Turinys

L3 OS KONSPEKTAS

L3_1

ELF analizė

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

```
$ ldd /bin/true

$ ldd /lib/libutil.so.1
```

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

```
$ objdump -f -h -p /bin/true

$ objdump -s -j.rodata /bin/true
```

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

```
$ readelf -h /bin/true

$ readelf -x .got /bin/true
```

nm – parodo ELF failo simbolių lentelę. Naudojimo pavyzdys:

```
$ nm -D /bin/true

$ nm -Du /bin/true
```

Proceso analizės įrankiai

Papildoma medžiaga:
MAN: strace(1), pldd(1), pstack(1), pfiles(1), lsof(8)

`*strace* (Unix: *truss*; BSD: *kdump/ktrace*)` – parodo proceso kviečiamus *syscall*'us ir jų parametrus. Naudojimo pavyzdys:

```
$ strace /bin/true
```

pldd, `(Solaris: *pstack*, *pfiles*)` – pagalbinės 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)*:

lsof - pagalbinė (nestandartinė) programa, parodanti proceso atidarytus failus, soketus.

```
$ pstack 22591
```

L3 OS KONSPEKTAS

gdb

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 /data/ld/ld3/hello1
```

Kompiliavimas:

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

C programos failo pavyzdys `hello2.c` (source:posix|hello2.c):

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

C programos kompiliavimas (su GCC kompiliatoriumi):

```
gcc -o hello2 hello2.c
```

gcc

`gcc` - vienas iš *GNU Compiler Collection* įrankių, skirtas C programų kompiliavimui.

Parinktys (ne visos):

- `--help` – pagalba (išveda kai kurių parinkčių aprašymus).
- `-o failas` – rezultatų failo vardas.
- `-c` – baigti darbą sukompiliavus failą, t.y. sustoti po kompiliavimo žingsnio, ryšių redaktorius nekviečiamas, sukuriamas objektinis failas.
- `-g` – į objektinį ir vykdomąjį failus įtraukia debugeriui skirtą informaciją (be jos gdb nerodys programos teksto). Yra daugiau panašių parinkčių konkrečiai nurodančių debugerio informacijos formatą (pvz.: `-gstabs`, `-gstabs+`).
- `-E` – baigti darbą po preprocesoriaus (gaunamas failas be preprocesoriaus direktyvų), rezultatas išvedamas į ekraną, ne į failą (smulkiau apie formatą).
- `-S` – baigti darbą prieš asemblerio paleidimą (gaunamas asemblerio failas).
- `-Wall` – rodyti visus perspėjimus.
- `-Werror` – perspėjimus laikyti klaidomis (aptikus klaidų rezultatas nekuriamas)
- `-pedantic` – griežtesnis sintaksės tikrinimas.
- `-###` – parodo žingsnius, kuriuos kompiliatoriaus vykdytų, bet jų nevykdo.
- `-pg` – į objektinį ir vykdomąjį failus įtraukia profaileriui skirtą informaciją.
- `-O0` – išjungiama visa optimizacija.
- `-O2` – įjungiamas vidutinis optimizacijos vykdymo greičiui lygis.
- `-Os` – įjungiama generuojamo kodo dydžio optimizacija.
- `-M` – nekompiliuoja, tik parodo nurodyto failo priklausomybes (`make` formatu).

- `-MM` – tas pats, kaip `-M`, tik nerodomos priklausomybės nuo sistemos (priklausančių OS) failų.
- `-m32` – kuriami 32bit failai (asemblerio, objektinis, vykdomasis, ir t.t.).
- `-m64` – kuriami 64bit failai.
- `-D XXX` – nustatyti makroso `XXX` reikšmę (tas pats, kaip `#define XXX ...` programoje).

Naudojimo pavyzdys (jei failas `tekstas.c` be klaidų – bus sukurtas `vykdomasis_failas`):

```
$ gcc -Wall -Werror -pedantic -o vykdomasis_failas -g tekstas.c
```

Make

Programuojant C ar kita kompiliuojama kalba pastoviai prireikia pergeneruoti 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) pergeneruoja failus.

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:

```
hello1: hello1.c
```

```
    gcc -c -g hello1.c

    gcc -o hello1 hello1.o
```

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).

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

```
SRC=$(wildcard *.c)

ALL=$(SRC:.c=)

CC=gcc

CFLAGS=-g -Wall -Werror -pedantic



all: ${ALL}
```

L3_1 Užduotys

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 `$HOME/lab3/uzs1/work/` kataloge, kad matytumėt, kas keičiasi – prieš kiekvieną bandymą ištrinkite visus failus iš `$HOME/lab3/uzs1/work/` katalogo):

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

lukkuz1@oslinux ~/lab3/uzs1/work $ gcc ../src/hello1.c

lukkuz1@oslinux ~/lab3/uzs1/work $ ls -l

total 16

-rwxr-xr-x 1 lukkuz1 users 15848 Apr 20 13:12 a.out

lukkuz1@oslinux ~/lab3/uzs1/work $ gcc ../src/hello2.c

lukkuz1@oslinux ~/lab3/uzs1/work $ gcc ../src/hello3.c

lukkuz1@oslinux ~/lab3/uzs1/work $ ls -l

total 16

-rwxr-xr-x 1 lukkuz1 users 16016 Apr 20 13:12 a.out

lukkuz1@oslinux ~/lab3/uzs1/work $

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

jau sukompiliuoja failą -o naudojamas sukurti bin compilerio failui.

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

Nukopijuokite `hello1.c` į `hello1a.c` ir kopijos 2'oje eilutėje (`return 0`) pakeiskite 0 į 1. Sukompiliuokite ir paleiskite abi programas. Kaip SHELL'e pamatyti, ką gražino `main` funkcija?

Pamatyti ką gražino main funkciją galime su $?

Išbandykite:

```
$ gcc -E ../src/hello1.c

$ gcc -E ../src/hello2.c
```

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

Pažiūrėkite, kaip kompiliatorius kuria vykdomąjį failą:

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

Išvestą rezultatą nusikopijuokite ir pažymėkite (paryškinkite) tas vietas, kuriose nurodomas kelias iki kompiliatoriaus, asemblerio, ryšių redaktoriaus (linkerio). Pakartokite atskirus kompiliatoriaus žingsnius juos vykdydami iš komandinės eilutės.

Naudodamiesi, ankstesniuose punktuose sužinota informacija, parašykite ir išbandykite komandas sukuriančias tarpinius failus:

- `hello2.i` (C tekstas be prerprocesoriaus direktyvų, t.y. `#include` pakeisti failų turiniais)

  gcc -E hello2.c -o hello2.i

- `hello2.s` (asemblerio tekstas)

  gcc -S hello2.i -o hello2.s

- `hello2.o` (objektinis ELF failas)

  gcc -c hello2.s -o hello2.o

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

  ld hello2.o -o hello2a

ELF užduotys

Naudodamiesi teorinėje dalyje aprašytais įrankiais:

- suraskite, kokias bibliotekas naudoja `hello2`:

Using built-in specs.

COLLECT_GCC=gcc

COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-pc-linux-gnu/10.2.0/lto-wrapper

Target: x86_64-pc-linux-gnu

Configured with: /var/tmp/portage/sys-devel/gcc-10.2.0-r5/work/gcc-10.2.0/configure --host=x86_64-pc-linux-gnu --build=x86_64-pc-linux-gnu --prefix=/usr --bindir=/usr/x86_64-pc-linux-gnu/gcc-bin/10.2.0 --includedir=/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/include --datadir=/usr/share/gcc-data/x86_64-pc-linux-gnu/10.2.0 --mandir=/usr/share/gcc-data/x86_64-pc-linux-gnu/10.2.0/man --infodir=/usr/share/gcc-data/x86_64-pc-linux-gnu/10.2.0/info --with-gxx-include-dir=/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/include/g++-v10 --with-python-dir=/share/gcc-data/x86_64-pc-linux-gnu/10.2.0/python --enable-languages=c,c++ --enable-obsolete --enable-secureplt --disable-werror --with-system-zlib --enable-nls --without-included-gettext --enable-checking=release --with-bugurl=https://bugs.gentoo.org/ --with-pkgversion='Gentoo Hardened 10.2.0-r5 p6' --enable-esp --enable-libstdcxx-time --disable-libstdcxx-pch --enable-shared --enable-threads=posix --enable-__cxa_atexit --enable-clocale=gnu --enable-multilib --with-multilib-list=m32,m64 --disable-fixed-point --enable-targets=all --enable-libgomp --disable-libssp --disable-libada --disable-systemtap --disable-vtable-verify --disable-libvtv --without-zstd --enable-lto --without-isl --enable-default-pie --enable-default-ssp

Thread model: posix

Supported LTO compression algorithms: zlib

gcc version 10.2.0 (Gentoo Hardened 10.2.0-r5 p6)

COMPILER_PATH=/usr/libexec/gcc/x86_64-pc-linux-gnu/10.2.0/:/usr/libexec/gcc/x86_64-pc-linux-gnu/10.2.0/:/usr/libexec/gcc/x86_64-pc-linux-gnu/:/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/:/usr/lib/gcc/x86_64-pc-linux-gnu/:/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../x86_64-pc-linux-gnu/bin/

LIBRARY_PATH=/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/:/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../lib64/:/lib/../lib64/:/usr/lib/../lib64/:/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../x86_64-pc-linux-gnu/lib/:/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../:/lib/:/usr/lib/

COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'

/usr/libexec/gcc/x86_64-pc-linux-gnu/10.2.0/collect2 -plugin /usr/libexec/gcc/x86_64-pc-linux-gnu/10.2.0/liblto_plugin.so -plugin-opt=/usr/libexec/gcc/x86_64-pc-linux-gnu/10.2.0/lto-wrapper -plugin-opt=-fresolution=/tmp/ccW7AqOx.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --eh-frame-hdr -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now /usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../lib64/Scrt1.o /usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../lib64/crti.o /usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/crtbeginS.o -L/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0 -L/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../lib64 -L/lib/../lib64 -L/usr/lib/../lib64 -L/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../x86_64-pc-linux-gnu/lib -L/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../.. hello2 -lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-needed -lgcc_s --pop-state /usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/crtendS.o /usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../lib64/crtn.o

/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../x86_64-pc-linux-gnu/bin/ld: /usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0/../../../../lib64/Scrt1.o: in function `_start':

(.text+0x20): undefined reference to `main'

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`: ?

L3 OS KONSPEKTAS

main: ?
hello2 vykdymo pradžia (entry point): ?
gdb sesija patikrinimui

lukkuz1@oslinux ~/lab3/uzs1/work $ objdump -t hello2 | grep -E '(main|_start)'

0000000000003df0 l .init_array 0000000000000000 __init_array_start

0000000000004000 w .data 0000000000000000 data_start

0000000000000000 F *UND* 0000000000000000 __libc_start_main@GLIBC_2.2.5

0000000000004000 g .data 0000000000000000 __data_start

0000000000000000 w *UND* 0000000000000000 __gmon_start__

0000000000001040 g F .text 000000000000002b _start

0000000000004010 g .bss 0000000000000000 __bss_start

0000000000001125 g F .text 000000000000000b main

- 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 reikalinga `gdb --tty` parinktis.

gdb --tty=/dev/tty2 hello3

sesija

- Modifikuokite bet kurį iš duotų C failų, kad jis lūžtų (pvz.: įdėkite `abort()` funkcijos iškvietimą, dalybą iš 0 ar bandymą skaityt 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 `GDB TUI`.

