Major Technique: Secondary Storage

Version 11/06/00 20:19 - 9

What can you do when you have run out of primary storage?

- Your *memory requirements* are larger than the available primary storage.
- You cannot reduce the system's *memory requirements* sufficiently.
- You can attach *secondary storage* to the device executing the system.

Sometimes your system's primary memory is just not big enough to fulfil your program's *memory requirements*.

For example, the Word-O-Matic™ word-processor for the Strap-It-On™ needs to be able to let users edit large amounts of text. Word-O-Matic also supports formatting text for display or printing, not to mention spelling checks, grammar checks, voice output, mail merging and the special StoryDone feature to write the endings for short stories. Unfortunately, the Strap-It-On has only 2Mb of RAM. How can the programmers even consider implementing Word-O-Matic when its code alone will occupy most of the memory space?

The are a number of other techniques in this book which can reduce a program's *memory requirements*. COMPRESSION can store the information in a smaller amount of memory. Testing applications under a memory limit will ensure programs fit well into a small memory space. You can reduce the system functionality by deleting features or reducing their quality. In many cases, however, these techniques will not reduce the program's memory requirements sufficiently: data that has to be accessed randomly is difficult to compress; programs have to provide the features and quality expected by the marketplace.

Yet for most applications there is usually some hope. Even in small systems, the amount of memory a program requires to make progress at any given time is usually a small fraction of the total amount of memory used. So the problem is not where to store the code and data needed by the program at any given moment; rather, the problem is where to store the rest of the code and data that may, or may not, be needed by the program in the future.

Therefore: *Use secondary storage as extra memory at runtime.*

Most systems have some form of reasonably fast *secondary storage*. Secondary storage is distinct from RAM, since the processor can't write to each individual memory addresses directly; but it's easy for applications to access secondary storage without user intervention. Most forms of secondary storage support *file systems* such that the data lives in files with text names and directory structures. Typically each file also supports *random access* to its data ("get me the byte at offset 301 from the start of the file").

If you can divide up your program and data into suitable pieces you can load into main memory only those pieces of code and data that you need at any given time, keeping the rest of the program on secondary storage. When the pieces of the program currently in main memory are no longer required you can somehow replace them with more relevant pieces from the secondary store.

There are many different kinds of secondary storage that can be modified and can be accessed randomly: Floppy Disks, Hard Disks, Flash filing systems, Bubble Memory cards, CD-ROM drives, writable CD ROM file systems, and gargantuan file servers accessed over a network. Palm Pilot systems use persistent 'Memory Records' stored in secondary RAM. Other forms of secondary storage provide only sequential or read-only access: tape, CD-ROM and web pages accessed over the Internet.

For example the Strap-It-On comes with a CyberStrap, which includes a 32Mb bubble memory store built into its strap along with interfaces for wrist-mounted disk drives. So the Word-O-Matic developers can rely on plenty of 'disk' to store data. Thus Word-O-Matic consists of several separate executables for APPLICATION SWITCHING; it stores each unused document in a DATA FILE; dictionaries, grammar rules and skeleton story endings exist as RESOURCE FILES; optional features are generally shipped as PACKAGES; and the most complex operations use object PAGING to make it seem that the RAM available is much larger than in reality.

Consequences

Being able to use SECONDARY STORAGE can be like getting a lot of extra memory for free — it greatly reduces your program's *primary memory requirements*.

However: the secondary storage must be managed, and information transferred between primary and secondary storage. This management has a *time performance* cost, and may also cost programmer *effort* and *programmer discipline*, impose *local restrictions* to support *global mechanisms*, require hardware *or operating system support*, and reduce the program's *usability*. Most forms of secondary storage require additional devices, increasing the system's *power consumption*.

❖ ❖ ❖

Implementation

There are a few key issues you must address to use secondary storage effectively:

- What is divided up: code, data, configuration information or some combination?
- Who does the division: the programmer, the system or the user?
- Who invokes the loading and unloading: the programmer, the system or the user?
- When does loading or unloading happen?

Generally, the more the program is subdivided and the finer the subdivision, the less the program depends on main memory, and the more use the program makes of secondary storage. Coarser divisions, perhaps addressing only code or only data, may require more main memory but place less pressure on the secondary storage resources.

Making programmers subdivide the program manually requires more effort than somehow allowing the system, or the user, to subdivide the program; and a finer subdivision will require more effort than a coarser one. As a result very fine subdivisions are generally only possible when the system provides them automatically; but creating an automatic system requires significant effort. Making the user divide up the program or data imposes little cost for programmers, but reduces the *usability* of the system.

There are similar trade-offs in deciding who controls the loading and unloading of the divisions. If the system does it automatically this saves work for everyone except the system-builders; otherwise the costs fall on the user and programmer. Sequential loading and unloading is the simplest to implement (and often the worst for the user). More complex schemes that load and unload code or data on demand can be much more seamless to the user, and can even make the reliance on secondary storage transparent to both users and programmers.

❖ ❖ ❖

Specialised Patterns

The rest of this chapter contains five specialised patterns describing different ways to use secondary storage. Figure 1 shows the patterns and the relationships between them: arrows show close relationships; springs indicate a tension between the patterns.

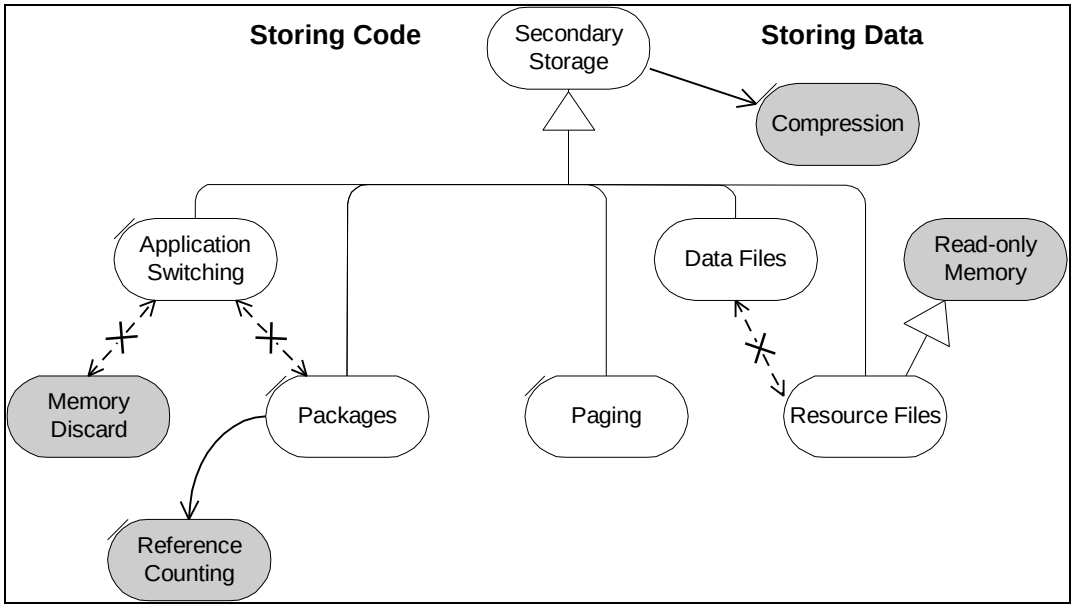

Figure 1: Secondary Storage Patterns

The patterns also form a sequence starting with simple patterns which can be implemented locally, relying only upon programmer discipline for correct implementation, and finishing with more complex patterns which require hardware or operating system support but require much less, if any, *programmer discipline*. Each pattern occupies a different place in the design space defined by the questions above, as follows:

APPLICATION SWITCHING requires the programmer to divide up the program into independent executables, only one of which runs at a time. The order in which the executables run can be determined by the executables themselves, by an external script, or by the user.

DATA FILES uses secondary storage as a location for inactive program data. These files may or may not be visible to the user.

RESOURCE FILES store static read-only data. When the program needs a resource (such as a font, an error message, or a window description), it loads the resource from file into temporary memory; afterwards it releases this memory.

PACKAGES store chunks of the program code. The programmer divides the code into packages, which are loaded and unloaded as required at runtime.

PAGING arbitrarily breaks the program down into very fine units (pages) which are shuffled automatically between primary and secondary storage. Paging can handle code and data, support read-only and shared information between different programs, and is transparent to most programmers and users.

All of these patterns in some sense trade facilities provided in the environment for work done by the programmer. The more complex the environment (compilation tools and runtime system), the less memory management work for the programmer; however a complex run-time environment takes both effort to develop, and has its own memory requirements at run-time. Figure 3 shows where each pattern fits in this scheme.

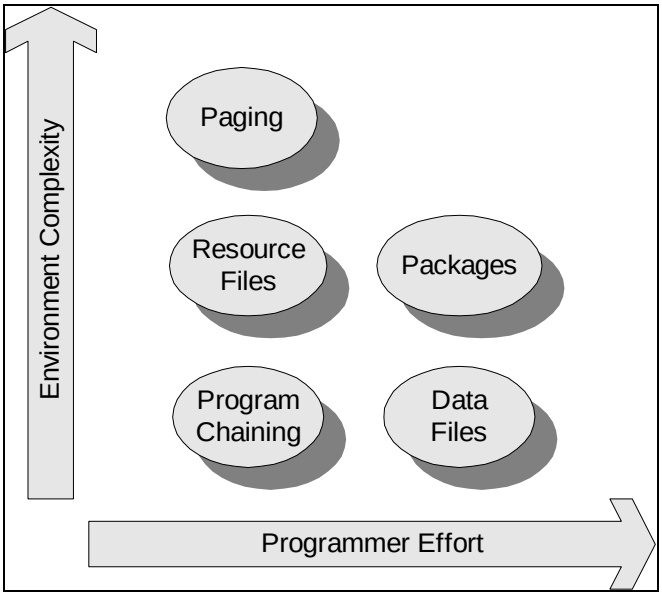

Figure 3: Implementation Effort vs. Environmental Complexity

See Also

READ-ONLY pieces of program or data can be deleted from memory without having to be saved back to secondary storage.

You can use COMPRESSION to reduce the amount of space taken on secondary storage.

Secondary Storage management is one of the primary functions of modern operating systems. More background information and detail on techniques for using Secondary Storage can be found in many operating systems textbooks [Tannenbaum 1992, Leffler, McKusik, Karels and Quarterman 1989, Goodheart and Cox 1994].

Application Switching

How can you reduce the memory requirements of a system that provides many different functions?

Also known as: Phases, Program Chaining, Command Scripts.

- Systems are too big for all the code and data to fit into main memory
- Users often need to do only one task at a time
- A single task requires only its own code and data to execute; other code and data can live on secondary storage.
- It's easier to program only one set of related tasks – one application – at a time.

Some systems are big – too big for all of the executable code and data to fit into main memory at the same time.

For example a Strap-It-On user may do word-processing, run a spreadsheet, read Web pages, do accounts, manage a database, play a game, or use the 'StrapMan' remote control facilities to manage the daily strategy of a large telecommunications network. How can the programmers make all this functionality work in the 2 Mb of RAM they have available – particularly as each of the StrapMan's five different functions requires 1Mb of code and 0.5 Mb of temporary RAM data?

Most systems only need a small subset of their functionality – enough to support one user task – at any given time. Much of the code and data in most systems is unused much of the time, but all the while it occupies valuable main memory space.

The more complex the system and the bigger the development team, the more difficult development becomes. Software developers have always preferred to split their systems architecture into separate components, and to reduce the interdependencies between these components. Components certainly make system development manageable, but they do not reduce main memory requirements.

Therefore: *Split your system into independent executables, and run only one at a time.*

Most operating systems support independent program components in the form of executable files on secondary storage. A running executable is called a process and its code and data occupies main memory. When a process terminates, all the main memory it uses is returned to the system.

Design the system so that behaviour the user will use together or in quick succession will be in the same executable. Provide facilities to start another executable when required, terminating the current one. The new process can reuse all the memory released by the terminated process.

In many operating systems this is the only approach supported; only one process may execute at a time. In MS-DOS the executable must provide functionality to terminate itself before another executable can run; in MacOS and PalmOs there is control functionality shared by all applications to support choosing another application and switching to it. [Chappell 1994, Apple 1985, Palm 2000]. In multi-tasking operating systems this pattern is still frequently used to reduce main memory requirements.

For example, no Strap-It-On user would want to do more than one of those tasks at any one time; it's just not physically possible given the small size of the screen. So each goes in a separate executable (word-processor, spreadsheet, web browser, accounting, database, Doom), and the Strap-It-On provides a control dialog that allows the user to terminate the current application and start another. Each application saves its state on exit and restores it on restart,

so that – apart from the speed of loading – the user has no way of telling the application has terminated and restarted. The StrapMan application, however, wouldn't fit in RAM as a single executable. So the StrapMan's authors split it into six different executables (one for the main program and one for each function), and made the main one 'chain' to each other executable as required.

Consequences

The *memory requirements* for each process are less than the *memory requirements* for the entire system. The operating system reclaims the memory when the process terminates, so this reduces *programmer effort* managing memory and reduces the effects of 'memory leaks'.

Different executables may be in different implementation languages, and be an interpreted or compiled as required. Some executables may also be existing 'legacy' applications, or utilities provided by the operating system. So APPLICATION SWITCHING may significantly reduce the *programmer effort* to produce the system, encouraging *reuse* and making *maintenance* easier. Script-based approaches can be very flexible, as scripts are typically very easy to modify. Also applications tend to be geared to stopping and starting regularly, so errors that terminate applications may not be so problematic to the user, increasing the system's robustness.

