Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 2 (IR3OS2)
Nastavnik: prof. dr Dragan Milićev
Školska godina: 2009/2010

Projekat iz predmeta Operativni sistemi 2

Milan Branković 119/07

Jun 2010

```
#ifndef _PART_H_
#define _PART_H_


      typedef unsigned long BlockSize;
      typedef unsigned long BlockNo;
      typedef unsigned long ClusterNo;
      typedef unsigned long ClusterSize;

      class PartitionImpl;

      class Partition {
      public:
            Partition(char *);
            virtual BlockSize getBlockSize() const;
            virtual BlockNo getStartBlock() const;
            virtual BlockNo getNumOfBlocks() const;

            virtual long getNumOfSectors() const;
            virtual long getNumOfCylinders() const;
            virtual long getNumOfHead() const;

            virtual int readBlock(BlockNo, char *buffer);
            virtual int writeBlock(BlockNo, const char *buffer);
            virtual ClusterSize getClusterSize() const;
            virtual int setClusterSize(BlockNo);
            virtual int setClusterOffset(BlockNo);
            virtual BlockNo getClusterOffset() const;
            virtual int readCluster(ClusterNo, char *buffer);
            virtual int writeCluster(ClusterNo, const char *buffer);

            virtual ~Partition();
      private:
            PartitionImpl *myImpl;
      };

      #endif
```

```
// file: PartInfo.h

#ifndef _PART_INFO_H_
#define _PART_INFO_H_

#include "part.h"

class PartInfo{
public:
      PartInfo(Partition* p);
      ~PartInfo();

      void setLetter (char c);
      char getLetter () const;

      void setFat1 (BlockNo );
      BlockNo getFat1 () const;

      void setFat2 (BlockNo );
      BlockNo getFat2 () const;

      void setRoot(BlockNo);
      BlockNo getRoot() const;

      Partition* getPart() const;

      int loadFat(BlockNo, char* fat);
      int storeFat(BlockNo, const char* fat);

      int loadRoot(BlockNo, char* root);
      int storeRoot(BlockNo, const char* root);

private:
      Partition* part;
      char letter;

      BlockNo fat1_start, fat2_start;
      BlockNo root_start;
      ClusterNo frst_free_cluster;

};

#endif
```

```
// File: fs.h

#ifndef _FS_H_
#define _FS_H_

typedef unsigned long BytesCnt;
typedef unsigned long EntryNum;

const unsigned int ENTRYCNT=64;
const unsigned int FNAMELEN=8;
const unsigned int FEXTLEN=3;

struct Entry {
  char name[FNAMELEN];
  char ext[FEXTLEN];
  char attributes;
  char reserved[14];
  unsigned long size;
  unsigned int firstCluster;
};

typedef Entry Directory[ENTRYCNT];

class KernelFS;
class Partition;
class File;
class FS {
public:

  FS ();
 ~FS ();

  static char mount(Partition* partition);
  static char unmount(char part);
  static char format(char part);
  static BytesCnt freeSpace(char part);
  static BytesCnt partitionSize(char part);
  char doesExist(char* fname);
  File* open(char* fname, char mode);
  char deleteFile(char* fname);

  char createDir(char* dirname);
  char readDir(char* dirname, EntryNum, Directory &);
  char deleteDir(char* dirname);
  char* pwd();
  char  cd(char* dirname);
private:
  KernelFS *myImpl;
};

#endif
```

```
// File: file.h

#ifndef _FILE_H_
#define _FILE_H_

#include "fs.h"

class KernelFile;

class File {
public:

  ~File();

  char write (BytesCnt, char* buffer);
  BytesCnt read (BytesCnt, char* buffer);
  char seek (BytesCnt);
  BytesCnt getFileSize ();
  char eof ();
  char truncate ();
  void close ();

private:
  friend class FS;
  friend class KernelFS;
  friend class KernelFile;
  File ();  //objekat fajla se moze kreirati samo otvaranjem

  KernelFile *myImpl;
};

#endif
```

```
// file: KernelFS.h

#ifndef _KERNEL_FS_H_
#define _KERNEL_FS_H_

#include "file.h"
#include "fs.h"
#include "kernelfile.h"
#include "kernelfs.h"
#include "lista.h"
#include "part.h"
#include "PartInfo.h"
#include "Monitor.h"
#include <windows.h>
#include <iostream>
using namespace std;

#define MAX_PATH_LENGTH 100

class KernelFS {

      static PartInfo* partArray[26];
      static int partLetter[26]; //koje je slovo zauzeto

      static unsigned long BootSectorSizeByte;
      static unsigned long FatSizeByte;
      static unsigned long RootSizeByte;

      char* pwdName;

      static Lista listOfAllFiles;// lista svih fajlova

      friend class PartInfo;
      friend class KernelFile;
      friend class Monitor;

      static HANDLE mutexFile, mutexDir; //za medjusobno iskljucenje
      static HANDLE mutexFAT, mutex;

public:
  KernelFS ();
  ~KernelFS ();

  static char mount(Partition* partition); //montira particiju
                                        // vraca dodeljeno slovo
  static char unmount(char part);  //demontira particiju oznacenu datim
               // slovom vraca 0 u slucaju neuspeha i 1 u slucaju uspeha
  static char format(char partL); //particija zadatu slovom se formatira sa
               // FAT16; vraca 0 u slucaju neuspeha i 1 u slucaju uspeha

  static BytesCnt freeSpace(char part); // vraca ukupan broj bajtova u
                        // slobodnim klasterima particije sa zadatim slovom
  static BytesCnt partitionSize(char part); // vraca ukupan broj bajtova
        //koji se koriste za smestanje podataka particije sa zadatim slovom

  char doesExist(char* fname); //argument je puna staza fajla
  File* open(char* fname, char mode);
  char deleteFile(char* fname);
```

```
  char createDir(char* dirname);
  char readDir(char* dirname, EntryNum, Directory&);
         //drugim argumentom se zadaje broj ulaza od kog se pocinje citanje
  char deleteDir(char* dirname);
  char* pwd();               //tekuci direktorijum
  char  cd(char* dirname);   //promena tekuceg direktorijuma



  /* metoda koja pravi apsolutnu putanju od relativne */
      void createApsolutePath (char* src, char* result);

      /* metoda koja dohvata roditeljsku putanju */
      void getParentPath(char* src, char* result);

      /*metoda koja dohvata samo ime direktorijuma */
      void getName(char* src, char* result);

      /* metoda koja ispituje da li treba da se montira u root */
      bool inRoot(char* src);

      /* metoda koja broji koliko ima kosih crta u putanji */
      int countNames(char* path);

      /* metoda koja vraca klaster poslednjeg montiranog dir-a  */
      unsigned int find( char* path, char* root, ClusterNo cs, PartInfo*
part);

      /* metoda koja cita jedan Entry iz klastera */
      int readEntry (char* cls, int num, Entry& ent);

      /* metoda koja secka na ime fajla i extenziju */
      void cutName(char* src, char* name, char* ext);

      /* metoda koja pravi apsolutnu putanju fajla */
      void createApsoluteFileName (char* src, char* result);

};

#endif
```

```
//file: kernelfile.h

#ifndef _KERNEL_FILE_H_
#define _KERNEL_FILE_H_

#include "file.h"
#include "PartInfo.h"
#include "Monitor.h"
#include <iostream>
using namespace std;

class Monitor;
class KernelFile {
public:

      KernelFile ( PartInfo* pi);
  ~KernelFile();
  KernelFile(const KernelFile* kf);

  char write (BytesCnt, char* buffer);
  BytesCnt read (BytesCnt, char* buffer);
  char seek (BytesCnt);
  BytesCnt getFileSize ();
  char eof ();
  char truncate ();
  void close ();

  void setFirstCls(ClusterNo );
  int setCurrentPos(BytesCnt);
  BytesCnt getEOF();
  ClusterNo getNextCls(ClusterNo , char* );
  void setFile(File* );
  char getMode();
  void setMode(char);

private:
  friend class FS;
  friend class KernelFS;

  File* file;
  PartInfo* p;
  char* cls; //klaster u kome nam je kursor
  char mode; // r, w, a
  BytesCnt currentPos, endOfFile;
  ClusterNo firstCls;
  ClusterSize clsSize;
  int inCls,numOfCls;
  Monitor fileMonitor;
};

#endif
```

```
// file : Lista.h

      #ifndef _LISTA_H_
      #define _LISTA_H_

      #include <iostream>
      using namespace std;

      #define DUZINA 50

      class KernelFile;

      class Lista {
      protected:
        struct Elem {                // ELEMENT LISTE:
          char* ime;
            KernelFile* kf;
            Elem* sled;
            Elem (char* i, KernelFile* k = 0, Elem* s=0){
                  ime = new char[DUZINA];strcpy( ime, i); kf = k; sled = s;
            }
        };
        Elem *prvi, *posl;
        int duz;
      private:
        void brisi ();
      public:
        Lista () { prvi = posl = 0; duz = 0; }
        Lista (char* i, KernelFile* kf) { prvi = posl = new Elem (i,

      kf); duz = 1; }
        ~Lista () { brisi (); }
        int duzina () const { return duz; }
        void naPocetak (char* ime, KernelFile* kf);
        KernelFile* uzmi (char* ime);
        KernelFile* uzmiSaKraja();
        int izbaci(char* );

        friend ostream& operator<< (ostream&, const Lista&);
