

OS laboratorinis darbas Nr. 3

POSIX C API**Turinys**

1 Įvadas.....	2
2 ELF formatas.....	3
3 Dinaminis ryšių redaktorius.....	4
4 make.....	5
5 LD 3 1/5.....	7
5.1 Teorija.....	7
5.1.1 C kompiliatorius, assembleris, linkeris.....	7
5.1.2 ELF analizės įrankiai.....	8
5.1.3 Proceso analizės įrankiai.....	8
5.1.4 Profaileris.....	9
5.2 Darbas.....	9
5.2.1 GCC kompiliatorius.....	10
5.2.2 ELF analizė ir trasavimas.....	11
5.2.3 Profaileris.....	13
5.3 Atsiskaitymas.....	13
5.4 Vertinimas.....	13
6 LD 3 2/5.....	14
6.1 Teorija.....	14
6.2 Darbas.....	14
6.2.1 open(), close(), perror(), exit().....	15
6.2.2 pathconf(), fpathconf().....	17
6.2.3 getcwd(), chdir(), fchdir().....	17
6.2.4 opendir(), fdopendir(), readdir(), closedir().....	18
6.2.5 stat(), fstat().....	18
6.2.6 statvfs(), fstatvfs().....	18
6.2.7 link(), unlink(), symlink(), remove(), rename(), mkdir(), rmdir(), creat().....	18
6.2.8 umask(), chmod(), fchmod().....	18
6.2.9 futimens().....	19
6.2.10 nftw().....	19
6.3 Atsiskaitymas.....	20
6.4 Vertinimas.....	21
7 LD 3 3/5.....	22
7.1 Teorija.....	22
7.2 Darbas.....	23
7.2.1 Sinchroninis I/O.....	23
7.2.2 Buferizuotas I/O.....	24
7.2.3 Asinchroninis I/O.....	24
7.2.4 Failų „mapinimas“ į RAM.....	26
7.3 Atsiskaitymas.....	28
7.4 Vertinimas.....	29
8 LD 3 4/5.....	30
8.1 Teorija.....	30
8.1.1 Dinaminis vykdomų duomenų užkrovimas.....	30
8.1.1.1 Dinaminės bibliotekos kūrimas.....	30

8.1.1.2 Bibliotekos prikabinimas ryšių redaktoriaus pagalba.....	31
8.1.1.3 Bibliotekos užkrovimas vykdymo metu.....	32
8.2 Darbas.....	33
8.2.1 Informacija apie procesams taikomus apribojimus.....	33
8.2.2 Programos darbo užbaigimas.....	35
8.2.3 Laiko informacija ir jos apdorojimas.....	35
8.2.4 Vykdyto prioritetas.....	37
8.2.5 Dinaminis užkrovimas.....	37
8.3 Atsiskaitymas.....	38
8.4 Vertinimas.....	38
9 LD 3 5/5.....	39
9.1 Teorija.....	39
9.1.1 Soketai.....	39
9.1.1.1 UNIX domain soketai (AF_UNIX).....	42
9.1.1.2 IPv4 soketai (AF_INET).....	45
9.1.1.3 IPv6 soketai (AF_INET6).....	46
9.2 Darbas.....	48
9.2.1 Pagalbinės tinklo funkcijos.....	48
9.2.2 UNIX domain soketai.....	49
9.2.3 IPv4 soketai.....	49
9.2.4 IPv6 soketai.....	50
9.3 Atsiskaitymas.....	50
9.4 Vertinimas.....	50

1 Įvadas

Moodle: POSIX.zip, ABI.pdf, APIs.pdf, C99.pdf

http://en.wikipedia.org/wiki/Memory_management_unit

http://en.wikipedia.org/wiki/Page_table

Šis laboratorinis darbas skirtas susipažinimui su programavimu UNIX (POSIX) aplinkoje. Jo metu bus apžvelgti programuotojui prieinami interfeisai (API), programų kūrimo bei derinimo priemonės.

Pirmoji UNIX OS buvo parašyta assembleriu, tačiau vėlesnės versijos perrašytos C kalba su assemblerio fragmentais. C kalbos naudotojams žymiai palengvino UNIX perkėlimą į kitas platformas (CPU architektūras). C kalba ir buvo sukurta siekiant perkelti UNIX į kitos architektūros mašiną.

UNIX aprašantis standartas – **POSIX** (*Portable Operating System Interface*): POSIX.1-2008 arba **SUSv4** (*Single Unix Specification Version 4*) arba **IEEE Std 1003.1-2008**. Šis standartas be viso kito aprašo ir daugumą ankstesniuose laboratoriniuose darbuose naudotų komandų, SHELL interpretatorių ir C funkcijas bei header failus.

C programavimo kalbą aprašo *ISO/IEC 9899 - Programming languages - C* standartas (**C11, ISO/IEC 9899:2011**).

Norint aiškiau suprasti OS darbą su atmintimi reiktų bent apytiksliai įsivaizduoti **MMU** (*Memory Management Unit* – atminties kontrolieris) veikimo principus ir jo vietą sistemoje su virtualia atmintimi. MMU verčia loginius adresus, kuriuos mato ir su kuriais dirba programos ir OS į fizinius adresus. MMU taip pat atlieka atminties segmentų apsaugą. Kiekvienas procesas mato vientisą atminties regioną, nors fiziškai proceso atminties segmentai gali būti išmėtyti RAM, be to proceso

vykdymo metu jų vieta RAM gali keistis. Proceso „matoma“ atminties erdvė gali būti didesnė už fiziškai sistemoje esantį RAM kiekį. Jei procesas kreipiasi į atminties ląstelę, kuriai MMU nėra priskirta fizinio segmento – įvyksta *page fault* pertraukimas, kurio apdorojimui išskviečiama OS funkcija, kuri arba nutraukia procesą, arba pataiso adresų transliavimo lentelę ir tęsia procesą.

2 ELF formatas

MAN: `elf(3ELF)`, `gelf(3ELF)`, `libelf(3ELF)`

http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

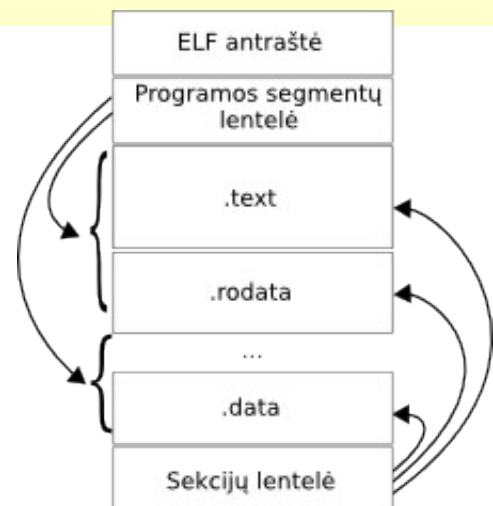
<http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html>

Moodle: `ELF_format.pdf`

ELF (*Executable and Linkable Format*) - tai failų formatas, skirtas statiniams ir dinaminiam vykdomiems (*executable*), objektiniams (*relocatable*), statinių bibliotekų, dinaminė bibliotekų (*shared*), *core* failams saugoti UNIX šeimos operacinėse sistemose. ELF failas sudarytas iš:

1. ELF antraštė

2. programos **segmentų lentelės**, aprašančios segmentus (segmentai "mapinami" į RAM);
3. kodo, duomenų ir kitos informacijos (t. y. turinio, į kurį rodo segmentų ir sekcijų aprašai);
4. **sekcijų lentelės**, kurioje saugoma ryšių redaktoriui (*linker*, *runtime-linker*, *dynamic linker/loader*) skirta informacija.



Priklausimai nuo ELF failo tipo, segmentų arba sekcijų lentelės gali nebūti, t. y. statiniam vykdomajam failui g.b. nereikalinga sekcijų lentelė (užtenka informacijos kur padėti kodą ir duomenis ir nuo kurio adreso pradėti vykdymą), o objektiniam failui nereikalinga segmentų lentelė (objektinis failas nevykdomas).

PASTABA: programavimo aplinka ant `os.if.ktu.lt` nepilna, todėl kompiliatorius nekuria statinių vykdomųjų failų.

Sekcijų pavadinimai skirtingose OS gali skirtis, keletas dažniausiai sutinkamų:

1. **.interp** – sekcijoje įrašytas pilnas kelias iki interpretatoriaus (dažniausiai – dinaminio ryšių redaktoriaus), t. y. programos, kuri prieš startuojant procesą sutvarko jo aplinką (išskiria RAM proceso kodui ir duomenims, sutvarko ryšius su dinaminėmis bibliotekomis ir t.t.);
2. **.text** – programos kodas;
3. **.data** – programos duomenys (kintamieji);
4. **.rodata** – programos tik skaitomi duomenys;
5. **.bss** (*Block Started by Symbol*) – skirta neinicializuotų duomenų saugojimui, t. y. paleidžiant procesą, OS šią sekciją užpildo nuliais;
6. **.init** – papildomas programos inicializavimo kodas, išskviečiamas prieš paleidžiant programą (C parašytoms programoms nereikalinga, C++ - reikalinga);
7. **.fini** – panaši į **.init** sekcija, kurios kodas išskviečiamas užbaigus pagrindinę programą;
8. **.got** (*Global Offset Table*) – „globalių poslinkių lentelė“, dinaminė ryšių lentelė;
9. **.plt** (*Procedure Linkage Table*) – „procedūrų jungčių lentelė“, dinaminė ryšių lentelė;

10. `.dynamic` – dinaminių ryšių informacija;
11. `.dynstr`, `.dynsym`, `.strtab`, `.symtab` – dinaminių ir statinių simbolių ir eilučių lentelės.
12. ...

3 Dinaminis ryšių redaktorius

MAN: ld.so.1(1)

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Dinaminių ryšių redaktoriaus (*run-time link-editor*, *runtime linker*, *dynamic linker*) paskirtis – užkrauti dinaminį ELF vykdomąjį failą ir jo naudojamas dinamines bibliotekas (*shared libraries*, *dynamic libraries*) į atmintį ir sutvarkyti dinaminių ryšių informaciją (kad paleista programa galėtų atrasti ir iškviešti funkcijas esančias dinaminėse bibliotekose).

Visus programos segmentus galima suskirstyti į modifikuojamus (*read-write*) ir nemodifikuojamus (*read-only*). Programos mašininis kodas laikomas nemodifikuojamame segmente. Šių apribojimų laikymąsi padeda užtikrinti MMU. Tokia ideologija leidžia nemodifikuojamus atminties segmentus daryti prieinamus keliems procesams vienu metu. Tačiau iškyla problema norint sukurti procesą, naudojantį dinamines bibliotekas: programa turi iškviešti bibliotekos funkcijas, todėl reiktų modifikuoti kodo segmento turinį įrašant užkrautos dinaminės bibliotekos funkcijų adresus, tačiau programos kodas yra nemodifikuojamame segmente. ELF atveju problema sprendžiama GOT ir PLT lentelių pagalba.

GOT (*Global Offset Table*) – `.got` sekcija, globalių objektų (duomenų ir funkcijų) adresų lentelė. Tai duomenų sekcija/segmentas (skaitomas, rašomas, nevykdomas), kurį turi kiekvienas ELF vykdomas failas ir dinaminė biblioteka. Šioje lentelėje proceso kūrimo metu įrašomi modulio (vykdomo failo ar bibliotekos) naudojamų kitų modulių objektų (kintamųjų, funkcijų) adresai bendroje proceso adresų erdvėje. Pavyzdžiui, jei procesas nori nuskaityti kurio nors kintamojo, aprašyto bibliotekoje, reikšmę, šį kintamąjį jis gali pasiekti pagal adresą GOT lentelėje.

PLT (*Procedure Linkup Table*) – `.plt` sekcija, perėjimų į funkcijas lentelė (*jump-table*). Tai programos kodo (vykdoma, tik skaitoma) sekcija/segmentas, kurį turi kiekvienas ELF vykdomas failas ir dinaminė biblioteka. PLT lentelėje yra kodo fragmentai kiekvienai globaliai funkcijai, esančiai kitame modulyje. Šie moduliai vykdo perėjimus (*jump*) GOT lentelėje įrašytais adresais.

Kodėl PLT lentelėje iškart neįrašomi atitinkamų funkcijų adresai (kad nereikėtų naudoti GOT lentelės įrašų)? Reikalas tas, kad UNIX dinaminis ryšių redaktorius palaiko „vėlyvą surišimą“ (*lazy binding*), t. y. funkcijos adresas išrišamas tik pirmojo jos iškvietimo metu. Kadangi PLT lentelė tik skaitoma, neišeina jos turinio keisti proceso veikimo metu. Todėl keičiamas GOT įrašo turinys. Pradžioje (iki iškviečiant funkciją), GOT lentelės įraše saugomas adresas, užtikrinantis valdymo perdavimą į dinaminio ryšių redaktoriaus funkciją atliekančią GOT lentelės taisymą.



1. pav. Dinaminio vykdomo ELF failo paleidimas.

Aplinkos kintamieji įtakojantys dinaminio ryšių redaktoriaus veikimą (ne visi):

- `LD_LIBRARY_PATH` – kelias, kur ieškoti dinaminių bibliotekų;
- `LD_PRELOAD` – priverstinai užkrauti išvardintas bibliotekas (ELF faile nėra nuorodos į šias bibliotekas – galima naudoti naudojamų dinaminių funkcijų perėmimui);
- `LD_BIND_NOW` – iškart (prieš paleidžiant programą) užpildyti dinaminių ryšių lentelę

(iškart įrašyt funkcijų adresus į GOT lentelę).

4 make

MAN: `make(1S)`, `gmake(1)`

INFO: `make` (aprašo `gmake`)

POSIX: `susv4/utilities/make.html`

[http://en.wikipedia.org/wiki/Make_\(28software\)](http://en.wikipedia.org/wiki/Make_(28software))

http://en.wikipedia.org/wiki/List_of_build_automation_software

Programuojant C ar kita kompiliuojama kalba pastoviai prireikia peregeneruoti failus pataisius išeitinį programos tekstą. Programavimo aplinkos paprastai turi vienokias ar kitokias priemones tam automatizuoti. POSIX standartas aprašo **make** komandą, kuri pagal taisykles aprašytas faile **makefile** arba **Makefile** (jei nėra *makefile*) peregeneruoja failus.

Skirtingose OS naudojamų *make* veikimas šiek tiek skiriasi, tačiau pagrindinis principas vienodas. Didelė dalis atviro kodo programų rašoma ir derinama naudojant *GNU make*, todėl OS, kurios turi savo *make*, dažnai papildomai instaliuojama ir *GNU make* (Solaris/BSD komandos vardas **gmake**). Yra ir daugiau panašias funkcijas atliekančių paketų (*mk*, *cmake*, *qmake*, *Ant*, ...). Kurį naudoti – pasirinkimo reikalas.

