CS 471 -- OPERATING SYSTEMS
Fall 1999

Notes on Virtual Machines

P. J. Denning
10/23/98
11/27/98 (revised)
11/1/99 (newly revised)
1/12/01 (more newly revised)

Virtual Machine Models for OS Levels

A virtual machine is a simulated computer. The term is used with at least four distinct but related meanings: a simulation of the hardware of a computer in a partition of the memory of that computer, simulation of an instruction set of one computer on another, a standard execution environment for any program, and an abstract machine manager for a class of objects. In these notes we will be concerned with the last two meanings. An executable program "wrapped" in a standard template will be called the Virtual Machine (VM) for the program. The OS module that creates, manages, and deletes program VMs is called the Virtual Machine Monitor (VMM) and is an abstract machine manager for the class of VMs.

In this course we have assumed hierarchies of abstract machines as a way to describe the levels of an operating system. The "instruction set" of an abstract machine consists of a the instruction set presented by lower levels, plus the names of new functions implemented at the current level, minus any lower-level instructions that are masked off by this level.

Each abstract machine is the manager of a set of objects of given type -- e.g., threads and semaphores are managed by the process manager. The abstract machine therefore creates and deletes objects of the types it manages and it performs operations on those objects. The operations are the functions implemented by the abstract machine. Suppose that the objects managed by a given level are of type X; we will call this abstract machine the "X manager". The X manager contains the functions

```
h = CREATE_X(init)
DELETE_X(h)
```

where h is a handle (a pointer whose format is discussed below) and "init" is the initial state of the object. The effect of CREATE_X is to create an instance of object X (with the given initial state) and give the user a pointer to the object. The object itself is managed within the storage system accessible to the X manager but not to the user. The user will be able to have operations performed on the object by calling the procedure name (part of the X manager) with handle h as a the parameter:

r = F(h,params)

where F is the desired function, h is the handle of the object, params are the parameters of the call, and r is the result.

In the hierarchy of levels, we start introducing handles in the process manager. (These are identifiers for processes and semaphores.) By the time we reach the files level, the user is responsible for acquiring, holding, and managing a variety of handles (threads, semaphores, secondary storage devices, virtual memories, interprocess communication objects, devices, and files). The directory level is added to assist the beleaguered user to manage handles. Individual directories associate user-chosen symbolic names with the system-chosen handles. Handles of related objects can be grouped in a single directory. Once the directory manager is in place, users can now refer to objects with symbolic path-names rather than directly by handles. This increases reliability in two ways: (1) users can organize handles by names that make sense to them, and (2) the system can safeguard the formats of handles because users do not have direct access to them.

At the user interface (shell) level, all handles have disappeared. The operating system gets them as needed from directories and passes them around as parameters internally.

The level implementing *User Virtual Machines* provides a method of encapsulating arbitrary programs inside a standard operating environment. A small set of operations is then sufficient to create, delete, start, stop, and interlink these VMs and information objects they use (files, devices, pipes). VMs can be used recursively -- a given machine can create a subordinate machine encapsulating a copy of the same program, with its own parameters. The basic operations of this level are:

vmh=CREATE_VMACH(*init*)	create a new VM with the given *init* state and its internal thread suspended; return a virtual machine handle
DELETE_VMACH(vmh)	delete the VM specified by the handle
COMPUTE	a parent VM requests that all its children VMs be started; parent then stops and waits until all children have exited
EXIT	a VM signals that it is done

There thus is an abstract machine, the VMM, whose instruction set is the four operations above. This abstract machine creates individual, program-executing VMs. Do not confuse the VMM with the program VMs.

