Theme Song: Cooking By The Book

BumbleBase is a disk-oriented database system, so at its foundation it has to manage data files on disk. In this assignment, we will build the machinery that will handle the low-level operations on our data files that will enable us to easily develop higher-level logic in future assignments.

Note: this assignment does not require you to write very many lines of code. However, the lines that you do write require a very detailed understanding of how a pager works. Before you begin coding, make sure you understand as much of the existing code as you can and understand how pages are being stored in memory - **a strong understanding of how pages are stored and implemented is absolutely integral to succeeding in the rest of the course; make sure you understand it as well as possible!** To that end, we highly recommend you attend the Pager Gearup and come to TA hours if you are stuck!

---

1 Background Knowledge

1.1 Mutual Exclution

Before we go on, we should explain the fundamentals of locking; you will encounter locks briefly in this assignment and revisit them in great detail in later assignments.

1.1.1 Mutexes

A **mutual exclution lock** (mutex) is a kind of lock that ensures only one thread/process is accessing a resource at any given point in time. Once someone has acquired the lock, everybody else must wait for the lock to be unlocked before they can acquire the lock. A common problem with locking protocols is running into **deadlocks**, which is essentially when two threads are waiting for each other to free their locks, while holding the lock that the other thread needs.

1.1.2 RWLocks

Another way that we handle concurrent page accesses is in the form of **readers-writers locks** (RWLocks). A RWLock is a kind of lock that allows many readers to access a resource at a time, but only one writer. In other words, if a thread is granted a read lock, they can be sure that everybody else with a lock is also a reader, and if a thread is granted a write lock, they can be sure that they are the only one with any sort of lock, both read or write.

This is slightly better than a regular mutex lock since it allows more than one reader to access a resource at a time. If your database experiences a very read-heavy load, this can decrease downtime as pages unnecessarily wait for resources to show up. The following is an example of how RWLocks are used:

```
lock := sync.RWLock{}
lock.RLock() // Gets the lock, no problem
lock.RLock() // Also no problem!
// lock.Lock() // Cannot get a write lock right now, since there are readers.
lock.RUnlock()
lock.RUnlock() // Now, lock is fully unlocked
lock.Lock() // So, we can get a write lock!
// lock.Lock() // However, because there is a write lock, nobody else can grab write locks
// lock.RLock() // ... or read locks
lock.Unlock() // Be sure to always unlock your locks when you're done!
```

```
// lock.Unlock() // But never too many times!
lock.Lock()
defer lock.Unlock() // It can be useful to say: "unlock this lock when this function exits".
```

1.2 Pages

Given a set of records, we have many potential strategies of storing them in a file. One strategy is called a heap file, where the database is represented as a single, unorganized file, and records are inserted wherever there is space for it. However, heap files are not particularly organized; if you built an index over the entire file, there won't be a good way to use it without reading the entire index into memory. As you insert and delete data, fragmentation will lead to wasted space. And so on.

Although we like to think of data in terms of bits and bytes, our disk and operating system handle data in units of pages. Formally, a **page** is the smallest amount of data that can be guaranteed to be written to disk in a single atomic operation, traditionally fixed at a uniform size such as 4096 bytes. In other words, our operating system exclusively read from and write to disk in blocks of 4096 bytes. We can leverage this feature of our operating system to create a more organized and flexible storage scheme.

It's worth noting that these days, multiple page sizes are supported on various architectures. Additionally, database systems are free to set their own page sizes to multiples of the underlying system's page size. However, these design choices come with various tradeoffs. **In BumbleBase, database pages will follow the page size of the hardware and operating system, i.e. database pages will contain 4096 bytes** (`PAGESIZE` in `pager.go`).

Each page is given a unique page id that can be used to determine the offset in the file where it belongs. In our scheme, the 0th page takes up bytes 0-4095, the 1th page takes up bytes 4096-8191, and so forth. In general, other systems may have other ways of mapping page id to page offset (using a page directory or other data structure).