In single-process environments, such as PalmOs, each process occupies the same memory space, so the amount of memory required is easy to *predict*, which improves *reliability*, makes *testing easier* and removes the effects of the *global* memory use on each *local* application. You only need to start the first process to get the system running, reducing *start-up times*. It's also easy to know what's happening in a single-process environment, simplifying *real-time* programming.

However: Dividing a large program into a good set of processes can be difficult, so a multi-process application can require significant *programmer effort* to design, and *local complexity* in the implementation.

If you have many executables, the cost of starting each and of transferring data can dominate the system's *run time performance*; this is also a problem if the control flow between different processes is complex — if processes are started and terminated frequently.

In single-process environments the user can use only the functionality in the current executable, so chaining tends to reduce the system's *usability*. If the user has to manage the processes explicitly, that also reduces the program's *usability*.

This pattern does not support background activities, such as TCP/IP protocols, interfacing to a mobile phone, or background downloading of email. Such activities must continue even when the user switches tasks. The code for background tasks must either be omitted (reducing *usability*), live in a separate process (increasing *programmer effort*), or be implemented using interrupt routing (requiring large amounts of specialised *programmer effort)*.

❖ ❖ ❖

Implementation

To implement Application Switching you have to divide up the system into separate components (see the SMALL ARCHITECTURE pattern). Communication between running processes can be difficult, so in general the split must satisfy these rules:

- The control flow between processes is simple
- There is little transient data passed between the processes.
- The division makes some kind of sense to the user.

Figure 3 shows the two main alternatives that you can use to implement process switching:

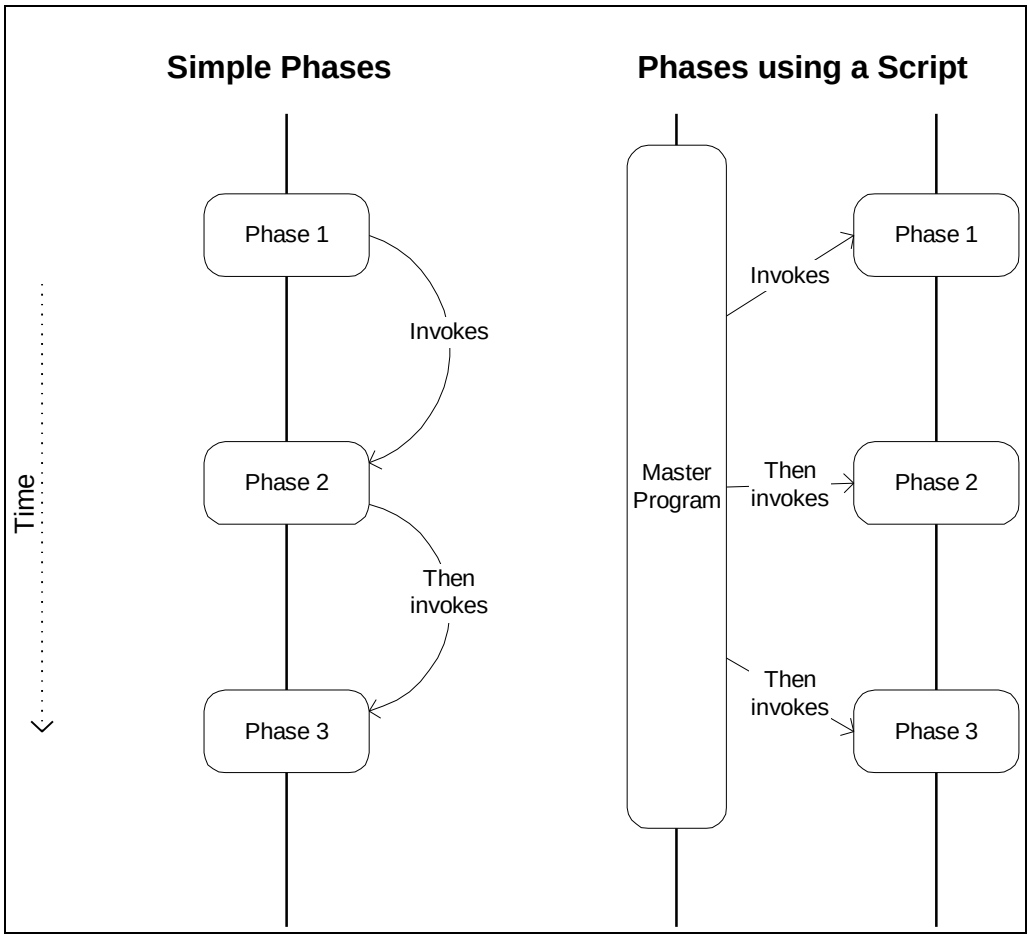

Figure 5: Two different approaches to implementing Phases

1. Program Chaining.

One process can pass control explicitly to the following process. This is called 'program chaining', after the '`CHAIN`' command in some versions of the BASIC programming language [Digital 1975; Steiner 1984]. Program Chaining requires that each executable to know which executable to go to next. This can be programmed explicitly by each application, or as part of an application framework library. Given such an application framework, each executable can use the framework to determine which application to switch to next, and to switch to that application, without requiring much programmer effort. The MacOs (task switcher) and PalmOs application frameworks do this [Apple 1984, Palm 2000].

2. Master Program.

Alternatively, a script or top-level command program can invoke each application in turn. A master program, by contrast, encourages reuse because each executable doesn't need to know much about its context and can be used independently. The UNIX environment pioneered the idea of small interoperable tools designed to work together in this way [Kernighan and Pike 1984]. Even with a master program, the terminating program can help determine which

application to execute next by passing information back to the master program using exit codes, or by producing output or temporary files that are read by the master program.

3. Communicating between Processes.

How can separate components communicate when only one process is active at a time? You can't use main memory, because that is erased when each process terminates. Instead you need to use one or more of the following mechanisms:

- Command line parameters and environment variables passed into the new process.
- Secondary storage files, records or databases written by one process and read by another.
- Environment-specific mechanisms. For example, many varieties of Basic complimented the `CHAIN` command with a `COMMON` keyword that specifies data preserved when a new process overwrites the current one [Steiner 1984].

4. Managing Data.

How do you make it seem to the user that an application never terminates, even when it is split up in separate processes? Many environments only support a small number of processes, maybe just one, but users don't want to have to recreate all their state each time they start up a new application. They want the illusion that the application is always running in the background.

The solution is for the application to save an application's state to SECONDARY STORAGE on exit, and to restore this state when the application's restarted. Many OO libraries and environments support ways of 'streaming' all the important objects – data and state – as a single operation. The approach requires a binary 'file stream', which defines stream functions to read and write primitive types (e.g. int, char, float, string). Each class representing the application's state then defines its own streaming functions.

When you are streaming out object-oriented applications, you need to ensure each object is streamed only once, no matter how many references there may be to it. A good way to deal with this is to have the 'file stream' maintain a table of object identifiers. Each time the stream receives a request to stream out an object it searches this table, and if it finds the object already there, it just saves a reference to the file location of the original instead of saving it again.

The Java libraries support persistence through the Serialization framework [Chan et al 1998]. Any persistent class implements the `Serializable` interface; it needs no other code – the runtime environment can serialize all its data members, following object references as required (and writing each object only once, as above). The classes `ObjectOutputStream` and `ObjectInputStream` provide methods to read and write a structure of objects: `writeObject` and `readObject` respectively. By convention the files created usually have the extension '`.ser`'; some applications ship initial '`.ser`' files with the Java code in the JAR archive.

Examples

Here's a very trivial example from an MS Windows 3.1 system. We cannot use the disk-checking program, scandisk, while MS Windows is running, so we chain it first, then run Windows:

```
@REM AUTOEXEC.BAT Command file to start MS Windows 3.1 from DOS
@REM [Commands to set paths and load device drivers omitted]
C:\WINDOWS\COMMAND\scandisk /autofix /nosummary
win
```

The following Java routine chains to a different process, terminating the current process:

```
    void ChainToCommand(String theCommand) throws IOException {
        Runtime.getRuntime().exec(theCommand);
        Runtime.getRuntime().exit( 0 );
    }
```

Note that if this routine is used to execute another Java application, it will create a new Java virtual machine before terminating the current one, and the two VMs will coexist temporarily, requiring significant amounts of memory.

The Unix `exec` family of functions is more suitable for single process chaining in low memory; each starts a new process in the space of the existing one [Kernighan and Pike 1984]. The following C++ function uses Microsoft C++'s `_execl` variant [Microsoft 1997]. It also uses the Windows environment variable '`COMSPEC`' to locate a command interpreter, since only the command interpreter knows where to locate executables and how to parse the command line correctly.

```
void ChainToCommand( string command )
{
   const char *args[4];
   args[0] = getenv( "comspec" );
   args[1] = "/c";
   args[2] = command.c_str();
   args[3] = 0;
   _execv( args[0], args );
}
```

The function never returns. Note that although all the RAM memory is discarded, `execl` doesn't close file handles, which remain open in the chained process. See your C++ or library documentation for '`execl`' and the related functions.

The following is some EPOC C++ code implementing streaming for a simple class, to save data to files while the application is switched. The class, `TSerialPortConfiguration`, represents configuration settings for a printer port. Most of its data members are either C++ `enum`'s with a small range of values, or one-byte integers (`char` in C++, `TInt8` in EPOC C++); `TOutputHandshake` is a separate class:

```
class TSerialPortConfiguration {
   // Various function declarations omitted…
   TBps iDataRate;
   TDataBits iDataBits;
   TStopBits iStopBits;
   TParity iParity;
   TOutputHandshake iHandshake;
   };
```

The functions `InternalizeL` and `ExternalizeL` read and write the object from a stream. Although the values `iDataRate` are represented internally as 4-byte integers and enums, we know we'll not loose information by storing them as PACKED DATA, in a single byte. The class `TOutputHandshake` provides its own streaming functions, so we use them:

```
EXPORT_C void TSerialPortConfiguration::InternalizeL(RReadStream& aStream)
   {
   iDataRate = (TBps) aStream.ReadInt8L();
   iDataBits = (TDataBits) aStream.ReadInt8L();
   iStopBits = (TStopBits) aStream.ReadInt8L();
   iParity = (TParity) aStream.ReadInt8L();
   iHandshake.InternalizeL(aStream);
   }

EXPORT_C void TSerialPortConfiguration::ExternalizeL(RWriteStream& aStream) const
   {
   aStream.WriteInt8L(iDataRate);
   aStream.WriteInt8L(iDataBits);
   aStream.WriteInt8L(iStopBits);
   aStream.WriteInt8L(iParity);
   iHandshake.ExternalizeL(aStream);
   }
```


❖ ❖ ❖

Known Uses

The PalmOs and early versions of the MacOs environments both support only a single user process at any one time; both provide and framework functions to simulate multi-tasking for the user. MacOs uses persistence while PalmOs uses secondary storage 'memory records' to save application data [Apple 1985, Palm 2000].

The UNIX environment encourages programmers to use processes by supporting *scripts* and making them executable in the same way as binary executables, with any suitable scripting engine [Kernighan and Pike 1984]. In Windows and the DOS environments, the only fully-supported script formats are the fairly simple BAT and CMD formats, although it's trivial to create a simple Windows BAT file to invoke more powerful scripting language such as Tcl [Ousterhout 1994] and Perl [Wall 1996].

The Unix Make utility manages application switching (and the DATA FILES required) to compile a program, generally running any preprocessors and the appropriate compiler process for each input file in turn, then running one or more linker processes to produce a complete executable [Kernighan and Pike 1984].

See Also

PACKAGES provide similar functionality within a single process – by delaying code loading until it's required. However whereas in PROCESS SWITCHING the operating system will discard the memory, code and other resources owned by a task when the task completes, a PACKAGE must explicitly release these resources.

PAGING provides much more flexible handling of both code and data.

The executables can be stored on SECONDARY STORAGE, using COMPRESSION.

The MEMORY DISCARD pattern has a similar dynamic to this pattern but on a much smaller scale. Where APPLICATION SWITCHING recovers all the memory occupied by an process only when it terminates, MEMORY DISCARD allows an application to recover the memory occupied by a group of objects in the middle of its execution.

Data File Pattern

What can you do when your data doesn't fit into main memory?

Also known as: Batch Processing, Filter, Temporary File

- Systems are too big for all the code and data to fit into RAM together
- The code by itself fits into RAM (or can be fitted using other patterns)
- The data doesn't fit into RAM
- Access to the data is sequential.

Sometimes programs themselves are quite small, but need to process a large amount of data — the *memory requirements* mean that the program will fit into main memory, but the data requires too much memory.

For example, the input and output data for the Word-O-Matic Text Formatter can exceed the capacity of the Strap-It-On's main memory when formatting a large book. How should the Word-O-Matic designers implement the program to produce the output PostScript data, let alone to produce all the index files and update all the cross references, when it's physically impossible to get them all into RAM memory?

Dividing the program up into smaller phases (as in APPLICATION SWITCHING) can reduce the memory required by the program itself, but this doesn't help reduce the memory requirements for the input and output. Similarly, COMPRESSION techniques may reduce the amount of secondary storage required to hold the data, but don't reduce the amount of main memory need to process it.

Yet most systems don't need you to keep all data in RAM. Modern operating systems make it simple to read and write from files on Secondary Storage. And the majority of processing tasks do not require simultaneous access to all the data.

Therefore: *Process the data a little at a time and keep the rest on secondary storage.*

Use sequential or random file access to read each item to process; write the processed data sequentially back to one or more files. You can also write temporary items to secondary storage until you're ready to use them. If you are careful, the amount of main memory needed for processing each portion will be much less than the total memory that would be required to process all the data in main memory. You need to be able to store both input and output as files in SECONDARY STORAGE, so the input and output data must be partitioned cleanly.