Makefile aprašo priklausomybių tarp išeities tekstų ir generuojamų failų grafą (nebūtinai medį), kurio viršūnėse – išeitiniai failai, mazguose – generuojami failai, o briaunose – generuojamų failų kūrimo komandos. Didesniame medyje dažnai gaunasi, kad dalis generuojamų failų gali būti kuriami vienu metu (lygiagrečiai), tai turi prasmę daugiaprocesorinėse sistemose ar klasteriuose (klasteriuose reikalinga paskirstyta *make* versija, pvz.: *pmake*). Apie lygiagretų vykdymą užsimenama POSIX specifikacijoje, bet jo palaikymas nestandartizuotas. *GNU make* turi **-j** parametą, nurodantį kiek maksimaliai užduočių vykdyti vienu metu.

Naudojimo pavyzdžiai (vietoj *make* g.b. *gmake*):

Atnaujinti failus pagal *makefile* arba *Makefile* aprašytas taisykles:

```
make
```

Atnaujinti *hello* failą pagal *makefile* arba *Makefile* aprašytas taisykles:

```
make hello
```

Parodyti, kokios komandos būtų paleistos *hello* atnaujinimui, bet jų nevykdyti:

```
make -n hello
```

Atnaujinti failus pagal taisykles aprašytas *ManoMakefile* faile:

```
make -f ManoMakefile
```

Taisyklių pavyzdžiai:

Generuoti *hello1*, jei pasikeitė *hello1.c* failas naudojant *make* taisykles pagal nutylėjimą (*default rules*):

```
hello1: hello1.c
```

Generuoti *hello1*, bet ne pagal *default* taisykles, o su nurodytomis *gcc* komandomis:

SVARBU: pirmas simbolis eilutėje, kurioje aprašoma atnaujinimo komanda (*gcc*) turi būti TAB, tarpai netinka! Galima naudoti daugiau nei vieną eilutę atnaujinimo komandų aprašymui (pirmas simbolis irgi TAB).

```
hello1: hello1.c
        gcc -c -g hello1.c
        gcc -o hello1 hello.o
```

GNU makefile, pagal kurį *gmake* iš kiekvieno *.c failo einamajame kataloge bandys sugeneruoti vykdomą failą su debuggerio informacija.

```
SRC=$(wildcard *.c)
ALL=$(SRC:.c=)
CC=gcc
CFLAGS=-g -Wall -Werror -pedantic

all: ${ALL}
```

make komandų taikymas neapsiriboja programų kūrimu. Lygiai taip pat sėkmingai ji gali būti naudojama nuotraukų ar video konvertavimui, HTML, XML, PDF ar kokių kitų dokumentų kūrimui, analizei ar modifikavimui, eksperimentų automatizavimui.

5 LD 3 1/5

Tikslas: susipažinti ir realiai pabandyti ELF failų kūrimo ir analizės įrankius.

5.1 Teorija

5.1.1 C kompiliatorius, assembleris, linkeris

MAN: gcc(1), ld(1), as(1), cpp(1)

http://en.wikipedia.org/wiki/GNU_Compiler_Collection, <http://gcc.gnu.org/>

<http://en.wikipedia.org/wiki/LLVM>, <http://llvm.org/>

XSI (*X/Open System Interfaces* – POSIX išplėtimas, aprašantis C ir SHELL interfeisus programų kūrimui) reikalavimus atitinkanti UNIX realizacija, be privalomų POSIX bazinių reikalavimų turi realizuoti ir papildomus XSI interfeisus, tame tarpe ir C programų kūrimo interfeisą (**c99**).

CPU tiesiogiai C kalbos nesupranta, todėl reikalingi įrankiai, kurie C kalbos tekstą paverstų į CPU suprantamą kalbą - mašininius kodus. "Išversta" C programa, kartus su OS skirta pagalbine informacija saugoma specialaus formato faile, UNIX atveju dažniausiai tam naudojami ELF formato failai.

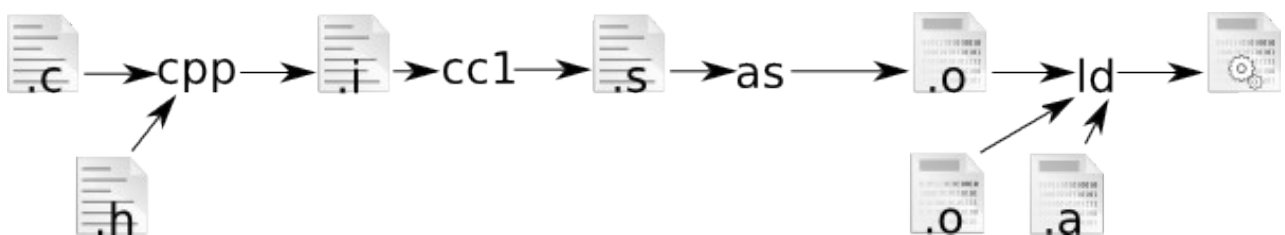
C programos failo pavyzdys **hello.c**:

```
#include <stdio.h>
int main () {
    printf( "Hello, World!\n" );
    return 0;
}
```

C programos kompiliavimas (su GCC kompiliatoriumi):

```
gcc -o hello hello.c
```

Einamajame kataloge turėtumėte gauti vykdomą failą **hello**.



2 pav. Vykdomo ELF failo kūrimas iš C programos teksto.

Kol gaunamas galutinis rezultatas *hello*, C kompiliatorius įvykdo keletą pagalbinių žingsnių, kai kuriems iškviestas atskiras programos (2 pav.):

1. preprocesorius (**cpp** – g.b. integruotas su kompiliatoriumi), interpretuojantis C preprocesoriaus direktyvas (šiuo atveju **#include** pakeičia **include.h** failo turinį), gaunamas vientisas C programos tekstas be preprocesoriaus direktyvų;
2. kompiliatorius (**cc1**, **cc**, **c99**, **lcc** ...), verčiantis C programos tekstą į assemblerio kalbos tekstą (gaunamas **hello.s** tekstinis assemblerio kalbos failas);
3. assembleris (**as**), verčiantis assemblerio failą **hello.s** į objektinį **hello.o** (ELF „relocatable“ dvejetainis failas);

4. ryšių redaktorius – „linkeris“ (**ld**), surenkantis **hello.o** ir reikalingas bibliotekas (**libc** ir kt.) į vykdomąjį failą **hello** (ELF dvejetainis vykdomasis failas).

GCC ne vienintelis UNIX sistemose naudojamas kompiliatorius. Kitas populiarėjantis, tačiau ne vienintelis paketas – LLVM.

5.1.2 ELF analizės įrankiai

MAN: ldd(1), dump(1), elfdump(1), nm(1)

ldd – parodo visas ELF failui ar bibliotekai reikalingas bibliotekas. Naudojimo pavyzdys:

```
ldd /bin/true
ldd /lib/libelf.so.1
```

dump (Linux/BSD: *objdump*) – Solaris programa parodo informaciją iš objektinio (tame tarpe ir ELF) failo: ELF headerio turinį, lenteles ir t.t. Naudojimo pavyzdys:

```
dump -f -o -h -v /bin/true
dump -s -d 1 /bin/true
```

elfdump (Linux/BSD: *readelf*) – Solaris programa parodo informaciją iš ELF failo. Naudojimo pavyzdys:

```
elfdump /bin/id
elfdump -G /bin/id
```

nm – parodo ELF failo simbolių lentelę. Naudojimo pavyzdys:

```
nm -x /bin/true
nm -u /bin/true
```

5.1.3 Proceso analizės įrankiai

MAN: gdb(1), truss(1), pldd(1), pstack(1), pfiles(1)

<http://en.wikipedia.org/wiki/Gdb>

<http://www.gnu.org/software/gdb/>

<http://rsquared.sdf.org/gdb/>

<http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html>

<http://sourceware.org/gdb/current/onlinedocs/gdb.html>

truss (Linux: *strace*; BSD: *kdump/ktrace*) – [jau naudota LD1 metu] parodo proceso kviečiamus *syscall*'us ir jų parametrus. Naudojimo pavyzdys:

```
truss /bin/true
```

pldd, **pstack**, **pfiles** – pagalbinės Solaris programos, parodančios informaciją apie nurodytą veikiantį procesą, atitinkamai: proceso naudojamas bibliotekas; proceso steką; proceso atidarytus failus, tinklo sujungimus, socketus. Naudojimo pavyzdys (čia 22591 – bet kokio nuosavo proceso PID):

```
pstack 22591
pfiles 22591
pldd 22591
```

gdb – GNU debugeris, skirtas programos trasavimui, klaidų analizei, *core* failų analizei. Pagrindinės funkcijos: programos paleidimas su norimais parametrais; pažingsninis vykdymas (ir

po vieną eilutę, ir po vieną mašininę (*assembler*) komandą); kintamųjų ir atminties turinio analizė; CPU registrų analizė; steko analizė. Naudojimo pavyzdys:

```
gdb /bin/true
break _start
run
backtrace
break main
continue
disassemble
info registers
info target
x/64x *($eip)
continue
quit
```

GDB sukurta ir grafinių X11 interfeisų: *DDD*, *KGdb*, bet ant *os.if.ktu.lt* X11 neinstaliuotas, todėl šių interfeisų nėra). Yra instaliuotas **gdbtui** – tekstinis gdb interfeisas.

Norint patogiai trasuoti C programas, jos turi būti sukompiliuotos su **-g** parametru, tada į ELF failą papildomai įdedama debuggeriui reikalinga informacija, leidžianti susieti kodą su išeities tekstu ir atmintį su kintamaisiais. Kompiliavimas:

```
gcc -g -o hello hello.c
```

Norint analizuoti core failą:

```
gdb nuluzus_programa core
```

Šiuo atveju GDB į atmintį užkraunama nulūžusio proceso būseną prieš pat OS jį nutraukiant (t. y. lyg procesas būtų sustabdytas prieš pat lūžimą). Naudojant GDB komandas galima ieškoti lūžimo priežasties (tikrinti kintamųjų reikšmes skirtinguose steko freimuose, tikrinti atminties turinį, registrų turinį ir t.t.).

5.1.4 Profaileris

MAN: gprof(1)

[http://en.wikipedia.org/wiki/Profiling_\(computer_programming\)](http://en.wikipedia.org/wiki/Profiling_(computer_programming))

gprof – profaileris, parodantis programos funkcijų iškviatimo grafą, iškviatimų skaičių ir vykdymo laikus. Norint naudoti *gprof*, reikia, kad C programa būtų sukompiliuota su **-pg** parametru, tada į ELF failą įdedamas papildomas kodas programos vykdymo metu rašantis informaciją į **gmon.out** failą. Ši informacija gali būti analizuojama su *gprof*.

```
gcc -pg hello.c -o hello
rm -f gmon.out
./hello
gprof hello
gprof -z hello
gprof -b hello
rm gmon.out
```

5.2 Darbas

PASTABA: jei norit patogiau redaguoti programų tekstus, t. y. naudotis įprastais teksto redaktoriais (ne Word) - galite naudoti:

Windows: WinSCP (<http://winscp.net>), Dokan (<http://dokan-dev.net/en/>)

Linux/BSD: SSHFS (<http://en.wikipedia.org/wiki/Sshfs>)

Naudojant konsolinius redaktorius (nano, vim) galite įsijungti spalvotą sintaksę (nano – susikurkite \$HOME/.nanorc pagal /home/kespaul/.nanorc; vim – spalvota sintaksė įjungta pagal nutylėjimą).

5.2.1 GCC kompiliatorius

Sukurkite savo namų kataloge **\$HOME/lab3/uzs1/src/** katalogą ir jame failus:

```
/* Failas: hello1.c */
int main() {
    return 0;
}
```

```
/* Failas: hello2.c */
#include <stdio.h>
int main() {
    printf( "Hello, World!\n" );
    return 0;
}
```

```
/* Failas: hello3.c */
#include <stdio.h>

int hellofunc( const char * s );

int hellofunc( const char * s ) {
    puts( s );
    return 1;
}

int main() {
    int a = 10, i;
    for( i=0; i < a; i++ ) {
        hellofunc( "Hello, World!" );
    }
    a = 0x762;
    return a;
}
```

Susikurkite darbinį katalogą **\$HOME/lab3/uzs1/work/** ir visas šio užsiėmimo užduotis atlikite jame.

```
mkdir $HOME/lab3/uzs1/work/
cd $HOME/lab3/uzs1/work/
```

Išbandykite (turite būti **\$HOME/lab3/uzs1/work/** kataloge, kad matytumėt, kas kaičiasi – prieš kiekvieną bandymą ištrinkite visus failus iš **\$HOME/lab3/uzs1/work/** katalogo):

```
gcc ../src/hello1.c
```

Sukurti failai: ???

```
gcc ../src/hello1.c -o hello1
```

Sukurti failai: ??? (kokiam tikslui naudojamas -o parametras?)

Išsiaiškinkite **gcc** parametrų prasmę:

--help – ???

-o – ???

-c – ???

-Wall – ???
 -Werror – ???
 -E – ???
 -S – ???
 -pedantic – ???
 -### – ???
 -g – ???
 -pg – ???
 -O0 – ???
 -O2 – ???
 -Os – ???
 -M – ???
 -MM – ???
 -m32 – ???
 -m64 – ???
 -D – ???

Nukopijuokite **hello1.c** į **hello1a.c** ir kopijos 3'ioje eilutėje (return 0) pakeiskite 0 į 1. Sukompiliuokite ir paleiskite abi programas.

Kaip SHELL'e pamatyti, ką gražino **main** funkcija? ???
 Išbandykite:

```
gcc -E hello1.c
gcc -E hello2.c
```

Koki skirtumą pastebite gautuose šių dviejų bylų išeities tekstuose? ??? (kaip -E pakeičia C tekstą)

Pažiūrėkite, kaip kompiliatorius kuria vykdomąjį failą:

```
gcc -### -o hello1 hello1.c
```

Išvestą rezultatą nusikopijuokite ir pažymėkite (paryškinkite) tas vietas, kuriose nurodomas kelias iki kompiliatoriaus, assemblerio, ryšių redaktoriaus (linkerio):

```
gcc -### vykdyimo_logas su išryškintais keliais
```

Naudodamiesi, ankstesniuose punktuose sužinota informacija, sukurkite tarpinius failus:

```
hello2.i
komanda
```

```
hello2.s
komanda
```

```
hello2.o
komanda
```

```
hello2a (vykdomas failas, iš savo sukurtų tarpinių failų, turi veikti, kaip hello2)
komanda
```

5.2.2 ELF analizė ir trasavimas

Sukompiliuokite **hello3** įdėdami debuggeriui skirtą informaciją. Paleiskite debuggerį **gdb**:

```
gdb hello3
```

Naudodami **gdb help** sistemą išsiaiškinkite, ką daro ir pabandykite **gdb** komandas:

- **help** – ???