Handles

A handle is a fixed-size bit pattern of the following form:

handle = (type, access, id)

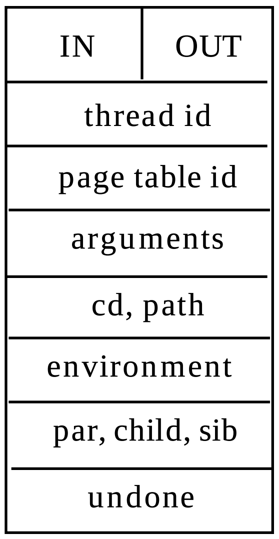
where "type" designates the particular X to which the handle points, "access" selects which of the functions of the X manager may be applied to this object, and "id" is a unique name for the object. The id should have enough bits that the total number of possible id's exceeds the number of distinct objects (of type X) that will ever be created. (A re-use of an id can cause a conflict between someone holding an old handle with the same id as a newly created object.) Typically, an id is constructed by joining a machine name with a timestamp.

Handles were first invented in 1965 by Jack Dennis and Earl Van Horn, who called them "capabilities". Their scientific paper inspired a line of computers called "capability machines" which dealt with objects pointed to by capabilities, and provided extensive hardware support to prevent capabilities from being tampered with. Capability machines were the first object-oriented computers; the first of them was built in 1968 and by 1972 there were commercial versions. They were found to be extraordinarily reliable because most common bugs triggered capability violations and stopped the machine before errors could propagate. Because capability machines also had complex hardware and operating systems, the RISC machine usurped them.

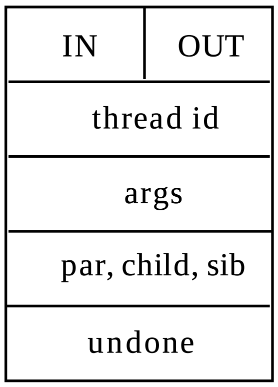
But the capability architecture didn't disappear. It was renamed "object oriented environments". Capabilities were renamed "handles". Type-checking by compiler and subroutine call parameter check replaced hardware as a means to prevent handles from being presented to the wrong managers and from being altered. Since most users did not try to get inside object-oriented environments, the incidence of users tampering with handles is quite low. Most object-oriented environments are as reliable as capability machines were.

Creating User Virtual Machines

A virtual machine in an operating system is a program in execution in a standard execution environment. The simulated computer contains a thread executing instructions and accessing data in a virtual memory; it has access to the file system through a current directory and path; it has access to environment variables; it has an argument list, a list of values provided when it was called; and it has access to a standard input object and a standard output object. The picture below depicts these elements of a virtual machine. The reduced form at the right is a simpler version sufficient for the base system of our project.



GENERAL CASE



OUR CASE

The field **IN** is intended to contain an the handle of an object (pipe, device, file) open for reading. The field **OUT** is intended to contain the handle of an object open for writing. The **thread id** is the process number from OS process manager. The **page table id** is the page table pointer from OS virtual memory. The **arguments** field is a list of 0 or more values provided at the time the procedure encapsulated in the virtual machine was called. The **cd** is the handle of the current directory and **path** is the directory search path. The **environment** contains any other variables that the system uses -- e.g., the string naming the owing user, the default printer, the default text editor. The **par, chil, sib** field contains pointers, respectively, to parent, child, and sibling virtual machines; these pointers link all the children machines to their creating parent. The variable **undone** counts the number of children that have yet to finish their computations; it is used to synchronize the parent with the complete computation performed by the children. A programmer can create one of these virtual machines with a call specifying all the values:

```
vmh = CREATE_VMACH(IN=“object open for reading”,
                   OUT=“object open for writing”,
                   thread_id=CR_THREAD(“procedure name”),
                   page_table_id=CR_VIRT_MEM(“cache file”),
                   arguments=“ argument list for procedure”
                   cd=“current directory of creator”
                   path=“path of creator”
                   environment=“environment of creator”,
                   par=“handle of parent virtual machine”,
                   chil=“handle of first child virtual machine”,
                   sib=“handle of sibling virtual machine”,
                   undone=0)
```

The effect is to create a VM set up to run a particular procedure. Its thread is initially in a suspended state, to be started later by the COMPUTE command.

