Rendszerközeli programozás

Projekt feladat

Hozz létre Linux alatt egy olyan C nyelvű programot, amely egy 1bit színmélységű („fekete-fehér”, kétszínű) bmp fájlt hoz létre, amely egy véletlenszerűen változó mennyiség időbeli változását szemléltető grafikont ábrázol. A program kétféle üzemmódban tudjon működni: az egyik előállítja az adatokat (ő a *küldő*), a másik ábrázolja azokat (ő a *fogadó*). A program két eltérő üzemmódban indított példánya (tehát egyetlen programról, de két folyamatról van szó) folyamatok közötti kommunikáció révén valósítsa meg az adatok átadását, vagy fájlon vagy socket-en keresztül! A program fejlesztése során az alábbi funkciókat implementáld a leírásnak megfelelően:

1. feladat

Hozd létre a C program fő programegységét, amely a parancssori argumentumokat is képes kezelni! Ha a ”`--version`” parancssori argumentummal indítjuk el a programot, akkor írja ki a program verziószámát, elkészültének dátumát és a fejlesztő nevét (esetleg tetszőleges további információkat, mindegyiket külön sorba)! Ha a ”`--help`” argumentummal indul a program, akkor adjon tájékoztatást a felhasználónak a futtatás lehetséges opcióiról, a lehetséges parancssori argumentumok jelentéséről! A fenti esetekben a végrehajtás ezután fejeződjön is be! A program üzemmódját a ”`-send`” vagy a ”`-receive`” parancssori argumentummal, adhatjuk meg. Az előbbi legyen az alapértelmezett azaz, ha egyik sincs megadva explicit üzemmód, akkor *küldő*ként viselkedjen a rendszer! A kommunikáció módját a ”`-file`” vagy a ”`-socket`” parancssori argumentummal, adhatjuk meg. Az előbbi legyen az alapértelmezett azaz, ha egyik sincs megadva egyik sem, akkor fájlt használjon a kommunikáció során! Az üzemmód és a kommunikációs mód kapcsolók együttes használata esetén a sorrendjük tetszőleges. Amennyiben a program nem érvényes argumentumot kap, akkor viselkedjen ugyanúgy, mint a ”`--help`” kapcsoló estén!
Amennyiben a futtatható állomány neve nem ”`chart`”, akkor a program írjon ki egy erre vonatkozó hibaüzenetet és álljon le!
A főprogram fejléce legyen a szokásos:

```
int main(int argc, char* argv[]);
```

Mivel a program többi része még nincs kész, az egyes futtatási lehetőségek esetén egyelőre csak egyszerűen írasd ki, mi fog majd akkor történni! (A végső verziónak nem kell tartalmaznia ezeket az üzeneteket.)

2. feladat

1. lépés

Egy képzeletbeli szenzor által rendszeres időközönként mért értékeket egy függvénnyel állítjuk elő. Ez a függvény egy 3 állapotú 1 dimenziós bolyongást implementál, azaz véletlenszerűen előállítja egész számok egy véges sorozatát, ahol bármely két szomszédos elem különbségének az abszolút érteke maximum 1. A kezdő érték legyen az $x_0=0$! A további elemeknél az $x_{i+1}=x_i+1$ eset `42,8571%` eséllyel forduljon elő! Az esetek `11/31`-ed részében $x_{i+1}=x_i-1$ állítás teljesüljön! A szomszédos értékek néha lehetnek azonosak is ($x_{i+1}=x_i$). A „mért” értékek darabszáma egyezzen meg a hívás pillanatában az adott negyedórából eltelt másodpercek számának és a 100-nak a maximumával! (Például délelőtt 9 óra 41 perc 27 másodperckor 687 darab egész számot kell generálni, de 10 óra 15 perc 24 másodperckor 100 darab egész számot.) A program minden futtatás esetén másik számsort állítson elő!
A függvény fejléce a következő legyen:

```
int Measurement(int **Values);
```

Az egyetlen paraméter egy `int*` típusú pointer címe legyen, mivel ez majd output paraméteként szolgál (pszeudocím szerinti paraméterátadású mutató). Ebben a függvényben foglalj le dinamikus memóriafoglalás segítségével egy megfelelő méretű tömböt, ebben legyen tárolva az előállított véletlenszerű adatsorozat és a terület címe legyen elmentve azon a memóriacímen, amit paraméterként kap a függvény (ezáltal az itt lefoglalt és inicializált terület címe visszajut a hívó programegységbe, ami így el tudja érni az adatsort). Végül a függvény térjen vissza az előállított értékek számával!