1.2.1 Disk Manager

To store a database on disk, we can either write directly to block devices, or have the operating system do this for us. The former is a pain to implement and port to other hardware platforms. Instead, we will do the latter. Since the operating system doesn't know anything about our data model, database files are treated just like normal files, meaning that all the regular read and write operations are available for us to use.

In BumbleBase, each table is stored as a separate file. To read a specific section of the table, we have to seek to the appropriate byte offset within the file, then read in the specified number of bytes. To minimize the number of blocks we read from disk, *the byte offset should be page-aligned (i.e. a multiple of the page size)*. We should read in one page's worth of data at a time. When writing to our database, we do the same thing: seek to the correct byte offset in the file, then write a page's worth of bytes to the file.

Here's an example to illustrate: consider a particular tuple, A, which exists in page P. To select A, we first read in the entire page P, and then search P for A. It might seem wasteful to read in so much extra data if we only need one tuple, but as we'll see soon, doing this makes it easier to cache data that we've read in predictable and reusable chunks and leverages the operating system's underlying construction for speed.

This functionality is provided for you. However, it's good to have a firm grasp of how we're writing data to and from pages.

1.2.2 Buffer Cache & Page Table

It would be wasteful to read a page into memory and immediately write it out; if we've already gone through the trouble of reading a page in, we should keep it around in case it is needed again in the near future. After all, some pages are often read very often, or might be read multiple times in a particular transaction. This process is called **caching**.

The cached data will be stored in a buffer that we first pre-allocate, then fill. Critically, the buffer must be page aligned, such that the data stored in a page-sized block is actually stored as a single page, and not sharded across two. The buffer is represented as a series of **frames**, each large enough to hold one page. When a page is brought into memory, it will reside in a frame.

To keep track of which pages are in which frames, we have a **page table**. A page table is a map from page ids to frame ids, and indicates where pages are currently being stored in memory.

Let's consider an example. In the following diagram, our program has requested pages 0, 1, 2, and 3. To the external program, it seems as through these pages physically exist in order - this is known as the "logical memory" abstraction. The pages *actually* live in one of 8 frames that we've allocated. To figure out which frame each page lives in, we consult the page table, which maps the page ids to the frame its stored in.

1.2.3 Pinning Pages

While a page is in use, we want to make sure that it doesn't get evicted from the cache. Pinning a page marks it as in use and prevents it from being evicted. Pinning is usually implemented as a thread-safe counter of the number of threads currently using the page. This is what we do as well (see the `pinCount` attribute).

As an example, let's say we request a new page from the pager. Since this is a new page, we know that we're the only person using it, so we set the pin count to 1. Once we're finished, we decrement the pin count. If nobody else had taken the page in the meantime, we would be clear to free the page since the pin count would be 0. However, if someone else *did* take the page in the meantime, then the reference count would be greater than 0, meaning we cannot free the page.

1.2.4 Getting and Evicting Pages

Whenever we get a page that is not already in our page table, we want to grab it from disk and keep track of it in our page table. Initially, all of our empty frames are stored in a **free list**; while this free list is non-empty, we should be using one of our free frames to store new pages. However, once all of our frames are full, we should evict old pages from memory and use those newly freed frames to store our new pages.

In addition to the free list, we also keep track of a **pinned list** and an **unpinned list**. While a page is in use, it should be in the pinned list, and while it is not in use, it should be in the unpinned list. Critically, when a page shifts from being in use to not being in use (*i.e.* the pin count drops to 0), it should be removed from the pinned list and inserted into the end of the unpinned list. Symettrically, when a page shifts from being not in use to being in use (*i.e.* the pin count increases to 1 from 0), it should be removed from the unpinned list and inserted into the end of the pinned list. A page should never move to the free list (think about why). Make sure you're able to wrap your head around what these three lists are used for!

