer:

- **testiranje komponent** ali **testiranje enot** (angl. *unit testing*), kjer preverjamo delovanje vsake izolirane enote posebej,

- **integracijske teste** (angl. *integration testing*), kjer preverjamo interakcijo med enotami prejšnje faze,

- **sistemske teste** (angl. *system testing*), kjer preverjamo delovanje našega sistema oz. aplikacije kot zaključeno celoto.

Vsaka faza testiranja pokriva en arhitekturni nivo.

2.2.1 Testiranje komponent

Testiranje komponent oz. **testiranje enot** preverja funkcionalnost vsake izolirane komponente programa posebej, kjer komponenta predstavlja funkcijo/proceduro ali razred (na višjem nivoju). Dolžnost pisanja testov v tej fazi pade na razvijalce programske opreme, da lahko že med pisanjem zahtevane funkcionalnosti zagotovijo pravilno delovanje. Vsaka funkcija je lahko pokrita z več testi, saj tako preverimo tudi robne pogoje. Testiranje enot samo po sebi ne preverja funkcionalnosti celotne aplikacije, ampak nam samo zagotavlja, da vsaka enota deluje pravilno in neodvisno od ostalih enot v programu.

2.2.2 Integracijski testi

Za razliko od testiranja komponent **integracijski testi** preverjajo pravilnost interakcije med komponentami. Po navadi komponente postopoma vključujemo v samo aplikacijo in z vsako vključitvijo ponovno izvajamo teste. Tako lahko veliko hitreje odkrijemo morebitne nepravilnosti v interakciji, zaradi manjšega števila samih komponent pa hitreje izoliramo modul, ki povzroča napako. Včasih testerji na tem nivoju namesto postopnega vključevanja vključijo vse komponente hkrati, čemur pravimo tudi *veliki pok* (angl. *big bang*).

2.2.3 Sistemski testi

Sistemski testi so namenjeni testiranju celotne aplikacije z namenom preverjanja pravilnega delovanja sistema. Poleg testiranja samega delovanja s takim testiranjem tudi zagotovimo, da naša končna aplikacija ne vpliva na delovanje okolja, v katerem teče (operacijskega sistema), ali ostale zunanje komponente (ostali programi, knjižnice ipd.).

2.3 Jedro Linux

Leta 1991 je za svoj računalnik Linus Torvalds, takrat študent računalništva, napisal jedro operacijskega sistema. Linusovo jedro Linux (angl. *Linux kernel*) je modularno monolitno jedro operacijskega sistema, kar pomeni, da je celotno jedro napisano kot en velik program, ki se izvaja v privilegiranem načinu delovanja (angl. *supervisor mode*). Ena izmed boljših lastnosti Linuxa je, že prej omenjena, modularnost, ki jedru omogoča nalaganje in izklapljanje t. i. LKM (angl. *linux kernel modules*) med samim delovanjem operacijskega sistema. Dandanes je jedro Linux zelo popularno, saj je od svojega spočetka zelo napredovalo na področju interoperabilnosti. Poganjamo ga lahko na skoraj katerikoli večji procesorski arhitekturi (npr. ARM, i386, IA-64 ipd.), z izklopom večine dodatnih (nepotrebnih) modulov pa celotno jedro zasede zelo majhno količino prostora.

Velja omeniti, da je izvorna koda jedra Linux prijetna za branje in zelo dobro komentirana, tako tehnično kot tudi vsebinsko. Skozi poglavje se bomo večkrat sklicavali na različne datoteke jedra, ki so javno dostopne na svetovnem spletu [5].

2.4 Sistemski klici

Sistemski klici so zelo pomembni, saj se uporabniški programi vedno izvajajo v zaščitenem načinu delovanja, zato nimajo dostopa do nekaterih funkcij CPE. Prav tako nimajo dostopa do celotnega pomnilniškega prostora, kar lahko predstavlja težavo pri delu z vhodno-izhodnimi napravami. Vse te težave rešujemo z uporabo t. i. sistemskih klicev (angl. *system calls*), kjer, preprosto povedano, uporabniški program prosi (pooblasti) operacijski sistem, da v njegovem imenu izvrši zahtevano funkcijo (npr. dostop do trdega diska).

Arhitektura sistemskih klicev znotraj jedra Linux je relativno preprosta. Poglejmo si najprej delovanje na najnižjem nivoju.

št. klica	**sistemski klic**
0	`read`
1	`write`
2	`open`
3	`close`
...	...
57	`fork`

Tabela 2.1: Nekaj pomembnejših sistemskih klicev

2.4.1 Sistemski nivo

Uporabniški program oz. programer mora najprej poznati številko sistemskega klica, ki bi ga rad izvedel (Tabela 2.1). Linux podpira sistemske klice, ki imajo do šest vhodnih parametrov, lahko pa jih razširimo tako, da dodatne parametre potisnemo na sklad. Številko sistemskega klica zapišemo v register `eax` (`rax` na 64-bitni arhitekturi). Parametre klica vpišemo po vrsti v registre `ebx`, `ecx`, `edx`, `esi`, `edi` in `ebp` (`rdi`, `rsi`, `rdx`, `r10`, `r8` in `r9` na 64-bitni arhitekturi). Sedaj nam ostane samo še dejanski klic prekinitve, zadolžene za sistemske klice. Na starejših sistemih se tako pokliče `int #0x80`, kasneje pa sta Intel in AMD dodala ukaz `sysenter/syscall`, ki omogočata hitrejši in varnejši sistemski klic.