      };

      #endif
```

```
// file: PartInfo.cpp

#include "PartInfo.h"
#include "kernelfs.h"
#include "part.h"

PartInfo::PartInfo(Partition* p){
            this->part = p;
}

PartInfo::~PartInfo(){
      delete part;
}

void PartInfo::setLetter(char c){
      this->letter = c;
}

char PartInfo::getLetter() const{
      return letter;
}

void PartInfo::setFat1(BlockNo n){
      this->fat1_start = n;
}

void PartInfo::setFat2(BlockNo n){
      this->fat2_start = n;
}
BlockNo PartInfo::getFat1() const{
      return fat1_start;
}
BlockNo PartInfo::getFat2() const{
      return fat2_start;
}
Partition* PartInfo::getPart() const{
      return part;
}
BlockNo PartInfo::getRoot() const{
      return root_start;
}
void PartInfo::setRoot(BlockNo start){
      root_start = start;
}

int PartInfo::loadFat(BlockNo start, char* fat){
      Partition * p = this->getPart();
            BlockSize blockSize = p->getBlockSize();

            int numOfBlocks1 = KernelFS::FatSizeByte / blockSize;
            if (KernelFS::FatSizeByte % blockSize) numOfBlocks1++;

            for(int j = 0; j<numOfBlocks1; j++)
                  if(p->readBlock( start + j , fat + j*blockSize) == 0)
return 0;

            return 1;
}
```

```
int PartInfo::storeFat(BlockNo start, const char* fat){
      Partition * p = this->getPart();
      BlockSize blockSize = p->getBlockSize();

      int numOfBlocks1 = KernelFS::FatSizeByte / blockSize;
      if (KernelFS::FatSizeByte % blockSize) numOfBlocks1++;

      for(int j = 0; j<numOfBlocks1; j++)
                  if(p->writeBlock( start + j , fat + j*blockSize) == 0)
return 0;

      return 1;
}

int PartInfo::loadRoot(BlockNo start, char* root){
      Partition * p = this->getPart();

            BlockSize blockSize = p->getBlockSize();

            int numOfBlocks1 = KernelFS::RootSizeByte / blockSize;
            if (KernelFS::RootSizeByte % blockSize) numOfBlocks1++;

            for(int j = 0; j<numOfBlocks1; j++)
                  if(p->readBlock( start + j, root + j*blockSize)== 0) return
0;

            return 1;
}

int PartInfo::storeRoot(BlockNo start, const char* root){
      Partition * p = this->getPart();
      BlockSize blockSize = p->getBlockSize();

      int numOfBlocks1 = KernelFS::RootSizeByte / blockSize;
      if (KernelFS::RootSizeByte % blockSize) numOfBlocks1++;

      for(int j = 0; j<numOfBlocks1; j++)
            if(p->writeBlock( start + j , root + j*blockSize) == 0) {
                  return 0;
            }
      return 1;
}
```

```
#include "fs.h"
#include "kernelfs.h"

FS::FS(){
	myImpl = new KernelFS();
}

FS::~FS(){
	delete myImpl;
}

char FS::mount(Partition* partition){
	return KernelFS::mount(partition);
}

char FS::unmount(char part){
	return KernelFS::unmount(part);
}

char FS::format(char part){
	return KernelFS::format(part);
}

BytesCnt FS::freeSpace(char part){
	return KernelFS::freeSpace(part);
}

BytesCnt FS::partitionSize(char part){
	return KernelFS::partitionSize(part);
}


char FS::doesExist(char* fname){
	return myImpl->doesExist(fname);
}
File* FS::open(char* fname, char mode){
	return myImpl->open(fname, mode);
}
char FS::deleteFile(char* fname){
	return myImpl->deleteFile(fname);
}
char FS::createDir(char* dirname){
	return myImpl->createDir(dirname);
}
char FS::readDir(char* dirname, EntryNum num, Directory &dir){
	return myImpl->readDir(dirname, num, dir);
}
char FS::deleteDir(char* dirname){
	return myImpl->deleteDir(dirname);
}
char* FS::pwd(){
	return myImpl->pwd();
}

char FS::cd(char* dirname){
	return myImpl->cd(dirname);
}
```

```
// file: file.cpp

#include "file.h"
#include "kernelfile.h"

File::File(){myImpl = 0;}
File::~File(){}
char File::write (BytesCnt cnt, char* buffer){
      return myImpl->write(cnt, buffer);
}
BytesCnt File::read (BytesCnt cnt, char* buffer){
      return myImpl->read(cnt, buffer);
}
char File::seek (BytesCnt cnt){
      return myImpl->seek(cnt);
}
BytesCnt File::getFileSize (){
      return myImpl->getFileSize();
}
char File::eof (){
      return myImpl->eof();
}
char File::truncate (){
      return myImpl->truncate();
}
void File::close (){
      myImpl->close();
}
```

```
#include "kernelfs.h"
#include "fs.h"
#include "part.h"
#include "file.h"
#include "kernelfile.h"
#include "lista.h"
#include <windows.h>
#include <cstring>
#include <iostream>
using namespace std;


PartInfo* KernelFS::partArray[]={0};
int KernelFS::partLetter[]={0};

unsigned long KernelFS::BootSectorSizeByte=512;
unsigned long KernelFS::FatSizeByte=131072;
unsigned long KernelFS::RootSizeByte=16384;

HANDLE KernelFS::mutex = CreateMutex( NULL,FALSE, NULL);
HANDLE KernelFS::mutexFile = CreateMutex( NULL,FALSE, NULL);
HANDLE KernelFS::mutexDir = CreateMutex( NULL,FALSE, NULL);
HANDLE KernelFS::mutexFAT = CreateMutex( NULL,FALSE, NULL);
Lista KernelFS::listOfAllFiles = Lista();

KernelFS::KernelFS() {
      pwdName = new char[MAX_PATH_LENGTH];
      pwdName[0] = '\0';

}

KernelFS::~KernelFS(){}

char KernelFS::mount(Partition *partition){
      WaitForSingleObject(mutex, INFINITE);
      if( partition != 0){
        int i;
        for( i =0; i<26; i++){
              if(partLetter[i]==0){
                  partLetter[i] = 1;
                  partArray[i] = new PartInfo(partition);
                  partArray[i]->setLetter('A' + i);

                  ReleaseMutex(mutex);
                  return partArray[i]->getLetter();
              }
        }

      }

      ReleaseMutex(mutex);
      return '\0';
}

char KernelFS::unmount(char part){
      WaitForSingleObject(mutex, INFINITE);
      if( 'A' <= part && part <= 'Z'){
        for(int i = 0; i<26; i++){
              if(partLetter[i] != 0){
```

```
            char letter = partArray[i]->getLetter();
                  if(part == letter){

                      partLetter[i] = 0;

                      Partition* p = partArray[i]->getPart();
                      delete p;

                      partArray[i] = 0;

                      ReleaseMutex(mutex);
                      return '1';
                  }
            }
        }
      }

      ReleaseMutex(mutex);
      return '0';


}

char KernelFS::format(char partL){
      WaitForSingleObject(mutex, INFINITE);
      if( 'A' <= partL && partL <= 'Z'){
            for(int i  = 0; i<26 ; i++){
                  if(partLetter[i]!=0){
                        if(partArray[i]->getLetter() == partL){
              Partition* part = partArray[i]->getPart();
              BlockNo bs = part->getBlockSize();
              BlockNo nb = part->getNumOfBlocks();
              ClusterNo cs = part->getClusterSize();
            BlockNo co = part->getClusterOffset();
            BlockNo start = part->getStartBlock();
            long nh = part->getNumOfHead();
            long ns = part->getNumOfSectors() * part->getNumOfCylinders();

            long bytesInSector = nb * bs / ns;
            long numSectorsInClusters = cs * bs / ns;
            long brSkrivenihSektora = 2*cs/ns;
            int numOfSectorsPerCylinder = part->getNumOfSectors();

            char buff[512]; //KernelFS::BootSectorSizeByte
            if(buff == 0) return '0';

            *(buff + 0x00) = '0';
            *(buff + 0x01) = '0';
            *(buff + 0x02) = '0';


            strcpy((buff + 0x03) , "IR3OS2  ");


            *(buff + 0x0b)=((bytesInSector & 0xFF00) >> 8);
            *(buff + 0x0c)=(bytesInSector & 0xFF);

            *(buff + 0x0d)=(numSectorsInClusters & 0xFF);
```

```
	*(buff + 0x0e)='1';
	*(buff + 0x0f) = '0';

	*(buff + 0x10)='2';

	*(buff + 0x11)=(0x200 & 0xFF00) >> 8;
	*(buff + 0x12)=(0x200 & 0xFF);

	if (ns == 0){
			*(buff + 0x20) = ((ns & 0xFF000000) >> 24);
			*(buff + 0x21) = ((ns & 0xFF0000) >> 16);
			*(buff + 0x22) = ((ns & 0xFF00) >> 8);
			*(buff + 0x23) = (ns & 0xFF);

			*(buff + 0x11) = '0';
			*(buff + 0x12) = '0';

		}
		else if (ns>0xFFFF){
			*(buff + 0x11) = ((ns & 0xFF00) >> 8);
			*(buff + 0x12) = (ns & 0xFF);

			*(buff + 0x20) = '0';
			*(buff + 0x21) = '0';
			*(buff + 0x22) = '0';
			*(buff + 0x23) = '0';
		}


	*(buff + 0x15) = (char) 0xf8;

unsigned long mod = (nb - co) % cs;
unsigned long clusterNumber = (nb - co)/cs;

if (mod != 0)     clusterNumber++;

clusterNumber = clusterNumber + 2; // First two virtual entries

long FATsize = clusterNumber*2 / bytesInSector;


	*(buff + 0x16) = ((FATsize & 0xFF00) >> 8);
	*(buff + 0x17) = (FATsize & 0xFF);

	*(buff + 0x18) = ((numOfSectorsPerCylinder & 0xFF00) >> 8);
	*(buff + 0x19)  = (numOfSectorsPerCylinder & 0xFF);

	*(buff + 0x1a) = (nh & 0xFF00) >> 8;
	*(buff + 0x1b) = (nh & 0xFF);

	*(buff + 0x1c) = ((brSkrivenihSektora & 0xFF000000) >> 24);
	*(buff + 0x1d) = ((brSkrivenihSektora & 0xFF0000) >> 16);