When a new page is requested and the free list is full, we need to evict an old page to create space. Notice that the first page in the unpinned list is most likely the best candidate to evict; it is not currently in use, and since we add new unpinned pages to the end of the unpinned list, it is the oldest such page. We have actually implemented what is known as a **least recently used cache**, or an **LRU cache**. This is one of many possible caching strategies we could have used, but is simple and effective for most use cases. Thus, we evict this page and use that frame for our

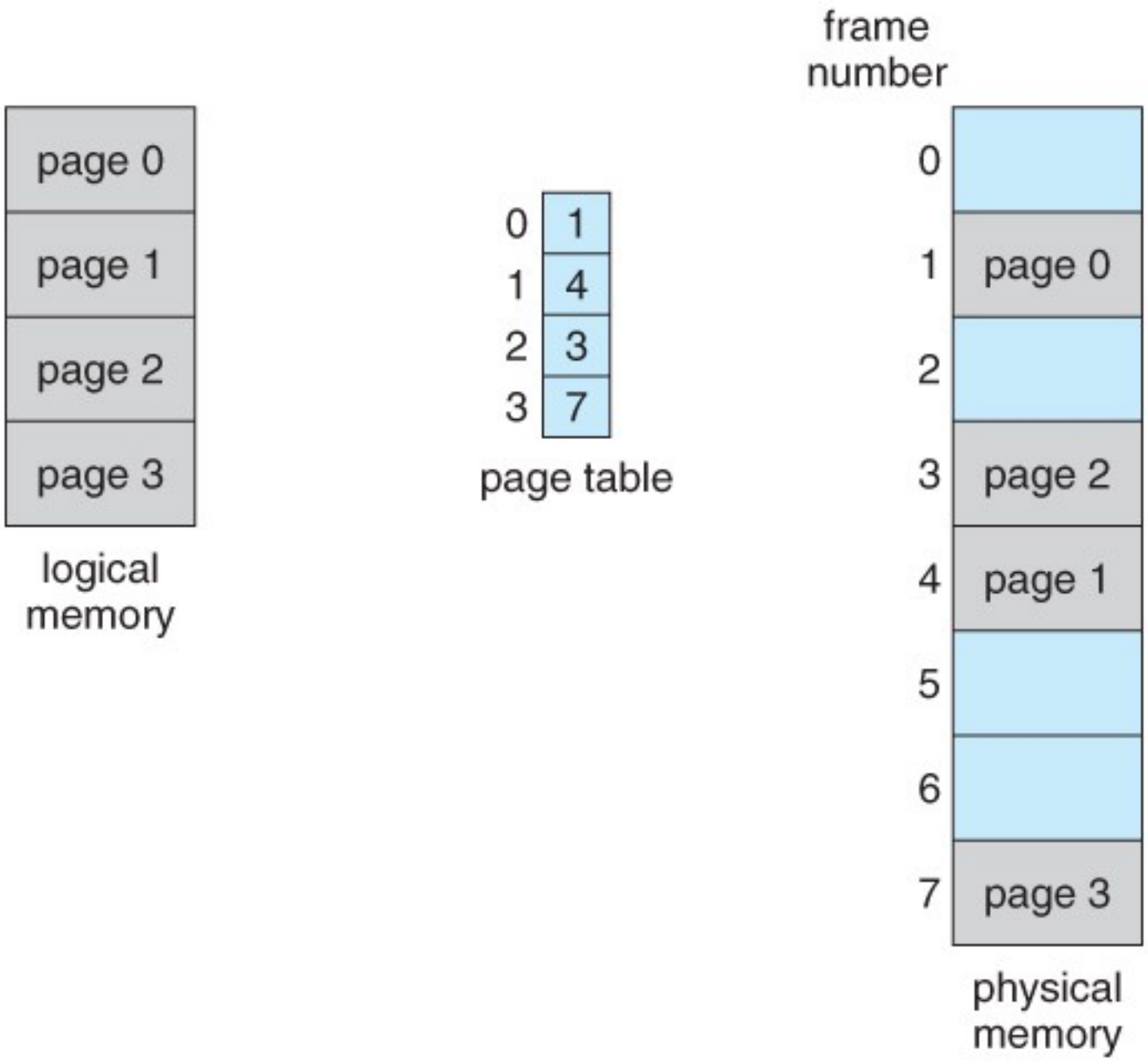

Figure 1: page table

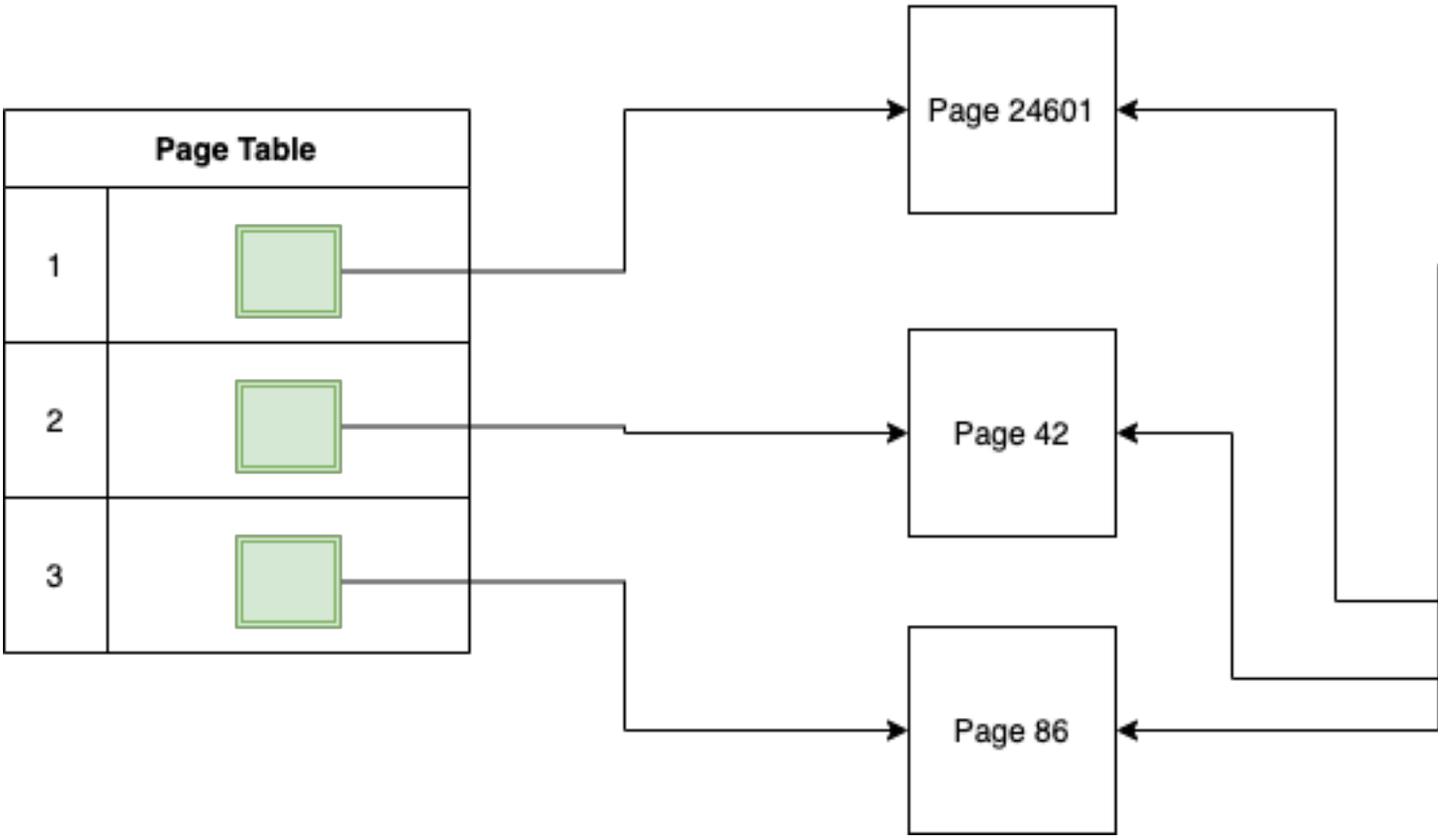

Figure 2: pager

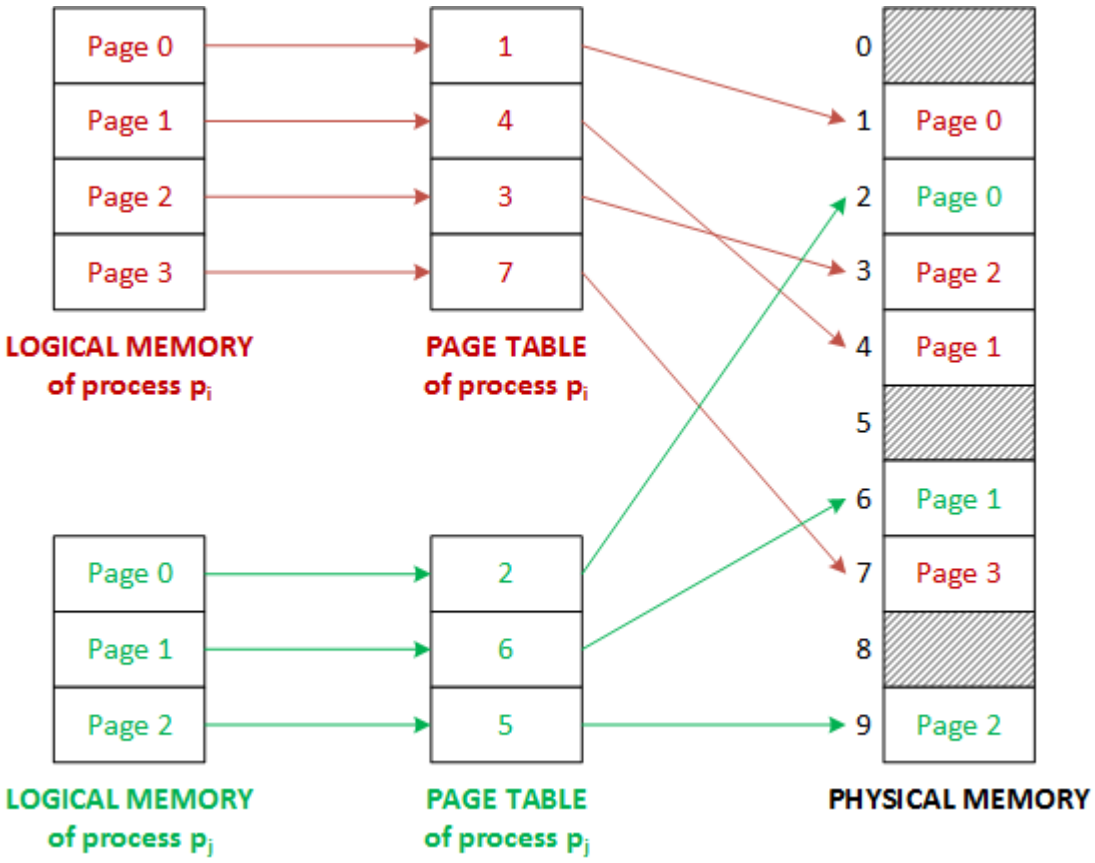

Figure 3: page tables

new page!

It's possible to have multiple pagers operating on the same allocated disk space. In the following example, each process thinks it has its own memory space when accessing the pages of a file. However, under the hood, the same amount of physical memory is still being used, but it is done in a way that is opaque to the user.

1.2.5 Dirty Pages

While data persistence is good, reading from and writing to the disk are expensive operations. In order to minimize the amount of writes, we avoid writing data to the disk unless something has changed. Concretely, if a user only performs SELECTs, but does not INSERT, DELETE, or UPDATE entries on a particular page, then there is no need to flush the page to disk, since the page hasn't changed.

*A page is considered **dirty** when it has been modified, and all dirty pages must be written out to disk before being evicted from the cache.* This guarantees that we persist all changes we make to the database.

