Lab 2:Buffer Manager/ Introduction to tinyFS

Due date: Tuesday, April 12, 11:59pm.

Lab Assignment

This is a group lab/homework assignment. You will be working on this assignment primarily outside of lab hours.

Overview

The primary goal for this lab is to introduce you to the lowest layer of DBMS functionality you will be using this quarter: the *paginated disk access layer*. In this course, the role of the paginated disk access layer is played by tinyFS, an emulator of a simple file system designed by Dr. Foaad Khosmood for the use in CSC 453: Operating Systems course. You are given the C .o library files of one implementation of tinyFS by Dr. Khosmood, adopted and extended for the use in our class. The new functionality includes the collection of library functions that allow for direct paginated access to the data stored in the TinyFS files in a way that is needed for the buffer manager to work properly.

Using tinyFS functionality, you will build a prototype buffer management system for a DBMS. At a later stage in the project, you will be able to extend the functionality of your prototype buffer manager to support FLOPPY.

TinyFS

TinyFS provides functionality to read and write blocks of fixed length to and from an emulated disk. Additionally, it provides functionality for creation of individual files inside the file system and their management (open, close, delete), which we will be using in our course project, as well as `read()` and `seek()` method for random file acces, which we won't be.

TinyFS is a disk *emulator*. It stores all the data placed into the file system in a single Linux file (its name is one of the settings described in the `tinyFS.h` header file - the default one is `tinyFS.dsk`).

Most of the functionality you will be using in this course is described in `libTinyFS.h` file, which is made available to you. Below we list the most important functions. Please consult that file for more in-depth descriptions of the functions.

`int tfs_mkfs(char *filename, int nBytes):` create a blank tinyFS system.

`int tfs_mount(char *filename):` mount a given tinyFS system. Allows one to work with multiple emulated disks in a single program.

`int tfs_unmount(char *filename):` unmount a given tinyFS system.

`fileDescriptor tfs_openFile(char *name):` opens a given tinyFS file.

`int tfs_closeFile(char *name):` closes a given tinyFS file.

`int tfs_writePage_size(fileDescriptor FS, unsigned int page, unsigned char * data, int size):` write a given number of bytes into a page identified by `FD, page` pair (file descriptor ID and the page Id).

`int tfs_writePage(fileDescriptor FS, unsigned int page, unsigned char * data):` write the given byte array to a page identified by `FD, page` pair (file descriptor ID and the page Id). *This is our bread-and-butter write() operation.*

`int tfs_readPage(fileDescriptor FS, unsigned int page, unsigned char *data):` read the contents of a disk page identified by file descriptor and page Id (within a file) into the byte array. *This is our bread-and-butter read() operation.*

`int tfs_deleteFile(fileDescriptor FS):` delete a file given its file descriptor.

`int tfs_readByte(file descriptor, char * buffer):` read one byte from the current file position. *This is for the sake of completeness. We will not be using this function.*

`int tfs_seek(fileDescriptor FS, int offset):` position the current file position to a given offset.

`int tfs_rename(char *oldname, char *newname):` rename a file.

`int tfs_readDir(void):` output the list of filenames stored on tinyFS.

`int tfs_readFileInfo(fileDescriptor FS):` outpus general file metadata.

`int tfs_numPages(fileDescriptor FD):` returns the number of pages in the file (including the inode page).

Buffer Manager Layer

Your task in this lab is to build a prototype buffer manager that:

1. maintains a buffer of disk blocks in main memory

2. communicates with the emulated disk to read blocks into the buffer and to flush blocks back to the emulated disk
3. implements one or more buffer replacement strategies discussed in class
4. provides a specified API to the next layer of the FLOPPY implementation.

Your implementation of the buffer manager is called a "prototype implementation" because at this stage, your implementation

1. will operate with empty pages: no functionality to actually put content on the pages will be implemented
2. won't provide the upper layer of FLOPPY with access to the page content

The above features are limitations of your prototype. To achieve the objectives of this lab, namely, (a) familiarity with tinyFS, and (b) implementation of an LRU and/or FIFO buffer replacement policy, presence of these features is irrelevant. They are, however, needed for proper communication with the next layer of FLOPPY: the record management layer. At a later stage of the project, you will extend the buffer manager to support these features.