	*(buff + 0x1e) = ((brSkrivenihSektora & 0xFF00) >> 8);
	*(buff + 0x1f) = (brSkrivenihSektora & 0xFF);

	*(buff + 0x24) = '1';
	*(buff + 0x25) = '0';

	*(buff + 0x26) = (char) 0x29;
```

```
	*(buff + 0x27) = '0';
	*(buff + 0x28) = '0';
	*(buff + 0x29) = '0';
	*(buff + 0x2a) = '0';

	strcpy((buff + 0x2b) , "DATA       ");

	strcpy((buff + 0x36) , "FAT16   ");

	for (int j=0x3E; j<0x1FE; j++) buff[j] = 0;


	*(buff + 0x1fe) = (char)0x55;
	*(buff + 0x1ff) = (char)0xAA;

	// writeBlock

	mod = sizeof(buff)%bs;
	unsigned long n = sizeof(buff)/bs;

	if (mod != 0) n++;

	for(unsigned long j = 0; j<n; j++){
		part->writeBlock(start + j, buff + j*bs);
	}

// KRAJ BOOT SEKTORA

	//insert fat1 & fat2
	char fat1[131072];//KernelFS::FatSizeByte
	if(fat1 == 0 ) return '0';

	fat1[0]=(char)0xf8;
	fat1[1]=(char)0;
	fat1[2]=(char)0xff;
	fat1[3]=(char)0xff;
	for(unsigned int j=4;j<KernelFS::FatSizeByte;j++) fat1[j]=(char)0;

	// writeBlock
	start+=n;
	partArray[i]->setFat1(start);

	mod = sizeof(fat1)%bs;
	n = sizeof(fat1)/bs;
	//start = start+ KernelFS::BootSectorSizeByte;

	if (mod != 0) n++;

	for(unsigned long j = 0; j<n; j++){
		part->writeBlock(start + j, fat1 + j*bs);
	}

	start+=n;
	partArray[i]->setFat2(start);
	//start = start+ KernelFS::BootSectorSizeByte + KernelFS::FatSizeByte;

	for(unsigned long j = 0; j<n; j++){
		part->writeBlock(start + j, fat1 + j*bs);
```

```
            }

      //KRAJ FATA
            char root[16384]; //KernelFS::RootSizeByte
            if (root == 0) return '0';

            *(root + 0x00) = partL;
            *(root + 0x01) = ':';
            *(root + 0x02) = ' ';
            *(root + 0x03) = ' ';
            *(root + 0x04) = ' ';
            *(root + 0x05) = ' ';
            *(root + 0x06) = ' ';
            *(root + 0x07) = ' ';

            *(root + 0x08) = (char) 0;;
            *(root + 0x09) = (char) 0;;
            *(root + 0x0a) = (char) 0;;

            *(root + 0x0b) = (char) 0x08;

            for (int j=0x0c; j<0x20; j++) root[j] = (char) 0;
            for(unsigned long j = 0x20; j<KernelFS::RootSizeByte; j++) root[j] =
(char) 0;

            // writeBlock
            start+=n;
            partArray[i]->setRoot(start);

            mod = sizeof(root)%bs;
            n = sizeof(root)/bs;
            //start = start+ KernelFS::BootSectorSizeByte +
2*KernelFS::FatSizeByte;

            if (mod != 0) n++;

            for(unsigned long j = 0; j<n; j++){
                  part->writeBlock(start + j, root + j*bs);
            }

      //KRAJ ROOT

      //CLUSTER_OFFSET_SIZE
            BlockNo
offSet=(KernelFS::BootSectorSizeByte+2*KernelFS::FatSizeByte+KernelFS::RootSizeByte
)/bs;
            part->setClusterSize(2);
            part->setClusterOffset(offSet);

            ReleaseMutex(mutex);
            return 1;
      }}}}

      ReleaseMutex(mutex);
      return '0';
}

BytesCnt KernelFS::freeSpace(char part){
      if (part<'A' || part > 'Z') return -1;
```

```
	for(int i = 0; i<26 ; i++){
		if(partLetter[i]!=0){
			if(partArray[i]->getLetter() == part){
		BytesCnt freeSpaceBytes = 0;

		Partition* p = partArray[i]->getPart();

		//num of fat entrys
		BlockNo numOfBlocks = p->getNumOfBlocks();
		BlockNo clusterOffset = p->getClusterOffset();
		ClusterSize clusterSize = p->getClusterSize();
		BlockSize blockSize = p->getBlockSize();

		int mod = (numOfBlocks - clusterOffset) % clusterSize;
		ClusterNo clusterNumber = (numOfBlocks - clusterOffset)/clusterSize;

		if (mod != 0) clusterNumber++;

//DOVUCI FAT
		int numOfBlocks1 = KernelFS::FatSizeByte / blockSize;
		if (KernelFS::FatSizeByte % blockSize) numOfBlocks1++;

		BlockNo start = partArray[i]->getFat1();

		char* fat = new char[numOfBlocks * blockSize];
		for(int j = 2; j<numOfBlocks1; j++)
			p->readBlock( start , fat + j*blockSize);

		for(ClusterNo j = 2; j<clusterNumber; j++)
			if(fat[j] == 0) freeSpaceBytes++;

		delete[] fat;

		freeSpaceBytes = 2 * freeSpaceBytes * clusterSize * blockSize;

		return freeSpaceBytes;
	}
			}
	}
	return 0;
}

BytesCnt KernelFS::partitionSize(char part){
	if (part<'A' || part > 'Z') return -1;

	for(int i = 0; i<26; i++){
		if(partLetter[i]!=0){
			if(partArray[i]->getLetter() == part){
				Partition* p = partArray[i]->getPart();
				BlockNo numOfBlocks = p->getNumOfBlocks();
				BlockNo clusterOffset = p->getClusterOffset();
				ClusterSize clusterSize = p->getClusterSize();
				BlockSize blockSize = p->getBlockSize();

				int mod = (numOfBlocks - clusterOffset) % clusterSize;
				int clusterNumber = (numOfBlocks -
clusterOffset)/clusterSize;

				if (mod != 0) clusterNumber++;
```

```
                        int partitionSize = clusterNumber * clusterSize *
blockSize;

                  return partitionSize;
      }
            }
      }
  return 0;
}


char* KernelFS::pwd(){
      return this->pwdName;
}


char KernelFS::doesExist(char* fname){
      char path[MAX_PATH_LENGTH];
      createApsolutePath(fname, path);

      int numNames = countNames(path);

      PartInfo* part = 0;
      for(int i = 0; i<26; i++){
            if(partArray[i] != 0 && partArray[i]->getLetter() == path[0]) {
                  part = partArray[i];
                  break;
            }
      }

      if(part == 0) return 0;
      if( strlen(path) == 3 || strlen(path) == 2) return 1;

      unsigned long cs = part->getPart()->getClusterSize() * part->getPart()-
>getBlockSize();
      char
            *root = new char[KernelFS::RootSizeByte];

      if( root == 0) return 0;

      if (part->loadRoot(part->getRoot(), root) ==0 ) return 0;


      char * pomRoot = &root[32];
      Entry* nizEnt, *nizEnt1;


      char pom[MAX_PATH_LENGTH], pom1[FNAMELEN];
      unsigned int first_cls = 0;
      Partition* p = part->getPart();
      int  j = 3;

      nizEnt = (Entry *) pomRoot;

      for( int i = 0; i< countNames(path); i++){
            if(i == 0) {
                  int k= 0 , ulaz = 0;
                  while(path[j] != '\\' && path[j] != '\0' && path[j]!= '.'){
                        pom[k++] = path[j++];
```

```
	}
	pom[k] = '\0';
	numNames--;

	for( k = 32; k<KernelFS::RootSizeByte; k+=sizeof(Entry), ulaz++){
		strncpy(pom1, root + k, FNAMELEN);
		if ( strcmp(pom, pom1) == 0) {
			if(numNames == 0) {
				delete [] root;

				pomRoot = 0;
				nizEnt = 0;
				nizEnt1 = 0;

				return 1;

			}
			first_cls = nizEnt[ulaz].firstCluster;

			break;
		}
	}
	if( k == KernelFS::RootSizeByte) return 0;
} else{

	char *ccls = new char[cs];
	if (ccls == 0) return 0;

	if( p->readCluster(first_cls, ccls) == 0) return 0;
	nizEnt1 = (Entry *) ccls;

	int k= 0, ulaz = 0;
	j++;
	while(path[j] != '\\' && path[j] != '\0' && path[j]!= '.'){
		pom[k++] = path[j++];
	}
	pom[k] = '\0';
	numNames--;

	for( k = 0; k<cs; k+=sizeof(Entry), ulaz++){
		strncpy(pom1, ccls + k, FNAMELEN);
		if ( strcmp(pom, pom1) == 0) {
			if(numNames == 0) {
				delete [] root;

				pomRoot = 0;
				nizEnt = 0;
				nizEnt1 = 0;

				return 1;

			}

			first_cls = nizEnt1[ulaz].firstCluster;

			break;
		}
	}
	if( k == cs) return 0;
```

```
                delete [] ccls;
            }
        }

        pomRoot = 0;
        nizEnt = 0;
        nizEnt1 = 0;

        delete [] root;

        return 0;
}

File* KernelFS::open(char* fname, char mode){

        WaitForSingleObject(mutexFile, INFINITE);

        if( mode != 'r' && mode != 'w' && mode != 'a') return 0;

        char tmp[MAX_PATH_LENGTH];
        createApsolutePath(fname, tmp);

        PartInfo* p = 0;
        for(int i = 0; i<26; i++){
            if(partArray[i] != 0 && partArray[i]->getLetter() == tmp[0]) {
                p = partArray[i];
                break;
            }
        }

        if(p == 0) return 0;

        File* tempFile = 0;
        KernelFile* ret = 0;

        int nr = 0;

        switch( mode){
            case 'r' :
                if(doesExist(tmp) == 0) return 0;

                ret = listOfAllFiles.uzmi(tmp);
                if(ret == 0) return 0;

                nr = ret->fileMonitor.startRead();

                if( nr > 1) {
                    KernelFile* novi;
                    novi = new KernelFile(ret);
                    ret = novi;
                    ret->setCurrentPos(0);
                    novi = 0;
                }

                ret->setMode('r');

                tempFile = new File();
                ret->setFile(tempFile);
```

```
                ret->seek(0);

                break;
          case 'w' :
                if(doesExist(tmp) == 0){ //ako ne postoji, pravimo ga
                      unsigned long cs  = p->getPart()->getClusterSize() * p-
>getPart()->getBlockSize(),first_cls;
                      char pom[MAX_PATH_LENGTH];
                      char name[FNAMELEN], ext[FEXTLEN];