For example Word-O-Matic stores its chapters as separate text files. The Word-O-Matic Text Formatter (nicknamed the 'Wombat') makes several passes over these files, see Figure XXX. The first pass scans all the chapter files in turn, locating the destinations of cross references and index entries in the file data, and writes all the information it needs to create each cross-reference to a temporary 'cross reference' file. Wombat's second pass then scans the cross-reference file to create an in-memory index to this file, then reads each chapter file, creating a transient version with the cross references and indexes included. It reads the cross-reference data by random access to the index file using the in-memory index. Since the page numbering changes as a result of the updates, Wombat also keeps an in-memory table showing how each reference destination has moved during this update. Finally Wombat's third pass reads each transient chapter file a bit at a time, and writes out the PostScript printout sequentially, making the corrections to the page numbers in the index and references using its in-memory table as it does. Using these techniques Wombat can format an entire book using as little as 50Kb of RAM memory.

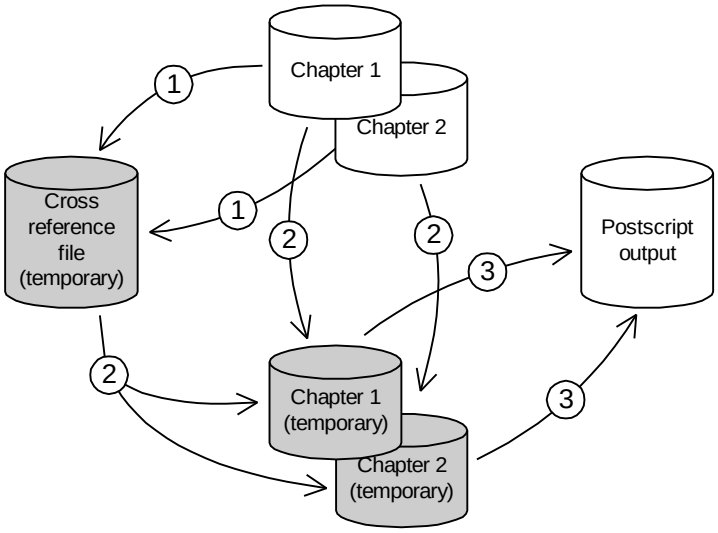

Figure 6: Wombat's Data Files and Phases

Consequences

The *memory requirements* for processing data piecemeal are reduced, since most of the data lives on secondary storage. The system's main memory requirements are also much more *predictable*, because you can allocate a fixed memory to support the processing, rather than a variable amount of memory to store a variable amount of data.

You can examine the input and output of functions in an application using utilities to look at the secondary storage files, which makes *testing* easier. Data Files also make it easy to split an application into different independent components linked only by their data files, reducing the *global* impact of *local* changes, and making *maintenance* easier. Indeed Data Files also make it much easier to implement phases, allowing APPLICATION SWITCHING; for example, Wombat's phase 1 is in a different executable from phases 2 and 3.

However: *Programmer effort* is required to design the program so that the data can be processed independently. Processing data incrementally adds *local complexity* to the implementation, which you could have avoided by processing the data *globally* in one piece. If you need to keep extra context information to process the data, then managing this information can add *global complexity* to the program.

Data chaining can provide slower *run-time performance* than processing all the input in one piece, since reading and writing many small data items is typically less efficient than reading or writing one large item. Repeated access to secondary storage devices can increase the system's *power consumption*, and can even reduce the lifetime of some secondary storage media, such as flash RAM and floppy disks. The limitations of data files – such as imposing ordering rules on the input, or needing the user or client software to manage files – can reduce the system's *usability*.

❖ ❖ ❖

Implementation

The Wombat example above illustrated the four main kinds of operation on data files:

1. Simple Sequential Input (reading each chapter in turn)

2. Simple Sequential Output (writing the final output file)
3. Random Access (reading and writing the cross-reference file)
4. Sequential output to several files (writing the temporary chapter files)

Here are some issues to consider when using data files:

1. Incremental Processing. One simple and common way to manipulate data files is to read an entire file sequentially from input, and/or to write a second file sequentially to the output (see figure XX). Incremental processing requires extra *programmer effort* to implement, because the program must be tailored specially to process its input file incrementally. Because the program processes one large file in small increments, the program is typically responsible for selecting the increments to process (although this can be left to the user by requiring them to indicate increment boundaries in the data file, or provided a collection of smaller data files).

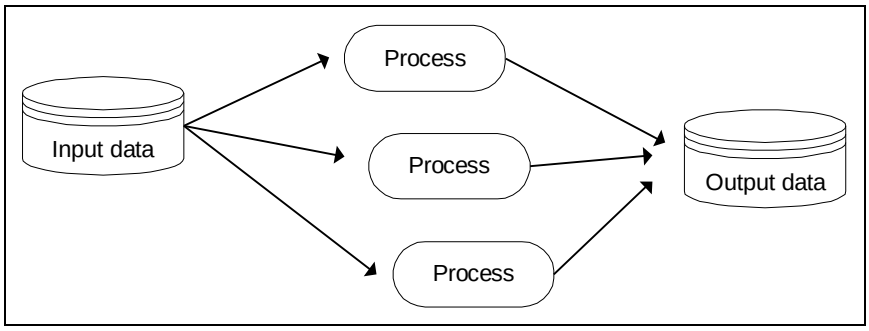

Figure 7: Incremental Processing

Because the whole input file is processed in a single operating systems process, incremental data chaining makes it easier to maintain global contextual information between each processing stage, and easier to produce the final output — the final output is just written incrementally from the program. Unfortunately, precisely because it works in one single long-running process, it can be more difficult to keep the *memory requirements* down to a minimum.

2. Subfile Processing. Rather than processing a single file sequentially, you can divide data up into a number of smaller subfiles. Write a program which processes one subfile, producing a separate output file. Run this program multiple times (typically sequentially) to process each subfile, and then combine the subfiles to produce the required output (see Figure YYY).

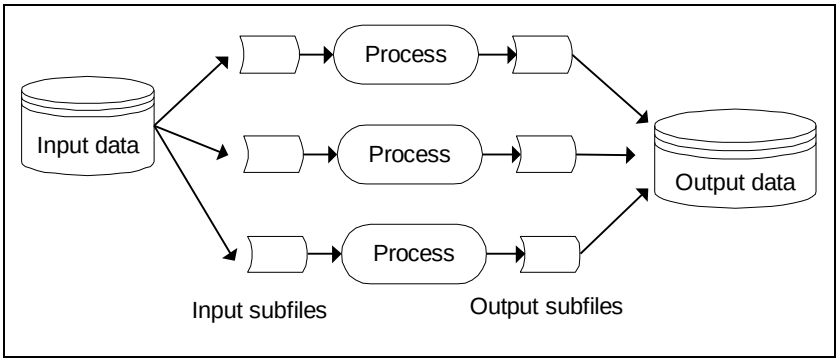

Figure 8: Subfile Processing

Subfile processing has several advantages, provided it's easy to divide up the data. Subfile processing tends to require less memory, since only a subset of the data is processed at a time;

and it is more robust to corruption and errors in the data files, since each such problem only affects one file. Unfortunately, splitting the files requires effort either on the part of the program or on the part of the user: co-ordinating the processing and combining the subfiles requires programmer effort. See the pattern APPLICATION SWITCHING for a discussion of techniques for communication between such processes.

Many compilers use subfile processing: they compile each code file separately, and only combine the resulting temporary object files in a separate link phase. Because of its enormous potential for reducing memory use, subfile processing was ubiquitous in old-time batch tape processing [Knuth 1998].

3. Random Access. Rather than reading and writing files sequentially (whether incrementally or using subfiles) you can access a single file randomly, selecting information and reading and writing it in any order. Random access generally requires more *programmer effort* than incremental or subfile processing, but is much more flexible: you don't have to determine the order items can be processed (and possibly divide them into subfiles) in advance.

To use random access, each process needs to be able to locate individual data items within the files on secondary storage. Generally, you will need an index, a list of offsets from the start of the file for each item of data required. Because the index will be used for most accesses to the file, it needs to be stored in main memory, or easily accessible from main memory. Effective indexing of files is a major science in its own right, but for simple applications there are two straightforward options:

- The file may contain its own index, perhaps at the start of the file, which is read into RAM by the process. RESOURCE FILES often use this approach.
- The application may scan the file on start up, creating its own index. The Wombat text processor did this with its cross-reference file.

More complicated systems may use indexes in different files from the data, or even have indexes to the index files themselves. File processing is covered in texts such as Folk, Zoellick and Riccardi [1988], Date [1999] and Elmasri and Navathe [2000].

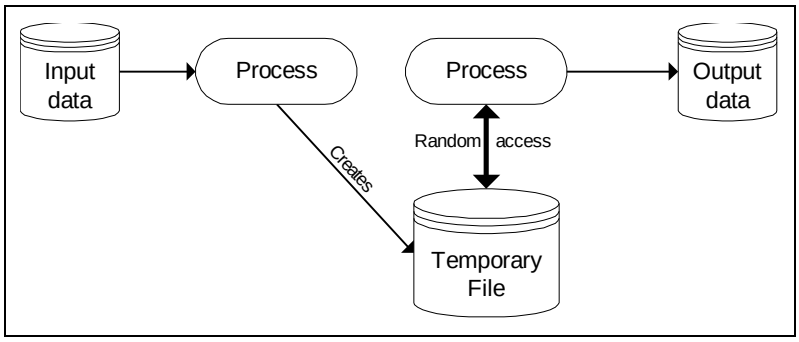

Figure 9: Random Access

Examples

1. Simple Subfile Processing

File compilation provides a typical example of subfile processing. The user splits each large program into a number of files, and the compiler processes each file individually. Then the linker '`ld`' combines the various '`.o`' output files into an executable program, `testprog`.


```
cc main.c
cc datalib.c
cc transput.c
ld –o testprog main.o datalib.o transput.o
```

2. Incremental Processing

The following Java code reverses the characters in each line in a file. It reads each line into a buffer, reverses the characters in the buffer, and then writes the buffer out into a second file. We call the `reverse` method with a `BufferedReader` and `BufferedWriter` to provide more efficient access to the standard input and output than direct access to the disk read and write functions, at a cost of some memory:

```
reverse(new BufferedReader(new InputStreamReader(System.in)),
        new BufferedWriter(new OutputStreamWriter(System.out)));
```

The `reverse` method does the work, using two buffers, a `String` and a `StringBuffer`, because `Strings` in Java cannot be modified.

```
public void reverse(BufferedReader reader, BufferedWriter writer)
   throws IOException {
   String line;
   StringBuffer lineBuffer;

   while ((line = reader.readLine())!=null) {
       lineBuffer = new StringBuffer(line);
       lineBuffer.reverse();
       writer.write(lineBuffer.toString());
       writer.newLine();
   }
   writer.close();
}
```

The important point about this example is that it requires only enough memory to hold the input and output buffers, and a single line of text to reverse, rather than the entire file, and so can handle files of any length without running out of memory.

3. Processing with Multiple Subfiles

Consider reversing all the bytes in a file rather than just the bytes in each line. The simple incremental technique above won't work, because it relies on the fact that processing one line does not affect any other lines. Reversing all the characters in a file involves the whole file, not just each individual line.

We can reverse a file without needing to store it all in memory by using subfiles on secondary storage. We first divide (`scatter`) the large file into a number of smaller subfiles, where each subfile is small enough to fit into memory, and then we can reverse each subfile separately. Finally, we can read (`gather`) each subfile in reverse order, and assemble a new completely reversed file.

```
public void run() throws IOException {
   scatter(new BufferedReader(new InputStreamReader(System.in)));
   gather(new BufferedWriter(new OutputStreamWriter(System.out)));
}
```

To scatter the file into subfiles, we read `SubfileSize` bytes from the input reader into a buffer, reverse the buffer, and then write it out into a new subfile


```
protected void scatter(BufferedReader reader) throws IOException {
   int bytesRead;
   while ((bytesRead = reader.read(buffer, 0, SubfileSize)) > 0) {
       StringBuffer stringBuffer = new StringBuffer(bytesRead);
       stringBuffer.append(buffer, 0, bytesRead);
       stringBuffer.reverse();
       BufferedWriter writer =
        new BufferedWriter(new FileWriter(subfileName(nSubfiles)));
       writer.write(stringBuffer.toString());
       writer.close();
       nSubfiles++;
   }
}
```

We can reuse the buffer each time we reverse a file (an example of FIXED ALLOCATION), but we have to generate a new name for each subfile. We also need to count the number of subfiles we have written, so that we can gather then all together again.

```
protected char buffer[] = new char[SubfileSize];

protected String subfileName(int n) {
   return "subxx" + n;
}

protected int nSubfiles = 0;
```

Finally, we need to gather all the subfiles together. Since the subfiles are already reversed, we just need to open each one starting with the last, read its contents, and write them to an output file.

```
protected void gather(BufferedWriter writer) throws IOException {
   for (nSubfiles--; nSubfiles >= 0; nSubfiles--) {
       File subFile = new File(subfileName(nSubfiles));
       BufferedReader reader =
           new BufferedReader(new FileReader(subFile));
       int bytesRead = reader.read(buffer, 0, SubfileSize);
       writer.write(buffer, 0, bytesRead);
       reader.close();
       subFile.delete();
      }
   writer.close();
}
```

❖ ❖ ❖

Known Uses

Most programming languages compile using subfile processing. C, C++, FORTRAN and COBOL programs are all typically compiled one file at a time, and the output object files are then combined with a single link phase after all the compilation phases. C and C++ also force the programmer to manage the 'shared data' for the compilation process in the form of header files [Kernighan and Ritchie 1988]. Java takes the same approach for compiling each separate class file; instead of a link phase Java class files are typically combined into a 'JAR' archive file using COMPRESSION [Chen et al 1998].