Buffer Manager Data Dtructures and API

Data Structures. The primary data structure the buffer manager will operate on is a `Buffer`. A single variable of type `Buffer` will represent all data structures necessary to maintain tinyFS disk blocks in main memory. To simplify the definition of the `Buffer struct` we also provide a simple `struct` to represent a single buffer slot.

Note: You must define all data types specified in this document in your code and use them as prescribed. However, the internal structure of your data types is not limited to the one provided here. You may choose to modify the structure of the specified data types to fit the needs of your implementation, **as long as** your changes do not affect the function declarations from the buffer manager API.

Buffer Manager parameters. There are two key parameters that control the size of your buffer:

BLOCK_SIZE: this `#define`d constant represents the number of bytes in a single disk block, i.e., the size of a single buffer slot. BLOCKSIZE is defined in the tinyFS.h file provided to you. It is set to 2048 bytes at the moment.

MAX_BUFFER_SIZE: this constant represents the **maximal** total number of buffer slots in the main memory buffer. You need to `#define` this constant in your code.

Block structure. Here is a simple description of a single buffer slot, a.k.a. a block:

```
typedef struct {             /* single disk block  */
   char[BLOCK_SIZE] block;   /* block content      */
```

```
   fileDescriptor FD;         /* tinyFS file descriptor */
   int pageId;               /* tinyFS page Id within the file */
} Block;
```

Alternatively, you can define the `DiskAddress` `struct` as

```
typedef struct {
   fileDescriptor FD;  /* tinyFs file descriptor */
   int pageId;         /* tinyFS page Id with the file */
} DiskAddress;
```

and use this `struct` as a field in the `Block` structure.

Here, `FD` is the tinyFS file descriptor identifying uniquely an open file on disk, and `pageId` is the tinyFS page id for a given disk page in the file. `block` contains the actual contents of the disk page/block.

Buffer. A buffer is essentially an array of `BUFFER_SIZE` blocks with some extra information associated with each slot. At the very least, the following are needed:

- **timestamp**: both LRU and FIFO buffer replacement strategies need to know when the slot was accessed. If implementing both techniques, you may need two timestamps - one for the time when the disk block was placed in the slot, and one for the last access to the slot.
- **pin**: blocks can be pinned in the buffer. A pinned block cannot be replaced. Pin flag specifies is the slot contains a pinned block.
- dirty flag: a page is dirty if it has been updated, but has not been flushed to disk. A dirty flag specifies if the page is currently dirty. Dirty pages must be flushed to disk before being evicted from the buffer.

A simple structure for the `Buffer` data type is:

```
typedef struct {   /* Main Memory Buffer */
   char * database;               /* name of the disk file used with this buffer  */
   int nBlocks;   /* number of buffer slots                              */
   Block[BUFFER_SIZE] pages;      /* the buffer itself. stores content                */
   long[BUFFER_SIZE] timestamp;    /* timestamp for LRU, FIFO and other eviction strategies */
   char[BUFFER_SIZE] pin;         /* Pinned Page flags                                   */
   char[BUFFER_SIZE] dirty;       /* Dirty Page flags                                    */
   int numOccupied;               /* Number of occupied buffer slots                     */

} Buffer;
```

Note, both `pin` and `dirty` flags are boolean, and therefore, the `pin` and `dirty` arrays could've been bitarrays. However, for simplicity, we use byte arrays here.

API. The external API for the buffer manager consists of the following functions:

function declaration	brief description
`int commence(char * Database, Buffer * buf, int nBlocks);`	initialize the buffer
`int squash(Buffer * buf);`	graciously end the work of the buffer
`int readPage(Buffer * buf, DiskAddress diskPage);`	read access to the disk page
`int writePage(Buffer *buf, DiskAddress diskPage);`	write access to the disk page
`int flushPage(Buffer *buf, DiskAddress diskPage);`	flush the page from buffer to disk
`int pinPage(Buffer *buf, DiskAddress diskPage);`	pin the page
`int unPinPage(Buffer *buf, DiskAddress diskPage);`	unpin the page
`int newPage(Buffer *buf, fileDescriptor FD, DiskAddress * diskPage);`	add a new disk page

Note: Anywhere in the API above where `DiskAddress` arguments are used, they can be replaced with a pair of arguments `fileDescriptor FD, int pageId` if you do not want to define the `DiskAddress` type in your implementation.