Naslednja faza sistemskega klica se že izvaja v privilegiranem načinu, in sicer znotraj rokovalnika pasti sistemskih klicev.

```
cmpl $(NR_syscalls), %eax
jae syscall_badsys
call *sys_call_table(,%eax,4)
```

Naloga te procedure je, da preveri veljavnost številke sistemskega klica, poišče rokovalnik za ta klic in ga pokliče. Zaradi podpore razhroščevanju (angl. *debugging*) bo jedro pred klicem rokovalnika in po njem sprožilo posebno past, če je le-ta omogočena (več o tem v Poglavju 2.5). Rezultat

sistemskega klica se pred vračanjem v zaščiteni način oz. uporabniški program zapiše v register `eax` (`rax` na 64-bitni arhitekturi). Jedro vrne kontrolo uporabniški aplikaciji z uporabo ukaza `iret` ali `sysexit/sysret` na modernejših sistemih oz. procesorjih.

2.4.2 Uporabniški (aplikacijski) nivo

Na višjem nivoju so sistemski klici realizirani s pomočjo t. i. ovojnih funkcij (angl. *wrapper functions*). Te ovojne funkcije same poskrbijo za nastavitev številk sistemskega klica in njegovih parametrov. Največ časa so ovojne funkcije poimenovane kar po samem klicu brez predpone *sys* (npr. ovojna funkcija za `sys_open` je `open`).

Ovojna funkcija izvede sistemski klic s standardno funkcijo `syscall`. Leta je malce bolj zapletena za uporabo, saj prepusti odgovornost nastavljanja parametrov v prave registre programerju, ki pa so lahko različni za različne arhitekture.

2.5 Sistemski klic ptrace

Klic `ptrace` je eden izmed najpomembnejših sistemskih klicev, namenjen razhroščevanju uporabniških aplikacij. V grobem nam omogoča, da opazujemo in spreminjamo tok izvajanja izbranega programa, prav tako pa lahko tudi pregledujemo in spreminjamo stanje procesorskih registrov ter njegov uporabniški prostor. Programu, ki sledi sistemskim klicem, bomo rekli opazovalec (angl. *tracer*), programu, ki mu sledimo, pa opazovanec (angl. *tracee*).

Pred samim sledenjem z uporabo klica `ptrace` se mora opazovalec najprej "pripeti"na opazovan proces z uporabo akcije `PTRACE_ATTACH`. Alternativno, kadar je opazovalec oče opazovanega procesa, lahko tudi otrok zahteva sledenje z uporabo opcije `PTRACE_TRACEME`. Večino težjega dela potem opravi jedro Linux, ki pred sistemskim klicem in po njem preveri vrednost posebne zastavice `_TIF_WORK_SYSCALL_ENTRY`. Če je nastavljena,

pokliče posebno funkcijo klica ptrace `syscall_trace_enter`.

```
long syscall_trace_enter(struct pt_regs* regs)
{
  ...
  if ((ret || test_thread_flag(TIF_SYSCALL_TRACE)) &&
          tracehook_report_syscall_entry(regs))
        ...
}
```

Ta poskrbi, da se izvede ustrezna logika glede na izbrane opcije sledenja preko ukaza ptrace. Poleg tega ta poskrbi tudi za ustavljanje opazovanca in seveda ustrezno obveščanje opazovalca s klicem `ptrace_report_syscall`.

```
static inline int ptrace_report_syscall(struct pt_regs *regs)
{
  int ptrace = current->ptrace;
  if (!(ptrace & PT_PTRACED))
    return 0;

  ptrace_notify(SIGTRAP |
    ((ptrace & PT_TRACESYSGOOD) ? 0x80 : 0));

  if (current->exit_code) {
    send_sig(current->exit_code, current, 1);
    current->exit_code = 0;
  }

  return fatal_signal_pending(current);
}
```

Znotraj te funkcije `ptrace_notify` ustavi opazovanca in čaka, da ga opazovalec nadaljuje z eno izmed opcij `PTRACE_CONT` ali `PTRACE_SYSCALL`. Poleg ustavljanja opazovanca procedura poskrbi tudi za pošiljanje signala `SIGTRAP` opazovalcu, ki mu na našo željo nastavi ustrezen bit, saj tako brez

večjih težav vemo, da je prejet signal posledica klica `ptrace`. Po opravljenem sistemskem klicu se izvrši podobna logika, ki zopet ustavi opazovanca in obvesti opazovalca tik pred vrnitvijo v uporabniški prostor.

Poglavje 3

Sledenje

Sledenje bi najlažje opisali kot razvijalsko tehniko, največkrat uporabljeno za namene razhroščevanja in boljšega razumevanja delovanja programske opreme. Programerji se z njim prvič srečamo že v šoli, kjer programom sledimo ročno s pomočjo svinčnika in papirja, tisti, ki svojo kariero nadaljujemo v svetu razvoja programske opreme, pa sledenje uporabljamo za beleženje informacij, ki so ključne za pravilno delovanje programa ali pa bi nam kasneje olajšale iskanje in odpravljanje napak v kodi. To lahko vključuje veliko stvari, kot so npr. trenutno stanje spremenljivk, trenutno stanje registrov v procesorju, stanje sklada, stanje pomnilnika itd. Sledenje je specializirana vrsta beleženja (angl. *logging*), kjer so beležene informacije nekoliko bolj nestrukturirane, beležijo pa se na karseda nizkem nivoju, saj so samo tako lahko programerjem v veliko pomoč. Te informacije se potem uporabljajo tako med samim razvojem programa kot tudi med delovanjem pri končnem uporabniku. Zaradi beleženja na tako nizkem nivoju so lahko rezultati sledenja zelo obsežni, hkrati pa sam izpis lahko pri končni namestitvi oz. uporabi drastično upočasni delovanje našega programa, zato mora imeti vsak program opcijo izklopa sledenja bodisi ob prevajanju bodisi med izvajanjem le-tega. Rezultati sledenja so klasično relativno kompleksni in uporabni samo programerjem ali pa sistemskim administratorjem, mi pa si bomo ogledali, kako lahko sledenje uporabimo tudi kot orodje za testiranje kode na sistemskem

nivoju.

