Linux cheatsheet

– Párhuzamos és eseményvezérelt programozás beágyazott rendszereken tárgyhoz –

Jelen írás célja mindazon Linux programozási eszköz és technológia rövid és közérthető módon történő bemutatása példákon keresztül, ami a tárgyból előkerül vagy említésre kerül. Az alapismeretek bemutatásán felül pusztán az érdekesség kedvéért több modernebb, nagyobb komplexitású példa is található az írásban, külön jelölve*. Linux programozásban kevésbé járatos olvasó számára a következő olvasási sorrendet ajánlom: C/C++ fordítás, Fájlkezelés, A `select()` hívás, Szálak, Speciális terminálbeállítások. Nem tartozik szorosan a tárgyhoz, de nagy segítség a fejlesztés során, ha grafikus felülettel áll rendelkezésre egy hibakereső, ezért nagyon ajánlom a fejlesztés során valamilyen fejlesztőeszköz használatát (pl.: Hibakeresés Visual Studio Code segítségével). A példakódok nagy része be van ágyazva a PDF-fájlba, a példák mellett a fájlokra mutató hivatkozások találhatók.

Ha a Linux egy beépített függvényéről szeretnél több információt megtudni, bátran használd a Linux `man` segédprogramját, a Linux `manpage`-et!

Tartalomjegyzék

1. C/C++ fordítás **2**
- 1.1. Fordítás közvetlenül terminálból . . . 2
- 1.2. Fordítás `Makefile` segítségével* . . . 2
- 1.3. Fordítás `CMake` segítségével* . . . 3

2. C/C++ fejlesztőkörnyezetek* **4**
- 2.1. Hibakeresés Visual Studio Code segítségével . . . 4

3. Fájlkezelés **7**
- 3.1. Egyszerű fájlkezelés . . . 7
- 3.2. IO-fájlkezelés . . . 9
- 3.3. A `select()` rendszerhívás . . . 10
- 3.4. A `poll()` rendszerhívás* . . . 12

4. Szálak **14**
- 4.1. Szálak életciklusa . . . 14
- 4.2. Szálak szinkronizációja . . . 16

5. Folyamatok **17**

6. Kommunikáció a szálak között **18**
- 6.1. Csővezetékek . . . 18
  - 6.1.1. Névtelen csővezetékek . . . 18
  - 6.1.2. Megnevezett csővezetékek . . . 19

1. C/C++ fordítás

A legegyszerűbb módja Linuxon a C/C++ fordításnak a GNU Compiler Collection avagy `gcc` (C++ esetén `g++`) használatával lehetséges. A `gcc` meghívása egyszerű projektek fordítása esetén történhet közvetlenül terminálból egy kézzel összeállított paranccsal, de komplexebb felépítésű projektek esetén fordítást segítő eszközök is igénybe vehetők, ilyen például a `Makefile` vagy a `CMake`, illetve a `ninja`.

Ha valamilyen oknál fogva nem áll rendelkezésre `gcc` az aktuális rendszeren, akkor ajánlott a `build-essential` csomag telepítésével a `gcc`-t letölteni és nem külön telepíteni, mert így rengeteg egyéb hasznos C/C++ fejlesztést segítő eszköz is szintén telepítésre kerül.

1.1. Fordítás közvetlenül terminálból

Legegyszerűbben a következőképp tudunk egyetlen fájlból álló C projekteket fordítani közvetlenül terminálból:

```
gcc input.c -o program
```

A fenti sor szerint a `gcc` bemeneti fájlként értelmezi az `input.c` fájlt, a keletkezett, lefordított, binárist pedig `program` néven menti a `-o` kapcsoló hatására. A fenti parancs több fájlból álló projektekre a következőképp általánosítható:

```
gcc input_1.c input_2.c -o program
```

Ebben az esetben a felsorolt összes forrásfájl a belefordíttatik a kimenetként megjelölt binárisba.

1.2. Fordítás `Makefile` segítségével*

`Makefile` segítségével a komplexebb, bonyolultabb függőségi viszonyokat (pl.: A modulhoz szükség van B fordítására, B-hez pedig C fordítására) felvonultató projektek is könnyen fordíthatók. A `Makefile` törzse ennek megfelelően *cél-függőség-recept* blokkokból áll, a következőképp:

```
<cél>: <függőség1> <függőség2> <...> <függőségN>
    <recept>
```

Általában C/C++ fordítás esetén a `célok` tárgykódok, tehát például egy `input.c` forrásfájlból `input.o` tárgykód keletkezik a fordítás során, majd később az így keletkezett tárgykódok lesznek egy binárisba linkelve. A `receptek` sora *egy TAB-karakterrel* kell legyen indentálva alapértelmezetten, tehát szóközök sorozata nem megfelelő![1]

A `Makefile` egyik további nagyon hatékony funkciója, hogy lehetővé teszi változók definiálást, ezzel leegyszerűsítve a modularizálhatóvá téve a `Makefile` blokkjaiban szereplő `recepteket`. `Makefile` változókat a Linux shell-ből ismerős módon lehetséges definiálni: `változó=érték`, a változók elérése szintén a Linux shellhez hasonlóan történik: `$(változó)`. Érdemes változóban tárolni a fordítóprogramot és a minden fordításkor alkalmazott kapcsolókat. A kommenteket # vezeti be.

Egy egyszerű példa `Makefile`-ra egy két fájlból (`input_1.c` és `input_2.c`) álló projekt esetén alább látható. A `gcc -c` kapcsoló hatására a csak a programok tárgykóddá történő fordítása történik meg *linkelés nélkül*, hogy az majd a legvégső fázisban, mikor már minden tárgykód készen áll történjen csak meg, lehetővé téve minden fájlokon átívelő referencia feloldását (pl.: `extern`-nel megjelölt változók).

```
CC=gcc # fordítóprogram
OPTS=-O0 -ggdb # ne történjen optimalizáció; legyen hibakersést segítő információk a tárgykódokban a GDB számára
COMPILE=$(CC) $(OPTS) # fordítási parancs
OUTPUT_NAME=program # kimeneti bináris neve

all: input_1.o input_1.o # alapértelmezett cél
    $(COMPILE) input_1.o input_2.o -o $(OUTPUT_NAME)

input_1.o: input_1.c # input_1.c fordítása
```

[1] A recepteket bevezető „separator” változtatható, ha szükséges, a következőképpen: `.RECIPEPREFIX=<bevezető karakter>`

```
10      $(COMPILE) -c input_1.c -o input_1.o
11
12  input_2.o: input_2.c # input_2.c fordítása
13      $(COMPILE) -c input_2.c -o input_2.o
14
```

A fordítás a `make` parancs segítségével indítható, mely alapértelmezetten az aktuális mappában `Makefile` néven keresi a parancsokat tartalmazó fájlt, ezért igen célszerű ezt az elnevezést használni minden esetben. Amennyiben van `all` – úgynevezett alapértelmezett – fordítási cél, akkor még a fordítási célt sem kell megadni és a fordítás egyszerűen így indítható:

```
1  make
```

A `input_N.c` → `input_N.o` fordítási receptek is automatizálhatók a következőképpen:

```
1  %.o: %.c # minden illeszkedő fájl fordítása történjen ilyen módon
2      $(COMPILE) -c $< -o $@
```

ahol `$<` az első függőség (jelen esetben `%.c`), azaz a bementi fájl, `$@` pedig a cél, a kimeneti fájl neve.

