National University of Singapore
School of Computing

CS3223: Database Systems Implementation
Semester 2, 2009/10

Assignment 1: Buffer Manager (10 marks)

Due: 11:59pm February 22, 2010 (Monday)

1 Introduction

In the previous assignment, we learned how to install `PostgreSQL`, set up a database cluster (using `initdb`), how to start/stop the `PostgreSQL` server process (using `pg_ctl`), how to create a specific database (using `createdb`), and how to query this database (using a client program `psql`). Now, we start with some actual C code hacking! Currently, `PostgreSQL` 8.4.2 uses a variant of the CLOCK replacement policy. In this assignment, you will modify the buffer manager component of `PostgreSQL` to implement the LRU replacement policy. The actual coding for this assignment is minimal, given the highly organized and modular PostgreSQL code (the code for the buffer manager policy is cleanly separated from the rest of the code base). However, you have to figure out what code to modify, which is non-trivial!

2 Overview of PostgreSQL

The following is a brief introduction to key aspects of `PostgreSQL` that are relevant for this assignment.

2.1 Process Architecture

`PostgreSQL` is implemented using a simple "process per user" client/server model. In this model there is one client process connected to exactly one server process. Each database cluster (containing a collection of databases) on a single host is managed by a single master process that is called `postmaster`. The `postmaster` process can be started directly in one of two ways:

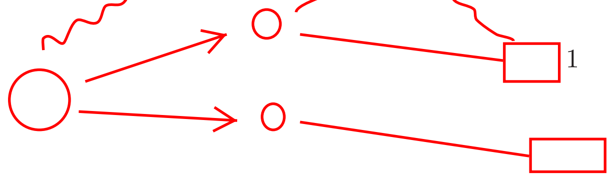
postmaster -D /home/sadm/data

or

postmaster -D /home/sadm/data > logfile 2>&1 &

As we learned in the previous assignment, the `postmaster` process can also be started indirectly using the `pg_ctl` command as follows: *pg_ctl -D /home/sadm/data -l logfile start.* This will start the `postmaster` server process in the background and put the output into the named log file. The -D option has the same meaning as when invoking postmaster directly.

Each time a frontend client process (e.g., `psql` application program) requests a connection to the database server, `postmaster` will spawn a new backend server process called `postgres`. From that point



on, the frontend client process and the backend server (i.e., `postgres`) communicate without intervention by the `postmaster`. Hence, the `postmaster` is always running, waiting for requests, whereas frontend and backend processes come and go.

2.2 Shared-memory Data Structures

In general, there could be multiple backend processes accessing a database at the same time. Therefore, access to shared-memory structures (e.g., buffer pool data structures) needs to be controlled to ensure consistent access and updates. To achieve this, `PostgreSQL` uses *locks* to control access to shared-memory structures by following a locking protocol: (before accessing a shared-memory structure, a process needs to acquire a lock for that structure; and upon completion of the access, the process needs to release the acquired lock) There are two basic types of locks available: *read/shared locks* and *write/exclusive locks*. A read lock should be used if read-only access is required; otherwise, a write lock should be used.

2.3 Buffer Manager

Thre are two types of buffers used in `PostgreSQL`: a *shared buffer* is used for holding a page from a globally accessible relation, while a *local buffer* is used for holding a page from a temporary relation that is locally accessible to a specific process. This assignment is about the management of `PostgreSQL`'s shared buffers.

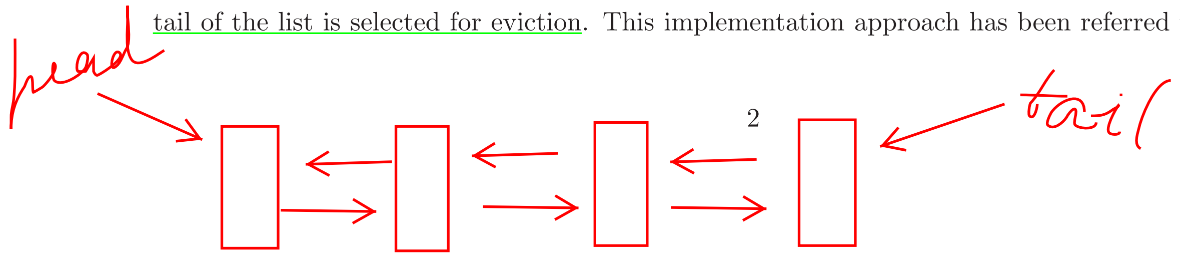
Initially, all the shared buffers managed by `PostgreSQL` are maintained in a *free list*. A buffer is in the free list if its contents are invalid and is therefore of no use to any client process. Whenever a new buffer is needed, `PostgreSQL` will first check if a buffer is available from its free list. If so, a buffer from the free list is returned to satisfy the buffer request; otherwise, `PostgreSQL` will use its buffer replacement policy to select a victim buffer for eviction to make room for the new request. `PostgreSQL` 8.4.2 currently implements a variant of the CLOCK replacement policy.