Computing with Virtual Machines

When a VM is ready to start the computation of its children, it issues the COMPUTE command. The COMPUTE command does not return to its caller until all the caller’s children have completed their computations.

The COMPUTE command is an operation of the VMM. COMPUTE places each child’s thread in the ready state; and it sets undone=N, the number of children. When a thread terminates, it calls the EXIT command, another operation of the VMM. EXIT subtracts 1 from the undone count of the parent and, if the undone count becomes 0, it signals the parent’s COMPUTE command to return.

This COMPUTE command is actually a simple version of job control. More sophisticated VMMs give more control. For example, a VM can have many concurrent subcomputations linked to it; as the parent, it can start and stop any one of them at any time. The parent VM can select one of the subcomputations to operate in the “foreground” -- where its input and output are directly connected to the keyboard and display. A “background” subcomputation is disconnected from the keyboard and display.

Because pipes, devices, and files follow the same data stream model, the programmer of the procedure running in a virtual machine can use the generic read and write commands (OS Level for I/O streams). For example, to read from the standard input, the programmer would write

```
a = READ(IN,n)
```

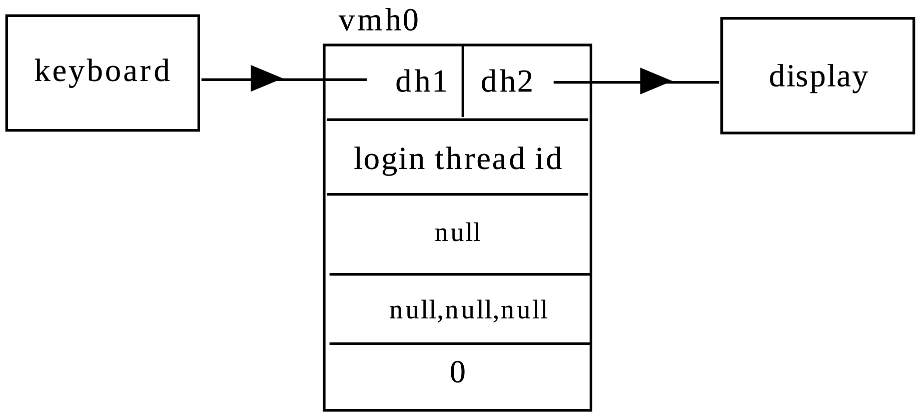
where a is an address in the virtual memory, IN is the input port of the virtual machine, and n is the number of bytes to read. Similarly, to write into the standard output, the programmer would write

```
WRITE(OUT,a,n)
```

Getting Started

When the system is initialized, it attaches a virtual machine instantiated with the login program to every workstation. This is depicted below, with vmh0 denoting the handle of one of these virtual machines. The **keyboard** is a device that inputs typed data. The **display** is a device that outputs bitmapped data. Both keyboard and display are part of a user’s workstation. (Most keyboard drivers echo typed characters to the same workstation’s display.) The handle **dh1** is the open-device handle of the keyboard; its access code indicates read-only. The handle **dh2** is the open-device handle of the display; its access code indicates write-only. The system initializer obtained these handles by making the calls:

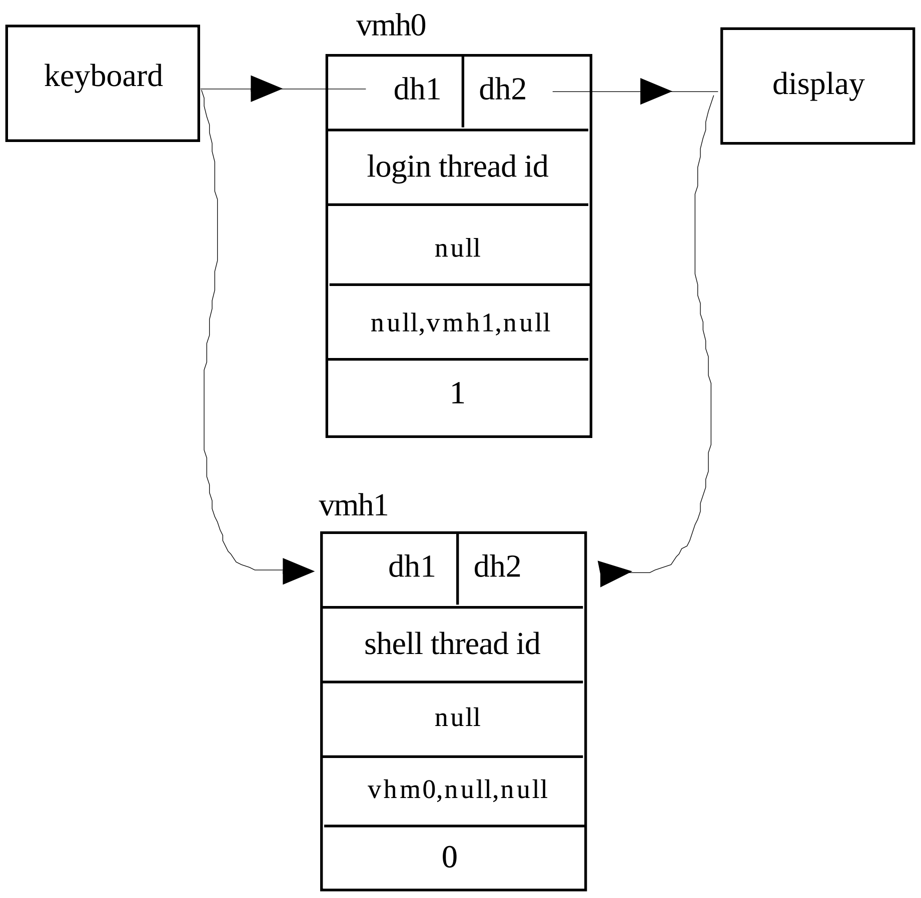
```
dh1 = OPEN_DEVICE(keyboard, read)
dh2 = OPEN_DEVICE(display, write)
```



When a user logs in, the login VM creates a VM containing the shell:

```
vmh1 = CREATE_VMACH(IN,
                    OUT,
                    CR_THR(”shell”),
                    args=null,
                    par=vmh0,
                    chil=null,
                    sib=null,
                    undone=0)
```

and it issues the COMPUTE command. The result is the VM structure depicted below. Even though vhm0 and vhm1 both have access to the keyboard and display, only the subcomputation (vmh1) is active and so the parent and child cannot conflict at the keyboard-display of the workstation.



