8.3 Buffers and Buffer Pools

Given the specifications of the disk drive from Example 8.1, we find that it takes about $9.5 + 11.1/2 + 3 \times 11.1 = 48.4$ ms to read one track of data on average when the interleaving factor is 3. Even if interleaving were unnecessary, it would still require $9.5 + 11.1/2 + 11.1 = 26.2$ ms. It takes about $9.5 + 11.1/2 + (1/256) \times 11.1 = 15.1$ ms on average to read a single sector of data. This is a good savings (slightly over one third the time for an interleaving factor of 3), but less than 1% of the data on the track are read. If we want to read only a single byte, it would save us less than 0.05 ms over the time required to read an entire sector. This is an insignificant savings in time when we consider the cost of seeking and rotational delay. For this reason, nearly all disk drives automatically read or write an entire sector’s worth of information whenever the disk is accessed, even when only one byte of information is requested.

Once a sector is read, its information is stored in main memory. This is known as buffering or caching the information. If the next disk request is to that same sector, then it is not necessary to read from disk again; the information is already stored in main memory. Buffering is an example of one method for minimizing disk accesses stated at the beginning of the chapter: Bring off additional information from disk to satisfy future requests. If information from files were accessed at random, then the chance that two consecutive disk requests are to the same sector would be low. However, in practice most disk requests are close to the location (in the logical file at least) of the previous request. This means that the probability of the next request “hitting the cache” is much higher than chance would indicate.

This principle explains one reason why average access times for new disk drives are lower than in the past. Not only is the hardware faster, but information is also now stored using better algorithms and larger caches that minimize the number of times information needs to be fetched from disk. This same concept is also used to store parts of programs in faster memory within the CPU, the so-called CPU cache that is prevalent in modern microprocessors.

Sector-level buffering is normally provided by the operating system and is often built directly into the disk drive controller hardware. Most operating systems maintain at least two buffers, one for input and one for output. Consider what would happen if there were only one buffer during a byte-by-byte copy operation. The sector containing the first byte would be read into the I/O buffer. The output operation would need to destroy the contents of the single I/O buffer to write this byte. Then the buffer would need to be filled again from disk for the second byte, only to be destroyed during output. The simple solution to this problem is to keep one buffer for input, and a second for output.
Most disk drive controllers operate independently from the CPU once an I/O request is received. This is useful since the CPU can typically execute millions of instructions during the time required for a single I/O operation. A technique that takes maximum advantage of this microparallelism is double buffering. Imagine that a file is being processed sequentially. While the first sector is being read, the CPU cannot process that information and so must wait or find something else to do in the meantime. Once the first sector is read, the CPU can start processing while the disk drive (in parallel) begins reading the second sector. If the time required for the CPU to process a sector is approximately the same as the time required by the disk controller to read a sector, it may be possible to keep the CPU continuously fed with data from the file. The same concept can also be applied to output, writing one sector to disk while the CPU is writing to a second output buffer in memory. Thus, in computers that support double buffering, it pays to have at least two input buffers and two output buffers available.

Caching information in memory is such a good idea that it is usually extended to multiple buffers. The operating system or an application program may store many buffers of information taken from some backing storage such as a disk file. This process of using buffers as an intermediary between a user and a disk file is called buffering the file. The information stored in a buffer is often called a page, and the collection of buffers is called a buffer pool. The goal of the buffer pool is to increase the amount of information stored in memory in hopes of increasing the likelihood that new information requests can be satisfied from the buffer pool rather than requiring new information to be read from disk.

As long as there is an unused buffer available in the buffer pool, new information can be read in from disk on demand. When an application continues to read new information from disk, eventually all of the buffers in the buffer pool will become full. Once this happens, some decision must be made about what information in the buffer pool will be sacrificed to make room for newly requested information.

When replacing information contained in the buffer pool, the goal is to select a buffer that has “unnecessary” information, that is, that information least likely to be requested again. Since the buffer pool cannot know for certain what the pattern of future requests will look like, a decision based on some heuristic, or best guess, must be used. There are several approaches to making this decision.

One heuristic is “first-in, first-out” (FIFO). This scheme simply orders the buffers in a queue. The buffer at the front of the queue is used next to store new information and then placed at the end of the queue. In this way, the buffer to be replaced is the one that has held its information the longest, in hopes that this information is no longer needed. This is a reasonable assumption when processing
moves along the file at some steady pace in roughly sequential order. However, many programs work with certain key pieces of information over and over again, and the importance of information has little to do with how long ago the information was first accessed. Typically it is more important to know how many times the information has been accessed, or how recently the information was last accessed.