In `PostgreSQL`, when a record is deleted (or modified), it is not physically removed (or changed) immediately; instead, `PostgreSQL` maintains multiple versions of a record to support a *multiversion concurrency control* protocol (to be covered in the later part of our lectures). Periodically, a *vacuuming* process will be run to remove "obsolete" versions of records that can be safely deleted from relations. If it happens that an entire page of records have been removed as part of this vacumming procedure, the buffer holding this page becomes invalid and is returned to the free list.

`PostgreSQL` also runs a background writer process (called `bgwriter`) that writes out dirty shared buffers to partly help speed up buffer replacement.

3 Implementing LRU

A simple approach to implement the LRU policy is to use a doubly-linked list to link up the buffer pages such that a page that is closer to the front of the list is more recently used than a page that is closer to the tail of the list. Thus, whenever a buffer page is referenced, it is moved to the front of the list; and whenever a replacement page is sought from the list, the unpinned buffer page that is closest to the tail of the list is selected for eviction. This implementation approach has been referred to as the *Stack*



LRU;unpinned

LRU method in some operating systems textbooks; here, the top and bottom of the stack correspond to the front and tail of the linked list, respectively. You will implement this Stack LRU approach for this assignment.

4 Getting Started

First, obtain the required assignment files as follows:

1. Log in to your zone account.
2. `wget http://www.comp.nus.edu.sg/~cs3223/assign/assign1.zip`
3. `unzip assign1.zip`

The `assign1` sub-directory created in your home directory contains the following directories and files:

- `MyCode/`
  - `bufmgr.c`: This is a modified version of the original file in `src/backend/storage/buffer/bufmgr.c`. You will need to use this modified version for this assignment; however, you do not need to make any further modifications to this file.
  - `freelist.c`: This is a modified version of the original file in `src/backend/storage/buffer/freelist.c`. You will need to make further modifications to this file for this assignment.
  - `buftest.c` and `stubs.c`: These are test programs.
  - `testcases/`: Contains test cases for test programs.
  - `makeresults.sh`: Script for running test programs.
  - `submit.sh`: Script to submit assignment deliverables.
  - `freelist.original.c` and `bufmgr.original.c`: original versions of freelist.c and bufmgr.c.
  - `Makefile`
- `PerformanceEvaluationScripts/`
  - `DoInitDB.sh`: Script to initialize database cluster
  - `PrepareTables.sh`: Script to create a database called "test" and create some tables
  - `prepare.idx.sql` and `prepare.sql`: Scripts for creating tables
  - `R` and `S`: Data files for loading tables
  - `Query.sh`: Script for running queries
  - `query.sql`: Sample test query file

Note that the instructions for this assigment assume that you have installed verison 8.4.2 of `PostgreSQL` in the default `/usr/local/pgsql` directory. You should have also updated your `.bash_profile` file by updating the PATH environment variable with the path `/usr/local/pgsql/bin` as described in *Assignment 0*. To ensure that you are indeed using your own installed version of *postgres* located at `/usr/local/pgsql/bin` (and not the default version installed by SoC located at `/usr/bin`), check using the command `which postgres`.

5 What to do

This section provides some guidelines on how you can go about implementing the LRU replacement policy in `PostgreSQL`.

1. Before you begin making changes to `PostgreSQL`, you should examine the existing code to understand how the buffer manager (specifically its CLOCK policy replacement) is being implemented. The existing code is not extremely clear, but it is understandable. It may take you a few hours (or more) to digest it. Since understanding the existing code is a significant part of the assignment, the TA and Professor will not assist you in your understanding of the code base (beyond what we discuss here). The actual buffer manager code is neatly separated from the rest of the code base. Its files are located in the `src/backend/storage/buffer/` and `src/include/storage/` directories. The two main files of interest for this assignment are **`bufmgr.c`** and **`freelist.c`**. Modified versions of these two files (to be used for this assignment) are given in the `MyCode/` directory. While **`bufmgr.c`** contains the implementation of the buffer manager, we are only interested in **`freelist.c`**, which defines the buffer replacement policy. The following is a brief description of some of the relevant files for this assignment:

    - `MyCode/freelist.c`: Has functions that implement the replacement strategy. This is the only file you need to modify.
    - `MyCode/bufmgr.c`: Implements the buffer manager.
    - `src/include/storage/buf_internals.h`: Contains the definition of each buffer frame (called *BufferDesc*). Most of the fields in *BufferDesc* are managed by other parts of the code.
    - `src/backend/storage/buffer/buf_init.c`: Some initialization of the buffer frames occur in this file. However, you should do all your initialization in `freelist.c` (see `StrategyInitialize` routine in `freelist.c`).
    - `src/backend/storage/buffer/README`: Useful description of the *Strategy* interface implemented by `freelist.c`.

2. For this assignment, the main file that you need to modify is `MyCode/freelist.c`. The given file is a slightly modified version of the original file `src/backend/storage/buffer/freelist.c`. You will need to make further modifications to this file to implement LRU (instead of CLOCK) replacement policy.

    As explained in the previous section, the LRU Stack implementation maintains the relative recency of accessed buffers using a linked list. Thus, whenever a buffer is accessed, its "position" within the stack needs to be adjusted. This adjustment can be classified into four cases:

    (a) If an accessed page is already in the buffer pool, then its buffer needs to be moved to the top of the stack.

    (b) If an accessed page is not in the buffer pool and a free buffer is available to hold this page, then the selected buffer from the free list needs to be inserted onto the top of the stack.

(c) If an accessed page is not in the buffer pool and the free list is empty, then the selected victim buffer needs to be moved from its current stack position to the top of the stack.

(d) If a buffer in the buffer pool is returned to the free list, then the buffer needs to be removed from the stack.

To help you implement the above stack adjustment operation, we have defined a new function in `freelist.c`:

```
void StrategyAdjustStack (int buf_id, bool delete)
```

The `StrategyAdjustStack` is used to adjust an input buffer, which is uniquely identified by its buffer index number given by `buf_id`, within the LRU stack. The boolean flag `delete` is used to indicate whether the buffer is to be deleted from the stack. The buffer will be removed from the stack if `delete` has a `true` value (i.e., case (d)); otherwise, the buffer will be moved/inserted to the top of the stack (i.e., case (a), (b), or (c)). You should refer to `MyCode/bufmgr.c` for an example of how `StrategyAdjustStack` is being used.

Besides implementing the `StrategyAdjustStack` function, you are free to make any necessary changes to `freelist.c`. Hint: you will probably need to modify the functions `StrategyGetBuffer` and `StrategyFreeBuffer` in `freelist.c` to make use of `StrategyAdjustStack`.

3. The file `MyCode/bufmgr.c` is a modified version of the original file `src/backend/storage/buffer/bufmgr.c`. Although you do not need to make further modifications to this file, it is important that you use this modified file (instead of the original version) for this assignment. A key modification made in this file is a call to the `StrategyAdjustStack` function within the `BufferAlloc` function. This is necessary for the correctness of the LRU replacement policy.

4. Edit `MyCode/Makefile` to make sure that the `PSQLPATH` variable is correctly set to the path of your `PostgreSQL` source tree. You can use this `Makefile` to compile your `freelist.c` as follows: `make freelist.o`.

5. Once your `freelist.c` compiles correctly, you are now ready to test your implementation using the test programs (`MyCode/buftest.c` and `MyCode/stubs.c`) that we provide. These programs tests the correctness of your buffer policy (`freelist.c`) by bypassing the `PostgreSQL` code and directly testing your `freelist.c` implementation.

testing

The file `buftest.c` provides a sequence of pin/unpin operations and lets you observe how your buffer frames get replaced using a small number of frames and disk blocks. The status of the buffers are displayed after each pin/unpin step using the following notations:

- Pages are labeled with one letter ('a' through 'e').
- A letter enclosed within square brackets [] represents an unpinned page (that is available for replacement), while a letter enclosed by parentheses () represents a pinned page.
- An empty buffer frame is represented by the pound symbol, #.

For example, $(d)[a][b](e)$ means that the pages a,b,d, and e are in the buffer pool, where d and e are the only pinned pages. Note that the ordering of the displayed buffer contents does not matter; what matters is the correct combination of pages shown after each operation.

The file `stubs.c` is provided to allow the code to compile without all the `PostgreSQL` code. Your code MUST compile (and run properly) with the code we provided.

