Portable Medical Laboratory Applications Software

by Jerome A. Silbert

Laboratory Service, Veterans Administration Medical Center
West Haven, Connecticut

Abstract

Portability implies that a program can be run on a variety of computers with minimal software revision. The advantages of portability are outlined and design considerations for portable laboratory software are discussed. Specific approaches for achieving this goal are presented.

Introduction

The present state of software is what musical composition would be if there were dozens of different musical notations, and the instruments had no standard musical scale. There are dozens of computer languages each with several different dialects. There are dozens of different computers, each with different operating systems. Small wonder there is more discord than harmony.

Despite dramatic advances in computer technology, programs for clinical and anatomic pathology still do not meet our requirements. Developing software for the clinical laboratory requires thousands of hours of effort by experts in computer science, laboratory science, and laboratory management. Although there are many programs written for clinical and anatomic pathology, virtually no opportunity for exchange among interested parties exists because of incompatibilities between languages and computers. The result is an enormous duplication of effort rewriting programs that have already been written.

One approach to this problem is to develop programs that are portable. A portable program can be run on a wide variety of computers. Portability permits dissemination of programs and serves as a basis for an evolving system of improved software. Advances in technology rest on a foundation of previous discovery and invention. Computer software is no different and is handicapped by the fact that so much effort is spent in redundant rather than cumulative innovation.

One positive consequence of portability is that the life-cycle of the software can be expected to exceed that of the hardware. Often users are hostage to a particular vendor of hardware because of the cost and inconvenience of rewriting programs for a different type of computer. Portability allows a greater freedom of choice in computers, so the user can take advantage of advances in hardware technology.

Portability is dramatically cost effective when there are a large number of users with a variety of computers. The value of portability is proportional to the value of the software supported by the portable environment. An investment of a few months in developing a portable environment can save years of work that would otherwise be spent "reinventing the wheel".

Is software portability possible? Can we avoid reinventing the wheel and being tied to a particular computer by our software? To answer these questions it is important to distinguish between the technical and political aspects of portability. To prove technical feasibility we must demonstrate that laboratory software can indeed run on a variety of computers. The political issue involves how widely such software would be accepted.

One of the political barriers to portability are functional differences between laboratories. A portable laboratory system must demonstrate its utility in different laboratory environments. Though political obstacles to portability are more formidable than the technological ones, until technical feasibility is demonstrated, discussions of political feasibility are moot.

At the West Haven Veterans Administration Medical Center we have been exploring the technical feasibility of portability by writing programs for the clinical laboratory. In this proof-of-concept demonstration, we have drawn our functional model from the book "Achieving the Optimum Information System for the Laboratory" (1). We believe that portability is technically feasible for medically oriented data processing. This paper will describe some of our efforts towards achieving portability in medical application programs.

Technical Feasibility

A number of software engineering techniques
provide a remarkable degree of portability considering the formidable variety of computers and operating systems. This is not to say that all software can be made portable. No matter how portably written, a multi-user data base will not run on a pocket calculator. Nevertheless, within defined constraints with respect to memory and operating system, sophisticated portable programs can be written.

Our quest began in the summer of 1978 when we started to write medical applications programs for the clinical chemistry laboratory. We were aware that most in-house software development efforts fail, but as long as we were going to make the effort we tried to develop a system that would be useful locally and in other laboratories as well.

In the beginning we were naive in assuming that portability could be achieved by simply writing programs in a "standard" language. After months of work, our hopes were rapidly dimming. The code was becoming difficult to manage, and we started to run out of memory. Unfortunately, this is a familiar lament to those who develop laboratory systems. Just when things were looking grim an article appeared in Science about portable software for X-ray Crystallography (2). Portability had been achieved, but would it work for us?

Further research led us to the proceedings of the conference "Software Standards in Chemistry" (3), and thence to a further discovery - the Software Tools Group (4). The "tools group" consists of computer specialists from around the country inspired by an outstanding text on software design called "Software Tools" (5). The tools group was involved in a unique effort to make software portability practical by developing portable programs which themselves could be used for further software development. (This paper was written using the software tools editor and document formatter).

The Virtual Operating System

Through the software tools group we became aware of a further extension of the portability concept involving what is termed the "Virtual Operating System" (6). A virtual image is one which only appears to be present, such as a reflection in a mirror. Similarly, a Virtual Operating System does not actually exist. However, a real computer, using a set of procedures called 'The Virtual Operating System Library' may be programmed to behave as if it were an idealized computer. The Virtual Operating System (VUS) provides a 'portable environment' where all the low-level operations such as opening, reading, writing and closing files are defined in a machine-independent manner.