```
#include <stdio.h>

#include <stdlib.h>


int main() {

  printf("Hello, World!\n");

  abort();

  return 0;
```

}

gcc -g hello1.c -o hello1

Advanced

gprof užduotys

Sukompiliuokite `hello3.c` (iš C compiler-tasks) su profailerio informacija. Paleiskite, išanalizuokite gautą `gmon.out` failą.

lukkuz1@oslinux ~/lab3/uzs1/src $ gcc -pg hello3.c

lukkuz1@oslinux ~/lab3/uzs1/src $ ls -l

total 48

-rwxr-xr-x 1 lukkuz1 users 16264 Apr 28 16:27 a.out

-rw-r--r-- 1 lukkuz1 users   370 Apr 28 16:27 gmon.out

-rw-r--r-- 1 lukkuz1 users   0 Apr 27 11:37 hello1a.c

-rw-r--r-- 1 lukkuz1 users   33 Apr 27 11:25 hello1.c

-rw-r--r-- 1 lukkuz1 users   84 Apr 27 11:26 hello2.c

-rwxr-xr-x 1 lukkuz1 users 16264 Apr 28 16:27 hello3

-rw-r--r-- 1 lukkuz1 users   294 Apr 27 11:27 hello3.c

lukkuz1@oslinux ~/lab3/uzs1/src $ gprof hello3 gmon.out

Flat profile:

Each sample counts as 0.01 seconds.

no time accumulated

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	10	0.00	0.00	hellofunc

% the percentage of the total running time of the

time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.

total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
function is profiled, else blank.

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in
the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) no time propagated

index % time self children called name

0.00 0.00 10/10 main [5]

[1] 0.0 0.00 0.00 10 hellofunc [1]

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

index A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so it is easier to look up where the function is in the table.

% time This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc,

these numbers will NOT add up to 100%.

self **This is the total amount of time spent in this function.**

children **This is the total amount of time propagated into this function by its children.**

called **This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.**

name **The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.**

For the function's parents, the fields have the following meanings:

self **This is the amount of time that was propagated directly from the function into this parent.**

children **This is the amount of time that was propagated from the function's children into this parent.**

called **This is the number of times this parent called the function `/' the total number of times the function was called. Recursive calls to the function are not**

included in the number after the `/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

[1] hellofunc

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

Dalį funkcijų profaileris rodo, kaip nenaudojamas. Patikrinkite (su debugeriu), ar tikrai jos neiškviečiamos programos vykdymo metu? Jei iškviečiamos – papildykite iškvietimų grafą. Išsisaugokite sesijos logą, kaip tai patikrinote.

L3_2

POSIX C funkcijos darbui su failų sistema¶

Kad galėtų atlikti kokius nors veiksmus su failu, katalogu, kanalu ar soketu – procesas turi pirmiausia tą objektą **atidaryti**, o baigęs darbą **uždaryti** (jei procesas pasibaigs neuždaręs visų savo objektų – OS nesugrius, bet kai kuriais atvejais duomenys gali likt neįrašyti). 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.

POSIX API: failų sistema

Katalogo atidarymas ir uždarymas

kespaul_open01.c (source:posix|kespaul_open01.c)

```c
/* 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 );
```

```
17    if( dskr == -1 ){
18       perror( name );
19       exit(1);
20    }
21    printf( "dskr = %d\n", dskr );
22    return dskr;
23 }
24
25 int kp_test_close(int fd){
26    int rv;
27    rv = close( fd );
28    if( rv != 0 ) perror ( "close() failed" );
29    else puts( "closed" );
30    return rv;
31 }
32
33 int main( int argc, char *argv[] ){
34    int d;
35    if( argc != 2 ){
36       printf( "Naudojimas:\n %s failas_ar_katalogas\n", argv[0] );
37       exit( 255 );
38    }
39    d = kp_test_open( argv[1] );
40    kp_test_close( d );
41    kp_test_close( d ); /* turi mesti close klaida */
42    return 0;
43 }
```

Darbinis katalogas ir jo keitimas

```
/*
Kęstutis
Paulikas
KTK
kespaul
*/
2  /* Failas: kespaul_getcwd01.c */
3
4  #include <stdio.h>
```

```
5  #include <stdlib.h>
6  #include <unistd.h>
7
8  int kp_test_getcwd();
9
10 int kp_test_getcwd(){
11    char *cwd;
12    cwd = getcwd( NULL, pathconf( ".", _PC_PATH_MAX) );
13    puts( cwd );
14    free( cwd );
15    return 1;
16 }
17
18 int main(){
19    kp_test_getcwd();
20    return 0;
21 }
```

Katalogų struktūros „apvaikščiojimas“

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);
```

Argumentai:

path – kelias nuo kur pradedamas darbas
fn – funkcijos vardas, kuri iškviečiama kiekvienam surastam failui ar katalogui apdoroti
fd_limit – maksimalus deskriptorių skaičius, kurį gali naudoti nftw() (veikia greičiau, jei jis ne mažesnis už maksimalų failų gylį, tačiau tai nebūtina – gali dirbti ir su mažiau deskriptorių)
flags – vėliavėlės:

- `FTW_CHDIR` – keisti darbinį katalogą (`chdir()`) į kiekvieną apdorojamą katalogą
- `FTW_DEPTH` – pirmiausia eiti gilyn (pirma apdoroti katalogo turinį, o tik po to patį katalogą)
- `FTW_MOUNT` – neitį į kitas failų sistemas (dirbti tik vienoje failų sistemoje)
- `FTW_PHYS` – neiti per simbolines nuorodas (symlinkus).

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

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

Argumentai:

fpath – surasto failo ar katalogo vardas
sb – nuoroda į struktūrą su informacija apie failą (žr.: `stat()` funkciją)
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

ftwbuf – nuoroda į struktūrą su 2 laukais:

1. **base** – poslinkis iki failo vardo fpath parametru perduotoje eilutėje, t. y. `nuroda_i_failo_vardą = fpath + ftwbuf->base`
2. **level** – kiek giliai failas ar katalogas, skaičiuojant nuo paieškos pradžios (paieškos pradžia – 0).

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

NFTW pavyzdys

```
#include <stdio.h>
#include <ftw.h>

struct FTW {
    int base;
    int level;
};

int print_file(const char* path, const struct stat* stat_buf, int typeflag, struct FTW* ftw_buf) {
    if (typeflag == FTW_D) { // if it's a file
        printf("%s\n", path); // print the file name
    }
    return 0; // return 0 to continue the traversal
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s directory_path\n", argv[0]);
        return 1;
    }
```

```
    int flags = FTW_NS; // use physical walk instead of following symbolic links
    int max_depth = 10; // limit traversal to a depth of 10 directories
    int result = nftw(argv[1], print_file, max_depth, flags);

    if (result == -1) { // check for errors
        perror("nftw");
        return 1;
    }

    return 0;
}
```

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 objektui.

Link()

int link(const char *existing_file_path, const char *new_file_path);

existing_file_path – jau egzistuojantis failas

new_file_path – naujas failas, tarp kurio sukūrti link ryšį su egzistuojančiu failu.

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source_file> <target_file>\n", argv[0]);
        exit(EXIT_FAILURE);
```

```
    }

    if (link(argv[1], argv[2]) == -1) {
        perror("link");
        exit(EXIT_FAILURE);
    }

    printf("Hard link created between %s and %s\n", argv[1], argv[2]);

    return 0;
}
```

Unlink()

int unlink(const char *pathname);

pathname – failas, kurį noriname unlink()

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    if (unlink(argv[1]) != 0) {
        perror("unlink");
        return 1;
```

```
    }


    printf("File '%s' has been unlinked.\n", argv[1]);

    return 0;

}
```

Symlink()

int symlink(const char *target_path, const char *link_path);

target path – failas, iš kurio bus linkas.

Link_path – naujas failas, į kurį rodys target_path.

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <target> <linkname>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const char *target = argv[1];
    const char *linkname = argv[2];

    int ret = symlink(target, linkname);
    if (ret == -1) {
        perror("symlink");
        exit(EXIT_FAILURE);
    }

    printf("Created symbolic link '%s' pointing to '%s'\n", linkname,
target);

    exit(EXIT_SUCCESS);
}
```

Remove()

int remove(const char *filename);

filename – failas, kurį norime ištrinti.

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        exit(1);
    }
    if (remove(argv[1]) == -1) {
        printf("Error deleting file %s\n", argv[1]);
        exit(1);
    }
    printf("File %s deleted successfully.\n", argv[1]);
    return 0;
}
```

Rename()

int rename(const char *old_filename, const char *new_filename);

old_filename – senas failas

new_filename – naujas failo pavadinimas

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <old_filename> <new_filename>\n",
argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    const char *old_filename = argv[1];
    const char *new_filename = argv[2];

    if (rename(old_filename, new_filename) == -1) {
        perror("rename");
        exit(EXIT_FAILURE);
    }

    printf("File renamed successfully!\n");

    return 0;
}
```

Mkdir()

int mkdir(const char* path, mode_t mode);

path-direktorijos pavadinimas

t mode – leidimai

- **S_IRWXU** (0700): Read, write, and execute by owner
- **S_IRUSR** (0400): Read by owner
- **S_IWUSR** (0200): Write by owner
- **S_IXUSR** (0100): Execute (search) by owner
- **S_IRWXG** (0070): Read, write, and execute by group
- **S_IRGRP** (0040): Read by group
- **S_IWGRP** (0020): Write by group
- **S_IXGRP** (0010): Execute (search) by group
- **S_IRWXO** (0007): Read, write, and execute by others
- **S_IROTH** (0004): Read by others
- **S_IWOTH** (0002): Write by others
- **S_IXOTH** (0001): Execute (search) by others

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <directory>\n", argv[0]);
        return 1;
    }
```

```
    if (mkdir(argv[1], 0200) == -1) {
        perror("mkdir");
        return 1;
    }

    printf("Directory '%s' created successfully.\n", argv[1]);

    return 0;
}
```