- **quit** – ???
- **list** – ???
- **run** – ???
- **break** – ???
- **next** – ???
- **step** – ???
- **finish** – ???
- **continue** – ???
- **print** – ???
- **display** – ???
- **x** – ???
- **backtrace** – ???
- **frame** – ???
- **disassemble** – ???
- **nexti** – ???
- **stepi** – ???
- **info** – ???
- **info registers** – ???
- **info target** – ???

Naudodami teorinėj daly aprašytais įrankiais:

suraskite, kokias bibliotekas naudoja *hello2*:

komanda

Naudojamos bibliotekos: ???

suraskite **_start** ir **main** funkcijų adresus, bei nuo kokio adreso prasideda *hello2* vykdymas. Patikrinkite su debugeriu, ar ELF analizės įrankiais gaunate teisingus rezultatus.

komanda

_start: ???

main: ???

hello2 vykdymo pradžia (*entry point*): ???

gdb sesija patikrinimui

Pabandykite su gdb paleisti *hello3* programą taip, kad *hello3* tekstą išvestų į kitą langą, nei paleistas gdb (t. y., kad viename SSH lange matytumėte debugerio interfeisą, o kitame – *hello3* išvedamas eilutes). Bus reikalingas gdb **--tty** parametras.

sesija

Modifikuokite bet kurį iš duotų C failų, kad jis lūžtų (pvz.: įdėkite *abort()* funkcijos iškvietaimą, dalybą iš 0 ar bandymą skaityti iš adreso NULL) ir būtų sukuriamas *core* failas. Sukompiliuokite su debuginimo informacija, gaukite *core* ir patikrinkite, ar galite matyti programos būseną užsaugotą

core (pvz.: ar matot kintamuosius su tokiom reikšmėm, kaip turėjo būti prieš lūžtant programai).

gdb sesija patikrinimui

Išbandykite *gdbtui*.

5.2.3 Profaileris

Sukompiliuokite **hello3.c** su profailerio informacija. Paleiskite, išanalizuokite gautą **gmon.out** failą.

Iškvietimų grafas: kuri funkcija kurią iškvietė ir kiek kartų (pradedant nuo ELF paleidimo adreso).

Dalį funkcijų profaileris rodo, kaip nenaudojamas. Patikrinkite (su debuggeriu), ar tikrai jos neiškviečiamos programos vykdymo metu? Taip/Ne? Jei iškviečiamos – papildykite iškvietimų grafą.

sesija patikrinimui

5.3 Atsiskaitymas

Atsakykite į klausimus <http://moodle.if.ktu.lt>

5.4 Vertinimas

- 50% už tai, kad atsakyta į Moodle klausimus užsiėmimo metu (t. y. už dalyvavimą ir darbą);
- ≤ 50% už atsakymų turinį (testo rezultatas, už darbo kokybę)

6 LD 3 2/5

Tikslas: darbas su failų sistema

6.1 Teorija

Headeriai: <sys/stat.h>, <dirent.h>, <unistd.h>, <sys/statvfs.h>, <fcntl.h>, <stdlib.h>, <stdio.h>, <ftw.h>

MAN: open(2), close(2), perror(), exit(), getcwd(3C), chdir(2), fchdir(2), pathconf(2), fpathconf(2), opendir(3C), fdopendir(3C), readdir(2), closedir(3C), stat(2), fstat(2), statvfs(2), fstatvfs(2), link(2), symlink(2), unlink(2), remove(3C), rename(2), umask(2), mkdir(2), rmdir(2), futimens(2), chmod(2), fchmod(2), nftw(3C)

http://en.wikipedia.org/wiki/File_descriptor

Ankstesnių laboratorinių darbų metu jau manipuliavot failų sistemos turiniu. Tai ką anksčiau darėte SHELL komandų pagalba, galima atlikti ir naudojant POSIX C funkcijas. Anksčiau naudotos SHELL komandos (ir pats SHELL) yra ne kas kita, o C programos, naudojančios tas pačias POSIX C funkcijas.

Kad galėtų atlikti kokius nors veiksmus su failu, katalogu, konvejeriu ar soketu – procesas turi pirmiausia tą objektą **atidaryti**, o baigęs darbą **uždaryti** (jei procesas pasibaigs neuždaręs visų savo objektų – UNIX'as nesugrius, bet kai kuriais atvejais duomenys gali likt neišrašyti). Kiekvienam proceso atidarytam objektui OS priskiria **deskriptorių** (sveiką neneigiamą skaičių). Kiekvienam procesui OS saugo to proceso naudojamų deskriptorių lentelę. Naujai sukurtas procesas iškart gauna tris deskriptorius:

1. **stdin** = 0 – standartinio įvedimo deskriptorius;
2. **stdout** = 1 – standartinio išvedimo deskriptorius
3. **stderr** = 2 – standartinis klaidų deskriptorius

PASTABA: jei stdin ir stderr surišti su tuo pačiu terminalu (pvz.: ir klaidos ir normalus programos išvedimas eina į terminalą), dažnai pasitaiko situacija, kad terminale matome programos išvestą informaciją ne ta tvarka, kaip ją išvedė programa. Taip atsitinka kai stdout buferizuotas, o stderr – ne (t. y. į stdout rašomas tekstas į terminalą išvedamas tik kai užbaigiama eilutė ar užpildomas buferis, o į stderr – iš karto).

Dalis šio užsiėmimo metu naudojamų funkcijų turi savo „antrininkes“: viena funkcija kaip parametą priima simbolių eilutę, o kita – deskriptorių (pvz.: *chmod()* ir *fchmod()*).

6.2 Darbas

Eksperimentams pasidarykite C programos šabloną pagal pavyzdį. Būtinai reikalavimai:

1. Vardas, Pavardė, Grupe failo komentare
2. loginas failo komentare
3. failo vardas su jūsų loginu jo pradžioje
4. visos funkcijos turi turėti Jus identifikuojantį fragmentą pavadinime (programos žingsnius išskirti į atskiras funkcijas)

Šablono pavyzdys:

```
/* Vardas Pavardė GRP loginas */
/* Failas: loginas_sablonas.c */

#include <stdio.h>

int vp_test();

int vp_test(){
    return 0;
}

int main( int argc, char * argv[] ){
    printf( "(C) 2013 Vardas Pavarde, %s\n", __FILE__ );
    return 0;
}
```

6.2.1 open(), close(), perror(), exit()

Išbandykite ir išsiaiškinkite, kaip veikia programa:

```

/* Kęstutis Paulikas KTK kespaul */
/* Failas: kespaul_open01.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int kp_test_open(const char *name);
int kp_test_close(int fd);

int kp_test_open(const char *name){
    int dskr;
    dskr = open( name, O_RDONLY );
    if( dskr == -1 ){
        perror( name );
        exit(1);
    }
    printf( "dskr = %d\n", dskr );
    return dskr;
}


int kp_test_close(int fd){
    int rv;
    rv = close( fd );
    if( rv != 0 ) perror ( "close() failed" );
    else puts( "closed" );
    return rv;
}


int main( int argc, char *argv[] ){
    int d;
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    if( argc != 2 ){
        printf( "Naudojimas:\n %s failas_ar_katalogas\n", argv[0] );
        exit( 255 );
    }
    d = kp_test_open( argv[1] );
    kp_test_close( d );
    kp_test_close( d ); /* turi mesti close klaida */
    return 0;
}


```


Kaip paleisti programą, kad būtų vykdomi *open()* ir *close()* iškvietai:


sesija


Kokia *argc*, *argv*, *argv[0]*, *argv[1]* prasmė: ??? 


Kodėl *dskr* lyginamas su -1 (kur tai parašyta dokumentacijoje): ??? 

Ką daro *open()* funkcija: ??? 

Ką daro *close()* funkcija: ??? 

Ką daro *exit()* funkcija ir ką reiškia jos parametras: ??? 


Ką daro *perror()* funkcija ir ką reiškia jos parametras: ??? 


PASTABA: klaidų pranešimų atvaizdavimui taip pat tinka *fmtmsg()* funkcija. 

6.2.2 pathconf(), fpathconf()

Funkcijos *pathconf()* ir *fpathconf()* leidžia sužinoti konkrečios UNIX OS failų sistemos nustatymus (maksimalūs kelių ilgiai, maksimalus link'ų skaičius ir t.t.). Funkcijoms nurodomas kelias (failas arba katalogas) arba atidarytas deskriptorius.

Sukurkite programą **loginas_pathconf.c** kuri *pathconf()* funkcijos pagalba sužinotų OS parametrus:

Maksimalų failo vardo ilgį (*_PC_NAME_MAX*): ??? 

Maksimalų kelio ilgį (*_PC_PATH_MAX*): ??? 

Patikrinkite gautus rezultatus **getconf** komandos pagalba:

```
getconf NAME_MAX .
```

```
getconf PATH_MAX .
```

6.2.3 getcwd(), chdir(), fchdir()

Išbandykite *getcwd()*, *chdir()*, *fchdir()* funkcijas, pvz.:

```
/* Kęstutis Paulikas KTK kespaul */
/* Failas: kespaul_getcwd01.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int kp_test_getcwd();

int kp_test_getcwd(){
    char *cwd;
    cwd = getcwd( NULL, 1024 );
    puts( cwd );
    free( cwd );
    return 1;
}

int main(){
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    kp_test_getcwd();
    return 0;
}
```

Kodėl kviečiama *free(cwd)*: ???

PASTABA: *getcwd()* veikimas, kai pirmu parametru nurodyta NULL – nestandartinis, kaip veikia žr.: *getcwd(3C)* ant os.if.ktu.lt.

Pataisykite programą, kad vietoj 1024 būtų naudojama *pathconf(".", _PC_PATH_MAX)* grąžinama reikšmė.

Sukurkite C programą **loginas_getcwd02.c** (loginas pakeiskite į savo loginą), kuri:

1. gautų ir atspausdintų einamo katalogo vardą su *getcwd()* (ir kviečiant *getcwd()* naidotų *pathconf(".", _PC_PATH_MAX)* grąžinamą reikšmę)
2. atidarytų einamą katalogą su *open()*, išimintų ir atspausdintų jo deskriptorių
3. nueitu į /tmp katalogą su *chdir()*
4. patikrintų su *getcwd()* ir atspausdintų koks dabar yra einamas katalogas
5. grįžtų į 2-ame žingsnyje atidarytą katalogą su *fchdir()*
6. įkelkite programą į Moodle

6.2.4 opendir(), fdopendir(), readdir(), closedir()

Sukurkite C programą **loginas_readdir01.c** (loginas pakeiskite į savo loginą), kuri:

1. atidarytų einamą katalogą su *opendir()* arba *fdopendir()*;
2. cikle nuskaitytų visus katalogo įrašus su *readdir()* ir išvestų kiekvieno įrašo i-node numerį (dirent struktūros **d_ino** laukas) ir failo vardą (dirent struktūros **d_name** laukas);
3. uždarytų katalogą su *closedir()*;
4. įkelkite programą į Moodle.

6.2.5 stat(), fstat()

Funkcijos *stat()* ir *fstat()* skirtos sužinoti informacijai apie failų sistemos objektus (failus, katalogus, spec. failus, ir t.t.).

Sukurkite programą **loginas_stat01.c** (loginas pakeiskite į savo loginą), kuri su *stat()* ar *fstat()* gautų informaciją apie komandinės elutės parametru jai nurodytą failą, katalogą ar kt. ir išvestų į ekraną *stat* struktūros turinį.

Palyginkite savo programos ir **stat** komandos grąžinamus rezultatus įvairiems failams. Turėtumėte matyti tą pačią informaciją (nebūtinai vienodai atvaizduotą).

6.2.6 statvfs(), fstatvfs()

Šios funkcijos panašios į *stat()* ir *fstat()*, tik grąžina informacija ne apie nurodytą failą, o apie failų sistemą, kurioje tas failas yra.

Nukopijuokite **loginas_stat01.c** į **loginas_statvfs01.c** ir papildykite, kad papildomai būtų išvedama ir *statvfs()* arba *fstatvfs()* grąžinamos struktūros *statvfs* informacija.

Sukurtą programą įkelkite į Moodle.

6.2.7 link(), unlink(), symlink(), remove(), rename(), mkdir(), rmdir(), creat()

Funkcijų pavadinimai akivaizdžiai rodo, kam jos skirtos. Visoms šioms funkcijoms parametrais nurodomi failų ar katalogų pavadinimai, *mkdir()* ir *creat()* papildomai nurodomos teisės, kurios turi būti nustatytos sukurtam katalogui.

PASTABA: jei ištrinamas failas, kurį turi atidarys koks nors procesas – įrašas kataloge panaikinamas, tačiau failo turinio atlaisvinimas atidedamas iki tol, kol jį rodantys deskriptoriai bus uždaryti.

6.2.8 umask(), chmod(), fchmod()

Funkcija *umask()* nustato proceso failų kūrimo maskę, t. y. kai procesas kurdamas failą ar katalogą (su *open()*, *creat()*, ar *mkdir()*) nurodo norimas teises, įvykdoma AND NOT operacija su šiomis teisėmis ir *umask()* nustatyta maske. Pavyzdžiui: norime sukurti failą su teisėmis 0777¹ (*rw-rw-rw-*), tačiau nustatyta maskė 00022, atliekama $00777 \text{ AND } 00022 = 00777 \text{ AND } 07755 = 00755$ (tas pats bitais: $000111111111 \text{ AND } 000000010010 = 000111111111 \text{ AND } 11111101101 = 00011101101$) – gauname failą su *rw-r-x-r-x* teisėmis.

Funkcijos *chmod()* ir *fchmod()* skirtos pakeisti failo teisėms nepriklausomai nuo *umask()* nustatytos maskės.

¹ Skaičius aštuntainėje sistemoje (ne dešimtainėje ar šešioliktainėje).

6.2.9 futimens()

Ši funkcija leidžia pakeisti failo modifikavimo ir paskutinio atidarymo datas. Panašiai veikia ir *utime()* funkcija, tačiau dabartinėje POSIX standarto versijoje *utime()* pažymėta, kaip pasenus (*obsolete*).

Sukurkite programą **loginas_misc01.c** (pakeiskite loginas į savo loginą), kuri demonstruotų bent vienos iš funkcijų *link()*, *unlink()*, *symlink()*, *remove()*, *rename()*, *mkdir()*, *rmdir()*, *creat()*, *umask()*, *chmod()*, *fchmod()*, *futimens()* veikimą

Sukurta programą įkelkite į Moodle.

