CMU-ITC-83-028

WHITE PAPER: A High Level Virtual Machine"

Document Number 003-001/002.00

DRAFT

ITC Internal

June 22nd, 1983

James Gosling

Information Technology Center
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

ABSTRACT

A virtual machine specification is presented which attempts a graceful merging of the concepts from both Unix and Accent. Both Unix and Accent present a 'virtual machine' (a.k.a. Kernal interface) to the user. These virtual machines are specified at a very high level, having no mention of any particular physical machine. This is the key to the many successful ports of Unix to dissimilar machines. In this paper yet another virtual machine is presented which attempts to gracefully merge the concepts of both Unix and Accent. These virtual machines have some orthogonal concepts, (Accent's IPC and Unix's file system) and some overlapping concepts (e.g. memory management). Where the two overlap, there is a very striking similarity; and since the concepts are nearly the same, Unix's version will be preferred. The task of writing this paper really boils down to adding into Unix those aspects of Accent which are not already addressed by Unix.

PREFACE

One of the goals of this exercise is to define an interface which can be trivially brought into existence on many machines: PERQs running ICI, UNIX or Accent with a with a Unix interface package; SUNs; IBM PC's under PC DOS (the C86 compiler at the ITC already implements much of this specification); IBM PC's with a co-processor (XENIX on a SRI-TECH board); and on the final ITC Workstation.

The base of the virtual machine consists of all the facilities of the C programming language. One could substitute Pascal, but I believe one loses more than one gains. (This whole paper will generally skip justifications in an attempt to be brief.)

In reading through the various documents, it's hard to distinguish UNIX's **file descriptors** and Accent's **ports**. A Unix file descriptor is really a process-local name for an object similar to an Accent port. When you write a program under Accent and declare something to be a port, what you really get is a process-local name that is the same as a Unix file descriptor. Hence, for this VM, I will merge the two. Also, a Unix process identifier is essentially a global version of an Accent Kernal port with some differences in protection mechanisms.

I'd like to use a completely neutral word instead of port or file descriptor ('handle' for instance), but to avoid a jargon explosion I'll stick to 'port.' I'll even use 'port' for the Unix 'process identifier.'

All calls on the virtual machine will be specified as procedure calls. Whether they are really procedure calls or they are inline macros or messages being passed off to another process is irrelevant. The only thing that is relevant is how they appear in a program which uses them. It is expected that there will be a series of implementations through time. For example, the file system might start out being implemented as it is now in Unix as hardware traps to the kernel, but they may eventually become message communication to some remote process. One of the important things about this specification is that such migration of implementation techniques must be transparent -- it must be below the visible level of this interface.

CONTENTS

LIST OF ILLUSTRATIONS

1.0 READING AND WRITING

- **result = read (port, buffer, len)**
- **result = write (port, buffer, len)**

Read and **write** read and write **len** bytes to or from the **buffer** from or to a **port**. The **result** indicates either the actual amount transferred or an error. Ports will have protocols associated with them: **Stream** or **Datagram**. One can think of stream as being datagrams with the protocol envelope stripped off, leaving only the data. Essentially, all file and terminal IO will be done via stream reads and writes. Properties such as timeout intervals and blocking are associated with a port through seperate control operations -- this is done to simplify the common case.

1.1 NAMES

- **port = open (name, flags)**
- **err = link (name1, name2)**
- **err = unlink (name)**
- **err = access (name, IntendedMode)**
- **err = chmod (name, NewMode)**
- **err = chdir (name)**

Open creates a port and asociates it with a **named** object. Detailed options are specified with the **flags**. In the simple case, **name** is the name of a file in the usual Unix herarchical name space (this name space may be distributed across many machines, it may even contain devices and other processes, but it is a single uniform name space); and **flags** specifies reading or writing to a flat byte stream file. Devices are integrated into ths name space: they appear as nodes (not necessarily leaves!) in the tree. For example, '/dev/lpt' might be the lineprinter. Remote services are also integrated into this same name space. It should be the case that if a object is opened using some name it should be possible to ignore the differences between files, devices, and remote servers.

Some possible flags are:

- **FRDONLY**
- The port will only be used for reading FWRONLY
- The port will only be used for writing FRDWR
- The port will be used for reading and writing FAPPEND
- Append on each write FCREATE
- Create file if it doesn't exist FTRUNCATE
- Truncate size to 0 FDGRAM
- Do datagram IO (defaults to stream) FIPC
- The name being established is one to which others can connect -- they will be connected to this process.

Link and **unlink** have their Unix semantics (establish an alternate name and remove a name). Symbolic links seem essential, but I will leave that issue open.

Access probes the accessibility of some object in the name space.

Chmod changes the accessibility of an object if the user is so priviledged.