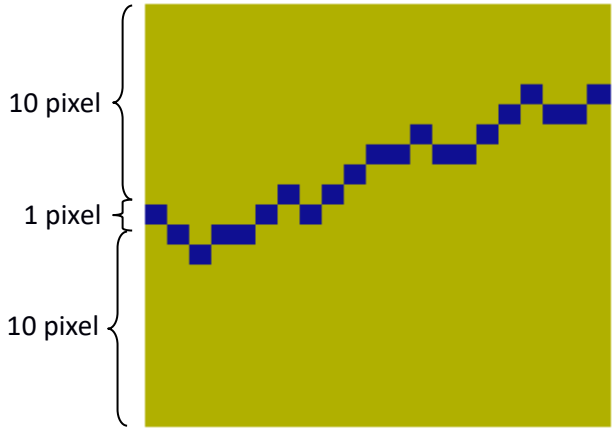
2. lépés

A *küldő* üzemmódban futó program esetén hívd meg a `main`-ben fenti `Measurement` függvényt! Ideiglenes tesztként itt a főprogramban (a hívás után) írasd ki a képernyőre az adatsort! (A végső verziónak nem kell tartalmaznia ezt a tesztet.)

3. feladat

1. lépés

Az előbbi feladatban előállított adatsort, mint egy időben változó fizikai mennyiség rendszeres időközönként megvalósult mérésének az eredményét egy grafikonon szeretnénk ábrázolni. Ehhez az adatsort, mint folytonos görbét egy BMP képfájl segítségével jelenítjük meg. A képfájlnak a színmélysége legyen 1, azaz csak két szint tartalmazhasson (az egyik az adatpont, a másik a háttér)! Nem kötelező, hogy ezek a fekete és a fehér színek legyenek, így a képfájlban definiálj egy (tetszőleges) kétszínű palettát az RGBA színteret alkalmazva! A kép pixelben megadott szélessége és magassága legyen egyenlő az ábrázolandó adatok számával! Ennek megfelelően tehát a kép minden oszlopában pontosan egy pixel lesz előtér színű, az ábrázolandó adat. Az adatsort úgy jelenítsd meg, hogy a legelső érték ($x_0=0$) az első oszlopnak a (függőlegesen) középső pixele legyen és ennek megfelelően jelenjen meg a többi érték is! (Amennyiben a kép magassága páros érték, akkor a két középső pixel bármelyike használható.) Ha valamelyik megjelenítendő érték túl nagy vagy túl kicsi lenne (azaz kilógna a képből), akkor azt a legfelső, illetve a legalsó pixel segítségével jelenítsd meg!



Egy példa grafikon az alábbi 21 elemű adatsor megjelenítésére:
{0, -1, -2, -1, -1, 0, 1, 0, 1, 2, 3, 3, 4, 3, 3, 4, 5, 6, 5, 5, 6}.

A képet létrehozó eljárás fejléce legyen az alábbi:

```
void BMPcreator(int *Values, int NumValues);
```

Az első paraméter az ábrázolandó értékeket tartalmazó tömb kezdőcíme, a második pedig az előbbi tömbben tárolt egész értékek száma. A létrehozandó képfile neve legyen `chart.bmp`, abban a könyvtárban legyen, hol a *fogadó* program el lett indítva és azt minden felhasználó olvashassa, de csak a tulajdonosnak legyen írási joga hozzá!

Az egy bit színmélységű tömörítetlen bitmap (BMP) képfájlok felépítéséről a mellékletben olvashatsz.

2. lépés

Tesztelés céljából a *küldő* üzemmódban futó programban a `Measurement` függvény meghívása után átmenetileg hívd meg a `BMPcreator` eljárást a kapott adatokra! (A végső verziónak nem kell tartalmaznia ezt a tesztet.)

4. feladat

1. lépés

Írj egy függvényt, amely a Linux fájlrendszer gyökerében lévő `"/proc"` könyvtárnak az alkönyvtáraiban található `"status"` nevű fájloknak a tartalmát vizsgálja meg. (Azok az alkönyvtárak tartalmaznak ilyen fájlokat, amelyek neve számjegy karakterrel kezdődik.) A fájl első sorának a formátuma: `"Name:\t%s\n"`. Ha a tabulátor és az újsor karakter között a `"bash"` karaktersorozatot található, akkor keressen az adott fájlban olyan sort, amely a `"Pid:\t"` sztriggel kezdődik majd ezt egy egész szám követi. A függvény térjen vissza ezzel az egész számmal, ha pedig egyáltalán nem talál ilyen fájlt egyik alkönyvtárban sem, akkora `-1` értékkel! (Persze a megnyitott fájlokat és könyvtárakat zárja be!) A függvény fejléce legyen a következő:

```
int FindPID();
```

2. lépés

Ideiglenes tesztként hívd meg a `FindPID` függvényt a főprogramban és ellenőrizd le manuálisan a megtalált érték helyességét! (A végső verziónak nem kell tartalmaznia ezt a tesztet.)