Če želimo sledenje uporabiti za testiranje naše programske kode, je najprej potreben premislek, čemu sploh slediti. Vsakdo, ki ima nekaj računalniškega znanja, ve, da uporabniške programe in strojno opremo v osnovi povezuje t. i. operacijski sistem. Programi z njim komunicirajo preko vnaprej dogovorjenih mehanizmov, ki jim pravimo sistemski klici.

V nadaljevanju si bomo ogledali, kako lahko sledimo sistemskim klicem v operacijskem sistemu Linux.

3.1 Sledenje sistemskim klicem

Operacijski sistem Linux ponuja kar nekaj možnosti oz. orodij za sledenje programski opremi, kot sta npr. ftrace (angl. *Function Tracer*) in LTTng (angl. *Linux Trace Toolkit*), vendar oba zahtevata vklop t. i. sledilnih točk, ki so bile vključene v jedro Linux šele proti koncu leta 2008. Poleg tega je za vklop teh točk potrebno ponovno prevajanje jedra Linux s posebno zastavico `CONFIG_FUNCTION_TRACER`, kar lahko uporabniku vzame kar precej časa.

Poleg zgoraj omenjenih orodij Linux vsebuje še poseben sistemski klic, imenovan `ptrace`, ki nam omogoča nadzor in pregled internega delovanja procesov ravno za potrebe razhroščevanja. Med drugim nudi tudi dostop do trenutnega stanja registrov v procesorju. Njegova najbolj znana uporaba je ravno prestrezanje in sledenje sistemskim klicem. Spodaj je predstavljen podpis funkcije ptrace.

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

Za nas bosta najpomembnejša prva dva parametra. Prvi opisuje akcijo, ki naj jo izvede ptrace, drugi pa pove številko PID procesa, nad katerim naj se ta akcija izvede. Uporabo ostalih dveh parametrov si bomo ogledali kasneje. Za potrebe diplomskega dela pride v poštev nekaj osnovnih akcij, ki jih omogoča ptrace, in sicer:

- PTRACE_ATTACH začne celotno sejo sledenja nad določenim procesom in s tem vklopi nekaj zastavic znotraj jedra Linux, ki smo jih omenjali v prejšnjem poglavju in omogočajo prestrezanje sistemskih klicev,
- PTRACE_SETOPTIONS omogoča nastavitev parametrov sledenja, ki jih bomo spoznali nekoliko kasneje,
- PTRACE_TRACEME, tako kot prva možnost, začne sejo sledenja, ampak z druge strani – otrok pove staršu, naj ga sledi,
- PTRACE_PEEKUSER prebere, med drugim, trenutno stanje registrov procesorja – nekatere smo omenili že prej,
- PTRACE_PEEKDATA prebere podatke iz pomnilniškega prostora procesa, ki mu sledimo,
- PTRACE_SYSCALL opazovani proces nemoteno nadaljuje do naslednjega klica (vhoda) ali izhoda iz sistemskega klica.

Če poznamo številko procesa PID, ki mu želimo slediti, lahko, opremljeni s tem znanjem, sestavimo enostaven program. Pripeti se moramo na številko procesa PID, ki ga želimo opazovati, in spremljati njegove prekinitve. Celotno sejo lahko začnemo s preprostima klicema funkcije ptrace.

```
ptrace(PTRACE_ATTACH, pid, 0, 0);
ptrace(PTRACE_SETOPTIONS, pid,
        0, PTRACE_O_TRACEEXIT | PTRACE_O_TRACESYSGOOD);
```

Središče tega programa bo zanka, ki bo čakala na prekinitve opazovanega programa in se na njih primerno odzvala. Ker smo našo sejo začeli z vključeno opcijo PTRACE_O_TRACESYSGOOD, bodo prekinitve zaradi sledenja označene s posebno zastavico v številki signala. Moramo se zavedati, da ptrace ustavi proces dvakrat na vsak sistemski klic – pri vstopu in izstopu iz tega klica – zato je pomembno, da ju med seboj ločimo. Pri vstopu v sistemski klic imamo na voljo vse vrednosti parametrov, vendar še nimamo rezultata. Tega dobimo šele ob izhodu. Ker sledimo samo enemu procesu,

je nemogoče, da bi ob vstopu v sistemski klic proces opravil nov klic, zato bomo ta dva koraka ločili z enostavno spremenljivko.

```
short in_sys_call = 0;
while (1)
{
    int observed_pid, status, signum;
    observed_pid = waitpid(pid, &status, 0);
    if (WIFSTOPPED(status))
    {
        signum = WSTOPSIG(status);
        if (signum & 0x80)
        {
                if (in_sys_call)
                        printf("System call exit\n");
                else
                        prinf("System call enter\n");
            in_sys_call != in_sys_call;
        }
    }
    if (WIFEXITED(status))
            break;
    ptrace(PTRACE_SYSCALL, pid, 0, 0);
}
```

V zanki čakamo naš opazovani proces, da se ustavi, in preberemo status te ustavitve s klicem na waitpid. Z uporabo posebnih makroukazov, ki preverijo določeno zastavico tega statusa, potem pogledamo, ali se je program dejansko ustavil ali pa samo oddal kakšen signal sam. Če ugotovimo, da je res prišlo do ustavitve, preverimo, kateri signal ga je ustavil. Ker smo ob inicializaciji sledenja uporabili opcijo PTRACE_O_TRACESYSGOOD, nam ptrace take signale označi s posebno zastavico v številki signala. Tako vemo, da je ustavitev res povzročil ptrace. Šele takrat lahko kličemo ptrace z ostalimi zahtevami, ki smo jih omenili prej. V primeru programa zgoraj trenutno samo izpišemo vstop in izstop iz sistemskega klica. Seveda pa ob koncu zanke ne smemo pozabiti preveriti, ali se je opazovani proces slučajno že končal, in nadaljevati izvajanje le-tega s ponovnim klicem na ptrace.

