A Guest Facility for Unicos*

Dennis M. Ritchie

Bell Laboratories
Murray Hill, NJ 07974

The COS system for Cray's X-MP series supports a guest facility, under which Unicos, their version of Unix® System V, can run as a subsystem in a fixed partition of the machine. The guest facility is unavailable when Unicos acts as a stand-alone system. This paper reports an experiment in which the author and David Slowinski added a limited version of the guest facility to Unicos, so that it can run itself as a subsystem.

COS's guest mechanism is analogous to that provided by other virtual-machine schemes, such as IBM's VM system, but is not as general. Cray supports it as a transition aid for sites converting to Unicos, and so far as I know, Unicos is the only guest system that has used the facility. As described below, Unicos is specially written to make it run as a guest, but the same binary system image runs both as a guest and in native, stand-alone mode. The main limitation in practice is that the machine is strictly partitioned exactly one CPU is dedicated to guest Unicos, and the memory and disks are split statically. There is also some overhead in doing I/O in the guest system, but it is relatively small. The real penalty is the partitioning of main memory, and to a lesser extent the dedication of the CPU.

Providing one accepts its limitations, the guest facility is convenient for its intended purpose. At AT&T Bell Laboratories, for example, we ran guest Unicos on an X-MP/24 for 18 months before finally converting to the native Unicos environment. This path was followed for several reasons: we started with an early, not especially stable version of Unicos to which we made many changes, and control and use of the machine was shared between a company-wide computation center that sought stability and a research organization that was willing to experiment. Using the guest facility, existing COS applications could be moved gradually to Unicos, which benefited the computer center and its customers.

Our Unicos system developers, however, valued the guest facility mainly because it simplified testing. It was possible to reboot and try new versions of Unicos merely by submitting remote COS jobs, an operation readily automated; the process took a minute or two. Although doing this required chasing away the Unicos users, they were mostly from our own organization and arrangements could be informal. In particular, rebooting Unicos did not disturb the paying customers of the computer center.

Converting to native Unicos operation significantly impeded system development work; we were back in a world more normal for expensive machines, in which development takes place at most a couple of evenings per week, announced well in advance. Therefore, we decided to create a mechanism by which Unicos could support itself as a guest system.

How Unicos Works

The Cray hardware does not make it possible to create a true virtual machine in the style of IBM's VM system. That is, one cannot take an arbitrary standalone operating system and create a cocoon around it so that it believes it is in sole control of the machine. However, certain aspects of the hardware, and the style in which it is used, do ease the job of running Unicos as a guest. The great advantage is that most I/O operations are done through a separate processor, the IOS, and the CPU conventionally communicates with the IOS by sending messages to it. Thus I/O operations, which are often the most problematical aspects of a VM system, are readily handled if the communication with the IOS can be intercepted and the IOS

Presented at USENIX Conference on Supercomputing, Pittsburgh, PA, Sept. 1988.

operations simulated. Another complicated aspect of a virtual machine system, paging and virtual memory, is avoided simply because the hardware doesn't support it even for native systems.

Unicos, in particular, was already written to run as a guest. In native mode, I/O is accomplished by sending request packets to the IOS through a CPU channel. Guest Unicos instead leaves these same request packets in a conventional memory buffer and signals the host system by executing a trapping instruction. The device drivers are written in a style that prepares request packets and passes them to a low-level common routine; this routine is one of a handful of places that cares whether Unicos is running as guest or in native mode.

When guest Unicos needs assistance from its host system, it places a request code in a register, and executes a ''normal exit'' instruction, just as system calls are specified by user programs in a conventional operating system for the X-MP, whether COS or Unicos. The only real difference is that Unicos's requests on its own behalf are much simpler than the facilities it provides to its own users.

Thus the key to making Unicos run as a guest under itself is to build an interpreter for these requests. There seemed to be two approaches to doing this. One way was to add the interpreter to the Unicos kernel. In this model, a normal-exit trap executed by a distinguished guest process in simulated monitor mode receives special treatment. I/O packets extracted from a communications area are sent (after minimal processing) directly to the IOS or elsewhere. This is precisely the way COS handles things, and it is appropriate when performance is important. It is difficult to develop and debug a system based on this approach, however, because all the work is done in the system kernel. Instead, we decided on a scheme that is less direct, and less efficient, but moves the work of the simulator into an ordinary user-mode program and requires few kernel changes.