                      char
                            *fat = new char[KernelFS::FatSizeByte],
                            *root = new char[KernelFS::RootSizeByte];

                      if (fat == 0) return 0;
                      if( root == 0) return 0;

                      WaitForSingleObject(mutexFAT, INFINITE);
                      if (p->loadRoot(p->getRoot(), root) ==0 ) return 0;
                      if (p->loadFat(p->getFat1(), fat) == 0) return 0;
                      ReleaseMutex(mutexFAT);

                      // root, FAT_first_free_cluster = FF, cluster

                      //rezervisemo klaster
                      int i;
                      for(i = 4; i<KernelFS::FatSizeByte; i+=2){
                            if(fat[i] == 0 && fat[i+1]== 0) {
                                  fat[i] = (char)0xFF;
                                  fat[i+1] = (char)0xFF;
                                  break;
                            }
                      }

                      if( inRoot(tmp)){
                            //ako pravimo u root- u

                            //trazimo prvi slobodan ulaz u root-u
                            unsigned int j;
                            for(j = 32; j<KernelFS::RootSizeByte; j+=
sizeof(Entry) ){

                                  if( root[j + 11] == 0xE5 ||
                                        root[j + 11] == 0xe5 ||
                                        root[j + 11] == 0x00 ||
                                        root[j + 11] == '0') break;
                            }

                            if (j == KernelFS::RootSizeByte) return 0;

                            //pravimo novi Entry za fajl
                            //pravimo novi Dir za fajl sa jednim Enrty-em,
                            //posto ne radi konverziju (char* ) Entry
                            Directory newDir;

                            getName(tmp, pom); //celo ime sa ext
                            cutName(pom, name, ext);
                            strncpy(newDir[0].name, name , FNAMELEN);
                            strncpy(newDir[0].ext, ext, FEXTLEN);
```

```
                              newDir[0].attributes = 0x01;  //uzeo sam zato sto mi
je 0x00 oznacava slobodan ulaz
                              for (int j=0; j<14; j++) (newDir[0].reserved)[j] = 0;
                              newDir[0].size = 0;
                              newDir[0].firstCluster = i/2;

                              //upis u root
                              char* nesto = (char* ) newDir;
                              for(int ii =0; ii<sizeof(Entry); ii++)root[j + ii] =
nesto[ii];
                              if (p->storeRoot(p->getRoot(), root) ==0 ) return 0;

                              nesto = 0;
                              delete [] root;

                        }else {
                              //ako nije montiranje u root-u

                              char *cls  = new char[cs];
                              if( cls == 0) return 0;

                              char pom1[FNAMELEN + FEXTLEN];

                              if ((first_cls = find(tmp, root, cs, p)) == 0) return
0;
                              if( p->getPart()->readCluster(first_cls, cls) == 0)
return 0;

                              delete [] root;

                              //trazimo koji je ulaz u klasteru slobodan
                              int  ii, ulaz = 2;
                              for(ii = 2*sizeof(Entry); ii<cs; ii+= sizeof(Entry),
ulaz++ ){

                                    if( cls[ii + 11] == 0xE5 ||
                                          cls[ii + 11] == 0xe5 ||
                                          cls[ii + 11] == 0x00 ||
                                          cls[ii + 11] == '0') break;
                              }

                              if (ii == cs) return 0;


                              Entry newEnt;

                              getName(tmp, pom1);
                              cutName(pom1, name, ext);
                              strncpy(newEnt.name, name , FNAMELEN);
                              strncpy(newEnt.ext, ext, FEXTLEN);
                              newEnt.attributes = 0x01;
                              for (int j=0; j<14; j++) newEnt.reserved[j] = 0;
                              newEnt.size = 0;
                              newEnt.firstCluster = i/2;


                              //upisujemo u klaster novi Entry i vracamo na
particiju
                              Entry* nizEnt;
```

```
                              nizEnt = (Entry *) cls;
                              nizEnt[ulaz] = newEnt;
                              cls = (char *) nizEnt;

                              if( p->getPart()->writeCluster(first_cls, cls) == 0)
return 0;

                              delete [] nizEnt;


                        }

                        //vracanje FAT-a na particiju
                        if (p->storeFat(p->getFat1(), fat) == 0) return 0;
                        if (p->storeFat(p->getFat2(), fat) == 0) return 0;

                        delete [] fat;

                        ret = new KernelFile(  p);
                        ret->setFirstCls(i/2);

                        //ubacujemo u listu svih fajlova
                        listOfAllFiles.naPocetak(tmp, ret);

                  }else {// ako fajl postoji

                        ret = listOfAllFiles.uzmi(tmp);
                        if (ret == 0) return 0;
                  }

                  ret->fileMonitor.startWrite();

                  ret->setMode('w');
                  tempFile = new File();
                  ret->setFile(tempFile);

                  //ret->seek(0);
                  //ret->truncate();

                  break;
            case 'a' :
                  /* proverimo da li fajl postoji, ako da nadjemo ga u listi,
                  i postavimo mu pokazivac na kraj fajla */

                  if(doesExist(tmp) == 0) return 0;


                  ret = listOfAllFiles.uzmi(tmp);
                  if( ret == 0) return 0;

                  ret->fileMonitor.startApend();

                  tempFile = new File();
                  ret->setFile(tempFile);

                  ret->setMode('a');
                  ret->setCurrentPos(ret->getEOF());

                  break;
```

```
		default : return 0;
	}
	ReleaseMutex(mutexFile);
	return ret->file;
}



char KernelFS::deleteFile(char* fname){
	char tmp[MAX_PATH_LENGTH];
	createApsolutePath(fname, tmp);

	if (doesExist(tmp) == 0) return 0;

	KernelFile* ret = 0;
	ret = listOfAllFiles.uzmi(tmp);
	if( ret == 0) return 0;
	if (ret->getMode() != 'c') return 0;


	PartInfo* p = 0;
	for(int i = 0; i<26; i++){
		if(partArray[i] != 0 && partArray[i]->getLetter() == tmp[0]) {
			p = partArray[i];
			break;
		}
	}

	if(p == 0) return 0;

	char
		*fat = new char[KernelFS::FatSizeByte],
		*root = new char[KernelFS::RootSizeByte];

	if (fat == 0) return 0;
	if( root == 0) return 0;

	WaitForSingleObject(mutexFile, INFINITE);

	if (p->loadRoot(p->getRoot(), root) ==0 ) return 0;
	if (p->loadFat(p->getFat1(), fat) == 0) return 0;

	char pom[MAX_PATH_LENGTH];
	getName( tmp, pom);
	char ime[FNAMELEN], ext[FEXTLEN];

	cutName(pom, ime, ext);

	if(inRoot(tmp)){

		unsigned int j;
		for(j = 32; j<KernelFS::RootSizeByte; j+= sizeof(Entry) ){

			if( root[j + 11] == 0x01 &&
				strncmp(root + j, ime, FNAMELEN) == 0) break;
		}

		if (j == KernelFS::RootSizeByte) return 0;
```

```
            //brisemo iz root-a i fata
            for( int i = 0; i<sizeof(Entry); i++) root[j + i] = (char) 0;

            ClusterNo first = ret->firstCls;
            for(unsigned long i = 4; i<KernelFS::FatSizeByte &&
                  first != 0xFFFF; i+=2){
                  if(i == first * 2){
                        first = ret->getNextCls(first, fat);

                        fat[i] = (char) 0x00;
                        fat[i + 1] = (char) 0x00;
                  }
            }

            if (p->storeRoot(p->getRoot(), root) == 0) return 0;

      }else{//ako je u nekom dir- u
            unsigned long cs  = p->getPart()->getClusterSize() * p->getPart()-
>getBlockSize(),first_cls;
            char *cls = new char[cs];
            if( cls == 0) return 0;

            char pom1[FNAMELEN];

            if ((first_cls = find(tmp, root, cs, p)) == 0) return 0;
            if( p->getPart()->readCluster(first_cls, cls) == 0) return 0;

            //trazimo koji je ulaz u klasteru od fajla
            int  ii, ulaz = 2;
            for(ii = 2*sizeof(Entry); ii<cs; ii+= sizeof(Entry), ulaz++ ){

                  if( cls[ii + 11] == 0x01 &&