The UNIX environment encourages programmers to use data files by providing many simple 'filter' executables: `wc, tee, grep, sed, awk, troff`, for example [Kernighan and Pike 1984]. Programmers can combine these using 'pipes'; the operating system arranges that each filter need only handle a small amount of data at a time.

Most popular applications use data files, and make the names of the current files explicit to the user. Microsoft's MFC framework enshrines this application design in its Document-View architecture [Prosise 1999], supporting multiple documents, where each document normally corresponds to a single data file. EPOC's AppArc architecture [Symbian 1999] supports only one document at a time; depending on the look and feel of the particularly environment this file name may not be visible to the user.

Some word processors and formatters support subfiles – for example Microsoft Word, TeX, and FrameMaker. [Microsoft Word 1997, Lamport 1986, Adobe 1997]. The user can create a *master document* that refers to a series of subdocuments. These subdocuments are edited individually, but when the document is printed each subdocument is loaded into memory and printed in turn. The application need keep only a small amount of global state in memory across subdocuments.

EPOC supports a kind of sub-file within each file, called a stream; each stream is identified using an integer ID and accessed using a simple persistence mechanism. This makes it easy to create many output subfiles and to access each one separately, and many EPOC applications use this feature. Components that use large objects generally persist each one in a separate stream; then they can defer loading each object in until it's actually required – the template class `TSwizzle` provides a MULTIPLE REPRESENTATION to make this invisible to client code [Symbian 1999]. EPOC's relational database creates a new stream for every 12 or so database rows, and for every binary object stored. This makes it easy for the DBMS server to change entries in a database – by writing a new stream to replace an existing one and updating the database's internal index to all the streams [Thoelke 1999].

Printer drivers (especially those embedded in bitmap-based printers) often use 'Banding', where the driver renders and prints only a part of the page at a time. Banding reduces the size of the output bitmap it must store, but also reduces the printing speed, as each page must be rendered several times, once for each band.

See Also

RESOURCE FILES is an alternative for read-only data. PAGING is much simpler for the programmer, though much more complex to implement. DATA FILES make it easier to implement APPLICATION SWITCHING. Each subfile can be stored on SECONDARY STORAGE, using COMPRESSION.

You can use either or both of FIXED ALLOCATION and MEMORY DISCARD to process each item read from a DATA FILE.

PIPES AND FILTERS [Shaw and Garland 1996] describes a software architecture style based around filters.

Rather than a simple Data File, you may need a full-scale database [Connolly and Begg 1999; Date 1999; Elmasri and Navathe 2000]. Wolfgang Keller and Jens Coldewey [1998] provide a set of patterns to store objects from OO programs into relational databases.

Resource Files Pattern

*How can you manage lots of configuration data**?***

- Much program data is *read-only* configuration information and is not modified by the program.
- The configuration data typically changes more frequently than program code.
- Data can be referenced from different phases of the program.
- You only need a few data items at any time.
- File systems support *random access*, which makes it easy to load a portion of a file individually.

Sometimes a program's *memory requirements* include space for a lot of *read-only* static data; typically the program only uses a small amount of this at any one time. For example Word-O-Matic needs static data such as window layouts, icon designs, font metrics and spelling dictionaries. Much of this information may be requested at any arbitrary time within the program, but when requested it is typically needed only for a short time. If the information is stored in main memory – if, for example, you hard-coded it into your program – it will increase the program's overall *memory requirements*.

Furthermore, you may need to change the configuration information separately from the program itself. What may seem to be different variants of the program (for different languages, or with different user interface themes) may use the same code but require different configuration data. Within a given configuration, many data items may be required at any time – window formats or fonts for example, so you cannot use APPLICATION SWITCHING techniques to bring this data in only for a given portion of the time. In general, however, much of the data will not be used at any given time.

Therefore: *Keep configuration data on secondary storage, and load and discard each item as necessary.*

Operating systems offer a simple way to store read-only static data: in a file on secondary storage. File systems provide random access, so it's easy to read just a single portion of a file, ignoring the remainder. You can load a portion of file into temporary memory, use it for a while, then discard it; you can always retrieve it again if you need it. In fact, with only a little additional complexity, you can make a file into a read-only database, containing data items each associated with a unique identifier.

Rather than hard-code each item of data specifically in the program code, you can give each item a unique identifier. When the program requires the data, it invokes a special routine passing the identifier; this routine loads the data from a 'resource file' and returns it to the program. The program may discard the loaded data item when it's no longer required. Typical resources are:

- Strings
- Screen Layouts
- Fonts
- Bitmaps, icons, and cursors.

For example, all of Word-O-Matic's window layouts, icon designs and text strings are stored in resource files. When they are required Word-O-Matic retrieves the data from the resource

file, and stores in a temporary memory buffer. The memory can be reused when the data is no longer required.

Consequences

The read-only static data doesn't clutter primary storage, reducing the program's *memory requirements*. Multiple programs can share the same resource file, reducing the *programmer effort* involved. Some operating systems share the loaded resources between multiple instances of the same program or library, further decreasing *memory requirements*. This also makes it easy to change the data without changing the program (e.g. to support multiple language strings), increasing the program's *design quality*.

However: this approach requires *programmer discipline* to place resources into the resource files, and to load and release the resources correctly. Loading and unloading resource files reduces the program's *time performance* somewhat. In particular they can impact its *start-up time*. Resource files also need *programmer effort* to implement, because you need some mechanism to unload (and reload) the resources. It's best if the *operating system* environment provides this support.

❖ ❖ ❖

Implementation

Since resource files are accessed randomly, applications need an index to locate data items (see DATA FILES). Most implementations of resource files hold this index in the resource file itself; typically at the start of the file. However this means that the resource file cannot simply be human-readable text, but must be *compiled*. Resource Compilers also typically convert the resource data into binary formats that can easily used by managed by application code, reducing the memory occupied and improving application performance.

In practice you usually need a logical separation between different resources in a system: the resources for one component are distinct from those for another, and the responsibility of separate teams. Thus most resource file frameworks support more than one resource file at a time.

Here are some things to consider when implementing resource files:

1. Making it easy for the programmer.

The task of loading and interpreting a resource is not a trivial one, so most systems provide library functions. You need basic functions to load and release the raw resources; typically you can also use more sophisticated functions:

- To manage the loading and release of resources, often from multiple files.
- To build graphical dialogs and constructs from the information
- To transfer bitmap and drawing resources (fonts, icons, cursors, drawing primitives) directly from the file to the screen without exposing their structure to the program.
- To insert parameters into the resource strings.

It's not just enough to be able to load an unload resources into systems at runtime; you also have to create the resources in the first place. Programming environments also provide facilities to help you produce resources:

- A Resource compiler – which creates a resource file database from a text file representation.

- Dialog Editors. These allow programmers to 'screen paint' screens and dialogs with user controls; programmers can then create resource-file descriptions from the results. An example is Microsoft Developer Studio (see Figure XX), but there are very many others.

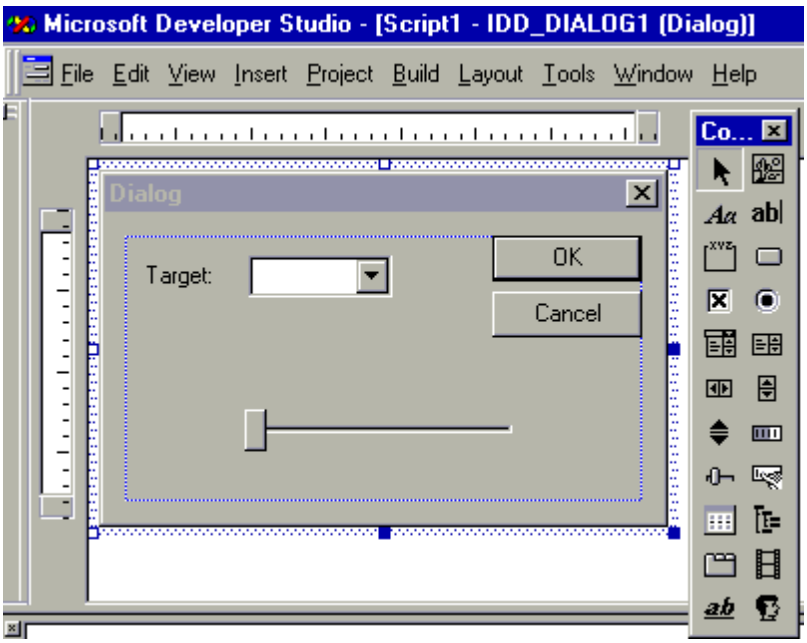

Figure 10: A Dialog Editor

2. Working with Resource Files to Save Memory.

Some resource file systems support compression. This has a small time overhead for each resource loaded, but reduces the file system space taken by the files. ADAPTIVE COMPRESSION algorithms are inappropriate for compression whole files, though, as it must be possible to decode any data item independently of the rest of the file. You can compress individual resources if they are large enough, such as images or sound files

It's worthwhile to take some effort to understand how resource loading works on your specific system as this can often help save memory. For example, Windows also supports two kinds of resource: `PRELOAD` and `LOADONCALL`. Preloaded resources are loaded when the program is first executed; a `LOADONCALL` resource loads only when the user code requests the specific resource. Clearly to save memory, you should prefer `LOADONCALL`. Similarly Windows 3.1 doesn't load strings individually, but only in blocks of 16 strings with consecutive ID numbers. So you can minimise memory use by arranging strings in blocks, such that the strings in a single block are all used together. By way of contrast, the Windows `LoadIcon` function doesn't itself access the resource file; that happens later when a screen driver needs the icon – so calling `LoadIcon` doesn't in itself use much memory. Petzold [1998] discusses the memory use of Windows resource files in more detail.

3. Font Files

You often want to treat font resources very differently from other kinds of resources. For a start, all applications will share the same set of fonts, and font descriptions tend to be much larger than other resources. A sophisticated font handling system will load only portions of each file as required by specific applications: perhaps only the implementation for a specific

font size, or only the characters required by the application for a specific string. The last approach is particularly appropriate for fonts for the Unicode characters, which may contain many thousands of images [Pike and Thompson 1993].

4. Implementing a resource file system.

Sometimes you need to implement your own resource file system. Here are some issues to consider:

4.1 Selecting variants. How will the system select which version of the resource files is loaded? There are various options. Some systems include only one resource file with each release. Others (e.g. most MS Windows applications) support variants for languages, but install only one; changing language means overwriting the files. Still other systems select the appropriate variant on program initialisation; for example chooses the variant by file extension (if the current language is number 01, application Word loads resource file `WORD.R01`). Other systems may even permit the system to change its language 'on the fly', although this is bound to require complex interactions between applications.

4.2 Inserting parameters into strings. The most frequent use of resources is in strings. Now displayed strings often contain variable parameters: "You have CC things to do, NN", where the number NN and the name CC vary according to the program needs. How do you insert these parameters?

A common way is to use C's `printf` format: "You have %d things to do, %s". This works reasonably, but has two significant limitations. First, the normal implementation of `printf` and its variants are liable to crash the program if the parameters required by the resource strings are not those passed to the strings. So a corrupt or carelessly constructed resource file can cause unexpected program defects. Second, the `printf` format isn't particularly flexible at supporting different language constructions – a German, for example, might want the two parameters in the other order: "%s: you have %d things to do.".

A more flexible alternative is to use numbered strings in the resource strings: "You have %1 things to do, %2". The program code has responsibility to convert all parameters to strings (which is simple, and can be done in a locale-sensitive way), and a standard function inserts the strings into the resource string. It is a trivial task to implement this function to provide default behaviour or an error message if the number of strings passed doesn't match the resource string.

Examples

Here's an example of an MS Windows resource file for an about box:

```
// About Box Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE  34, 22, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About DEMOAPP"
FONT 8, "MS Sans Serif"
BEGIN
    ICON            2,IDC_STATIC,11,17,18,20
    LTEXT           "Demonstration Application by Charles Weir",
                     IDC_STATIC,40,10,79,8
    LTEXT           "Copyright \251 1999",IDC_STATIC,40,25,119,8
    DEFPUSHBUTTON   "OK",IDOK,176,6,32,14,WS_GROUP
END
```

The C++ code to use this using the Microsoft Foundation Classes is remarkably trivial:

```
/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog

CAboutDlg::CAboutDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CAboutDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CAboutDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}
```

Note the explicit syntax of the comment, `{{AFX_DATA_INIT(CAboutDlg)`; this allows other Microsoft tools and 'wizards' to identify the location; the Wizard can determine any variable fields in the dialog box, and insert code to initialise them and to retrieve their values after the dialog has completed. In this case there are no such variables, so no code is present.

❖ ❖ ❖

Known Uses

Virtually all Apple Macintosh and MS Windows GUI programs use resource files to store GUI resources, especially fonts [Apple 1985; Petzold 1998]. EPOC stores all language-dependent information (including compressed help texts) in resource files, and allows the system to select the appropriate language at run-time. EPOC's Unicode font handling minimises the memory use of the font handler with a FIXED-SIZE MEMORY buffer to store a cached set of character images.EPOC16 used compression to reduce the size of its resource files [Edwards 1997].