---

2 Assignment Spec

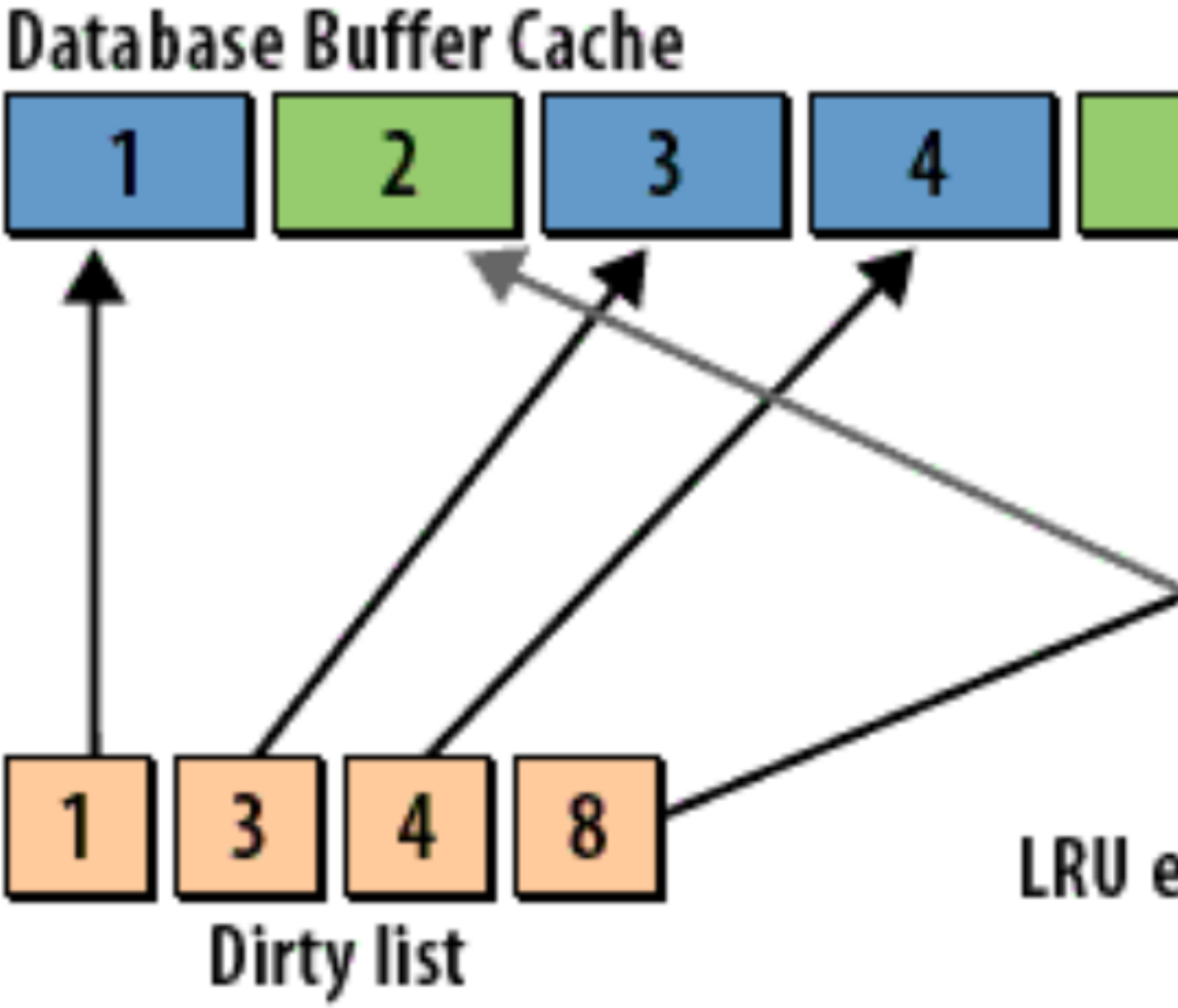

Figure 4: lists

2.1 Overview

In this assignment, you will implement a pager that can read pages in from disk, write them out to disk, and do so in a multithreaded environment.