Each function is described below.

`int commence(char * Database, Buffer * buf, int nBlocks)`

This function is to be called exactly once at the beginning of the work of any program that needs to employ the buffer. Paramters:

`char *Database`	name of the tinyFS file storing the disk (database)
`Buffer *buf`	the buffer created and initialized by the function
`int nBlocks`	number of buffer slots in the buffer

The function proceeds as follows:

1. **tinyFS file check.** If a tinyFS file with the given name exists, open it with tinyFS. Otherwise, create a new tinyFS disk with the given name.
2. **Buffer initialization.** Create and initialize the buffer data structure with `nBlocks` buffer slots. Associate it with the specified tinyFS disk. Load any necessary pages (if needed) into the buffer.

This function should return a status code that your implementation understands.

`int squash(Buffer * buf)`

This function is called exactly once at the end of the work with the buffer. It closes the buffer and finishes all the work. Parameters:

`Buffer *buf` the buffer to be closed/disposed of by the function

The function proceeds as follows:

1. Unpins all pinned pages.
2. Flushes all dirty pages to disk.
3. Clears all buffer slots.
4. Closes the tinyFS disk associated with the buffer.

5. Frees the memory occupied by the buffer.

The function returns a status code that you implementation understands.

Notice that this could have been simplified, by simply flushing all dirty pages to disk and then freeing the memory, but you should perform a graceful teardown of the buffer as specified above because I asked you nicely.

`int readPage(Buffer * buf, DiskAddress diskPage)`

This function provides **read access** to the specified block of the tinyFS disk. Parameters:

`Buffer *buf`	the buffer structure
`DiskAddress diskPage`	tinyFS id of the disk page to be read

The function shall proceed as follows.

1. **Buffer check.** Check if the given page is already in the buffer. If yes, update the page access timestamp according to the rules of your buffer replacement policy and stop. Otherwise, proceed below.
2. **Invoke buffer replacement policy.** Determine the buffer slot into which the the requested block will be loaded. The steps are:
   (a) If there are empty slots in the buffer, pick any slot.
   (b) If there are no empty slots, but there are unpinned pages, pick a page according to the currently operating buffer replacement policy.
   (c) If all pages are pinned, return an "all pages pinned" exit code.
3. **Evict the page.** If a page needs to be evicted, check if it is dirty. If it is, flush it to disk.
4. **Bring the page to buffer.** Replace the evicted page with the requested one in the selected buffer slot.
5. **Update page access timestamp.**

Also, if the file containing the disk page has not been opened yet, open it ud.

The function shall return an exit code that your implementation understands.

`int writePage(Buffer *buf, DiskAddress diskPage)`

This function provides the **write access** to the specified **tinyFS** block. Parameters:

`Buffer *buf`	the buffer structure
`DiskAddress diskPage`	tinyFS id of the disk page to be written to

This function will not actually change the content of the disk page. Instead, it simply performs the activities, necessary to let the buffer manager know that a write operation to the contents of a disk page has been performed. Thus, it shall proceed as follows.

1. **Find the page.** Find the disk page in the buffer. If it is not in the buffer, read it into the buffer.
2. **Mark the page dirty.**[1]
3. **Update page access timestamp.**

The write access to a disk block *does not force write the page back to disk.* This is done by the buffer replacement policy using the `flush` operation.

The function shall return an exit status code that your implementation understands.