Rmdir()

int rmdir(const char *pathname);

pathname – direktorijos pavadinimas

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <dir>\n", argv[0]);
        return 1;
    }

    if (rmdir(argv[1]) == -1) {
        perror("rmdir");
        return 1;
    }

    printf("Directory '%s' successfully removed.\n", argv[1]);

    return 0;
```

```
}
```

Creat()

int creat(const char *pathname, mode_t mode);

pathname – failo pavadinimas

t mode -leidimai

- **S_IRWXU** (0700): Read, write, and execute by owner
- **S_IRUSR** (0400): Read by owner
- **S_IWUSR** (0200): Write by owner
- **S_IXUSR** (0100): Execute (search) by owner
- **S_IRWXG** (0070): Read, write, and execute by group
- **S_IRGRP** (0040): Read by group
- **S_IWGRP** (0020): Write by group
- **S_IXGRP** (0010): Execute (search) by group
- **S_IRWXO** (0007): Read, write, and execute by others
- **S_IROTH** (0004): Read by others
- **S_IWOTH** (0002): Write by others
- **S_IXOTH** (0001): Execute (search) by others

```
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>


int main(int argc, char *argv[]) {

    if (argc < 2) {

        printf("Usage: %s filename\n", argv[0]);

        return 1;

    }


    int fd = creat(argv[1], 0644);

    if (fd == -1) {

        perror("Error creating file");

        return 1;
```

```
    }

    printf("File '%s' created with descriptor %d\n", argv[1], fd);

    close(fd);

    return 0;
}
```

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

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 00777 (rwxrwxrwx), tačiau nustatyta maskė 00022, atliekama 00777 AND NOT 00022 = 00777 AND 07755 = 00755 (tas pats bitais: 000111111111 AND (NOT 000000010010) = 000111111111 AND 111111101101 = 000111101101) – gauname failą su `rwxr-xr-x` teisėmis.
Funkcijos `chmod()` ir `fchmod()` skirtos pakeisti failo teisėms nepriklausomai nuo `umask()` nustatytos maskės.

Umask()

mode_t umask(mode_t mask);

t mask – leidimai

- **`S_IRWXU`** (0700): Read, write, and execute by owner
- **`S_IRUSR`** (0400): Read by owner
- **`S_IWUSR`** (0200): Write by owner
- **`S_IXUSR`** (0100): Execute (search) by owner
- **`S_IRWXG`** (0070): Read, write, and execute by group
- **`S_IRGRP`** (0040): Read by group
- **`S_IWGRP`** (0020): Write by group

- `S_IXGRP` (0010): Execute (search) by group
- `S_IRWXO` (0007): Read, write, and execute by others
- `S_IROTH` (0004): Read by others
- `S_IWOTH` (0002): Write by others
- `S_IXOTH` (0001): Execute (search) by others

```
#include <sys/stat.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>



int main() {

    mode_t old_mask = umask(0011); // set umask to clear write permission for group and others

    int fd = creat("example.txt", 0644); // create new file with permission 644

    umask(old_mask); // reset umask to previous value

    return 0;

}
```

Chmod()

int chmod(const char *pathname, mode_t mode);

pathname – failas

t mode – leidimai

- `S_IRWXU` (0700): Read, write, and execute by owner
- `S_IRUSR` (0400): Read by owner
- `S_IWUSR` (0200): Write by owner
- `S_IXUSR` (0100): Execute (search) by owner
- `S_IRWXG` (0070): Read, write, and execute by group
- `S_IRGRP` (0040): Read by group
- `S_IWGRP` (0020): Write by group
- `S_IXGRP` (0010): Execute (search) by group
- `S_IRWXO` (0007): Read, write, and execute by others
- `S_IROTH` (0004): Read by others
- `S_IWOTH` (0002): Write by others
- `S_IXOTH` (0001): Execute (search) by others

```
#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>


int main(int argc, char *argv[]) {

    if (argc != 3) {

        fprintf(stderr, "Usage: %s <path> <mode>\n", argv[0]);

        return 1;

    }


    const char* path = argv[1];

    mode_t mode = atoi(argv[2]);


    if (chmod(path, mode) == -1) {

        perror("chmod");

        return 1;

    }


    printf("Successfully changed mode of %s to %o\n", path, mode);

    return 0;

}
```

Fchmod()

int chmod(const char *pathname, mode_t mode);

```
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>
```

```
#include <sys/stat.h>

int main(int argc, char *argv[]) {
    int fd, mode;
    mode_t old_mode;

    if (argc < 3) {
        printf("Usage: %s <file> <mode>\n", argv[0]);
        return 1;
    }

    fd = open(argv[1], O_RDWR);
    if (fd < 0) {
        printf("Failed to open file '%s'\n", argv[1]);
        return 1;
    }

    // Get the old file mode
    old_mode = fchmod(fd, 0);
    if (old_mode == -1) {
        printf("Failed to get file mode\n");
        return 1;
    }

    // Convert the mode string to octal
    mode = strtol(argv[2], NULL, 8);

    // Set the new file mode
    if (fchmod(fd, mode) == -1) {
        printf("Failed to set file mode\n");
        return 1;
    }

    printf("Changed file mode from %o to %o\n", old_mode, mode);

    close(fd);
```

```
    return 0;
}
```

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*).

```
#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>



int main(int argc, char *argv[]) {

    if (argc != 2) {

        fprintf(stderr, "Usage: %s <file>\n", argv[0]);

        exit(EXIT_FAILURE);

    }



    int fd = open(argv[1], O_RDWR);

    if (fd == -1) {

        perror("open");

        exit(EXIT_FAILURE);

    }



    struct timespec new_times[2];

    new_times[0].tv_nsec = UTIME_OMIT;  // keep atime unchanged

    new_times[1].tv_nsec = UTIME_NOW;   // set mtime to current time



    int result = futimens(fd, new_times);

    if (result == -1) {
```

```
        perror("futimens");

        exit(EXIT_FAILURE);

    }



    printf("Access and modification times of '%s' have been updated.\n",
argv[1]);



    close(fd);

    exit(EXIT_SUCCESS);

}
```

Futimens()

int futimens(int fd, const struct timespec times[2]);

Užduotys

Eksperimentams pasidarykite C programos šabloną pagal pavyzdį. Būtini reikalavimai:

1. Vardas, Pavarde, 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:

L3 OS KONSPEKTAS

```
/* Lukas Kuzmickas IFF-1/6 lukkuz1 */
/* Failas: login_sablonas.c */

#include <stdio.h>

int vp_test();

int vp_test(){
   int d = 5 + 5;
   return d;
}

int main( int argc, char * argv[] ){
   printf( "(C) 2023 Lukas Kuzmickas, %s\n", __FILE__ );
   int d = vp_test();
   printf("Ats: %d\n", d);
   return 0;
}
```

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

Išbandykite ir išsiaiškinkite, kaip veikia source:posix|kespaul_open01.c programa:

- Kaip paleisti programą, kad būtų vykdomi `open()` ir `close()` iškvietimai:

Open()

Funkcija `open()` naudojama atverti failą arba sukurti naują failą. Ji priima tris argumentus:

1. `const char *pathname` - kelias (absoliutus arba reliatyvus), kuriame yra norimas failas. Tai gali būti failo vardas arba failo kelias iki failo.

2. `int flags` - reikšmė, kuri nusako, kaip `open()` turi atverti failą. Ši reikšmė gali būti sudaryta iš kelių flagų, sudedamų su bitų operacijomis.

Funkcija `open()` turi kelis skirtingus parametrus (`flags`), kurie nurodo, kokios operacijos bus atliekamos su atidarytu failu. Šie parametrai yra:

- `O_RDONLY`: Failas atidaromas tik skaitymui;

- `O_WRONLY`: Failas atidaromas tik rašymui;

- `O_RDWR`: Failas atidaromas skaitymui ir rašymui;

- `O_CREAT`: Sukuria failą, jei jis neegzistuoja;

- `O_EXCL`: Grąžina klaidą, jei failas jau egzistuoja;

- `O_TRUNC`: Apkerpa failą iki 0 dydžio, jei jis egzistuoja;

- `O_APPEND`: Rašant failą, duomenys yra pridedami į failo pabaigą;

- `O_NONBLOCK`: Failo operacijos bus vykdomos neblokuojant;

Šie parametrai gali būti kombinuojami naudojant bitinį aritmetikos operatorių OR (`|`), pvz. `O_CREAT | O_WRONLY`.

3. `mode_t mode` - reikšmė, kuri nusako naujo failo leidimus, jei `open()` sukuria naują failą. Tai yra `umask()` reikšmės ir failo leidimų operacijos sandauga.

`mode_t` yra C kalbos tipas, naudojamas nurodyti failo kūrimo režimą (permissions). Tai yra skaitinis reikšmės tipas, kurio reikšmės yra interpretuojamos kaip bitų rinkiniai, kurie nurodo leidimus (permissions) failams. Tai yra 3 arba 4 skaitmenų aibė, pirmasis skaitmuo nurodo failo savininko teises, antrasis - failo grupės teises, o trečiasis - kitiems vartotojams suteiktas teises. Jei yra ketvirtasis skaitmuo, jis nustato ypač svarbias teises (pvz. setuid bitas). Kiekvienas skaitmuo turi savo unikalią reikšmę, susidedančią iš 3 leidimo tipų:

- Skaityti (read) - reikšmė 4

- Rašyti (write) - reikšmė 2

- Vykdyti (execute) - reikšmė 1

Skaičiaus reikšmės gali būti sumuojamos, kad gautumėte daugiau nei vieną leidimą. Pvz. jei norite suteikti savininkui skaitymo, rašymo ir vykdymo teises, skaitmenų reikšmė bus `7` (4+2+1).

Reikšmės, kurios gali būti naudojamos kaip mode_t, apibrėžtos įvairiose konstantose, pvz.:

- `S_IRWXU` - visos teisės savininkui

- `S_IRUSR` - skaitymo teisės savininkui

- `S_IWUSR` - rašymo teisės savininkui

- `S_IXUSR` - vykdymo teisės savininkui

- `S_IRWXG` - visos teisės grupės nariams

- `S_IRGRP` - skaitymo teisės grupės nariams

- `S_IWGRP` - rašymo teisės grupės nariams

- `S_IXGRP` - vykdymo teisės grupės nariams

- `S_IRWXO` - visos teisės kitiems vartotojams

- `S_IROTH` - skaitymo teisės kitiems vartotojams

- `S_IWOTH` - rašymo teisės kitiems vartotojams

- `S_IXOTH` - vykdymo teisės kitiems vartotojams

Visi šie makrai gali būti suderinami su bitine OR operacija. Pvz., jei norite sukurti failą su visomis teisėmis savininkui, skaitymo ir vykdymo teisėmis grupes

```
#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>



int main() {

    int fd;

    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH; // 0644
mode



    fd = open("myfile.txt", O_CREAT | O_WRONLY, mode);

    if (fd == -1) {

        perror("open");

        exit(1);

    }



    write(fd, "Hello, world!\n", 14);



    close(fd);



    return 0;

}
```

L3 OS KONSPEKTAS

Close()

`close()` yra funkcija, skirta uždaryti atidarytą failo deskriptorių. Ji turi vieną argumentą - atidaryto failo deskriptorių, kuriuo bus naudojamasi uždarymo metu. Funkcija grąžina 0, jei sėkmingai uždarytas failas, ir -1, jei įvyko klaida.

Funkcijos deklaracija yra tokia:

```
int close(int fd);
```

Kur `fd` yra failo deskriptorius, kurį reikia uždaryti.

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main() {
   int fd = open("test.txt", O_CREAT | O_WRONLY, 0644);
   if (fd == -1) {
      perror("open");
      return 1;
   }

   printf("Writing to file...\n");
   write(fd, "Hello, world!", 13);

   printf("Closing file...\n");
   int result = close(fd);
   if (result == -1) {
      perror("close");
      return 1;
   }
```

```
    return 0;
}
```

Taip pat reikia atkreipti dėmesį, kad ši funkcija grąžina failo deskriptorių, kuris naudojamas atlikti operacijas su atvertu failu.

sesija

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

Argumentai paduodami mūsų programai, panašiai, kaip scriptų rašyme

argc – bendras argumentų skaičius (dažniausiai pirmas argumentas yra mūsų -o failas)

argv – argumentų character masyvas.