3.2 Dostop do registrov procesorja

Zgornji primer programa ne izpisuje skoraj nič uporabnega, zato ga bomo še malce nadgradili. Med našim sledenjem in prestrezanjem klicev nas zanimajo predvsem:

- vrednosti vhodnih parametrov,
- rezultat sistemskega klica,
- številka sistemskega klica.

Vse tri lahko dobimo z enostavnim vpogledom v stanje registrov procesorja. Ob klicu bo številka sistemskega klica shranjena v registru `eax`, ob izhodu pa bo to vrednost zamenjal rezultat klica. Vrednosti parametrov lahko preberemo iz ostalih registrov, ki smo jih omenili že v Poglavju 2. Za to bomo potrebovali enostavno funkcijo in nekaj makrov.

```
#define offsetof(a, b) __builtin_offsetof(a, b)
#define get_reg(pid, name) \
    __get_reg(pid, offsetof(struct user, regs.name))
long __get_reg(int pid, int offset)
{
    long val = ptrace(PTRACE_PEEKUSER, pid, offset);
    return val;
}
```

Ker lahko pri klicu dobimo samo njegovo številko, ne pa tudi imena, moramo imeti le-ta shranjena nekje v našem programu. Imena in številke sistemskih klicev so na vseh distribucijah Linux enaki, zato so lahko te tabele pripravljene že vnaprej. Naša bo za vsak sistemski klic vsebovala njegovo ime in število vhodnih parametrov. To sedaj upoštevamo pri izpisu sistemskega klica v naši glavni zanki.

```
if (in_sys_call)
{
    int result = get_reg(pid, eax);
    printf("%d\n", result);
}
else
{
    int call_num = get_reg(pid, eax);
    printf("%s() = ", sysents[call_num].func_name);
}
```

Rezultat, ki ga dobimo na tej točki, je že kar uporaben in informativen. Hitro lahko vidimo, da skoraj vsak, tudi enostaven, program naredi ogromno sistemskih klicev med svojim izvajanjem. Če dobljene podatke spremljamo malce bolj podrobno in selektivno, zagotovo lahko pridemo do še boljših rezultatov. Seveda pa ima zato naš program manjšo pomanjkljivost. Vsakemu procesu lahko začne slediti šele potem, ko se ta začne izvajati. Tako vedno izgubimo prvih nekaj klicev, ki so ravno tako lahko zelo pomembni.

3.3 Nadzorovano sledenje

Prej omenjeni problem najlažje rešimo tako, da sami zaženemo proces, ki bi ga radi opazovali. Le tako lahko začnemo sledenje takoj ob njegovem zagonu. Izvedba bo enostavna, in sicer bomo začeli nov proces s klicem na `fork()`, potem pa znotraj tega procesa poklicali ptrace in začeli izvajanje želenega programa. Zaradi enostavnosti je v primeru uporabljen kar program `ls`.

```
static void exec_or_die()
{
    ptrace(PTRACE_TRACEME);
    raise(SIGSTOP);
    execvp("ls", "ls", NULL);
    exit(1);
}
```

Naš novi proces pokliče ptrace z akcijo, ki začne njegovo lastno sejo sledenja in to tudi sporoči staršu. Takoj zatem se pokliče `raise`, v katerem samemu sebi pošlje signal `SIGSTOP`. Za starša to pomeni, da ga bo otrok počakal pred

klicem na execvp, dokler starš ne nadaljuje njegovega izvajanja. V naši glavni zanki bo zato prvi klic na waitpid samo naš `SIGSTOP`, ko ga nadaljujemo, pa bomo že lahko ujeli prvi sistemski klic programa ls. Ker ukaz `execvp` v pomnilniku prepiše naš program s programom, ki bi ga radi izvedli, se iz funkcije nikoli ne vrnemo. Izjema so razne napake, kot je npr. napačno podana pot do programa. Zaradi tega klicu `execvp` sledi ukaz `exit(1)`, ki bo v primeru napake naš program ustavil.

Pogledali smo si osnove sledenja sistemskim klicem v operacijskem sistemu Linux, vendar bi nadalje to radi izkoristili kot uporabno orodje za pomoč pri sistemskem testiranju, zato si bomo v nadaljevanju pogledali nekaj uporabnih aplikacij sledenja.

3.4 Drevo ustvarjenih procesov

Kadar opazovani program ustvarja nove procese, nas večino časa zanima, koliko jih je ustvaril on, in tudi, koliko njegovi otroci. Ravno zaradi razvejane strukture takšnega izpisa mu pravimo drevo procesov. Kako ugotoviti, kdaj je naš opazovani program ustvaril nov proces, ne bi smelo biti problematično, saj moramo samo spremljati klice na funkcijo `fork()`. Zaplete se, ko začnemo razmišljati o njegovih otrocih, vendar lahko to rešimo s hitrim premislekom. Če bi poleg starša opazovali tudi njegove otroke, bi lahko z enako tehniko prišli tudi do procesov, ki so jih ustvarili le-ti. Tako lahko nadaljujemo do poljubne globine. V ta namen bomo v našem programu potrebovali posebno tabelo, kjer bomo hranili številke PID vseh procesov, ki se pojavijo tekom sledenja, in za vsak proces še številko PID njegovega starša.

```
#define MAX_PROCESSES 50
struct process
{
    int pid;
    int parent;
};
struct process process_tab[MAX_PROCESSES]
```