Szokás még definiálni még úgynevezett álcélokat, melyeknek nincsenek függőségei, de alapértelmezetten mégsem futnak le, csak külön `make <cél>` hívás hatására. Célszerű álcélként definiálni a fordításkor keletkező fájlok törlésére szolgáló `clean` célt:

```
1  clean:
2      rm *.o $(OUTPUT_NAME) # tárgykódok és linkelt bináris törlése
```

Az álcélokat a fájlban külön jelölni kell a következőképp: `.PHONY: <álcél1> <álcél2> <...> <álcélN>`

Továbbá, hogy a fordítás után is átlátható maradjon a projekt gyökérkönyvtára, célszerű egy könyvtárat meghatározni, ahova a fordítás során keletkezett fájlok fognak kerülni (pl.: `build`).

A korábbi példa immáron teljessé kiegészítve:

```
1  CC=gcc # fordítóprogram
2  OPTS=-O0 -ggdb # ne történjen optimalizáció; legyen hibakersést segítő információk a tárgykódokban a GDB számára
3  COMPILE=$(CC) $(OPTS) # fordítási parancs
4  OUTPUT_NAME=program # kimeneti bináris neve
5
6  all: input_1.o input_1.o # alapértelmezett cél
7      $(COMPILE) input_1.o input_2.o -o $(OUTPUT_NAME)
8
9  %.o: %.c # minden illeszkedő fájl fordítása történjen ilyen módon
10     $(COMPILE) -c $< -o $@
11
12 clean: # tárgykódok és linkelt bináris törlése
13     rm *.o $(OUTPUT_NAME)
14
15 .PHONY: clean
```

1.3. Fordítás `CMake` segítségével*

Nagy és bonyolult projektek fordításához elengedhetetlen nem csak a fordítást segítő, de a projektstruktúrát könnyen átláthatóvá tevő rendszerek használata, amikkel a projektek fordítási beállításai globálisan, illetve specifikusan, fájlonként is könnyen konfigurálhatók. Sok ilyen rendszer létezik, nem ritkán maga a fejlesztőkörnyezet rendelkezik ilyen funkcióval (pl.: Visual Studio). Ilyen az igen széles körben elterjedt multiplatform CMake.

A fordítási beállítások az úgynevezett `CMakeLists`-ben vannak letárolva, általában a `CMakeLists.txt` nevű fájlban a projekt gyökérkönyvtárában.

A korábbi két fájlból álló projekt `CMakeLists`-je az alábbi módon néz ki:

```
cmake_minimum_required(VERSION 3.10) # minimális CMake-verzió
project(example) # projektnév

# fordítandó fájlok
set(sources_to_compile input_1.c input_2.c)

# fordítási kapcsolók
set(CMAKE_C_FLAGS "-Wall")

# kimenet-fájl összerendelés, kimenet meghatározása
add_executable(program ${sources_to_compile})
```

A beállításoknak tartalmaznia kell a CMake-környezet legalacsonyabb megkövetelt verzióját, mely a fordításhoz szükséges (`cmake_minimum_required(VERSION ...)`), a projekt nevét (`project(...)`), illetve egy sort, mely meghatározza a kimenetet és a hozzátartozó fájlokat, melyeket a kimenet létrehozásához le kell fordítani (a példában futtatható fájl a kimenet: `add_executable(<projektnév> <fájlok>)`).

A sor-kommenteket # vezeti be. Változókat (mind a rendszer, mind a felhasználó által meghatározott) a `set(<változónév> <tartalom>)` paranccsal állíthatunk be. A változót elérni később a `${változónév}` szintaxissal lehetséges. A nyelv kisbetű-nagybetű érzéketlen!

CMake-ben is természetesen lehetőségünk van mind a C, mind a C++ fordító számára parancssori kapcsolókat átadni, a `CMAKE_C_FLAGS` és a `CMAKE_CXX_FLAGS` változókon keresztül (pl.: `set(CMAKE_CXX_FLAGS "-Wall")`).

A fordítás legegyszerűbben a

```
cmake .
cmake --build .
```

parancspárossal lehetséges. Valójában a `cmake` fordítás előtt a `CMakeLists.txt`-t értelmezi és valamilyen, alacsonyabb szintű strukturált fordításvezérlő scriptet generál, majd a meghívja a scripthez tartozó programot. Egyik leggyakoribb megoldás, hogy a `cmake` `Makefile`-t generál, majd meghívja a `make`-et.

Jól látható, hogy a `cmake` használata sokkal kényelmesebb a `Makefile`-hoz képest, azonban vigyázat, jóval komolyabb előkövetelmény is tartozik az egyszerűbb használathoz: a `cmake`-nek telepítve kell lennie az adott rendszerre! Egyébként a `cmake` nem csak Linuxon, hanem Windows operációs rendszerre is elérhető, sőt például a Visual Studio egy ideje már rendelkezik `cmake`-integrációval.

2. C/C++ fejlesztőkörnyzetek*

Linuxon több, jól használható, modern fejlesztőrendszer áll rendelkezésre C/C++ fejlesztésre. A szerencsére igen soktagú listából kettőt emelnék ki: a JetBrains cég CLion termékét és a Visual Studio Code testreszabható fejlesztőkörnyzetet.

A CLion az IntelliJ C/C++ fejlesztésre átalakított változata mely felvonultatja az említett fejlesztőkörnyezet minden, C/C++ világban is értelmezhető eszközét, illetve C/C++ specifikusokkal egészíti ki. Többek között rendelkezik `valgrind`-integrációval, ami nagyban megkönnyíti a memóriakezelési hibák felderítését. A CLion `cmake`-projekteken tud dolgozni. Komolyabb projektek esetén ajánlom használatát, mert egy nagyon hatékony eszköz.

A továbbiakban az open source Visual Studio Code használatával fogunk foglalkozni.

2.1. Hibakeresés Visual Studio Code segítségével

Visual Studio Code segítségével, akár bővítmények nélkül is automatizálni tudjuk a fordítás és a hibakereséssel együtt történő indítás folyamatát. Ehhez meg kell adnunk a környezet számára, hogy milyen fordítási és milyen hibakeresési feladatok állnak rendelkezésre a projektkörnyezetben. Minden projektspecifikus beállítást a VS Code a projektkönyvtárban elhelyezett `.vscode` mappában keres.

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "shell",
            "label": "Build project",
            "command": "make",
            "problemMatcher": [
                "$gcc"
            ],
        },
    ]
}
```

tasks.json

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug project",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/program",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "preLaunchTask": "Build project"
        }
    ]
}
```