3. lépés

A programban eddig definiált (és a jövőben definiálandó) alprogramokat szervezd ki egy külön saját, `".h"` kiterjesztésű header állományba, amit a main függvény forráskódjában inkludálsz is!

5. feladat

1. lépés

A `FindPID` nevű processzus azonosítót kereső függvényt módosítsd úgy, hogy ne a `"bash"` nevet, hanem a `"chart"` nevet figyelje/keresse, valamint hagyja figyelmen kívül azt az esetet, amikor egy `"status"` fájlban megtalált érték megegyezik a program saját PID-jével!

2. lépés

Hozz létre egy eljárást, az alábbi paraméterezéssel:

```
void SendViaFile(int *Values, int NumValues);
```

A `Values` mutató egy egészeket tartalmazó tömb kezdőcímét kapja paraméterátadás során, míg a `NumValues` változó fogja tárolni a tömbben lévő egészek darabszámát. Az eljárás hozzon létre egy `"Measurement.txt"` nevű szöveges fájlt az adott felhasználó saját alapértelmezett könyvtárában és soronként írja bele a tömbben lévő értékeket a fájlba (`"%d\n"` formátummal)! Miután bezárta a fájlt, hívja meg az eljárás a `FindPID` nevű függvényt! Ha a függvény `-1` értékkel tér vissza, akkor a program írjon ki egy hibaüzenetet, arra vonatkozóan, hogy nem talál *fogadó* üzemmódban működő folyamatot (process-t) majd a program álljon le egy hibakóddal (nem 0 visszatérési értékkel)! Amennyiben a `FindPID` más értékkel tér vissza, az eljárás küldjön neki egy felhasználói 1-es szignált (`SIGUSR1`)!

3. lépés

Írj egy eljárást, amely az adott felhasználó alapértelmezett könyvtárában lévő `"Measurement.txt"` nevű szöveges fájlt megnyitja, a tartalmát beolvassa és eltárolja egy memóriaterületre, melyet szükség esetén dinamikus memóriafoglalással bővít (mivel nem tudjuk

előre az elemek számát). Ezt követően az eljárás hívja meg a `BMPcreator` eljárást és adja át neki a beolvasott értékeket és azok darabszámát! Végül a dinamikusan lefoglalt memóriaterület legyen felszabadítva! Az eljárás fejléce, legyen az alábbi (a paraméter értékét nem fogjuk felhasználni, de szükséges):

```
void ReceiveViaFile(int sig);
```

4. lépés

Módosítsd a főprogramot úgy, hogy a *küldő* üzemmódban futó és fájlon keresztüli kommunikációt folytató folyamatban hívd meg a `Measurement` függvényt, majd az általa előállított adatokat add át paraméterként a `SendViaFile` eljárásnak, végül szabadítsd fel azt a dinamikusan foglalt területet, ahol a „mérési" adatok tárolva vannak! (Ezt követően a program álljon le!)
Módosítsd továbbá a főprogramot úgy, hogy a *fogadó* üzemmódban futó és fájlon keresztüli kommunikációt folytató folyamatban készülj fel felhasználói 1-es szignál érkezésére, ilyen esetben a `ReceiveViaFile` eljárás kerüljön automatikusan meghívásra. Addig is a folyamat kerüljön (szignálra) várakozó állapotba egy végtelen ciklusban, azaz szignál érkezése után sem álljon le a program, hanem a szigálkezelő lefutása után várjon újabb szignálra!

6. feladat

1. lépés

Írj egy eljárást, amely UDP protokoll segítségével a localhost (IPv4 cím: `127.0.0.1`) `3333`-as portját figyelő *fogadó* üzemmódú szoftverrel kommunikál. Az eljárás fejléce így nézzen ki:

```
void SendViaSocket(int *Values, int NumValues);
```

Itt az első paraméterként kapott memóriacím egy tömb kezdőcíme, a második pedig a tömbben lévő egész típusú értékek száma. Az eljárás a socketen keresztül küldje el a *fogadó*nak a `NumValues` változó értékét (32 bites fix pontos egészként)! Ezután várjon a szerver válaszára, ami szintén egyetlen 4 bájtos egész szám lesz (`int`). Ha a küldött és a kapott értékek eltérőek, akkor egy hibaüzenet után a program egy eddigiektől eltérő hibakóddal álljon le! Ha az értékek megegyeznek, akkor az eljárás a `Values` címen kezdődő tömb `NumValues` darab `int` típusú értékét küldje át egyetlen üzenetben a *fogadó*nak. Ezután ismét várjon egy válaszüzenetre, ami egy 4 bájtos egész szám lesz. Ha a küldött adatok bájtban megadott mérete és a most kapott értékek eltérőek, akkor is egy hibaüzenet után hibakóddal álljon le a program!