Treba je še paziti, ker `fork()` ni edina funkcija, ki lahko ustvari nov proces. Poleg tega poznamo tudi `vfork()` in `clone()`, zato moramo uloviti vse tri. Ker nam vse tri funkcije vrnejo številko PID novega otroka, je najlažje klic preverjati ob izhodu iz klica. Takrat je v registru `eax` zapisan rezultat sistemskega klica, zato lahko do njegove številke pridemo preko posebnega registra `orig_eax`, ki hrani vrednost registra ob vstopu.

```
if (in_sys_call)
{
    int result = get_reg(observed_pid, eax);
    int call_num = get_reg(observed_pid, orig_eax);
    // clone, fork, vfork
    if (call_num > 55 && call_num < 59)
    {
            ptrace_attach(result);
            new_process(result, observed_pid);
    }
}
```

Funkcija `ptrace_attach` deluje na enak način kot primer pripenjanja zgoraj. Druga funkcija, `new_process`, prejme številki PID starša in njegovega novega otroka. Vse, kar naredi, je, da v naši tabeli procesov naredi nov zapis in ustrezno nastavi vrednosti.

```
process_tab[process_len].pid = result;
process_tab[process_len].parent = observed_pid;
```

Po zaključenem sledenju lahko z enostavno rekurzivno funkcijo, katere primer se nahaja nekoliko nižje, izpišemo nastalo drevo procesov, po naraščajočih vrednostih številk PID pa lahko določimo vrstni red njihovih nastankov.

```
static void
process_tree_rec(int parent, int depth)
{
    int i;
    // Ustvari niz, ki bo pri izpisu predstavljal globino drevesa
    char* prepend = prepare_lines(depth);
    for (i = 0; i < process_len; i++)
    {
        struct process* proc = processs_tab[i];
        if (!proc->pid)
            return;
        if (ctrl->pid == parent)
            continue;
        if (ctrl->parent == parent) {
            print_process(prepend, proc);
            process_tree_rec(proc->pid, depth + 1);
        }
    }
}
```

3.5 Preštevilčenje številk PID

Pri testiranju nam številke PID lahko, med drugim, povedo tudi nekaj o zaporedju nastajanja procesov. Operacijski sistem te številke vedno izdaja v naraščajočem vrstnem redu, vendar nikoli ne vemo, pri kateri številki bo začel in ali je mogoče kakšna vmesna številka že rezervirana. S preštevilčenjem si lahko ta proces olajšamo in procesom dodeljujemo svoje številke.

V našem programu bomo vedno začeli z 0, naslednike pa dodeljevali sekvenčno v naraščajočem vrstnem redu. Najboljše mesto za to je znotraj funkcije `new_process`, kjer samo dodamo enostaven stavek.

```
process_tab[process_len].renum_pid = renum_seed++;
```

Uporabniku lahko tudi ponudimo možnost, da številčenje začne pri poljubnem številu, ki ga poda kot argument programu. Zadnja stvar, na katero ne smemo pozabiti, je, da pri izpisu drevesa procesov izpišemo njihove nove številke.

```
int pid = renumber ? proc->renum_pid : proc->pid;
printf("%s%d\n", prepend, pid);
```

3.6 Odprte datoteke

Programi velikokrat za svoje delovanje uporabljajo datoteke, a vendar vsi vemo, da je najpogostejša napaka pri uporabi datotek to, da jo ob koncu pozabimo zapreti. Če bi želeli slediti vsem odpiranjem in zapiranjem datotek, lahko enostavno spremljamo dva sistemska klica:

- `open(char* path, int oflags)` nam odpre datoteko na poti `path` z določenimi opcijami, kot sta npr. `O_RDONLY` ali `O_RDWR`, kot rezultat pa nam vrne številko deskriptorja odprte datoteke,

- `close(int fd)`, ki zapre deskriptor, katerega številko prejme kot argument.

V splošnem bi bilo za potrebe čistega testiranja dovolj samo preverjanje, da ima vsak open tudi pripadajoči klic na close, vendar nam to ne bi povedalo veliko. Potrebujemo vsaj še ime datoteke, zato se sama logika sledenja in hranjenja podatkov nekoliko zaplete. Ker lahko programer oz. uporabnik datoteko odpre z uporabo različnih relativnih poti, je najboljši način za enolično identifikacijo njena številka, imenovana inode (angl. *index node number*). Za to potrebujemo dve tabeli, in sicer eno, ki bo hranila preslikave iz številke deskriptorja v inode, ter drugo, ki bo hranila podatke o vseh do sedaj odprtih datotekah, vključno z njihovo številko inode. Obe strukturi sta enostavni, in sicer:

- preslikava vsebuje samo številko deskriptorja in kazalec na strukturo, ki predstavlja datoteko,

```
struct fd_to_file_ent
{
    int fd_no;
    struct file_ent* file_ent;
    struct fd_to_file_ent* next;
};
```

- deskriptor datoteke vsebuje malce več podatkov, za statistiko pa je pomembno samo število klicev na `open`, `close`, `read` in `write`.

```
struct file_ent
{
    ino_t inode_no;
    int fd_no;
    char* filename;
    int opens;
    int closes;
    long reads;
    long writes;
};
```

Tabeli lahko izvedemo na mnogo različnih načinov, kot so npr. vektorji, povezani seznami, razpršene tabele, binarna drevesa ipd. Na tej točki velja tudi premisliti, koliko in kako pogosto se bosta omenjeni tabeli spreminjali. Hitro lahko ugotovimo, da se bodo preslikave med datotečnimi deskriptorji in številko inode spreminjali večkrat in veliko pogosteje. Če to upoštevamo pri implementaciji, lahko preslikovalno tabelo implementiramo kot preprosto razpršeno tabelo, tabelo deskriptorjev pa kot vektor.