Many computer games use resource files – from hand-helds with extra static ROM, to early microcomputers backed with cassette tapes and floppies, and state-of-the-art game consoles based on CD-ROMs. The pattern allows them to provide many more screens, levels, or maps than could possibly fit into main memory. Each level is stored as a separate resource in secondary storage, and then loaded when then user reaches that level. Since the user only plays on one level at any time, *memory requirements* are reduced to the storage required for just one level. This works well for arcade-style games where users play one level, then proceed to the next (if they win) or die (if they lose), because the sequence of levels is always predictable. Similarly many of the variations of multi-user adventure games keep the details of specific games: locations, local rules, monsters, weapons as resource files; as they tend to be large, they are often stored COMPRESSED.

See Also

DATA FILES provide writable data storage; APPLICATION SWITCHING and PACKAGES do for code what RESOURCE FILES does for unmodifiable data.

Each resource file can be stored in SECONDARY STORAGE or READ-ONLY MEMORY, and may use COMPRESSION..

Petzold [1998] and Microsoft [1997] describe Microsoft Windows Resource files. Tasker et al [2000] describes using EPOC resource files.

Packages

Also known as: Components, Lazy Loading, Dynamic Loading, Code Segmentation.

How can you manage a large program with lots of optional pieces?

- You don't have space in memory for all the code and is static data.
- The system has lots of functionality, but not all will be used simultaneously
- You may require any arbitrary combination of different bits of functionality.
- Development works best when there's a clear separation between developed components.

Some big programs are really small programs much of the time — the *memory requirements* of all the code are much greater the requirements for the code actually used in any given run of the program. For example Strap-It-On's Spin-the-Web™ web browser can view files of many different kinds at once, but it typically reads only the StrapTML local pages used by its help system. Yet the need to support other file types increases the program's code *memory requirements* even when they are not needed.

In these kinds of programs, there is no way of predicting in advance which features you'll need, nor of ordering them so that only one is in use at the same time. So the APPLICATION SWITCHING pattern cannot help, but you still want the benefits of that pattern – that the memory requirements of the system are reduced by not loading all of the program in to main memory at the same time.

Therefore: *Split the program into packages, and load each package only when it's needed.*

Any run-time environment which stores code in disk files must have a mechanism to activate executables loaded from disk. With a relatively small amount of effort, you can extend this mechanism to load additional executable code into a running program. This will only be useful, though, if most program runs do not need to load most of this additional code.

You need to divide the program into a *main program* and a collection of independently loaded *packages*. The main program is loaded and starts running. When it needs to use a facility in a package, a code routine somewhere will load the appropriate package, and call the package directly.

For example, the core of Spin-the-Web is a main program that analyses each web page, and loads the appropriate viewer as a package.

Consequences

The program will *require less memory* because some of its code is stored on SECONDARY STORAGE until needed.

The program will *start up quicker*, as only the small main program needs to be loaded initially, and can begin running with *less memory* than would otherwise be required. Because each package is fairly small, subsequent packages can be loaded in quickly without pauses for changing phases (as would be caused by the APPLICATION SWITCHING pattern).

Because packages aren't statically linked into the application code, dynamic loading mechanisms allow third parties or later developers to add functionality without changing or even stopping the main program. This significantly increases the system's *usability* and *maintainability*.

However: *Programmer effort* is needed to divide up the program into packages.

Many environments never unload packages, so the program's *memory requirements* can steadily increase, and the program can still run out of memory unless any given run uses only a small part of its total functionality. It takes *programmer effort* to implement the dynamic loading mechanism and to make the packages conform to it, and to define the strategy of when to load and unload the packages; or to optimise the package division and minimise the loading overhead. This mechanism can often be reused across programs, or it may be provided by the *operating system;* on the other hand many environments provide no support for dynamic loading.

Because a package isn't loaded until it's required dynamic loading means that the system may not detect a missing package until well after the program has loaded; this slightly reduces the program's *usability*. Also if access to the package is slow (for example, over the Web), the time taken to load a package can reduce the program's responsiveness, which also reduces the program's *usability*. This arbitrary delay also makes PACKAGES unsuitable for *real-time* operations.

Packages can be located remotely and changed independently from the main program. This produces *security* implications – a hostile agent may introduce viruses or security loopholes into the system by changing a package.

❖ ❖ ❖

Implementation

To support packages, you need three things:

1) A system that loads code into RAM to execute it.
2) A partition of the software into packages such that normally only a subset of the packages need be active.
3) Support for dynamically loadable packages — usually position independent or relocatable object code.

Here are some issues to consider when using or implementing packages as part of your system.

1. Processes as Packages

Perhaps the simplest form of a package is just a separate process. With careful programming, two processes that run simultaneously can appear to users as a single process, although there can be a significant cost in performance and program complexity to achieve this. Implementing each package in separate processes has several key advantages:

- The package and main program will execute in separate address spaces, so a fatal error in the package will not necessarily terminate the main program.
- The memory allocated to the package can be discarded easily, simply by terminating the process when the package is no longer needed.
- In some cases, the desired package may already exist as an application in its own right. For example we may want packages to do word-processing, drawing, or spreadsheet managing. Such applications exist already, and are implemented as separate processes.

There are two common approaches to use processes as packages:

1. The client can execute the process in the same way that, say, the operating system shell might do. It runs the process until its complete, perhaps reading its standard output (see DATA FILES)

2. The client can use operating system Inter-Process Communication (IPC) mechanisms to communicate with the process.

This second approach is taken by some forms of the Microsoft ActiveX ('COM') frameworks, by IBM's System Object Model (SOM) and by frameworks based on CORBA [Box 1998; Szyperski 1997; Henning and Vinoski 1999; Egremont 1998]. Each uses some form of PROXY [Gamma et al 1995, Buschmann et al 1996] to give the client access to objects in the package object. The *Essential Distributed Objects Survival Guide* [Orfali 1996] for a discussion and comparison of these environments.

2.Using Dynamically Linked Libraries as C++ Packages

You can also consider using Shared, or Dynamically Linked, Libraries (DLLs) as packages. Normally an executable loads all its DLLs during initialisation, so DLLs do not behave as packages by default. Most environments, however, provide additional mechanisms to load and unload DLLs 'on the fly'.

Some frameworks support delayed loading of DLLs: you can implement Microsoft COM objects, for example, as DLLs that load automatically when each object is first accessed. Although COM's design uses C++ virtual function tables, many other languages have provided bindings to access COM objects [Box 1998].

Other environments simply provide mechanisms to load the DLL file into RAM, and to invoke a function within the DLL. How can you use this to implement PACKAGES?

Typically you can identify their externally callable, *exported*, functions in DLLs either by function name or by function ordinal (the first exported function is ordinal 0, the second, 1, etc.). With either approach it would be quite a task to provide stubs for all the client functions and patch each to the correct location in the DLL.

Instead you can use object-orientation's dynamic binding to provide a simpler solution. This requires just a single call to one DLL entry point (typically at index 0 or 1). This function returns a pointer to a single instance of a class that supports an interface known to the client. From then on the client may call methods on that instance; the language support for dynamic linking ensures that the correct code executes. Typically this class is an ABSTRACT FACTORY or provides FACTORY METHODS [Gamma et al 1995]. Figure 11 shows such a library and the classes it supports.

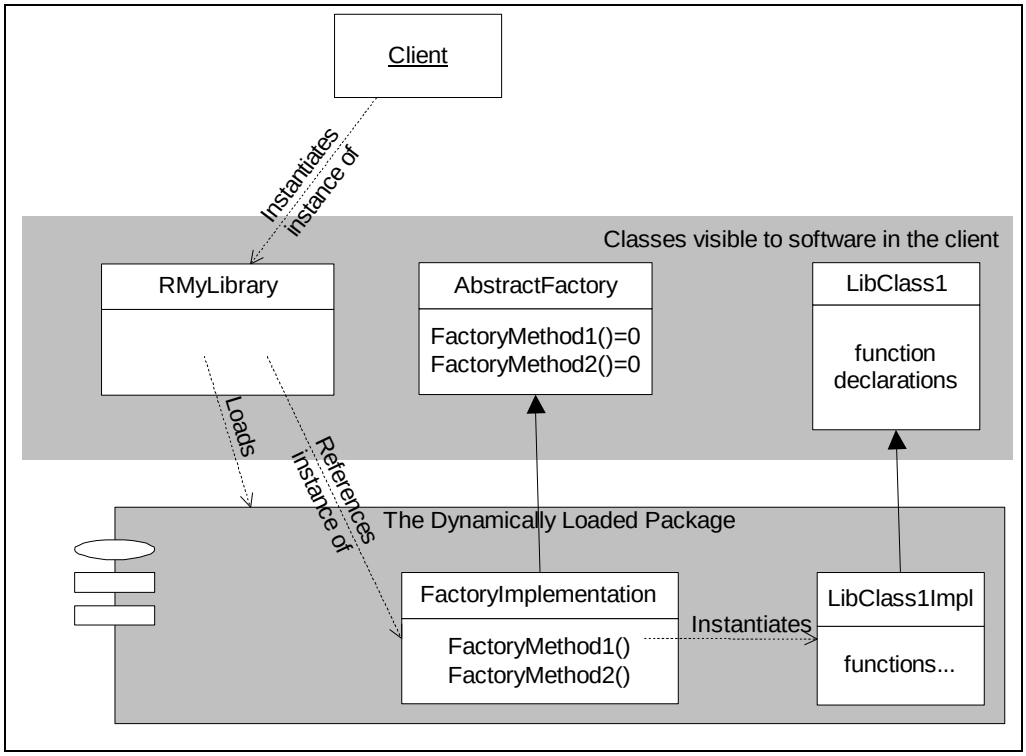

Figure 11: Dynamically loaded package with abstract factory

3. Implementing Packages using Code Segmentation

Many processor architectures and operating systems provide *code segmentation.* This supports packages at the machine code or object code level. A segmented architecture considers a program and the data that it accesses to be made up of some number of independent segments, rather than one monolithic memory space [Tannenbaum 1992].

Typically each segment has its own memory protection attributes — a data segment may be readable and writable by a single process, where a code segment from a shared library could be readable by every process in the system. As with packages, individual segments can be swapped to and from secondary storage by the operating system, either automatically or under programmer control. Linkers for segmented systems produce programs divided up into segments, again either automatically or following directives in the code.

Many older CPUs supported segmentation explicitly, with several *segment registers* to speed-up access to segments, and to ensure that the code and data in segments can be accessed irrespective of the segment's physical memory. Often processor restrictions limited the maximum size of each segment (64K in the 8086 architecture). More modern processor architectures tend to combine SEGMENTATION with PAGING.

4. Loading Packages

If you're not using segmentation or Java packages, you'll have to write some code somewhere in each application to load the packages. There are two standard approaches to where you put this code:

4.1 Manual loading: The client loads the package explicitly. This is best when:

1) The client must identify which it requires of several packages with the same interface. (E.g. loading a printer driver), or

2) The library provides relatively simple functionality, and it's clear when it needs to be unloaded.

4.2 Autoloading: The client calls any function supported by the library. This function is actually a stub provided by the client; when called it loads the library and invokes the appropriate entry point. This is better when:

1) You want a simple interface for the client, or
2) There are many packages with complicated interdependencies, so there's no easy algorithm to decide when to load a package.

Both approaches are common. For example, the Microsoft's COM framework and most EPOC applications do explicit loading; the Emacs text editor does autoloading [Box 1998; Tasker 2000; Stallman 1984].

5. Unloading packages

You'll save most memory if there's a mechanism to unload packages that are no longer required. To do this you need also a way to detect when there is no longer a need for the loaded code. In OO-environments this is easy to decide: the loaded code is no longer needed when there are no instances of objects supported by the package. So you can use REFERENCE COUNTING or GARBAGE COLLECTION to decide when to unload the code.

Loading a package takes time, so some implementations choose to delay unloading packages even when clients notify them that they may do so. Ideally they must unload these cached packages when system memory becomes short – the CAPTAIN OATES PATTERN.

6. Version Control and Binary Compatibility

You need to make sure that each package loaded works correctly with the component that loaded it – even if the two pieces of code were developed and released at different times. This requirement is often called 'binary compatibility', and is distinct from 'source code compatibility'. The requirements of 'binary compatibility' depend both on the language and on the compilation system used, but typically include:

- New versions of the clients expect the same externally visible entry points, parameters and return values; new services support the same ones.
- New clients don't add extra parameter values; new services don't add extra returned values. This is related to the rules for sub-typing – see Meyer [1997].
- New services support the same externally visible state as before.
- New services don't add new exceptions or error conditions unless existing clients have a way to handle them.

The problem of version control can become a major headache in development projects, when teams are developing several packages in parallel. Java, for example, provides no built-in mechanism to ensure that two packages are binary compatible; incompatible versions typically don't fail to load, but instead produce subtle program defects. To solve this problem, some environments provide version control in the libraries. Solaris, for example, supports major and minor version numbers for its DLLs. Minor version changes retain binary compatibility; major ones do not.

Drossopoulou et al [1998] discusses the rules for Java in more detail. [Symbian Knowledgebase 2000] discusses rules for C++ binary compatibility.

7. Optimising Packages

If you are using PACKAGES, you'll only have a fraction of the total code and data in memory at any given time – the *working set*. What techniques can you use to keep this working set to a minimum? You need to ensure that code that is used together is stored in the same package. Unfortunately, although organizing compiled code according to classes and modules is a good start, it doesn't provide an optimum solution. For example each of the many visual objects in the Strap-it-On's Mind-Mapping application have functionality to create themselves from vague text descriptions, to render animated pictures on the screen, to interact in weird and stimulating ways, to save themselves to store and to restore themselves again. Yet a typical operation on a mind-map will use only one of these types of functionality – but in every class (see figure XX).