2. lépés

Írj egy másik socket programozáson alapuló eljárást is, amely egy végtelen ciklusban UDP szegmenseket vár a `3333`-as porton. Az első kapott szegmensben feltételezzük, hogy mindig egy 4 bájtos hasznos tartalom van (egy `int` változó értéke). Az eljárás nyugtaként küldje vissza a kapott értéket a *küldő* üzemmódú kliens folyamatnak, valamint dinamikusan foglaljon le ennyi darab egész szám számára egy folyamatos memóriaterületet! Ide tárolja el a második üzenetben kapott adatokat, amelyeknek a bájtban megadott méretét nyugtaként juttassa vissza a küldőnek! A kapott adatokkal kerüljön meghívásra a `BMPcreator` eljárás, végül a lefoglalt memóriaterületek legyenek felszabadítva és az eljárás várjon újabb üzenetet egy *küldő*től! (Feltehetjük, hogy egy adott pillanatban csak egyetlen *küldő* aktív és az mindig elküldi mind a két üzenetet.). Az eljárás fejléce egyszerűen legyen ez:

```
void ReceiveViaSocket();
```

3. lépés

Módosítsd a főprogramot úgy, hogy a *küldő* üzemmódban futó és socketen keresztüli kommunikációt folytató folyamatban hívd meg a `Measurement` függvényt, majd az általa előállított adatokat add át

paraméterként a `SendViaSocket` eljárásnak, végül szabadítsd fel azt a dinamikusan foglalt területet, ahol a mérési adatok tárolva vannak! (Ezt követően a program álljon le!)
Módosítsd továbbá a főprogramot úgy, hogy a *fogadó* üzemmódban futó és socketen keresztüli kommunikációt folytató folyamatban a `ReceiveViaSocket` eljárás kerüljön meghívásra.

7. feladat

1. lépés

Írj egy szignálkezelő eljárást (természetesen ez is a header állományba kerüljön), amely háromféle szignál kezelésére képes és az alábbi fejléccel rendelkezik:

```
void SignalHandler(int sig);
```

Ha az eljárás megszakítási szignált (`SIGINT`) kap, akkor írjon ki egy elköszönő üzenetet és a program álljon le és adjon vissza egy `0` értéket!
Ha az eljárás felhasználói 1-es szignált (`SIGUSR1`) kap, akkor írjon ki egy hibaüzenetet arra vonatkozóan, hogy a fájlon keresztüli küldés szolgáltatás nem elérhető!
Ha az eljárás alarm szignált (`SIGALRM`) kap, akkor írjon ki egy hibaüzenetet arra vonatkozóan, hogy a szerver nem válaszol (időkereten belül), majd a program egy hibakóddal álljon le!

2. lépés

A főprogramot bármiféle kommunikáció megvalósítása előtt készítsd fel `SIGINT` és `SIGUSR1` szignálok érkezésére, amelyeket a `SignalHandler` eljárás fog lekezelni!
Módosítsd továbbá a `SendViaSocket` alprogramot úgy, hogy az első üzenet elküldése után érkező `SIGALRM` szignál esetén automatikusan hívja meg a `SignalHandler` szignálkezelő eljárást, majd indíts el egy 1 másodperces időzítőt! Amennyiben az első üzenet elküldése utáni egy másodpercben választ kap a szervertől a program, akkor hagyja figyelmen kívül az érkező alarm szignált!

8. feladat

Az alábbiak szerint módosítsd a főprogram azon részét, amely kizárólag a `"--version"` kapcsoló használata esetén aktív! A kiíratandó összes sort más-más szál (thread) valósítsa meg párhuzamosan, így a kimenetek a képernyőn „véletlenszerű” sorrendben jelennek meg.

9. feladat

Készíts (rövidített) dokumentációt a programodhoz, amely tartalmazza a következőket:

- A használandó fordítóprogram és annak szükséges kapcsolói.
- Rendszerkövetelményeket.
- Felhasználói útmutatást a programhoz.
- A program által visszaadott értékek magyarázatát.
- Az elkészített alprogramok rövid leírását (cél, paraméterezés, visszatérési érték).

