Blekinge Tekniska Högskola

DV1460 – Realtid- och operativsystem

Laboration IV - Simulering av filsystem

Författare: Carina Nilsson, Cuong Phung

Innehåll

1 Syfte **2**

2 Förutsättningar **2**

3 Redovisning **2**

4 Hjälpfiler **2**

4.1 Klassdiagram 3

4.2 Kompilering och körning 4

5 Uppgift **4**

6 Bedömning **5**

1 Syfte

Syfte med laborationen är att du ska utveckla en grundläggande teknisk förståelse för hur ett filsystem är organiserat och hur de nödvändigaste funktionerna kan implementeras.

2 Förutsättningar

Det finns ett antal hjälpfiler att ladda ner från kurshemsidan på It's Learning

Det är ett krav att filsystemet ska byggas på en simulerad disk i form av en 2-dimensionell byte-array 250×512 byte stor. (250 block à 512 byte vardera). "Disken" får bara läsas och skrivas blockvis, dvs. man läser eller skriver alltid 512 byte i taget.

I övrigt är du fri att definiera dina datastrukturer som du vill.

3 Redovisning

Redovisningen sker gruppvis. Uppgiften ska redovisas genom att ladda upp källkod och rapport på It It's Learning i form av en arkivfil (exempelvis .zip). Rapporten ska följa riktlinjerna i "Introduktion till rapportskrivning".

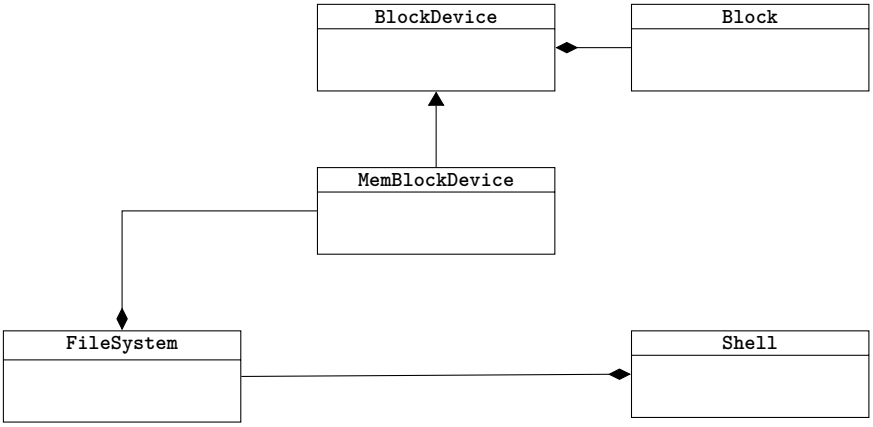
Observera att det är inte tillåtet att kopiera kod som någon annan än gruppmedlemmarna har konstruerat.

4 Hjälpfiler

Till din hjälp finns det redan ett halvklart filsystem i form av 5 klasser (Se figur 1) skrivna i C++. Det finns en färdiggjord `Makefile` för att underlätta för dig som väljer att kompilera i kommandotolken. Instruktioner på hur du kan kompilerar finner du nedan.

Det är också tillåtet att ignorera hjälpfilerna och implementera din egen lösning.

4.1 Klassdiagram



Figur 1: Klassdiagram för hjälpfilerna.

Block Är en klass som har en dynamisk `char`-array, vars ”default-konstruktor” allokerar minne för 512 tecken (byte).

BlockDevice En rent virtuell klass, där flera funktioner inte är implementerade, dessa är implementerade i klassen **MemBlockDevice**. Har en dynamisk allokerat fält innehållande **Block**-objekt.

MemBlockDevice Ärver från BlockDevice och där finns flera funktioner. Standard-konstruktorn kommer att generera 250 block med en blockstorlek på 512 byte. Nedan följer en kort beskrivning på de nödvändigaste funktionerna:

- `int writeBlock(int blockNr, ...)` har som inparameter antingen: `std::string`, `std::vector<char>` eller `char[]`. Returnerar -2 när inparametern och blockstorleken inte är av samma längd, -1 vid anrop av ett felaktigt blocknummer och 1 vid lyckad skrivning. *Notera!* om man använder `char[]` så görs ingen kontroll på fältets längd.
- `Block readBlock(int blockNr)`. Läser ett block, kastar ett `std::out_of_range` om anger ett blocknummer som inte existerar.