Argv[0] – mūsų paduodamas argumentas pvz: ./failas, kai padarome gcc -o failas failas.c

Argv[1] – pirmasis paduotas argumentas pvz: ./failas failas1

$#

$*

$1

$2 ir t..

- Kodėl dskr lyginamas su -1 (kur tai parašyta dokumentacijoje): ?
- Ką daro `open()` funkcija: ?

  Atidaro failą

- Ką daro `close()` funkcija: ?

  Uždaro failą

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

  Išmeta baigtinį „exit" kodą.

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

  Klaidų tikrinimas, error handling funkcija. (catch try blokas c kalboje)

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

pathconf(), fpathconf()

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

`pathconf()` funkcija priima du argumentus:

1. `const char *path` - kelias (path) į failą ar direktoriją, kuriame reikia patikrinti nustatymus.

2. `int name` - reikšmė, nusakanti konkretų nustatymą, kurio informaciją norite gauti. Tai gali būti vienas iš standartinių simbolių (pvz., `_PC_NAME_MAX`, `_PC_PATH_MAX`, `_PC_FILESIZEBITS`, ir t.t.) arba kuri nors kita papildoma reikšmė, kuri yra susijusi su konkrečiu operaciniu sistemos nustatymu.

- 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 .
```

```
#include <stdio.h>

#include <unistd.h>


int main() {

    long name_max = pathconf(".", _PC_NAME_MAX);

    long path_max = pathconf(".", _PC_PATH_MAX);


    printf("Maksimalus failo vardo ilgis: %ld\n", name_max);

    printf("Maksimalus kelio ilgis: %ld\n", path_max);


    return 0;

}
```

getcwd(), chdir(), fchdir()

Išbandykite `getcwd()`, `chdir()`, `fchdir()` funkcijas, pvz. (source:posix|kespaul_getcwd01.c):

- Kodėl kviečiama `free(cwd)`: ?

Išvaloma atmintis, kai atspausdinama direktorija.

PASTABA: `getcwd()` veikimas, kai pirmu parametru nurodyta `NULL` – nestandartinis, kaip veikia žr.: `man getcwd`.

- Sukurkite C programą **loginas_getcwd02.c** (loginas pakeiskite į savo loginą), kuri:
  1. gautų ir atspausdintų einamo katalogo vardą su `getcwd()` (ir kviečiant `getcwd()` naudotų `pathconf(".",_PC_PATH_MAX)` grąžinamą reikšmę)
  2. atidarytų einamą katalogą su `open()`, įsimintų ir atspausdintų jo deskriptorių
  3. nueitų į `/tmp` katalogą su `chdir()`
  4. patikrintų su `getcwd()` ir atspausdintų koks dabar yra einamasis katalogas
  5. grįžtų į 2-ame žingsnyje atidarytą katalogą su `fchdir()` (ir patikrintų/parodytų, kad tikrai grįžo)
  6. įkelkite programą į Moodle

```
#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <limits.h>

#include <stdlib.h>


int main() {

   char cwd[PATH_MAX];

   int dir_fd, tmp_fd;


   // Gauti ir atspausdinti einamąjį katalogą

   if (getcwd(cwd, pathconf(".", _PC_PATH_MAX)) == NULL) {

      perror("getcwd() error");

      exit(EXIT_FAILURE);

   }

   printf("Einamasis katalogas: %s\n", cwd);


   // Atidaryti einamąjį katalogą ir atspausdinti deskriptorių

   dir_fd = open(".", O_RDONLY);

   if (dir_fd == -1) {

      perror("open() error");
```

```
    exit(EXIT_FAILURE);
}
printf("Einamojo katalogo deskriptorius: %d\n", dir_fd);

// Nueiti į /tmp katalogą ir patikrinti dabartinį katalogą
if (chdir("/tmp") == -1) {
    perror("chdir() error");
    exit(EXIT_FAILURE);
}
if (getcwd(cwd, pathconf(".", _PC_PATH_MAX)) == NULL) {
    perror("getcwd() error");
    exit(EXIT_FAILURE);
}
printf("Dabartinis katalogas: %s\n", cwd);

// Grįžti į einamąjį katalogą ir patikrinti, ar grąžta
if (fchdir(dir_fd) == -1) {
    perror("fchdir() error");
    exit(EXIT_FAILURE);
}
if (getcwd(cwd, pathconf(".", _PC_PATH_MAX)) == NULL) {
    perror("getcwd() error");
    exit(EXIT_FAILURE);
}
printf("Grįžimo į einamąjį katalogą patikrinimas: %s\n", cwd);

// Uždaryti atidarytą katalogą
if (close(dir_fd) == -1) {
    perror("close() error");
```

```
    exit(EXIT_FAILURE);

  }


  return 0;

}
```

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.

```
#include <stdio.h>

#include <dirent.h>


int main() {

  DIR *dir;

  struct dirent *dp;


  if ((dir = opendir(".")) == NULL) {

    perror("opendir() klaida");

    return -1;

  }


  while ((dp = readdir(dir)) != NULL) {

    printf("Inodo numeris: %ld, Failo vardas: %s\n", dp->d_ino, dp->d_name);

  }


  closedir(dir);
```

```
    return 0;

}
```

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:
  - savininko ID
  - dydį
  - i-node numerį
  - leidimus
  - failo tipą (katalogas/failas/kanalas/soketas...)
- Palyginkite savo programos ir *stat* komandos grąžinamus rezultatus įvairiems failams. Turėtumėte matyti tą pačią informaciją (nebūtinai vienodai atvaizduotą).

```
#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>



int main(int argc, char *argv[]) {

    if (argc != 2) {

        printf("Usage: %s <file or directory>\n", argv[0]);

        exit(EXIT_FAILURE);

    }



    struct stat file_stat;

    if (stat(argv[1], &file_stat) == -1) {

        perror("stat");

        exit(EXIT_FAILURE);

    }



    printf("Savininko ID: %d\n", file_stat.st_uid);

    printf("Dydis: %ld\n", file_stat.st_size);
```

```
    printf("I-node numeris: %ld\n", file_stat.st_ino);

    printf("Leidimai: %o\n", file_stat.st_mode & 0777);



    if (S_ISREG(file_stat.st_mode))

        printf("Failas\n");

    else if (S_ISDIR(file_stat.st_mode))

        printf("Katalogas\n");

    else if (S_ISCHR(file_stat.st_mode))

        printf("Charakterinis įrenginys\n");

    else if (S_ISBLK(file_stat.st_mode))

        printf("Blokinis įrenginys\n");

    else if (S_ISFIFO(file_stat.st_mode))

        printf("FIFO arba kanalas\n");

    else if (S_ISSOCK(file_stat.st_mode))

        printf("Socket arba lizdas\n");



    return 0;

}
```

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:
  - failų sistemos bloko dydis
  - failų sistemos ID
  - failų sistemos dydis
  - maksimalų failo kelio/vardo ilgis
- Sukurtą programą įkelkite į Moodle.

```
#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <sys/vfs.h>
```

```
#include <sys/statvfs.h>

int main(int argc, char *argv[]) {
    struct stat sb;
    struct statvfs vfs;
    char *path;

    if (argc != 2) {
        fprintf(stderr, "Naudojimas: %s <failas ar katalogas>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    path = argv[1];

    if (stat(path, &sb) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    if (statvfs(path, &vfs) == -1) {
        perror("statvfs");
        exit(EXIT_FAILURE);
    }

    printf("Failo informacija:\n");
    printf("  Savininko ID: %u\n", sb.st_uid);
    printf("  Dydžio: %ld baitai\n", sb.st_size);
    printf("  I-node numeris: %lu\n", sb.st_ino);
    printf("  Leidimai: %o\n", sb.st_mode & 0777);
    printf("  Tipas: ");

    switch (sb.st_mode & S_IFMT) {
        case S_IFBLK:
            printf("blokinis įrenginys\n");
            break;
        case S_IFCHR:
            printf("simbolinis įrenginys\n");
            break;
```

```
        case S_IFDIR:
            printf("katalogas\n");
            break;
        case S_IFIFO:
            printf("FIFO/kanalas\n");
            break;
        case S_IFLNK:
            printf("simbolinis nuoroda\n");
            break;
        case S_IFREG:
            printf("įprastas failas\n");
            break;
        case S_IFSOCK:
            printf("socket'as\n");
            break;
        default:
            printf("nežinomas\n");
            break;
    }

    printf("Failų sistemos informacija:\n");
    printf("  Blokų dydis: %ld baitai\n", vfs.f_bsize);
    printf("  Failų sistemos ID: %ld\n", vfs.f_fsid);
    printf("  Failų sistemos dydis: %ld blokai\n", vfs.f_blocks);
    printf("  Maksimalus failo kelio/vardo ilgis: %ld\n", vfs.f_namemax);

    exit(EXIT_SUCCESS);
}
```

nftw()

- Sukurkite ir išbandykite programą **loginas_nftw02.c**, kuri su `nftw()` išvaikščiotų 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.

```
#define _XOPEN_SOURCE 500 // for nftw()

#include <stdio.h>

#include <stdlib.h>

#include <ftw.h>
```

```
int list_files(const char *path, const struct stat *statptr, int typeflag, struct FTW *ftwbuf)
{
   if (typeflag == FTW_F) {
      printf("%s\n", path);
   }
   return 0;
}

int main(int argc, char *argv[])
{
   if (argc != 2) {
      fprintf(stderr, "Usage: %s <directory>\n", argv[0]);
      exit(EXIT_FAILURE);
   }

   if (nftw(argv[1], list_files, 10, FTW_PHYS) == -1) {
      perror("nftw");
      exit(EXIT_FAILURE);
   }

   return 0;
}
```

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

Kitos funkcijos

- 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
- Sukurtą programą įkelkite į Moodle.

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <directory>\n", argv[0]);
        return 1;
    }

    if (mkdir(argv[1], 0200) == -1) {
        perror("mkdir");
        return 1;
    }

    printf("Directory '%s' created successfully.\n", argv[1]);

    return 0;
}
```

L3_3

POSIX C I/O¶

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)
- pagalbinės (klaidų apdorojimo, buferių „išstūmimo“ į diską ir t.t.).

