COMP9315 DBMS Implementation Final Exam

[Instructions] [Notes] [PostgreSQL] [C]
[Q1] **[Q2]** [Q3] [Q4] [Q5] [Q6] [Q7] [Q8]

Question 2 (14 marks)

In this question, you need to complete a program to simulate a nested-loop join that uses a buffer pool with a "clock-sweep" replacement strategy.

The code for this question is in the `q2` directory, which contains:

- `bnl.c` ... main program for buffered nested loop
- `bufpool.c` ... implementation of a buffer pool ADT
- `bufpool.h` ... interface definitions for buffer pool ADT
- `Makefile` ...for building the program
- `tests/` ... directory containing test cases

The program simulates a nested loop join via a command called `bnl`, using parameters supplied on the command line:

- `OuterPages` ... the number of data pages in the join outer relation
- `InnerPages` ... the number of data pages in the join inner relation
- `nSlots` ... the number of buffers in the buffer pool

The main program implements the following:

```
initialise pool with nSlots buffers
for pidR in 0 .. OuterPages-1 {
    pageR = request(R,pidR)
    for pidS in 0 .. InnerPages-1 {
        pageS = request(S,pidS)
        ... check for join matches ...
        release(S,pidS)
    }
    release(R,pidR)
}
print buffer pool state and usage stats
```

As it simulates the join execution, it computes the following statistics:

- total number of *requests* on the buffer pool
- total number of *releases* on ththe buffer pool
- total number of *hits* on the buffer pool
- total number of *reads* into the buffer pool

You can build the `bnl` executable using the `make` command. The program runs the simulation and then prints the final state of the buffer pool and the usage statistics. Since the simulation doesn't consider output, we don't need to allocate a buffer for output. Since we never change the values in the pages we read, the #writes count will always be zero.

Some (slightly edited) examples of use from a working instance of `bnl`:

```
$ ./bnl
Usage: ./bnl OuterPages InnerPages Slots

$ ./bnl 3 2 1
-- not enough buffers to execute the join
Failed to find slot for S00

$ ./bnl 3 4 5
Frames:     [00] [01] [02] [03] [04]
Contents:    S03  R02  S00  S01  S02
PinCount:      0    0    0    0    0
Popularity:    2    2    2    2    2
Clock: 1

#requests: 15
#releases: 15
#hits    : 4
#reads   : 11

$ ./bnl 3 4 2
-- minimum buffers to execute the join
-- every request results in a read
Frames:     [00] [01]
Contents:    R02  S03
PinCount:      0    0
Popularity:    1    2
Clock: 0

#requests: 15
#releases: 15
#hits    : 0
#reads   : 15

$ ./bnl 5 4 9
-- enough buffers to hold relations in pool
-- each page is read exactly once
Frames:     [00] [01] [02] [03] [04] [05] [06] [07] [08]
Contents:    R00  S00  S01  S02  S03  R01  R02  R03  R04
PinCount:      0    0    0    0    0    0    0    0    0
Popularity:    2    3    3    3    3    2    2    2    2
Clock: 0

#requests: 25
#releases: 25
#hits    : 16
#reads   : 9
```

Assume that the buffer pool initially starts empty and that empty slots are used first, before any replacement is considered. Assume also that buffers and pages are indexed starting from 0.

By default, the program just shows the final state of the buffer pool and the usage statistics. If you wish to monitor it's progress (perhaps for debugging), there are some commented-out debugging statements in `bufpool.c`. You can add them back by changing all of the `#if 0` to `#if 1`. Don't forget to restore them before `check`ing or `submit`ting; otherwise all your tests will fail.

A detailed example of output from `bnl` is available in the file `trace.txt` in the `q2` directory. This shows the output if you turn on all of the debugging statements, as shown above. It also shows some debugging internal to `findVictim()`; you can add something like this to your code if you want, but you should remove or comment it before submission.

Your Task: complete the `findVictim()` function in `bufpool.c`

The `findVictim()` function determines the slot to be replaced when a request is made for a page not currently in the buffer pool, and when all of the buffers in the pool are full. To do this it uses a "clock hand" (`pool->clock`) which does a circular scan of the buffer pool. It uses the following approach to choose the "victim":

- a buffer with a non-zero popularity count cannot be evicted
- a buffer with a non-zero pin count cannot be evicted
- if a buffer is considered, but not evicted, its popularity count is decremented by 1
- a buffer's popularity count cannot drop below 0, and is capped at `MAX_USAGE` (3)

Popularity counts are updated as follows:

- the popularity count is incremented by 1 when a page is requested
- the popularity count is incremented by 1 when a page is released

The second may seem counter-intuitive, but it gives us a handle on when the page is being "looked at". Note that above two popularity count updates are already implemented in the `request_page()` and `release_page()` functions. The popularity count is also, as noted above, decremented when it is considered by `findVictim()`, but not selected.

The `findVictim()` function behaves roughly as follows:

```
while (haven't found a victim*) {
    check the buffer under the clock hand
    if it has zero pin and popularity counts, found a victim
    decrement popularity count, if > 0
    advance clock hand
}
*if no victim found after enough attempts, return NONE
write buffer out if dirty
advance clock hand
```

Note that the above is an abstract view of the process; you can't literally follow the control structures above and expect to get a working solution. Note also that the clock hand should always be left one slot beyond the selected victim.

The `tests` directory contains a number of test cases for the `bnl` program. You can execute an individual test case by running a command like

```
$ sh tests/01.sh
```

which runs the "`3 4 5`" example above.

You should also be able to devise your own test cases easily enough.

To help you check whether your program is working correctly, there is a script called `run_tests.sh` which will run the program against all of the tests and report the results. It will also add the output from your program into the `tests` directory; comparing your output

against the expected output might help you to debug your code. You can run the testing script as:

```
$ sh run_tests.sh
```

Once your function is working (passes all tests), follow the submission instructions below. Even if it fails some (or even all) tests, you should submit because you can get *some* marks. If your program does not compile, or if you simply submit the supplied code, then your "answer" is worth zero marks.

Submission Instructions:

- Type your answer to this question into the file called `bnl.c`
- Submit via: **`give cs9315 exam_q2 bufpool.c`**
  or via: Webcms3 > exams > Final Exam > Submit Q2 > Make Submission

End of Question