Lab 4: Filsystem

Författare: Kim Svensson Sand och Lukas Städe
Datum: 2016-12-13
Kurs: DV1460 H16 Lp1 Realtids- och operativsystem

Sammanfattning

Vi har i den här laborationen implementerat ett filsystem i c++ för att se hur ett filsystem kan vara uppbyggt och hur några av funktionerna kan implementeras. I filsystemet har vi implementerat funktioner för att en användare som kör programmet ska kunna hantera filsystemet med hjälp av specifika kommandon. Vi utgick ifrån hjälpfiler som innehåller några klasser som redan implementerar ett halvfärdigt filsystem. Filsystemet är uppbyggt enligt en trädstruktur då det kändes naturligt och implementerat i Visual Studio 2015, men det är också testat i skolans Linuxmiljö.

Innehållsförteckning

1 Inledning

I den här laborationen ska vi undersöka hur ett filsystem kan vara uppbyggt och hur man skulle kunna implementera några av funktionerna som är nödvändiga i ett sådant. Data i filsystemet ska lagras i en byte-array som är 250 block stor där varje block är 512 byte. Att skriva och läsa från byte-arrayen ska alltid ske blockvis, alltså 512 byte i taget.

De funktioner som ska implementeras är följande shell-kommandon:
`format` - Formaterar disken och återställer filsystemet.
`quit` - Avslutar körningen av filsystemet.
`createImage <realFileName>` - Sparar den simulerade diskens tillstånd i en fil utanför det simulerade filsystemet som kan användas för återskapa tillståndet.
`restoreImage<realFileName>` - Återskapar ett tidigare sparat tillstånd från en fil utanför det simulerade filsystemet.
`create <filePath>` - Skapar en fil och låter dig göra input till filen.
`cat <filePath> -` Skriver ut innehållet i en fil.
`ls` - Skriver ut innehållet i en katalog i en lista.
`cp <filePath1> <filePath2>` - Kopierar en fil.
`mkdir <directoryPath>` - Skapar en ny katalog.
`cd <directoryPath>` - Navigerar till en katalog.
`pwd` - Skriver ut pathen från cwd(current working directory).
`rm <filePath>` - Tar bort en fil.

I den här laborationen måste vi planera och välja en struktur för vårt operativsystem och sen använda oss av den strukturen när vi implementerar våra funktioner.

2 Metod

För att implementera filsystemet hade vi tillgång till hjälpfiler som implementerar ett halvfärdigt filsystem.