Another approach is called “least frequently used” (LFU). LFU tracks the number of accesses to each buffer in the buffer pool. When a buffer must be reused, the buffer that has been accessed the fewest number of times is considered to contain the “least important” information, and so it is used next. LFU, while it seems intuitively reasonable, has many drawbacks. First, it is necessary to store and update access counts for each buffer. Second, what was referenced many times in the past may now be irrelevant. Thus, some time mechanism where counts “expire” is often desirable. This also avoids the problem of buffers that slowly build up big counts because they get used just often enough to avoid being replaced (unless counts are maintained for all sectors ever read, not just the sectors currently in the buffer pool).

The third approach is called “least recently used” (LRU). LRU simply keeps the buffers in a linked list. Whenever information in a buffer is accessed, this buffer is brought to the front of the list. When new information must be read, the buffer at the back of the list (the one least recently used) is taken and its “old” information is either discarded or written to disk, as appropriate. This is an easily implemented approximation to LFU and is the method of choice for managing buffer pools unless special knowledge about information access patterns for an application suggests a special-purpose buffer management scheme.

Many operating systems support virtual memory. Virtual memory is a technique that allows the programmer to pretend that there is more main memory than actually exists. This is done by means of a buffer pool reading blocks from disk. The disk stores the complete contents of the virtual memory; blocks are read into main memory as demanded by memory accesses. Naturally, programs using virtual memory techniques are slower than programs whose data are stored completely in main memory. The advantage is reduced programmer effort since a good virtual memory system provides the appearance of larger main memory without modifying the program. Figure 8.6 illustrates the concept of virtual memory.

When implementing buffer pools, there are two basic approaches that can be taken regarding the transfer of information between the user of the buffer pool and the buffer pool class itself. The first approach is to pass “messages” between the two. This approach is illustrated by the following abstract class:
Figure 8.6 An illustration of virtual memory. The complete collection of information resides on disk (physical memory). Those sectors recently accessed are held in main memory (virtual memory). In this example, copies of Sectors 1, 7, 5, 3, and 8 from physical memory are currently stored in the virtual memory. If a memory access to Sector 9 is received, one of the sectors currently in main memory must be replaced.

This simple class provides an interface with two functions, `insert` and `getbytes`. The information is passed between the buffer pool user and the buffer pool through the `space` parameter. This is storage space, at least `sz` bytes long, which the buffer pool can take information from (the `insert` function) or put information into (the `getbytes` function). Parameter `pos` indicates where in the buffer pool’s storage space the information will be placed.

The alternative approach is to have the buffer pool provide to the user a direct pointer to a buffer that contains the necessary information. Such an interface might look as follows:
class BufferPool {
public:
    // Return pointer to the requested block
    virtual void* getblock(int block) = 0;
    // Set the dirty bit for buffer buff
    virtual void dirtyblock(int block) = 0;
    // Tell the size of a buffer
    virtual int blocksize() = 0;
};

In this approach, the user of the buffer pool is made aware that the storage space is divided into blocks of a given size, where each block is the size of a buffer. The user requests specific blocks from the buffer pool, with a pointer to the buffer holding the requested block being returned to the user. The user may then read from or write to this space. If the user writes to the space, the buffer pool must be informed of this fact. The reason is that, when a given block is to be removed from the buffer pool, the contents of that block must be written to the backing storage if it has been modified. If the block has not been modified, then it is unnecessary to write it out.

A further problem with the second approach is the risk of stale pointers. When the buffer pool user is given a pointer to some buffer space at time $T_1$, that pointer does indeed refer to the desired data at that time. If further requests are made to the buffer pool, it is possible that the data in any given buffer will be removed and replaced with new data. If the buffer pool user at a later time $T_2$ then refers to the data referred to by the pointer given at time $T_1$, it is possible that the data are no longer valid because the buffer contents have been replaced in the meantime. Thus the pointer into the buffer pool’s memory has become “stale.” To guarantee that a pointer is not stale, it should not be used if intervening requests to the buffer pool have taken place.