6.2.10 nftw()

Funkcija *nftw()* skirta katalogų medžio „apvaikščiojimui“ pradedant nuo nurodyto pradinio katalogo ir eina gilyn. Funkcijos antraštė:

```
int nftw(const char *path, int (*fn)(const char *, const struct stat *, int,
struct FTW *), int fd_limit, int flags);
```

Parametrai:

1. **path** – kelias nuo kur pradedamas darbas
2. **fn** – funkcijos vardas, kuri iškviečiama kiekvienam surastam failui ar katalogui apdoroti
3. **fd_limit** – maksimalus deskriptorių skaičius, kuri gali naudoti nftw() (veikia greičiau, jei jis ne mažesnis už maksimalų failų gylį, tačiau tai nebūtina – gali dirbti ir su mažiau deskriptorių)
4. **flags** – vėliavėlės:
 1. FTW_CHDIR – keisti darbinį katalogą (*chdir()*) į kiekvieną apdorojamą katalogą
 2. FTW_DEPTH – pirmiausia eiti gilyn (pirma apdoroti katalogo turinį, o tik po to patį katalogą)
 3. FTW_MOUNT – neitį į kitas failų sistemas (dirbti tik vienoje failų sistemoje)
 4. FTW_PHYS – neiti per simbolines nuorodas (symlinkus).

Funkcijos, iškviečiamos kiekvienam surastam failui ar katalogui antraštė (*fn* gali būti bet koks vardas):

```
int fn(const char *fpath, const struct stat *sb, int tflag, struct FTW
*ftwbuf);
```

Parametrai:

1. **fpath** – surasto failo ar katalogo vardas
2. **sb** – nuoroda į struktūrą su informacija apie failą (žr.: *stat()* funkciją)
3. **tflag** – vėliavėlė duodanti papildomą informaciją apie surastą objektą
 1. FTW_D – surastas katalogas
 2. FTW_DNR – surastas katalogas, bet trūksta teisų jį nuskaityti
 3. FTW_DP – surastas katalogas ir jis jau apvaikščiotas (dėl nurodyto *nftw()* *flags* FTW_DEPTH parametro)
 4. FTW_F – surastas failas
 5. FTW_NS – sb struktūra neužpildyta (*stat()* nepavyko, nes trūko teisų)
 6. FTW_SL – surasta simbolinė nuoroda
 7. FTW_SLN – surasta simbolinė nuoroda, rodanti į neegzistuojantį failą ar katalogą
4. **ftwbuf** – nuoroda į struktūrą su 2 laukais:
 1. **base** – poslinkis iki failo vardo fpath parametru perduotoje eilutėje, t. y. fpath + ftwbuf->base = nuoroda_į_failo_vardą
 2. **level** – kiek giliai failas ar katalogas, skaičiuojant nuo paieškos

pradžios (paieškos pradžia – 0).

Funkcija *nftw()* baigia darbą kai apvaikšto visą katalogų medį, įvyksta klaida (ne dėl teisų trūkumo), arba *fn()* parametru nurodyta funkcija grąžina ne 0 (tokiu atveju *nftw()* grąžina *fn()* grąžintą reikšmę).

Programos pavyzdys:

```
/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_nftw01.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <ftw.h>

int kp_ftwinfo(const char *p, const struct stat *st, int fl, struct FTW
*fbuf);

int kp_ftwinfo(const char *p, const struct stat *st, int fl, struct FTW
*fbuf){
    static int cnt = 0;
    puts( p );
    cnt++;
    if( cnt >= 10 ) return cnt;    /* pabandykite uzkomentuoti sita elute */
    return 0;
}

int main(){
    int rv;
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    rv = nftw( ".", kp_ftwinfo, 20, 0 );
    if( rv == -1 ){
        perror( "nftw failed" );
        exit( 1 );
    }
    if( rv != 0 ){
        printf( "ntfw fn function returnd %d != 0 -> nftw finished\n", rv );
    }
    return 0;
}
```

Sukurkite ir išbandykite programą **loginas_nftw02.c**, kuri su *nftw()* išvaikščiota Jūsų namų katalogą ir atspausdintų visų jame esančių failų pavadinimus (t. y. pradėtų paiešką nuo Jūsų namų katalogo ir naudotų **FTW_PHYS**, kad neišeitų iš jo radus simbolinę nuorodą). Įkelkite programos tekstą į Moodle.

PASTABA: dirbant su failais taip pat gali praversti *fnmatch()* funkcija.

6.3 Atsiskaitymas

Atsakyti į Moodle klausimus.

Įkelti į Moodle Jūsų sukurtus failus (loginas t.b. pakeistas į jūsų tikrą loginą):

1. loginas_pathconf.c
2. loginas_getcwd02.c
3. loginas_readdir01.c
4. loginas_stat01.c
5. loginas_statvfs01.c

6. loginas_misc01.c
7. loginas_nftw02.c

6.4 Vertinimas

Už ne savo sukurtų failų įkėlimą visas užsiėmimas vertinamas 0%.

- 25% už tai kad atsakyta į Moodle klausimus užsiėmimo metu;
- 25% už tai kad įkelti failai
- $\leq 25\%$ už atsakymų turinį
- $\leq 25\%$ už įkeltų programų turinį
 - failai, netenkinantys šablono reikalavimų vertinami 0%
 - + ~15% už stilių (išlaikytas vienodas programavimo stilius, kodas lengvai suprantamas)
 - + ~5% jei programa daro tai, ką turi daryt
 - + ~5% jei programa „atspari kvailiams“ (už tinkamą klaidų apdorojimą)

7 LD 3 3/5

Tikslas: darbas su failais

7.1 Teorija

Headeriai: <stdio.h>, <unistd.h>, <sys/mman.h>, <stdio.h>

MAN: open(2), close(2), read(2), write(2), lseek(2), fsync(3C), fopen(3C), fdopen(3C), fclose(3C), fgetpos(3C), fsetpos(3C), fread(3C), fwrite(3C), fgetc(3C), getc(3C), getchar(3C), fgets(3C), fputs(3C), gets(3C), puts(3C), fscanf(3C), fprintf(3C), scanf(3C), printf(3C), sscanf(3C), sprintf(3C), aio_read(3C), aio_write(3C), aio_return(3C), aio_error(3C), aio_cancel(3C), mmap(2), munmap(2), msync(2)

<http://en.wikipedia.org/wiki/Stdio>

http://en.wikipedia.org/wiki/Asynchronous_I/O

<http://en.wikipedia.org/wiki/Mmap>

Šis užsiėmimas skirtas susipažinimui su UNIX naudojamais failų skaitymo būtais. Visas apžvelgiamas funkcijas galima suskirstyti į grupes pagal veikimo tipą:

1. „paprastas“ žemo lygio sinchroninis skaitymas/rašymas naudojant deskriptorių;
2. buferizuotas skaitymas/rašymas naudojant C *stdio* bibliotekos funkcijas, dirbančias su FILE struktūra (abstraktesnis interfeisas, po apačia naudoja pirmos grupės funkcijas);
3. asinchroninis skaitymas/rašymas;
4. failų „mapinimas“ į proceso adresų erdvę.

Pagal atliekamus veiksmus funkcijas galima suskirstyti:

- atidarymas/uždarymas
- skaitymas/rašymas
- pozicijos faile nustatymas (nuo kur bus vykdoma sekanti skaitymo/rašymo operacija)
- pagalbines (klaidų apdorojimo, buferių „išstūmimo“ į diską ir t.t.).

Visos funkcijų grupės naudoja su **open()** atidarytų failų deskriptorius (išskyrus *stdio* bibliotekos funkcijas, kuriose *open()* gali būti pakeista į *fopen()*). Funkcija *open()* jau pažįstama iš ankstesnio užsiėmimo, tačiau iki šio ją naudojome tik skaitomo failo atidarymui. Pilnas *open()* aprašas:

```
int open(const char *path, int oflag, /* mode_t mode */);
```

Parametras *oflag* nurodo failo atidarymo režimą. Iki šiol naudojome *O_RDONLY*, tačiau yra ir daugiau konstantų, kurios gali būti apjungiamos aritmetine **OR** operacija (ne logine, t. y. OR atliekama kiekvienam bitui). Formaliai pagal POSIX specifikaciją:

1. *oflag* turi būti sudarytas iš tik vienos iš vėliavėlių:
 - a) *O_EXEC*
 - b) *O_RDONLY* – failas atidaromas tik skaitymui**
 - c) *O_RDWR* – failas atidaromas skaitymui ir rašymui**
 - d) *O_SEARCH*
 - e) *O_WRONLY* – failas atidaromas tik rašymui**
2. gali būti (bet nebūtina) su OR prijungta bet kurios vėliavėlės iš šių:

- a) **O_APPEND** – atidarius, operacijų **poslinkis (offset)** nustatomas į failo galą (t. y. kad, rašymo operacija papildytu failą)
- b) **O_CLOEXEC**
- c) **O_CREAT** – jei failo nėra jis **sukuriamas**, o jo **prieigos teisės nusako mode parametras** (įvertinant *umask()* maskę, lygiai taip kaip ankstesnio užsiėmimo metu nagrinėtai *creat()* funkcijai)
- d) **O_DIRECTORY**
- e) **O_DSYNC** – (SIO) ...
- f) **O_EXCL**
- g) **O_NOCTTY**
- h) **O_NOFOLLOW**
- i) **O_NONBLOCK**
- j) **O_RSYNC** – (SIO) ...
- k) **O_SYNC** – (**SIO, Synchronized Input and Output**) papildomi reikalavimai duomenų vientisumo užtikrinimui (supaprastintai: rašymo operacija baigiasi tada, kai duomenys fiziškai įrašomi į diską, o ne į OS kešą). Nemaišyti su *Synchronous I/O* (priešinga sąvoka asinchroniniam I/O) – tai ne tas pats.
- l) **O_TRUNC** – atidaromo failo **turinys ištrinamas** (failas tampa 0 baitų dydžio)
- m) **O_TTY_INIT**

Pavyzdžiui, failo atidarymas rašymui sukuriant, jei tokio nėra ir išvalant duomenis, jei yra:

```
d = open( "failas.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644 );
```

Čia *mode* parametras 0644 – skaičius aštuntainėje sistemoje (ne dešimtainėje ar šešioliktainėje). Vietoj 0644 galima irgi nurodyti išraišką (smulkiau žr.: *chmod(2)*):

```
S_IRUSR | S_IWUSR | S_IRGRP S_IROTH
```

7.2 Darbas

7.2.1 Sinchroninis I/O

Sukurkite programą **loginas_rw01.c**, kuri:

1. atidarytų komandinėje eilutėje nurodytą failą tik skaitymui su *open()*;
2. atidarytų kitą komandinėje eilutėje nurodytą failą tik rašymui (sukurtų, jei nėra, išvalytų turinį jei jau yra);
3. nukopijuotų iš skaitomo failo į rašomą komandinėje eilutėje nurodytą baitų skaičių (jei tiek baitų nėra – tiek kiek yra, t. y. visą failą) naudojant *read()* ir *write()*;
4. uždarytų abu failus su *close()*.

Išbandykite programą su failais ir įrenginiais, pvz.: nuskaitykite 1MB iš */dev/zero* ar */dev/urandom* į failą Jūsų namų kataloge (turėtų gautis nuliais ar atsitiktiniais skaičiais užpildytas 1MB failas).

Sukurkite programą **loginas_seek01.c**, kuri:

1. sukurtų failą (su *open()* ar *creat()*);
2. nueitų į 1MB gilyn į failą su *lseek()*;

3. įrašytų 1 baitą;
4. uždarytų failą su `close()`.

Kokio dydžio failas gavosi (koks jo dydis, ir kiek vietos jis užima diske: ??? (ką apie jį rodo `ls`, `du`, `stat` komandos).

7.2.2 Buferizuotas I/O

Nukopijuokite `loginas_rw01.c` į `loginas_frww01.c` ir pakeiskite, kad vietoj `open()` būtų naudojama `fopen()`, vietoj `close()` – `fclose()`, vietoj `read()` – `fread()`, vietoj `write()` – `fwrite()`.

Išbandykite naują programą. Turėtų gautis toks pat rezultatas.

Kuo skiriasi `fgetc()` ir `getc()`: ???

7.2.3 Asinchroninis I/O

Asinchroninis I/O (AIO) vyksta lygiagrečiai su programos darbu. Programa nurodo OS iš kokio deskriptoriaus kiek duomenų nuskaityti ir kur juos padėti. Apie AIO operacijos pabaigą OS programai gali pranešti signalo pagalba (su signalais dirbsit ateinančiuose užsiėmimuose, todėl kol kas į juos nesigilinsim).

AIO funkcijos, kaip parametą, naudoja **`aioctx`** struktūrą, kurios laukai apibrėžia AIO operacijos nustatymus. Struktūros `aioctx` laukai (tik tie, kurie aktualūs lab. darbui):

1. **`aio_fildes`** – deskriptorius iš kurio bus skaitoma (jis t.b. atidarytas)
2. **`aio_offset`** – pozicija faile, nuo kur bus vykdoma AIO operacija
3. **`aio_buf`** – buferio adresas, į kur dėti nuskaitytus ar iš kur imti rašomus duomenis (vieta buferiui turi išskirta)
4. **`aio_nbytes`** – kiek baitų skaityti ar rašyti (t.b. ne daugiau, nei išskirta vietos buferiui).