A dokumentáció feltöltendő változata, ne legyen szerkeszthető és hordozható formátumú legyen!

10. feladat

Védd meg a programodat! Bizonyítsd be, hogy a te alkotásod! Válaszolj a feltett kérdésekre! Hajtsd végre a kért módosításokat!

További elvárások

A program és annak alprogramjai pontosan úgy működjenek, ahogy az a leírásban szerepel! Akár két külön fejlesztő programjainak is képesnek kell lennie a feladat együttes ellátására. A program lehetőleg legyen hatékony, stílusos, könnyen olvasható! A program legyen saját fejlesztésű! A program forráskódját ne tedd elérhetővé mások számára, ne oszd meg! (A félév zárása után sem!)

Melléklet

Az egybites színmélységű tömörítetlen BMP képfájlok szerkezete

Az ilyen képfájlok tartalma két részre osztható: **fejrész**(ek) és **pixeltömb**. Többféle megoldás és lehetőség van a pontos szerkezetre vonatkozóan, de itt csak a projekt során alkalmazandó szerkezet kerül bemutatásra.

Fejrész

Különböző általános leírást, definíciókat, tulajdonságokat tartalmaz (például: fájlméret, felbontás, színmélység, tömörítési technika, paletta, stb). Egy BMP képet jelentő bináris fájltartalom a fejrész mezőkkel kezdődik. A fájl az alábbi, hexadecimális formában megadott, bájtsorozatot kell tartalmazza:

```
0x42, 0x4d, 0x__, 0x__, 0x__, 0x__, 0x00, 0x00, 0x00, 0x00,
0x3e, 0x00, 0x00, 0x00, 0x28, 0x00, 0x00, 0x00, 0x__, 0x__,
0x__, 0x__, 0x__, 0x__, 0x__, 0x__, 0x01, 0x00, 0x01, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61, 0x0f,
0x00, 0x00, 0x61, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x__, 0x__, 0x__, 0xff, 0x__, 0x__,
0x__, 0xff
```

Az üresen hagyott 3.-6. bájtok területére a fájl teljes méretét kell írni bájtban (62bájt+a pixeltömb mérete) előjel nélküli fixpontos egész számként little-endian bájtsorrendben, azaz úgy, ahogy egy `unsigned int` típusú változó a memóriában el van tárolva. Ugyanilyen módszerrel kell megadni a 19.-22. és a 23.-26. bájtok értékét, ahol előbbi a kép pixelben megadott szélessége, utóbbi pedig a magassága (pixelsorok száma). Az utolsó kétszer 4 bájt területén kell megadni az alkalmazandó paletta színeit. Mivel az egybites színmélység csak két színt enged meg (azaz minden pixel színazonosítója 1 biten lesz eltárolva), a pixeltömbben a `0` bit jelenti a paletta első színét, az `1` bit pedig a másodikat. A paletta elemeit (azaz a két színt) RGBA színtérben kell megadni 32 biten. Ez azt jelenti, hogy az első bájt (0x00-0xff közötti érték) a kék színkomponens értéke, a második bájt a zöld, a harmadik a piros, és az utolsó az átlátszóságot adja meg (0: teljesen átlátszó, 255: egyáltalán nem átlátszó).

Pixeltömb

A kép egyes pixeleinek a színinformációit tartalmazza sorfolytonosan, balról jobbra és alulról felfelé haladva. Minden képpont 1 biten van eltárolva. Így tehát ha a pixeltömb első bájtjának értéke (hexadecimálisan) 0x0f, akkor ez azt jelentheti, hogy a kép legalsó sorának bal szélső négy pixele a paletta első elemeként megadott színnel rendelkezik, míg a sor következő négy képpontja a paletta második színével jelenítendő meg. Egy BMP kép minden sora úgy van eltárolva, hogy annak mérete 4 bájt egész számú többszöröse. Tehát ha egy kép szélessége nem osztható 32-vel, akkor ún. kitöltő (padding) biteket kell alkalmazni a megfelelő méret eléréséhez. Ezek a (szükség esetén) sorok végén előforduló bitek tehát nem kódolnak szín indexeket (vizuálisan nem megjelenítendőek), értékük tetszőleges, általában `0`. Ennek megfelelően, például egy pepita (sakktábla) mintás 22×22 pixel felbontású kép első két sor a pixeltömbben így írható le:

```
0x55, 0x55, 0x54, 0x00, 0xaa, 0xaa, 0xa8, 0x00
```