Failo atidarymas

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 šiol ją naudojom 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 naudojom `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:
    1. `O_EXEC`
    2. **`O_RDONLY`** – failas atidaromas **tik skaitymui**
    3. **`O_RDWR`** – failas atidaromas **skaitymui ir rašymui**
    4. `O_SEARCH`
    5. **`O_WRONLY`** – failas atidaromas **tik rašymui**
2. gali būti (bet nebūtina) su `OR` prijungta bet kurios vėliavėlės iš šių:
    1. **`O_APPEND`** – atidarius, operacijų **poslinkis (*offset*) nustatomas į failo galą** (t. y. kad, rašymo operacija papildytu failą)
    2. O_CLOEXEC
    3. **`O_CREAT`** – jei failo nėra jis **sukuriamas**, o jo **prieigos teises nusako `mode` parametras** (įvertinant `umask()` maskę, lygiai taip kaip ankstesnio užsiėmimo metu nagrinėtai `creat()` funkcijai)
    4. `O_DIRECTORY`
    5. `O_DSYNC` – (SIO) ...
    6. `O_EXCL`
    7. `O_NOCTTY`
    8. `O_NOFOLLOW`
    9. `O_NONBLOCK`
    10. `O_RSYNC` – (SIO) ...
    11. **`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.
    12. **`O_TRUNC`** – atidaromo failo **turinys ištrinamas** (failas tampa 0 baitų dydžio)

13. `O_TTY_INIT`

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

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

5. Čia `mode` parametras `0644` – skaičius aštuntainėje sistemoje (ne dešimtainėj ar šešioliktainėj). Vietoj `0644` galima irgi nurodyti išraišką (smulkiau žr.: `chmod(2)`):

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

Asinchroninis I/O

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

1. **`aio_fildes`** – deskriptorius iš kurio bus skaitoma ar į kurį rašoma (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 (smulkią 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 aiocb 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 nuskaityt atsitiktinius duomenis)
3. kol vyksta skaitymas „ką nors nuveikti" (`kp_test_dummy()` funkcija)
4. patikrinti kas gavosi (ar nuskaitė 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>
```

```c
#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;
```

```
42    memset( (void *)aiorp, 0, sizeof( struct aiocb ) );
43    aiorp->aio_fildes = d;
44    aiorp->aio_buf = buf;
45    aiorp->aio_nbytes = count;
46    aiorp->aio_offset = 0;
47    rv = aio_read( aiorp );
48    if( rv != 0 ){
49       perror( "aio_read failed" );
50       abort();
51    }
52    return rv;
53 }
54
55 int kp_test_dummy( const void *data, int n ){
56    int i, cnt = 0;
57    for( i = 0; i < n; i++ )
58       if( ((char*)data)[i] == '\0' ) cnt++;
59    printf( "Number of '0' in data: %d\n", cnt );
60    return 1;
61 }
62
63 int kp_test_aio_read_waitcomplete( struct aiocb *aiorp ){
64    const struct aiocb *aioptr[1];
65    int rv;
66    aioptr[0] = aiorp;
67    rv = aio_suspend( aioptr, 1, NULL );
68    if( rv != 0 ){
69       perror( "aio_suspend failed" );
70       abort();
71    }
72    rv = aio_return( aiorp );
73    printf( "AIO complete, %d bytes read.\n", rv );
74    return 1;
75 }
76
77 int main( int argc, char * argv[] ){
78    int d;
```

```
79      struct aiocb aior;
80      char buffer[BUFFLEN];
81      printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
82      d = kp_test_open( "/dev/urandom" );
83      kp_test_aio_read_start( d, &aior, buffer, sizeof(buffer) );
84      kp_test_dummy( buffer, sizeof(buffer) );
85      kp_test_aio_read_waitcomplete( &aior );
86      kp_test_close( d );
87      kp_test_dummy( buffer, sizeof(buffer) );
88      return 0;
89  }
```

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 nuskaitomas tik tada, kai kreipiamasi į atitinkamą RAM regioną, įrašomi kai iškviečiama `msync()`, failai uždaromi arba OS nuožiūra (pvz.: trūkstant 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

```
/*
Kestutis
Paulikas
KTK
kespaul
*/
2   /* Failas: kespaul_mmap01.c */
3
4   #include <stdio.h>
```

```c
#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 ){
```

```
42     void *a = NULL;
43     lseek( d, size - 1, SEEK_SET );
44     write( d, &d , 1 );        /* iraso bile ka i failo gala */
45     a = mmap( NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, d, 0 );
46     if( a == MAP_FAILED ){
47        perror( "mmap failed" );
48        abort();
49     }
50     return a;
51  }
52
53  int kp_test_munamp( void *a, int size ){
54     int rv;
55     rv = munmap( a, size );
56     if( rv != 0 ){
57        puts( "munmap failed" );
58        abort();
59     }
60     return 1;
61  }
62
63  int kp_test_usemaped( void *a, int size ){
64     memset( a, 0xF0, size );
65     return 1;
66  }
67
68  int main( int argc, char * argv[] ){
69     int d;
70     void *a = NULL;
71     printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
72     if( argc != 2 ){
73        printf( "Naudojimas:\n %s failas\n", argv[0] );
74        exit( 255 );
75     }
76     d = kp_test_openw( argv[1] );
77     a = kp_test_mmapw( d, SIZE );
78     kp_test_usemaped( a, SIZE );
```

```
79      kp_test_munamp( a, SIZE );
80      kp_test_close( d );
81      return 0;
82  }
```

Užduotys

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()`.

Read()

`read()` funkcijai yra trys argumentai:

`int fd` - failo deskriptorius, kurį gauname iš `open()` funkcijos arba kitos failų operacijos, su kuriuo norime skaityti.

`void *buf` - rodyklė į atminties bloką, į kurį norime nukopijuoti skaitytus duomenis.

`size_t count` - kiek baitų norime nuskaityti iš failo į buferį.

Write()

write() funkcijai yra reikalingi trys argumentai:

1. `int fd`: failo deskriptorius, kuriame bus rašoma informacija;

2. `const void *buf`: rodyklė į atminties vietą, kurioje yra duomenys, kuriuos norime įrašyti;

3. `size_t count`: duomenų skaičius baitais, kurį norime įrašyti į failą.

```
#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>


#define BUFFER_SIZE 2048


int main(int argc, char *argv[]) {

   // check if two command line arguments are provided

   if (argc != 4) {

      printf("Usage: %s input_file output_file num_bytes\n", argv[0]);

      return 1;

   }


   // open the input file for reading

   int input_fd = open(argv[1], O_RDONLY);

   if (input_fd == -1) {

      printf("Error opening input file: %s\n", argv[1]);

      return 1;

   }


   // open the output file for writing

   int output_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR |
S_IRGRP | S_IWGRP | S_IROTH);

   if (output_fd == -1) {

      printf("Error opening output file: %s\n", argv[2]);

      return 1;

   }


   // copy the specified number of bytes from the input file to the output file
```

```
int num_bytes = atoi(argv[3]);

char buffer[BUFFER_SIZE];

int bytes_read, bytes_written = 0;



      bytes_read = read(input_fd, buffer, BUFFER_SIZE);



if (num_bytes > bytes_read) {

   bytes_written = write(output_fd, buffer, bytes_read);

   if (bytes_written != bytes_read) {

      printf("Error writing to output file: %s\n", argv[2]);

      return 1;

   }

}

      else

      {

               bytes_written = write(output_fd, buffer, num_bytes);

      }



if (bytes_read == -1) {

   printf("Error reading input file: %s\n", argv[1]);

   return 1;

}



   printf("Successfully copied %d bytes from %s to %s\n", bytes_written, argv[1],
argv[2]);



   // close the files

   close(input_fd);

   close(output_fd);
```

```
    return 0;

}
```

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()`.

Lseek()

Funkcijai `lseek()` įeina trys argumentai:

1. failo deskriptorius (file descriptor) - tai sveikasis skaičius, kuris unikaliai identifikuoja atidarytą failą.

2. offset - tai baitų skaičius, kiek reikia paslinkti failo rodyklę.

3. whence - tai reikšmė, kuri nusako, kaip apskaičiuoti naują failo rodyklės poziciją, naudojant offset. `whence` gali įgyti tris reikšmes:

- `SEEK_SET`: nauja pozicija nustatoma atsižvelgiant į failo pradžią. Offset nurodo, kiek baitų reikia paslinkti failo pradžios poziciją.

- `SEEK_CUR`: nauja pozicija nustatoma atsižvelgiant į esamą failo poziciją (t.y. tam tikrą baitų kiekį nuo esamos pozicijos). Offset nurodo, kiek baitų reikia paslinkti nuo dabartinės failo pozicijos.

- `SEEK_END`: nauja pozicija nustatoma atsižvelgiant į failo pabaigą. Offset nurodo, kiek baitų reikia paslinkti failo pabaigos poziciją.

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

```
lukkuz1@oslinux ~/lab3_3/c_uzduotys $ ls -l failas9

-rw------- 1 lukkuz1 users 1048577 Apr 30 12:42 failas9

lukkuz1@oslinux ~/lab3_3/c_uzduotys $ du failas9

4       failas9

lukkuz1@oslinux ~/lab3_3/c_uzduotys $ du
```

140 .

lukkuz1@oslinux ~/lab3_3/c_uzduotys $ du failas9

4 failas9

lukkuz1@oslinux ~/lab3_3/c_uzduotys $ stat failas9

File: failas9

Size: 1048577 Blocks: 8 IO Block: 4096 regular file

Device: 801h/2049d Inode: 1326766 Links: 1

Access: (0600/-rw-------) Uid: (2221/ lukkuz1) Gid: (100/ users)

Access: 2023-04-30 12:42:03.108798745 +0300

Modify: 2023-04-30 12:42:03.108798745 +0300

Change: 2023-04-30 12:42:03.108798745 +0300

Birth: -

lukkuz1@oslinux ~/lab3_3/c_uzduotys $

Buferizuotas I/O

Nukopijuokite loginas_rw01.c į **loginas_frw01.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.