AIO funkcijos (smulkia ir tikslią informaciją apie kiekvieną iš jų galite rasti *man* puslapiuose arba POSIX specifikacijoje):

1. **`aio_read`** – pradėti skaitymo AIO operaciją
2. **`aio_write`** – pradėti rašymo AIO operaciją
3. **`aio_suspend`** – sustabdyti programos darbą, kol pasibaigs bent viena parametrais nurodyta AIO operacija (funkcijai paduodamas nuorodų į `aioctx` struktūras masyvas)
4. **`aio_return`** – grąžina nurodytos AIO operacijos rezultatą (`read()` ar `write()` grąžintą reikšmę)
5. **`aio_cancel`** – nutraukti nurodytą AIO operaciją arba visas su nurodytu deskriptoriumi susijusias AIO operacijas
6. **`aio_error`** – grąžina AIO operacijos klaidos kodą (jei klaida buvo)

Toliau pateiktas AIO naudojančios programos pavyzdys. Programos idėja:

1. atidaryti „`/dev/urandom`“ skaitymui
2. su `aio_read()` pabandyti nuskaityti 1MB duomenų (turėtų nuskaityti atsitiktinius duomenis)
3. kol vyksta skaitymas „ką nors nuveikti“ (`kp_test_dummy()` funkcija)
4. patikrinti kas gavosi (ar nuskaite ir kiek duomenų)
5. uždaryti deskriptorių
6. su ta pačia „ką nors nuveikti“ funkcija parodyti, kad duomenys kintamajame, kurį `aio_read()` naudojo nuskaitytų duomenų saugojimui pasikeitė (nors programa pati nieko į kintamąjį nerašė – peršasi išvada – duomenis pakeitė OS vykdydama AIO operaciją)


```

/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_aio01.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <aio.h>

#define BUFFLEN 1048576

int kp_test_open(const char *name);
int kp_test_close(int fd);
int kp_test_aio_read_start( const int d, struct aiocb *aiorp, void *buf,
const int count );
int kp_test_dummy( const void *data, int n );
int kp_test_aio_read_waitcomplete( struct aiocb *aiorp );

int kp_test_open(const char *name){
    int dskr;
    dskr = open( name, O_RDONLY );
    if( dskr == -1 ){
        perror( name );
        exit(1);
    }
    printf( "dskr = %d\n", dskr );
    return dskr;
}

int kp_test_close(int fd){
    int rv;
    rv = close( fd );
    if( rv != 0 ) perror ( "close() failed" );
    else puts( "closed" );
    return rv;
}

int kp_test_aio_read_start( const int d, struct aiocb *aiorp, void *buf,
const int count ){
    int rv = 0;
    memset( (void *)aiorp, 0, sizeof( struct aiocb ) );
    aiorp->aio_fildes = d;
    aiorp->aio_buf = buf;
    aiorp->aio_nbytes = count;
    aiorp->aio_offset = 0;
    rv = aio_read( aiorp );
    if( rv != 0 ){
        perror( "aio_read failed" );
        abort();
    }
    return rv;
}

int kp_test_dummy( const void *data, int n ){
    int i, cnt = 0;

```

```

    for( i = 0; i < n; i++ )
        if( ((char*)data)[i] == '\0' ) cnt++;
    printf( "Number of '0' in data: %d\n", cnt );
    return 1;
}

int kp_test_aio_read_waitcomplete( struct aiocb *aiorp ){
    const struct aiocb *aioptr[1];
    int rv;
    aioptr[0] = aiorp;
    rv = aio_suspend( aioptr, 1, NULL );
    if( rv != 0 ){
        perror( "aio_suspend failed" );
        abort();
    }
    rv = aio_return( aiorp );
    printf( "AIO complete, %d bytes read.\n", rv );
    return 1;
}

int main(){
    int d;
    struct aiocb aior;
    char buffer[BUFFLEN];
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    d = kp_test_open( "/dev/urandom" );
    kp_test_aio_read_start( d, &aior, buffer, sizeof(buffer) );
    kp_test_dummy( buffer, sizeof(buffer) );
    kp_test_aio_read_waitcomplete( &aior );
    kp_test_close( d );
    kp_test_dummy( buffer, sizeof(buffer) );
    return 0;
}

```

Kiek duomenų programa nuskaity: ??? (ar nuskaity užsiprašytą 1MB)

Sukurkite programą **loginas_aio02.c** (galite naudoti pavyzdžio ar savo anksčiau sukurtų programų fragmentus), kuri iš „/dev/urandom“ su `aio_read()` nuskaitytų 1MB duomenų (t. y. tiek kiek prašoma).

1. bus reikalingi pakartotiniai `aio_read()` iškvietimai;
2. reikės keisti adresą kur rašyti duomenis, kad neperrašytų ant jau nuskaitytų.

Pabandykite originalią ir savo programas su „/dev/random“ ir atsitiktinių duomenų failu. 1MB atsitiktinių duomenų failo kūrimas:

```
dd if=/dev/urandom of=atsitiktiniai_duomenys_1MB bs=1024 count=1024
```

7.2.4 Failų „mapinimas“ į RAM

Tai dar vienas OS naudojamas I/O būdas (*memory-mapped file I/O*), kai failai prijungiami į proceso adresų erdvę. Tada procesas gali dirbti su failu lygiai taip, kaip su atmintimi: rašant į atmintį automatiškai rašoma į failą diske ar įrenginį, skaitant – skaitoma iš disko ar įrenginio. Atminties ir failo ar įrenginio turinio sinchronizavimui naudojami OS virtualios atminties valdymo mechanizmai (tie patys, kaip „swapinimo“ valdymui). Naudojamas „vėlyvas“ („*lazy*“) skaitymas ir rašymas, t. y. duomenų blokas nuskaitymas tik tada, kai kreipiamasi į atitinkamą RAM regioną, įrašomi kai iškviečiama `msync()`, failai uždaromi arba OS nuožiūra (pvz.: trūkstam laisvo RAM).

Naudojamos funkcijos:

1. **mmap()** – failo „mapinimas“ (nurodomas atidaryto failo deskriptorius),

- grąžina adresą, nuo kur prasideda failas proceso atminty
2. **munmap()** – failo „numapinimas“ (atminties ir deskriptoriaus sąryšio panaikinimas)
 3. **msync()** – priverstinis duomenų išstūmimas į diską arba OS kešo išvalymas (kad sekantys skaitymai vyktų iš failo, o ne kešo), funkcijai gali būti nurodomos vėliavėlės:
 1. **MS_ASYNC** – rašoma asinchroniškai (*msync()* grįžta iš karto)
 2. **MS_SYNC** – rašoma sinchroniškai (*msync()* grįžta tik kai duomenys fiziškai įrašyti)
 3. **MS_INVALIDATE** – išvalo kešą

Toliau pateiktas programos pavyzdys, kuri sukuria argumentu nurodytą naują 1MB failą (jei toks jau yra – failo neliečia ir grąžina klaidą), prijungia jį prie proceso su *mmap()*, užpildo failą *0xF0* reikšmėm ir uždaro.

```
/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_mmap01.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/time.h>
#include <string.h>

#define SIZE 1048576

int kp_test_openw(const char *name);
int kp_test_close(int fd);
void* kp_test_mmapw( int d, int size );
int kp_test_munamp( void *a, int size );
int kp_test_usemaped( void *a, int size );

int kp_test_openw(const char *name){
    int dskr;
    dskr = open( name, O_RDWR | O_CREAT | O_EXCL, 0640 );
    if( dskr == -1 ){
        perror( name );
        exit( 255 );
    }
    printf( "dskr = %d\n", dskr );
    return dskr;
}

int kp_test_close(int fd){
    int rv;
    rv = close( fd );
    if( rv != 0 ) perror ( "close() failed" );
    else puts( "closed" );
    return rv;
}

void* kp_test_mmapw( int d, int size ){
    void *a = NULL;
    lseek( d, size - 1, SEEK_SET );
    write( d, &d , 1 ); /* įraso bile ka i failo gala */
}
```

```

a = mmap( NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, d, 0 );
if( a == MAP_FAILED ){
    perror( "mmap failed" );
    abort();
}
return a;
}

int kp_test_munamp( void *a, int size ){
    int rv;
    rv = munmap( a, size );
    if( rv != 0 ){
        puts( "munmap failed" );
        abort();
    }
    return 1;
}

int kp_test_usemaped( void *a, int size ){
    memset( a, 0xF0, size );
    return 1;
}

int main( int argc, char * argv[] ){
    int d;
    void *a = NULL;
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    if( argc != 2 ){
        printf( "Naudojimas:\n %s failas\n", argv[0] );
        exit( 255 );
    }
    d = kp_test_openw( argv[1] );
    a = kp_test_mmapw( d, SIZE );
    kp_test_usemaped( a, SIZE );
    kp_test_munamp( a, SIZE );
    kp_test_close( d );
    return 0;
}

```

Sukurkite programa **loginas_mmap02.c**, kuri nukopijuotų failus naudojant *mmap()* (kad būtų paprasčiau laikykime, kad failų dydžiai iki 100MB, t. y. abu telpa į 32bit proceso erdvę):

1. atidarytu ir prijungtu 2 programos argumentais nurodytus failus su *mmap()* (vieną iš jų tik skaitymui, tik skaitomo failo dydį galite sužinoti su *fstat()* funkcija)
2. nukopijuotų vieno failo turinį į kitą (su *memcpy()* ar paprastu ciklu)
3. atjungtu abu failus
4. uždarytu abu deskriptorius

7.3 Atsiskaitymas

Atsakyti į Moodle klausimus.

Ikelti savo sukurtas programas į Moodle:

1. loginas_rw01.c
2. loginas_seek01.c
3. loginas_fr01.c
4. loginas_aio02.c
5. loginas_mmap02.c

7.4 Vertinimas

Už ne savo sukurtų failų įkėlimą visas užsiėmimas vertinamas 0%.

- 25% už tai kad atsakyta į Moodle klausimus užsiėmimo metu;
- 25% už tai kad įkelti failai
- $\leq 25\%$ už atsakymų turinį
- $\leq 25\%$ už įkeltų programų turinį
 - failai, netenkinantys šablono reikalavimų vertinami 0%
 - + ~15% už stilių (išlaikytas vienodas programavimo stilius, kodas lengvai suprantamas)
 - + ~5% jei programa daro tai, ką turi daryt
 - + ~5% jei programa „atspari kvailiams“ (už tinkamą klaidų apdorojimą)

8 LD 3 4/5

Tikslas: darbas su procesais (limitais, skaitliukais), core, atmintis.

8.1 Teorija

Headeriai: <stdlib.h>, <unistd.h>, <sys/resource.h>, <time.h>, <dlfcn.h>

MAN: sysconf(3C), getrlimit(), setrlimit(), exit(), atexit(), _exit(), _Exit(), abort(), time(), clock(), strptime(), strftime(), difftime(), gmtime(), localtime(), mktime(), getdate(), getrusage(), times(), sleep(), nanosleep(), nice(), getpriority(), setpriority(), dlopen(), dlclose(), dlsym(), dlerror()

http://en.wikipedia.org/wiki/Dynamically_loaded_library

http://en.wikipedia.org/wiki/Position-independent_code

http://en.wikipedia.org/wiki/Address_space_layout_randomization

http://en.wikipedia.org/wiki/X86_calling_conventions

http://en.wikipedia.org/wiki/Calling_convention

8.1.1 Dinaminis vykdomų duomenų užkrovimas

Apie ELF failo turinį, jo užkrovimą į RAM ir paleidimą jau buvo kalbėta, tačiau iki šiol žiūrėjome iš pagrindinės programos pusės. Žiūrint iš dinaminių modulių pusės yra keletas specifinių detalių:

1. visi dinaminės bibliotekos objektai, kurie skirti naudojimui iš pagrindinės programos, turi būti matomi dinaminiam ryšių redaktoriui (t. y. eksportuoti)
2. vykdomas kodas bibliotekoje turi būti tinkamas vykdymui nepriklausomai koku adresu biblioteka bus užkrauta į RAM (**Position Independent Code – PIC**).

Reikalavimas, kad bibliotekos kodas būtų PIC atsiranda todėl, kad iš anksto nežinome, koku adresu biblioteka bus užkrauta į proceso adresų erdvę. Be to bibliotekos kodo puslapis fiziniame RAM puslapyje į skirtinguose procesuose gali būti matomas skirtinguose adresuose (dėl MMU).

PIC kodas gali būti naudojamas ne vien bibliotekoms. Jo privalumas – galimybė atsitiktinai išdėstyti kodą, duomenis, steką ir t.t. proceso adresų erdvėje, taip apsunkinant įsilaužimą į sistemą ar privilegijų pakėlimą pasinaudojant programos klaidomis.

Nors laboratorinio darbo metu neturėtų dėl to kilti problemų, visgi reiktų žinoti apie galimas problemas maišant skirtingais kompiliatoriais sugeneruotą kodą. Yra ne vienas būdas, kaip perduoti parametrus iškvičiamai funkcijai (**calling convention**). Kuris būdas naudojamas priklauso nuo konkretaus kompiliatoriaus arba **ABI** (*Application Binary Interface*). Pagrindinės *calling convention* savybės:

1. kaip perduodami argumentai: registrais, per steką, ...
2. kokia tvarka argumentai talpinami į steką
3. kas atsakingas už steko atlaisvinimą: kviečianti ar iškviesta funkcija
4. papildomos savybės: kaip stekas žemiau steko registro gali būti naudojamas funkcijos kintamiesiems saugoti (**red zone**); steko išlyginimas (**alignment**)

8.1.1.1 Dinaminės bibliotekos kūrimas

Dinaminės bibliotekos išeities tekstas iš esmės niekuo nesiskiria nuo paprastos programos teksto. Kadangi bibliotekos funkcijų ir kintamųjų aprašai bus reikalingi ir kviečiančioje programoje, patogumo dėlei jie iškelti į atskirą *kspaul_lib01.h* header failą.

```

/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_lib01.h */

#ifndef kespaul_lib01_h
#define kespaul_lib01_h

/* apie "extern" zr.: C99.pdf 6.2.2 */
extern int kespaul_libfunc01( const char *s );

extern double kespaul_libvar01;

#endif

```

```

/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_lib01.c */

#include <stdio.h>
#include "kespaul_lib01.h"

double kespaul_libvar01;

int kespaul_libfunc01( const char *s ){
    printf( "Dynamic library for testing, %s\n", __FILE__ );
    printf( "\tparameter: \"%s\"\n", s );
    printf( "\tlib variable = %f\n", kespaul_libvar01 );
    return 0;
}

```

Kompiliuojant dinaminę biblioteką reikalingi papildomi parametrai:

1. **-fpic** – nurodo, kad generuojamas PIC mašininis kodas;
2. **-shared** – nurodo, kad kuriamas dinaminis failas, kuris gali būti su kitais formuojant procesą;
3. **-o libVARDAS.so** – jau iš anksčiau pažįstamas *-o* parametras, bet jam nurodomas kuriamos bibliotekos vardas.

Kompiliavimas:

```
gcc -fpic -Wall -pedantic -shared -o libkespaul01.so kespaul_lib01.c
```

PASTABA: vietoj *-fpic* veikia ir *-fPIC*. Pagal gcc manualą ir FreeBSD 9.0 /usr/share/mk/bsd.lib.mk atrodo, kad *-fPIC* reikalingas tik sparc64 ir gal kokiai egzotikai.

8.1.1.2 Bibliotekos prikabinimas ryšių redaktoriaus pagalba

Pagrindinėje programoje includinamas tas pats dinaminės bibliotekos *kespaul_lib01.h* header failas (pavyzdyje jis padėtas einamojo katalogo *lib* pakatalogyje).

```

/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_test_lib01a.c */

#include <stdio.h>
#include "lib/kespaul_lib01.h"

int main(){
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    kespaul_libfunc01( "Library test 1" );
    kespaul_libvar01 = 5.1;
    kespaul_libfunc01( "Library test 2" );
    return 0;
}

```

```

gcc -g -Wall -Werror -pedantic -Llib -o kespaul_test_lib01a \
    kespaul_test_lib01a.c -lkespaul01

```

Kompiliuojant panaudoti papildomi parametrai:

1. **-L** – nurodo, kur papildomai ieškoti bibliotekų, kurios turi būti prikabinotos prie kuriamo vykdomojo failo (šiuo atveju nurodyta, kad papildomai ieškoti *lib* pakatalogyje einamajame kataloge);
2. **-l** – nurodo kokias bibliotekas papildomai prikabinti prie kuriamo vykdomojo failo (šiuo atveju nurodyta *kespaul01*, t.y. ieškos *libkespaul01.so* failo).