`int flushPage(Buffer *buf, DiskAddress diskPage)`

This function flushes the given page back to disk. Paramters:

`Buffer *buf`	the buffer structure
`DiskAddress diskPage`	tinyFS id of the disk page to be flushed

The function shall proceed as follows:

1. **Find the page.** If the page is in the buffer, proceed. If it is not - exit with an error code (cannot flush a page that is not in the buffer).
2. **Write the page to disk** using tinyFS API.
3. **Unset dirty flag.** Once the page is flushed, it is no longer considered dirty. Unset the dirty flag.

Flushing the page is usually performed by the buffer manager itself (asynchronous write). Because of this, there is no need to change the last access timestamp when the page is flushed.

The function shall return a status code your implementation understands.

`int pinPage(Buffer *buf, DiskAddress diskPage)`

`int unPinPage(Buffer *buf, DiskAddress diskPage)`

These two functions pin and unpin the specified page in the buffer. Pinned pages cannot be removed from the buffer by the buffer replacement policies. Each function shall return a status code that your implementation understands. Pinning and unpinning a page shall also affect the last access timestamp.

int newPage(Buffer *buf, FileDescriptor FD, DiskAddress * diskPage)

This function creates a new disk page on the tinyFS disk. Parameters:

[1] If you want better encapsulation, create a pair of functions to set and unset the dirty flag and use them here and throughout the project.

`Buffer *buf`	the buffer structure
`FileDescriptor FD`	the descriptor of the file the new page belongs to
`DiskAddress * diskPage`	tinyFS id of the disk page created (returned by the function)

The function should discover the next "open" tinyFS page Id in a given file and create a tinyFS page with this ID on the tinyFS disk. It should also allocate a buffer slot for storing this page and store an empty page in the slot[2] Or, in other words, this function (a) allocates a slot for a page with the new pageId and then flushes that page to disk. At the end of the function's operation, the new page shall reside in the buffer, unless all pages in the buffer were pinned, in which case the function shall exit with an error message and not try to allocate/flush a new page.

This follows the idea that all communication with the disk goes through the buffer. If the buffer is inflexible, no disk operations with new pages can be performed.

Please make sure you properly maintain page numbers. The tinyFS implementation of `tfs_writePage()` will write content to *any* page number. If the number does not exist, it automatically creates that page. It does not check for page Ids being in order. This responsibility lies **with the buffer manager layer**.

The function shall return a status code that your implementation understands.

Buffer Replacement Policies

You are required to implement one buffer replacement policy, and you may implement multiple ones and tailor your API to specify in the `commence()` function, which buffer replacement policy is to be used for a given instance of the buffer.

Two buffer replacement policies that are straightforward to implement are LRU and FIFO. I recommend LRU. You can also implement any other policies discussed in the class or found in literature, for example, random selection of the buffer slot (Chris Lupo is a fan of it, for example).

You are responsible for maintaining, inside the `Buffer` structure, all data structures needed to support **all** your buffer replacement policies, and to maintain these data structures appropriately throughout the lifetime of the buffer.

Test API

In addition to the buffer manager API to be used by the next layer of ArferDB, you should also implement a number of functions that provide information about the current state of the buffer. These functions will be used for testing purposes.

Implement the following functions:

function declaration	brief description
`void checkpoint(Buffer * buf);`	print the current state of the buffer
`int pageDump(Buffer *buf, int index);`	print the contents of a buffer slot
`int printPage(Buffer *buf, DiskAddress diskPage);`	print the contents of a page from the buffer
`int printBlock(Buffer *buf, DiskAddress diskPage);`	print the contents of a page on disk

[2] Your current system operates with all empty pages, but eventually, actual content will be added to them. For this function though, you will always put an empty page into the buffer slot - except later, that page will have some header information on it.

```
void checkpoint(Buffer * buf);
```

This function shall take as input the pointer to the buffer structure and shall print out the information about the state of the buffer. Essentially, for each disk slot, the function shall print the following information:

- The tinyFS blockID of the page in the slot. If the slot is empty, indicate that as well.
- All timestampts associated with the slot that are tracked.
- The value of the pin flag (i.e., whether the page is pinned).
- The value of the dirty page flag (i.e., whether the page is dirty).
- Values of any other attributes you have associated with the buffer slot.