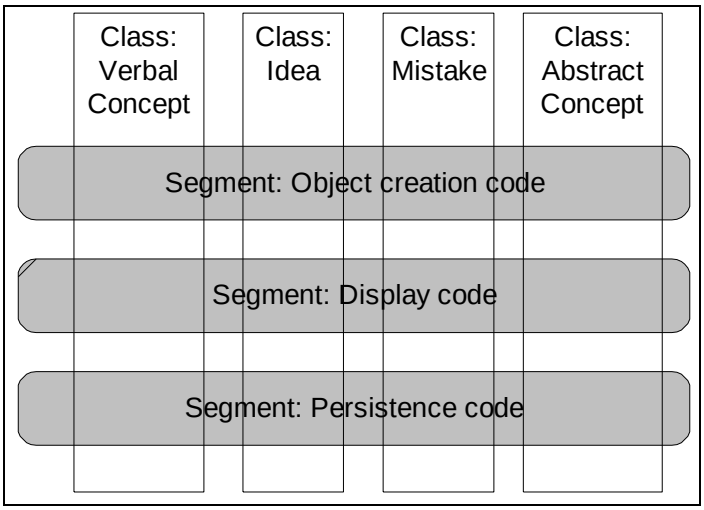

Figure 13: Example - Class divisions don't give appropriate segments

You could reorganise the code so that the compilation units correspond to your desired segments – but the results would be difficult to manage and for programmers to maintain. Using the terminology of Soni et al [1995], the problem is that we must organise the compiled code according to the *execution architecture* of the system, while the source code is organised according to its *conceptual architecture*. Most development environments provide *profilers* that show this execution architecture, so it's possible for programmers to decide a segmentation structure – at the cost of some *programmer effort* – but how should they implement it?

Some compilation environments provide a solution. Microsoft's C++ Compiler and DEC's FORTRAN compiler, for example, allow the user to partition each compilation unit into separate units of a single function, called '`COMDATs`'. Programmers can then order these into appropriate segments using a Link option: `/ORDER:@filename` [Microsoft 1997]. Sun's SparcWorks' `analyzer` tool automates the procedure still further, allowing 'experiments' with different segmentation options using profiling data, and providing a utility (`er_mapgen`) to generate the linker map file directly from these experiments.

For linkers without this option, an alternative is to pre-process the source files to produce a single file for each function, and then to order the resulting files explicitly in the linker command line. This requires additional *programmer discipline*, since it prevents us making code and data local to each source file.

Example

This EPOC C++ example implements the 'Dynamically loaded package with abstract factory' approach illustrated in Figure 11. This component uses animated objects in a virtual reality application. The animated objects are of many kinds (people, machinery, animals, rivers, etc.), only a few types are required at a time, and new implementations will be added later. Thus the implementations of the animated objects live in Packages, and are loaded on demand.

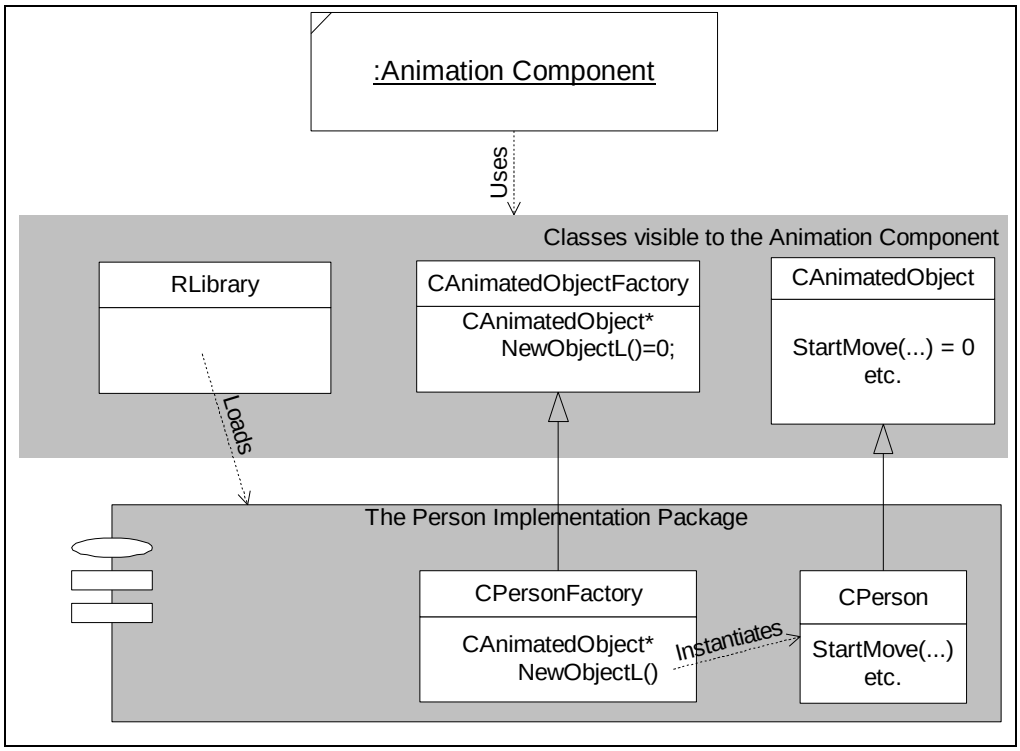

Figure 14: The Example Classes

1. Implementation of the Animation Component

EPOC implements packages as DLLs. The code in the animation component must load the DLL, and keep a handle to if for as long as it has DLL-based objects using its code. It might create a new `CAnimatedObject` using C++ something like the following (where the current object has a longer lifetime than any of the objects in package, and `iAnimatedObjectFactory` is a variable of type `CAnimatedObjectFactory*`)

```
iAnimatedObjectFactory = CreateAnimatedObjectFactoryFromDLL( fileName );
    CAnimatedObject* newAnimatedObject =
        iAnimatedObjectFactory->NewAnimatedObjectL();
```

The implementation of `CreateAnimatedObjectFactoryFromDLL` is as follows. It uses the EPOC class `RLibrary` as a handle to the library; the function `RLibrary::Load` loads the library; `RLibrary::Close` unloads it again. As with all EPOC code, it must implement PARTIAL FAILURE if loading fails. Also `libraryHandle` is a stack variable, so we must ensure it is `Close`'d if any later operations do a PARTIAL FAILURE themselves, using the cleanup stack function, `CleanupClosePushL`.

```
CAnimatedObjectFactory* CreateAnimatedObjectFactoryFromDLL(const TDesC& aFileName)
{
    RLibrary libraryHandle;
    TInt r=libraryHandle.Load(aFileName);
    if (r!=KErrNone)
        User::Leave(r);
    CleanupClosePushL(libraryHandle);
```


We must ensure that the library is the correct one. In EPOC every library (and data file) is identified by three Unique Identifier (UID) integers at the start of the file. The second UID (index 1) specifies the type of file:

```
if(libraryHandle.Type()[1]!=TUid::Uid(KUidAnimationLibraryModuleV01))
        User::Leave(KErrBadLibraryEntryPoint);
```

EPOC DLLs export functions by ordinal rather than by name [Tasker 1999a]. By convention a call to the library entry point at ordinal one returns an instance of the FACTORY OBJECT, `CAnimatedObjectFactory`.

```
    typedef CAnimatedObjectFactory *(*TAnimatedObjectFactoryNewL)();
    TAnimatedObjectFactoryNewL libEntryL=
        reinterpret_cast<TAnimatedObjectFactoryNewL>(libraryHandle.Lookup(1));
    if (libEntryL==NULL)
        User::Leave(KErrBadLibraryEntryPoint);
    CAnimatedObjectFactory *factoryObject=(*libEntryL)();
    CleanupStack::PushL(factoryObject);
```

We'll keep this factory object for the lifetime of the package, so we pass the `RLibrary` handle to its construction function:

```
    factoryObject->ConstructL(libraryHandle);
    CleanupStack::Pop(2); // libraryHandle, factoryObject
    return factoryObject;
}
```

The `CAnimatedObjectFactory` factory object is straightforward. It merely stores the library handle. Like almost all EPOC objects that own resources, it derives from the `CBase` base class, and provides a `ConstructL` function [Tasker et al 2000]. Some of its functions will be called across DLL boundaries; we tell the compiler to generate the extra linkup code using the EPOC `IMPORT_C` and `EXPORT_C` macros.

```
class CAnimatedObjectFactory : public CBase {
public:
    IMPORT_C ~CAnimatedObjectFactory();
    IMPORT_C void ConstructL(RLibrary& aLib);
    IMPORT_C virtual CAnimatedObject * NewAnimatedObjectL()=0;
private:
    RLibrary iLibraryHandle;
};
```

The implementations of the construction function and destructor are simple:

```
EXPORT_C void CAnimatedObjectFactory::ConstructL(RLibrary& aLib) {
    iLibraryHandle = aLib;
}

EXPORT_C CAnimatedObjectFactory::~CAnimatedObjectFactory() {
    iLibraryHandle.Close();
}
```

2. Implementation of the Package

The package itself must implement the entry point to return a new factory object, so it needs a class that derives from `CAnimatedObjectFactory`:

```
class CPersonFactory : public CAnimatedObjectFactory {
public:
    virtual CAnimatedObject * NewAnimatedObjectL();
};

CAnimatedObject * CPersonFactory::NewAnimatedObjectL() {
    return new(ELeave) CPerson;
}
```

The package also needs the class to implement the `CAnimatedObject` object itself:

```
class CPerson : public CAnimatedObject {
public:
    CPerson();
    // etc.
};
```

Finally, the library entry point simply returns a new instance of the concrete factory object (or null, if memory fails). `EXPORT_C` ensures that this function is a library entry point. In MS C++ we ensure that the function corresponds to ordinal one in the library by editing the 'DEF' file [Microsoft 1997]

```
EXPORT_C CAnimatedObjectFactory * LibEntry() {
    return new CPersonFactory;
}
```

❖ ❖ ❖

Known Uses

Most modern operating systems (UNIX, MS Windows, WinCE, EPOC, etc.) support dynamically linked libraries [Goodheart and Cox 1994, Petzold 1998, Symbian 1999]. Many applications delay the loading of certain DLLs, particularly for *add-ins* – added functionality provided by third parties. Lotus Notes loads viewer DLLs when needed; Netscape and Internet Explorer dynamically load viewers such as Adobe PDF viewer; MS Word loads document converters and uses DLLs for add-in extensions such as support for Web page editing. Some EPOC applications explicitly load packages: the Web application loads drivers for each transport mechanism (HTTP, FTP, etc.) and viewers for each data type.

Printer drivers are often implemented as packages. This allows you to add new printer drivers without restarting any applications. All EPOC applications dynamically load printer drivers where necessary. MS Windows 95 and NT do the same.

Many Lisp systems use dynamic loading. GNU Emacs, for example, consists of a core text editor package plus auto-loading facilities. Most of the interesting features of GNU Emacs exist as packages: intelligent language support, spelling checkers, email packages, web browsers, terminal emulators, etc [Stallman 1984].

Java makes great use of dynamic loading. Java loads each class only when it needs it, so each class is effectively a package. Java implementations may discard classes once they don't need them any more, using garbage collection, although many environments currently do not. Java applets are also treated as dynamically loading packages by Web browsers. A browser loads and run applets on pages it is displaying, and then stops and unloads applets when their containing pages are no longer displayed. [Lindholm and Yellin 1999]. The Palm Spotless JVM loads almost all classes dynamically, even those like String that are really part of the Java Language [Taivalsaari et al 1999].

Many earlier processors supported segmentation explicitly in their architecture. The 8086 and PDP-11 processors both implement segment registers. Programmers working in these environments often had to be acutely aware of the limitations imposed by fixed segment sizes; MS Windows 1, 2 and 3 all reflected the segmented architecture explicitly in the programming interfaces [Hamacher 1984; Chappell 1994].

See Also

APPLICATION SWITCHING is a simpler alternative to this pattern, which is applicable when the task divides into independent phases. PAGING is a more complex alternative. Unloaded packages can live on SECONDARY STORAGE, and maybe use COMPRESSION.

ABSTRACT FACTORY provides a good implementation mechanism to separate the client interfaces from the package implementations. VIRTUAL PROXIES can be used to autoload individual packages [Gamma et al 1995]. You may need REFERENCE COUNTING or GARBAGE COLLECTION to decide when to unload a package.

Coplien's *Advanced C++ Programming Styles and Idioms* [1994] describes dynamically loading C++ functions into a running program.

Paging Pattern

Also known as: Virtual Memory, Persistence, Backing Store, Paging OO DBMS.

How can you provide the illusion of infinite memory?[1]

- The *memory requirements* for the programs code and data are too big to fit into RAM.
- The program needs *random access* to all it's code and data.
- You have a fast *secondary storage* device, which can store the code and data not currently in use.
- To decrease *programmer effort* and *usability*, programmers and users should not be aware that the program is using secondary storage.

Some systems' *memory requirements* are simply too large to fit into the available memory. Perhaps one program's data structures are larger than the system's RAM memory, or perhaps a whole system cannot fit into main memory, although each individual component is small enough on its own.