Gautas vykdomasis failas *kespaul_test_lib01a* naudos *libkespaul01.so* biblioteką, kurios nėra */lib*, */usr/lib* ir panašiuose kataloguose sistemoje, todėl reikia dinaminių ryšių redaktoriui **LD_LIBRARY_PATH** aplinkos kintamojo pagalba nurodyti, kur dar ieškoti trūkstamų bibliotekų. Paleidimas:

```

export LD_LIBRARY_PATH=${PWD}/lib
./kespaul_test_lib01a

```

PASTABA: os.if.ktu.lt LD_LIBRARY_PATH pagal nutylėjimą nenustatytas, bet jei sistemoje šiam kintamajam jau priskirta reikšmė, reiktų ją tik papildyti, pvz.: **export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:\${PWD}/lib**

Jei neturite galimybės įdėti bibliotekos į */lib*, */usr/lib* ar panašų katalogą, kur dinaminių ryšių redaktorius bibliotekų ieško pagal nutylėjimą, galima kompiliavimo metu ryšių redaktoriaus parametru nurodyti kur papildomai ieškoti bibliotekų. Tuomet **LD_LIBRARY_PATH** nustatinėti nebereikės. Be to nebūtina nurodyti absoliutų kelią, veikia ir santykinis (bet nuo katalogo iš kur paleidžiamas vykdomasis failas, o ne to, kuriame yra vykdomasis failas). Šia galimybe nereikėtų piktnaudžiauti – šitaip įrašytus kelius būna iškrapštyt. Kelias įrašomas ryšių redaktoriaus (parametras **-Wl**) parametru **-rpath**:

```

gcc -g -Wall -Werror -pedantic -Llib -o kespaul_test_lib01a \
    kespaul_test_lib01a.c -lkespaul01 -Wl,-rpath=${PWD}/lib

```

Kaip jau buvo minėta, dinaminės bibliotekos išeities tekstas iš esmės nesiskiria nuo programos išeities teksto – šį pavyzdį galima sukompiliuoti į vieną vykdomą failą visiškai nenaudojant atskiros *libkespaul01.so* bibliotekos:

```

gcc -g -Wall -Werror -pedantic -o kespaul_test_lib01a2 \
    kespaul_test_lib01a.c lib/kespaul_lib01.c

```

8.1.1.3 Bibliotekos užkrovimas vykdymo metu

POSIX leidžia dinamines bibliotekas užkrauti ir nesinaudojant dinaminių ryšių redaktoriumi. Tam

naudojamos funkcijos:

1. `dlopen()` – atidaro nurodytą dinaminę biblioteką ir įjungia į proceso adresų erdvę
2. `dlclose()` – uždaro dinaminę biblioteką ir atlaisvina jai skirtus adresus
3. `dlsym()` – prikabintoje bibliotekoje ieško nurodytos funkcijos ar kintamojo pagal nurodytą vardą
4. `dlerror()` – įvykus klaidai grąžina eilutę su klaidos aprašymu

Užkomentuotas `dlsym()` iškvietimas ir po jo einanti eilutė daro tą patį. C standartas neleidžia `void*` konvertavimo į funkcijos rodyklę, todėl užkomentuotai eilutei griežtai standarto reikalavimus atitinkantis kompiliatorius mes perspėjimą.

```
/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_test_lib01b.c */

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int (*fptr)(const char *s);

double *pd;

int main(){
    void *dl;
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    dl = dlopen( "lib/libkespaul01.so", RTLD_LAZY | RTLD_LOCAL );
    if( dl == NULL ){
        puts( dlerror() );
        exit(1);
    }
    pd = dlsym( dl, "kespaul_libvar01" );
    if( pd == NULL ){
        puts( dlerror() );
        exit(1);
    }
    /* fptr = (int (*)(char*)) dlsym( dl, "kespaul_libfunc01" ); */
    *(void**)&fptr = dlsym( dl, "kespaul_libfunc01" );
    if( fptr == NULL ){
        puts( dlerror() );
        exit(1);
    }
    *pd = 5.2;
    (*fptr)( "Library test (manualy loaded)" );
    dlclose( dl );
    return 0;
}
```

Šio failo kompiliavimas įprastas. Naudojamos dinaminės bibliotekos kompiliavimo metu niekaip nurodyti nereikia, bibliotekos header failas irgi nenaudojamas.

8.2 Darbas

8.2.1 Informacija apie procesams taikomus apribojimus

Funkcija `sysconf()` labai panaši į pirmo šio laboratorinio darbo užsiėmimo metu nagrinėtą

pathconf() funkcija, tik grąžina informaciją ne apie failų sistemą, o apie visą sistemą (t. y. grąžina parametrus, kurie nepriklauso nuo to, kuri failų sistema naudojama).

confstr() – analogiška *sysconf()*, tik nustatymams, kurie grąžina ne sveiko skaičiaus reikšmes, o nuorodas į simbolių eilutes.

sysconf() naudojimo pavyzdys:

```
/* Kęstutis Paulikas KTK kespaul */
/* Failas: kespaul_limits01.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int kp_memsize();

int kp_memsize(){
    long pagesize, pages;
    pagesize = sysconf( _SC_PAGESIZE );
    pages = sysconf( _SC_PHYS_PAGES );
    printf( "System:\n\tpage size: %ld\n\tpages: %ld\n", pagesize, pages );
    printf( "Total system RAM: %.1f MB\n", (double)pagesize * pages / 1024 /
1024 );
    return 1;
}

int main(){
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    kp_memsize();
    return 0;
}
```

Funkcijos **getrlimit()** ir **setrlimit()** leidžia sužinoti ir pakeisti procesui taikomus resursų apribojimus: maksimalų *core* failo dydį; maksimalų procesui skirtą CPU laiką; maksimalų deskriptorių skaičių, max atminties dydį, max kuriamo failo dydį ir t.t.

getrlimit() ir *setrlimit()* naudojimo pavyzdys:

```
/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_limits02.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/resource.h>

int kp_change_filelimit( int nslimit, int nhlimit );
int kp_test_filelimit( const char *fn );

int kp_change_filelimit( int nslimit, int nhlimit ){
    struct rlimit rl;
    getrlimit( RLIMIT_NOFILE, &rl );
    printf( "RLIMIT_NOFILE %ld (soft) %ld (hard)\n", rl.rlim_cur, rl.rlim_max
);
}
```

```

    rl.rlim_cur = nsrlimit;
    rl.rlim_max = nhrlimit;
    setrlimit( RLIMIT_NOFILE, &rl );
    getrlimit( RLIMIT_NOFILE, &rl );
    printf( "RLIMIT_NOFILE %ld (soft) %ld (hard)\n", rl.rlim_cur, rl.rlim_max
);
    return 1;
}

int kp_test_filelimit( const char *fn ){
    int n;
    for( n = 0; -1 != open( fn, O_RDONLY ); n++ );
    printf( "Can open %d files\n", n );
    return 1;
}

int main( int argc, char *argv[] ){
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    kp_change_filelimit( 20, 100 );
    kp_test_filelimit( argv[0] );
    return 0;
}

```

Sukurkite programą, kuri **loginas_cpulimit01.c** kuri nustatytu CPU limitą 5s (*RLIMIT_CPU*) ir patikrinkite, ar limitas suveikia (nustatę limitą užsukite amžiną ciklą su skaitliuku).

Suveikus limitui programa turėtų mest *core*.

Kiek iteracijų padarė amžinas ciklas: **???** (galite tai sužinoti iš *core* failo; jei įtariate, kad skaitliukas persipildė – pabandykite perkompiliuoti į *64bit* su kompiliatoriaus parametru *-m64* arba sumažinti CPU laiko limitą).

Pataisykite, kad programa nemestų *core* (nustatykite *RLIMIT_CORE* limitą į 0)

Įkelkite programą į Moodle.

8.2.2 Programos darbo užbaigimas

Yra ne vienas C programos užbaigimo būdas. Su dauguma jau susidūrėte: *return* iš *main()* funkcijos, *exit()*, *abort()*. Yra dar vienas – naudojant ***_exit()*** arba ***_Exit()*** funkciją (jų abiejų veikimas identiškas). Ši funkcija skirta „grubiam“ išėjimui iš programos, t. y. neatliekant jokių veiksmų išskyrus resursų atlaisvinimą.

Sukurkite programą **loginas_exit01.c** ir joje su *atexit()* priregistruokite kelias savo funkcijas (bent 3). Pagal programai paduotą parametą išeikite su *_Exit()*, *exit()*, *abort()* arba *return*.

Kaip veikia: išskviečia priregistruotas funkcijas, neišskviečia, išskviečia ne visas, kokia tvarka išskviečia???

Kaip keičiasi priregistruotų funkcijų iškvietimas, jei iš programos išeinama su *_Exit()*, *exit()*, *return*, *abort()*: **???** (kuriais atvejais išskviečiama, kuriais ne)

Įkelkite programą į Moodle.

8.2.3 Laiko informacija ir jos apdorojimas

PASTABA: jei reikia surasti, kaip tiksliai aprašytas tipas – galima naudoti **gcc -E kazkoks.h** ir rezultatuose ieškoti aprašo. Kitas būdas – pasidaryt minimalią programą ir analizuot duomenis su debuggeriu.

Laiko informacija naudojama dviem tikslams: absoliučiam laikui ir resursų (CPU) naudojimui sekti. Naudojami tipai:

1. **time_t** – sekundės nuo 1970 sausio 1 00:00 UTC1 (UNIX Epochos pradžios),

- sveikas skaičius (*long*)
- 2. **clock_t** – mikrosekundės (*long*)
- 3. **struct tm** – struktūra su laukais (datoms, visi laukai *int* tipo):
 - 1. **tm_sec** – sekundė (0 – 60)
 - 2. **tm_min** – minutė (0 – 59)
 - 3. **tm_hour** – valanda (0 – 23)
 - 4. **tm_mday** – mėnesio diena (1 – 31)
 - 5. **tm_mon** – mėnuo (0 – 11)
 - 6. **tm_year** – metai nuo 1900 (t. y. reikia +1900, kad gaut realius)
 - 7. **tm_wday** – savaitės diena (0 – 6, 0=sekmadienis)
 - 8. **tm_yday** – metų dieną (0 – 365)
 - 9. **tm_isdst** – vasaros laiko vėliavėlė (>0 – vasaros laikas, =0 – žiemos laikas, <0 – jei nėra informacijos)
- 4. **struct tms** – struktūra su laukais (visi laukai *clock_t* → *long* tipo):
 - 1. **tms_utime** – CPU laikas praleistas vartotojo režime
 - 2. **tms_stime** – CPU laikas praleistas sistemos režime
 - 3. **tms_cutime** – proceso vaikų CPU laikas praleistas vartotojo režime
 - 4. **tms_cstime** – proceso vaikų CPU laikas praleistas sistemos režime
- 5. **struct timeval** – struktūra su laukais (resursų apskaitai):
 - 1. **tv_sec** – sekundės (*time_t* → *long*)
 - 2. **tv_usec** – mikrosekundės (*useconds_t* → *long*)
- 6. **struct rusage** – struktūra resursų apskaitai su laikais:
 - 1. **ru_utime** – CPU laikas praleistas vartotojo režime (*struct tm*)
 - 2. **ru_stime** – CPU laikas praleistas sistemos režime (*struct tm*)
 - 3. ... kiti laukai nestandartizuoti

Naudojamos funkcijos:

- 1. laiko informacijai gauti
 - 1. **clock()** – (*clock_t*) grąžina proceso sunaudotą laiką (user+system)
 - 2. **time()** – (*time_t*) dabartinis laikas nuo 1970.01.01
 - 3. **getrusage()** – (*rusage*) grąžina proceso arba jo vaikų sunaudotą CPU laiką
 - 4. **times()** – (*tms*) grąžina proceso ir jo vaikų sunaudotą CPU laiką
- 2. laiko informacijai konvertuoti
 - 1. **ctime()** – *time_t* → *char** (vietinis laikas)
 - 2. **asctime()** – *tm* → *char**
 - 3. **localtime()** – *time_t* → *tm* (vietinis laikas)
 - 4. **gmtime()** – *time_t* → *tm* (UTC)
 - 5. **mktime()** – *tm* → *time_t*
 - 6. **strftime()** – *tm* → *char**
 - 7. **strptime()** – *char** → *tm*

Norint su **clock()** išmatuoti proceso sunaudotą CPU laiką konkrečiam darbui, *clock()* reikia iškviešti prieš pradedant darbą ir po jo. Sunaudotą laiką rodo šių dviejų reikšmių skirtumas. Kad paversti laiką į sekundes reikia gautą skaičių padalinti iš **CLOCKS_PER_SEC**. Programos pavyzdys:

```
/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_clock01.c */

#include <stdio.h>
#include <time.h>

int kp_test_clock();
```

```

int kp_test_clock(){
    clock_t start, end, diff;
    int i;
    start = clock();
    for( i = 0; i< 0x7FFFFFFF; i++);
    end = clock();
    diff = end - start;
    printf( "CLOCKS_PER_SEC = %ld\n start=%ld\n end=%ld\n diff=%ld, diff(s)=
%f\n", CLOCKS_PER_SEC, start, end, diff, (double)diff / CLOCKS_PER_SEC );
    return 1;
}

int main( int argc, char * argv[] ){
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    kp_test_clock();
    return 0;
}

```

Programa ant *os.if.ktu.lt* užtrunka ~5-10s. Užlaikymo ciklo reikšmė **0x7FFFFFFF** – maksimalus teigiamas *32bit* skaičius, todėl jei norite naudoti didesnę arba kompiliuokite *64bit*, arba keiskite programos logiką.

Sukurkite programą **loginas_clock02.c**, kuri atspausdintų dabartinį laiką lokaliai ir UTC (GMT) formatais (pvz.: naudojant *time()*, *localtime()*, *gmtime()*, *asctime()* funkcijas).

Programą įkelkite į Moodle.

Programos darbo rezultatas:

sesija

8.2.4 Vykdyto prioritetas

Sukurkite programą **loginas_nice01.c**, kuri:

1. nustatytų savo prioritetą su *nice()* į programos parametru nurodytą (galite naudoti *atoi()* funkciją parametro konvertavimui į *int*)
2. užsuktų maždaug 5s trunkantį ciklą (iteracijų skaičių galite parinkti eksperimentiškai arba sužinoti iš *loginas_cpulimit01.c* failo)
3. atspausdintų pradžios ir pabaigos laikus bei jų skirtumą
4. atspausdintų informaciją apie proceso sunaudotą CPU laiką (*user + system*).

Programą įkelkite į Moodle.

Paleiskite vienu metu dvi *loginas_nice01* programas iš skirtingų sesijų (skirtingų SSH langų) – vieną 0 prioritetu, kitą – +19 prioritetu. Pateikite abiejų atspausdintus atsakymus:

sesija1

sesija2

Kiek maždaug skiriasi programų vykdymo laikas: ???