Dopolnitev glavne zanke je enostavna, saj samo dodamo dva stavka if.

```
if (in_sys_call)
{
    int result = get_reg(observed_pid, eax);
    int call_num = get_reg(observed_pid, orig_eax);
    // open
    if (call_num == 2)
    {
        int path = get_reg(observed_pid, ebx);
        mark_open(observed_pid, result, path);
    }
    // close
    else if (call_num == 3)
    {
        mark_close(observed_pid, result);
    }
}
```

Funkcija mark_open za opazovano številko PID poskuša poiskati številko deskriptorja, ki je podana kot drugi argument. Če ga ne najde, z uporabo klica ptrace prebere vrednost niza na naslovu path (prvi argument sistemskega klica open), prebere njegov inode s klicem funkcije stat in ustvari nova zapisa v obeh tabelah. Na koncu samo povečamo število odpiranj. Za mark_close velja podobna logika, samo da povečujemo število zapiranj in odstranjujemo preslikave, ko se deskriptorji zaprejo. Po koncu takšnega sledenja imamo tako za opazovani proces seznam vseh uporabljenih datotek in njihovo pripadajoče število odpiranj ter zapiranj. Če želimo, lahko poleg tega spremljamo tudi število bajtov, prebranih iz datoteke in zapisanih v njo.

3.7 Signali

Prejete in poslane signale moramo preverjati na dva različna načina. Prejete bomo zaznali tako, da bomo dopolnili stavek if v naši glavni zanki, saj bomo o njegovem prejemu obveščeni.

```
while (1)
{
    // ...
    if (WIFSTOPPED(status))
    {
        if (signum & 0x80)
                // ...
        else
        {
            printf("[%d] >> %s <<\n", proc->print_pid,
                get_signame(signum));
            // statistika
            mark_sig(SIG_RECEIVED, proc, signum);
        }
    }
}
```

Poslane signale pa bomo prejeli tako, da bomo prestrezali klice na sistemski klic kill. Prvi argument tega klica predstavlja prejemnika, drugi pa številko signala, ki ga pošilja proces. Do argumentov sistemskega klica bomo prišli z

uporabo funkcije `syscall_arg`, ki glede na zaporedno številko vhodnega argumenta prebere vrednost iz ustreznega registra procesorja.

```
if (in_sys_call)
{
    int result = get_reg(observed_pid, eax);
    int call_num = get_reg(observed_pid, orig_eax);
    // kill
    if (call_num == 62)
    {
        int kill_pid = syscall_arg(observed_pid, 0);
        int kill_sig = syscall_arg(observed_pid, 1);
        mark_sig(SIG_SENT, proc, kill_sig);
        // printf
    }
}
```

3.8 Ostali načini uporabe

Ogledali smo si nekaj uporab sledenja kot orodje pri testiranju, obstaja pa jih še veliko več. Linux pozna, trenutno, 316 sistemskih klicev in skoraj vsakega lahko na pameten način izkoristimo za pridobivanje pomembnih statistik. Tako bi lahko npr. spremljali celoten vhod in izhod iz programa (ukazna vrstica in datoteke), komunikacijo preko omrežja, delo z datotečnim sistemom, delo z jedrom operacijskega sistema, porabo in vsebino pomnilnika ipd. Nekatere enostavne uporabe si bomo ogledali v naslednjem poglavju.

3.9 Odin

Pridobljeno znanje je med drugim obrodilo tudi konkreten program, ki se imenuje *Odin*. Ta, podobno kot `strace`, sledi poljubnemu programu in seveda vsem njegovim sistemskim klicem. Na podlagi tega nam zgradi uporabna poročila, ki jih lahko uporabimo pri testiranju programov.

3.9.1 Izvedba

Program je zaradi preglednosti in lažje razumljivosti razdeljen na tri večje sklope oz. dele:

1. inicilizacijo in zagon testnega programa,
   `init(argc, argv);`

2. glavno sledilno zanko,
   `trace();`

3. izpis končnih rezultatov.
   `output();`

Osnovna logika implementacije je podobna primeru, ki smo ga pokazali v prejšnjem poglavju. V nadaljevanju si bomo pogledali, kako vsak od njih, v grobem, deluje.

Polno izvorno kodo lahko najdete na https://github.com/davidmohar/odin.

3.9.2 Inicializacija in zagon testnega programa

Glavna naloga funkcije `init` je razčleniti vhodne argumente in zagnati proces, ki mu želimo slediti. Ob zagonu programa brez parametrov bo funkcija izpisala navodila za uporabo in ga končala. Po tem naredi `fork()` in izvede podan program vključno z morebitno podanimi argumenti, na svoji strani pa alocira prvi proces – svojega otroka.

```
pid = fork();
/* otrok */
if (!pid)
{
    exec_or_die(argc -- argv_offset, argv + argv_offset);
}
/* starš */
ctrl = alloc_ctrl(pid, 0);
ctrl->status |= TC_INIT;
```

argv_offset tukaj predstavlja odmik argumentov otroka v starševi tabeli, ki jo dobimo ob zagonu. Funkcija, ki izvede otroka, je sestavljena podobno kot tista v prejšnjem poglavju. Najprej nastavi ustrezne preusmeritve, če smo seveda to želeli, in zahteva sledenje. Temu takoj sledi klic funkcije trace().

