COMP 242 Class Notes
Section 10: Organization of Operating Systems

1. ORGANIZATION OF OPERATING SYSTEMS

We have studied in detail the organization of Xinu. Naturally, this organization is far from the only one used in current operating systems. In many respects it is the simplest of the ones adopted. In this discussion we first look at the main characteristics of the Xinu organization and then study variations of it that are supported in other operating systems.

1.1 Xinu

All operating system services are provided by the kernel.

The kernel is a bunch of library routines linked to user programs.

The kernel routines are "encapsulated" in files.

The user processes and the kernel share a common physical memory.

The set of procedures that can be executed is determined when the OS is started.

An advantage of this organization is that it is simple. Moreover, information to be communicated among user processes and between a user process and the kernel does not have to be copied.

Some disadvantages are:

The kernel routines are encapsulated in files instead of module-like structures.

Address spaces are not protected.

The total address space available is limited to the size of physical memory.

Procedures cannot be dynamically loaded.

1.2 Pilot

Pilot is an operating system targeted towards personal computers. It is different from Xinu in the following respects:

The kernel routines are encapsulated within Mesa modules and monitors.

The user processes and the kernel share a common virtual memory instead of physical memory.

Programs can be dynamically loaded and linked.

Thus an important disadvantage of this organization is that address spaces are not protected by the OS. (This problem is alleviated to some extent by strong type checking in Mesa) This disadvantage has to be weighed against the advantage of efficient sharing of information.

1.3 Unix

Unix is different from Xinu in the following main respects:

There is a distinction between kernel and rest of the operating system since some of the OS functions are provided through user processes such as the *swapper*.

Processes execute in separate virtual address spaces.

Kernel code executes in a special privileged mode and a separate virtual address space.

Dynamic loading of programs is supported (but not dynamic linking since address spaces are not shared)

1.4 Server-Based Operating Systems

While Unix does support the distinction between kernel and rest of the operating system, the major functions of the operating system such as file management and terminal I/O are implemented in the kernel. Several operating systems are more serious about providing OS functions through utility processes. They provide servers for a large number of operating system services such as memory management, file management, I/O etc. 242-Xinu is sort of an example of a server-based system since the terminal service is provided by user-level processes. It is not a typical example because the terminal servers can access the kernel address space and can also be considered kernel processes (discussed below).

Providing OS services through servers supports the separation of **policy** from **mechanism**. The kernel provides a minimal set of services that are *mechanisms* used by servers to enforce *policy*. This separation is important in systems that expect to evolve since policy changes do not involve modification of the kernel. It is also useful in building distributed operating systems, since servers can be accessed from remote machines (if the IPC is distributed).

One potential disadvantage of providing an OS function through a server is slowness of response. Invocation of a service through a server includes the following steps:

A kernel call made by the client to send a request to the server.

A kernel call made by the server to receive the message.

Possible copying of data from client's address space to server's address space.

A kernel call made by the server to send results to the client.

A kernel call made by the client to receive the results if the original request was asynchronous.

Possible copying of data from server's address space to client's address space.

If the service is invoked through a kernel then only one kernel call is made and no copying of data between client and server address spaces is needed. (Data may be copied to and from the kernel's address space in either case.)

An important question in server-based operating systems is: how much of the OS functionality can be provided through servers? Clearly some functionality is essential in the kernel. The kernel needs to be able to run processes, in particular the servers. It also needs to support interprocess communication primitives that allow clients to make requests and servers to deliver results. Often a kernel (or at least a part of it) is guaranteed to be always resident in memory. Moreover, kernel code executed more efficiently than user-level server code. Therefore, code that reacts to interrupts is also included in the kernel.

Finally, an operating system such as Unix allows the kernel to run in a special privileged mode that can change memory mapping registers, use physical addresses, etc. Therefore OS functions that require this mode are also provided by the kernel.