General information about the buffer (which tinyFS disk it is associated with, how many slots are occupied/empty, and any other data you store in the buffer structure) should also be printed.

```
int pageDump(Buffer *buf, int index)

int printPage(Buffer *buf, DiskAddress diskPage)

int printBlock(Buffer *buf, DiskAddress diskPage)
```

These three functions print the contents of individual pages. They should have consistent output format. However, they differ in how the data to be printed is retrieved.

- `pageDump()` takes as input the pointer to the buffer and the index of a buffer slot. If the specified index exists, this function prints out the contents of the page at the buffer slot.
- `printPage()` takes as input the pointer to the buffer and the tinyFS blockId (`diskPage`). It shall find the page with the given block id in the buffer and output its contents from the buffer. If a page with the block Id does not exist on disk - report an error message. If the page exists on disk, but is not in the buffer, print `"Page not in the buffer"` message.
- `printBlock()` takes as input the pointer to the buffer and the in tinyFS blockId. Unlike `printPage()`, this function prints the contents of the block with the given block id *directly from the tinyFS disk*. ***This is the only function that is allowed to access tinyFS disk contents outside the communication with the buffer.***

These three functions might not contribute much to the current assignment (we are not modifying the contents of the tinyFS pages, so page dumps are not necessary). However, they will be very useful for us at the next stage. In particular, comparing results of `printPage()` vs. `printBlock()` you can check up on the work of the `writePage()` and `flushPage()` functions and validate them.

Test Harness

To illustrate your work, provide a simple test harness that can be used in a batch mode to process buffer management commands. The test harness takes as input a name of a file that contains a number of buffer manager commands, parses them one by one and executes them.

For simplicity, the test harness is responsible for maintaining only one buffer at a time (although it can create a buffer, destroy it, create another one, and so on multiple times).

Buffer management commands basically mirror the buffer manager and the test APIs. The commands are:

Command	API call	Explnation
`start <DBName> <Size>`	`commence()`	Initialize buffer with `<Size>` slots for tinyFS disk `<DBName>`
`end`	`squash()`	Finish the work of the currently active buffer
`read <Page>`	`readPage()`	Read access to the tinyFS block `<Page>`
`write <Page>`	`writePage()`	Write access to the tinyFS block `<Page>`
`flush <Page>`	`flushPage()`	flush tinyFS block `<Page>`
`pin <Page>`	`pinPage()`	pin tinyFS block `<Page>` in the buffer
`unpin <Page>`	`unpinPage()`	unpin tinyFS block `<Page>` in the buffer
`new <NPages>`	`newPage()`	create `<NPages>` new tinyFS pages
`check`	`checkpoint()`	print the current status of the buffer

Name your test harness program `bufferTest.c`. The program shall read each line of the input file, parse the command, execute the operation, echo the operation to `stdout` and provide the information about status of the operation (successful, failed, etc.).

Example. Consider the following test file `bufferTest1.txt`:

```
start Foo.disk 5
new 20
read 1,1
read 1,2
read 2,1
read 2,2
read 1,3
pin 1,1
check
read 2,2
read 1,1
write 1,1
write 2,1
read 3,1
read 3,2
unpin 1,1
check
read 2,1
read 4,1
read 4,2
read 4,3
read 4,4
read 4,5
check
end
```

Here, the first number in `read` and `write` commands is the file descriptor number. Later, you may be able to replace it with other means of file identification.

This test creates a buffer with five (5) slots, associates it with a tinyFS disk file `Foo.disk` (presumably a new file), creates 20 new pages and proceeds with read, write, pin and unpin requests. There are three checkpoints.

Submission Instructions

Your submission shall contain the following:

- All `.c` and `.h` files needed to run your buffer manager.
- `bufferTest.c`: the test harness.
- `README` file that contains:
  - Team name, names of all team members
  - Breakdown of contributions of each team member[3].
  - Compilation and running instructions.

Submit using the following command:

```
$ handin dekhtyar 468-lab02 <files>
```

[3] This won't affect individual grades, but I would like to see how each team distributes the work.