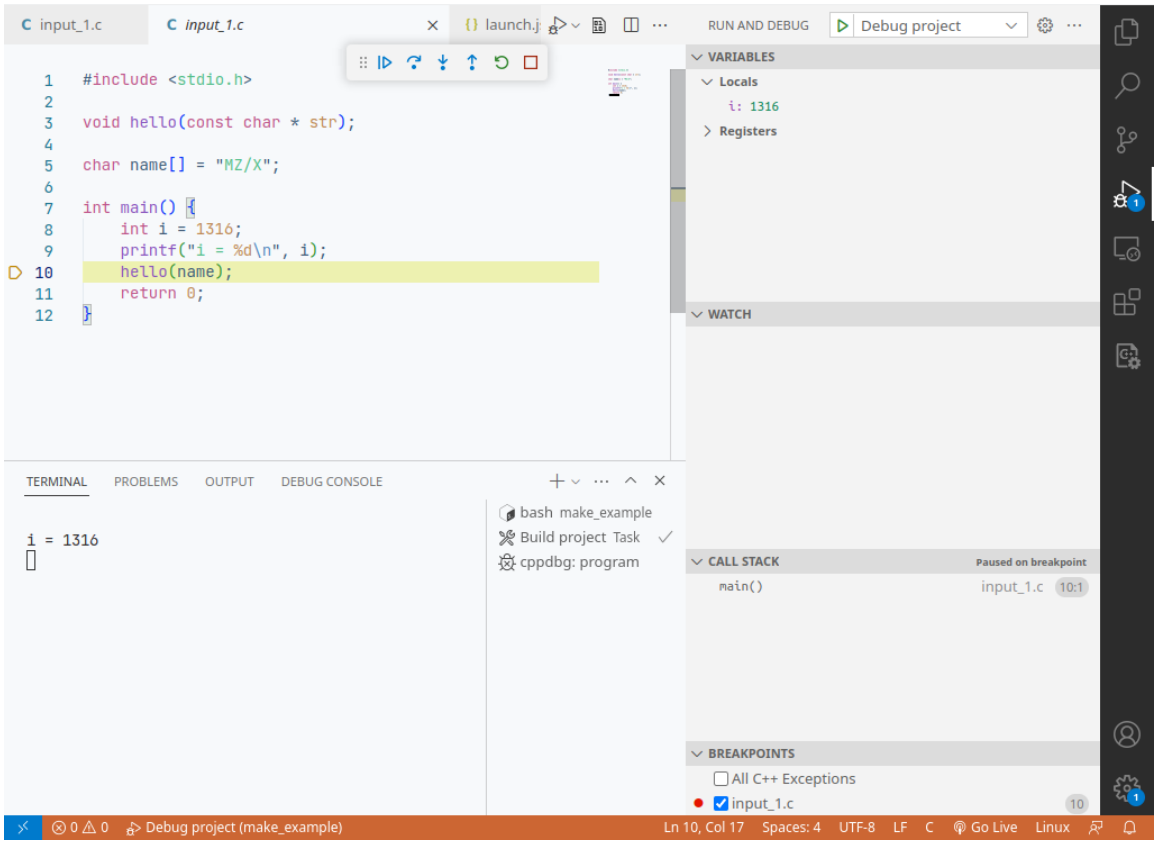
launch.json

A fordítási feladatokat az ezen könyvtárban elhelyezett `tasks.json` fájlban lehetséges megadni. A `tasks` tömb tartalmazza a feladatokat. Minden `task` rendelkezik egy `type` mezővel, ami azt adja meg a környezet számára, hogy milyen módon kell a `command` mezőben megadott parancsot értelmezze. A `label` mezőben kell megadni a feladat nevét, amivel később tudunk hivatkozni az adott feladatra; lehet benne whitespace. A `problemMatcher` tömb azt tartalmazza, hogy milyen módon dolgozza fel a VS Code a keletkező hibaüzeneteket a konzolra történő, strukturált kiíratáshoz. Mivel `gcc`-t fogunk használni, ezért a tömb egyetlen elemének a `"$gcc"` értéket adjuk.

A hibakeresési, futtatási feladatokat a `launch.json` fájlban kell elhelyezni a környezet számára. A futtatási/-hibakeresési beállításokat a `configurations` tömb tartalmazza. A `name` mező, a beállítás nevét kell, hogy tartalmazza; lehet benne whitespace. A `type` mezőben kell megadnunk az adott futtatás-konfiguráció típusát, jelen esetben `cppdgb` értékkel. A hibakereső képes általa indított, vagy már futó folyamaton is működni, ennek meghatározására szolgál a `request` mező, ami a mi esetünkben a `launch` értéket veszi fel. A futtatandó programot *abszolút elérési útvonallal* a `program` mezőben kell megadni. (Ennek az értékét mi határoztuk meg a `Makefile`-ban.) A program számára további parancssori paraméterek adhatók át az `args` tömbben. Ha a `stopAtEntry` értéke `true`, akkor a program belépési pontján rögtön megállítja a hibakereső a programot. A `cwd` mező a futtatás során alkalmazott munkakönyvtárat tartalmazza, a mi esetünkben célszerű a projekt gyökérkönyvtárára állítanunk. Ha a futtatás során nem a VS Code beépített terminálját szeretnénk használni, akkor ezt az `externalConsole` igazra állításával megtehetjük. Fontos még számunkra a `preLaunchTask` mező, hisz ennek segítségével automatikusan, minden futtatás előtt indítható fordítás. A mező értéke a korábban a `tasks.json`-ban a fordítási konfiguráció `label` mezőjében megadott `Build project` kell, hogy legyen ehhez.

A hibakeresés csak akkor fog működni, ha van is a binárisban hibakeresési információ, azaz `-ggdb` kapcsolóval lett fordítva! Célszerű letiltani az optimalizációt (`-O0` kapcsoló), hogy tényleg szekvenciális végrehajtást lássunk és ne kelljen kitalálni, hogy az optimalizáció során a fordító mit változtatott a kódon, és miért úgy működik az adott program.

Ha mindent jól előkészítettünk, akkor a töréspontok elhelyezése után egy az alábbihoz hasonló, interaktív környezet kell elénk táruljon, miután futtattuk lefordítottuk és elindítottuk az alkalmazásunkat:


input_1.c
input_1.c
launch.j
RUN AND DEBUG
Debug project
1 #include <stdio.h>
2
3 void hello(const char * str);
4
5 char name[] = "MZ/X";
6
7 int main() {
8 int i = 1316;
9 printf("i = %d\n", i);
10 hello(name);
11 return 0;
12 }
VARIABLES
Locals
i: 1316
Registers
WATCH
TERMINAL
PROBLEMS
OUTPUT
DEBUG CONSOLE
i = 1316
bash make_example
Build project Task
cppdbg: program
CALL STACK
Paused on breakpoint
main()
input_1.c 10:1
BREAKPOINTS
All C++ Exceptions
input_1.c 10
0 0
Debug project (make_example)
Ln 10, Col 17
Spaces: 4
UTF-8
LF
C
Go Live
Linux

3. Fájlkezelés

Linux alatt nem csak a az adatállományok, de sok minden más rendszereszköz is a fájlrendszerbe leképezve, „fájlként” jelenik meg. Alacsony szinten Linuxban minden, az alkalmazás által megnyitott fájlt egy egész szám, úgynevezett `File Descriptor` (`FD`) azonosít. Az első három FD-nak kitüntetett szerepe van: a nullás a standard input, az egyes a standard output és a kettes a standard error FD-a; a felsorolt három FD mindig „nyitva van”.

3.1. Egyszerű fájlkezelés

További fájl descriptorok nyithatók például az `open()` hívással. Az így kapott FD-okon pl.: a `read()`, `write()`, `close()` rendszerhívásokkal végezhetünk műveleteket. A pontos paraméterekről a `manpage`-ekben találhatunk részletes leírást: pl.: `man 2 read` a `read()` használatáról szolgáltat információkat. Fontos megjegyezni, hogy egy adott állományra célszerűen csak egy fájlkezelési szintet szabad elindítani: azaz például, ha a standard outputra a FD-án keresztül `write()` használatával és magasabb szintről `printf` függvényhívások segítségével is, akkor nem garantálható, hogy a kimeneten valóban a hívási sorrendben jelennek meg az adatok.