3.9.3 Sledilna zanka

Glavna zanka je preprosta in sledi vzorcu in prejšnjega poglavja. Začne se s klicem funkcije waitpid, ki nam med drugim vrne tudi številko PID otroka, ki je prekinitev sprožil. Pri tej prvi iteraciji začne z izvajanjem našega otroka. Preko te številke poiščemo naš deskriptor procesa, ki ga predstavlja spodnja struktura.

```
struct trace_ctrl
{
    int pid;
    int parent;
    int status;
    int exit_status;
    int lastsyscall;

    struct fd_to_file_ent* fd_mappings;
    struct file_ent** file_ents;
    int file_ents_count;
    int num_files;

    clock_t start_time;
    clock_t end_time;
    double running_time;

    int syscall_stat[SYSCALL_COUNT];
    struct signal_stat* signal_stats;
};
```

Vsaka skupina elementov je namenjena drugi funkciji našega programa. Prvi dve predstavljata osnovne podatke o procesu, zadnjem sistemskem klicu in odprtih datotekah. Zadnji dve skupini sta namenjeni statistiki, kot so čas izvajanja, število sistemskih klicev in število poslanih ter prejetih signalov.

Naslednji korak je, da preverimo, ali je prekinitev povzročil ptrace in ali se je kakšen od procesov že zaključil. Takrat zmanjšamo število procesov, ki jih opazujemo, izračunamo končni čas njegovega izvajanja in preverimo, ali lahko sploh nadaljujemo z izvajanjem. V nasprotnem primeru preverimo, ali gre za vstop ali izhod iz sistemskega klica. Za to uporabimo makroukaz spodaj.

```
#define PROCINSYSCALL(proc) \
    (proc->status & TC_INSYSCALL) == TC_INSYSCALL
```

Ta podatek je pomemben, ker bomo večino dela opravili pri izstopu iz klica, saj imamo takrat na voljo tudi njegov rezultat. Tam vzamemo njegovo številko in rezultat ter odvisno od sistemskega klica naredimo naslednje:

- `clone`, `fork` in `vfork` – takoj začnemo sejo sledenja in alociramo nov deskriptor procesa, ki je bil ravnokar ustvarjen,

```
ptrace_attach(syscallresult);
alloc_ctrl(syscallresult, pid);
```

- `open`, `read`, `write` in `close` – glede na deskriptor datoteke našemu procesu dodamo novo odprto datoteko ali pa popravimo števce branj, pisanj in zapiranj,
- `kill` – glede na podane argumente (številko PID in številko signala) ustrezno dopolnimo statistiko signalov in v primeru polnega izpisa to tudi izpišemo.

Če smo v sistemski klic vstopali, v deskriptor zapišemo njegovo številko in popravimo statistiko sistemskih klicev. V primeru, da smo vstopali v klic `exit`, to ustrezno tudi izpišemo in si zapomnimo izhodni status. Na koncu vedno obrnemo zastavico, ki nam govori, ali je trenutni proces v sistemskem klicu.

```
proc->status ^= TC_INSYSCALL;
```

Če prekinitve otroka ni povzročil ptrace, gre verjetno za prejet signal in to seveda ustrezno označimo v našem deskriptorju. Glavno zanko ponavljamo, dokler sledimo še vsaj enemu procesu.

Poglavje 4

Uporaba

Program uporabljamo na podoben način, kot se uporablja `strace`. Najosnovnejša opcija zahteva samo ukaz, s katerim izvedemo program, ki mu želimo slediti (npr. `odin ps` za sledenje programu `ps`). Seveda *Odin* podpira nekoliko več stikal, in sicer:

-v vključi razširjen (angl. *verbose*) izpis sledenja,
-p ob koncu izvajanja izpiše drevo procesov, ki jih je program ustvaril,
-r num preštevilči vse številke PID z začetkom pri num,
-m izpiše preslikovalno tabelo preštevilčenih številk PID v originalne,
-t poleg drevesa procesov izpiše tudi čas izvajanja procesov,
-s izpiše statistiko opravljenih sistemskih klicev,
-f izpiše poročilo uporabe datotek (število odpiranj, zapiranj, prebrano, zapisano),
-a vklopi izpis prejetih in poslanih signalov,
-i datoteka preusmeri izbrano datoteko v *stdin* testnega programa,
-o datoteka preusmeri *stdout* testnega programa v izbrano datoteko,
-e datoteka preusmeri *stderr* testnega programa v izbrano datoteko.

4.1 Izpis končnih rezultatov

Izpis rezultatov opravimo glede na naše podane vhodne argumente. Zaradi lažje predstavitve si bomo pogledali kar nekaj konkretnih izpisov pod različnimi nastavitvami. Sledili bomo enostavnemu programu, ki bo ustvaril nekaj procesov, odprl nekaj datotek, pošiljal signale ipd.

4.2 Testni program

Za prikaz delovanja programa Odin bomo sestavili enostaven program, ki nam bo omogočal prikaz delovanja vseh funkcij našega programa. Njegova implementacija je podana v primeru spodaj. Seveda lahko Odin uporabimo na poljubnem operacijskem sistemu Linux.

```
int main()
{
    int pid = fork();
    int fd = open("/dev/null", O_RDWR);
    int fd2 = open("program.c", O_RDONLY);
    int fd3 = open("./program.c", O_RDONLY);

    close(fd);
    close(fd2);
    close(fd3);

    pid = fork();
    if (pid != 0)
    {
        kill(pid, SIGURS1);
    }

   int fd4 = open("../odin/../odin/../odin/program.c", O_RDONLY);
   return 2;
}
```

Predstavljen testni program bomo uporabili v primerih v nadaljevanju.

4.3 Drevo procesov

Če naše orodje zaženemo s parametrom -p, bo ta izpisal drevo procesov, ki ga opazovani program ustvari, dodatno pa tudi izhodni status vsakega nastalega procesa. Iz izpisa lahko vidimo, da naš testni program ustvari dva procesa, eden izmed njegovih otrok pa še dodatno enega. Seveda imajo vsi izhodni status 2, saj smo tako zaključili naš testni program.