FileSystem Har ett **MemBlockDevice**-objekt. Meningen är att du ska implementera API-funktionerna som filsystemet tillhandahåller: `createFile(...)`, `createFolder(...)`, osv. Det mesta av din kod kommer att hamna här.,

Shell Är egentligen ingen klass, utan har flera funktioner däribland `main()`. Här finns redan kommandotolk implementerad.

Du är fri att ta bort, lägga till och modifiera koden enligt ditt tycke. Vissa funktioner kanske behöver modifieras för att uppnå vissa betygskriterier.

4.2 Kompilering och körning

Det finns redan en s.k. `Makefile`, denna fil innehåller kommandon för hur man kompilerar koden. Enklast är att gå till din projektmapp och skriva:

```
make all
```

vilket kommer kompilera all kod och generera en körbar fil (`Shell`) som ligger i katalogen `bin/`. Tänk på att om du väljer att lägga till fler filer i projektet så måste du uppdatera `Makefile`.

Det går också bra att importera alla källkodsfiler (.cpp och .h) i en IDE (t.ex Visual Studio, Eclipse osv.) och kompilera och köra därifrån.

5 Uppgift

Uppgiften utförs i grupper om två. Annan gruppstorlek ska beviljas av handledaren. Grupper med fler än tre deltagare godtas inte.

Ett filsystem ska konstrueras. Sekundärminnet som organiseras av filsystemet är simulerat som en 2-dimensionell byte-array (se Avsnittet Hjälpfiler, väljer du att ignorera hjälpfilerna se till att din implementation har en matris av 250 st block med 512 byte i varje) . Filsystemet ska vara en hierarkisk trädstruktur som kan ha godtyckligt många undernivåer. Systemet ska klara av namn av typen `/aaa/bbb/ccc...` Roten betecknas `/`. Se också till att din lösning inte genererar minnesläckor.

Vad som ska implementeras för att ge respektive betyg anges nedan.

- **Tips 1** Tänk noga igenom hur strukturerna för filsystemet ska se ut innan du börjar koda.
- **Tips 2** Om ni är osäkra för hur ert filsystem är tänkt att bete sig för olika kommandon går det bra att testa hur respektive kommando fungerar i valfritt Linux/UNIX-system. Exempelvis genom att testa hur kommandona fungerar på en dator i laborationssalen.

6 Bedömning

För att godkännas på uppgiften ska följande Shell-kommandon implementeras med hjälp av filsystemets API:

`format` bygger upp ett tomt filsystem, dvs. formatterar disken.

`quit` lämnar körningen <Redan gjord i filsystemet>

`createImage <filnamn>` sparar det simulerade systemet på en fil på datorns hårddisk så att den går att återskapa vid ett senare tillfälle.

`restoreImage <filnamn>` återstället systemet från en fil på datorns hårddisk.

`create <filnamn>` skapar en fil på den simulerande disken (datainnehållet skrivs in på en extra tom rad)

`cat <filnamn>` skriver ut innehållet i filen filnamn på skärmen.

`ls` listar innehållet i aktuellt katalog (filer och undermappar).

`copy <fil1> <fil2>` skapar en ny fil `fil2` som är en kopia av den existerande fil `fil1`.

`mkdir <katalognamn>` skapar en ny tom katalog på den simulerande disken.

`cd <katalognamn>` ändrar aktuell katalog till den angivna katalogen på den simulerade disken.

`pwd` skriver ut den fullständiga sökvägen ändå från roten till filnamnet.

`rm <filnamn>` tar bort angiven fil från den simulerade disken.

Tänk på att din lösning även ska kunna hantera:

- **Antingen** relativa sökvägar eller absoluta sökvägar.
- Det ska vara möjligt att (för vissa kommandon) att använda sig av aktuell katalog (`.`), eller referera ett steg upp (mot roten) i hierarkin med hjälp av (`..`). T.ex

```
/user/home$: cd ..
/user/$:
```

I rapporten ska det finnas:

- En tydlig beskrivning av den logiska strukturen i *ert* filsystem.
- Tydlig beskrivning över metoderna i ert API (hur anropas och vad de gör).
- Ett simpelt klassdiagram samt en kort beskrivning av varje klass.