Az alábbi rövid programrészlet a fájlkezelés alapjait mutatja be: `fd_example.c`

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

const char *lorem = "Lorem ipsum dolor sit amet, consectetur adipiscing elit,\
                     sed do eiusmod tempor incididunt ut labore et dolore magna\
                     aliqua. Ut enim ad minim veniam, quis nostrud exercitation\
                     ullamco laboris nisi ut aliquip ex ea commodo consequat.";

const unsigned N = 40;

void file_error() {
    write(STDERR_FILENO, "File error!", 11);
}

int main() {
    // fájl megnyitása írásra (O_WRONLY), ha még nem létezik, hozd létre (O_CREAT)
    int fd = open("lorem.txt", O_CREAT | O_WRONLY);
    if (fd == -1) { // hibaellenőrzés,
        // ha gond van, üzenet kiírása a standard outputra
        file_error();
    }

    // írjuk ki az első N karaktert a megnyitott fájlba
    write(fd, lorem, N);

    // zárjk le a fájlt
    close(fd);

    // -----------------

    // nyissuk meg a fájlt, amibe kiírtuk előző az szöveg első N karakterét
    fd = open("lorem.txt", O_RDONLY);
    if (fd == -1) { // hibaellenőrzés,
        // ha gond van, üzenet kiírása a standard outputra
        file_error();
    }

    // ugorjuk át az első 10 karaktert
    lseek(fd, 10, SEEK_SET);

    // olvassunk max. N karaktert
    char buf[N];
    ssize_t n_read = read(fd, buf, N);
    if (n_read == -1) {
        file_error();
    }

    // zárjuk be a fájlt
    close(fd);

    // írassuk ki a beolvasott szöveget a standard outputra
    write(STDOUT_FILENO, buf, n_read);

    return 0;
}
```

3.2. IO-fájlkezelés

Linux alatt egyes kommunikációs perifériák is (pl.: UART perifériák, soros portok stb.) is fájlként érhetők el. Természetesen az adatállományoktól eltérően sokszor szükség van az IO-perifériák különböző beállításainak módosítására is használat előtt.

Linux alatt például soros portok beállításainak módosítására jól használható a `termios` eszköz (`man termios`). Alább egy egyszerű példa látható, ami megnyitja a `/dev/ttyUSB0` virtuális soros portot és beállítja annak sebességét 115200bps-ra 8-bites keretméret alkalmazása mellett. `termios_example.c`

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <string.h>

const char str[] = "Eum quia reprehenderit\
                    rerum earum minima possimus aut in.";

int main() {
    // termios struktúra létrehozása és nullázása
    struct termios serial;
    memset(&serial, 0, sizeof(struct termios));

    // struktúra kitöltése
    serial.c_cflag = CS8 | CREAD | CLOCAL; // 8-bites keretméret, vétel engedélyezése, modem control tiltása
    serial.c_cc[VMIN] = 1; // karakterenkénti olvasás engedélyezése
    serial.c_cc[VTIME] = 5; // nem-kanonikus olvasás időlimitje tizedmásodpercben
    cfsetospeed(&serial, B115200); // adó sebességének beállítása
    cfsetispeed(&serial, B115200); // vevő sebességének beállítása

    // soros port megnyitása írásra és olvasásra
    int serial_fd = open("/dev/ttyUSB0", O_RDWR);
    if (serial_fd == -1) { // hibakezelés, üzenet kiírása, ha szükséges, az STDERR-re
        write(STDERR_FILENO, "Could not open serial port!", 27);
    }

    // korábban egybegyűjtött beállítások alkalmazása
    tcsetattr(serial_fd, TCSANOW, &serial); // TCSANOW = "alkalmazás azonnal"

    // adat küldése a soros porton
    write(serial_fd, str, strlen(str));

    // egy karakter visszaolvasása
    char c;
    read(serial_fd, &c, 1);

    // karakter kiírása a standard outputra
    write(STDOUT_FILENO, &c, 1);

    // soros port bezására
    close(serial_fd);

    return 0;
}
```

Fontos megjegyezni, hogy maga az IO-eszköz megnyitása, ahogy a korábbi példában az adatállomány-fájl esetében is, egy egyszerű `open()` rendszerhívással történik. A `termios` beállításai elkülönülnek (`serial` objektum), és csak később, a már megnyitott FD-ra alkalmazva jutnak érvényre a `tcsetattr()` hívás hatására.

Előfordulhat, hogy a programnak nincs joga hozzáférni az adott soros porthoz, ezen egyszerűen lehet segíteni,

a felhasználót hozzá kell adni a `dialout` csoporthoz és egy kijelentkezés-bejelentkezés után már lesz jogosultsága – és ezáltal az általa indított alkalmazásoknak is – megnyitni a (virtuális) soros portot:

```
sudo adduser $USER dialout
```

Soros portok egyszerű írására és olvasására célszerű terminál program a `GTKTerm`.

3.3. A `select()` rendszerhívás

A korábbi példában a program visszatérése előtt egy karakter megérkezésére várunk a soros port felől, másként fogalmazva, a soros porton történő adatfogadás *eseményére* várakozunk.

Amíg pusztán egy FD-t kell figyelnünk – mivel csak egy FD-on várunk valamilyen eseményre –, addig nagyon egyszerű dolgunk van, hiszen egy egyszerű `read()`-hívás ezt a célt pontosan kielégíti.

Ha több eseményforrásra kell figyelnünk – azaz több, különböző FD-on érkező eseményre kell szimultán várakoznunk –, akkor a puszta `read()`-hívásokkal megvalósítható megoldás már nem fog jól működni, hiszen egy, a szekvenciális végrehajtás szerint korábban álló blokkoló `read()` hívás miatt hiába tudna olvasni egy később álló `read()`, nem jut oda a végrehajtás. A párhuzamos várakozás problémájára kínál egyszerű megoldást a `select()` rendszerhívás (`man select`).

A `select()` hívás öt paramétert vár, mielőtt meghívnánk a függvényt, rendelkeznünk kell az összes FD-ral, amiken eseményekre szeretnénk várakozni.

- ☐ `int nfds`: „a FD-ok közül a legnagyobb értékű + 1”. A rendszer a FD-okat szekvenciálisan, növekvő sorrendben osztja ki biztosan amíg nem zárunk be egy fájlt sem, mivel az „első”, standard input FD-a nulla, ezért a magyarázat elején álló összefüggés épp a FD-ok számát adja; innen a paraméter elnevezése, és ebből a paraméterből tudja a `select()`, hogy *(legfeljebb)* mennyi FD-t kell figyeljen.

- ☐ `fd_set * readfds`: azon FD-ok halmaza, amiken elérhető beérkező adatra kívánunk várakozni. A halmazok meghatározása az `FD_ZERO()`, `FD_SET()` hívásokkal lehetséges egy korábban létrehozott `fd_set` objektum segítségével.

  ```
  fd_set rfds;
  FD_ZERO(&rfds); // struktúra nullázása; létező fd_set-re mutató pointer átadása
  FD_SET(STDIN_FILENO, &rfds); // STDIN hozzáadása a halmazhoz
  FD_SET(serial_fd, &rfds); // korábban megnyitott soros port hozzáadása a halmazhoz
  ```

  Ha nem kívánunk ilyen halmazt megadni, akkor egy üres halmaz helyett átadhatunk `NULL`-t.

- ☐ `fd_set * writefds`: hasonló értelmű mint a korábbi `readfds`, csak írásra.

- ☐ `fd_set * exceptfds`: hasonló értelmű mint a korábbi `readfds`, speciális eseményekre várakozhatunk vele.

- ☐ `struct timeval * timeout`: ezen paraméter segítségével időlimitet tudunk meghatározni a várakozás számára egy korábban kitöltött `struct timeval` objektum címének átadásával. A struktúrának kettő mezője van az alábbiak szerint:

  ```
  struct timeval tv;
  tv.tv_sec = 1; // timeout másodperc részének beállítása
  tv.tv_usec = 200; // timeout mikrosecundum részének beállítása
  ```

  Ezen paraméter `NULL` értékűre állítása letiltja a timeout-funkcionalitást, a hívás addig blokkol, amíg nem következik be legalább egy esemény a vizsgált FD-okon.

