Sam Caldwell
Rohith Prakash

Lab 5 Report

**OBJECTIVES**

The goals of Lab 5 were to interface a SD card using our OS, designing and implementing a file system, and extending our shell to support file/disk IO and input redirecton, testing accuracy and performance along the way. An important part of the lab was layering the different interfaces for the card: serial, eDisk.c, eFile.c, and the shell/OS. Abstract hardware interfaces are a necessary feature for platform-independence.

**SOFTWARE DESIGN**

```
int eFile_Init(void) {
  // initialize file system
      int i;
  volatile unsigned long delay;
      if(_sysInit)
            return 1;
  eDisk_Init(DRIVE);
  eDisk_ReadBlock(_blockBuff, DIRECTORY);
      memcpy(_eFile_List, _blockBuff, sizeof(eFile_File) * MAX_FILES);
      //count files
      num_files = 0;
      for(i = 1; i < MAX_FILES; i++)
            if(_eFile_List[i].name[0])
                  num_files++;
      _wOpen = _rOpen = 0;
      _sysInit = 1;
  // profiling code : set PB0 low when reading, high otherwise
  SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOB;
  delay = SYSCTL_RCGC2_R;
  GPIO_PORTB_DIR_R |= 0x01; // output
  GPIO_PORTB_DEN_R |= 0x01;  // digital mode
  GPIO_PORTB_DATA_R |= 0x01;  // initialize high
      return 0;
}

int eFile_Format(void)
{
      eFile_WClose();
      eFile_RClose();
  // erase disk, add format
      memset(_eFile_List, 0, sizeof(eFile_File) * MAX_FILES);
  // create free space manager entry in the directory
  strcpy(_freeList->name, "spc_mgr");
  _freeList->firstBlock = 1;
  _freeList->lastBlock = BLOCKS - 1;
```

```
  _freeList->size = 0;
  // write the space manager to the first entry in the directory
      _eFile_ClearBlockBuff();
  memcpy(_blockBuff, _eFile_List, sizeof(eFile_File) * MAX_FILES);
  eDisk_WriteBlock(_blockBuff, DIRECTORY);
  _eFile_MakeFreeList();
      num_files = 0;
  return 0;
}

int eFile_Create(const char name[FILE_NAME_SIZE]) {
  // create new file, make it empty
      int i;
      if(_freeList->firstBlock == 0 || _eFile_Find(name))
            return 1;

      for(i = 1; i < MAX_FILES; i++)
      {
            if(_eFile_List[i].firstBlock == 0) //free space in
directory
            {
                  eFile_File* file = &_eFile_List[i];
                  char new_name[FILE_NAME_SIZE];
                  memcpy(new_name, name, FILE_NAME_SIZE);
                  new_name[FILE_NAME_SIZE - 1] = 0;
                  strcpy(file->name, new_name);    // write file name
                  file->firstBlock = file->lastBlock = _freeList-
>firstBlock; // first block is first available block
                  _eFile_ClearBlockBuff();
                  eDisk_ReadBlock(_blockBuff, _freeList->firstBlock);
                  _freeList->firstBlock = (_blockBuff[0] |
(_blockBuff[1] << 8)); //update freeList
                  _eFile_ClearBlockBuff();
                  memset(_blockBuff, 0, sizeof(_blockBuff[0]) * 2); //
first 2 bytes are a null ptr
                  _blockBuff[WRITE_INDEX] = WRITE_INDEX + 2;  // start
writing to 5th byte
                  _blockBuff[WRITE_INDEX + 1] = 0;
                  eDisk_WriteBlock(_blockBuff, file->firstBlock); // set
next to zero
                  memcpy(_blockBuff, _eFile_List, sizeof(_blockBuff));
// copy dir
                  eDisk_WriteBlock(_blockBuff, DIRECTORY); // write dir
                  num_files++;
                  return 0;
            }
      }
      return 1; // no free space
}

int eFile_WOpen(const char name[FILE_NAME_SIZE]) {
  // open a file for writing
```