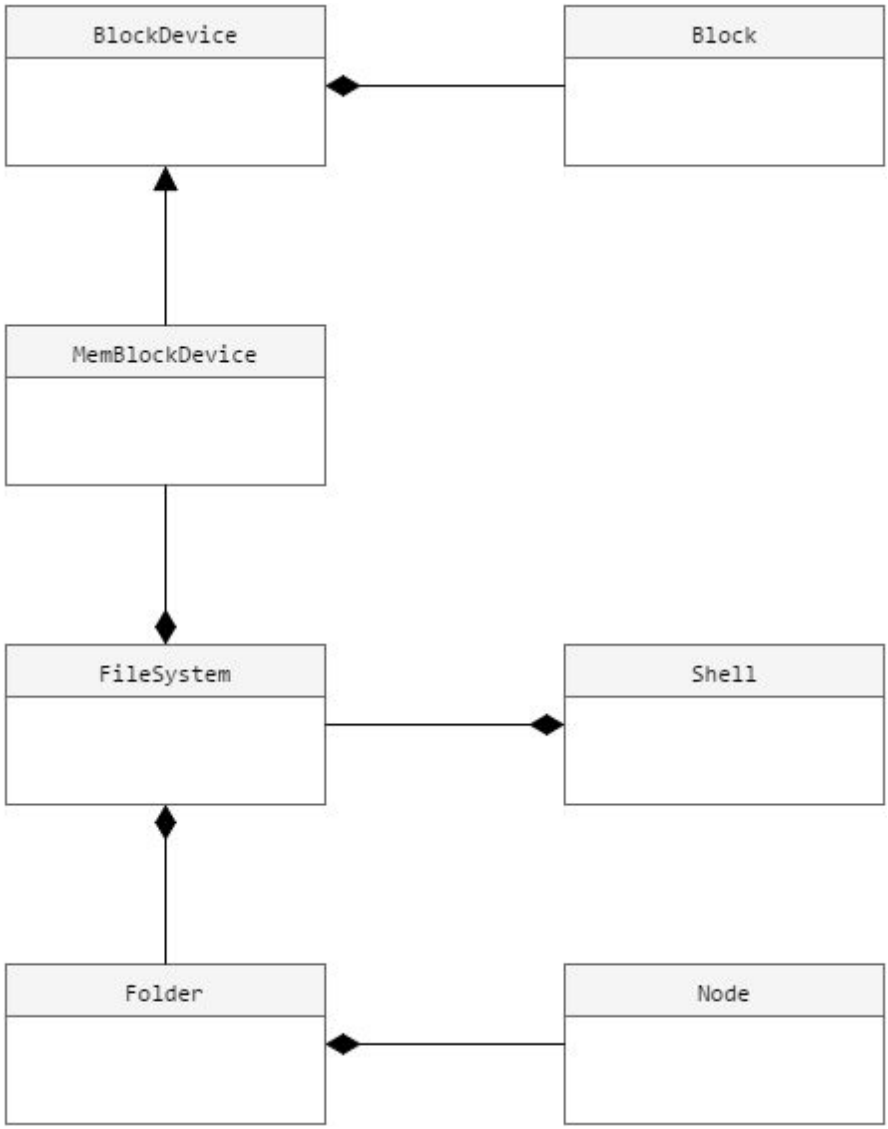
2.1 Hjälpfiler

Hjälpfilerna implementerar ett redan halvfärdigt filsystem i c++ som skapar byte-arrayen att lagra data i, har färdiga funktioner för att till exempel skriva och läsa från block och en implementerad kommandotolk.

Följande finns implementerat i hjälpfilerna:
`Block` - En klass som har en dynamisk char-array. Dess default-konstruktor allokerar 512 byte.
`BlockDevice` - En virtuell klass. Har ett dynamiskt allokerat fält innehållande Block-objekt.
`MemBlockDevice` - Ärver från BlockDevice. Default-konstruktorn kommer att skapa 250 block av storlek 512 byte. Innehåller funktioner för att skriva och läsa från block.
`FileSystem` - Har ett MemBlockDevice-objekt. Här implementerade vi funktionerna till filsystemet.

`Shell` - Innehåller `main()` och har en kommandotolk med olika kommandon för att anropa funktionerna till filsystemet.

Hjälpklasserna finns med i ett klassdiagrammet, se figur1.



Figur1, Klassdiagram för filsystemet där Folder och Node inte ingår i hjälpfilerna.

2.2 Miljö

Då hjälpfilerna var implementerade i c++ valde vi att implementera färdigt filsystemet i en windowsmiljö i Microsoft Visual Studio 2015. Vi testade också att exekvera filsystemet på skolans datorer med operativsystemet Debian.

2.3 Struktur

Innan vi började implementera funktioner började vi fundera över vilken struktur vårt filsystem skulle ha. Vi valde att implementera två nya klasser:

`Folder` - Motsvarar en katalog och innehåller en vector med `Folder*` och en vector med `Nodes*`, som motsvarar filer. Varje `Folder` innehåller också en pekare till sin förälder.

```
class Folder {
public:
	Folder();
	Folder(string name, Folder * parent);
	~Folder();
	void createFolder(string name);
	void createNode(string name, int size, int blockNr);
	string getName();
	void setName(string newName);
	void setParent(Folder * parent);
	int getnrOfFolders();
	int getnrOfNodes();
	Folder* getParent();
	Folder* getFolder(string name);
	Node* getNode(string name);

	vector<Folder*> getFolders();
	vector<Node*> getNodes();
	int deleteNode(Node* node);
private:
	Folder *parent;
	vector<Folder *> folders;
	vector<Node *> nodes;
	int nrOfNodes;
	int nrOfFolders;
	string name;
};
```