A `select()` akkor fog visszatérni, ha olyan esemény következik be, amelyre várakoztunk, vagy, ha letelt a timeout, vagy, ha valamilyen belső hiba történt. Hiba esetén a visszatérési érték -1, egyébként pedig a teljesült várakozási feltételek – bekövetkezett események – száma: ha valamilyen esemény bekövetkezett akkor legalább 1, ha letelt a timeout, akkor 0.

Visszatéréskor a `select()` a FD-halmazokba írja vissza, hogy mely FD-okon történt esemény – más szóval: a halmazban csak azok a FD-ok maradnak benne, amiken események következtek be. A tartalmazást az `FD_ISSET()` hívással lehetséges vizsgálni, mely egy logikai értelmű értéket ad vissza:

```
FD_ISSET(STDIN_FILENO, &rfds)...
```

Mivel a `select()` visszatérésekor minden pointerként átadott paramétere módosulhat, ezért minden egyes `select()`-hívás előtt azok kezdeti értékét vissza kell állítani.

A következőkben lássunk egy példát, mely a standard inputról beolvasott adatokat a soros porta, az onnan beolvasott adatokat pedig a standard output-ra továbbítja! `select_example.c`

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <string.h>
#include <stdbool.h>
#include <sys/time.h>

int main() {
    // termios struktúra létrehozása és nullázása
    struct termios serial;
    memset(&serial, 0, sizeof(struct termios));

    // struktúra kitöltése
    serial.c_cflag = CS8 | CREAD | CLOCAL; // 8-bites keretméret, vétel engedélyezése, modem control tiltása
    serial.c_cc[VMIN] = 1; // karakterenkénti olvasás engedélyezése
    serial.c_cc[VTIME] = 5; // nem-kanonikus olvasás időlimitje tizedmásodpercben
    cfsetospeed(&serial, B115200); // adó sebességének beállítása
    cfsetispeed(&serial, B115200); // vevő sebességének beállítása

    // soros port megnyitása írásra és olvasásra
    int serial_fd = open("/dev/ttyUSB0", O_RDWR);
    if (serial_fd == -1) { // hibakezelés, üzenet kiírása, ha szükséges, az STDERR-re
        write(STDERR_FILENO, "Could not open serial port!", 27);
    }

    // korábban egybegyűjtött beállítások alkalmazása
    tcsetattr(serial_fd, TCSANOW, &serial); // TCSANOW = "alkalmazás azonnal"

    bool exit_loop = false; // program bezárása, ha true
#define BUFLEN (32) // buffer hossza
    char linebuf[BUFLEN]; // sorbuffer

    while (!exit_loop) {
        // olvasási FD-halmaz létrehozása és kitöltése
        fd_set rfds;
        FD_ZERO(&rfds); // halmaz törlése, nincs memóriaszemét-újrahasznosítás :)
        FD_SET(STDIN_FILENO, &rfds); // STDIN hozzáadása
        FD_SET(serial_fd, &rfds); // soros port hozzáadása

        // időintervalum beállítása
        struct timeval tv;
        tv.tv_sec = 3; // 3 másodperc
        tv.tv_usec = 0; // 0 us

        // select hívás: a legnagyobb értékű FD biztosan a serial_fd
        int ret = select(serial_fd + 1, &rfds, NULL, NULL, &tv);
        if (ret == -1) { // hiba történt
            write(STDOUT_FILENO, "Error!\n", 6);
```

```
50              exit_loop = true;
51          } else if (ret > 0) { // bekövetkezett legalább egy esemény, amire vártunk
52              if (FD_ISSET(STDIN_FILENO, &rfds)) { // adat van a standard inputon
53                  // max. BUFLEN mennyiségű karakter olvasása az STDIN-ről
54                  ssize_t n = read(STDIN_FILENO, linebuf, BUFLEN);
55                  /* hibakezeléstől most eltekintünk */
56                  if ((n == 1) && (linebuf[0] == 'x')) { // x -> kilépés
57                      exit_loop = true;
58                  } else { // nem akarunk kilépni
59                      write(serial_fd, linebuf, n); // n karakter írása a soros portra
60                  }
61              }
62              if (FD_ISSET(serial_fd, &rfds)) { // bejövő adat van a soros porton
63                  ssize_t n = read(serial_fd, linebuf, BUFLEN); // ...
64                  write(STDOUT_FILENO, linebuf, n); // ...
65              }
66          } else { // letelt a timeout
67              write(STDOUT_FILENO, "Timeout\n", 9);
68          }
69      }
70
71      // soros port bezására
72      close(serial_fd);
73
74      return 0;
75  }
```

3.4. A `poll()` rendszerhívás*

Az idők során megszületett az igény egy korszerűbb, jobban kezelhető és nem mellesleg a `select()` nehézségeit nem hordozó, hiányosságait pótló `select()`-hez hasonló rendszerhívás megalkotására. Így született meg a `select()`-hez képest bővebb funkcionalitást nyújtó `poll()` rendszerhívás (`man poll`).

A függvény három paraméterrel rendelkezik:

- ☐ `struct pollfd * fds`: A figyelendő FD-ok *tömbje* – a tömb első elemére mutató pointer –, aminek elemei a következő struktúrával rendelkeznek:

  ```
  1  struct pollfd {
  2      int   fd;         /* file descriptor */
  3      short events;     /* requested events */
  4      short revents;    /* returned events */
  5  };
  ```

  A struktúra `fd` mezője tartalmazza a figyelendő FD értékét. Az `events` bitmező értéke határozza meg, milyen eseményekre várakozunk, lényegében itt váltjuk ki a `select()` `readfds`, `writefds`, `exceptfds` paramétereit és bővítjük ki további funkcionalitással. Az `revents` mezőt később értelmezzük.

  Az `events` bitmező számunkra fontos (rész)értékei a következők:

  - `POLLIN`: beérkező, olvasásra kész adatra várunk; ezen részértékkel valósítható meg, amit a `select()` hívásban `readfds` kitöltésével tudtunk elérni.
  - `POLLOUT`: a FD-ra lehet írni; a `select()` `writefds` paraméterének funkcióját valósítja meg.
  - `POLLRDHUP`: a stream socketet a másik fél a maga részéről (legalább „félig”) lezárta

  A `poll()` a negatív FD-értékeket figyelmen kívül hagyja, ezért igény szerint az `fds` tömb elemeiben pusztán az FD negálásával (értelem szerűen a 0-s FD kivételével), anélkül, hogy a tömböt újraépítenénk, a vizsgálatok a kiválasztott FD-ra tilthatók, kihasználva, hogy az FD-ok előjeles számként (`int`) vannak tárolva.

- `nfds_t nfds`: Az `fds` tömb mérete.
- `int timeout`: Maximális várakozási idő *ezredmásodpercben*. Negatív érték esetén a hívás addig blokkol, amíg nem következik be valamelyik esemény, amire várakozunk vagy egy, a figyelt FD-okat érintő kivétel.

  A függvény visszatérési értéke a `select()` rendszerhívásával megegyező.

  A visszatérés során a bekövetkezett eseményeket a `poll()` az `fds` tömb elemeinek `revents` mezőibe menti, ami szintén bitmezőként viselkedik. Az `revents` értékeket vehet fel az `event` mezőben bekapcsolt eseményekből, de ezen kívül bizonyos, nem letiltható részértékek is megjelenhetnek benne:

  - `POLLERR`: a FD-on valamilyen hiba történt, kb. az `exceptfds` funkcióját valósítja meg.
  - `POLLNVAL`: érvénytelen kérés, nincs nyitva az alkalmazásban az adott FD
  - `POLLHUP`: jelzi, hogy valamilyen oknál fogva az adott FD-ról nem lehetséges további olvasás; általában félig lezárt kapcsolatot jelez, pl. pipe-ok vagy stream socketek esetén

A korábban bemutatott példában a `select()` hívást `poll()`-ra cserélve a következőképp fest: `poll_example.c`

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <string.h>
#include <stdbool.h>
#include <sys/time.h>
#include <poll.h>

int main() {
    // termios struktúra létrehozása és nullázása
    struct termios serial;
    memset(&serial, 0, sizeof(struct termios));

    // struktúra kitöltése
    serial.c_cflag = CS8 | CREAD | CLOCAL; // 8-bites keretméret, vétel engedélyezése, modem control tiltása
    serial.c_cc[VMIN] = 1; // karakterenkénti olvasás engedélyezése
    serial.c_cc[VTIME] = 5; // nem-kanonikus olvasás időlimitje tizedmásodpercben
    cfsetospeed(&serial, B115200); // adó sebességének beállítása
    cfsetispeed(&serial, B115200); // vevő sebességének beállítása

    // soros port megnyitása írásra és olvasásra
    int serial_fd = open("/dev/ttyUSB0", O_RDWR);
    if (serial_fd == -1) { // hibakezelés, üzenet kiírása, ha szükséges, az STDERR-re
        write(STDERR_FILENO, "Could not open serial port!", 27);
    }

    // korábban egybegyűjtött beállítások alkalmazása
    tcsetattr(serial_fd, TCSANOW, &serial); // TCSANOW = "alkalmazás azonnal"

    bool exit_loop = false; // program bezárása, ha true
#define BUFLEN (32) // buffer hossza
    char linebuf[BUFLEN]; // sorbuffer

    // FD-ok tömbje a poll() számára
#define PFDSLEN (2)
    struct pollfd pfds[PFDSLEN];

    // STDIN hozzáadása
    pfds[0].fd = STDIN_FILENO;
    pfds[0].events = POLLIN; // beérkező adatok figyelése