```
#include <stdio.h>

#include <stdlib.h>



#define BUFFER_SIZE 2048



int main(int argc, char *argv[]) {

    // check if three command line arguments are provided

    if (argc != 4) {

        printf("Usage: %s input_file output_file num_bytes\n", argv[0]);

        return 1;

    }
```

```
// open the input file for reading
FILE *input_file = fopen(argv[1], "rb");
if (input_file == NULL) {
    printf("Error opening input file: %s\n", argv[1]);
    return 1;
}

// open the output file for writing
FILE *output_file = fopen(argv[2], "wb");
if (output_file == NULL) {
    printf("Error opening output file: %s\n", argv[2]);
    return 1;
}

// copy the specified number of bytes from the input file to the output file
int num_bytes = atoi(argv[3]);
char buffer[BUFFER_SIZE];
int bytes_read, bytes_written = 0;

bytes_read = fread(buffer, 1, BUFFER_SIZE, input_file);

if (num_bytes > bytes_read) {
    bytes_written = fwrite(buffer, 1, bytes_read, output_file);
    if (bytes_written != bytes_read) {
        printf("Error writing to output file: %s\n", argv[2]);
        return 1;
    }
}
```

```
    else {
        bytes_written = fwrite(buffer, 1, num_bytes, output_file);
    }

    if (bytes_read == 0) {
        printf("Error reading input file: %s\n", argv[1]);
        return 1;
    }

    printf("Successfully copied %d bytes from %s to %s\n", bytes_written, argv[1], argv[2]);

    // close the files
    fclose(input_file);
    fclose(output_file);

    return 0;
}
f
```

fopen()

Funkcija `fopen()` yra C kalbos bibliotekos dalis, skirta atidaryti failus tam, kad galėtų būti skaityti ar rašomi. `fopen()` funkcijai reikia du argumentus: failo pavadinimas ir režimas, kuriuo bus atidarytas failas.

Failo pavadinimas gali būti tekstas, turintis kelius iki failo, arba tiesiog failo pavadinimas, jei failas yra tame pačiame kataloge, kur yra vykdomasis failas.

Režimas nurodo, kaip bus atidarytas failas. Galimi režimai yra:

- `"r"` - failas atidaromas skaitymui;
- `"w"` - failas atidaromas rašymui, turinys bus išvalytas;
- `"a"` - failas atidaromas rašymui, nauji duomenys bus pridėti prie esančio failo galo;

- `"r+"` - failas atidaromas skaitymui ir rašymui, pradžioje;

- `"w+"` - failas atidaromas skaitymui ir rašymui, turinys bus išvalytas;

- `"a+"` - failas atidaromas skaitymui ir rašymui, nauji duomenys bus pridėti prie esančio failo galo.

`fopen()` grąžina nuorodą į failo struktūrą, kuri naudojama vėliau, kai atliekami operacijos su failu. Tai yra rodyklė į failą.

Fclose()

Reikia vieno argumento, rodyklės į failą, kurio srautą norime uždaryti.

Fwrite()

`fwrite()` yra C kalbos funkcija, skirta įrašyti duomenis į failą. Funkcija priima keturis argumentus:

1. Pointeris į duomenų masyvą, kuris turi būti įrašytas į failą.
2. Dydis, nurodo vieno elemento dydį baitais.
3. Kiekis, nurodo kiek elementų norima įrašyti į failą.
4. Failo pointeris, kuris buvo grąžintas iš `fopen()` funkcijos.

`fwrite()` grąžina kiek elementų buvo sėkmingai įrašyta į failą.

Fread()

`fread()` yra C standartinė biblioteka, skirta skaityti iš duomenų srauto, t. y. iš failo, kuriame saugomi tam tikro tipo duomenys, ir juos įrašyti į atminties sritį, pvz., į buferio masyvą. Funkcija yra aprašoma kaip:

```
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
```

`fread()` funkcijai yra perduodami keturi argumentai:

L3 OS KONSPEKTAS

1. `ptr` - rodyklė į atminties vietą, kurioje bus saugomas nuskaitytas turinys.

2. `size` - dydis baitais, kiekvieno nuskaityto objekto (pvz., struktūros) reikšmė.

3. `count` - skaičius objektų, kuriuos reikia nuskaityti.

4. `stream` - rodyklė į `FILE` tipo struktūrą, kuri reprezentuoja duomenų srautą, iš kurio bus nuskaitomi duomenys.

Funkcija grąžina skaičių, nurodantį sėkmingai nuskaitytų objektų skaičių. Jei nuskaityta mažiau objektų, nei yra prašoma, funkcija grąžina reikšmę, mažesnę nei `count`. Jei `fread()` grąžina nulį, tai gali reikšti, kad pasiektas failo pabaigos signalas, arba kad įvyko klaida. Klaidą galima patikrinti `ferror()` funkcija.

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

Vienas naudoja stdin, kitas gali naudoti belekokį srautą, kad gautų IO.

L3 OS KONSPEKTAS

Asinchroninis I/O

- Kiek duomenų nuskaito **source:posix|kespaul_aio01.c** programa: ? (ar nuskaito užsiprašytą 1MB)
- Sukurkite programą **loginas_aio02.c** (galite naudoti pavyzdžio ar savo anksčiau sukurtų programų fragmentus), kuri iš `/dev/random` 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.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

#include <aio.h>

#include <errno.h>



#define BUFSIZE 1024



int main(int argc, char *argv[]) {

    int fd;

    struct aiocb aior;

    char buf[BUFSIZE];

    ssize_t nread;



    if (argc != 3) {

        fprintf(stderr, "Usage: %s <filename> <num_bytes>\n", argv[0]);

        exit(1);

    }



    fd = open(argv[1], O_RDONLY);

    if (fd == -1) {

        perror("open failed");
```

```
    exit(1);
}

memset(&aior, 0, sizeof(struct aiocb));
aior.aio_fildes = fd;
aior.aio_buf = buf;
aior.aio_nbytes = atoi(argv[2]);
aior.aio_offset = 0;

if (aio_read(&aior) == -1) {
    perror("aio_read failed");
    exit(1);
}

while (aio_error(&aior) == EINPROGRESS) {}

if ((nread = aio_return(&aior)) == -1) {
    perror("aio_return failed");
    exit(1);
}

printf("%ld bytes read\n", nread);

if (close(fd) == -1) {
    perror("close failed");
    exit(1);
}

return 0;
```

}gcc -o lukkuz1_aio02 lukkuz1_aio02.c -lrt

- 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
```

Failų „mapinimas“ į RAM

- 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. atidarytų ir prijungtų 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. atjungtų abu failus
    4. uždarytų abu deskriptorius

```
#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 <string.h>

#define MAX_FILE_SIZE 100 * 1024 * 1024  // Maximum file size of 100 MB

int main(int argc, char *argv[]) {
    int fd1, fd2;
    char *map1, *map2;
    struct stat file_stat;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s file1 file2\n", argv[0]);
        exit(1);
    }

    // Open and map the first file for reading
    fd1 = open(argv[1], O_RDONLY);
    if (fd1 == -1) {
```

```
        perror("open file1 failed");
        exit(1);
    }
    if (fstat(fd1, &file_stat) == -1) {
        perror("fstat file1 failed");
        exit(1);
    }
    if (file_stat.st_size > MAX_FILE_SIZE) {
        fprintf(stderr, "file1 is too large, maximum size is %d bytes\n",
MAX_FILE_SIZE);
        exit(1);
    }
      if (file_stat.st_size == 0) {
    fprintf(stderr, "file1 is empty\n");
    exit(1);
    }

    map1 = mmap(NULL, file_stat.st_size, PROT_READ, MAP_SHARED, fd1, 0);
    if (map1 == MAP_FAILED) {
        perror("mmap file1 failed");
        exit(1);
    }

    // Open and map the second file for writing
    fd2 = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0640);
    if (fd2 == -1) {
    perror("open file2 failed");
    exit(1);
    }

      if (ftruncate(fd2, file_stat.st_size) == -1) {
    perror("ftruncate file2 failed");
    exit(1);
    }
```

```
    map2 = mmap(NULL, file_stat.st_size, PROT_READ | PROT_WRITE, MAP_SHARED,
fd2, 0);
    if (map2 == MAP_FAILED) {
        perror("mmap file2 failed");
        exit(1);
    }

    // Copy the contents of the first file to the second file
    memcpy(map2, map1, file_stat.st_size);

    // Unmap and close both files
    if (munmap(map1, file_stat.st_size) == -1) {
        perror("munmap file1 failed");
        exit(1);
    }
    if (munmap(map2, file_stat.st_size) == -1) {
        perror("munmap file2 failed");
        exit(1);
    }
    if (close(fd1) == -1) {
        perror("close file1 failed");
        exit(1);
    }
    if (close(fd2) == -1) {
        perror("close file2 failed");
        exit(1);
    }

    return 0;
}
```

Aio pavyzdžiai

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <aio.h>
#include <errno.h>

#define BUFSIZE 1024

int main(int argc, char *argv[]) {
    int fd;
    struct aiocb aior;
    char *buf = "Hello, world!\n";
    ssize_t nwrite;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(1);
    }

    fd = open(argv[1], O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open failed");
        exit(1);
    }

    memset(&aior, 0, sizeof(struct aiocb));
    aior.aio_fildes = fd;
    aior.aio_buf = buf;
    aior.aio_nbytes = strlen(buf);
    aior.aio_offset = 0;

    if (aio_write(&aior) == -1) {
        perror("aio_write failed");
        exit(1);
    }

    while (aio_error(&aior) == EINPROGRESS) {}

    if ((nwrite = aio_return(&aior)) == -1) {
        perror("aio_return failed");
        exit(1);
    }

    printf("%ld bytes written\n", nwrite);

    if (close(fd) == -1) {
        perror("close failed");
        exit(1);
```

```
    }

    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <aio.h>
#include <errno.h>

#define BUFSIZE 1024

int main(int argc, char *argv[]) {
    int fd;
    struct aiocb aior;
    char buf[BUFSIZE];
    ssize_t nread;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(1);
    }

    fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open failed");
        exit(1);
    }

    memset(&aior, 0, sizeof(struct aiocb));
    aior.aio_fildes = fd;
    aior.aio_buf = buf;
    aior.aio_nbytes = BUFSIZE;
    aior.aio_offset = 0;

    if (aio_read(&aior) == -1) {
        perror("aio_read failed");
        exit(1);
    }

    while (aio_error(&aior) == EINPROGRESS) {}
```

```
    if ((nread = aio_return(&aior)) == -1) {
        perror("aio_return failed");
        exit(1);
    }

    printf("%ld bytes read: %.*s\n", nread, (int)nread, buf);

    if (close(fd) == -1) {
        perror("close failed");
        exit(1);
    }

    return 0;
}
```

```
int aio_read(struct aiocb *aiocbp);
struct aiocb {
    int aio_fildes;    /* File descriptor */
    off_t aio_offset;  /* File offset */
    volatile void *aio_buf; /* Buffer */
    size_t aio_nbytes; /* Number of bytes to read */
    int aio_reqprio;   /* Request priority */
    struct sigevent aio_sigevent; /* Signal event to notify upon completion */
    int aio_lio_opcode; /* Operation to be performed */
};
int aio_write(struct aiocb *aiocbp);
```