Clearly, the second approach places many obligations on the user of the buffer pool. These obligations include knowing the size of a block, not corrupting the buffer pool’s storage space, informing the buffer pool when a block has been modified, and avoiding use of stale pointers. So many obligations make this approach prone to error. The advantage is that there is no need to do an extra copy step when getting information from the user to the buffer. If the size of the records stored is small, this is not an important consideration. If the size of the records is large (especially if the record size and the buffer size are the same, as typically is the case when implementing B-trees, see Section 10.5), then this efficiency issue may become important. Note however that the in-memory copy time will always be far less than the time required to write the contents of a buffer to disk. For applications where disk I/O is the bottleneck for the program, even the time to copy lots of information between the buffer pool user and the buffer may be inconsequential.
You should note that the implementations for class BufferPool above are not templates. Instead, the space parameter and the buffer pointer are declared to be void*. When a class is a template, that means that the record type is arbitrary, but that the class knows what the record type is. In contrast, using a void* pointer for the space means that not only is the record type arbitrary, but also the buffer pool does not even know what the user’s record type is. In fact, a given buffer pool might have many users who store many types of records.

Another thing to note about the buffer pool is that the user decides where a given record will be stored but has no control over the precise mechanism by which data are transferred to the backing storage. This is in contrast to the memory manager described in Section 12.4 in which the user passes a record to the manager and has no control at all over where the record is stored.

8.4 The Programmer’s View of Files

As stated earlier, the C++ programmer’s logical view of a random access file is a single stream of bytes. Interaction with a file can be viewed as a communications channel for issuing one of three instructions: read bytes from the current position in the file, write bytes to the current position in the file, and move the current position within the file. You do not normally see how the bytes are stored in sectors, clusters, and so forth. The mapping from logical to physical addresses is done by the file system, and sector-level buffering is done automatically by the operating system.

When processing records in a disk file, the order of access can have a great effect on I/O time. A random access procedure processes records in an order independent of their logical order within the file. Sequential access processes records in order of their logical appearance within the file. Sequential processing requires less seek time if the physical layout of the disk file matches its logical layout, as would be expected if the file were created on a disk with a high percentage of free space.

C++ provides several mechanisms for manipulating binary files. One of the most commonly used is the fstream class. Following are the primary fstream class member functions for manipulating information in random access disk files. These functions constitute an ADT for files.

- open: Open a file for processing.
- read: Read some bytes from the current position in the file. The current position moves forward as the bytes are read.
write: Write some bytes at the current position in the file (overwriting the
bytes already at that position). The current position moves forward as the
bytes are written.

- seekg and seekp: Move the current position in the file. This allows bytes
at arbitrary places within the file to be read or written. There are actually two
“current” positions: one for reading and one for writing. Function seekg
changes the “get” or read position, while function seekp changes the “put”
or write position.

- close: Close a file at the end of processing.

8.5 External Sorting

We now consider the problem of sorting collections of records too large to fit in
main memory. Since the records must reside in peripheral or external memory, such
sorting methods are called external sorts in contrast to the internal sorts discussed
in Chapter 7. Sorting large collections of records is central to many applications,
such as processing payrolls and other large business databases. As a consequence,
many external sorting algorithms have been devised. Years ago, sorting algorithm
designers sought to optimize the use of specific hardware configurations, such as
multiple tape or disk drives. Most computing today is done on personal computers
and low-end workstations with relatively powerful CPUs, but only one or at most
two disk drives. The techniques presented here are geared toward optimized pro-
cessing on a single disk drive. This approach allows us to cover the most important
issues in external sorting while skipping many less important machine-dependent
details. Readers who have a need to implement efficient external sorting algorithms
that take advantage of more sophisticated hardware configurations should consult
the references in Section 8.9.

When a collection of records is too large to fit in main memory, the only prac-
tical way to sort it is to read some records from disk, do some rearranging, then
write them back to disk. This process is repeated until the file is sorted, with each
record read perhaps many times. Armed with the basic knowledge about disk drives
presented in Section 8.2, it should come as no surprise that the primary goal of an
external sorting algorithm is to minimize the amount of information that must be
read from or written to disk. A certain amount of additional CPU processing can
profitably be traded for reduced disk access.

Before discussing external sorting techniques, consider again the basic model
for accessing information from disk. The file to be sorted is viewed by the pro-
gramer as a sequential series of fixed-size blocks. Assume (for simplicity) that