                        strncmp( cls + ii, ime, FNAMELEN) == 0) break;
            }

            if (ii == cs) return 0;

            //brisemo iz klastera Entry i vracamo na particiju
            for(int i = 0; i<sizeof(Entry); i++) cls[ii + i] = (char) 0;

            if( p->getPart()->writeCluster(first_cls, cls) == 0) return 0;

            delete [] cls;

            //brisemo iz fata
            ClusterNo first = ret->firstCls;
            for(unsigned long i = 4; i<KernelFS::FatSizeByte && first != 0xFFFF;
i+=2){
                  if(i == first * 2){
                        first = ret->getNextCls(first, fat);

                        fat[i] = (char) 0x00;
                        fat[i + 1] = (char) 0x00;
                  }
            }
      }

      if (p->storeFat(p->getFat1(), fat) == 0) return 0;
      if (p->storeFat(p->getFat2(), fat) == 0) return 0;
```

```
	//izbacujemo iz liste svih fajlova
	listOfAllFiles.izbaci(tmp);


	delete [] root;
	delete [] fat;

	ReleaseMutex(mutexFile);
	return 1;
}

char KernelFS::createDir(char* dirname){
	char tmp[MAX_PATH_LENGTH];
	createApsolutePath(dirname, tmp);

	if (doesExist(tmp) == 1) return 0;

	WaitForSingleObject(mutexDir, INFINITE);

	PartInfo* p = 0;
	for(int i = 0; i<26; i++){
		if(partArray[i] != 0 && partArray[i]->getLetter() == tmp[0]) {
			p = partArray[i];
			break;
		}
	}

	if(p == 0) return 0;

	unsigned long cs  = p->getPart()->getClusterSize() * p->getPart()-
>getBlockSize(),first_cls;
	char pom[MAX_PATH_LENGTH];
	char
		*cls = new char[cs],
		*fat = new char[KernelFS::FatSizeByte],
		*root = new char[KernelFS::RootSizeByte];

	if (fat == 0) return 0;
	if( root == 0) return 0;
	if( cls == 0) return 0;

	WaitForSingleObject(mutexFAT, INFINITE);
	if (p->loadRoot(p->getRoot(), root) ==0 ) return 0;
	if (p->loadFat(p->getFat1(), fat) == 0) return 0;
	ReleaseMutex(mutexFAT);

	// root, FAT_first_free_cluster = FF, cluster


	//rezervisemo klaster
	int i;
	for(i = 4; i<KernelFS::FatSizeByte; i+=2){
		if(fat[i] == 0 && fat[i+1]== 0) {
			fat[i] = (char)0xFF;
			fat[i+1] = (char)0xFF;
			break;
		}
	}
```

```
if( inRoot(tmp)){
      //ako je root

      if( p->getPart()->readCluster(i/2, cls) == 0) return 0;

      //trazimo prvi slobodan ulaz u root-u
      unsigned int j;
      for(j = 32; j<KernelFS::RootSizeByte; j+= sizeof(Entry) ){

            if( root[j + 11] == 0xE5 ||
                  root[j + 11] == 0xe5 ||
                  root[j + 11] == 0x00 ||
                  root[j + 11] == '0') break;
      }

      if (j == KernelFS::RootSizeByte) return 0;


      Directory newDir;

      getName(tmp, pom);
      strncpy(newDir[0].name, pom , FNAMELEN);
      strncpy(newDir[0].ext, "dir", FEXTLEN);
      newDir[0].attributes = 0x10;
      for (int j=0; j<14; j++) (newDir[0].reserved)[j] = 0;
      newDir[0].size = 0;
      newDir[0].firstCluster = i/2;

      getParentPath(tmp, pom );
      strncpy(newDir[1].name, pom , FNAMELEN);
      strcpy(newDir[1].ext, "\0");
      newDir[1].attributes = 0x08;
      for (int i=0; i<14; i++) (newDir[1].reserved)[i] = 0;
      newDir[1].size = 0;
      newDir[1].firstCluster = 1; //root_cls = 1

//upisivanje u klaster
      delete [] cls;

      cls = (char * )newDir;
      for(int ii =2* sizeof(Entry); ii<cs; ii++)cls[ii] = (char)0;

//upis u root
      for(int ii =0; ii<sizeof(Entry); ii++)root[j + ii] = cls[ii];
      if (p->storeRoot(p->getRoot(), root) ==0 ) return 0;

      delete [] root;

}else {
      //ako je poddirektorijum
      char pom1[FNAMELEN];

      if ((first_cls = find(tmp, root, cs, p)) == 0) return 0;
      if( p->getPart()->readCluster(first_cls, cls) == 0) return 0;

      delete [] root;

      //trazimo koji je ulaz u klasteru slobodan
```

```
      int  ii, ulaz = 2;
      for(ii = 2*sizeof(Entry); ii<cs; ii+= sizeof(Entry), ulaz++ ){

            if( cls[ii + 11] == 0xE5 ||
                  cls[ii + 11] == 0xe5 ||
                  cls[ii + 11] == 0x00 ||
                  cls[ii + 11] == '0') break;
      }

      if (ii == cs) return 0;


      Entry newEnt;

      getName(tmp, pom1);
      strncpy(newEnt.name, pom1 , FNAMELEN);
      strncpy(newEnt.ext, "dir", FEXTLEN);
      newEnt.attributes = 0x10;
      for (int j=0; j<14; j++) newEnt.reserved[j] = 0;
      newEnt.size = 0;
      newEnt.firstCluster = i/2;

      //upisujemo u klaster novi Entry i vracamo na particiju
      Entry* nizEnt;
      nizEnt = (Entry *) cls;
      nizEnt[ulaz] = newEnt;
      cls = (char *) nizEnt;

      if( p->getPart()->writeCluster(first_cls, cls) == 0) return 0;

      delete [] nizEnt;

      //dovlacimo novi klaster za tekuci dir koji pravimo
      if( p->getPart()->readCluster(i/2, cls) == 0) return 0;

      Directory newDir;
      newDir[0] = newEnt;

      getParentPath(tmp, pom );
      getName(pom, pom1);
      strncpy(newDir[1].name, pom1 , FNAMELEN);
      strcpy(newDir[1].ext, "\0");
      newDir[1].attributes = 0x08;
      for (int i=0; i<14; i++) (newDir[1].reserved)[i] = 0;
      newDir[1].size = 0;
      newDir[1].firstCluster = first_cls;

//upisivanje u klaster
      cls = (char * )newDir;
      for(int ii =2* sizeof(Entry); ii<cs; ii++)cls[ii] = (char)0;

} // kraj poddirektorijuma

//vracanje klastera i FAT-a na particiju
if (p->storeFat(p->getFat1(), fat) == 0) return 0;
if (p->storeFat(p->getFat2(), fat) == 0) return 0;
if( p->getPart()->writeCluster(i/2, cls) == 0) return 0;

delete [] fat;
```

```
	cls = 0;

	ReleaseMutex(mutexDir);
	return 1;
}

char KernelFS::readDir(char* dirname, EntryNum num, Directory& dir){
	char tmp[MAX_PATH_LENGTH];
	createApsolutePath(dirname, tmp);

	if (doesExist(tmp) == 0) return 0;

	PartInfo* p = 0;
	for(int i = 0; i<26; i++){
		if(partArray[i] != 0 && partArray[i]->getLetter() == tmp[0]) {
			p = partArray[i];
			break;
		}
	}

	if(p == 0) return 0;

	WaitForSingleObject(mutexDir, INFINITE);

	unsigned long first_cls,
		cs  = p->getPart()->getClusterSize() * p->getPart()->getBlockSize();
	char *cls = new char[cs],
	      *root = new char[KernelFS::RootSizeByte];
	Entry* nizEnt;


	if( root == 0) return 0;
	if( cls == 0) return 0;

	if (p->loadRoot(p->getRoot(), root) ==0 ) return 0;

	int j = 0, //broj procitanih ulaza
		numE = num;

	if( strlen(tmp) == 3 || strlen(tmp) == 2){

		char* pomRoot = &root[32];
		nizEnt = (Entry*) pomRoot;

		Entry ent;
		if(num == 0){//ako treba od pocetka

			//mora ovako jer je sale pogresio oko struktura
			strncpy(ent.name, root, FNAMELEN);
			strncpy(ent.ext, root + FNAMELEN, FEXTLEN);
			ent.attributes = root[FNAMELEN + FEXTLEN];
			for(int i = 0; i<15; i++) ent.reserved[i] = root[FNAMELEN +
FEXTLEN + 1 + i];
			ent.firstCluster = 0;
			ent.size = 0;

			dir[j++] = ent;
		}
```

```
            for(int i = numE; i<ENTRYCNT - 1; i++){
                  char name[FNAMELEN]; name[0] = '\0';
                  strncpy(name, nizEnt[i].name, FNAMELEN);
                  if( name[0] != '\0' && name[0] != ' ' && name[0] != 0 && name[0]
!= '0')dir[j++] = nizEnt[i];
                  else break;
            }

            pomRoot = 0;
      }else {
            //sirimo putanju
            int n = strlen(tmp);
            tmp[n++] = '\\'; tmp[n] = '\0';

            if ((first_cls = find(tmp, root, cs, p)) == 0) return 0;  //klaster
pretposlednjeg dir-a
            if( p->getPart()->readCluster(first_cls, cls) == 0) return 0;

            nizEnt = (Entry *) cls;

            for(int i = num; i<ENTRYCNT; i++){
                  if (nizEnt[i].attributes != 0x00 )dir[j++] = nizEnt[i];
                  else break;
            }
      }

      nizEnt = 0;
      delete[] root;
      delete[] cls;

      ReleaseMutex(mutexDir);
      return j;
}


char KernelFS::deleteDir(char* dirname){
      char tmp[MAX_PATH_LENGTH];
      createApsolutePath(dirname, tmp);

      if (doesExist(tmp) == 0) return 0;

      PartInfo* p = 0;
      for(int i = 0; i<26; i++){
            if(partArray[i] != 0 && partArray[i]->getLetter() == tmp[0]) {
                  p = partArray[i];
                  break;
            }
      }

      if(p == 0) return 0;

      unsigned long cs = p->getPart()->getClusterSize() * p->getPart()-
>getBlockSize(), first_cls;
      char
            *fat = new char[KernelFS::FatSizeByte],
            *root = new char[KernelFS::RootSizeByte];


      if (fat == 0) return 0;
```

```
	if( root == 0) return 0;

	WaitForSingleObject(mutexFAT, INFINITE);
	if (p->loadRoot(p->getRoot(), root) ==0 ) return 0;
	if (p->loadFat(p->getFat1(), fat) == 0) return 0;
	ReleaseMutex(mutexFAT);

	char pom[FNAMELEN];
	getName(tmp, pom);

	WaitForSingleObject(mutexDir, INFINITE);

	if( inRoot(tmp)){
		//ako je root

		char ime[FNAMELEN];
		char* dst = &root[32];  //VAZNO!!!!!!!!!!