```

```
43      // soros port hozzáadása
44      pfds[1].fd = serial_fd;
45      pfds[1].events = POLLIN; // beérkező adat figyelése
46
47      // timeout meghatározása
48      int timeout = 3000; // 3 másodperc
49
50      while (!exit_loop) {
51         // poll hívás
52          int ret = poll(pfds, PFDSLEN, timeout);
53          if (ret == -1) { // hiba történt
54              write(STDOUT_FILENO, "Error!\n", 6);
55              exit_loop = true;
56          } else if (ret > 0) { // bekövetkezett legalább egy esemény, amire vártunk
57              if (pfds[0].revents & POLLIN) { // adat van a standard inputon
58                  // max. BUFLEN mennyiségű karakter olvasása az STDIN-ről
59                  ssize_t n = read(STDIN_FILENO, linebuf, BUFLEN);
60                  /* hibakezeléstől most eltekintünk */
61                  if ((n == 1) && (linebuf[0] == 'x')) { // x -> kilépés
62                      exit_loop = true;
63                  } else { // nem akarunk kilépni
64                      write(serial_fd, linebuf, n); // n karakter írása a soros portra
65                  }
66              }
67              if (pfds[1].revents & POLLIN) { // bejövő adat van a soros porton
68                  ssize_t n = read(serial_fd, linebuf, BUFLEN); // ...
69                  write(STDOUT_FILENO, linebuf, n); // ...
70              }
71          } else { // letelt a timeout
72              write(STDOUT_FILENO, "Timeout\n", 9);
73          }
74      }
75
76      // soros port bezására
77      close(serial_fd);
78
79      return 0;
80  }
```

4. Szálak

A Linux ütemezőjének ütemezési alapegysége a *szál*. Egy szál eléri az őt magában foglaló folyamat teljes kontextusát, egy folyamat FD-ai, heap-je közös, a stack értelem szerűen minden egyes szál esetén egyedi, de az egyes szálak elérhetik másik szálak stackjét. Ilyenkor *megfelelően elhelyezett kölcsönös kizárás* alkalmazásával kell védeni a hívó stacket, hogy ne legyen visszagörgetve, amíg a másik szál dolgozik a benne tárolt változókon. Bár az effajta elérés lehetséges, inkább kerülendő, helyette a szálak (és folyamatok) közötti valamely kommunikációs megoldás alkalmazása javallott.

Szálakat létre tudunk hozni, le tudunk állítani és tudunk várakozni a befejeződésükre (többek között).

Minden alkalmazás rendelkezik egy főszállal, amely az alkalmazás belépési pontjául megjelölt rutint futtatja (pl. a `main()`-t).

4.1. Szálak életciklusa

A szálak „*születnek, élnek és az emlékek földjére érnek*”, mégpedig az alábbi módokon:

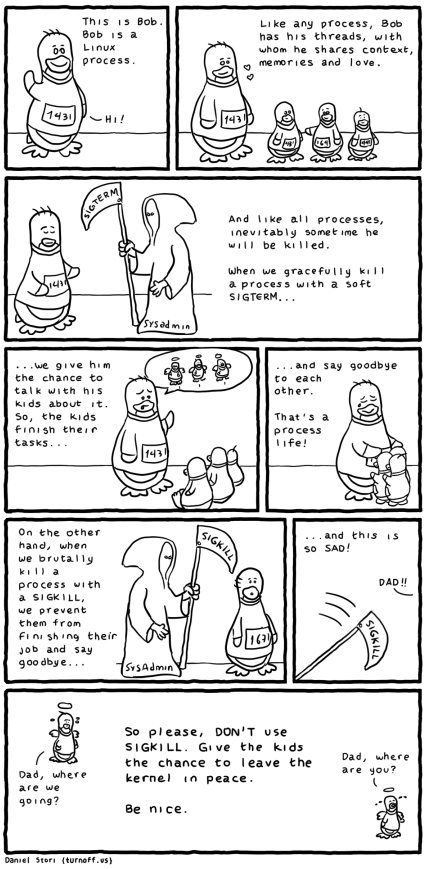
Lérehozás A szálakat a `pthread_create()` rendszerhívással tudunk létrehozni (`man pthread_create`), melynek négy paramétere van:

- ☐ `pthread_t *thread`: Mutató egy már létező `pthread_t` objektumra. Sikeres szálindítás esetén ide lesz letárolva a szál egyedi azonosítója, aminek segítségével később a szálat kezelni tudjuk.
- ☐ `const pthread_attr_t *attr`: A létrehozandó szál attribútumait itt tudjuk meghatározni. Egyelőre ezzel a lehetőséggel nem foglalkozunk, ha a paraméterként `NULL`-t adunk át a szál az alapértelmezett attribútumokkal fog létrejönni.
- ☐ `void *(*start_routine) (void *)`: Az új szálban indítandó „rutinhoz" tartozó függvényre mutató pointer – az új szál függvénypointere. A függvénynek egy (`void *`) típusú paramétere van és visszatérési típusa is `void *`.
- ☐ `void *arg`: A szálban futtatott függvény egyetlen paraméterének értéke itt adható meg.

A rendszerhívás visszatérési értéke hibajelzés értelmű: ha 0-val tér vissza, akkor a szál létrehozása megtörtént, ha ettől eltérő értékkel, akkor az hibakódként értelmezendő.

Egy egyszerű példa két szál létrehozására az alább látható:

```c
#include <stdio.h>
#include <pthread.h>

// SZÁL 1.
void * thread_1(void * arg) {
    printf("[2] Első szál fut!\n");
    return NULL; // szál vége, visszatérés
}

// SZÁL 2.
void * thread_2(void * arg) {
    printf("[2] Második szál fut!\n");
    return NULL; // szál vége, visszatérés
}

// főprogram (főszál)
int main() {
    pthread_t t1, t2; // szálleírók létrehozása
    pthread_create(&t1, NULL, thread_1, NULL); // első szál indítása
    printf("[1] Első szál indítása...\n");
    pthread_create(&t2, NULL, thread_2, NULL); // második szál indítása
    printf("[1] Második szál indítása...\n");

    /* ... */

    return 0;
}
```



A szálak használatához az alkalmazáshoz a `pthread` könyvtárat hozzá kell linkelni (`gcc ... -lpthread`)!