In a server-based operating system how does a process get in touch with a server? Some services (such as name-services) could be associated with well known port numbers as in Unix. A process could be born with references to communication ports of other services (such as standard input and output). These references may be inherited from its parent or specified by the user through command arguments or an initialization file. Still others (such as file services) it could get by contacting a

name server, which allows a server to register a service name and a port reference, and answers queries that ask for a port reference corresponding to a service name.

1.5 Kernel Processes

So far we have assumed that the kernel is a bunch of procedures encapsulated within files or modules and monitors. We now discuss a different approach that structures the kernel as a bunch of cooperating **kernel processes**. Each of these processes could be responsible for a logical subset of the services offered by the kernel. Thus one process could be responsible for scheduling, another for managing disk I/O, a third for processing information received from other computers and so on.

There are some advantages of supporting kernel processes:

The kernel is well-structured since kernel services are encapsulated within the processes that provide them. This encapsulation is similar to the one provided by modules and monitors. (Recall the Lauer and Needham paper on the duality of operating systems).

Kernel code does not have to run as part of user processes. In a system that does not support kernel processes, all kernel code runs as part of user processes. For example in Xinu, when a service call is made, the kernel code to service the call runs as part of the process that made the call. In a system that supports kernel processes, the service call would result in a message to a kernel process, which would then execute the appropriate code. In some ways the latter approach is more 'elegant'.

In a multiprocessor system kernel processes could execute concurrently.

The idea of kernel processes may seem contradictory to our discussion of the minimum functionality provided by a kernel. If a kernel is responsible for supporting processes and interprocess communication, how can it use these facilities itself? Here are two solutions:

Kernel processes and communication among them may be supported by the programming language. Charlotte, a distributed operating system developed at Madison, uses this approach.

A portion of the kernel, which we shall call the **nugget**, could support kernel processes and communication among these processes, and higher layers could be implemented as communicating kernel processes. This approach is implemented in Sun Unix.

Should kernel processes be any different from ordinary user processes? It is important to execute kernel code efficiently, therefore these processes should be lightweight processes.

2. DISTRIBUTED OPERATING SYSTEMS

So far we have concentrated on the organization of uniprocessor systems. As we saw earlier, a (symmetric) mutiprocessor OS is structured like a uniprocessor system - though we do need changes to scheduling and other components of the system. We now consider the organization systems running on multiple computers with local memories and connected by a network.

2.1 Long-Haul Networks

As we saw before, these networks are collections of widely scattered computers connected by a common communication network. Communication in such systems is relatively slow (9.6 to 56 Kbps) and unreliable and typically through telephone lines, microwave links, and satellite channels.

Long-haul networks provide several services to their users:

The ability to mail information from one site to another.

The ability to post news on **bulletin boards**, which may be read by any user on the system.

The ability to find information about users on different computers through services such as *finger*.

The ability to copy files from one system to another.

The ability to logon at a remote site. The local computer directs user input to the remote site and displays output received from the remote computer.

WWW access.

The notion of a long-haul network requires the ability to communicate among remote processes. It also requires schemes such as replication/caching for overcoming the lack of speed and reliability of the network and also the absence of a global clock. It does not, however, require a special organization and any operating system that communicate across the network can offer these services.

2.2 Local-Area Networks

A **local-area** network consists of independent computers confined to small geographical area. Since this area is small, the communication network can be fast (3-100 Mbps) and reliable.

A local-area network offers, in addition to the services offered by long-haul networks, *sharing* of expensive devices such as printers and large secondary store. This sharing is important when the nodes are **workstations** – powerful personal computers that come with high-quality displays and small or no local disks. Typically, a network of workstations has one or more special machines connected to large disks that run *disk server* or *file server* processes. These processes allow processes on other machines to use the disks for storing files. Thus LANs are built on top of server-based operating systems.

A disk server provides each client with a small virtual disk since the only service it normally provides is sharable secondary storage. A disk server provides each client with a small virtual disk and primitives to read and write blocks on the virtual disk. The client can use these primitives to build its own file system on top of the virtual disk. A file server, on the other hand, allows access at the file-level, and provides primitives to open, read, and write files. (Often a workstation that provides sharable secondary store is called a **disk server machine** or a **file server machine**, since the only service it normally provides is sharable secondary storage.)