Mmap pavyzdžiai

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

int main(int argc, char *argv[]) {
    int fd;
    struct stat file_stat;
    char *map;
```

```
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(1);
    }

    // Open the file for reading
    fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open failed");
        exit(1);
    }

    // Get the file size
    if (fstat(fd, &file_stat) == -1) {
        perror("fstat failed");
        exit(1);
    }

    // Map the file into memory
    map = mmap(NULL, file_stat.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (map == MAP_FAILED) {
        perror("mmap failed");
        exit(1);
    }

    // Print the contents of the file
    printf("%s", map);

    // Unmap and close the file
    if (munmap(map, file_stat.st_size) == -1) {
        perror("munmap failed");
        exit(1);
    }
    if (close(fd) == -1) {
        perror("close failed");
        exit(1);
    }

    return 0;
}
```

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

- **`addr`**: The desired starting address for the mapping. It can be **`NULL`**, in which case the system will choose a suitable address.
- **`length`**: The length of the mapping in bytes.

- **`prot`**: The protection of the mapping, which can be a combination of **`PROT_NONE`**, **`PROT_READ`**, **`PROT_WRITE`**, and **`PROT_EXEC`**.
- **`flags`**: The type of mapping, which can be a combination of **`MAP_SHARED`**, **`MAP_PRIVATE`**, **`MAP_FIXED`**, **`MAP_ANONYMOUS`**, **`MAP_NORESERVE`**, **`MAP_POPULATE`**, and **`MAP_HUGETLB`**.
- **`fd`**: The file descriptor of the file to be mapped. This argument is ignored when **`MAP_ANONYMOUS`** is set in the **`flags`** parameter.
- **`offset`**: The offset from the beginning of the file at which to start the mapping. This argument is ignored when **`MAP_ANONYMOUS`** is set in the **`flags`** parameter.

int munmap(void *addr, size_t length);

- **`addr`**: The starting address of the mapping returned by **`mmap()`**.
- **`length`**: The length of the mapping in bytes. This should be the same as the **`length`** argument passed to **`mmap()`**.

L3_4

Pagrindinės *calling convention* savybės:

1. kaip perduodami argumentai: registrais, per steka, …;
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*).

Informacija apie procesams taikomus apribojimus

Funkcija **`sysconf()`** labai panaši į pirmo šio laboratorinio darbo užsiėmimo metu nagrinėtą `pathconf()` funkciją, tik grąžina informaciją ne apie failų sistemą, o apie visą sistemą (t. y. grąžina parametrus, kurie nepriklauso nuo to, kuri failų sistema naudojama).

Sysconf()

Vienas argumentas.

- **`_SC_ARG_MAX`**: maksimalus argumentų skaičius, kurį procesas gali perduoti **`exec()`** funkcijai
- **`_SC_CHILD_MAX`**: maksimalus vaikių procesų skaičius, kurį gali turėti vienas naudotojas
- **`_SC_CLK_TCK`**: laikrodžio impulsų skaičius per sekundę
- **`_SC_NGROUPS_MAX`**: maksimalus papildomų grupių ID skaičius procesui
- **`_SC_OPEN_MAX`**: maksimalus atvirų failų skaičius vienu metu, kurį gali turėti procesas
- **`_SC_PAGESIZE`**: puslapio dydis baitais
- **`_SC_RTSIG_MAX`**: maksimalus realaus laiko signalų skaičius, kuris yra prieinamas procesui
- **`_SC_THREAD_ATTR_STACKADDR`**: minimalus steko dydis baitais gijos atributų objektui
- **`_SC_THREAD_DESTRUCTOR_ITERATIONS`**: kiek kartų gali būti bandoma sunaikinti gijos specifinio duomenų rakto ir susijusių gijos specifinių duomenų reikšmes
- **`_SC_THREAD_KEYS_MAX`**: maksimalus gijų specifinių duomenų raktų skaičius, kurį gali turėti procesas
- **`_SC_THREAD_STACK_MIN`**: minimalus steko dydis baitais gijai
- **`_SC_THREAD_THREADS_MAX`**: maksimalus aktyvių gijų skaičius, kurį gali turėti procesas
- **`_SC_TTY_NAME_MAX`**: maksimalus TTY pavadinimo ilgis
- **`_SC_VERSION`**: realizacijos specifinės **`POSIX.1`** standarto versijos numeris

Confstr()

The `confstr()` function is a system call in the C programming language that retrieves system configuration information at runtime, similar to the `sysconf()` function. However, instead of taking a single argument, `confstr()` takes two arguments:

- `name`: specifies the system configuration value to retrieve. This argument should be one of the following macros defined in the `<unistd.h>` header file:

- `_CS_PATH`: the search path used by the `execvp()` function

- `_CS_POSIX_V6_ILP32_OFF32_CFLAGS`: POSIX.1-2001 standard C compiler flags for an ILP32 data model with a 32-bit `long` and `int`

- `_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS`: POSIX.1-2001 standard C compiler flags for an ILP32 data model with a 64-bit `long` and a 32-bit `int`

- `_CS_POSIX_V6_LP64_OFF64_CFLAGS`: POSIX.1-2001 standard C compiler flags for an LP64 data model with a 64-bit `long` and `int`

- `_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS`: POSIX.1-2001 standard C compiler flags for an LP64 data model with a 64-bit `long` and `int`

- `buf`: a character buffer in which the configuration value is stored.

The `confstr()` function returns the number of characters stored in `buf`, not including the terminating null character. If `buf` is not large enough to hold the configuration value, the function returns the number of characters required for the buffer size.

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

`sysconf()` naudojimo pavyzdys:

```c
/* 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 );
```

```
15    printf( "Total system RAM: %.1f MB\n", (double)pagesize * pages / 1024 / 1024 );
16    return 1;
17 }
18
19 int main(){
20    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
21    kp_memsize();
22    return 0;
23 }
```

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

int main()
{
    long size;
    char *buf;
    size_t len;

    // get the required size of the buffer
    size = confstr(_CS_GNU_LIBC_VERSION, NULL, 0);

    // allocate the buffer of required size
    buf = (char *) malloc(size);
    if (buf == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // retrieve the configuration string
    len = confstr(_CS_GNU_LIBC_VERSION, buf, size);
    if (len == 0) {
        perror("confstr");
        exit(EXIT_FAILURE);
```

```
    }

    // print the configuration string
    printf("GNU libc version: %s\n", buf);

    // free the buffer
    free(buf);

    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, maksimalų kuriamo failo dydį ir t.t. `getrlimit()` ir `setrlimit()` naudojimo pavyzdys:

Getrlimit()

`getrlimit()` funkcijai yra du argumentai:

1. **`resource`**: Tai yra resursas, kurio apribojimus norite gauti. Šis argumentas gali būti viena iš daugelio konstantų, nurodančių specifinį resursą, pvz.:
    - **`RLIMIT_CPU`**: Centrinio procesoriaus naudojimo laiko trukmės riba (sekundėmis).
    - **`RLIMIT_FSIZE`**: Failo dydžio riba, kurio galima sukurti arba rašyti į jį (baitais).
    - **`RLIMIT_DATA`**: Duomenų srities (sąsajos segmentų) dydžio riba, kurio procesas gali naudoti (baitais).
    - **`RLIMIT_STACK`**: Peržiūros steko dydis, kurioje saugomi procesų funkcijų kvietimų kontekstai (baitais).
    - **`RLIMIT_CORE`**: Branduolio failo dydžio riba, kurio galima sukurti, kai procesas nusikaltimo metu sustoja (baitais).
    - ir kt.
2. **`rlim`**: Tai yra struktūra, į kurią **`getrlimit()`** funkcija įrašys dabartinius apribojimus. Struktūra **`rlimit`** turi du narius:
    - **`rlim_cur`**: Tai yra dabartinis apribojimo dydis.
    - **`rlim_max`**: Tai yra maksimalus apribojimo dydis. Tik privilegijuoti naudotojai gali padidinti apribojimą iki šio dydžio.

L3 OS KONSPEKTAS

Setrlimit()

`setrlimit()` funkcijai yra du argumentai:

1. **`resource`**: Tai yra resursas, kuriam reikia nustatyti apribojimus. Šis argumentas gali būti viena iš daugelio konstantų, nurodančių specifinį resursą, pvz.:
    - **`RLIMIT_CPU`**: Ribojama centrinio procesoriaus naudojimo laiko trukmė (sekundėmis).
    - **`RLIMIT_FSIZE`**: Ribojamas failo dydis, kurį galima sukurti arba rašyti į jį (baitais).
    - **`RLIMIT_DATA`**: Ribojama duomenų sritis (sąsajos segmentai) dydis, kuriuos procesas gali naudoti (baitais).
    - **`RLIMIT_STACK`**: Ribojamas peržiūros steko dydis, kurioje saugomi procesų funkcijų kvietimų kontekstai (baitais).
    - **`RLIMIT_CORE`**: Ribojamas branduolio failo dydis, kurį galima sukurti, kai procesas nusikaltimo metu sustoja (baitais).
    - ir kt.
2. **`rlim`**: Tai yra struktūra, kurioje yra numatytieji ir nauji apribojimai, kurie turi būti nustatyti. Struktūra **`rlimit`** turi du narius:
    - **`rlim_cur`**: Tai yra naujas apribojimo dydis. Jei šis narys yra **`RLIM_INFINITY`**, resursas nėra apribotas.
    - **`rlim_max`**: Tai yra maksimalus apribojimo dydis. Tik privilegijuoti naudotojai gali padidinti apribojimą iki šio dydžio.