Kernel Changes

The two related kernel changes required to create our guest facility are modeled on the 'exchange' mechanism of the hardware. An exchange package, in Cray's terminology, is a data structure containing the most important user-visible registers and a collection of internal machine state variables, including the base and limit registers used for memory mapping. When the machine changes state, because of an internally-generated trap or an exogenous interrupt, the active exchange package is swapped with a new exchange package from memory. The usual operating system maintains an exchange package for each running user process, and one per processor for kernel execution.

A new system call, named *exguest,* takes as operands a hardware exchange package and a pointer to storage for machine registers not contained in the exchange package. The new call swaps the contents of the argument exchange package and registers with the system's copy of exchange information and registers for the calling process, and then begins execution using the new exchange package. In effect, it forces an extended version of the hardware exchange sequence, and is used to hand control from the simulator program to the simulated guest operating system.

Another change, needed to arrange proper return from *exguest,* affects the handling of exchanges into the kernel. Previously, these had always been treated by Unicos as device interrupts, or exceptions (like floating-point errors), or as Unicos system calls (normal exit traps). With the change, the system consults a new flag to determine whether the running process is under control of a guest operating system (that is, has executed the *exguest* call). If the flag is set, traps incurred are not treated normally, but instead generate a return from the simulator program's *exguest* call; during the return, the exchange information passed to *exguest* is changed to reflect the state of the hardware at the time of the trap.

Of course, this description is simplified. The exchange package passed to *exguest* must be checked for legitimacy, because it will ultimately be turned over to the hardware. The most important complication is that the base and limit values in the exchange package must be adjusted properly. When finally presented to the hardware, they must represent absolute machine addresses, while the addresses generated by the simulator are relative to itself. *Exguest* must arrange that these addresses are relocated properly, and that the locations available to the simulated operating system and its users fit within the address space of the simulator.

The Simulator Program

The simulator program runs as an ordinary user process. It does not need special permissions except that, as described below, it may be useful to allow it to read disk devices that most users cannot access directly. It begins by allocating a large amount of memory to serve as an arena in which to run the simulated operating system, and loading the operating system code into this space. The main loop of the simulator repeatedly hands control to the guest system with *exguest;* on return, it examines the resulting exchange package to see what is being requested. Requests fall into two classes: those from the simulated operating system itself, which either map into appropriate I/O operations or ask to run a user program, and traps incurred by user-mode programs running inside the guest operating system. These are treated simply by forwarding control to the guest operating system.

More explicitly, the flow of control in the simulator program can be illustrated by the following pseudo-code:

```
forever {
        exguest(kernelXP)
        if kernelXP.requestreg == IOREQUEST
                doIO
        else if kernelXP.requestreg == RUNUSER
                exguest(kernelXP.paramreg)
        else
                panic
}
```

That is: exchange to the guest kernel; when it exchanges back, it is either requesting an I/O operation, or running a user program. In the former case, it suffices to do the needed I/O and continue. In the latter, the request carries a parameter specifying the desired user exchange package. The second *exguest* call in the example exchanges control to this simulated user process, and returns when the process makes a system call or incurs some other trap. The user exchange package, which resides in the simulated kernel's address space, is updated by the exchange, so it suffices merely to continue; the guest kernel will process the user's system call or trap.

The guest kernel handles I/O operations by bundling parameters into a queue in a conventional spot, and the *doIO* routine pulls requests out, does the transmission, and updates a reply queue. The requests are elementary; they specify a device, an address within the device, the transmission direction, a memory address, and a count. In order to make the simulator useful at all, the devices and their addresses have to be mapped somehow into a real file system on a disk or SSD. Therefore, the simulator contains a table indexed by device and sector address range that yields a Unicos device name and offset. A side table contains file descriptors for devices that have previously been used by the simulator. When the guest system refers to a sector on a particular device, the main table is consulted to map the request into an address within the corresponding Unicos device file; ordinary Unicos I/O calls are used to read or write the requested information, after opening the device if necessary.