Takšen izpis je pri testiranju pomemben, kadar je pomembno tudi drevo procesov, ki ga ustvari naš program. Tako lahko hitro preverimo število ustvarjenih procesov in seveda tudi določimo, v kakšnem vrstnem redu so ti nastali.

```
> odin -p ./program
[proctree] Created Process Tree
7444 (exit status 2)
- 7445 (exit status 2)
-- 7446 (exit status 2)
- 7447 (exit status 2)
```

4.4 Statistika uporabe datotek

Za izpis statistike uporabljenih datotek moramo program zagnati s parametrom -f. V tem izpisu lahko najdemo število odpiranj in zapiranj vsake datoteke, ki jo uporablja program med svojih izvajanjem. Poleg tega sta dodatno izpisana tudi podatka o številu prebranih in zapisanih bajtov.

Na takšni statistiki po večini preverjamo, ali ima vsak klic funkcije open tudi pripadajoči klic funkcije close. V nasprotnem primeru lahko v našem programu pride do puščanja pomnilniškega prostora.

```
> odin -f ./program
[filestat] File usage statistics
[?] Format: file_name num_opens num_closes read written
[*] PID: 7464
/etc/ld.so.cache    1    1    0    0
/dev/null           1    1    832  0
program.c           3    3    0    0
...
```

4.5 Statistika signalov

Če naše orodje poženemo brez parametrov, nam bo ta izpisal statistiko poslanih in prejetih signalov. S pomočjo izpisa lahko nato preverimo, ali naš program pošilja pravilne signale svojim otrokom ali pa med izvajanjem prejme napačen oz. nepričakovan signal.

```
> odin ./program
[sigstat] Signal statistics
[?] Format: SIGNAME(signum) sent received
[*] PID: 7464
SIGABRT (5)     0    4
SIGSEGV(10)     1    0
SIGTSTP(19)     0    1
[*] PID: 7465
SIGABRT (5)     0    2
SIGSEGV(10)     1    0
SIGTSTP(19)     0    1
[*] PID: 7466
SIGABRT (5)     0    1
SIGSEGV(10)     0    1
SIGTSTP(19)     0    1
...
```

4.6 Statistika sistemskih klicev

Z uporabo opcije `-s` nam Odin izpiše statistiko opravljenih sistemskih klicev. Izpis nam pove, kolikokrat je bil kateri klic izveden. Takšne informacije so zelo priročne predvsem v fazi razhroščevanja, saj lahko hitro vidimo, česa naš program ne počne ali česa ne bi smel početi.

```
> odin -s ./program
[callstat] Syscall statistics
[?] Format: syscall num_calls
[*] PID: 7470
read          1
write         1
open          6
close         6
fstat         3
mmap          8
mprotect      4
munmap        1
brk           1
ioctl         1
access        3
clone         2
execve        1
arch_prctl    1
...
```

Poglavje 5

Zaključek

V prvem delu diplomskega dela smo predstavili osnovno teorijo samega testiranja in jedra Linux, kjer smo opisali osnovne metode ter nivoje testiranja, v nadaljevanju pa smo si pogledali izvedbo sistemskih klicev v operacijskem sistemu Linux. Tukaj smo spoznali še posebej uporaben sistemski klic, imenovan `ptrace`, ki nam omogoča opazovanje interakcije z operacijskim sistemom. Ogledali smo si njegovo izvedbo na sistemskem nivoju in ugotovili, da imajo sistemski klici operacijskega sistema Linux vgrajeno zelo dobro podporo za njihovo prestrezanje in posledično tudi sledenje.

V nadaljevanju smo se osredotočili na sledenje in posledično na uporabo sistemskega klica `ptrace`. Poleg vsega zgoraj naštetega nam sistemski klic `ptrace` ne omogoča samo sledenja in prestrezanja rezultatov sistemskih klicev, temveč tudi vpogled v registre procesorja ter pomnilniškega prostora, ki ga opazovana aplikacija uporablja. Hitro smo lahko videli, da nam bo to omogočilo zbiranje zelo uporabnih podatkov na relativno enostaven način. Tako smo si pogledali nekaj zanimivih statistik, kot sta npr. drevo ustvarjenih procesov in seznam odprtih datotek. Že ta dodatna dva izpisa nam lahko zelo veliko pomagata pri odkrivanju napak, kot je npr. uhajanje pomnilnika zaradi odprtih datotek.

Na koncu smo predstavili še končni izdelek diplomskega dela, in sicer program *Odin*, ki glede na podane vhodne parametre vrača različne zanimive

statistike, na katerih lahko potem izvajamo teste. Prostora za razširitev je še veliko in tudi sami bi radi videli, da se temu programu doda še kakšen uporaben izpis oz. sledenje kakšnemu novemu sistemskemu klicu.

Literatura

[1] P. Bourque, R. E. Fairley, et al, "Guide to the Software Engineering Body of Knowledge, Version 3.0", *IEEE Computer Society*, 2014; www.swebok.org

[2] D. P. Bovet, M. Cesati, "Understanding the Linux Kernel, Third Edition", *O'Reilly Media, Inc.*, str. 398–455, 2006.

[3] R. Love, "Linux Kernel Development, Third Edition", *Pearson Education, Inc.*, str. 69–83, 2010.

[4] B. Meyer, "Seven Principles of Software Testing", *Computer, IEEE Computer Society*, št. 41, zv. 8, str. 99–101, 2008.

[5] The Linux Kernel Archives, dostopno na:
https://www.kernel.org/

[6] Manifesto for Agile Software Development, pridobljeno: 23.11.2015 ob 18:40, dostopno na: http://agilemanifesto.org

[7] Wikipedia, Agile software development, pridobljeno: 23.11.2015 ob 18:20, dostopno na:
https://en.wikipedia.org/wiki/Agile_software_development

[8] Wikipedia, Linux kernel, pridobljeno: 10.12.2014 ob 17:43, dostopno na:
https://en.wikipedia.org/wiki/Linux_kernel