Megszűnés és megszüntetés Egy szál a következő esetekben szűnik meg:

- ☐ ha *visszatér* a `start_routine()`-ból, vagy
- ☐ ha egy `pthread_exit()` hívással *kilép*, vagy
- ☐ ha *leállítják*, vagy
- ☐ ha az őt létrehozó és birtokló folyamatban *bármelyik szál* `exit()`-tel *kilép* vagy a `főszál futása befejeződik` a belépési pontból (`main()`-ből) való visszatéréssel.

4.2. Szálak szinkronizációja

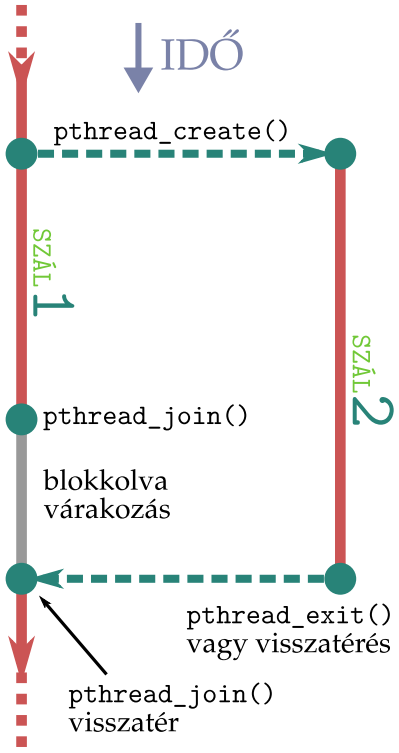
Egymással együttműködő szálak esetén fontos, hogy legyen lehetőség arra, hogy az egyik szál a másik befejeződését bevárja, esetleg a főszál ne térjen vissza – ne álljon le – addig, amíg még egy általa indított szál nem fejezte be a működését. Ezen típusú várakozás, szálak közötti szinkronizáció megvalósítására szolgál a `pthread_join()` rendszerhívás (`man pthread_join`). A `pthread_join()` függvénynek kettő paramétere van: az elsőben azon a szál leíróját kell átadnunk, amire várakozni kívánunk, a másodikban pedig opcionálisan lehetőségünk van egy `void**` pointert átadni, ahova a szál visszatérési értéke kerül. (Ez a paraméter opcionális, ha `NULL`-t adunk át, akkor a szál visszatérési értéke el lesz dobva.)

A korábban vázolt, szálak közötti szinkronizáció folyamatát legegyszerűbben a szálak egymásba csatlakozásként képzelhetjük el. A szakirodalomban ezt „rendezvous”-nak nevezik, a továbbiakban, a korábbi értelmezés szerint a szálak egymásba csatlakozásának fogom nevezni a szinkronizációt.

A korábbi példa teljessé kiegészítve becsatlakozással az alábbi:

`pthread_example.c`

```c
#include <stdio.h>
#include <pthread.h>

// SZÁL 1.
void * thread_1(void * arg) {
    printf("[2] Első szál fut!\n");
    return NULL; // szál vége, visszatérés
}

// SZÁL 2.
void * thread_2(void * arg) {
    printf("[2] Második szál fut!\n");
    return NULL; // szál vége, visszatérés
}

// főprogram (főszál)
int main() {
    pthread_t t1, t2; // szálleírók létrehozása
    pthread_create(&t1, NULL, thread_1, NULL); // első szál indítása
    printf("[1] Első szál indítása...\n");
    pthread_create(&t2, NULL, thread_2, NULL); // második szál indítása
    printf("[1] Második szál indítása...\n");

    printf("[3] Várakozás az első szálra...\n");
    pthread_join(t1, NULL); // várakozás az első szál befejeződésére
    printf("[3] Várakozás a második szálra...\n");
    pthread_join(t2, NULL); // várakozása a második szál befejeződésére

    return 0;
}
```



Szálak egymásba csatlakozása

Első futtatás:

```
[1] Első szál indítása...
[2] Első szál fut!
[1] Második szál indítása...
[3] Várakozás az első szálra...
[2] Második szál fut!
[3] Várakozás a második szálra...
```

Második futtatás:

```
[1] Első szál indítása...
[1] Második szál indítása...
[3] Várakozás az első szálra...
[2] Második szál fut!
[2] Első szál fut!
[3] Várakozás a második szálra...
```

Harmadik futtatás:

```
[1] Első szál indítása...
[2] Első szál fut!
[1] Második szál indítása...
[3] Várakozás az első szálra...
[3] Várakozás a második szálra...
[2] Második szál fut!
```

Az ütemezés megnyilvánulása a szálak futási sorrendjében

Az ütemezés megnyilvánulása A fenti példát többször futtatva a jobb oldalt látható, eltérő eredményeket kapjuk a terminálon. A különböző lefutási sorrendek oka, hogy az ütemező nem feltétlen a szálak létrehozási sorrendjében fogja azokat futtatni, beütemezni. Jól látható, hogy van olyan eset, amikor az egyik szál már be is fejezte működését, holott a második (érdemben) még el sem indult.

A korábbi példa a szálparaméterek felhasználásával a következőképp egyszerűsíthető:
`pthread_example_params.c`

```c
#include <stdio.h>
#include <pthread.h>

// SZÁL routine
void * thread_routine(void * arg) {
    printf("[2] %s szál fut!\n", (const char *) arg);
    pthread_exit(0); // kilépés a szálból
    return NULL; // szál vége, visszatérés, ide nem fogunk eljutni
}

// főprogram (főszál)
int main() {
    pthread_t t1, t2; // szálleírók létrehozása
    pthread_create(&t1, NULL, thread_routine, "Első"); // első szál indítása
    printf("[1] Első szál indítása...\n");
    pthread_create(&t2, NULL, thread_routine, "Második"); // második szál indítása
    printf("[1] Második szál indítása...\n");

    printf("[3] Várakozás az első szálra...\n");
    pthread_join(t1, NULL); // várakozás az első szál befejeződésére
    printf("[3] Várakozás a második szálra...\n");
    pthread_join(t2, NULL); // várakozása a második szál befejeződésére

    return 0;
}
```

A szálak létrehozásához nyugodtan használhatjuk ugyanazt a függvényt, hisz a függvény használ fel semmilyen állapotinformációt – másként fogalmazva a szál állapotának, *kontextusának* nem (változó) része a függvény, és a függvény reentráns.

Lecsatolt szálak Lehetőség van annak kifejezésére, hogy a szál által végrehajtott folyamat valamilyen háttérben futó, *daemon* funkcionalitást valósít meg, a szál visszatérési értékére nem vagyunk kíváncsiak, nem várjuk, hogy a szál visszatérjen. Ilyen esetekben a szál lecsatolható a `pthread_detach()` (`man pthread_detach`) hívással, és a továbbiakban a szál nem tud más szálba becsatlakozni.