Chdir changes the 'viewpoint' of the program with respect to the global name space: names can be given where the base of the name space is the root of the file system; or relative to the 'current directory' which is just a shorthand way of talking about where the user is working now. It takes the name of the directory as an argument.

- **err = close (port)**
- **err = ioctl (port, opcode, argrecord)**
- **err = stat (port, &information)**
- **err = seek (port, base, offset)**
- **err = pipe (portpair)**
- **port = interpose (him, hisport)??????**
- **select = (readports, writeports, exceptports, timeout)**
- **(= PortsWithMessages)**
- **position = tell (port)**

Close disassociates a port from a process.

Ioctl performs some control operation on the thing at the other side of the port. This is often device specific, but is done in a uniform way so that if two types of objects respond similarly to the same types of operations, then the operation invocation is written similarly, so that replacing one object with another becomes easy. A classic case concerns files and the 'console.' It makes no sense to seek to the beginning of the console or to turn echoing off in a file, but if all that a program does is write streams of characters, then substituting one for the other makes sense and can be easily done.

Pipe creates two ports, one with read access and one with write access. The two ports are connected to each other such that data written to one can be read from the other. Writing to a pipe is done with exactly the same operations as are used for writing to a file or to a terminal -- a pipe can easily be substituted for either. One uses pipe rather than two opens in order to avoid creating a name, keeping the communication channel anonymous.

2.1 PROCESS CONTROL

- **port = fork (kind)**
- **exec (program, args...)**
- **port = wait (&status)**
- **exit (status)**

Fork creates a clone of the current process, both carry on executing with exactly the same context. The only difference is that in one (the parent) the returned **port** can be used to access the other, and in the other (the child) the **port** will be NULL. **Exec** overlays the current process with a new process image. Exec and fork are decoupled parts of what some systems think of as an integrated **initiate** operation. This decoupling is done for several reasons:

- Exec and fork are useful alone; exec as a **chain** operation, and **fork** in parallel processing servers.

- The time between forking and doing an exec in the child is used by the child to set up the final environment for the program to be invoked. Setting up this environment can be a complex process and forcing this complexity to always be present in the initiate operation would obscure the simple cases. This could, of course, be handled by macro wizardry. A more important point is that some of the setup would be difficult to express without this split. For example, if I wanted the standard output (console) of some program to be connected to the mailer. This way I could get the effect of having it's output mailed to someone. With an initiate operation, you either have to be able to express arbitrarily complex operations in the instantiation template, or the caller is going to have to perform some of them, disturbing his own environment (i.e., if the initiator had to create the mailer then the initiator would have to create a port to the mailer and pass it on, disturbing his environment).

- Exec and fork are simple. I'd like to keep them that way.

Wait is used to wait for the completion of a child process, the port returned identifies the child that completes.

2.2 EXCEPTION HANDLING

- **old = signal (SignalIdent, handler)**
- **err = kill (port, SignalIdent)**
- **err = pause (port)   signal (SIGSTP, ...)**
- **err = resume (port)   signal (SIGCONT, ...)**

There are two kinds of exceptions: in and out of line. An inline exception takes the form of a status code returned from some operation. This comes in two parts: all functions return a thing which may be the distinguished 'something has gone wrong' value. Detailed information is in the global variable **errno**. This can be wrapped in the following syntax:

```
OnErrorIn (port = open
("/dev/prt", FWRONLY))
        case ENONEXISTANT:
printf ("I can't find the printer.");
                          port =
StandardOutput;
                          break;
```

Out of line exceptions take the form of "spontaneous" procedure calls. To each of a set of events a procedure may be attached which is supposed to handle the occurance of the event. Such events are timers going off or addressing violations. The handler has the choice of either resuming the operation, or terminating it and doing a 'non-local goto' to an outer environment.

2.3 INTRA PROCESS OPERATIONS

- **space = malloc (size)**
- **err = free (space)**
- **alarm (interval)**
- **setuid, getgid, seteuid, setegid, setpriority (value ,port)**
- **value = getuid, getgid, ...(port)**
- **time = time (0)**
- **times (&DetailedTime)**

Malloc and **free** are used to allocate space. Lower lever notions like Unix's sbrk or Accent's ValidateMemory underly them.

2.4 OTHER STUFF

Most other 'utility' abstractions can be built on top of these:

abort	generate a fault
abs	integer absolute value
abspath	determine absolute pathname
atof,atoi,atol	convert ASCII to numbers
atoh,atoo	convert ASCII to hexadecimal or octal

atoo	convert ASCII to octal
atot	convert ASCII date string to time

convert

boolarg	parse boolean argument or ask user
cdate	convert date to ASCII
chrarg	parse character argument or ask user
ci	command interpreter
crypt, setkey,encrypt	DES encryption
ctime,localtime,gmtime,asctime,timezone	date and time to ASCII
curses	screen functions with "optimal" cursor motion

A.0 ???

???