		Entry *nizE = (Entry *) dst;

		unsigned int j, ulaz = 0;
		for(j = 32; j<KernelFS::RootSizeByte; j+= sizeof(Entry) , ulaz++){

			strcpy(ime , nizE[ulaz].name);

			if (strncmp(ime, pom, FNAMELEN) == 0){
				first_cls = nizE[ulaz].firstCluster;

				for(int ii = 0; ii<sizeof(Entry); ii++) root[j+ ii] =
(char) 0;

				fat[first_cls * 2] = (char) 0;
				fat[first_cls * 2 + 1] = (char) 0;

				break;
			}

		}

		if (j == KernelFS::RootSizeByte) return 0;

		if (p->storeRoot(p->getRoot(), root) ==0 ) return 0;

		dst = 0;
		nizE = 0;

	}else {
		//ako je poddirektorijum
		Directory myDir;
		if(readDir(tmp,0,myDir) >2 ){
			delete [] fat;
			delete [] root;
			return 0;

		}

		first_cls = find(tmp, root, cs, p);
		if (first_cls == 0) return 0;
```

```
            char *cls = new char[cs];
            if( cls == 0) return 0;

            if( p->getPart()->readCluster(first_cls, cls) == 0) return 0;

            //trazimo u kojem je ulazu u klasteru nas dir
            int  ii, ulaz;
            Entry* nizEnt;
            nizEnt = (Entry *)cls;

            for(ii = 2*sizeof(Entry), ulaz = 2; ii<cs; ii+= sizeof(Entry), ulaz++
){
                  char pom1[FNAMELEN];
                  for(int k = 0; k<FNAMELEN; k++) pom1[k] = cls[ii + k];
                  if(strncmp(pom1, pom, FNAMELEN) == 0){
                        unsigned int first_cls_pom = nizEnt[ulaz].firstCluster;

                        //oslobadjamo u FAT- u redni broj klastera
                        fat[first_cls_pom * 2] = (char) 0;
                        fat[first_cls_pom * 2 + 1] = (char) 0;

                        //brisemo Entry iz roditeljskog klastera
                        for(int i =0; i<FNAMELEN; i++)nizEnt[ulaz].name[i] = (char)
0;
                        for(int i =0; i<FEXTLEN; i++)nizEnt[ulaz].ext[i] = (char)
0;
                        nizEnt[ulaz].attributes = 0xe5;
                        for (int i=0; i<14; i++) (nizEnt[ulaz].reserved)[i] = 0;
                        nizEnt[ulaz].size = 0;
                        nizEnt[ulaz].firstCluster = 0;

                        //vracamo roditeljski klaster na particiju
                        cls = (char * )nizEnt;
                        if( p->getPart()->writeCluster(first_cls, cls) == 0) return
0;

                        break;
                  }

            }

            delete nizEnt;//ovim brisemo i cls

            if (ii == cs) return 0;

      }

      //vracanje FAT-a na particiju
      if (p->storeFat(p->getFat1(), fat) == 0) return 0;
      if (p->storeFat(p->getFat2(), fat) == 0) return 0;

      delete [] fat;
      delete [] root;

      ReleaseMutex(mutexDir);
      return 1;
}

char KernelFS::cd(char* dirname){
```

```
	char tmp[MAX_PATH_LENGTH];

	createApsolutePath(dirname, tmp);

	char slovo = tmp[0];
	if ( partLetter[slovo - 'A'] == 0) return 0; //ako ne postoji

	if (strlen(tmp) == 3){  //ako je root
		strcpy(pwdName, tmp);
		return 1;
	}

	for(int i = 0; i<26; i++)
	{
		if(partArray[i] != 0 && partArray[i]->getLetter() == slovo)
		{
			if( doesExist(tmp) == 0 ) return 0;
			strcpy(pwdName, tmp);
			return 1;
		}
	}

	return 0;
}


//KREIRANJE APSOLUTNE PUTANJE FAJLA
void KernelFS::createApsolutePath(char *src, char *result){
	if( src[1] == ':'){
		int n = strlen(src);
		result[0] = toupper(src[0]);
		if(src[n] == '\\') src[n] = '\0';
		for(int i = 1; i<n || src[i] == '\0'; i++) result[i] = src[i];
		return ;
	}

	if(src[0] == '.' && src[1]== '\\'){
		strcpy(result , pwdName);

		int pwdLen = strlen(result);
		if(result[pwdLen] != '\\') result[pwdLen] = '\\';
		strcpy(result + pwdLen + 1, src + 2);
		return;

	}

	if(src[0] == '.' && src[1] == '.'){
		int br = 0, j =0, n;
		char pom[MAX_PATH_LENGTH], name[MAX_PATH_LENGTH];
		strcpy( pom, src);
		while(pom[0] == '.' && pom[1] == '.'){
			strcpy(pom, pom + 3);
			br++;
		}
		strcpy( name,pom);

		while(br > 0){
			strcpy(pom , strrchr(pwdName, '\\'));
			j+=strlen(pom);
```

```
                  n = strlen(pwdName) - strlen(pom);
                  strncpy(pom, pwdName, n);
                  pom[n] = '\0';

                  br--;
            }

            int pwdLen = strlen(pom);
            if(pom[pwdLen] != '\\') {pom[pwdLen++] = '\\'; pom[pwdLen] = '\0'; }
            strcat(pom, name);

            strcpy(result, pom);

            return;
      }

      if(src[0]!= '.' && src[1]!= ':' && src[1]!= '\\' && src[1]!= '.'){
            char ok[MAX_PATH_LENGTH];
            strcpy(ok, pwdName);
            int n = strlen(ok);
            if(n != 3){strcat(ok, "\\");}

            strcat(ok, src);

            strcpy(result, ok);

            return;
      }

}

//DOHVATANJE PUTANJE RODITELJA
void KernelFS::getParentPath(char* src, char* result){
      char tmp[MAX_PATH_LENGTH];
      strcpy(tmp, src);
      int n = strlen(tmp);
      for(int i = n-1; i>0; i--)
            if(tmp[i] == '\\'){
                  tmp[i] = '\0';
                  break;
            }

      n = strlen(tmp);
      for(int i = 0; tmp[i]!= '\0'; i++) result[i] = tmp[i];
      //strcpy(result, tmp);
      result[n] = '\0';

}

//DOHVATANJE SAMO IMENA DIREKTORIJUMA
void KernelFS::getName(char *src, char *result){
      char pom[MAX_PATH_LENGTH];
      strcpy(pom , strrchr(src, '\\'));
      strcpy(result, pom + 1);
}

//ISPITUJE DA LI TREBA DA SE MONTIRA U ROOT
bool KernelFS::inRoot(char *src){
      int br = 0;
```

```
	for(int i =0; i< strlen(src); i++)
		if (src[i] == '\\') br++;
	return br == 1;

}
//BROJANJE KOLIKO IMA DIREKTORIJUMA U PUTANJI
int KernelFS::countNames(char* path){
	int br = 0, i = 0;
	while(path[i] != '\0'){
		if(path[i++] == '\\') br++;
	}
	return br;
}
//VRACANJE KLASTERA POSLEDNJE MONTIRANOG DIR-A
unsigned int KernelFS::find (char* path, char* root, ClusterNo cs, PartInfo* part){

	char * pomRoot = &root[32];
	Entry* nizEnt, *nizEnt1;


	char pom[MAX_PATH_LENGTH], pom1[FNAMELEN];
	unsigned int first_cls = 0;
	Partition* p = part->getPart();
	int  j = 3;

	nizEnt = (Entry *) pomRoot;

	for( int i = 0; i< countNames(path)- 1; i++){
		if(i == 0) {
			int k= 0 , ulaz = 0;
			while(path[j] != '\\'){
				pom[k++] = path[j++];
			}
			pom[k] = '\0';

			for( k = 32; k<KernelFS::RootSizeByte; k+=sizeof(Entry), ulaz++){
				strncpy(pom1, root + k, FNAMELEN);
				if ( strcmp(pom, pom1) == 0) {
					first_cls = nizEnt[ulaz].firstCluster;

					break;
				}
			}
			if( k == KernelFS::RootSizeByte) return 0;
		} else{

			char *ccls  = new char[cs];
			if (ccls == 0) return 0;


			if( p->readCluster(first_cls, ccls) == 0) return 0;
			if (ccls == 0) return 0;
			nizEnt1 = (Entry *) ccls;

			int k= 0, ulaz = 2;
			j++;
			while(path[j] != '\\'){
				pom[k++] = path[j++];
			}
```

```
                  pom[k] = '\0';

                  for( k = 2*sizeof(Entry); k<cs; k+=sizeof(Entry), ulaz++){
                        strncpy(pom1, ccls + k, FNAMELEN);
                        if ( strcmp(pom, pom1) == 0) {
                              first_cls = nizEnt1[ulaz].firstCluster;

                              break;
                        }
                  }
                  if( k == cs) return 0;

                  delete [] ccls;
            }
      }

      pomRoot = 0;
      nizEnt = 0;
      nizEnt1 = 0;

      return first_cls;

}

//CITANJE ENTRY-A IZ KLASTERA
int KernelFS::readEntry(char *cls, int num, Entry &ent){
      Entry* pomEnt = (Entry *) cls;

      ent = pomEnt[num];