The VUS is essential to portability. The application programs address the real computer only by way of the VUS. Thus the problems associated with running programs on different computers are reduced to the implementation of the VUS and not the programs supported by the library.

The procedures of the VUS are divided into two categories:

Primitives: procedures that have a standard user interface and action, but are usually customized to the operating system of the target computer and tailored to run efficiently on that operating system. For example the primitive OPEN that is used to open a file.

Library Routines: procedures that have a standard user interface and action, but need not be customized to the target operating system and are usually written in a portable manner. For example the library routine ITOC that converts an integer to a string of characters.

From relatively few primitives, other more complex procedures can be built for the output of error messages, remarks, and building lines of text for formatting reports. A key concept in portability is to start with a VUS tailored to the target system and from this build the applications programs.

It is imperative that the procedures of the VUS conform to a uniform standard. That is, take the same arguments and return the same status conditions. As applied to portable laboratory software, we had to define a VUS to provide the low-level functions necessary for a laboratory system. Fortunately, much of the work had already been done at Lawrence Berkley Laboratories in California where the software tools group originated. There already was a VUS that defined many of the procedures necessary to implement a portable environment (4).

VUS Extensions

We soon discovered that the VUS did not contain certain procedures crucial to clinical laboratory applications, such as random access. Since random access is essential to data base applications, it was necessary to design a flexible yet portable means of handling random access. Random access I/O primitives were defined following the suggestions in the "Software Standards in Chemistry" (3). Since data in random access files may be characters, integers or other forms of data representation, we decided to simply consider the data in the file as a string of bytes and transfer them without any changes.

Another deficiency was the absence of primitives to implement sharing the data base among multiple users. Analysis of the multiuser issue identified two major requirements. The first was that information written to a file be available, i.e. shared, with other users as soon as it is written. The second was that some means of protecting data while it was being updated was mandatory in order to prevent corruption of the data base.

The way the shared file issue was dealt with
is instructive as it follows the principles of portable software design. The object was to define a function so that it could be implemented on a wide variety of computers. Part of the definition of shared files was that they be random access files. We defined a function to open random access files with four access modes: READONLY, WRITEONLY, READWRITE, and SHARED. All files opened with the shared access mode would be shared with users who also opened the file in a shared mode. Files opened in other access modes would not be shared. It was illegal to open the same file in an unshared access mode if it was previously opened as shared by the same user. The details of the most efficient implementation of shared access differ depending upon the target operating system.

As a first pass, we took what could be termed the "lowest common denominator approach". That is to implement sharing in a way that is workable on a wide variety of operating systems. This method was to open the file, perform the I/O and then close the file again. This resulted in a "flushing" of the I/O buffers and forced all reads and writes to the disk. This slows down the program - especially when a large number of records are read or written - but this situation is generally temporary until the sharing primitives are optimized for the target operating system. Many operating systems allow shared files to be opened using the same I/O buffer for users sharing the file and this is more efficient than the lowest common denominator approach. It is also possible to define a function to open a file in a shared mode handles all the I/O and keeps its own buffers - passing out the data and taking in the data from various users and sharing can be implemented through design of this process.

The second issue, that of protecting a file or record requires what we term "locks". Suppose users "A" and "B" share a file and "B" reads and updates a record from this file. Meanwhile, before "B" has had a chance to write the record back, "A" reads the same record and also changes it. Then "B" writes its version of the record back into the file. Then "A" writes its version of the record back. Unfortunately "A"'s record wrote right over the change that "B" made. This is why locks are needed.

With locks, "B" first locks the record, then reads it. If "A" wishes to read the record he first attempts to lock it - but now finds the record locked - therefore "A" has to wait until it becomes unlocked. This insures that "B"'s record will be updated before "A" reads it. The question was how to define our locking procedure and implement it in a portable environment.

After some trial and error we decided that it was best to lock the entire data base as the efficiency gained by locking individual records and files was lost in the overhead of keeping track of which records and files were locked. In order to increase access to the data base we provided for a difference between locking to read and locking to update. Data-base readers will not lock out other readers but will lock out those processes that seek to alter the data base. Writers will lock out both readers and other writers.

We called the locking process the "COP" because it was a traffic director for users requesting information from the data base. The COP takes two arguments: a request to lock or unlock, and an access-mode to read or write. The COP "negotiates" for access to the data base with the other processes in the system and returns control to a requesting process when it obtains access. If the COP waits too long it will time out with a message to the user.