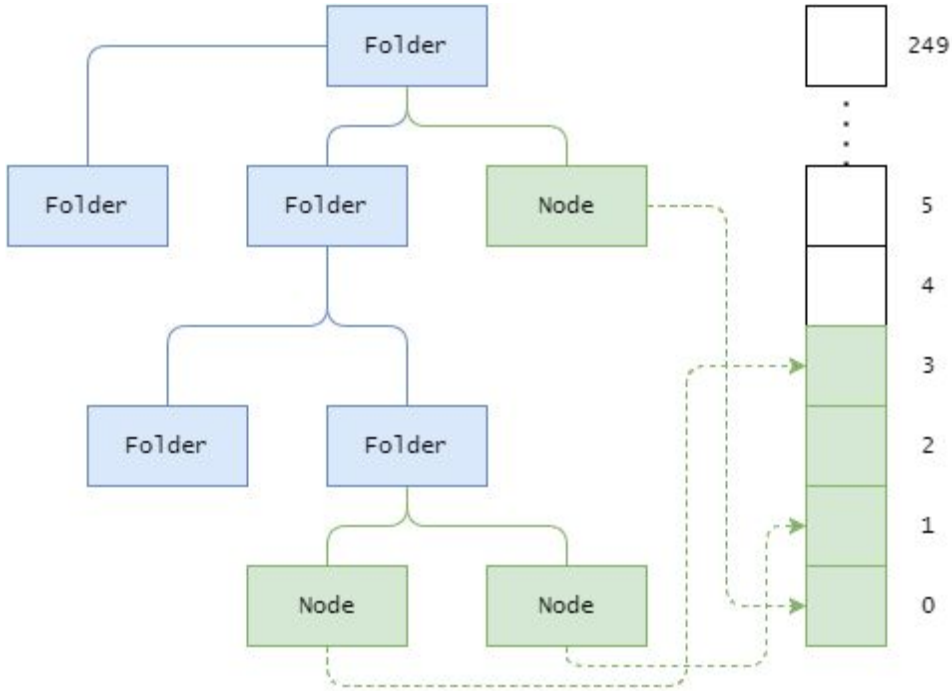
`Node` - En klass som motsvarar en fil. Den innehåller filens namn, filens storlek i block och i vilket block filen börjar.

```
class Node {
  private:
    int blockNr;
    int size;
    std::string name;
  public:
    Node();
    Node(std::string name, int size, int blockNr);
    ~Node();

      int getSize();
      std::string getName();
```

```
        int getBlockNr();
        void setBlockNr(int blockNr);
        void setSize(int size);
};
```

Klasserna finns med i klassdiagrammet, se figur1.



figur2, Diagram över strukturen i filsystemet.

Strukturen i vårt filsystem är att varje Folder-objekt kan innehålla pekare till flera Folders och Nodes och varje folder har en pekare till sin egen förälder. Detta ger att vi får en trädstruktur med kataloger och filer där vi kan navigera upp och ner i trädet via folder-pekarna. När en fil skapas så läggs innehållet i ett block. Innehållet har en viss storlek och ett block där filen börjar. Storleken och blocket lagras i en nod tillsammans med filnamnet, vilket gör att vi via noden kan hitta vårt filinnehåll i block-arrayen, se figur2. För att hålla koll på lediga platser i arrayen valde vi att skapa en blockMap i fileSystem-klassen. `blockMap` är en int-array som motsvarar varje block i MemBlockDevice. Om plats 0 i int-arrayen är 0 innebär det att plats 0 i block-arrayen är tom, och 1 innebär att platsen är använd.

2.4 Funktioner

Vi implementerade några privata funktioner i `FileSystem` för att användas i de funktioner som ska anropas via shell:

`vector<string> parsePath(string path)` - Delar upp en path i substrängar och lägger in dem i en `vector` och returnerar den.

`string getFileContent()` - En funktion för att låta användare skriva filinnehållet i en skapad fil. Funktionen tar input tills användaren trycker enter och sen skriver :q. Detta för att användaren ska kunna lägga in newlines i sin fil.