      return 1;
}

//SECKANJE NA IME I EXT
void KernelFS::cutName(char *src, char *name, char *ext){
      int i = 0 ,k = 0;
      while(src[i]!= '.' && i<FNAMELEN){
            name[i] = src[i];
            i++;
      }
      name[i++] = '\0';

      while(k<FEXTLEN){
            ext[k++] = src[i++];
      }
      if(!(k == FEXTLEN)) ext[k] = '\0';
}

//KREIRANJE APSOLUTNE PUTANJE FAJLA
void KernelFS::createApsoluteFileName(char *src, char *result){
      if( src[1] == ':'){
            int n = strlen(src);
            result[0] = toupper(src[0]);
            if(src[n] == '\\') src[n] = '\0';
            for(int i = 1; i<n || src[i] == '\0'; i++) result[i] = src[i];
            return ;
      }

      if(src[0] == '.' && src[1]== '\\'){
```

```
            strcpy(result , pwdName);

            int pwdLen = strlen(result);
            if(result[pwdLen] != '\\') result[pwdLen] = '\\';
            strcpy(result + pwdLen + 1, src + 2);
            return;

      }

      if(src[0] == '.' && src[1] == '.'){
            int br = 0, j =0, n;
            char pom[100];
            while(src[0] == '.' && src[1] == '.'){
                  strcpy(pom, src + 3);
                  strcpy(src, pom);
                  br++;
            }

            while(br > 0){
                  strcpy(pom , strrchr(pwdName, '\\'));
                  j+=strlen(pom);
                  n = strlen(pwdName) - strlen(pom);
                  strncpy(pom, pwdName, n);
                  pom[n] = '\0';
                  strcpy(pwdName, pom);

                  br--;
            }
            strncat(result, pwdName, strlen(pwdName) - j);
            int pwdLen = strlen(result);
            if(result[pwdLen] != '\\') result[pwdLen] = '\\';
            strcat(result + pwdLen + 1, src);

            return;
      }

      if(src[0]!= '.' && src[1]!= ':' && src[1]!= '\\' && src[1]!= '.'){
            char ok[MAX_PATH_LENGTH];
            strcpy(ok, pwdName);
            int pwdLen = strlen(ok);
            if(ok[pwdLen - 1] != '\\') ok[pwdLen] = '\\';
            strcat(ok, src);

            strcpy(result, ok);

            return;
      }
}
```

```
// file: kernelfile.cpp

#include "file.h"
#include "Monitor.h"
#include "kernelfile.h"
#include "kernelfs.h"
#include <iostream>
using namespace std;

KernelFile::KernelFile ( PartInfo* pi){
	this->currentPos = 0;
	this->endOfFile = 0;
	this->firstCls = 0;
	this->inCls = 1;
	this->numOfCls = 1;
	this->mode = 'c';

	this->p = pi;
	this->clsSize = p->getPart()->getClusterSize() * p->getPart()-
>getBlockSize();
	this->cls = new char[clsSize];

	fileMonitor = Monitor();
	mutex = CreateMutex( NULL,FALSE, NULL);

	copy = false;

}
KernelFile::~KernelFile(){
	delete [] cls;
	p = 0;
	delete file;
}

KernelFile::KernelFile(const KernelFile* kf){
	copy = true;

	this->currentPos = currentPos;
	this->endOfFile = kf->endOfFile;
	this->firstCls = kf->firstCls;
	this->inCls = kf->inCls;
	this->numOfCls = kf->numOfCls;
	this->mode = kf->mode;

	this->p = kf->p;
	this->clsSize = kf->clsSize;
	this->cls = new char[this->clsSize]; //kf->cls;
	for(int i = 0; i< this->clsSize; i++) this->cls[i] = kf->cls[i];
	//if( this->p->getPart()->readCluster(this->firstCls, this->cls) == 0)
cout<<"GRESKA\n";

	fileMonitor = kf->fileMonitor;

}
char KernelFile::write (BytesCnt cnt, char* buffer){
	if(mode == 'c') return 0;
	if( currentPos > endOfFile) return 0;

	BytesCnt forWrite = cnt;
```

```
	/* pisi do kraja klastera, ako si upisao premalo dovuci
	novi klaster (a prethodni vrati na particiju)i upisi u njega */

	for(int i = 0; i< forWrite; i++){
		if((currentPos % clsSize) != 0 || currentPos == 0){
			cls [currentPos % clsSize]  = buffer[i];
			currentPos++;
			if (currentPos > endOfFile) endOfFile++;
			else if (currentPos == endOfFile) return 1;

		} else {

			ClusterNo first = firstCls;
			char fat[131072]; //KernelFS::FatSizeByte;
			WaitForSingleObject(KernelFS::mutexFAT, INFINITE);
			p->loadFat(p->getFat1() , fat);
			ReleaseMutex(KernelFS::mutexFAT);

			if ( inCls == numOfCls){
				unsigned long i;
				for(i = 4; i<KernelFS::FatSizeByte; i+=2){
					if(fat[i] == 0 && fat[i+1]== 0) {
						fat[i] = (char)0xFF;
						fat[i+1] = (char)0xFF;

						numOfCls++;

						p->storeFat(p->getFat1(), fat);
						p->storeFat(p->getFat2(), fat);

						break;
					}
				}

				int num = 1;
				while ( inCls > num){
					first = getNextCls(first, fat);
					num++;
				}

				//azuriramo ulaz u fatu za fajl
				fat[first * 2] = (char)( ((unsigned short)i/2) & 0x00FF);
				fat[first * 2 + 1] = (char)( ((unsigned short)i/2) &
0xFF00);

				p->storeFat(p->getFat1() , fat);
				p->storeFat(p->getFat2() , fat);


			} else {
				int num = 1;
				while ( inCls > num){
					first = getNextCls(first, fat);
					num++;
				}
			}
```

```
                if( p->getPart()->writeCluster(first, cls) == 0) return 0;
                first = getNextCls(first, fat);
                inCls++;
                if( p->getPart()->readCluster(first, cls) == 0) return 0;

                cls [currentPos % clsSize]  = buffer[i];
                currentPos++;
                if (currentPos >= endOfFile) endOfFile++;
          }
     }

     return 1;
}
BytesCnt KernelFile::read (BytesCnt cnt, char* buffer){
     if(mode == 'c') return 0;
     if( currentPos > endOfFile) return 0;

     BytesCnt forRead = cnt;
     if( currentPos+ cnt > endOfFile) forRead = endOfFile - currentPos;

     /* citaj do kraja klastera, ako si procitao premalo dovuci
     sledeci klaster (a prethodni vrati na particiju)i citaj iz njega */

     for(int i = 0; i< forRead; i++){
          if((currentPos % clsSize) != 0 || currentPos == 0){
                buffer[i] = cls [currentPos % clsSize];
                currentPos++;

          } else {

                ClusterNo first = firstCls;
                char fat[131072];//KernelFS::FatSizeByte
                p->loadFat(p->getFat1() , fat);

                int num = 1;
                while ( inCls > num){
                      first = getNextCls(first, fat);
                      num++;
                }

                first = getNextCls(first, fat);
                inCls++;
                if( p->getPart()->readCluster(first, cls) == 0) return 0;

                buffer[i] = cls [currentPos % clsSize];
                currentPos++;
          }
     }

     return forRead > 0? forRead : 0;
}
char KernelFile::truncate (){

     if(mode == 'c') return 0;
     if( currentPos > endOfFile) return 0;
     if( currentPos == endOfFile) return 1;

     unsigned long cnt = endOfFile - currentPos;
     int num = cnt / clsSize; num++;
```

```
	if(inCls == num) endOfFile = currentPos;
	else if (inCls > num){//unazad
		ClusterNo first = firstCls,last = firstCls;
		char fat[131072];//KernelFS::FatSizeByte
		p->loadFat(p->getFat1() , fat);

		inCls = 1;

		int num1 = 1;
		while(num1< numOfCls){
			last = getNextCls(last, fat);
			num1++;
		}

		num1 = 1;
		while ( num > num1){
			first = getNextCls(first, fat);
			num1++;
			inCls++;
		}

		if( p->getPart()->writeCluster(last, cls) == 0) return 0;
		if( p->getPart()->readCluster(first, cls) == 0) return 0;

		endOfFile = currentPos;

	}else if (inCls < num){//unapred
		ClusterNo first = firstCls,last = firstCls;
		char fat[131072];//KernelFS::FatSizeByte
		p->loadFat(p->getFat1() , fat);

		int num1 = 1;
		while (num1 < inCls) {
			first = getNextCls(first, fat);