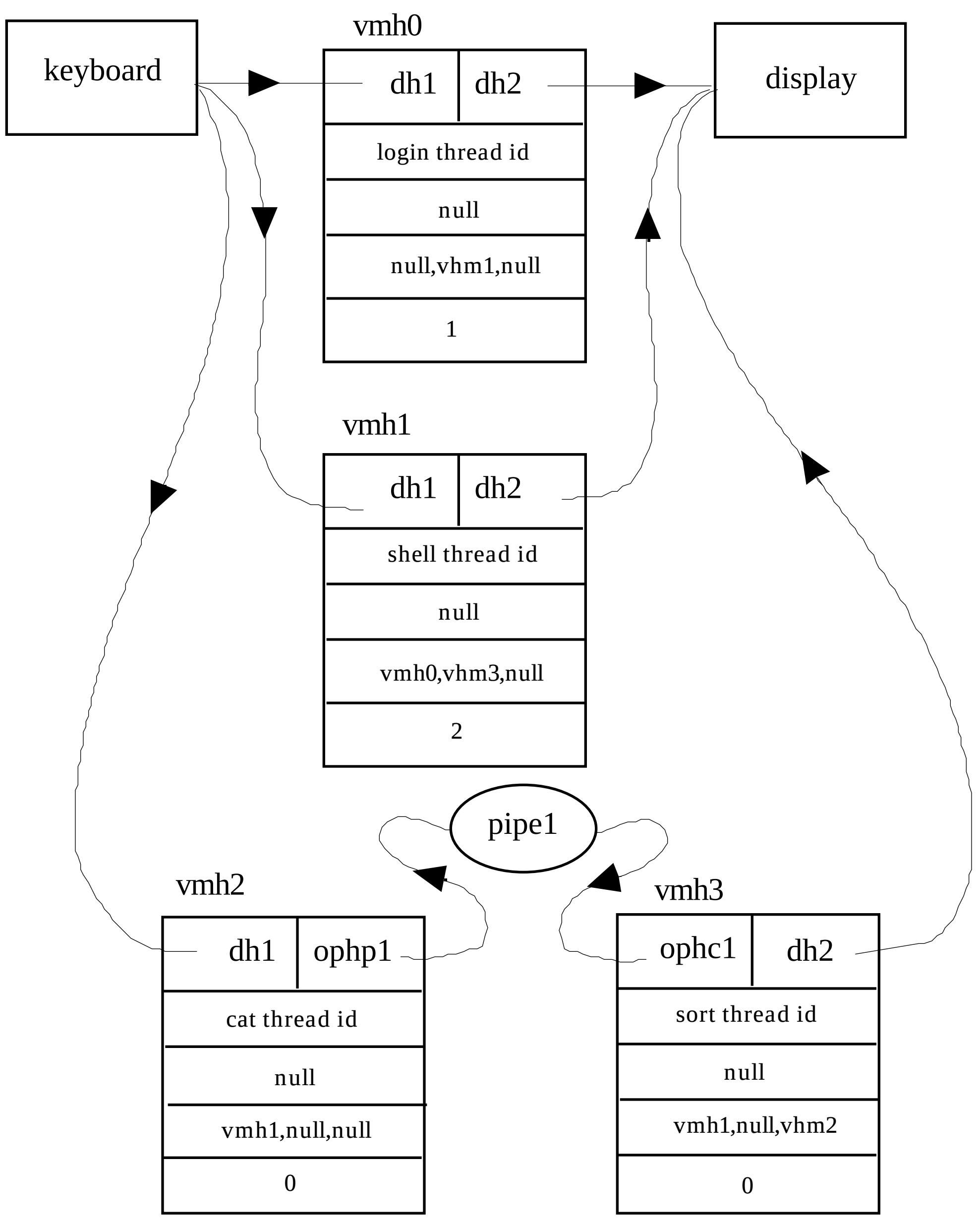
Now, suppose that the user types

```
cat | sort
```

to the shell. (This command string is picked up the by the shell since the login UVM is now blocked awaiting termination of the shell.) The shell parses this into three tokens representing the pipeline of three objects (“cat”, “pipe1”, “sort”). It then creates the objects and links them together in the sequence specified by the command line:

```
pipe1 = CREATE_PIPE()
ophp1 = OPEN(pipe1,producer)
ophc1 = OPEN(pipe1,consumer)
vmh2 = CREATE_VMACH(IN,ophp1,”cat”,args=null, par=vmh1,child=null,sib=null)
vmh3 = CREATE_VMACH(ophc1,OUT,”sort”,args=null,par=vhm1,child=null,sib=vmh2)
```

After the shell executes the COMPUTE command, the VM configuration will be as shown below.



Now, because both login and shell are blocked awaiting completion of their COMPUTE commands, inputs typed on the keyboard will be read by the cat machine (vhm2) and passed down through the pipe to the sort machine, which will in turn pass the sorted list to the display.

A more sophisticated shell will accept notations that specify files or alternative devices as inputs or outputs of a pipeline, and notations that specify arguments for particular commands in a pipeline. Examples from the Unix shell:

```
cat < file1 | sort
cat < file1 | sort > file2
cat file1 file2 file3 | sort > file4
cat -v < file1 | sort -a > file2
```