Virtual Memory

This presents memory-mapped files using Java.

9.1 Memory-Mapped Files in Java

Next, we present the facilities in the Java API for memory-mapping files. To memory-map a file requires first opening the file and then obtaining the `FileChannel` for the opened file. Once the `FileChannel` is obtained, we invoke the `map()` method of this channel, which maps the file into memory. The `map()` method returns a `MappedByteBuffer`, which is a buffer of bytes that is mapped in memory. This is shown in Figure 9.1.

The API for the `map()` method is as follows:

```
map(mode, position, size)
```

The `mode` refers to how the mapping occurs. Figure 9.1 maps the file as `READ_ONLY`. Files can also be mapped as `READ_WRITE` and `PRIVATE`. If the file is mapped as `PRIVATE`, memory mapping takes place via the copy-on-write technique described in Section 9.3. Then any changes in the mapped file result only in changes to the object instance of the `MappedByteBuffer` performing the mapping.

The `position` refers to the byte position where the mapping is to begin, and the `size` indicates how many bytes are to be mapped from the starting position. Figure 9.1 maps the entire file—that is, from position 0 to the size of the `FileChannel`, `in.size()`. It is also possible to map only a portion of a file or to obtain several different mappings of the same file.

In Figure 9.1, we assume that the page size mapping the file is 4,096 bytes. (Many operating systems provide a system call to determine the page size; however, this is not a feature of the Java API.) We then determine the number of pages necessary to map the file in memory and access the first byte of every page using the `get()` method of the `MappedByteBuffer` class. This has the effect of demand-paging every page of the file into memory (on systems supporting that memory model). The API also provides the `load()` method of the `MappedByteBuffer` class, which loads the entire file into memory using demand paging.

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MemoryMapReadOnly
{
   // Assume the page size is 4 KB
   public static final int PAGE_SIZE = 4096;

   public static void main(String args[]) throws IOException {
      RandomAccessFile inFile = new RandomAccessFile(args[0],"r");

      FileChannel in = inFile.getChannel();
      MappedByteBuffer mappedBuffer =
       in.map(FileChannel.MapMode.READ_ONLY, 0, in.size());
      long numPages = in.size() / (long)PAGE_SIZE;
      if (in.size() % PAGE_SIZE > 0)
        ++numPages;

      // we will "touch" the first byte of every page
      int position = 0;
      for (long i = 0; i < numPages; i++) {
         byte item = mappedBuffer.get(position);
         position += PAGE_SIZE;
      }

      in.close();
      inFile.close();
    }
}
```

Figure 9.1 Memory-mapping a file in Java.

Many operating systems provide a system call that releases (or unmaps) the mapping of a file. Such an action releases the physical pages that mapped the file in memory. The Java API provides no such features. A mapping exists until the `MappedByteBuffer` object is garbage-collected.

Programming Problems

9.1 Write a program that implements the FIFO and LRU page-replacement algorithms presented in this chapter. First, generate a random page-reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm, and record the number of page faults incurred by each algorithm. Implement the replacement algorithms so that the number of page frames can vary as well. Assume that demand paging is used. Your algorithms will be based on the abstract class depicted in Figure 9.2.

Design and implement two classes—`LRU` and `FIFO`—that extend `ReplacementAlgorithm`. Each of these classes will implement the

```
public abstract class ReplacementAlgorithm
{
   // the number of page faults
   protected int pageFaultCount;

   // the number of physical page frame
   protected int pageFrameCount;

   // pageFrameCount - the number of physical page frames
   public ReplacementAlgorithm(int pageFrameCount) {
      if (pageFrameCount < 0)
         throw new IllegalArgumentException();

      this.pageFrameCount = pageFrameCount;
      pageFaultCount = 0;
   }

   // return - the number of page faults that occurred.
   public int getPageFaultCount() {
      return pageFaultCount;
   }

   // int pageNumber - the page number to be inserted
   public abstract void insert(int pageNumber);
}
```

Figure 9.2 ReplacementAlgorithm abstract class.

`insert()` method, one class using the LRU page-replacement algorithm and the other using the FIFO algorithm.

a. `PageGenerator`—a class that generates page-reference strings with page numbers ranging from 0 to 9. The size of the reference string is passed to the `PageGenerator` constructor. Once a `PageGenerator` object is constructed, the `getReferenceString()` method returns the reference string as an array of integers.

b. `Test`—used to test your FIFO and LRU implementations of the `ReplacementAlgorithm` abstract class. `Test` is invoked as

```
java Test <reference string #> <# of page frames>
```

Once you have implemented the FIFO and LRU algorithms, experiment with a different number of page frames for a given reference string and record the number of page faults. Does one algorithm perform better than the other? For a given reference-string size, what is the optimal number of page frames?