`string getFileFromBlock(string stringFromBlock)` - Tar innehållet från ett block, hittar filinnehållet och returnerar det som en sträng. Funktionen loopar igenom innehållet tills den hittar :q på en egen rad som markerar slutet av filinnehållet.

`bool hasSpace(size_t start, size_t size, int blockMap[])` - Kollar om en fil av storlek `size` får plats om man lägger in den på plats `start`.

`string getPathFromRoot(Folder* currentFolder)` - Returnerar en sträng med pathen från root-katalogen till `currentFolder`.

`Folder* getFolderFromPath(vector<string> path)` - Tar en path och returnerar en pekare till katalogen som path syftar till. Returnerar `nullptr` om pathen inte hittas.

`Node* getNodeFromPath(vector<string> path)` - Tar en path och returnerar en pekare till noden som pathen syftar till. Returnerar `nullptr` om pathen inte hittas.

`int writeFile(string fileContent, int size)` - Skriver `fileContent` som är `size` block stor till `mMemBlockDevice` i FileSystem. Returnerar `blockNr` om skrivningen lyckades, returnerar -1 om skrivningen misslyckades och returnerar -2 om ingen ledig plats hittades. `fileContent` delas upp i `size` bitar av storlek 512 byte och fylls upp med blanksteg om de är för små. `blockMap` och `hasSpace()` används för att hitta en tillräckligt stor ledig plats i block-arrayen och slutligen skrivs varje bit till ett eget block.

`string getFolderString(Folder* folder)` - Returnerar en sträng med alla folder paths rekursivt från folder separerade med \n.

`string getNodeString(Folder* folder)` - Returnerar en sträng med alla node paths rekursivt från folder separerade med \n.

`int restoreFile(string path, string fileContent)` - Skapar en fil med pathen `path` som innehåller `fileContent`.

De här funktionerna används av följande funktioner som anropas från shell när användaren skriver motsvarande kommando:
`create <path>` anropar `void createFile(string path)` som skapar en fil med pathen `path` och låter användaren skriva in innehållet i filen. För att avsluta input till filen ska användaren skriva in :q på en egen rad.

`cat <path>` anropar `void readFile(string path)` som skriver ut innehållet i filen med pathen `path` på skärmen.

pwd anropar `void printCurrentPath()` och skriver ut pathen till current working directory, alltså katalogen användaren befinner sig i.

`mkdir <path>` anropar `int createFolder(string path)` som skapar en katalog med pathen `path`.

`append <source> <destination>` anropar void `appendToFile(string source, string destination)` som tar sourceFilens innehåll och lägger till det i slutet av destFilens innehåll.

`mv <source> <destination>` anropar `void moveFile(string oldPath, string newPath)` och flyttar filen med path `oldPath` till `newPath`.

`rm <file-path>` anropar void `removeFile(string path)` som tar bort filen med pathen `path`.

cp <source> <destination> anropar `void copyFile(string source, string destination)` som kopierar sourceFilens innehåll till en ny plats i block-arrayen och skapar en ny Node med pathen destination som "pekar" till det nya innehållet.

`createImage <real-file>` anropar `void createImage(string realPath)` som skapar en ny fil med pathen `realPath` utanför det simulerade filsystem och innehåller alla kataloger och filer som finns i det nuvarande filsystemet. Filsystemet kan då återskapas med `restoreImage <real-file>`.

Filen är uppbyggd enligt:

```
folder:
<folder1>
<folder2>
...

nodes:
<file1>
<file1Content>
<file2>
<file2Content>
...
```

`restoreImage <real-file>` anropar `void restoreImage(string realFile)` som formaterar den nuvarande disken och återskapar ett filsystem från en fil där filen är uppbyggd enligt ovan.

`format` anropar `void formatDisk()` som skapar ett helt nytt tomt filsystem.

`cd <path>` anropar `string goToFolder(string path)` som sätter katalogen med pathen `path` som current working directory. Funktionen returnerar en sträng med pathen till den nya katalogen.

`ls` anropar `void listDir()` som listar kataloger och filer på skärmen. Funktionen lister filer och kataloger separat för att användaren ska vet vad som är filer och vad som är kataloger.