5. Folyamatok

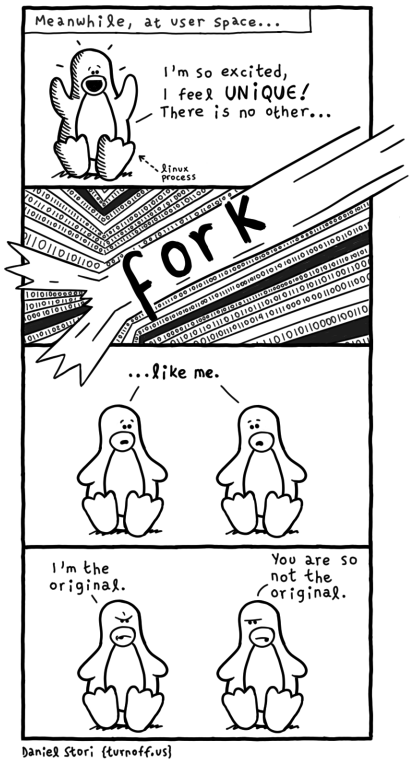
Minden szál egy folyamat része, ami meghatározza a benne futó szálak bővebb kontextusát. Linux alatt új folyamatot (legegyszerűbben) a `fork()` (`man fork`) rendszerhívással tudunk létrehozni.

A `fork()` rendszerhívás hatására a `fork()`-ot hívó folyamatról készül egy másolat, mely *gyermekfolyamat* csak az azonosítójában (PID – Process ID) különbözik a *szülőfolyamattól*. Többek között a gyermekfolyamat megörökli a szülő nyitott fájljait, virtuális címterét – azaz azonos memóriacímek a szülőben és a gyermekben ugyanazon értelemmel bírnak, ugyanarra a fizikai memóriahelyre mutatnak, és sok további részletet a kontextusból.

A `fork()` visszatérése után, magától értetődő módon, a végrehajtás az őt meghívó függvényben folytatódik, hogy melyik a kettő azonos folyamat közül a szülő és melyik a gyermek a `fork()` visszatérési értéke mutatja meg: a szülőfolyamatban a visszatérési érték a gyermekfolyamat PID-je, a gyermekfolyamatban a visszatérési érték *mindig nulla*.

Íme egy példa, ami bemutatja a `fork()` működését: `fork_example.c`

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

int main() {
    printf("Még csak egy van!\n");

    // fork!
    pid_t pid = fork();

    // folyamat saját azonosítójának lekérése
    pid_t ownpid = getpid();

    // innentől már kettő példány fut párhuzamosan
    printf("----\n----\nMár kettő van! [%d]\n", ownpid);
    if (pid == 0) { // gyermek-ág
        printf("Ez a gyermek!\n");
    } else { // ős-ág
        printf("Ez az ős! A gyermekfolyamat PID-je: %d\n", pid);
    }

    // mindkét folyamat így záródik le
    printf("A folyamat befejeződik! [%d]\n", ownpid);

    return 0;
}
```



6. Kommunikáció a szálak között

6.1. Csővezetékek

Szálak közötti kommunikáció sok esetben egyirányú, sorrendtartó kommunikációs csatornákkal, FIFO-kkal, a fizikai világból kölcsönvett elnevezés alapján *csővezetékekkel* hatékonyan megvalósítható. A Linux kettőféle csővezetéket különböztet meg az alapján, hogy a fájlrendszerben a csővezeték név szerint, elérési úttal reprezentálva van-e vagy nem: *névtelen* és *megnevezett* csővezetéket.

6.1.1. Névtelen csővezetékek

A kettő típus közül az egyszerűbben kezelhető a névtelen csővezeték, létrehozása a `pipe()` vagy a `pipe2()` rendszerhívással történik (`man pipe`).

A `pipe()` rendszerhívás egyetlen paramétere egy FD tömb, amibe a létrehozott csővezeték két végéhez tartozó FD-okat helyezi el: a `pipefd[0]` a cső kimenetéhez – tehát a csak olvasható végéhez –, a `pipefd[1]` pedig a cső bemenetéhez – tehát a csak írható végéhez – tartozik. A függvény visszatérési értéke hibakód értelmű, nem nulla visszatérés hibát jelent, az `errno` fogja tartalmazni a hibakódot.

A `pipe2()` rendszerhívás a `pipe()`-ot további beállítások (`flagek`) hozzáadásának lehetőségével egészíti ki.

A korábbi példa csővezetékes kommunikációval kiegészítve alább látható: `pipe_example.c`

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

#define BUFSIZE (32)
int pipefd[2]; // csővezeték FD-ai

// SZÁL 1.
void * thread_1(void * arg) {
    // egy szó beolvasása a terminálról
    printf("(1) Kérek egy szót: ");
    char buf[BUFSIZE];
    scanf("%s", buf);

    // beolvasott adat továbbítása a csővezetéken
    write(pipefd[1], buf, strlen(buf) + 1); // átküldjük a nulla lezárást is!

    return NULL; // szál vége, visszatérés
}

// SZÁL 2.
void * thread_2(void * arg) {
    // adat kiolvasása a csővezetékből
    char buf[BUFSIZE];
    read(pipefd[0], buf, BUFSIZE);
    printf("(2) Szó: %s\n", buf); // kiírhatjuk közvetlenül, mert van nulla lezárás

    return NULL; // szál vége, visszatérés
}

// főprogram (főszál)
int main() {
    // csővezeték létrehozása
    pipe(pipefd);

    // szálak lérehozása
    pthread_t t1, t2; // szálleírók létrehozása
    pthread_create(&t1, NULL, thread_1, NULL); // első szál indítása
    pthread_create(&t2, NULL, thread_2, NULL); // második szál indítása

    pthread_join(t1, NULL); // várakozás az első szál befejeződésére
    pthread_join(t2, NULL); // várakozása a második szál befejeződésére

    return 0;
}
```

A második szál a `read()`-hívás hatására blokkolva várakozik, hogy a csővezetékről tudjon olvasni, azaz a csővezetékre adat kerüljön, más szóval a csővezetékre írjon az első szál.

6.1.2. Megnevezett csővezetékek

A *megnevezett csővezetékek* le vannak képezve a fájlrendszerbe és fájlnévvel rendelkeznek, létrehozásuk az `mkfifo()` rendszerhívással történik (`man 3 mkfifo`). Az így létrejövő speciális fájlt – a korábban bemutatottak szerint – `open()` hívással kell megnyitni külön-külön írásra (`O_WRONLY`) és olvasásra (`O_RDONLY`). A csővezeték csak akkor fog működni, ha mindkét vége meg van nyitva, mindaddig amíg ez meg nem történik, a korábban hívott `open()` blokkolni fog.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

const char pipename[] = "csovezetek";
#define BUFSIZE (32)

// SZÁL 1.
void * thread_1(void * arg) {
    // egy szó beolvasása a terminálról
    printf("(1) Kérek egy szót: ");
    char buf[BUFSIZE];
    scanf("%s", buf);

    // csővezeték bemenő felének megnyitása
    int fdwrite = open(pipename, O_WRONLY);

    // beolvasott adat továbbítása a csővezetéken
    write(fdwrite, buf, strlen(buf) + 1); // átküldjük a nulla lezárást is!

    return NULL; // szál vége, visszatérés
}

// főprogram (főszál)
int main() {
    // megnevezett csővezeték létrehozása
    unlink(pipename); // esetleges korábbi csővezeték törlése
    int ret = mkfifo(pipename, 0666); // új létrehozása

    // új szál lérehozása
    pthread_t t1; // szálleírók létrehozása
    pthread_create(&t1, NULL, thread_1, NULL); // új szál indítása

    // csővezeték kimenetének megnyitása
    int fdread = open(pipename, O_RDONLY);

    // olvasás a csővezetékről
    char buf[BUFSIZE];
    read(fdread, buf, BUFSIZE);
    printf("(F) Szó: %s\n", buf);

    // várakozás az indított külön szál befejeződésére
    pthread_join(t1, NULL);

    return 0;
}
```