8.2.5 Dinaminis užkrovimas

Perdarykite bet kurias dvi anksčiau darytas C programas į **loginas_testlib02a.c**, **loginas_testlib02b.c** ir **loginas_testlib02.h**, taip, kad galėtumėte sukurti dinamines bibliotekas *libloginas02a.so* ir *libloginas02b.so* ir kad kiekviena iš jų turėtų bent vieną funkciją tokiu pat vardu, parametrais ir grąžinama reikšme (pvz.: *int vp_testlib(int a);*). Kitaip sakant abiejų bibliotekų kompiliavimui turėtų tikt tas pats *loginas_testlib02.h* header failas.

kompiliavimo sesija

Sukurkite testinę programą **loginas_libtest02.c**, kuri naudotų bent vieną biblioteką ir iškvieštų jos funkciją (dinaminio ryšių redaktoriaus pagalba).

kompiliavimo sesija

vykdymo sesija

Sukurkite testinę programą **loginas_dynload02.c**, kuri vykdymo metu (nieko neperkompilijuojant) galėtų užkrauti komandinėje eilutėje nurodytą biblioteką ir iškvieštų jos funkciją.

vykdymo sesija1

vykdymo sesija2

Sukurtų programų tekstus įkelkite į Moodle.

8.3 Atsiskaitymas

Atsakyti į Moodle klausimus.

Įkelti savo sukurtas programas į Moodle:

1. **loginas_cpulimit01.c**
2. **loginas_exit01.c**
3. **loginas_clock02.c**
4. **loginas_nice01.c**
5. **loginas_testlib02a.c**
6. **loginas_testlib02b.c**
7. **loginas_testlib02.h**
8. **loginas_libtest02.c**
9. **loginas_dynload02.c**

8.4 Vertinimas

Už ne savo sukurtų failų įkėlimą visas užsiėmimas vertinamas 0%.

- 25% už tai kad atsakyta į Moodle klausimus užsiėmimo metu;
- 25% už tai kad įkelti failai
- $\leq 25\%$ už atsakymų turinį
- $\leq 25\%$ už įkeltų programų turinį
 - failai, netenkinantys šablono reikalavimų vertinami 0%
 - + ~15% už stilių (išlaikytas vienodas programavimo stilius, kodas lengvai suprantamas)
 - + ~5% jei programa daro tai, ką turi daryt
 - + ~5% jei programa „atspari kvailiams“ (už tinkamą klaidų apdorojimą)

9 LD 3 5/5

Tikslas: darbas su tinklu

9.1 Teorija

Headeriai: <sys/socket.h>, <arpa/inet.h>, <netdb.h>

MAN: socket(3xnet), socketpair(3xnet), bind(3xnet), getsockname(3xnet), listen(3xnet), accept(3xnet), connect(3xnet), shutdown(3xnet), send(3xnet), sendto(3xnet), sendmsg(3xnet), recv(3xnet), recvfrom(3xnet), recvmsg(3xnet), htonl(3xnet), htons(3xnet), ntohl(3xnet), ntohs(3xnet), inet_addr(3xnet), inet_ntoa(3xnet), inet_ntop(3xnet), inet_pton(3xnet), getaddrinfo(3xnet), getnameinfo(3xnet), freeaddrinfo(3xnet), gai_strerror(3xnet)

POSIX.zip: [susv4/functions/V2_chap02.html#tag_15_10](http://www.susv4.org/functions/V2_chap02.html#tag_15_10)

http://en.wikipedia.org/wiki/Network_socket

http://en.wikipedia.org/wiki/Transport_Layer

http://en.wikipedia.org/wiki/OSI_model

http://en.wikipedia.org/wiki/Unix_domain_socket

<http://www.thomasstover.com/uds.html>

http://en.wikipedia.org/wiki/Berkeley_sockets

<http://en.wikipedia.org/wiki/Endianness>

<http://tools.ietf.org/html/rfc791>

Pagal RFC 791 (*Internet Protocol*) APPENDIX B: *Data Transmission Order*, duomenų perdavimui tinklu naudojamas **big endian** kodavimas. Kadangi skirtingi CPU naudoja įvairius kodavimus (x86 – *little endian*, ARM – g.b. perjungtas ir į *big endian*, ir į *little endian*), POSIX standarte yra numatytos funkcijos 2 ir 4 baitų duomenų konvertavimui iš CPU naudojamos baitų tvarkos į IP tinkle naudojamą baitų tvarką: **htons()**, **htonl()** ir atgal: **ntohs()**, **ntohl()**. Įvairios OS turi funkcijas ir 8 baitų konvertavimui, bet jos nestandartinės. Visos šios funkcijos leidžia lengviau kurti programas, kurios korektiškai dirbs su IP tinklu ir *big endian*, ir *little endian* kodavimą naudojančiose mašinose.

9.1.1 Soketai

Soketu (**socket**) priklausomai nuo konteksto vadinami du susiję, bet ne vienodi dalykai:

- OS požiūriu – soketas, tai objektas aprašantis ryšio kanalo galinį tašką (IP atveju soketas turi IP adresą, portą, protokolą, TCP protokolo atveju papildomai gali turėti (jei yra sujungimas – *established connection*) nutolusio taško adresą ir portą);
- programos požiūriu – soketas, tai sveikas skaičius, identifikuojantis programos atidarytą objektą su kuriuo galima dirbti labai panašiai, kaip su failų deskriptoriumi (skaityti, rašyti, uždaryti ir t.t.), kitaip sakant soketas ir yra deskriptorius, bet susietas su kitokiu OS objektu.

Kompilijuojant programą, naudojančią soketus ant **os.if.ktu.lt** (*Solaris 11*) turi būti papildomai prijungta *libxnet.so* biblioteka (arba *libsocket.so* ir *libnsl.so* bibliotekos), pvz:

```
gcc -lxnet ...
```

Pagrindinės soketų API funkcijos:

```
int socket(int domain, int type, int protocol);
```

Funkcija sukuria naują soketą. Parametrai:

- **domain** – nurodo kokia protokolų šeima bus naudojama:
 - **AF_UNIX** – lokali mašina
 - **AF_INET** – IPv4
 - **AF_INET6** – IPv6
- **type** – komunikavimo tipą nurodantis parametras:
 - **SOCK_STREAM** – patikimas, dvikryptis baitų kanalas, gali būti palaikomas „out-of-band“ duomenų perdavimas (*connection*);
 - **SOCK_DGRAM** – nepatikimas paketų perdavimas (*connectionless*);
 - **SOCK_SEQPACKET** – patikimas dvikryptis kanalas įrašų perdavimui (*connection*);
 - **SOCK_RAW** – (POSIX neprivalomas) leidžia prieiti prie perdavimo protokolo informacijos, bet jų naudojimui reikia *root* teisių;
- **protocol** – paprastai yra tik vienas nurodyto tipo protokolas šeimoje, todėl paprastai parametru nurodomas 0 (išimtis – **SOCK_RAW**, kuriam reikia nurodyti konkretų protokolą).

```
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

Funkcija soketui priskiria adresą soketui. Parametrai:

- **socket** – soketas, kuriam priskiriamas adresas;
- **address** – struktūra nusakanti adresą, struktūros dydis ir laukai priklauso nuo protokolų šeimos
- **address_len** – adreso struktūros dydis (sveikas skaičius)

```
int listen(int socket, int backlog);
```

Funkcija padaro, kad soketas lauktų prisijungimų (klausytųsi). Parametrai:

- **socket** – soketas;
- **backlog** – kiek leisti laukiančių prisijungimų eilėje;

```
int accept(int socket, struct sockaddr *address, socklen_t *address_len);
```

Funkcija iš laukiančių prisijungimų eilės paima sekantį ir sukuria „sujungimo soketą“ (*connection socket*). Parametrai:

- **socket** – besiklausantis soketas;
- **address** – NULL, arba nuoroda į struktūrą, į kuria funkcija įrašo kliento soketo informaciją;
- **address_len** – adreso struktūros dydis (iškviečiant čia įrašytas adreso struktūros dydis, grįžus – įrašytos adreso struktūros dydis)

```
int shutdown(int socket, int how);
```

Funkcija nutraukia soketų sujungimą (pilnai arba tik viena kryptimi). Parametrai:

- **socket** – sujungtas soketas;
- **how** – nutraukimo tipas:
 - **SHUT_RD** – uždraudžia tolimesnį skaitymą iš soketo (priėmimą);
 - **SHUT_WR** – uždraudžia tolimesnį rašymą į soketą (siuntimą);
 - **SHUT_RDWR** – uždraudžia tolimesnį duomenų perdavimą abiem kryptimis


```
int connect(int socket, const struct sockaddr *address, socklen_t
address_len);
```

Funkcija sujungia soketus. Parametrai:

- **socket** – sukurtas soketas;
- **address** – nuoroda į struktūrą, aprašančią kitos pusės adresą;
- **address_len** – adreso struktūros dydis

```
ssize_t send(int socket, const void *message, size_t length, int flags);
```

```
ssize_t sendto(int socket, const void *message, size_t length, int flags,
const struct sockaddr *dest_addr, socklen_t dest_len);
```

```
ssize_t sendmsg(int socket, const struct msghdr *msg, int flags);
```

Funkcijos duomenų siuntimui per soketą, *send()* su *flags=0* veikia taip pat, kaip *write()*. Parametrai:

- **socket** – soketas per kurį siunčiama (į kurį rašoma);
- **message** – siunčiami duomenys;
- **length** – *message* dydis;
- **flags** – vėliavėlės nusakančios perdavimo tipą;
- **dest_addr** – nuoroda į gavėjo adresą nusakančią struktūrą;
- **dest_len** – gavėjo adreso struktūros dydis;
- **msg** – nuoroda į struktūrą, aprašančią siunčiamus duomenis, ši struktūra leidžia perduoti „ancillary data“ ir „out-of-band“ duomenis.

```
ssize_t recv(int socket, void *message, size_t length, int flags);
```

```
ssize_t recvfrom(int socket, void *restrict message, size_t length, int
flags, struct sockaddr *restrict address, socklen_t *restrict address_len);
```

```
ssize_t recvmsg(int socket, struct msghdr *msg, int flags);
```

Funkcijos duomenų priėmimui per soketą, *recv()* su *flags=0* veikia kaip *read()*. Parametrų prasmė analogiška kaip *send*()* funkcijose.

Supaprastinti soketų API naudojančių programų pavyzdžiai:

SOCK_DGRAM serveris:

```
s = socket( ... , SOCK_DGRAM, ... );
bind( s, ... );
while( salyga ){
    recvfrom( s, ... );
    sendto( s, ... );
}
close( s );
```

SOCK_DGRAM klientas:

```
s = socket( ... , SOCK_DGRAM, ... );
sendto( s, ... );
recvfrom( s, ... );
close( s );
```

SOCK_STREAM serveris:

```
s = socket( ... , SOCK_STREAM, ... );
bind( s, ... );
listen( s, ... );
while( salyga ){
    sc = accept( s, ... );
    while( salyga2 ){
        recv( sc, ... );
        send( sc, ... );
    }
    shutdown( sc, ... );
    close( sc );
}
close( s );
```

SOCK_STREAM klientas:

```
s = socket( ... , SOCK_STREAM, ... );
connect( s, ... );
while( salyga2 ){
    send( s, ... );
    recv( s, ... );
}
shutdown( s, ... );
close( s );
```

Šiose programose:

- *SOCK_DGRAM* vietoj *sendto()* ir *recvfrom()* gali būti naudojamos *sendmsg()* ir *recvmsg()* funkcijos;
- *SOCK_STREAM* vietoj *send()* ir *recv()* gali būti naudojamos *write()* ir *read()*, o taip pat ir *sendto()* ar *sendmsg()* ir *recvfrom()* ar *recvmsg()*, tik joms nurodytas nutolusios pusės adresas bus ignoruojamas;
- *SOCK_STREAM* pavyzdžiuose siuntimo ir priėmimo funkcijos gali būti sukeistos vietos, t.y. serveris pirma siunčia, o tik paskui laukia duomenų iš kliento, o klientas atvirkščiai.

9.1.1.1 UNIX domain soketai (AF_UNIX)

PAPILDOMAI (į lab. darbo apimtį neįeina): apie atidarytų failų deskriptorių ir *credentials* perdavimą UNIX soketais žr.: apie „*ancillary data*“, „*msgcred*“, *SCM_RIGHTS* (<sys/socket.h>), *SCM_CREDS* (Solaris naudoja *SCM_UCRED*).

UNIX domain soketai skirti komunikavimui tarp procesų toje pačioje mašinoje naudojant tą patį soketų API. Jie neturi nieko bendro nei su „*loopback*“ tinklo interfeisu, nei su DNS domenu.

Veikimo principu *UNIX* soketai artimiausi SHELL konvejeriams („|“, *mkfifo*), kuriuos jau pažįstate (arba su *pipe()* sukurtiems kanalams, su kuriais dar susipažinsite). Soketo „adresas“ – failo vardas.

Prieiga prie šio failo valdoma failų sistemos priemonėmis (teisės). Prie serverio soketo gali jungti keli klientai vienu metu. Veikiantis tinklas nereikalingas, kad veiktų *UNIX* soketai. Kadangi viskas vyksta vienoje mašinoje, kernelis tiksliai žino, koks useris ir procesas kiekvienoje sujungimo pusėje, o tai leidžia išvengti papildomo autentifikavimosi (serveris ir taip gali sužinoti, kas yra kitame sujungimo gale). *UNIX* soketai leidžia perduoti atidaryto failo deskriptorių kitam procesui.

AF_UNIX serverio pavyzdys:

```
/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_un01s.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <strings.h>
#include <unistd.h>

#define BUFLLEN 10

int kp_unserver( const char *name );

int kp_unserver( const char *name ){
    int s, rv;
    struct sockaddr_un sa;
    char buf[BUFLLEN];

    s = socket( AF_UNIX, SOCK_DGRAM, 0 );
    if( s == -1 ){
        perror( "socket failed" );
        exit( 1 );
    }

    sa.sun_family = AF_UNIX;
    strncpy( sa.sun_path, name, sizeof( sa.sun_path ) - 1 );
    printf( "binding socket=%d to %s ...\n", s, sa.sun_path );
    rv = bind( s, (const struct sockaddr *)&sa, sizeof( sa ) );
    if( rv != 0 ){
        perror( "bind failed" );
        exit( 1 );
    }

    puts( "waiting for incoming data..." );
    rv = recvfrom( s, buf, sizeof(buf) - 1, 0, NULL, NULL );
    if( rv == -1 ){
        perror( "recvfrom failed" );
        abort();
    }
    buf[BUFLLEN-1] = '\0';
    buf[rv] = '\0';
    printf( "\treceived %d bytes: \"%s\"\n", rv, buf );

    puts( "closing and unlinking..." );
    close( s );
    unlink( name );

    puts( "server stopped." );
    return 0;
}
```

```

int main( int argc, char * argv[] ){
    const char *name = "kespaul_unserver.sck" ;
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    if( argc > 1 ) name = argv[1];
    kp_unserver( name );
    return 0;
}

```

Šis serveris atidaro komandinės eilutės parametru nurodytą UNIX soketą ir laukia ateinančių duomenų. Gautus duomenis atspausdina ir baigia darbą. Jei dėl kokių nors priežasčių serveris pasibaigia „nešvariai“ – failų sistemoje lieka soketo failas (pagal nutylėjimą *kespaul_unserver.sck*) ir antrą kartą serveris normaliai nebepasileidžia (reikia ištrinti soketo failą).

AF_UNIX kliento pavyzdys:

```

/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_un01c.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <strings.h>
#include <unistd.h>

#define BUFLen 10

int kp_unclient( const char *name, const char *msg );

int kp_unclient( const char *name, const char *msg ){
    int s, rv;
    struct sockaddr_un sa;

    s = socket( AF_UNIX, SOCK_DGRAM, 0 );
    if( s == -1 ){
        perror( "socket failed" );
        exit( 1 );
    }

    sa.sun_family = AF_UNIX;
    strncpy( sa.sun_path, name, sizeof( sa.sun_path ) - 1 );
    printf( "sending \"%s\" to socket=%d (%s) ...\n", msg, s, sa.sun_path );
    rv = sendto( s, msg, strlen(msg) + 1, 0, (const struct sockaddr *)&sa,
sizeof( sa ) );
    if( rv == -1 ){
        perror( "sendto failed" );
        exit( 1 );
    }

    puts( "closing..." );
    close( s );

    puts( "client stopped." );
    return 0;
}

int main( int argc, char * argv[] ){
    const char *name = "kespaul_unserver.sck", *msg = "testas";

```

```

printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
if( argc > 1 ) msg = argv[1];
if( argc > 2 ) name = argv[2];
kp_unclient( name, msg );
return 0;
}

```

Klientui komandinės eilutės parametrais nurodomas siunčiamas pranešimas (pagal nutylėjimą „testas“) ir soketo failas (pagal nutylėjimą „kespaul_unserver.sck“) į kur siųsti. Pasiuntęs pranešimą klientas baigia darbą.

9.1.1.2 IPv4 soketai (AF_INET)

Pagal OSI modelį *IPv4* soketas – tai transporto sluoksnio objektas (*Layer 4, Transport Layer*). Kitaip sakant – tai abstrakcija paslepianti žemesnius sluoksnius (maršrutizavimą, paketų pametimą, pamestų kartojimą, paketų rikiavimą ir kt.), bet leidžianti naudotis srauto valdymo (*flow control*) priemonėmis (buferių dydžiai, timeout'ai, lango dydis ir t.t.). Naudojantis soketais galima realizuoti aukštesnių OSI lygių protokolus (HTTP, SSH, SSL/TLS ...).

AF_INET kliento pavyzdys:

```

/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_ipv4_01c.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>

#define BUFLLEN 32768

int kp_ipv4_client( const char *host, int port, const char *msg );

int kp_ipv4_client( const char *host, int port, const char *msg ){
    int s, rv, n;
    struct sockaddr_in sa;
    char buf[BUFLLEN];
    s = socket( AF_INET, SOCK_STREAM, 0 );
    if( s == -1 ){
        perror( "socket failed" );
        exit( 1 );
    }

    memset( &sa, 0, sizeof( sa ) );
    sa.sin_family = AF_INET;
    sa.sin_port = htons( port );
    rv = inet_pton( AF_INET, host, &sa.sin_addr );
    if( rv != 1 ){
        if( rv == -1 ){
            perror( "inet_pton error" );
            exit( 1 );
        }
        puts( "inet_pton wrong address" );
        exit( 1 );
    }
}

```

```

    rv = connect( s, (struct sockaddr *)&sa, sizeof( sa ) );
    if( rv != 0 ){
        perror( "connect failed" );
        exit( 1 );
    }

    n = strlen( msg );
    /* rv = write( s, msg, n );*/
    rv = send( s, msg, n, 0 );
    if( rv != n ){
        if( rv == -1 ){
            perror( "send failed" );
            exit( 1 );
        }
        puts( "send error" );
        exit( 1 );
    }

    /* rv = read ( s, buf, sizeof(buf) - 1 );*/
    rv = recv ( s, buf, sizeof(buf) - 1, 0 );
    if( rv == -1 ){
        perror( "recv error" );
        exit( 1 );
    }
    buf[rv] = '\0';
    buf[BUFLen - 1] = '\0';
    puts( buf );

    shutdown( s, SHUT_RDWR );
    close( s );
    return 0;
}

int main( int argc, char * argv[] ){
    int p;
    const char *host = "127.0.0.1";
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    if( argc > 1 ) host = argv[1];
    p = 80;
    kp_ipv4_client( host, p, "GET / HTTP/1.0\r\n\r\n" );
    return 0;
}

```

Šis pavyzdys sukuria IPv4 TCP soketą ir bando pabendrauti su komandinės eilutės argumentu nurodytu IP adresu HTTP protokolu. Kitaip sakant čia supaprastintas WWW kliento pavyzdys, t. y. programa siunčianti dvi eilutes (tik eilučių pabaigos žymimos `\r\n`, o ne `\n` – smulkiau žr. HTTP protokolo aprašymą RFC 1945 <https://www.ietf.org/rfc/rfc1945>):

```
GET / HTTP/1.0
```

nurodytam WWW serveriui ir nuskaitanti atsakymą.

Vietoj `send()/recv()` galima naudoti užkomentuotas `write()/read()` funkcijas.

9.1.1.3 IPv6 soketai (AF_INET6)

AF_INET6 kliento pavyzdys (patch'as):

```

--- kespaul_ipv4_01c.c  Tue Mar 26 14:06:12 2013
+++ kespaul_ipv6_01c.c  Wed Mar 27 11:33:58 2013
@@ -1,5 +1,5 @@
  /* Kestutis Paulikas KTK kespaul */
  /* Failas: kespaul_ipv4_01c.c */
  /* Failas: kespaul_ipv6_01c.c */

  #include <stdio.h>
  #include <stdlib.h>
@@ -10,13 +10,13 @@

  #define BUFLLEN 32768

  -int kp_ipv4_client( const char *host, int port, const char *msg );
  +int kp_ipv6_client( const char *host, int port, const char *msg );

  -int kp_ipv4_client( const char *host, int port, const char *msg ){
  +int kp_ipv6_client( const char *host, int port, const char *msg ){
      int s, rv, n;
      - struct sockaddr_in sa;
      + struct sockaddr_in6 sa;
      char buf[BUFLLEN];
      - s = socket( AF_INET, SOCK_STREAM, 0 );
      + s = socket( AF_INET6, SOCK_STREAM, 0 );
      if( s == -1 ){
          perror( "socket failed" );
          exit( 1 );
@@ -23,9 +23,9 @@
      }

      memset( &sa, 0, sizeof( sa ) );
      - sa.sin_family = AF_INET;
      - sa.sin_port = htons( port );
      - rv = inet_pton( AF_INET, host, &sa.sin_addr );
      + sa.sin6_family = AF_INET6;
      + sa.sin6_port = htons( port );
      + rv = inet_pton( AF_INET6, host, &sa.sin6_addr );
      if( rv != 1 ){
          if( rv == -1 ){
              perror( "inet_pton error" );
@@ -70,10 +70,10 @@

      int main( int argc, char * argv[] ){
          int p;
          - const char *host = "127.0.0.1";
          + const char *host = "::1";
          printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
          if( argc > 1 ) host = argv[1];
          p = 80;
          - kp_ipv4_client( host, p, "GET / HTTP/1.0\r\n\r\n" );
          + kp_ipv6_client( host, p, "GET / HTTP/1.0\r\n\r\n" );
          return 0;
      }
  
```

Tai tokia pat programa, kaip ir ankstesniame skyriuje, tik dirbanti ne IPv4, o IPv6 protokolu. Čia pateikti tik šių dviejų programų skirtumai. Skirtumai gauti *diff* komanda:

```
diff -u kespaul_ipv4_01c.c kespaul_ipv6_01c.c
```

Jei nenorite rankomis redaguoti failo – galite šiuos pataisymus užsaugoti į failą ir jį „pritaikyti“

(*apply*) originalo kopijai, t. y. (čia *failas.patch* – failas, kuriame yra patch'o tekstas):

```
cp kespaul_ipv4_01c.c kespaul_ipv6_01c.c
patch -p0 < failas.patch
```

9.2 Darbas

9.2.1 Pagalbinės tinklo funkcijos

POSIX standartas numato ir keletą funkcijų konvertavimui tarp hostų vardų ir IPv4/6 adresų (numerų). Žemiau pateiktas pavyzdys *getaddrinfo()*, *getnameinfo()*, *gai_strerror()*, *freeaddrinfo()* funkcijas.

Funkcija **getaddrinfo()** pagal nustatymus, nurodytus *hints* struktūroje (**struct addrinfo**) bando išrišti vardus į adresus (vienas vardas gali išsirišti į daugiau nei vieną adresą) ir grąžina vienkryptį sąrašą (pirmo sąrašo elemento adresas įrašomas į *result* pointerį). Grąžinto sąrašo elementai surišti **ai_next** laukų pagalba (t. y. *result*→*ai_next* rodo į sekantį sąrašo elementą).

Įvykus klaidai (pvz.: jei vardas neužregistruotas) – klaidos priežastį galima sužinot su funkcija **gai_strerror()**, kuri grąžina nuorodą į klaidos aprašymą (*char**).

Funkcija **getnameinfo()** pagal jai nurodytus parametrus gali arba išrišinėti adresą į vardą, arba versti adresą į simbolių eilutę (pvz.: atvaizdavimui ekrane).

Kadangi *getaddrinfo()* grąžina sąrašą, šio sąrašo sunaikinimas kai jis nebereikalingas – programuotojo atsakomybė. Tai atlieka **freeaddrinfo()** funkcija.

```
/* Kestutis Paulikas KTK kespaul */
/* Failas: kespaul_resolv01.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAXL      255    /* pagal getconf _POSIX_HOST_NAME_MAX arba
HOST_NAME_MAX */

int kp_resolve( const char *name );

int kp_resolve( const char *name ){
    struct addrinfo hints, *result, *rp;
    int rv;
    char buf[MAXL];
    printf( "resolving: %s\n\n", name );
    memset( &hints, 0, sizeof( hints ) );
    hints.ai_family = AF_UNSPEC;
    hints.ai_flags = AI_ALL | AI_V4MAPPED;
    rv = getaddrinfo( name, NULL, &hints, &result );
    if( rv != 0 ){
        printf( "getaddrinfo error: %s\n", gai_strerror( rv ) );
        exit( 1 );
    }
}
```



```

    for( rp = result; rp != NULL; rp = rp->ai_next ){
        rv = getnameinfo( rp->ai_addr, rp->ai_addrlen, buf, MAXL, NULL, NULL,
NI_NUMERICHOST );
        if( rv == 0 ){
            printf( "resolved IP: %s\n", buf );
        }
        rv = getnameinfo( rp->ai_addr, rp->ai_addrlen, buf, MAXL, NULL, NULL,
0 );
        if( rv == 0 ){
            printf( "resolved FQDN: %s\n", buf );
        }
        puts( "" );
    }

    freeaddrinfo( result );
    return 0;
}

int main( int argc, char * argv[] ){
    const char *name = "localhost";
    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
    if( argc > 1 ) name = argv[1];
    kp_resolve( name );
    return 0;
}

```

Šis pavyzdys bando nurodytą komandos parametru vardą išrišti į jį atitinkančius IPv4 ir IPv6 adresus, atspausdina kiekvieną rezultatą ir bando kiekvieną rezultatą išrišti atgal į DNS vardą.

Išbandykite pavyzdį be parametru, su www.ktu.lt, www.google.lt, www.cisco.com. Modifikuokite pavyzdį (sukurkite **loginas_resolv02.c**), kad būtų būtų išrišami tik IPv4 adresai (programa nebandytu išrišti IPv6 adresų, o gautus IPv4 adresus spausdintų įprastam, o ne IPv6mapped formate). Įkelkite sukurta programą į Moodle.

9.2.2 UNIX domain soketai

Sukurkite **AF_UNIX SOCK_STREAM** (ne **SOCK_DGRAM**) soketus naudojantį serverį **loginas_un02s.c** ir klientą **loginas_un02c.c**.

- Turi būti numatytas normalus serverio darbo užbaigimas (pvz.: pabendravus su vienu ar N klientų, gavus specifinį pranešimą ir pan.)
- vienos sesijos metu perduoti „daug“ duomenų (>1MB) bent viena kryptimi.

Programas įkelkite į Moodle.

Kiek maksimaliai išeina perduoti domenų vienu paketu su **SOCK_DGRAM** (eksperimentams galite naudoti *kspaul_un01*.c*, bet pasikeiskite buferio dydį): ???

Kas nutinka su **SOCK_DGRAM** paketu perduotais duomenimis, jei priimanti pusė su *recvfrom()* nuskaityto nevisą paketą (pametama, nuskaityta sekančiu kreipiniu į *recvfrom()* ir t.t.): ???

9.2.3 IPv4 soketai

Sukurkite **AF_INET SOCK_STREAM** serverio programą **loginas_ipv4_02s.c**, su kuria galėtų susišnekėti pagal *kspaul_ipv4_01c.c* pavyzdį sukurta klientas (modifikuokite pavyzdį ir sukurkite **loginas_ipv4_02c.c**). Programų tekstus įkelkite į Moodle.

- serverį galėsite paleisti tik ant nenaudojamų portų;
- norint atidaryti portus, kurių numeriai mažesni už 1024 reikalingos root userio teisės.

9.2.4 IPv6 soketai

Modifikuokite `loginas_ipv4_02s.c` ir `loginas_ipv4_02c.c` (sukurkite **`loginas_ipv6_02s.c`** ir **`loginas_ipv6_02c.c`**), kad jos dirbtų IPv6 protokolu. Programų tekstus įkelkite į Moodle. Išbandykite su kuriais *os.if.ktu.lt* mašinos IPv6 adresais veikia šios programos: ???

9.3 Atsiskaitymas

Atsakyti į Moodle klausimus.

Įkelkite savo sukurtas programas į Moodle:

1. `loginas_resolv02.c`
2. `loginas_un02s.c`
3. `loginas_un02c.c`
4. `loginas_ipv4_02s.c`
5. `loginas_ipv4_02c.c`
6. `loginas_ipv6_02s.c`
7. `loginas_ipv6_02c.c`

9.4 Vertinimas

Už ne savo sukurtų failų įkėlimą visas užsiėmimas vertinamas 0%.

- 25% už tai kad atsakyta į Moodle klausimus užsiėmimo metu;
- 25% už tai kad įkelti failai
- $\leq 25\%$ už atsakymų turinį
- $\leq 25\%$ už įkeltų programų turinį
 - failai, netenkinatys šablono reikalavimų vertinami 0%
 - + ~15% už stilių (išlaikytas vienodas programavimo stilius, kodas lengvai suprantamas)
 - + ~5% jei programa daro tai, ką turi daryt
 - + ~5% jei programa „atspari kvailiams“ (už tinkamą klaidų apdorojimą)