```
/*
Kestutis
Paulikas
KTK
kespaul
*/
```

```
2  /* Failas: kespaul_limits02.c */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <sys/resource.h>
10
11 int kp_change_filelimit( int nslimit, int nhlimit );
12 int kp_test_filelimit( const char *fn );
```

```
13
14 int kp_change_filelimit( int nslimit, int nhlimit ){
15    struct rlimit rl;
16    getrlimit( RLIMIT_NOFILE, &rl );
17    printf( "RLIMIT_NOFILE %ld (soft) %ld (hard)\n", rl.rlim_cur, rl.rlim_max );
18    rl.rlim_cur = nslimit;
19    rl.rlim_max = nhlimit;
20    setrlimit( RLIMIT_NOFILE, &rl );
21    getrlimit( RLIMIT_NOFILE, &rl );
22    printf( "RLIMIT_NOFILE %ld (soft) %ld (hard)\n", rl.rlim_cur, rl.rlim_max );
23    return 1;
24 }
25
26 int kp_test_filelimit( const char *fn ){
27    int n;
28    for( n = 0; -1 != open( fn, O_RDONLY ); n++ );
29    printf( "Can open %d files\n", n );
30    return 1;
31 }
32
33 int main( int argc, char *argv[] ){
34    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
35    kp_change_filelimit( 20, 100 );
36    kp_test_filelimit( argv[0] );
37    return 0;
38 }
```

L3 OS KONSPEKTAS

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 `kespaul_lib01.h` header failą.

`kespaul_lib01.h` (source:posix|lib/kespaul_lib01.h)

```
1  /* Kestutis Paulikas KTK kespaul */
2  /* Failas: kespaul_lib01.h */
3
4  #ifndef kespaul_lib01_h
5  #define kespaul_lib01_h
6
7  /* apie "extern" zr.: C99.pdf 6.2.2 */
8  extern int kespaul_libfunc01( const char *s );
9
10 extern double kespaul_libvar01;
11
12 #endif
```

`kespaul_lib01.c` (source:posix|lib/kespaul_lib01.c)

```
1  /* Kestutis Paulikas KTK kespaul */
2  /* Failas: kespaul_lib01.c */
3
4  #include <stdio.h>
5  #include "kespaul_lib01.h"
6
7  double kespaul_libvar01;
8
9  int kespaul_libfunc01( const char *s ){
10    printf( "Dynamic library for testing, %s\n", __FILE__ );
11    printf( "\tparameter: \"%s\"\n", s );
12    printf( "\tlib variable = %f\n", kespaul_libvar01 );
13    return 0;
14 }
```

Kompiliuojant dinaminę biblioteką reikalingi papildomi parametrai:

1. **`-fpic`** – nurodo, kad generuojamas PIC mašininis kodas;
2. **`-shared`** – nurodo, kad kuriamas dinaminis failas, kuris gali būti apjungiamas 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
```

Bibliotekos prikabinimas ryšių redaktoriaus pagalba

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

kespaul_test_lib01a.c (source:posix|kespaul_test_lib01a.c)

```c
/* 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 prikabintos 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: 193.219.36.233 mašinoje LD_LIBRARY_PATH pagal nutylėjimą nenustatytas, bet jei sistemoje šiam kintamąjam 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 kelia, 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 sunku "iškrapštyti". 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
```

L3 OS KONSPEKTAS

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

dlopen() funkcija turi tris pagrindinius argumentus:

1. "filename" - tai bibliotekos failo pavadinimas, kuri norima įkelti.
2. "flag" - nurodo būdą, kaip biblioteka turėtų būti įkelta. Galimi variantai: RTLD_LAZY, RTLD_NOW, RTLD_GLOBAL ir RTLD_LOCAL.
3. "error" - jei įvyksta klaida, į šį kintamąjį bus įrašomas klaidos pranešimas.

2. **`dlclose()`** – uždaro dinaminę biblioteką ir atlaisvina jai skirtus adresus

dlclose() funkcija priima vieną argumentą:

1. **`void *handle`** - rodyklė į atidarytą bibliotekos rankinę. Tai yra rodyklė į atminties sritį, kurioje saugoma informacija apie atidarytą biblioteką.

3. **`dlsym()`** – prikabintoje bibliotekoje ieško nurodytos funkcijos ar kintamojo pagal nurodytą vardą

Funkcija dlsym() turi šiuos argumentus:

1. void *handle - atviro dydžio rankenos rodyklė, grąžinta iš funkcijos dlopen().
2. const char *symbol - simbolio pavadinimas, kurio adresas ieškomas.

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.

`kespaul_test_lib01b.c` (source:posix|kespaul_test_lib01b.c)

```
1 /* Kestutis Paulikas KTK kespaul */
2 /* Failas: kespaul_test_lib01b.c */
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <dlfcn.h>
```

```
7
8  int (*fptr)(const char *s);
9
10 double *pd;
11
12 int main(){
13    void *dl;
14    printf( "(C) 2013 kestutis Paulikas, %s\n", __FILE__ );
15    dl = dlopen( "lib/libkespaul01.so", RTLD_LAZY | RTLD_LOCAL );
16    if( dl == NULL ){
17       puts( dlerror() );
18       exit(1);
19    }
20    pd = dlsym( dl, "kespaul_libvar01" );
21    if( pd == NULL ){
22       puts( dlerror() );
23       exit(1);
24    }
25    /* fptr = (int (*)(char*)) dlsym( dl, "kespaul_libfunc01" ); */
26    *(void**)(&fptr) = dlsym( dl, "kespaul_libfunc01" );
27    if( fptr == NULL ){
28       puts( dlerror() );
29       exit(1);
30    }
31    *pd = 5.2;
32    (*fptr)( "Library test (manualy loaded)" );
33    dlclose( dl );
34    return 0;
35 }
```

Šio failo kompiliavimas įprastas, tik reikia nurodyti, jog bus naudojama dinaminio užkrovimo biblioteką `libdl.so`. Pačios bibliotekos, kuri bus užkrauta vykdymo metu, kompiliuojant nurodyti nereikia, bibliotekos header failas irgi gali būti nenaudojamas:

```
$ gcc -g -Wall -Werror -pedantic -ldl -o kespaul_test_lib01b kespaul_test_lib01b.c
```

L3 OS KONSPEKTAS

Užduotys

Užduotys darbui su procesais (POSIX C API)

Informacija apie procesams taikomus apribojimus

- Sukurkite programą **loginas_cpulimit01.c**, kuri nustatytu CPU limitą 1s (`RLIMIT_CPU`) ir patikrinkite, ar limitas suveikia (nustatę limitą užsukite amžiną ciką 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)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>

int main() {
    struct rlimit core_limit;
    core_limit.rlim_cur = 0; // nustatome core limitą į 0
    core_limit.rlim_max = 0;
    if (setrlimit(RLIMIT_CORE, &core_limit) != 0) {
        perror("setrlimit");
        return 1;
    }
    int i = 0;
    while (1) {
        i++;
    }
    printf("Iteracijų skaičius: %d\n", i);
    return 0;

}
```

- Įkelkite programą į Moodle.

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 parametrą išeikite su `_Exit()`, `exit() – LIFO tvarka` , `abort() – nebus iškviečiamos` arba `return – kai programa pabaigia darba`.
- Kaip veikia: iškviečia priregistruotas funkcijas, neiškviečia, iškviečia ne visas, kokia tvarka iškviečia?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

void func1() {
    printf("Funkcija 1\n");
}

void func2() {
    printf("Funkcija 2\n");
}
```

```c
void func3() {
    printf("Funkcija 3\n");
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Programa turi buti paleista su 1 parametru\n");
        exit(EXIT_FAILURE);
    }

    if (atexit(func1) != 0 || atexit(func2) != 0 || atexit(func3) != 0) {
        fprintf(stderr, "Nepavyko priregistruoti funkciju su atexit()\n");
        exit(EXIT_FAILURE);
    }

    char *param = argv[1];

    if (strcmp(param, "_Exit") == 0) {
        _Exit(EXIT_SUCCESS);
    }
    else if (strcmp(param, "exit") == 0) {
        exit(EXIT_SUCCESS);
    }
    else if (strcmp(param, "abort") == 0) {
        abort();
    }
    else if (strcmp(param, "return") == 0) {
        return EXIT_SUCCESS;
    }
    else {
        fprintf(stderr, "Netinkamas programa parametras: %s\n", param);
        exit(EXIT_FAILURE);
    }

    return 0;

}
```

- Įkelkite programą į Moodle.

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 tikti tas pats `loginas_testlib02.h` header failas.

Lukkuz1_testlib02a.c

```c
#include "lukkuz1_testlib02.h"


int vp_testlib(int a) {
   return a * 2;
}
```

Lukkuz1_testlib02b.c

```
#include "lukkuz1_testlib02.h"



int vp_testlib(int a) {

   return a / 2;

}
```

Lukkuz1_testlib02.h

```
#ifndef lukkuz1_testlib02_h

#define lukkuz1_testlib02_h


int vp_testlib(int a);


#endif /* lukkuz1_testlib02_h */
```

lukkuz1@oslinux ~/lab3_4/c_uzduotys $ gcc -shared -fPIC loginas_testlib02a.c -o libloginas02a.so

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

```
#include <stdio.h>

#include <dlfcn.h>

#include "lukkuz1_testlib02.h"



int main() {

   void* lib_handle_1;

   void* lib_handle_2;

   int (*vp_testlib_1)(int);

   int (*vp_testlib_2)(int);

   char *error;



   // Load library 1

   lib_handle_1 = dlopen("libloginas02a.so", RTLD_LAZY);

   if (!lib_handle_1) {
```

```
    fprintf(stderr, "%s\n", dlerror());
    return 1;
}

// Load library 2
lib_handle_2 = dlopen("libloginas02b.so", RTLD_LAZY);
if (!lib_handle_2) {
    fprintf(stderr, "%s\n", dlerror());
    return 1;
}

// Load function vp_testlib from library 1
vp_testlib_1 = dlsym(lib_handle_1, "vp_testlib");
if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    return 1;
}

// Load function vp_testlib from library 2
vp_testlib_2 = dlsym(lib_handle_2, "vp_testlib");
if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    return 1;
}

// Call vp_testlib functions from libraries 1 and 2
printf("Result 1: %d\n", vp_testlib_1(10));
printf("Result 2: %d\n", vp_testlib_2(5));
```

```
    // Unload libraries

    dlclose(lib_handle_1);

    dlclose(lib_handle_2);



    return 0;

}
```

lukkuz1@oslinux ~/lab3_4/c_uzduotys $ gcc -o main lukkuz1_libtest02.c -ldllukkuz1@oslinux ~/lab3_4/c_uzduotys $ ./main

Result 1: 20

Result 2: 2

lukkuz1@oslinux ~/lab3_4/c_uzduotys $

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

- Sukurtų programų tekstus įkelkite į Moodle.

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



#include "lukkuz1_testlib02.h"


int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Reikia nurodyti 2 argumentus: bibliotekos pavadinimą ir skaičių\n");
        return 1;
    }


    // Įkeliame biblioteką
    void *lib_handle = dlopen(argv[1], RTLD_LAZY);
    if (!lib_handle) {
```

```
        printf("Klaida: nepavyko užkrauti bibliotekos: %s\n", dlerror());
        return 1;
    }

    // Pasiimame funkcijos rodyklę
    int (*vp_testlib)(int) = dlsym(lib_handle, "vp_testlib");
    const char *dlsym_error = dlerror();
    if (dlsym_error) {
        printf("Klaida: nepavyko rasti funkcijos: %s\n", dlsym_error);
        dlclose(lib_handle);
        return 1;
    }

    // Iškviečiame funkciją ir spausdiname rezultatą
    int arg = atoi(argv[2]);
    int result = vp_testlib(arg);
    printf("vp_testlib(%d) = %d\n", arg, result);

    // Atlaisviname bibliotekos resursus
    dlclose(lib_handle);

    return 0;
}
```