To run the test programs, execute `./makeresults.sh`. The `makeresults.sh` script will run each of the 10 test cases in `MyCode/testcases/testcase*.c` and generate an output file. The generated results are `./testcases/testcase*.c.soln`. You do not need to submit these result files. You can also create your own additional test cases to test your code more thoroughly.

reinstall

6. After you have fully tested your `freelist.c` using the test programs, you are now ready to install your LRU implementation in the actual `PostgreSQL` source tree and perform some experiments. To install your changes, you need to copy both `MyCode/freelist.c` and `MyCode/bufmgr.c` into `postgresql-8.4.2/src/backend/storage/buffer`, and re-install `PostgreSQL`. This can be done by simply running `make mypgsql` within `MyCode/` directory.

7. After successfully installing your files with `PostgreSQL`, you are now ready to run some experiments using the scripts in `PerformanceEvaluationScripts/` to create tables, load data, and run queries.

   - `cd PerformanceEvaluationScripts`
   - `./DoInitDB.sh $HOME/data`
     This step initializes a new database cluster directory at `$HOME/data`. This step can be skipped if you have previously completed *Assignment 0*.
   - `./PrepareTables.sh $HOME/data`
     This step creates a new database called `test` in the database cluster directory at `$HOME/data` and creates two tables with data.
   - `./Query.sh $HOME/data`
     This step runs a test query on the database `test` created in the database cluster directory at `$HOME/data`.

6 Debugging PostgreSQL

As explained in Section 2, `PostgreSQL` is a client/server system, where a frontend client process (e.g., `psql`) interacts with a backend server process (i.e., `postgres` that is created by the master server process `postmaster`) via inter-process communication. There are two methods of debugging `PostgreSQL`. The first and simpler method is to print out debugging information from within the server process. The second and more sophisticated method is to use a debugger tool (e.g., gdb) to insert breakpoints and watchpoints within the code.

For this assignment, it should be adequate to use the first debugging method.

6.1 Printing Debugging Information

To display debugging information on the screen, you can use `PostgreSQL`'s logging/printing feature: `elog(DEBUG1,<format>,<arg>)`. Please see `bufmgr.c` for example uses of this routine.

6.2 Using gdb

To debug `PostgreSQL` using gdb, you should first configure and compile `PostgreSQL` with debugging enabled as follows:

```
./configure --enable-debug --enable-depend
```

After `PostgresSQL` has been compiled and installed, you can start the debugging as follows. First, start gdb with `gdb postgres`. Then within gdb, execute `run -B 10 -D $HOME/data`. Here, the ``-B 10'' option limits the size of the buffer pool to 10 page frames.

6.3 How to kill postmaster process

As explained in *Assignment 0*, the normal procedure to shutdown PostgreSQL is using the pg_ctl command as follows: `pg_ctl stop -D data`.

In some situations where the shutting down has not been done properly, you may have to resort to a more brutal way to kill the server process using the kill command as follows:

```
kill -INT ‘head -1 $HOME/data/postmaster.pid‘
```

7 What & How to Submit

For this assignment, you will need to submit two files:

1. `freelist.c`: This is your version of freelist.c implementing LRU policy.
2. `README.txt`: This is a short text file providing the following information:
   (a) The names of the team members
   (b) A concise description of the key changes that you have made to implement LRU. You should highlight any new structures used and any subtle points about your code.

You should submit the above two files before the deadline using the script MyCode/submit.sh as follows:

- `cd MyCode`
  Make sure that both your *README.txt* and *freelist.c* files are in the current directory, and that you have disabled any debugging printing statements in your code.
- Run the script: `./submit.sh SUNFIRE-USERID CS3223-TEAM-NUMBER`
  This script will email out your *README.txt* and *freelist.c* files via your sunfire account to your CS3223 TA. You will need to provide your sunfire account userid and your assignment team number as parameter values for this script, and then enter your sunfire's account password when prompted to do so. If you see the message ''*Permission denied (gssapi-keyex,gssapi-with-mic,publickey,keyboard-interactive)*'', it means that your attempt to submit via sunfire has failed and you will need to repeat this step again.

Late submission penalty: There will be a late submission penalty of 1 mark per day up to a maximum of 3 days. If your assignment is late by more than 3 days, it will not be graded and your team will receive 0 credit for the assignment. Do start working on your assignment early!

8 Acknowledgement

Thanks to Joe Hellerstein for permission to adapt his assignment material.