By setting up the simulator tables appropriately, it is possible to control access to all the disk and SSD storage on the machine, and in particular for the simulator to see the devices on which user's files are kept. However, the simulator accesses file systems directly, through the raw disk device, and thus bypasses Unicos's cacheing strategy. The simulator should not be allowed to write a disk partition that is also mounted as an ordinary file system. One simple way to manage things is to create a root partition, containing a near-copy of the real root, for the simulator's exclusive use. A few files in the simulator's root file system are modified; for example, the script that mounts new file systems after startup mounts user file systems read-only, so that the simulator never attempts to change them.

Asynchronous events

The scheme described above for I/O works well for operations on the disk and SSD that complete quickly and can be made synchronous. Some kinds of I/O, and other things, occur asynchronously and must be handled by other means.

Console output, which is specified by a special kind of IOS packet, is straightforward, but input occurs asynchronously, and waiting for it must not block the simulation. Therefore, the simulator loop above must be supplemented by code that checks its standard input for characters every now and then. When it finds any, it generates an appropriate packet, and inserts it into the packet-input queue of the guest system. Similarly, Unicos expects to receive clock interrupts periodically. Therefore, each time around its loop, the simulator checks the real-time clock, and if it has advanced sufficiently, simulates a clock interrupt by setting the appropriate bit in the kernel exchange package. Fortunately, Unicos is written in such a way that it is insensitive to wide variations in the frequency of clock interrupts, so there is no need to be concerned about timing them precisely.

Limitations

The current simulator is a crude program, and is not at all suitable as a production tool, as the COS guest facility is. It is useful for system development.

There is an unavoidable cost in using our approach, in that I/O operations in the guest system turn into ordinary Unicos I/O calls in the simulator. This means they are much slower than they would be if the simulator were integrated into the operating system itself.

Moreover, the only I/O devices we support are the console (and only one console), the disks, and the SSD. In our installation, we use a low-speed CPU channel to interface to our Datakit® network, and this channel is not included in the simulation. The problem here is not the simulator itself, as much as is the interface to Datakit presented by the operating system. It appears to user programs as a set of devices each referring to a single virtual circuit on the network. However, the low-level interface to the hardware involves messages specifying the circuit number. The operating system does not provide a raw Datakit device in same way it does for disk devices. Equally important, Datakit circuit setup involves communication with a network controller, and the controller software does not provide a way of managing two independent sets of connections (host and guest) over the same physical channel.

A process running as a guest operating system is an ordinary Unicos process, and necessarily is large, because it must contain not only the operating system code but also enough memory for its own user programs. While it is active, it swaps against the rest of the system load. Moreover, the simulator (as written) has two unappetizing choices in managing its CPU usage. Either it runs all the time, faithfully simulating the idle loop, or it detects that the guest system is idle and waits a bit. Unfortunately, the obvious ways of waiting encourage the real Unicos system to swap the simulator out, or otherwise delay it. The former burns one whole CPU, the latter makes it quite slow. Of course, this situation resembles the one we accepted while we were running guest Unicos in production, but now the production programs use more memory, and the accounting statistics show the simulator CPU usage explicitly. Nevertheless, the simulator load is quite perceptable to the users, and they are not always assuaged by assuring them that a year or so ago they would not have been able to run their programs at all, or that the alterative is dedicated development time.

Summary

Our intent in creating a Unicos guest facility was to make it possible to do system development without disturbing production users, not to build a a mechanism that itself would permit running production codes. Even so, there remain many loose ends. For example, as discussed above, our networking hardware is not supported. Nevertheless, it is useful for its intended purpose, and it is much easier to check the integrity of new system code under the guest mechanism than to arrange for standalone development time. Also, the scale of the project was small. The user-level simulator program contains only about 500 lines of C code, and the operating system additions total less than 200. The effort seems well repaid.