		}

		inCls = 1;

		while ( num > inCls) {
			last = getNextCls(last, fat);
			inCls++;

		}

		if( p->getPart()->writeCluster(first, cls) == 0) return 0;
		if( p->getPart()->readCluster(last, cls) == 0) return 0;

		endOfFile = currentPos;
	}

	return 1;
}

char KernelFile::seek (BytesCnt cnt){

	if(cnt < 0 || cnt > endOfFile) return 0;
	if(mode == 'c')   return 0;
```

```
	int num = cnt / clsSize; num++;

	if(inCls == num) currentPos = (cnt == 0? 0: cnt-1);
	else if (inCls > num){//unazad
		ClusterNo first = firstCls,last = firstCls;
		char fat[131072];//KernelFS::FatSizeByte
		p->loadFat(p->getFat1() , fat);

		inCls = 1;

		int num1 = 1;
		while(num1< numOfCls){
			last = getNextCls(last, fat);
			num1++;
		}

		num1 = 1;
		while ( num > num1){
			first = getNextCls(first, fat);
			num1++;
			inCls++;
		}

		if( p->getPart()->writeCluster(last, cls) == 0) return 0;
		if( p->getPart()->readCluster(first, cls) == 0) return 0;

		currentPos = (cnt == 0? 0: cnt-1);

	}else if (inCls < num){//unapred
		ClusterNo first = firstCls,last = firstCls;
		char fat[131072];//KernelFS::FatSizeByte
		p->loadFat(p->getFat1() , fat);

		int num1 = 1;
		while (num1 < inCls) {
			first = getNextCls(first, fat);
		}

		inCls = 1;

		while ( num > inCls) {
			last = getNextCls(last, fat);
			inCls++;

		}

		if( p->getPart()->writeCluster(first, cls) == 0) return 0;
		if( p->getPart()->readCluster(last, cls) == 0) return 0;

		currentPos = (cnt == 0? 0: cnt-1);
	}

	return 1;
}

BytesCnt KernelFile::getFileSize (){
	return endOfFile + 1;
}
```

```
char KernelFile::eof (){
	if(endOfFile == currentPos) return 2;
	if(endOfFile > currentPos) return 0;
	if(endOfFile < currentPos) return 1;
	return 0;
}

void KernelFile::close (){
	if (mode == 'c') return;

	int nr;
	switch(mode){
		case 'r' :
			nr = fileMonitor.endRead();
			if(this->copy == false && nr > 0){
				return;
			}
			break;
		case 'w': fileMonitor.endWrite(); break;
		case 'a': fileMonitor.endApend(); break;
	}

//	ReleaseMutex(this->mutex);
	mode = 'c';
}

// POSTAVLJANJE PRVOG KLASTERA FAJLA
void KernelFile::setFirstCls(ClusterNo cls){
	firstCls = cls;
}
//POSTAVLJANJE KURSORA U FAJLU
int KernelFile::setCurrentPos(BytesCnt cnt){
	int num = cnt / clsSize; num++;

	ClusterNo first = firstCls;
	char fat[131072]; //KernelFS::FatSizeByte
	p->loadFat(p->getFat1() , fat);

	if(inCls == num) currentPos = cnt;
	else {
		inCls = 1;

		int num1 = 1;
		while ( num > num1){
			first = getNextCls(first, fat);
			num1++;
			inCls++;
		}

		if( p->getPart()->writeCluster(first, cls) == 0) return 0;
		first = getNextCls(first, fat);
		inCls++;
		if( p->getPart()->readCluster(first, cls) == 0) return 0;

		currentPos = cnt;

	}

	return 1;
```

```
}
//VRACANJE KRAJA FAJLA
BytesCnt KernelFile::getEOF(){
	return endOfFile;
}
//DOBIJANJE SLEDECEG KLASTERA
ClusterNo KernelFile::getNextCls(ClusterNo cluster, char* fat){
	unsigned long ulaz = cluster *2;
	unsigned short result = 0xFFFF;

	result = (unsigned short)(fat[ulaz + 1]<<8)|(fat[ulaz]);

	return result;
}
//POSTAVLJANJE FAJLA
void KernelFile::setFile(File * f){
	this->file = f;
	f->myImpl = this;
}
char KernelFile::getMode(){
	return mode;
}
void KernelFile::setMode(char c){
	mode = c;
}
```

```
// file: lista.cpp

//#include "kernelfile.h"
#include "lista.h"
#include "kernelfs.h"
#include <iostream>
using namespace std;


void Lista::brisi () {                          // Praznjenje liste.
  while (prvi) { Elem* stari = prvi; prvi = prvi->sled; delete stari; }
  posl = 0; duz = 0;
}

void Lista::naPocetak(char* ime, KernelFile* kf){
      prvi = new Elem (ime, kf, prvi);  duz++;
      if(prvi == 0) posl = prvi;
}

KernelFile* Lista::uzmi (char* i) {
  Elem *tek = prvi, *pret = 0;
  if(prvi == 0) return 0;

  while (tek)
        if (strncmp(i , tek->ime, MAX_PATH_LENGTH) != 0) {
      pret = tek; tek = tek->sled;
    } else {
          return tek->kf;
        }
  return 0;
}

KernelFile* Lista::uzmiSaKraja () {
      Elem *tek = prvi;
      if(prvi == 0) return 0;

      KernelFile* kf = 0;
      int br = duz - 1;
      if(br > 0){
            while(br>0){
                  tek = tek->sled;
                  br--;
            }
            kf = posl->kf;
            posl = tek; posl->sled = 0;
            duz--;
      } else if (br == 0){
            kf = prvi->kf;
            posl = prvi = 0;
            duz = 0;
      }

      return kf;
}

int Lista::izbaci (char* i) {
  Elem *tek = prvi, *pret = 0;
  while (tek)
        if (strncmp(i , tek->ime, MAX_PATH_LENGTH) != 0) {
```

```
      pret = tek; tek = tek->sled;
    } else {
      Elem *stari = tek;
      tek = tek->sled;
      if (!pret) prvi = tek; else pret->sled = tek;
        if(tek == 0) posl = pret; //if(stari == posl)
      delete stari;
        duz--;
        return 1;
    }
      return 0;
}

ostream& operator<< (ostream& it, const Lista& lst) { // Pisanje.
  it << '(';
  for (Lista::Elem* tek=lst.prvi; tek; tek=tek->sled)
  { it << tek->ime; it << ','; it<< tek->kf;
    if (tek->sled) it << ',';  it<<'\n';}
  return it << ')';
}
```

```
//file: monitor.cpp

#include "lista.h"
#include "Monitor.h"
#include "KernelFS.h"
#include <windows.h>
#include <iostream>
using namespace std;

Monitor::Monitor(){
	nr = nw = na = dr = dw = da = 0;
	rMutex = CreateSemaphore(NULL, 0, MAX_SEM_CNT, NULL);
	wMutex = CreateSemaphore(NULL, 0, MAX_SEM_CNT, NULL);
	aMutex = CreateSemaphore(NULL, 0, MAX_SEM_CNT, NULL);
	db = CreateSemaphore(NULL, 1, 1, NULL);
}

int Monitor::startRead(){
	WaitForSingleObject(db, INFINITE);
	if(nw > 0 || na > 0){
		dr++; ReleaseSemaphore(db, 1, NULL);
		ReleaseMutex(KernelFS::mutexFile);
		WaitForSingleObject(rMutex, INFINITE);
		WaitForSingleObject(KernelFS::mutexFile, INFINITE);
	}
	nr++;
	if(dr > 0) {dr--; ReleaseSemaphore(rMutex, 1, NULL);}
	else ReleaseSemaphore(db, 1, NULL);

	return nr;
}

int Monitor::endRead(){
	WaitForSingleObject(db, INFINITE);
	nr--;
	if (nr == 0 && dw > 0) { dw--; ReleaseSemaphore(wMutex, 1, NULL);}
	else if( nr==0 && da > 0){ da--; ReleaseSemaphore(aMutex, 1, NULL);}
	else ReleaseSemaphore(db, 1, NULL);

	return nr;
}

int Monitor::startWrite(){
	WaitForSingleObject(db, INFINITE);
	if (nr > 0 || nw > 0 || na > 0) {
		dw++; ReleaseSemaphore(db, 1, NULL);
		ReleaseMutex(KernelFS::mutexFile);
		WaitForSingleObject(wMutex, INFINITE);
		WaitForSingleObject(KernelFS::mutexFile, INFINITE);
	}
	nw = nw + 1;
	ReleaseSemaphore(db, 1, NULL);

	return nw;
}

int Monitor::endWrite(){
	WaitForSingleObject(db, INFINITE);
	nw--;
```

```
	if (dr > 0) { dr--; ReleaseSemaphore(rMutex, 1, NULL); }
	else if (dw > 0) { dw--; ReleaseSemaphore(wMutex, 1, NULL);}
	else if(da > 0) { da--; ReleaseSemaphore(aMutex, 1, NULL);}
	else ReleaseSemaphore(db, 1, NULL);

	return nw;
}

int Monitor::startApend(){
	WaitForSingleObject(db, INFINITE);
	if (nr > 0 || nw > 0 || na > 0) {
		da++; ReleaseSemaphore(db, 1, NULL);
		ReleaseMutex(KernelFS::mutexFile);
		WaitForSingleObject(aMutex, INFINITE);
		WaitForSingleObject(KernelFS::mutexFile, INFINITE);
	}
	na = na + 1;
	ReleaseSemaphore(db, 1, NULL);

	return na;
}

int Monitor::endApend(){
	WaitForSingleObject(db, INFINITE);
	na--;
	if (dr > 0) { dr--; ReleaseSemaphore(rMutex, 1, NULL); }
	else if(da > 0) { da--; ReleaseSemaphore(aMutex, 1, NULL);}
	else if (dw > 0) { dw--; ReleaseSemaphore(wMutex, 1, NULL);}

	else ReleaseSemaphore(db, 1, NULL);

	return na;
}
```