`<path>` innebär en giltig path med separatorn /. För att till exempel skapa en fil i katalogen mDir kan pathen `mDir/mFile` användas. Om ingen katalog skrivs innan filnamnet skapas filen i current working directory. Om pathen innehåller namn som inte finns, ska ett felmeddelande skrivas ut på skärmen.

3 Resultat

Vi har implementerat filsystem och implementerat funktioner för några kommandon. Filsystemet är implementerat i c++ och implementationen utgår ifrån ifrån ett redan halvt implementerat filsystem.

Alla kod för filsystemet finns i src-mappen och för att bygga filsystemet används kommandot `make all`. Den exekverbara filen `shell` kommer då att skapas i mappen bin. För att köra shell används kommandot `./bin/shell` förutsatt att användaren befinner sig i samma mapp som `Makefile`. När `shell` körs kan användaren mata in olika kommandon för att hantera filsystemet. För att se filsystemets kommandon kan användaren använda kommandot `help`.

4 Slutsats

I den här laborationen utgick vi från ett halvfärdigt filsystem och gjorde klart det med vissa funktioner som skulle finnas. Anvisningarna för laborationen var ganska öppna, så så länge de angivna funktionerna implementerades var det ganska öppet hur filsystemet skulle implementeras i övrigt. Vi valde att utgå från hjälpfilerna då det hade tagit längre tid att skapa och testa ett filsystem från grunden än att sätta sig sig in i de klasserna som redan fanns. Hjälpfilerna var inte så svåra att sätta sig in i och i de enda funktionerna vi använde var

```
int writeBlock(int blockNr, const std::string &strBlock)
```

och

```
Block readBlock(int blockNr) const
```

från `MemBlockDevice`-klassen.

Något vi var tvungna att tänka på var att allting skulle skrivas blockvis. Alltså att alla strängar som vi ville skriva till vårt filsystem var tvungna att vara 512 byte stora. Vi gjorde detta genom att fylla ut våra strängar med blanksteg och att markera slutet på filinnehållet med :q. Genom att markera filinnehållet med :q låter vi också användaren skriva i filen med \n tills användaren vill avsluta, istället för att avsluta input när användaren trycker på enter.

När vi skulle bygga upp filsystemet kändes det mest naturligt att implementera filsystemet i en trädstruktur där varje nod i trädet har koll på sin förälder och sina barn. Det gjorde det enkelt att kunna navigera i filträdet via pekare. Vi valde också att filerna skulle ingå i trädet, men ha en lite annan struktur. Vi implementerade en klass som liknar iNodes fast i våra noder har vi ett namn, filens storlek och filens adress i block-arrayen. För att hålla reda på vilka blocks om var lediga arrayen använda vi en int array där 0 markerade blocket som ledigt och 1 som använt.

Det viktigaste var att vi strukturerade upp hur vi ville att filsystemet skulle fungera innan vi började implementera det. När vi började att implementera funktionerna upptäckte vi ganska snart att mycket av koden kunde flyttas ut i egna funktioner och sen återanvändas vid flera kommando-anrop. När vi började flytta ut delar i egna funktioner blev allting mindre komplext och det var lättare att testa och implementera nya funktioner.

I vårt filsystem har vi valt att inte bry oss om fragmenteringsproblem och vi har också valt att använda kontinuerlig allokering av våra minnesblock, vilket innebär att om en fil har storleken tre block så allokeras tre block i rad i block-arrayen. Vi hade kunnat försöka undvika fragmentering genom att dela upp varje fil i flera block som inte behöver vara bredvid varandra, men det hade gjort implementationen mer komplex och det var inte heller nödvändigt för uppgiften.

Att implementera ett väldigt enkelt filsystem behöver inte vara så svårt, men om man vill ha mer funktionalitet kommer också komplexiteten att öka. Har man en bra grund-struktur kommer det att vara enklare att lägga in ny funktionalitet, men i ett filsystem vill man helst att prestandan ska vara bra och man vill också undvika till exempel fragmentering. Något som också är bra att tänka på att är att det inte är så lätt att testa ett filsystem som är tänkt att användas av användare. För varje funktion som läggs till behövs det testas ännu mer för att garantera att filsystemet fungerar.