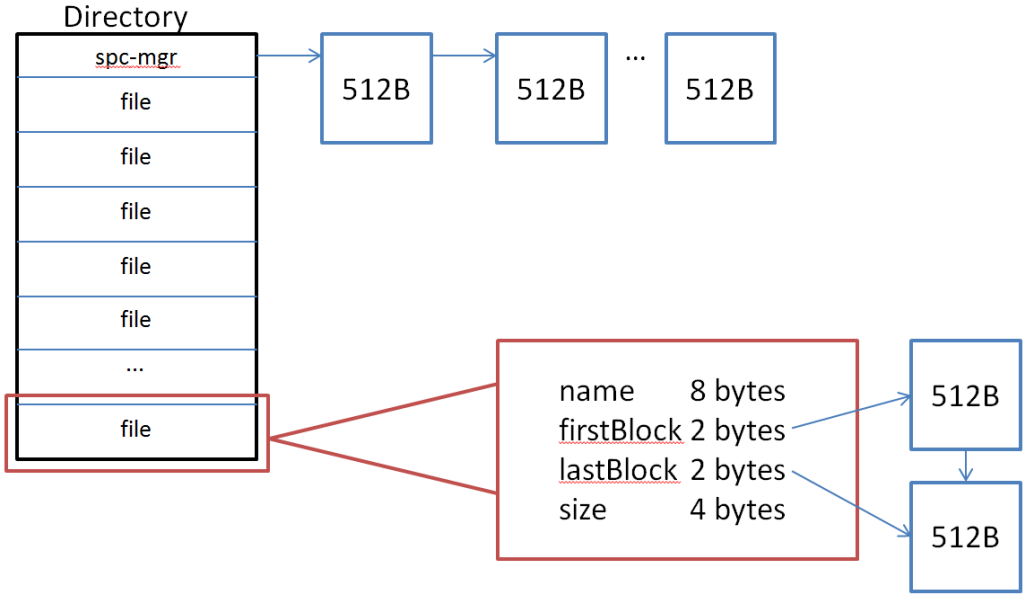
```
      eFile_File* file = _eFile_Find(name);
      if(_wOpen || (file == NULL && eFile_Create(name)))
            return 1;
      file = _eFile_Find(name);
      _wOpen = 1;
      eDisk_ReadBlock(_writeBuff, file->lastBlock);
      _wIndex = (_writeBuff[WRITE_INDEX] | (_writeBuff[WRITE_INDEX + 1]
<< 8));
      _wSector = file->lastBlock;
      _wFile = file;
  return 0;
}

int eFile_Write(const char data) {
      if(!_wOpen)
            return 1;
      if(_wIndex >= BLOCK_SIZE) // allocate a new block
      {
            if(_freeList->firstBlock == 0) // no free space
                        return 1;
            _writeBuff[0] = (_freeList->firstBlock & 0xFF);
            _writeBuff[1] = (_freeList->firstBlock >> 8); // update
next block
            _writeBuff[WRITE_INDEX] = ((BLOCK_SIZE + 1) & 0xFF);
            _writeBuff[WRITE_INDEX + 1] = ((BLOCK_SIZE + 1) >> 8);
            _wFile->lastBlock = _freeList->firstBlock;
            eDisk_WriteBlock(_writeBuff, _wSector); // commit finished
block
            _wSector = _freeList->firstBlock; // update sector to write
to
            eDisk_ReadBlock(_writeBuff, _wSector); // get new block
            _writeBuff[WRITE_INDEX] = WRITE_INDEX + 2;  // start writing
to 5th byte
            _writeBuff[WRITE_INDEX + 1] = 0;
            _freeList->firstBlock = (_writeBuff[0] | (_writeBuff[1] <<
8)); //update freeList
            _wIndex = WRITE_INDEX + 2; // start writing to 5th byte
      }
      _writeBuff[_wIndex++] = data;
      _wFile->size++;
  return 0;
}

int eFile_Close(void) {
      if(!_sysInit)
            return 1;
      if(_wOpen || _rOpen)
      {
            eFile_WClose();
            eFile_RClose();
      }
      // TODO: write directory
```

```
      _eFile_ClearBlockBuff();
      memcpy(_blockBuff, _eFile_List, sizeof(_blockBuff));
      eDisk_WriteBlock(_blockBuff, DIRECTORY);
  return (_sysInit = 0);
}

int eFile_WClose(void) {
  // close the file for writing
  if(!_wOpen)
        return 1;
      _writeBuff[WRITE_INDEX] = _wIndex & 0xFF;
      _writeBuff[WRITE_INDEX + 1] = (_wIndex >> 8);
      eDisk_WriteBlock(_writeBuff, _wSector);
      _eFile_ClearBlockBuff();
      memcpy(_blockBuff, _eFile_List, sizeof(_blockBuff)); // copy dir
      eDisk_WriteBlock(_blockBuff, DIRECTORY);
      _wFile = NULL;
      return (_wOpen = 0);
}

int eFile_ROpen(const char name[FILE_NAME_SIZE]) {
  // open a file for reading
  eFile_File* file = _eFile_Find(name);
      if(_rOpen || file == NULL)
            return 1;
      _rOpen = 1;
      eDisk_ReadBlock(_readBuff, file->firstBlock);
      _rSize = (_readBuff[WRITE_INDEX] | (_readBuff[WRITE_INDEX + 1] <<
8));
      _rSector = file->firstBlock;
      _rIndex = WRITE_INDEX + 2; // start reading from 5th byte
  return 0;
}

int eFile_ReadNext(char *pt) {
  // get next byte
  if(!_rOpen || _rIndex >= _rSize)
            return 1;
      if(_rIndex >= BLOCK_SIZE)
      {
            _rSector = (_readBuff[0] | (_readBuff[1] << 8)); // get
next sector
            if(_rSector == 0) // sanity check
                  return 1;
            eDisk_ReadBlock(_readBuff, _rSector); // load next block
into ram
            _rIndex = WRITE_INDEX + 2; // start from 5th byte
            _rSize = (_readBuff[WRITE_INDEX] | (_readBuff[WRITE_INDEX +
1] << 8)); // load this block's size
      }
      *pt = _readBuff[_rIndex++];
  return 0;
```

```
}

int eFile_RClose(void) {
  // close the file for writing
  if(!_rOpen)
          return 1;
  return (_rOpen = 0);
}

int eFile_Directory(int(*fp)(const char *format, ...)) {
      int i;
      fp("%d B\t.\n", sizeof(eFile_File) * num_files);
      for(i = 1; i < MAX_FILES; i++)
      {
            eFile_File* file = &_eFile_List[i];
            if(file->firstBlock)
                  fp("%d B\t%s\n", file->size, file->name);
      }
  return 0;
}

int eFile_List(char list[MAX_FILES][8])
{
      int i, num = 0;
      for(i = 1; i < MAX_FILES; i++)
      {
            eFile_File* file = &_eFile_List[i];
            if(file->firstBlock)
                  strcpy(list[num++], file->name);
      }
      list[num][0] = 0;
      return 0;
}

int eFile_Delete(const char name[FILE_NAME_SIZE]) {
  // remove this file
  eFile_File* file = _eFile_Find(name);
      if(file == NULL || (_wFile != NULL && strcmp(name, _wFile->name)
== 0))
            return 1;
      eDisk_ReadBlock(_blockBuff, _freeList->lastBlock);
      _blockBuff[0] = (file->firstBlock & 0xFF);
      _blockBuff[1] = (file->firstBlock >> 8);
      eDisk_WriteBlock(_blockBuff, _freeList->lastBlock);
      file->firstBlock = 0;
      file->size = 0;
      strcpy(file->name, "");
      _eFile_ClearBlockBuff();
      memcpy(_blockBuff, _eFile_List, sizeof(eFile_File) * MAX_FILES);
      eDisk_WriteBlock(_blockBuff, DIRECTORY);
      num_files--;
  return 0;
```

```
}

int eFile_RedirectToFile(const char *name) {
  if(eFile_WOpen(name)) // creates file if doesn't exist
           return 1;  // cannot open file
      RT_StreamToFile(1);
  return 0;
}

int eFile_EndRedirectToFile(void) {
  RT_StreamToFile(0);
      if(eFile_WClose())
            return 1;
  return 0;
}
```