The COP itself is not a primitive but is constructed from three primitives added to the VUS. The issue of data base locking is part of a rather complex subject called "concurrent programming". It would not be appropriate here to engage in a lengthy discourse on concurrent programming on the VUS, however a few important points can be made. The primitives necessary for concurrent programming are called semaphores.

Semaphores serve as signals between processes that indicate when a process wants exclusive use of the data base. We defined two semaphores that operate on a "traffic light" (in keeping with the analogy of the COP) that can be seen by all user programs. The semaphore called "RED" changes a green light to red and returns a status indicating "wait". The second semaphore called "GREEN" changes the light from red back to green. The most important feature of these semaphores is that they carry out their operations as if they were a single uninterruptable machine instruction.

There are a number of ways that these semaphores can be implemented. One approach is to take advantage of the fact that many multiuser operating systems allow a file to be exclusively opened. If one attempts to exclusively open a file and can not, this is like the light being red. Exclusively opening the file causes the green light to turn red. Closing the file causes the red light to turn green. This is not the most efficient way of implementing semaphores. In many operating systems the COP could be made faster by substituting what is called a "test and set" instruction in the place of the exclusive file open. The point is that the COP can be tailored to the target operating system without changing its logical function, only the primitives need be changed.

**Implementation language**

The reader may have noted that no mention has yet been made about the implementation language. This is because the concepts discussed so far are language independent. The language that we are using is RATFOR (for RATional FORtran). The RATFOR preprocessor language enables one to write
FORTKAN code in a fashion that is much easier to understand. It provides features that are not available in FORTKAN, yet, the RATFOR is translated by the preprocessor (itself implemented in FORTKAN) into a portable form of standard FORTKAN. In sum, RATFOR makes writing programs easier and provides much of the advantages of languages with a modern control structure such as "PASCAL", "L" or "ADA".

We make no apologies for using venerable old FORTKAN. It is precisely because it is old and venerable that almost every computer supports FORTKAN and this is a great aid to portability.

Data base Organization

The clinical laboratory is heavily dependent on the storage and retrieval of data. Given the amount, variety and complex interrelationships of laboratory data, data base organization is a primary consideration for the design of laboratory software.

To provide a greater degree of flexibility and to make future modifications easier, we decided to make a firm distinction between the logical description of the data base and its physical implementation. The logical data base was organized in accord with the guidelines of the Conference on Data base System Languages (CODASYL)(7) and the VOS extended to include machine-independent functions necessary to implement a multiuser Data Manipulation Language.

A Data Manipulation Language is analogous to the VOS in that it provides a standard interface to information in the data base. Most of the Data Manipulation Language procedures need not be customized to the target operating system and are written in a portable manner. Just as the VOS serves as an interface between the program and a real machine, the Data Manipulation Language serves as an interface between the program and the data base. And, just as the VOS allows us to change machines, the Data Manipulation Language allows us to optimize the physical organization of the data base without having to change the application programs. Of course, if one changes the logical organization of the data base then at least some of the application programs would have to be changed.

Conclusion

To run complex laboratory programs on a variety of different computers one must consider program and system design at many levels. Much of good design deals with the management of complexity, and a useful technique for dealing with complexity is to take a little bit at a time and explain it, until we find that the small bits we have explained add up to a rather large chunk of knowledge. The system discussed here can be thought of as a hierarchy of systems. At the top are the application programs that directly support the laboratory's daily operations. These applications access data through a data base system with a special data manipulation language. The data base rests on a library of procedures that implement a VOS which, in turn, is based on a real computer.

Presently, VOS required to develop a multiuser data base is complete. Several laboratory modules are already running on the VOS and selected chemistry results are being transmitted by CRT terminal to the hospital's critical care units. We are in the process of developing a Data Manipulation Language based on CODASYL concepts that will serve as the foundation of a more powerful and sophisticated data base system.

Laboratory system software of the type we are developing represents a tremendous amount of effort. We hope that others will be able to benefit from our labors. We are interested in collaborating with other sophisticated software development groups to help portable data base and laboratory software achieve its full potential.

References

1- Achieving the Uptimum Information System for the Laboratory, J. Lloyd Johnson Associates, Northbrook, IL. (1975)
2- At Last There Is a Way to Take It with You, Science 207:746 (1980)
3- Software Standards in Chemistry, NRCC Proceedings No. 7 (1979)
4- Software Tools Users Group, 1259 El Camino Real, #242 Menlo Park, CA. 94025