For example, the Strap-It-On's weather prediction system, Rain-Sight™, loads a relatively small amount of weather information from it's radio network link, and attempts to calculate whether the user is about to get rained on. To do that, it needs to work with some very large matrices indeed – larger than can fit in memory even if no other applications were present at all. So the Rain-Sight marketing team have already agreed to distribute a 5 Gb 'coin-disk' pack with every copy of the program, ample for the Rain-Sight data. The problem facing the Rain-Sight developers is how to use it.

You can manage data on secondary storage explicitly using a DATA FILE. This has two disadvantages.

- The resulting code needs to combine processing the data with shuffling it between primary and secondary storage. The result will be *complex* and *difficult to maintain*, costing *programmer effort* to implement and *programmer discipline* to use correctly, because programmers will have to understand both domain-specific requirements of the program, and the fine points of data access.
- In addition this approach will tend to be *inefficient* for *random access* to data. If you read each item each time it's accessed, and write it back after manipulating it, this will require a lot of slow secondary storage access.

Other techniques, such as COMPRESSION and PACKED DATA, will certainly reduce the RAM memory requirements, but can only achieve a finite reduction; ultimately any system can have more data than will fit into RAM.

Therefore: *Keep a system's code and data on secondary storage, and move them to and from main memory as required.*

No software is completely random in its access to memory; at any given time a typical system will be working with only a small subset of the code and data it has available. So you need to

---

[1] *Sometimes the program is just too big, to complex, or you are too lazy to segment, subdivide, chain, phase, slice, dice, vitamise, or food process the code any more. Why **should** programmers have to worry about memory! Infinite memory for all is a **right**, not a privilege! Those small memory guys are just **no-life-losers**!*

keep only a relatively small *working set* in memory; the rest of the system can stay on secondary storage. The software can access this working set very fast, since it's in main memory. If you need to access information on secondary storage, you must change the working set, reading the new data required, and writing or discarding any data that occupied that space beforehand.

You must ensure that the working set changes only slowly – that the software exhibits *locality of reference*, and tends to access the same objects or memory area in preference to completely random access. It helps that memory allocators will typically put items allocated together in the same area of memory. So objects allocated together will typically be physically near to each other in memory, particularly when you're using FIXED ALLOCATION.

There are three forms of this pattern in use today [Tannenbaum 1992, Goodheart 1994]:

Demand Paging is the most familiar form. The memory management hardware, or interpreter environment, implements *virtual memory* so that there is an additional *page table* that maps addresses used in software to *pages* of physical memory. When software attempts to access a memory location without an entry in the page table, the environment frees up a page by saving its data, and loads the new data from secondary storage into physical memory, before returning the address of the physical memory found.

Swapping is a simpler alternative to paging, where the environment stops a process, and writes out all its data to Secondary Storage. When the process needs to process an event, the environment reloads all the data from secondary storage and resumes. This approach is common on portable PCs, where the entire environment is saved to disk, though the intent there is to save power rather than memory.

Object Oriented Databases are similar to Demand Paging, but the unit of paged memory is an object and its associated owned objects (or perhaps, for efficiency, a cluster of such objects). This approach requires more programmer effort than demand paging, but makes the data persistent, and allows multiple processes to share objects.

So, for example, the Rain-Sight team decided to use Paging to make use of their disk. The Strap-OS operating system doesn't support hardware-based paging, so the team hacked a Java interpreter to implement paging for each Java object. The team then defined objects to implement each related part of the Rain-Sight matrices (which are always accessed together), giving them acceptable performance and an apparent memory space limited only by the size of the 'coin disk'.

Consequences

Paging is the ultimate escape of the memory-challenged programmer. The programmer is **much less** aware of paging than any other technique, since paging provides the illusion of essentially infinite memory — the program's *memory requirements* are no longer a problem. So paging tends to increase other aspects of a system's *design quality*, and *maintainability* because memory requirements are no longer an overriding issue.

Paging needs little *programmer effort* and *programmer discipline* to use, because it doesn't need a logical decomposition of the program. Because paging does not require any artificial division of programs into phases or data into files it can make systems more *usable*. Programs using paging can easily accommodate more memory by just paging less, so paging improves *scalability*, as well.

Paging can make good *local* use of the available memory where the program's memory use is distributed *globally* over many different components, since different components will typically use their data at different times.

However: Paging reduces a program's *time performance*, since some memory accesses require secondary storage reads and writes. It also reduces the predictability of response times, making it unsuitable for *real-time systems*. Paging performs badly if the memory accesses do not exhibit locality of reference, and this may require *programmer effort* to fix.

Paging needs fast secondary storage to perform well. Of course 'fast' is a relative term; lots of systems have used floppy disks for paging. Because paging tends to make lots of small data transfers rather than a few large ones, the latency of the secondary storage device is usually more important than its throughput. Furthermore, PAGING's continuous use of secondary storage devices increases the system's *power consumption*, and reduces the lifetime of storage media such as flash RAM and floppy disks.

Since paging doesn't require *programmer discipline*, a program's *memory requirements* can tend to increase in paged systems, requiring more secondary storage and impacting the program's *time performance*. Paging requires no *local* support from within programs, but requires low-level *global* support, often provided by the *hardware and operating system*, or an interpreter or data manager. Because intermediate information can be paged out to secondary storage, paging can affect the *security* of a system unless the secondary storage is as well protected as the primary storage.

❖ ❖ ❖

Implementation

Paging is typically supported by two main data structures, see Figure XXX.

P*age frames* live in main memory, and contain the 'paged in' RAM data for the program. Each page frame also has control information: the secondary storage location corresponding to the current data and a *dirty bit*, set when the page memory has been changed since loading from secondary storage.

The *page table* also lives in main memory, and has an entry for each page on secondary storage. It stores that page is resident in memory, and if so, in which page frame it is stored. Figure XXX below shows a Page Table and Page Frames.

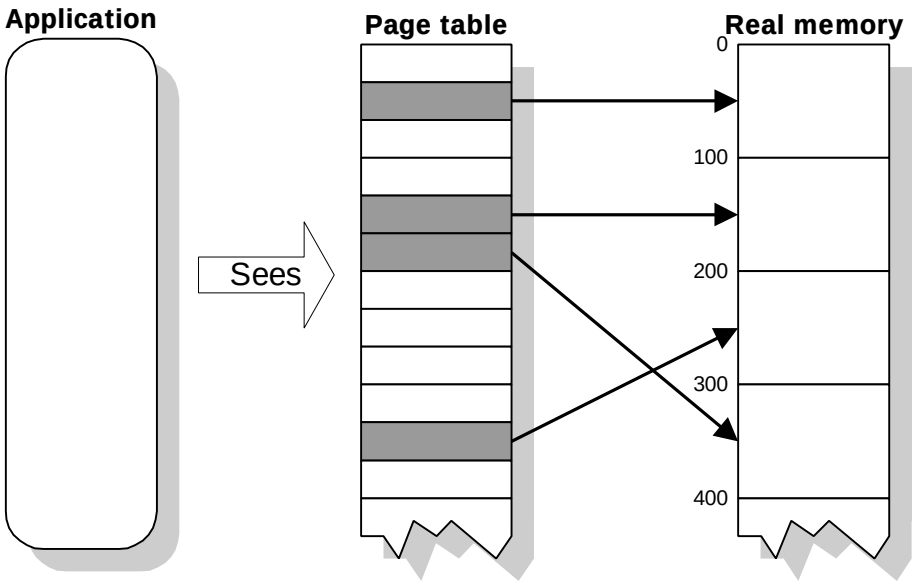

Figure 15: Page Table and Page Frames

As you run a paging application, it accesses memory via the page table. Memory that is paged in can be read and written directly: writing to a page should set the page's dirty bit. When you try to access a page that is 'paged out ' (not in main memory) the system must load the page from secondary storage, perhaps saving an existing page in memory back to secondary storage to make room for the new page. Trying to access a page in secondary storage page is called a *page fault*. To handle a page fault, or to allocate a new page, the system must find a free page frame for the data. Normally, the frame chosen will already contain active data, which must be discarded, and if the dirty bit is set, the system must write the contents out to secondary storage. Once the new frame is allocated, or its contents are loaded from secondary storage, the page table can be updated and the program's execution continue.

Here are some issues to consider when implementing paging.

1. Intercepting Memory Accesses

Probably the single most difficult part of implementing paging is the need to intercept memory accesses. In addition, this intercept must distinguish access for writing, which must set the 'dirty bit', from access for read, which doesn't.

There are several possible mechanisms:

MMU	Many modern systems have a Memory Management Unit (MMU) in addition to the Central Processing Unit (CPU). These provide a set of *virtual memory maps (*typically one for each process), which map the memory locations requested by the code to different real memory addresses. If the program accesses an address that hasn't been loaded, this causes a page fault interrupt, and the interrupt driver will load the page from secondary storage. The MMU also distinguishes pages as read-only and read-write. An attempt to write to a read-only page also causes a page-fault interrupt, which makes it easy to set the dirty bit for that page.

Interpreter	It's fairly straightforward to implement paging for interpreted environments. The run-time interpreter must implement any accesses to the program or its data, so it is relatively easy to intercept accesses and to distinguish reads from writes.
Process Swap	When you swap entire processes, you don't need to detect memory access as processes are not running when they are swapped out.
Data Manager	For programs in an environment with no built-in paging, we can use 'smart pointers' to classes to intercept each access to an object. Then a *data manager* can ensure that the object is in store and to manage loading, caching and swapping. In this case it's appropriate to page entire objects in and out, rather than arbitrarily sized pages.

2. Page Replacement

How can you select which page frame to choose to free up to take a new or loaded page? The best algorithm is to remove the page that will be needed the furthest into the future —the least important page for the system's immediate needs [Tannenbaum 1992]. Unfortunately this is usually impossible to implement, so instead you have to guess the future on the basis of the recent past. Removing the *least frequently used (LFU)* page provides the most accurate estimation, but is quite difficult to implement. Almost as effective but easier to implement is a *least recently used (LRU)* algorithm, which simply requires keeping a list of all page frames, and moving each page to the top of this list as it is used. Choosing a page to replace at random is easy to implement and provides sufficient performance for many situations.

Most implementations of MMU paging incorporate Segmentation techniques as well (see PACKAGES). Since you already have the process's virtual data memory split into pages, it's an obvious extension to do the same for code. Code is READ-ONLY, and typically needs only very trivial changes when it's loaded from secondary storage to memory. So there's no point in wasting space in the swap file; you can take the code pages directly from the code file when you want them and discard them when they're no longer needed.

3. Working Set Size

A program's *working set size* is the minimum amount of memory it needs to run without excessive page faults. Generally, the larger the page size, the larger the working set size. A program's working set size determines whether it will run well under any given paging system. If the working set size is larger than the real memory allocated to page frames, then there will be an excessive number of page faults. The system will start *thrashing*, spending its time swapping pages from main memory to secondary storage and back but making little progress executing the software.

To avoid thrashing, do less with your program, add real memory to the system, or optimise the program's memory layout using the techniques discussed in the PACKAGES PATTERN.

4. Program Control of Paging

Some programs do not have *locality of reference*. For example a program might traverse all its data reading each page exactly once in order. In this case, each page will be paged in only once, and the best page to replace will be the *most* frequently or recently used. To help in such a case, some systems provide an interface so that the programmer can control the paging system. For example, the interface might support a request that a particular page be paged out.

Alternatively it might allow a request that a particular page be *paged in* — for example the program above will know which page will be needed *next* even before processing the current page.

Other program code may have real-time constraints. Device drivers, for example, must typically respond to interrupt events within microseconds. So device driver data must not be paged out. Most systems support this by 'tagging' certain areas of code as with different attributes. For example, Microsoft's Portable Executable Format supports Pre-load and Memory-resident options for its 'virtual device driver' executable files [Microsoft 1997]

Example

The following code implements a simple framework to page individual C++ objects.

It's very difficult indeed to intercept all references to a C++ object without operating system support – in particular we can't intercept the '`this`' pointer in a member function. So it's a bad idea to page instances of any C++ class with member functions. Instead we make each object store its data in a separate data structure and access that structure through special member functions that control paging. The object itself acts as a PROXY for the data, storing a page number rather than a pointer to the data. Figure XX below shows a typical scenario:

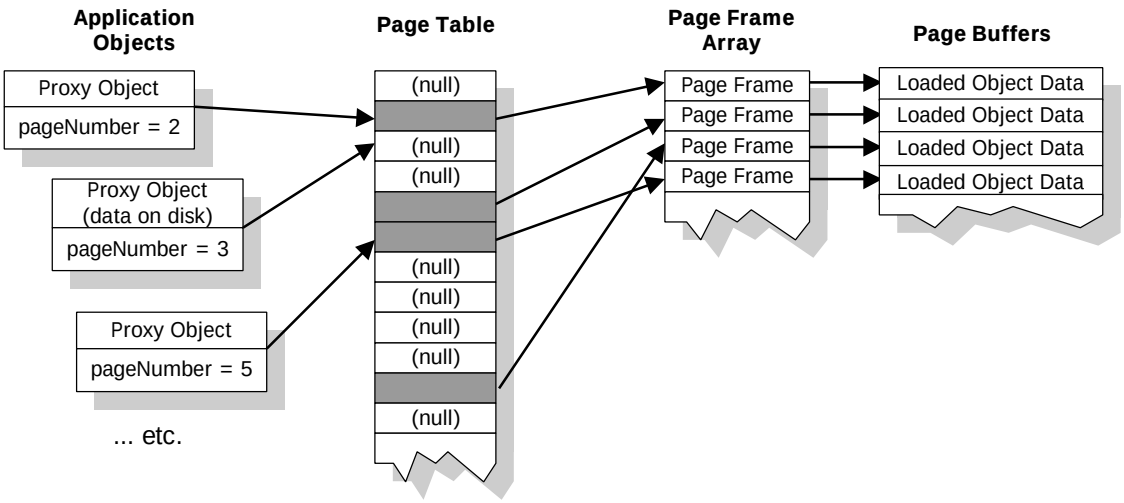

Figure 16: Objects in memory for the Paging Example

The Page Table optimises access to the data in RAM: if its entry is non-null for a particular page, that page is loaded in RAM and the application object can access its data directly. If an application object tries to access a page with a null Page Table entry, it means that object's data isn't loaded. In that case the paging code will save or discard an existing page frame and load that object's data from disk.