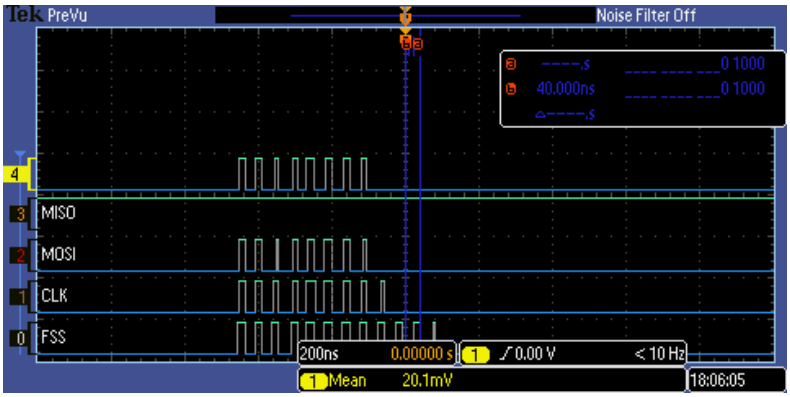
```
// new interpreter commands
static _SH_CommandPtr _SH_CommandList[] = {
        {"sd_format", &_SH_Format},
        {"write", &_SH_Write},
        {"read", &_SH_Read},
        {"cat", &_SH_Read},
        {"rm", &_SH_Rm},
        {"touch", &_SH_Create},
        {"file_test", &_SH_FileTest},
        {"ls", &_SH_DirectoryList},
        {"hexdump", &_SH_HexDump},
        {"sector", &_SH_SectorDump},
        {"diskinfo", &_SH_DiskInfo}
};
```