Each approach to shared secondary storage has its advantages. A disk server is simple and supports customized file systems. A file server gives higher-level primitives and allows sharing of files among its different clients.

A network of computers will often have several machines connected to sharable secondary storage. Therefore it is useful if files could be shared among these machines also. (Note this feature is different from providing special programs that can transfer files from one computer to another.) There are two complementary schemes for naming files in a system that supports such sharing. The simpler scheme incorporates the machine name in the file. It is not convenient to move files around in this system. The alternative scheme makes the file location *transparent* to the user. Under this approach the operating system maintains tables and uses algorithms which allow it to find a file anywhere in the network.

A popular example of a transparent file system is the Network File System (NFS) developed by Sun. It allows a directory belonging to a remote disk to be *mounted* on a local directory, much in the way the root directory of a logical disk (also called file system) can be mounted on some other directory. For instance the command:

```
mount A:/a/b l
```

mounts the directory '/a/b' on machine 'A' to the local directory 'l'. Now the file '/a/b/f' on 'A' can be accessed as 'l/f'.

2.3 Multicomputers

Local-area (and long-haul) networks run independent, possibly different, operating systems that offer limited services for sharing. An alternative approach is to present a single operating system that manages all the computers. The operating system can run different processes of an application on different computers. In addition, if a particular site becomes overloaded, the operating system can *migrate* processes running on the computer to other sites. There are several potential benefits of migration: load balancing, move process to data and other resources being accessed frequently, fault tolerance, execution on specialized hardware, etc. Migration requires a mechanism to *gather load information*, a distributed policy that decides *when a process should be moved*, and a mechanism to affect the *transfer*. Migration has been demonstrated in systems such as Locus, Demos/MP, and Charlotte.

Migration poses several problems. A system cannot simply copy the state associated with the migrating process to the destination process since some of this state may be host relative. (e.g. process id's, pending Unix signals and messages). A process may therefore be migratable only under certain conditions. Some systems do not guarantee that all calls will behave the same when the process is migrated. Even if there is no host relative state, the system must ensure that all messages directed to the migrated process reach the new destination. In some systems such as Charlotte, there are explicit bound ports established between communicating processes. When a process with existing links moves, the system can update the information at the other end of the bound port. The situation is compounded by the fact that two processes at opposite ends of bound ports might simultaneously move. In a system with input ports, a forwarding message can be left at the original host, which is returned to the sender. Because of these problems, some systems only migrate new processes (in the scheduler Q) and not those that have already executed and built-up site specific state. One of the potential drawbacks in process migration is migration cost: the time taken to migrate a process might be less than the time required to complete the process. A related problem is latency: during migration, the process does not respond to the user. Two solutions have been proposed to address this problem: One is to do precopying - the process continues to run at the original computer until is completely copied to the remote computer. Another one, adopted by Accent, is lazy copying, the complete memory image of a process is not copied when it is migrated. Instead, the state is copied on reference, and studies show this works well because programs tend to use only a small part of their state. This approach also solves the migration cost problem. Processes might be moved only during certain states, e.g suspended, to ensure that their interactive response time does not suffer. Migration might also increase the cost of accessing resources that were on the original machine. These resources can be migrated with the process.

The benefits of migration, thus, have to be weighed against the overheads involved.

How is the operating system on a multicomputer organized? Typically, each machine has a copy of the kernel, which provides the minimum functionality that includes communication between remote processes. Most of the operating system tasks are handled by servers which reside on different machines. Often a service is provided by a team of distributed servers instead of a centralized server, for several reasons:

Each computer connected to devices needs servers on that machine to manage the devices, as in 242-Xinu, which creates terminal servers on each machine connected to a tty device.

Requests to local servers may be satisfied faster that remote servers.

A centralized server may become a bottleneck.

A service may be lost if the machine running the centralized server goes down.