The following are the files you will need to alter:

- `./pkg/storage/pager.go`

And the following are the files you will need to read:

- `./pkg/storage/page.go`

2.2 Implementation-specific Details

- Note that our `Page` struct is actually a logical frame! A particular `Page` instance will hold many different pages over its lifetime.
- We use three lists to keep track of pages: `freeList`, `unpinnedList`, and `pinnedList` - a page must be in exactly one of these pages.
- The `freeList` is a queue of pre-allocated pages. We have already populated the list for you in `NewPager`.
- The `unpinnedList` keeps track of pages in memory that have yet to be evicted, but are not currently in use. We should put newly unpinned pages at the tail of the `unpinnedList`. When we run out of pages from the free list, we should pop from the head of the `unpinnedList` to get a new page. This way, we implement LRU cache eviction.
- The `pinnedList` is a list of all blocks currently being used by the database. We should put newly pinned pages at the tail of the `pinnedList`.
- Remember to grab the `ptMtx` whenever mutating any of the lists/maps of the page table! Otherwise, our code won't be safe in a multithreaded setting.
- If your tests are timing out, it is most likely due to a mutex deadlock. Be sure that for every lock there is a corresponding unlock! The `defer` keyword may be of use here.
- Do **NOT** use the `Lock` or `Unlock` methods in the `page.go` file yet! These will be used in a later assignment; you are not expected to/supposed to do any page locking here - only page table locking!

2.3 Pager

Implement the following functions:

```
func (pager *Pager) newPage(pagenum int64) (*Page, error)
```

`newPage` should check if there are pages in the free list, and return one from there if it exists. Otherwise, it should evict a page from the head of the unpinned list, flush that page's content out to memory, and return the clean page. If there are no pages in either list, then throw an error. Do not forget to set the returned page's variables correctly!

```
func (pager *Pager) GetPage(pagenum int64) (page *Page, err error)
```

`GetPage` should check the page table for the requested page. If it is an invalid page number (*e.g.* too large or too small), then we should throw an error.

If the page is in the page table, then it is either in the unpinned or pinned list. If it is in the unpinned list, then we *note that it is actively being used* and return it. Otherwise, we have already noted it's actively being used.

If the page is not in the page table, then we will need to grab a new page from disk. In the case that we are requesting a page that doesn't exist yet (*e.g.* the page number has never been seen before), then we don't need to read anything from disk. Otherwise, we read the data from disk into our page (hint: use `ReadPageFromDisk`). Finally, ensure that the page table is up to date and return the page. Remember to `Get` the page before returning it, and remember to check for errors and handle the lists properly in each error case!

```
func (pager *Pager) FlushPage(page *Page)
```

`FlushPage` should write the page's data out *to the correct offset* in the file. Hint: use `file.WriteAt`. We should only do this if the file exists and the page is dirty. It doesn't matter if the page is pinned - while technically this could result in a torn write, we will ensure that this won't happen at a higher level later on.

```
func (pager *Pager) FlushAllPages()
```

`FlushAllPages` should flush all pages. Note that all of the pages that need flushing are either in the pinned list or the unpinned list - iterating through them should achieve the desired result.

2.4 Testing

Run our unit tests on Gradescope.

2.5 Getting started

To get started, download the following: pager. From now on, you'll be getting stencils in these `.zip` files; typically, you'll just need to drop the packages into the `pkg/` folder. In this case, you should put the `pager` folder into the `pkg` folder. We've provided an updated `main.go` file here. When finished, your filestructure should look like this:

```
./
    - cmd/
        - bumble/ (new `main.go`)
    - pkg/
        - config/
        - list/
        - pager/
        - repl/
```

Feedback

As this is a new course, we appreciate any feedback you have for us! If you enjoyed this assignment, hated this assignment, or have other thoughts to share, please do so here!