1. Example Client Implementation

Here's an example client implementation that uses the paging example. It's a simple bitmap image containing just pixels. Other paged data structures could contain any other C++ primitive data types or `structs`, including pointers to objects (though not pointers to the paged data structure instances, of course, as these will be paged out).


```
typedef char Pixel;

class BitmapImageData {
    friend class BitmapImage;
    Pixel pixels[SCREEN_HEIGHT * SCREEN_WIDTH];
};
```

The PROXY class, `BitmapImage`, derives its paging functionality from the generic `ObjectWithPagedData`. The main constraint on its implementation is that all accesses to the data object must be through the base class `GetPagedData` functions, which ensure that the data is paged into RAM. It accesses these through functions to cast these to the correct type:

```
class BitmapImage : public ObjectWithPagedData {
private:
    BitmapImageData* GetData()
        { return static_cast<BitmapImageData*>(GetPagedData()); }
    const BitmapImageData* GetData() const
        { return static_cast<const BitmapImageData*>(GetPagedData()); }
```

The constructor must specify the `PageFile` object and initialise the data structure. Note that all these functions can be inline:

```
public:
    BitmapImage( PageFile& thePageFile )
        : ObjectWithPagedData(thePageFile) {
        memset( GetData(), 0, sizeof(BitmapImageData) );
    }
```

And all functions use the `GetData` functions to access the data. Note how the C++ `const`-correctness ensures that we get the correct version of the data function; non-`const` accesses to `GetData`() will set the 'dirty bit' for the page so it gets written back to file when paged out.

```
    Pixel GetPixel(int pixelNumber) const {
        return GetData()->pixels[pixelNumber];
    }
    void SetPixel(int pixelNumber, Pixel newValue) {
        GetData()->pixels[pixelNumber] = newValue;
    }
};
```

And that's the full client implementation. Simple, isn't it?

To use it we need to set up a page file – here's one with just four page buffers:

```
PageFile pageFile("testfile.dat", sizeof( BitmapImageData ), 4 );
```

And then we can use can use `BitmapImage` as any other C++ object:

```
BitmapImage* newImage = new BitmapImage(pageFile);
newImage->SetPixel(0, 0);
delete newImage;
```

2. Overview of the Paging Framework

Figure XXX below shows the logical structure of the Paging Framework using UML notation [Fowler and Scott 1997]. The names in normal type are classes in the framework; the others are implemented as follows:

- *Page Table Entry* is a entry in the `pageTable` pointer array.
- *Page in RAM* is a simple (void*) buffer.
- *Page on Disk* is a fixed page in the disk file.
- *Client Implementation* is any client class, such as the BitmapImage class above.

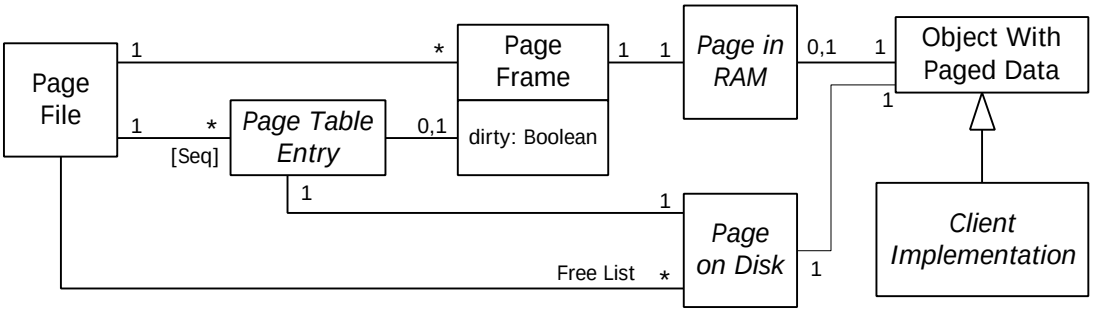

Figure 17: UML Diagram: Logical structure of the object paging system

The page frames and page table are a FIXED DATA STRUCTURE, always occupying the same memory in RAM.

3. Implementation of `ObjectWithPagedData`

ObjectWithPagedData is the base class for the Client implementation classes. It contains only the page number for the data, plus a reference to the PageFile object. This allows us to have several different types of client object being paged independently.

```
class ObjectWithPagedData {
private:
    PageFile& pageFile;
    const int pageNumber;
```

All of its member operations are `protected`, since they're used only by the client implementations. The constructor and destructor use functions in `PageFile` to allocated and free a data page:

```
ObjectWithPagedData::ObjectWithPagedData(PageFile& thePageFile)
    : pageFile( thePageFile ),
      pageNumber( thePageFile.NewPage() )
{}

ObjectWithPagedData::~ObjectWithPagedData() {
    pageFile.DeletePage(pageNumber);
}
```

We need both `const` and non-`const` functions to access the paged data. Each ensures there's a page frame present, then accesses the buffer; the non-`const` version uses the function that sets the dirty flag for the page frame:

```
const void* ObjectWithPagedData::GetPagedData() const {
    PageFrame* frame = pageFile.FindPageFrameForPage( pageNumber );
    return frame->GetConstPage();
}

void* ObjectWithPagedData::GetPagedData() {
    PageFrame* frame = pageFile.FindPageFrameForPage( pageNumber );
    return frame->GetWritablePage();
}
```

3. Implementation of `PageFrame`

A `PageFrame` object represents a single buffer of 'real memory'. If the page frame is 'In Use', a client object has accessed the buffer and it hasn't been saved to disk or discarded. The member `currentPageNumber` is set to the appropriate page if the frame is In Use, or else to `INVALID_PAGE_NUMBER`. `PageFrame` stores the 'dirty' flag for the buffer, and sets it when any client accesses the `GetWritablePage` function.

```
class PageFrame {
    friend class PageFile;

private:
    enum { INVALID_PAGE_NUMBER = -1 };
    bool  dirtyFlag;
    int   currentPageNumber;
    void* bufferContainingCurrentPage;
```

The constructor and destructor simply initialise the members appropriately:

```
PageFrame::PageFrame( int pageSize )
    : bufferContainingCurrentPage( new char[pageSize] ),
      dirtyFlag( false ),
      currentPageNumber(PageFrame::INVALID_PAGE_NUMBER)
{}

PageFrame::~PageFrame() {
    delete [] (char*)(bufferContainingCurrentPage);
}
```

`GetConstPage` and `GetWritablePage` provide access to the buffer:

```
const void* PageFrame::GetConstPage() {
    return bufferContainingCurrentPage;
}

void* PageFrame::GetWritablePage() {
    dirtyFlag = true;
    return bufferContainingCurrentPage;
}
```

And the other two member functions are trivial too:

```
int PageFrame::PageNumber() {
    return currentPageNumber;
}
bool PageFrame::InUse() {
    return currentPageNumber != INVALID_PAGE_NUMBER;
}
```

4. Implementation of `PageFile`

The `PageFile` object manages all of the important behaviour of the paging system. It owns the temporary file, and implements the functions to swap data buffers to and from it.

`PageFile`'s main structures are as follows:

`pageTable` is a vector, with an entry for each page in the page file. These entries are `null` if the page is swapped to secondary storage, or point to a `PageFrame` object is the page is in RAM.

`pageFrameArray` contains all the `PageFrame` objects. It's an array to make it easy to select one at random to discard.

`listOfFreePageNumbers` contains a queue of pages that have been deleted. We cannot remove pages from the page file, so instead we remember the page numbers to reassign when required.

So the resulting private data is as follows:

```
class PageFile {
    friend class ObjectWithPagedData;
private:
    vector<PageFrame*> pageTable;
    vector<PageFrame*> pageFrameArray;
    list<int> listOfFreePageNumbers;
    const int pageSize;
    fstream fileStream;
```

`PageFile`'s constructor must initialise the file and allocate all the FIXED DATA STRUCTURES. It requires a way to abort if the file open fails; this example simply uses a variant of the ASSERT macro to check:

```
PageFile::PageFile( char* fileName, int pageSizeInBytes, int nPagesInCache )
    : fileStream( fileName,
                  ios::in| ios::out | ios::binary | ios::trunc),
      pageSize( pageSizeInBytes ) {
    ASSERT_ALWAYS( fileStream.good() );
    for (int i = 0; i<nPagesInCache; i++) {
        pageFrameArray.push_back( new PageFrame( pageSize ) );
    }
}
```

The destructor tidies up memory and closes the file. A complete implementation would delete the file as well:

```
PageFile::~PageFile() {
    for (vector<PageFrame*>::iterator i = pageFrameArray.begin();
         i != pageFrameArray.end(); i++ )
        delete *i;
    fileStream.close();
}
```

The function `NewPage` allocates a page on disk for a new client object. It uses a free page on disk if there is one, or else allocates a new `pageTable` entry and expands the page file by writing a page of random data to the end.

```
int PageFile::NewPage() {
    int pageNumber;
    if (!listOfFreePageNumbers.empty()) {
        pageNumber = listOfFreePageNumbers.front();
        listOfFreePageNumbers.pop_front();
    } else {
        pageNumber = pageTable.size();
        pageTable.push_back( 0 );
        int newPos = fileStream.rdbuf()->pubseekoff( 0, ios::end );
        fileStream.write(
            (char*)pageFrameArray[0]->bufferContainingCurrentPage,
            PageSize() );
    }
    return pageNumber;
}
```

The corresponding `DeletePage` function is trivial:

```
void PageFile::DeletePage(int pageNumber) {
    listOfFreePageNumbers.push_front(pageNumber);
}
```

The function `FindPageFrameForPage` assigns a `PageFrame` for the given page number and ensures that the page is in RAM. If there's already a `PageFrame` for the page, it just returns the pointer; otherwise it finds a `PageFrame` and fills it with the requested page from disk.

```
PageFrame* PageFile::FindPageFrameForPage( int pageNumber ) {
    PageFrame* frame = pageTable[pageNumber];
    if (frame == 0) {
        frame = MakeFrameAvailable();
        LoadFrame( frame, pageNumber );
        pageTable[pageNumber] = frame;
    }
    return frame;
}
```

The function `MakeFrameAvailable` assigns a frame by paging out or discarding an existing page, chosen at random.

```
PageFrame* PageFile::MakeFrameAvailable() {
    PageFrame* frame = pageFrameArray[ (rand() * pageFrameArray.size()) /
                                       RAND_MAX ];
    if (frame->InUse()) {
        SaveOrDiscardFrame( frame );
    }
    return frame;
}
```

The function that provides the meat of the paging algorithm is `SaveOrDiscardFrame`. This writes out the page to the corresponding location in file – if necessary – and resets the page table entry.

```
void PageFile::SaveOrDiscardFrame( PageFrame* frame ) {
    if (frame->dirtyFlag) {
        int newPos = fileStream.rdbuf()->pubseekoff( frame->PageNumber() *
                                                     PageSize(), ios::beg );
        fileStream.write( (char*)frame->bufferContainingCurrentPage,
                          PageSize() );
        frame->dirtyFlag = false;
    }
    pageTable[frame->PageNumber()] = 0;
    frame->currentPageNumber = PageFrame::INVALID_PAGE_NUMBER;
}
```

And finally, the corresponding function to load a frame is as follows:

```
void PageFile::LoadFrame( PageFrame* frame, int pageNumber ) {
    int newPos = fileStream.rdbuf()->pubseekoff( pageNumber * PageSize(),
                                                 ios::beg );
    fileStream.read( (char*)frame->bufferContainingCurrentPage, PageSize() );
    frame->currentPageNumber = pageNumber;
 }
```

❖ ❖ ❖

Known Uses

Almost every modern disk operating system provides paged virtual memory, including most versions of UNIX including LINUX, Mac OS, and MS Windows [Goodheart 1994; Card, Dumas, Mével 1998, Microsoft 1997]

OO Databases almost all use some form of object paging. ObjectStore uses the UNIX (or NT) paging support directly, but replaces the OS-supplied paging drivers with drivers that suit the needs of OO programs with persistent data [Chaudhri 1997].

Infocom games implemented a paged interpreter on machines like Apple-IIs and early PCs, paging main memory to floppy disks [Blank and Galley 1995]. This enabled games to run on machines with varying sizes of memory — although of course games would run slower if there was less main memory available. The LOOM system implemented paging in Smalltalk for Smalltalk [Kaehler and Krasner 1983].

See Also

The other patterns in this chapter — PROCESSES, DATA FILES, PACKAGES, and RESOURCE FILES— provide alternatives to this pattern. Paging can also use COPY ON WRITE to optimise access to *read-only* storage, and can be extended to support SHARING. System Memory is a *global* resource, so some operating systems implement CAPTAIN OATES, discarding segments from different processes rather than from the process that requests a new page.

An INTERPRETER [Gamma et al 1995] can make PAGING transparent to user programs. VIRTUAL PROXIES and BRIDGES [Gamma et al 1995], and ENVELOPE/LETTER or HANDLE/BODY [Coplien 1994] can provide paging for objects without affecting the objects' client interfaces.