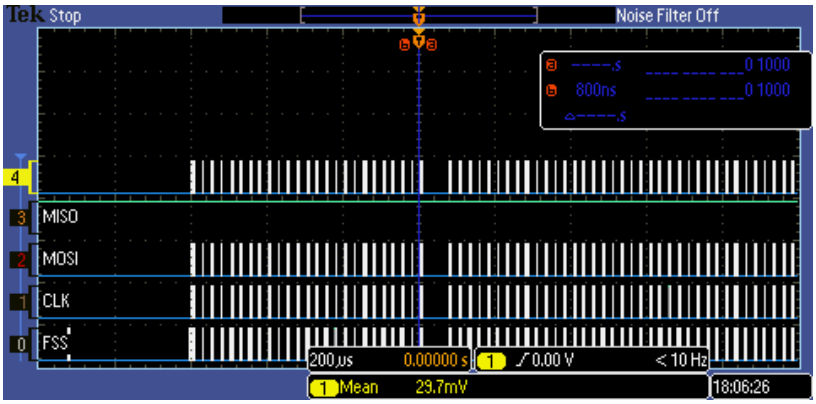
**MEASUREMENT DATA**

Our system took 14 ms to write and 12 ms to read 10 512 byte blocks, so the read bandwidth is ~ 426,667 bytes/second and the write bandwidth is ~365,714 bytes/second.

Command frame:




Data frame:




1) Yes. Our implementation has external fragmentation. External fragmentation is when a single file is potentially allocated using non-adjacent sectors in disk. If we create and write 1 byte to 2 files then write 2k bytes to the first file, the first file will allocate sectors 1, 3, 4, 5, and 6. This demonstrates external fragmentation as the file is not allocated using solely adjacent/sequential sectors.

2) Internal fragmentation is the result of a sector being allocated for a file, but the file not using the entirety of that sector. The average number of bytes wasted due to internal fragmentation for a randomly-sized file is 254 bytes. Therefore, the expected wastage for 10 files is 2,540 bytes.

3) Roughly the same as the read/write latency is the biggest issue, and we are not writing large enough blocks of data at a time to result in a noticeable bandwidth change.

4) We can store 31 files in our disk (the free space manager is treated as a hidden file). We can increase this limit by having an expandable directory. The last “file” in the directory could indicate where the next sector of the directory is located, and we could load that sector to continue reading the directory.

5) Yes, we do. If a file is open, any thread can write to it. The write function itself does not place a semaphore on the writing, so therefore the functions writing data must use semaphores to handle the write synchronization. This is because the write function only writes 1 byte. We believed that we would often require writing more than one byte at a time to the file, so we wished to place the semaphores around the redirected printf function to ensure that the data we wished to write was written correctly.